Module8

Classes and Objects

Table of Contents

Up to this point, you have been writing programs that did not use any of C++'s object-oriented capabilities. Thus, the programs in the preceding modules reflected structured programming, not object-oriented programming. To write object-oriented programs, you will need to use classes. The class is C++'s basic unit of encapsulation. Classes are used to create objects. Classes and objects are so fundamental to C++ that much of the remainder of this book is devoted to them in one way or another.

Class Fundamentals

Let's begin by reviewing the terms class and object. A class is a template that defines the form of an object. A class specifies both code and data. C++ uses a class specification to construct objects. Objects are instances of a class. Thus, a class is essentially a set of plans that specify how to build an object. It is important to be clear on one issue: a class is a logical abstraction. It is not until an object of that class has been created that a physical representation of that class exists in memory.

When you define a class, you declare the data that it contains and the code that operates on that data. While very simple classes might contain only code or only data, most real-world classes contain both.

Data is contained in instance variables defined by the class, and code is contained in functions. The code and data that constitute a class are called members of the class.

CRITICAL SKILL 8.1: The General Form of a Class

A class is created by use of the keyword class. The general form of a simple class declaration is class class-name

```
{
        private data and functions
        public:
        public data and functions
} object-list;
```

Here class-name specifies the name of the class. This name becomes a new type name that can be used to create objects of the class. You can also create objects of the class by specifying them immediately after the class declaration in object-list, but this is optional. Once a class has been declared, objects can be created where needed.

A class can contain private as well as public members. By default, all items defined in a class are private. This means that they can be accessed only by other members of their class, and not by any other part of your program. This is one way encapsulation is achieved—you can tightly control access to certain items of data by keeping them private.

To make parts of a class public (that is, accessible to other parts of your program), you must declare them after the public keyword. All variables or functions defined after the public specifier are accessible by other parts of your program. Typically, your program will access the private members of a class through its public functions. Notice that the public keyword is followed by a colon.

Although there is no syntactic rule that enforces it, a well-designed class should define one and only one logical entity. For example, a class that stores names and telephone numbers will not normally also store information about the stock market, average rainfall, sunspot cycles, or other unrelated information. The point here is that a well-designed class groups logically connected information. Putting unrelated information into the same class will quickly destructure your code!

Let's review: In C++, a class creates a new data type that can be used to create objects.

Specifically, a class creates a logical framework that defines a relationship between its members. When you declare a variable of a class, you are creating an object. An object has physical existence and is a specific instance of a class. That is, an object occupies memory space, but a type definition does not.

CRITICAL SKILL 8.2: Defining a Class and Creating Objects

To illustrate classes, we will be evolving a class that encapsulates information about vehicles, such as cars, vans, and trucks. This class is called Vehicle, and it will store three items of information about a vehicle: the number of passengers that it can carry, its fuel capacity, and its average fuel consumption (in miles per gallon).

The first version of Vehicle is shown here. It defines three instance variables: passengers, fuelcap, and mpg. Notice that Vehicle does not contain any functions. Thus, it is currently a data-only class. (Subsequent sections will add functions to it.)

```
class Vehicle {
public:
  int passengers; // number of passengers
  int fuelcap;    // fuel capacity in gallons
  int mpg;        // fuel consumption in miles per gallon
};
```

The instance variables defined by Vehicle illustrate the way that instance variables are declared in general. The general form for declaring an instance variable is shown here:

type var-name;

Here, type specifies the type of variable, and var-name is the variable's name. Thus, you declare an instance variable in the same way that you declare other variables. For Vehicle, the variables are preceded by the public access specifier. As explained, this allows them to be accessed by code outside of Vehicle.

A class definition creates a new data type. In this case, the new data type is called Vehicle. You will use this name to declare objects of type Vehicle. Remember that a class declaration is only a type description; it does not create an actual object. Thus, the preceding code does not cause any objects of type Vehicle to come into existence.

To actually create a Vehicle object, simply use a declaration statement, such as the following:

```
Vehicle minivan; // create a Vehicle object called minivan
```

After this statement executes, minivan will be an instance of Vehicle. Thus, it will have "physical" reality.

Each time you create an instance of a class, you are creating an object that contains its own copy of each instance variable defined by the class. Thus, every Vehicle object will contain its own copies of the instance variables passengers, fuelcap, and mpg. To access these variables, you will use the dot (.) operator. The dot operator links the name of an object with the name of a member. The general form of the dot operator is shown here:

object.member

Thus, the object is specified on the left, and the member is put on the right. For example, to assign the fuelcap variable of minivan the value 16, use the following statement:

```
minivan.fuelcap = 16;
```

In general, you can use the dot operator to access instance variables and call functions. Here is a complete program that uses the Vehicle class:

```
// A program that uses the Vehicle class.

#include <iostream>
using namespace std;

// Declare the Vehicle class.
class Vehicle {
public:
  int passengers; // number of passengers
  int fuelcap;    // fuel capacity in gallons
  int mpg;        // fuel consumption in miles per gallon
};

int main() {
  Vehicle minivan; // create a Vehicle object
  int range;
  // Assign values to fields in minivan.
  minivan.passengers = 7;
  minivan.fuelcap = 16;
  minivan.mpg = 21;

  // Compute the range assuming a full tank of gas.
  range = minivan.fuelcap * minivan.mpg;

  cout << "Minivan can carry " <<  minivan.passengers <<
          " with a range of " << range << "\n";

  return 0;
}
```

Declare the **Vehicle** class.

Create an instance of **Vehicle** called **minivan**.

Notice the use of the dot operator to access a member.

Let's look closely at this program. The main( ) function creates an instance of Vehicle called minivan. Then the code within main( ) accesses the instance variables associated with minivan, assigning them values and then using those values. The code inside main( ) can access the members of Vehicle because they are declared public. If they had not been specified as public, their access would have been limited to the Vehicle class, and main( ) would not have been able to use them.

