12 Object-Oriented Programming

OBJECTIVES

- *Understand the difference between procedural programming and object-oriented programming.*
- *Understand the principles of object-oriented programming.*
- *Learn how to read classes and use objects in C++.*
- *Understand how objects help programmers reuse code.*
- *Understand containment and inheritance in C++ object-oriented programming.*

Overview

As the field of computer science has grown over the years, several different methods have been developed for programming computers. The first programming was done by flipping switches on a control panel or feeding machine language instructions into a computer. Recall from Chapter 2 that the development of assembly language was followed by the development of high-level languages which made programming easier.

One of the more recent developments in programming is known as object-oriented programming (OOP). Object-oriented programming in some ways continues the trend toward higher-level programming languages. OOP, however, does more than that. Object-oriented programming changes the way a programmer uses data and functions.

Object-oriented programming is not unique to C++. Many languages support object-oriented programming. In this chapter you will learn about the object-oriented method of programming, its benefits, and how to read and use the C++ implementation of object-oriented programming.

CHAPTER 12, SECTION 1

Procedural Programming vs. Object-Oriented Programming

The different methods used for writing programs are known as paradigms. A ***paradigm*** is a model or a set of rules that define a way of programming. There are two primary paradigms used to program computers today: the procedural paradigm and the object-oriented paradigm.

PROCEDURAL PARADIGM

The procedural paradigm is the paradigm you have used in this book up to this point. The ***procedural paradigm*** focuses on the idea that all algorithms in a program are performed with functions and data that a programmer can see, understand, and change. In a program written procedurally, the focus is on the functions that will process the data. The programmer then devises ways to pass the required data to and from the functions which do the processing. To be successful writing procedural programs, the programmer must understand how all data is stored and how the algorithms of the program work.

Consider, for example, what you learned in Chapter 6 about strings. After you learned what strings are, you then had to learn how strings are stored in memory using character arrays and how strings end with a null terminator. Without the knowledge of how strings are implemented in C++, you could not successfully store strings in your programs. Procedural programming typically requires this kind of knowledge about the way data is stored.

Procedural programming also requires that you know how data is stored in order to write the functions that will process the data. For example, recall the exercise from Chapter 6 in which you copied a string into a character array which was not large enough to hold the string. The result was a loss of data in the sur-

rounding bytes of memory. In procedural programming, the programmer must be concerned with many similar technical details.

Procedural programming is not all bad. In fact, procedural programming has served many programmers well for many years and will continue to do so for some time. But computer scientists are always searching for a better way to develop software. By taking a look at the world around them, computer scientists discovered that the world consists of objects that perform work and interact with each other. When applied to programming computers, the result is a different paradigm: object-oriented programming.

Extra for Experts

There are many different programming paradigms in computer science. Some of the paradigms are procedural, functional, object-oriented, and logic. Most common languages such as C, FORTRAN, and standard Pascal are procedural. C++, Smalltalk, and Java are well known object-oriented languages. The functional paradigm includes languages like LISP and Scheme. Finally, Prolog is a language in the logic paradigm.

OBJECT-ORIENTED PARADIGM

The ***object-oriented paradigm*** centers on the idea that all programs can be made up of separate entities called ***objects***. A program built from these objects is called an ***object-oriented program***. Each of the objects used to build the program has a specific responsibility or purpose. For example, an object in an object-oriented program might store a string of characters. The string itself, and all of the operations that can be performed with the string are part of the object. The string object can initialize itself, store a string provided to it, and perform other functions such as reporting the length of the string the object holds. In other words, instead of an object being directly manipulated by other parts of the program, an object manipulates itself. Building programs using the object-oriented paradigm is called ***object-oriented programming*** or ***OOP***.

On the Net

For a history of the object-oriented paradigm, see topic 12.1.1 at http://www.ProgramCPP.com.

Communication among objects is similar to communication among people. You cannot look inside of someone's head to see what they know. You must ask questions and allow the person to provide a response. In *OOP*, data is transferred through ***messages*** which are exchanged among objects. The data in an object is not intended to be accessed directly by code which is outside of the object (see Figure 12-1). For example, the string object mentioned above can be initialized by sending a message to the object. You could also send a message to the object asking for the length of the string currently stored in the object. The string object would then respond with the requested information. Inside the object is the code (called ***methods***) necessary to perform the operations on the object.