When you run the program, you will see the following output:

```
Minivan can carry 7 with a range of 336
```

Before moving on, let's review a fundamental principle: each object has its own copies of the instance variables defined by its class. Thus, the contents of the variables in one object can differ from the contents of the variables in another. There is no connection between the two objects except for the fact that they are both objects of the same type. For example, if you have two Vehicle objects, each has its own copy of passengers, fuelcap, and mpg, and the contents of these can differ between the two objects. The following program demonstrates this fact:

```
// This program creates two Vehicle objects.

#include <iostream>
using namespace std;

// Declare the Vehicle class.
class Vehicle {
public:
  int passengers; // number of passengers
  int fuelcap;    // fuel capacity in gallons
  int mpg;        // fuel consumption in miles per gallon
};

int main() {
  Vehicle minivan; // create a Vehicle object
  Vehicle sportscar; // create another object

  int range1, range2;

  // Assign values to fields in minivan.
  minivan.passengers = 7;
  minivan.fuelcap = 16;
  minivan.mpg = 21;

  // Assign values to fields in sportscar.
  sportscar.passengers = 2;
  sportscar.fuelcap = 14;
  sportscar.mpg = 12;

  // Compute the ranges assuming a full tank of gas.
  range1 = minivan.fuelcap * minivan.mpg;
  range2 = sportscar.fuelcap * sportscar.mpg;

  cout << "Minivan can carry " <<  minivan.passengers <<
          " with a range of " << range1 << "\n";

  cout << "Sportscar can carry " <<  sportscar.passengers <<
          " with a range of " << range2 << "\n";

  return 0;
}
```

minivan and **sportscar** each have their own copies of **Vehicle**'s instance variables.

The output produced by this program is shown here:

```
Minivan can carry 7 with a range of 336

Sportscar can carry 2 with a range of 168
```

As you can see, minivan's data is completely separate from the data contained in sportscar. Figure 8-1 depicts this situation.

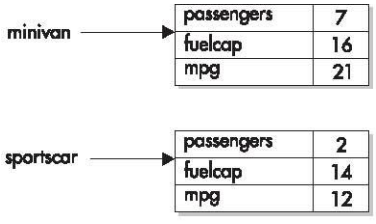

Figure 8-1 One object's instance variables are separate from another's.

1. A class can contain what two things?
2. What operator is used to access the members of a class through an object?
3. Each object has its own copies of the class' _____.

CRITICAL SKILL 8.3: Adding Member Functions to a Class

So far, Vehicle contains only data, but no functions. Although data-only classes are perfectly valid, most classes will have function members. In general, member functions manipulate the data defined by the class and, in many cases, provide access to that data. Typically, other parts of your program will interact with a class through its functions.

To illustrate member functions, we will add one to the Vehicle class. Recall that main( ) in the preceding examples computed the range of a vehicle by multiplying its fuel consumption rate by its fuel capacity. While technically correct, this is not the best way to handle this computation. The calculation of a vehicle's range is something that is best handled by the

Vehicle class itself. The reason for this conclusion is easy to understand: The range of a vehicle is dependent upon the capacity of the fuel tank and the rate of fuel consumption, and both of these quantities are encapsulated by Vehicle. By adding a function to Vehicle that computes the range, you are enhancing its object-oriented structure.

To add a function to Vehicle, specify its prototype within Vehicle's declaration. For example, the following version of Vehicle specifies a member function called range( ), which returns the range of the vehicle:

```
// Declare the Vehicle class.
class Vehicle {
public:
  int passengers; // number of passengers
  int fuelcap;    // fuel capacity in gallons
  int mpg;        // fuel consumption in miles per gallon

  int range();   // compute and return the range ◄—— Declare the range( ) member function.
};
```

Because a member function, such as range( ), is prototyped within the class definition, it need not be prototyped elsewhere.

To implement a member function, you must tell the compiler to which class the function belongs by qualifying the function's name with its class name. For example, here is one way to code the range( ) function:

// Implement the range member function. int Vehicle::range() {

return mpg * fuelcap; }

Notice the :: that separates the class name Vehicle from the function name range( ). The :: is called the scope resolution operator. It links a class name with a member name in order to tell the compiler what class the member belongs to. In this case, it links range( ) to the Vehicle class. In other words, :: states that this range( ) is in Vehicle's scope. Several different classes can use the same function names. The compiler knows which function belongs to which class because of the scope resolution operator and the class name.

The body of range( ) consists solely of this line:

return mpg * fuelcap;

This statement returns the range of the vehicle by multiplying fuelcap by mpg. Since each object of type Vehicle has its own copy of fuelcap and mpg, when range( ) is called, the range computation uses the calling object's copies of those variables.

Inside range( ) the instance variables fuelcap and mpg are referred to directly, without preceding them with an object name or the dot operator. When a member function uses an instance variable that is defined by its class, it does so directly, without explicit reference to an object and without use of the dot operator. This is easy to understand if you think about it. A member function is always invoked relative to some object of its class. Once this invocation has occurred, the object is known. Thus, within a member function, there is no need to specify the object a second time. This means that fuelcap and mpg inside range( ) implicitly refer to the copies of those variables found in the object that invokes range( ). Of course, code outside Vehicle must refer to fuelcap and mpg through an object and by using the dot operator.

A member function must be called relative to a specific object. There are two ways that this can happen. First, a member function can be called by code that is outside its class. In this case, you must use the object's name and the dot operator. For example, this calls range( ) on minivan:

range = minivan.range();

The invocation minivan.range( ) causes range( ) to operate on minivan's copy of the instance variables. Thus, it returns the range for minivan.

The second way a member function can be called is from within another member function of the same class. When one member function calls another member function of the same class, it can do so directly, without using the dot operator. In this case, the compiler already knows which object is being operated upon. It is only when a member function is called by code that does not belong to the class that the object name and the dot operator must be used.

The program shown here puts together all the pieces and missing details, and illustrates the range( ) function:

```
// A program that uses the Vehicle class.

#include <iostream>
using namespace std;

// Declare the Vehicle class.
class Vehicle {
public:
  int passengers; // number of passengers
  int fuelcap;    // fuel capacity in gallons
  int mpg;        // fuel consumption in miles per gallon

  int range();   // compute and return the range <-- Declare range( )
};

// Implement the range member function.
int Vehicle::range() { <-- Implement range( ).
  return mpg * fuelcap;
}

int main() {
  Vehicle minivan; // create a Vehicle object
  Vehicle sportscar; // create another object

  int range1, range2;

  // Assign values to fields in minivan.
  minivan.passengers = 7;
  minivan.fuelcap = 16;
  minivan.mpg = 21;

  // Assign values to fields in sportscar.
  sportscar.passengers = 2;
  sportscar.fuelcap = 14;
  sportscar.mpg = 12;
```

```
  // Compute the ranges assuming a full tank of gas.
  range1 = minivan.range();  <-- Call range() on Vehicle objects.
  range2 = sportscar.range();

  cout << "Minivan can carry " <<  minivan.passengers <<
          " with a range of " << range1 << "\n";

  cout << "Sportscar can carry " <<  sportscar.passengers <<
          " with a range of " << range2 << "\n";

  return 0;
}
```

This program displays the following output:

Minivan can carry 7 with a range of 336

Sportscar can carry 2 with a range of 168

1. What is the :: operator called?
2. What does :: do?
3. If a member function is called from outside its class, it must be called through an object using the dot operator. True or false?

Project 8-1 Creating a Help Class

If one were to try to summarize the essence of the class in one sentence, it might be this: A class encapsulates functionality. Of course, sometimes the trick is knowing where one "functionality" ends and another begins. As a general rule, you will want your classes to be the building blocks of your larger application. To do this, each class must represent a single functional unit that performs clearly delineated actions. Thus, you will want your classes to be as small as possible—but no smaller! That is, classes that contain extraneous functionality confuse and destructure code, but classes that contain too little functionality are fragmented. What is the balance? It is at this point that the science of programming becomes the art of programming. Fortunately, most programmers find that this balancing act becomes easier with experience.

To begin gaining that experience, you will convert the help system from Project 3-3 in Module 3 into a Help class. Let's examine why this is a good idea. First, the help system defines one logical unit. It simply displays the syntax for the C++ control statements. Thus, its functionality is compact and well defined. Second, putting help in a class is an esthetically pleasing approach. Whenever you want to offer the help

system to a user, simply instantiate a help-system object. Finally, because help is encapsulated, it can be upgraded or changed without causing unwanted side effects in the programs that use it.

Step by Step

1. Create a new file called HelpClass.cpp. To save you some typing, you might want to copy the file from Project 3-3, Help3.cpp, into HelpClass.cpp.

2. To convert the help system into a class, you must first determine precisely what constitutes the help system. For example, in Help3.cpp, there is code to display a menu, input the user's choice, check for a valid response, and display information about the item selected. The program also loops until q is pressed. If you think about it, it is clear that the menu, the check for a valid response, and the display of the information are integral to the help system. How user input is obtained, and whether repeated requests should be processed, are not. Thus, you will create a class that displays the help information, the help menu, and checks for a valid selection. These functions will be called helpon( ), showmenu( ),and isvalid( ), respectively.

3. Declare the Help class, as shown here:

```
// A class that encapsulates a help system.
class Help {
public:
  void helpon(char what);
  void showmenu();
  bool isvalid(char ch);
};
```

Notice that this is a function-only class; no instance variables are needed. As explained, data-only and code-only classes are perfectly valid. (Question 9 in the Mastery Check adds an instance variable to the Help class.)

4. Create the helpon( ) function, as shown here:

```
// Display help information.
void Help::helpon(char what) {
  switch(what) {
    case '1':
      cout << "The if:\n\n";
```

```
      cout << "if(condition) statement;\n";
      cout << "else statement;\n";
      break;
    case '2':
      cout << "The switch:\n\n";
      cout << "switch(expression) {\n";
      cout << "  case constant:\n";
      cout << "    statement sequence\n";
      cout << "    break;\n";
      cout << "  // ...\n";
      cout << "}\n";
      break;
    case '3':
      cout << "The for:\n\n";
      cout << "for(init; condition; increment)";
      cout << " statement;\n";
      break;
    case '4':
      cout << "The while:\n\n";
      cout << "while(condition) statement;\n";
      break;
    case '5':
      cout << "The do-while:\n\n";
      cout << "do {\n";
      cout << "  statement;\n";
      cout << "} while (condition);\n";
      break;
    case '6':
      cout << "The break:\n\n";
      cout << "break;\n";
      break;
    case '7':
      cout << "The continue:\n\n";
      cout << "continue;\n";
      break;
    case '8':
      cout << "The goto:\n\n";
      cout << "goto label;\n";
      break;
  }
  cout << "\n";
}
```

5. Create the showmenu( ) function:

```
// Show the help menu.
void Help::showmenu() {
```

```
  cout << "Help on:\n";
  cout << "  1. if\n";
  cout << "  2. switch\n";
  cout << "  3. for\n";
  cout << "  4. while\n";
  cout << "  5. do-while\n";
  cout << "  6. break\n";
  cout << "  7. continue\n";
  cout << "  8. goto\n";
  cout << "Choose one (q to quit): ";
}
```

6. Create the isvalid( ) function, shown here:

```
// Return true if a selection is valid.
bool Help::isvalid(char ch) {
  if(ch < '1' || ch > '8' && ch != 'q')
    return false;
  else
    return true;
}
```