Note

Messages can do more than simply initialize an object or return a length of a string. A message can perform a high-level task such as sort information in the object. A string object could even include a method to check the spelling of the text in the string.

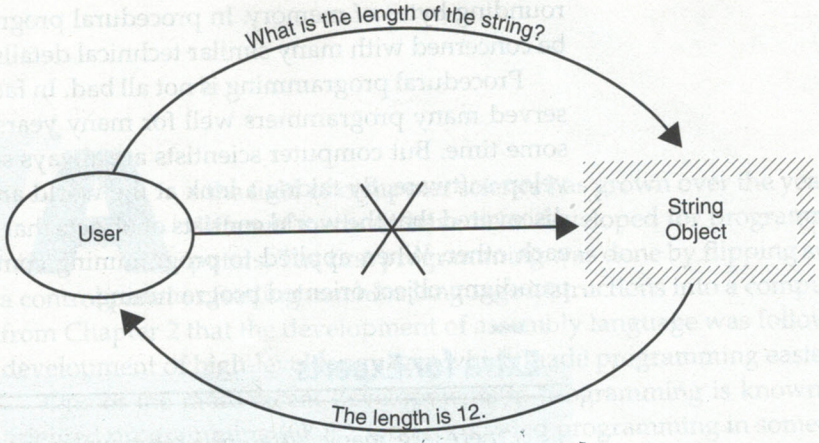

FIGURE 12-1
To get data, objects must pass messages back and forth. Objects cannot simply take the data they need.

Communication among objects takes the form of messages because objects hide all of their data and internal operations. This "hiding" of data and code is known as ***encapsulation***. By using encapsulation, objects can protect and guarantee the integrity of their data. In procedural programming, poorly written functions can often change important data, causing problems throughout the program. The threat of a poorly written function changing data is reduced when using the object-oriented paradigm.

Believe it or not, you have already started to use some of the object-oriented features of the C++ language. The first program you compiled and ran contained the statement below.

```
cout << "My first C++ program.\n";
```

The streams **cin** and **cout** are actually objects which represent console input (usually the keyboard) and console output (usually the monitor), respectively. Now that you know something about the object-oriented paradigm, you can understand that the statement is actually a message to the **cout** object directing the object to display the indicated text.

There is more to object-oriented programming than what has been mentioned in this section. The first step, however, is to visualize how a program can be implemented using objects, learn what is in an object, and understand how objects communicate with other parts of the program. Object-oriented programming in C++ does not take the place of what you have already learned. Instead, it extends the features of the language and gives you a new way to organize programs. The structures and data types you have been using are also used in object-oriented programs. In fact, what you have learned in previous chapters will provide the foundation you need to be successful as an object-oriented programmer.

Note

When you change format options with **setf** *and* **unsetf**, *you are actually sending a message to the* **cout** *object.*

On the Net

In some programming languages, such as Smalltalk and Java, practically everything is object-oriented. These languages are sometimes referred to as pure OOP languages. C++, however, is more of a hybrid language. You can make use of OOP, but you can also program without using OOP. For more information about OOP purity and other OOP languages, see http://www.ProgramCPP.com. See topic 12.1.2.

SECTION 12.1 QUESTIONS

1. What programming paradigm has been used in the previous chapters of this book?
2. How do objects communicate with other objects?
3. Where is the data that an object manipulates stored?
4. What is encapsulation?
5. Identify a C++ object that you have already used.

PROBLEM 12.1.1

Select one of the programs you have written previously in this course and propose a way that the program could be implemented using objects.

PROBLEM 12.1.2

Use a Web search engine to locate information on the WWW about object-oriented programming (OOP). Also search for OOA (object-oriented analysis) and OOD (object-oriented design). After you see what your search finds, go to http://www.ProgramCPP.com and check out the links to object-oriented programming resources found in topic 12.1.3.

CHAPTER 12, SECTION 2

Object-Oriented Programming in C++

Now that you understand some of the concepts of the object-oriented paradigm, you can begin to create programs which use objects. Several different programming languages incorporate the object-oriented paradigm. Each language which supports OOP implements the paradigm in a slightly different way. In this book, you will learn the C++ implementation.

In this section, you will run a program that uses a circle object. You will see how encapsulation and messages make it possible for you to use the object without knowing how the data is stored in the object or how the algorithms are implemented.