7. Rewrite the main( ) function from Project 3-3 so that it uses the new Help class. The entire listing for HelpClass.cpp is shown here:

```
/*
   Project 8-1

   Convert the Help system from Project 3-3 into
   a Help class.
*/

#include <iostream>
using namespace std;

// A class that encapsulates a help system.
class Help {
public:
  void helpon(char what);
  void showmenu();
  bool isvalid(char ch);
};

// Display help information.
void Help::helpon(char what) {
  switch(what) {
    case '1':
```

```
      cout << "The if:\n\n";
      cout << "if(condition) statement;\n";
      cout << "else statement;\n";
      break;
    case '2':
      cout << "The switch:\n\n";
      cout << "switch(expression) {\n";
      cout << "  case constant:\n";
      cout << "    statement sequence\n";
      cout << "    break;\n";
      cout << "  // ...\n";
      cout << "}\n";
      break;
    case '3':
      cout << "The for:\n\n";
      cout << "for(init; condition; increment)";
      cout << " statement;\n";
      break;
    case '4':
      cout << "The while:\n\n";
      cout << "while(condition) statement;\n";
      break;
    case '5':
      cout << "The do-while:\n\n";
      cout << "do {\n";
      cout << "  statement;\n";
      cout << "} while (condition);\n";
      break;
    case '6':
      cout << "The break:\n\n";
      cout << "break;\n";
      break;
    case '7':
      cout << "The continue:\n\n";
      cout << "continue;\n";
      break;
    case '8':
      cout << "The goto:\n\n";
      cout << "goto label;\n";
      break;
  }
  cout << "\n";
}

// Show the help menu.
```

```
void Help::showmenu() {
  cout << "Help on:\n";
  cout << "  1. if\n";
  cout << "  2. switch\n";
  cout << "  3. for\n";
  cout << "  4. while\n";
  cout << "  5. do-while\n";
  cout << "  6. break\n";
  cout << "  7. continue\n";
  cout << "  8. goto\n";
  cout << "Choose one (q to quit): ";
}

// Return true if a selection is valid.
bool Help::isvalid(char ch) {
  if(ch < '1' || ch > '8' && ch != 'q')
    return false;
  else
    return true;
}

int main()
{
  char choice;
  Help hlpob; // create an instance of the Help class.

  // Use the Help object to display information.
  for(;;) {
    do {
      hlpob.showmenu();
      cin >> choice;
    } while(!hlpob.isvalid(choice));

    if(choice == 'q') break;
    cout << "\n";

    hlpob.helpon(choice);
  }

  return 0;
}
```

When you try the program, you will find that it is functionally the same as in Module 3. The advantage to this approach is that you now have a help system component that can be reused whenever it is needed.

CRITICAL SKILL 8.4: Constructors and Destructors

In the preceding examples, the instance variables of each Vehicle object had to be set manually by use of a sequence of statements, such as:

minivan.passengers = 7; minivan.fuelcap = 16; minivan.mpg = 21;

An approach like this would never be used in professionally written C++ code. Aside from being error prone (you might forget to set one of the fields), there is simply a better way to accomplish this task: the constructor.

A constructor initializes an object when it is created. It has the same name as its class and is syntactically similar to a function. However, constructors have no explicit return type. The general form of a constructor is shown here:

class-name( ) {

// constructor code }

Typically, you will use a constructor to give initial values to the instance variables defined by the class, or to perform any other startup procedures required to create a fully formed object.

The complement of the constructor is the destructor. In many circumstances, an object will need to perform some action or series of actions when it is destroyed. Local objects are created when their block is entered, and destroyed when the block is left. Global objects are destroyed when the program terminates. There are many reasons why a destructor may be needed. For example, an object may need to deallocate memory that it had previously allocated, or an open file may need to be closed. In C++, it is the destructor that handles these types of operations. The destructor has the same name as the constructor, but is preceded by a ~. Like constructors, destructors do not have return types.

Here is a simple example that uses a constructor and a destructor:

```
// A simple constructor and destructor.

#include <iostream>
using namespace std;

class MyClass {
public:
  int x;
```

```
  // Declare constructor and destructor.
  MyClass();  // constructor
  ~MyClass(); // destructor
};

// Implement MyClass constructor.
MyClass::MyClass() {
  x = 10;
}

// Implement MyClass destructor.
MyClass::~MyClass() {
  cout << "Destructing...\n";
}

int main() {
  MyClass ob1;
  MyClass ob2;

  cout << ob1.x << " " << ob2.x << "\n";

  return 0;
}
```

Declare constructor ar
destructor for MyClas

The output from the program is shown here:

10 10

Destructing...

Destructing...

In this example, the constructor for MyClass is

// Implement MyClass constructor. MyClass::MyClass() {

x = 10; }

Notice that the constructor is specified under public. This is because the constructor will be called from code defined outside of its class. This constructor assigns the instance variable x of MyClass the value 10. This constructor is called when an object is created. For example, in the line

MyClass ob1;

the constructor MyClass( ) is called on the ob1 object, giving ob1.x the value 10. The same is true for ob2. After construction, ob2.x also has the value 10.

The destructor for MyClass is shown next:

// Implement MyClass constructor. MyClass::~MyClass() {

```
cout << "Destructing...\n"; }
```

This destructor simply displays a message, but in real programs, the destructor would be used to release one or more resources (such as a file handle or memory) used by the class.

1. What is a constructor and when is it executed?
2. Does a constructor have a return type?
3. When is a destructor called?

CRITICAL SKILL 8.5: Parameterized Constructors

In the preceding example, a parameterless constructor was used. While this is fine for some situations, most often you will need a constructor that has one or more parameters. Parameters are added to a constructor in the same way that they are added to a function: just declare them inside the parentheses after the constructor's name. For example, here is a parameterized constructor for MyClass:

```
Myclass::MyClass(int i) { x = i;

}
```

To pass an argument to the constructor, you must associate the value or values being passed with an object when it is being declared. C++ provides two ways to do this. The first method is illustrated here:

```
MyClass ob1 = MyClass(101);
```

This declaration creates a MyClass object called ob1 and passes the value 101 to it. However, this form is seldom used (in this context), because the second method is shorter and more to the point. In the second method, the argument or arguments must follow the object's name and be enclosed between parentheses. For example, this statement accomplishes the same thing as the previous declaration:

```
MyClass ob1(101);
```

This is the most common way that parameterized objects are declared. Using this method, the general form of passing arguments to a constructor is

class-type var(arg-list);

Here, arg-list is a comma-separated list of arguments that are passed to the constructor.

NOTE: Technically, there is a small difference between the two initialization forms, which you will learn about later in this book. However, this difference does not affect the programs in this module, or most programs that you will write.

Here is a complete program that demonstrates the MyClass parameterized constructor:

```
// A parameterized constructor.

#include <iostream>
using namespace std;

class MyClass {
public:
  int x;

  // Declare constructor and destructor.
  MyClass(int i);  // constructor
  ~MyClass(); // destructor
};

// Implement a parameterized constructor.
MyClass::MyClass(int i) {
    x = i;
}

// Implement MyClass destructor.
MyClass::~MyClass() {
  cout << "Destructing object whose x value is " <<
          x  <<" \n";
}

int main() {
  MyClass t1(5);
  MyClass t2(19);

  cout << t1.x << " " << t2.x << "\n";

  return 0;
}
```

Add a parameter to **MyClass**( ).

Pass arguments to **MyClass** constructor.

The output from this program is shown here:

```
5 19
Destructing object whose x value is 19
Destructing object whose x value is 5
```

In this version of the program, the MyClass( ) constructor defines one parameter called i, which is used to initialize the instance variable, x. Thus, when the line

MyClass ob1(5);

executes, the value 5 is passed to i, which is then assigned to x.

Unlike constructors, destructors cannot have parameters. The reason for this is easy to understand: there is no means by which to pass arguments to an object that is being destroyed. Although the situation is rare, if your object needs to be "passed" some data just before its destructor is called, you will need to create a specific variable for this purpose. Then, just prior to the object's destruction, you will need to set that variable.

Adding a Constructor to the Vehicle Class

We can improve the Vehicle class by adding a constructor that automatically initializes the passengers, fuelcap, and mpg fields when an object is constructed. Pay special attention to how Vehicle objects are created.

```
// Add a constructor to the vehicle class.

#include <iostream>
using namespace std;

// Declare the Vehicle class.
class Vehicle {
public:
```

```
  int passengers; // number of passengers
  int fuelcap;    // fuel capacity in gallons
  int mpg;        // fuel consumption in miles per gallon

  // This is a constructor for Vehicle.
  Vehicle(int p, int f, int m);

  int range();   // compute and return the range
};

// Implement the Vehicle constructor.
Vehicle::Vehicle(int p, int f, int m) {
  passengers = p;
  fuelcap = f;
  mpg = m;
}

// Implement the range member function.
int Vehicle::range() {
  return mpg * fuelcap;
}

int main() {
  // Pass values to Vehicle constructor.
  Vehicle minivan(7, 16, 21);
  Vehicle sportscar(2, 14, 12);

  int range1, range2;

  // Compute the ranges assuming a full tank of gas.
  range1 = minivan.range();
  range2 = sportscar.range();

  cout << "Minivan can carry " <<  minivan.passengers <<
          " with a range of " << range1 << "\n";

  cout << "Sportscar can carry " <<  sportscar.passengers <<
          " with a range of " << range2 << "\n";

  return 0;
}
```

Vehicle's constructor initializes **passengers**, **fuelcap**, and **mpg**.

Pass the vehicle information to **Vehicle** using its constructor.

Both minivan and sportscar were initialized by the Vehicle( ) constructor when they were created. Each object is initialized as specified in the parameters to its constructor. For example, in the line

Vehicle minivan(7, 16, 21);

the values 7, 16, and 21 are passed to the Vehicle( ) constructor. Therefore, minivan's copy of passengers, fuelcap, and mpg will contain the values 7, 16, and 21, respectively. Thus, the output from this program is the same as the previous version.

An Initialization Alternative

If a constructor takes only one parameter, then you can use an alternative method to initialize it. Consider the following program:

```
// An alternate initialization method.

#include <iostream>
using namespace std;

class MyClass {
public:
  int x;

  // Declare constructor and destructor.
  MyClass(int i);  // constructor
  ~MyClass(); // destructor
};

// Implement a parameterized constructor.
MyClass::MyClass(int i) {
    x = i;
}

// Implement MyClass destructor.
MyClass::~MyClass() {
  cout << "Destructing object whose x value is " <<
          x  <<" \n";
}

int main() {
  MyClass ob = 5;  // calls MyClass(5)

  cout << ob.x << "\n";

  return 0;
}
```

An alternative syntax for initializing an object.

Here, the constructor for MyClass takes one parameter. Pay special attention to how ob is declared in main( ). It uses this declaration:

```
MyClass ob = 5;
```

In this form of initialization, 5 is automatically passed to the i parameter in the MyClass( ) constructor. That is, the declaration statement is handled by the compiler as if it were written like this:

```
MyClass ob (5);
```

In general, any time that you have a constructor that requires only one argument, you can use either ob(x) or ob = x to initialize an object. The reason is that whenever you create a constructor that takes one argument, you are also implicitly creating a conversion from the type of that argument to the type of the class.

Remember that the alternative shown here applies only to constructors that have exactly one parameter.

1. Assuming a class called Test, show how to declare a constructor that takes one int parameter called count.
2. How can this statement be rewritten?
   Test ob = Test(10);
3. How else can the declaration in the second question be rewritten?

CRITICAL SKILL 8.6: Inline Functions

Before we continue exploring the class, a small but important digression is in order. Although it does not pertain specifically to object-oriented programming, one very useful feature of C++, called an inline function, is frequently used in class definitions. An inline function is a function that is expanded inline at the point at which it is invoked, instead of actually being called. There are two ways to create an inline function. The first is to use the inline modifier.

Ask the Expert

Q: Can one class be declared within another? That is, can classes be nested?