CLASSES

Before an object can be created in C++, the compiler must have a definition to know how to create that type of object. The definition for an object is known as a *class*. Using a class is similar to using the basic data types in C++. For example, suppose that a class is created that supports objects which represent circles. This class is named **`circle`**. The class **`circle`** is what tells the compiler how to create

each object of type **circle**. You can declare a **circle** object like you declare the other data types in C++.

For example, the statement below declares an integer named **my_int**. The statement instructs the compiler to reserve memory for the data.

```
int my_int;
```

In the same way, to declare an object of the type **circle**, the code below would be used.

```
circle my_circle;
```

When the compiler encounters this declaration, it refers to the **circle** definition to find out how much memory to reserve for a **circle** object. The object is then created in memory and given the name **my_circle**. In the same way that many different variables of the same data type can be created, many different objects of the type **circle** can be created. Each different object of the type **circle** is independent of the other **circle** objects in the program.

Because objects are more complex than the simple data types, a different term is used to describe the declaration of an object. When you declare an object, you say that you have ***instanciated*** the object. In other words, you have created an ***instance*** of a class. An instance is the data for one object that has the behaviors as defined by the class. If you were to instanciate two **circle** objects, you would have two independent objects which could be used to represent two distinct circles.

Note

When multiple objects are instanciated from the same class, the code required to perform the operations (methods) is not duplicated in memory for each object. The data for each object is stored separately in memory, but all objects defined by the same class share the same code.

EXERCISE 12-1 UNDERSTANDING OBJECTS

1. Load the program *OOP.CPP*.
2. Look at the program and see how the objects are instanciated and how messages are passed between the objects. Notice that there is no way of knowing how the **circle** objects store the radius or how they calculate area.
3. Run the program.
4. Leave the source code file open while you analyze the program in the paragraphs which follow.

On the Net

Another way to look at the relationship between a class and an object is to consider examples in the real world. If the definition of a human being were a class, then you and I are instances of that class. We are not the same person, but we both have the characteristics of the human class. An automobile can be thought of as an instance of a class. Any pickup truck you see on the road is an instance of the pickup truck class of vehicles. For more examples of classes and objects, see http://www.ProgramCPP.com. See topic 12.2.1.

Let's analyze the source code you ran in the previous exercise. After the beginning comments, there are two compiler directives.

```
#include "circle.h"                    // contains the circle class
#include <iostream.h>
```

The first compiler directive includes the header file **circle.h**. The **circle.h** header file contains the definition for the **circle** class. Without the class definition, the compiler would not know how to create a **circle** object nor would it know what properties a **circle** object has. The next compiler directive includes the **iostream.h** header file so that the program can get input from the user and output data to the screen.

Note

You may have noticed that the **#include** *directives in this example use different characters around the header file name. Use quotation marks ("") when the header file is a source file in the same location as your program source code. Use the less than and greater than symbols (<>), sometimes called angle brackets, when the header file is one of the compiler's pre-compiled library functions.*

In the main function, the program instanciates two copies of the **circle** class, **Circle_One** and **Circle_Two**. A variable to hold a radius and an area of a circle are also declared. The variables **User_Radius** and **Area** are used in your program and are not part of either object.

```
int main()
{
 circle Circle_One;                    // instanciate objects
 circle Circle_Two;                    // of type circle
 float User_Radius;
 double Area;
```

The program then prompts the user for the radius of the first circle. After the program receives the user's response, it sends a message to **Circle_One** requesting that it set its radius to **User_Radius**.

```
cout << "\nWhat is the radius of the first circle? ";
cin  >> User_Radius;

Circle_One.SetRadius(User_Radius); // Send a message to Circle_One telling
                                   // it to set its radius to User_Radius
```

After the radius of **Circle_One** has been set to **User_Radius**, the program prompts the user for the radius of the second circle and sets the radius of **Circle_Two** using the same function that was used to set the radius of **Circle_One**. You can see that messages are sent to the **circle** objects by using the object's name, a period, and the message. The period used between the identifier and the message is called the ***class-member operator*** or ***member selection operator***.

```
    cout << "\nWhat is the radius of the second circle? ";
    cin  >> User_Radius;

    Circle_Two.SetRadius(User_Radius);  // Send a message to Circle_Two telling
                                        // it to set its radius to User_Radius
```

Finally, the program sends a message to each of the **circle** objects and requests their area. The area of **Circle_One** is assigned to **Area** then output to the screen. Then the area of the second circle is also retrieved and output to the screen.

```
    Area = Circle_One.Area();           // Send a message to Circle_One asking
                                        // for its area
    cout.setf(ios::fixed);
    cout << "\nThe area of the first circle is " << Area << ".\n";

    Area = Circle_Two.Area();           // Send a message to Circle_Two asking
                                        // for its area
    cout << "\nThe area of the second circle is " << Area << ".\n";
    cout.unsetf(ios::fixed);
    return 0;
}
```

There are many different ways the radius could be stored in the object as well as different ways the area could be calculated. Is the size of the circle stored in the form of the radius or the diameter? How is the area calculated? Neither of these things are known. To use the object, however, you do not need this information.