A: Yes, it is possible to define one class within another. Doing so creates a nested class. Since a class declaration does, in fact, define a scope, a nested class is valid only within the scope of the enclosing class. Frankly, because of the richness and flexibility of C++'s other features, such as inheritance, discussed later in this book, the need to create a nested class is virtually nonexistent.

For example, to create an inline function called f that returns an int and takes no parameters, you declare it like this:

```
inline int f()
{
// ...
}
```

The inline modifier precedes all other aspects of a function's declaration.

The reason for inline functions is efficiency. Every time a function is called, a series of instructions must be executed, both to set up the function call, including pushing any arguments onto the stack, and to return from the function. In some cases, many CPU cycles are used to perform these procedures. However, when a function is expanded inline, no such overhead exists, and the overall speed of your program will increase. Even so, in cases where the inline function is large, the overall size of your program will also increase. For this reason, the best inline functions are those that are small. Most large functions should be left as normal functions.

The following program demonstrates inline:

```
// Demonstrate inline.

#include <iostream>
using namespace std;

class cl {
  int i; // private by default
public:
  int get_i();
  void put_i(int j);
} ;

inline int cl::get_i()
{
  return i;
}

inline void cl::put_i(int j)
{
  i = j;
}

int main()
{
  cl s;

  s.put_i(10);
  cout << s.get_i();

  return 0;
}
```

get_i( ) and put_i( ) are expanded inline.

It is important to understand that technically, inline is a request, not a command, that the compiler generate inline code. There are various situations that might prevent the compiler from complying with the request. Here are some examples:

- Some compilers will not generate inline code if a function contains a loop, a switch,ora goto.

- Often, you cannot have inline recursive functions.
- Inline functions that contain static variables are frequently disallowed.

Remember: Inline restrictions are implementation-dependent, so you must check your compiler's documentation to find out about any restrictions that may apply in your situation.

Creating Inline Functions Inside a Class

The second way to create an inline function is by defining the code to a member function inside a class definition. Any function that is defined inside a class definition is automatically made into an inline function. It is not necessary to precede its declaration with the keyword inline. For example, the preceding program can be rewritten as shown here:

```
#include <iostream>
using namespace std;
class cl {
  int i; // private by default
public:
  // Automatic inline functions.
  int get_i() { return i; }
  void put_i(int j) { i = j; }
} ;

int main()
{
  cl s;

  s.put_i(10);
  cout << s.get_i();

  return 0;
}
```

Define **get_i( )** and **put_i( )** inside their class.

Notice the way the function code is arranged. For very short functions, this arrangement reflects common C++ style. However, you could write them as shown here:

```
class cl {
  int i; // private by default
public:
  // inline functions
  int get_i()
  {
    return i;
  }

  void put_i(int j)
  {
     i = j;
  }
};
```

Short functions, like those illustrated in this example, are usually defined inside the class declaration. In-class, inline functions are quite common when working with classes because frequently a public function provides access to a private variable. Such functions are called accessor functions. Part of successful object-oriented programming is controlling access to data through member functions. Because most C++ programmers define accessor functions and other short member functions inside their classes, this convention will be followed by the rest of the C++ examples in this book. It is an approach that you should use, too.

Here is the Vehicle class recoded so that its constructor, destructor, and range( ) function are defined inside the class. Also, the passengers, fuelcap, and mpg fields have been made private, and accessor functions have been added to get their values.

```
// Defines constructor, destructor, and range() function in-line.

#include <iostream>
using namespace std;

// Declare the Vehicle class.
class Vehicle {
  // These are now private.
  int passengers; // number of passengers
  int fuelcap;    // fuel capacity in gallons
  int mpg;        // fuel consumption in miles per gallon
public:
  // This is a constructor for Vehicle.
  Vehicle(int p, int f, int m) {
    passengers = p;
    fuelcap = f;
    mpg = m;
  }

 // Compute and return the range.
  int range() { return mpg * fuelcap; }

  // Accessor functions.
  int get_passengers() { return passengers; }
  int get_fuelcap() { return fuelcap; }
  int get_mpg() { return mpg; }
};

int main() {
  // Pass values to Vehicle constructor.
  Vehicle minivan(7, 16, 21);
  Vehicle sportscar(2, 14, 12);

  int range1, range2;

  // Compute the ranges assuming a full tank of gas.
  range1 = minivan.range();
  range2 = sportscar.range();

  cout << "Minivan can carry " <<  minivan.get_passengers() <<
          " with a range of " << range1 << "\n";
```

Make these variables private.

Define functions inline and access private variables through accessor functions.

Because the member variables of Vehicle are now private, the accessor function get_passengers( ) must be used inside main( ) to obtain the number of passengers that a vehicle can hold.

1. What does inline do?

2. Can an inline function be declared inside a class declaration?

3. What is an accessor function?

Project 8-2 Creating a Queue Class

As you may know, a data structure is a means of organizing data. The simplest data structure is the array, which is a linear list that supports random access to its elements. Arrays are often used as the underpinning for more sophisticated data structures, such as stacks and queues. A stack is a list in which elements can be accessed in first-in, last-out (FILO) order only. A queue is a list in which elements can be accessed in first-in, first-out (FIFO) order only. Thus, a stack is like a stack of plates on a table; the first down is the last to be used. A queue is like a line at a bank; the first in line is the first served.

What makes data structures such as stacks and queues interesting is that they combine storage for information with the functions that access that information. Thus, stacks and queues are data engines in which storage and retrieval is provided by the data structure itself, and not manually by your program. Such a combination is, obviously, an excellent choice for a class, and in this project, you will create a simple queue class. In general, queues support two basic operations: put and get. Each put operation places a new element on the end of the queue. Each get operation retrieves the next element from the front of the queue. Queue operations are consumptive. Once an element has been retrieved, it cannot be retrieved again. The queue can also become full if there is no space available to store an item, and it can become empty if all of the elements have been removed.

One last point: there are two basic types of queues, circular and non-circular. A circular queue reuses locations in the underlying array when elements are removed. A non-circular queue does not and eventually becomes exhausted. For the sake of simplicity, this example creates a non-circular queue, but with a little thought and effort, you can easily transform it into a circular queue.

Step by Step

1. Create a file called Queue.cpp.

2. Although there are other ways to support a queue, the method we will use is based upon an array. That is, an array will provide the storage for the items put into the queue. This array will be accessed through two indices. The put index determines where the next element of data will be stored. The get index indicates at what location the next element of data will be obtained. Keep in mind that the get operation is consumptive, and it is not possible to retrieve the same element twice. Although the queue that we will be creating stores characters, the same logic can be used to store any type of object. Begin creating the Queue class with these lines:

```
const int maxQsize = 100;

class Queue {
  char q[maxQsize]; // this array holds the queue
  int size; // maximum number of elements the queue can store
  int putloc, getloc; // the put and get indices
```

The const variable maxQsize defines the size of the largest queue that can be created. The actual size of the queue is stored in the size field.

3. The constructor for the Queue class creates a queue of a given size. Here is the Queue constructor:

```
public:

// Construct a queue of a specific length.
Queue(int len) {
  // Queue must be less than max and positive.
  if(len > maxQsize) len = maxQsize;
  else if(len <= 0) len = 1;

  size = len;
  putloc = getloc = 0;
}
```

If the requested queue size is greater than maxQsize, then the maximum size queue is created. If the requested queue size is zero or less, a queue of length 1 is created. The size of the queue is stored in the size field. The put and get indices are initially set to zero.

4. The put( ) function, which stores elements, is shown next:

```
// Put a character into the queue.
void put(char ch) {
  if(putloc == size) {
    cout << " -- Queue is full.\n";
    return;
  }

  putloc++;
  q[putloc] = ch;
}
```

The function begins by checking for a queue-full condition. If putloc is equal to the size of the queue, then there is no more room in which to store elements. Otherwise, putloc is incremented, and the new element is stored at that location. Thus, putloc is always the index of the last element stored.

5. To retrieve elements, use the get( ) function, shown next:

```
// Get a character from the queue.
char get() {
  if(getloc == putloc) {
    cout << " -- Queue is empty.\n";
    return 0;
  }

  getloc++;
  return q[getloc];
}
```

Notice first the check for queue-empty. If getloc and putloc both index the same element, then the queue is assumed to be empty. This is why getloc and putloc were both initialized to zero by the Queue constructor. Next, getloc is incremented and the next element is returned. Thus, getloc always indicates the location of the last element retrieved.

6. Here is the entire Queue.cpp program:

```
/*
   Project 8-2

   A queue class for characters.
*/
#include <iostream>
using namespace std;

const int maxQsize = 100;

class Queue {
  char q[maxQsize]; // this array holds the queue
  int size; // maximum number of elements the queue can store
  int putloc, getloc; // the put and get indices
public:

  // Construct a queue of a specific length.
  Queue(int len) {
    // Queue must be less than max and positive.
    if(len > maxQsize) len = maxQsize;
    else if(len <= 0) len = 1;

    size = len;
    putloc = getloc = 0;
  }

  // Put a character into the queue.
  void put(char ch) {
    if(putloc == size) {
      cout << " -- Queue is full.\n";
      return;
    }

    putloc++;
    q[putloc] = ch;
  }
```

```
  // Get a character from the queue.
  char get() {
    if(getloc == putloc) {
      cout << " -- Queue is empty.\n";
      return 0;
    }

    getloc++;
    return q[getloc];
  }
};

// Demonstrate the Queue class.
int main() {
  Queue bigQ(100);

  Queue smallQ(4);
  char ch;
  int i;

  cout << "Using bigQ to store the alphabet.\n";
  // put some numbers into bigQ
  for(i=0; i < 26; i++)
    bigQ.put('A' + i);

  // retrieve and display elements from bigQ
  cout << "Contents of bigQ: ";
  for(i=0; i < 26; i++) {
    ch = bigQ.get();
    if(ch != 0) cout << ch;
  }

  cout << "\n\n";
  cout << "Using smallQ to generate errors.\n";

  // Now, use smallQ to generate some errors
  for(i=0; i < 5; i++) {
    cout << "Attempting to store " <<
                  (char) ('Z' - i);

    smallQ.put('Z' - i);

    cout << "\n";
  }
  cout << "\n";
```

```
  // more errors on smallQ
  cout << "Contents of smallQ: ";
  for(i=0; i < 5; i++) {
    ch = smallQ.get();

    if(ch != 0) cout << ch;
  }

  cout << "\n";
}
```

7. The output produced by the program is shown here:

```
Using bigQ to store the alphabet.
Contents of bigQ: ABCDEFGHIJKLMNOPQRSTUVWXYZ
Using smallQ to generate errors.
Attempting to store Z
Attempting to store Y
Attempting to store X
Attempting to store W
Attempting to store V -- Queue is full.

Contents of smallQ: ZYXW -- Queue is empty.
```

8. On your own, try modifying Queue so that it stores other types of objects. For example, have it store ints or doubles.

CRITICAL SKILL 8.7: Arrays of Objects

You can create arrays of objects in the same way that you create arrays of any other data type. For example, the following program creates an array of MyClass objects. The objects that comprise the elements of the array are accessed using the normal array-indexing syntax.

```
// Create an array of objects.

#include <iostream>
using namespace std;

class MyClass {
  int x;
public:
  void set_x(int i) { x = i; }
  int get_x() { return x; }
};

int main()
{
  MyClass obs[4]; <-- Create an array of objects.
  int i;

  for(i=0; i < 4; i++)
    obs[i].set_x(i);

  for(i=0; i < 4; i++)
    cout << "obs[" << i << "].get_x(): " <<
            obs[i].get_x() << "\n";

  return 0;
}
```

This program produces the following output:

```
obs[0].get_x(): 0
obs[1].get_x(): 1
obs[2].get_x(): 2
obs[3].get_x(): 3
```

CRITICAL SKILL 8.8: Initializing Object Arrays

If a class includes a parameterized constructor, an array of objects can be initialized. For example, here MyClass is a parameterized class, and obs is an initialized array of objects of that class.

```
// Initialize an array of objects.

#include <iostream>
using namespace std;

class MyClass {
  int x;
public:
  MyClass(int i) { x = i; }
  int get_x() { return x; }
};
```

```
int main()
{
  MyClass obs[4] = { -1, -2, -3, -4 };
  int i;

  for(i=0; i < 4; i++)
    cout << "obs[" << i << "].get_x(): " <<
            obs[i].get_x() << "\n";

  return 0;
}
```

One way to initialize an array of objects.

In this example, the values –1 through –4 are passed to the MyClass constructor function. This program displays the following output:

```
obs[0].get_x(): -1
obs[1].get_x(): -2
obs[2].get_x(): -3
obs[3].get_x(): -4
```

Actually, the syntax shown in the initialization list is shorthand for this longer form:

```
MyClass obs[4] = { MyClass(-1), MyClass (-2),
                   MyClass (-3), MyClass (-4) };
```

Another way to initialize the array.

As explained earlier, when a constructor takes only one argument, there is an implicit conversion from the type of that argument to the type of the class. The longer form simply calls the constructor directly.

When initializing an array of objects whose constructor takes more than one argument, you must use the longer form of initialization. For example:

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

class MyClass {
  int x, y;
public:
  MyClass(int i, int j) { x = i; y = j; }
  int get_x() { return x; }
  int get_y() { return y; }
};

int main()
{
  MyClass obs[4][2] = {
    MyClass(1, 2), MyClass(3, 4),
    MyClass(5, 6), MyClass(7, 8),
    MyClass(9, 10), MyClass(11, 12),
    MyClass(13, 14), MyClass(15, 16)
  };

  int i;

  for(i=0; i < 4; i++) {
    cout << obs[i][0].get_x() << ' ';
    cout << obs[i][0].get_y() << "\n";
    cout << obs[i][1].get_x() << ' ';
    cout << obs[i][1].get_y() << "\n";
  }

  return 0;
}
```

The long initialization form must be used when two or more arguments are required by the object's constructor.

In this example, MyClass' constructor takes two arguments. In main( ), the array obs is declared and initialized using direct calls to MyClass' constructor. When initializing arrays you can always use the long form of initialization, even if the object takes only one argument. It's just that the short form is more convenient when only one argument is required. The program displays the following output:

```
1 2
3 4
5 6
7 8
9 10
11 12
13 14
15 16
```

CRITICAL SKILL 8.9: Pointers to Objects

You can access an object either directly (as has been the case in all preceding examples), or by using a pointer to that object. To access a specific element of an object when using a pointer to the object, you must use the arrow operator: –>. It is formed by using the minus sign followed by a greater-than sign.

To declare an object pointer, you use the same declaration syntax that you would use to declare a pointer for any other type of data. The next program creates a simple class called P_example, defines an object of that class called ob, and defines a pointer to an object of type P_example called p. It then illustrates how to access ob directly, and how to use a pointer to access it indirectly.

```
// A simple example using an object pointer.

#include <iostream>
using namespace std;

class P_example {
  int num;
public:
  void set_num(int val) { num = val; }
  void show_num(){ cout << num << "\n"; }
};

int main()
{
  P_example ob, *p; // declare an object and pointer to it

  ob.set_num(1); // call functions directly on ob
  ob.show_num();

  p = &ob; // assign p the address of ob
  p->set_num(20); // call functions through a pointer to ob
  p->show_num();

  return 0;
}
```

Pointers to objects are used like other types of pointers.

Notice the use of the arrow operator.

Notice that the address of ob is obtained using the & (address of) operator in the same way that the address is obtained for any type of variable.

As you know, when a pointer is incremented or decremented, it is increased or decreased in such a way that it will always point to the next element of its base type. The same thing occurs when a pointer to an object is incremented or decremented: the next object is pointed to. To illustrate this, the preceding program has been modified here so that ob is a two-element array of type P_example. Notice how p is incremented and decremented to access the two elements in the array.

```
// Incrementing and decrementing an object pointer.

#include <iostream>
using namespace std;

class P_example {
  int num;
```

```
public:
  void set_num(int val) { num = val; }
  void show_num(){ cout << num << "\n"; }
};

int main()
{
  P_example ob[2], *p;

  ob[0].set_num(10);  // access objects directly
  ob[1].set_num(20);

  p = &ob[0];  // obtain pointer to first element
  p->show_num(); // show value of ob[0] using pointer
  p++;  // advance to next object
  p->show_num(); // show value of ob[1] using pointer

  p--;  // retreat to previous object
  p->show_num(); // again show value of ob[0]

  return 0;
}
```

The output from this program is 10, 20, 10.

As you will see later in this book, object pointers play a pivotal role in one of C++'s most important concepts: polymorphism.

1. Can an array of objects be given initial values?
2. Given a pointer to an object, what operator is used to access a member?

Object References

Objects can be referenced in the same way as any other data type. No special restrictions or instructions apply.

Module 8 Mastery Check

1. What is the difference between a class and an object?
2. What keyword is used to declare a class?

3. What does each object have its own copy of?
4. Show how to declare a class called Test that contains two private int variables called count and max.
5. What name does a constructor have? What name does a destructor have?
6. Given this class declaration:

```
class Sample {
  int i;
public:
  Sample(int x) { i = x }
  // ...
};
```

   show how to declare a Sample object that initializes i to the value 10.
7. When a member function is declared within a class declaration, what optimization automatically takes place?
8. Create a class called Triangle that stores the length of the base and height of a right triangle in two private instance variables. Include a constructor that sets these values. Define two functions. The first is hypot( ), which returns the length of the hypotenuse. The second is area( ), which returns the area of the triangle.
9. Expand the Help class so that it stores an integer ID number that identifies each user of the class. Display the ID when a help object is destroyed. Return the ID when the function getID( ) is called.