Also notice that you can only change the properties of the **circle** objects by using messages. To set the radius, you have to send a message to the **circle** object stating the new radius. What would happen if you sent the object a negative radius? The object could check the new radius to make sure it was positive, and not set the new value if it was incorrect. This principle of data protection and encapsulation is fundamental to the object-oriented paradigm.

SECTION 12.2 QUESTIONS

1. What is a class?
2. Suppose you have access to a class named **string**. Write a declaration for an object named **lastname** of type **string**.
3. When you declare an object, you have created an _____ of a class.
4. What is the term that describes the period used to separate an object's identifier from the message to the object?
5. What must you know about the **circle** class in order to use it?

PROBLEM 12.2.1

Write a program similar to *OOP.CPP* that instanciates a circle object, sets the radius of the circle to 1.5, then obtains the area of the circle from the object. The

program should then set the radius of the circle to the value provided by the **Area** method. As output, the program should provide the area of the circle with a radius of 1.5 and the area of the circle at the end of the program.

PROBLEM 12.2.2

Write a program that instanciates a bucket object based on the class definition in *BUCKET.H* on your work disk. See if you can write the program without looking at the definition in *BUCKET.H*. The program should perform the following operations:

1. Instanciate an object of type **bucket**.
2. Use the **SetGallonSize()** method to set the bucket size to 5.0 gallons.
3. Use the **FillBucket()** method to fill the bucket with 3.2 gallons of water.
4. Use the **GetWeight()** method to output the weight of the 3.2 gallons of water.

CHAPTER 12, SECTION 3

Designing and Implementing a Class

As you saw in the previous section, using a class can be easy and uses syntax which is similar to what you have used before. At some point, however, you will need to design a class or modify an existing class. To do so, you will have to understand how a class is implemented in C++. Implementing a class is not difficult, but it does involve some new syntax that you may have not seen before. In this section, you will be introduced to class design and the implementation of classes by analyzing the **circle** class you used in the previous section.

DESIGNING A CLASS

Designing a class requires you think in an object-oriented way. For example, consider a typical telephone answering machine. It encapsulates the functions of an answering machine as well as the data (your incoming and outgoing messages). The buttons on the answering machine are the equivalent of messages. Pushing the Play button sends a message to the answering machine to play the stored messages. It is not hard to understand how an answering machine is an object that contains all of the storage and functions it needs within itself.

On the Net

Can you implement a simulated answering machine using object-oriented programming? Design a class that could implement an answering machine. You can see one implementation at http://www.ProgramCPP.com. See topic 12.3.1.

To design a class, you must think of computer programs in the same way you think of an answering machine or other objects around you. If you were to design the **circle** class you used in the previous section, you should first take into account what needs to be stored and what functions are necesssary. In other words, you define the purpose of the object. The purpose will determine how an object is coded, what data it will hold, and how its operations will be implemented.

In the case of the circle, all that is required to define a circle is a radius. You then decide what functions the object needs to perform. For example, you may want the circle to report its area and circumference. The circle also needs to be able to set its radius.

A class should be designed with enough functions and data to perform its responsibilities—no more and no less. You've never seen an answering machine that can function as a stapler. Likewise, a class should not perform an unrelated task. You also do not need to store more data than is necessary. For example, since the radius is all that is necessary to define a circle, you shouldn't store both a radius and a diameter.

Object-oriented design (often abbreviated OOD) involves much more than the guidelines outlined here. In Case Study VI you will learn more about object-oriented design.

IMPLEMENTING A CLASS

The best way to learn how to implement a class is to study the code of an implemented class. Figure 12-2 is the header file **circle.h** which was used in Exercise 12-1. You have used header files such as **iostream.h** and **math.h** before. However, you may not have ever opened one to see what's inside. A header file is a source code file that typically includes code or declarations of code that you intend to include in more than one program. Classes are normally defined in header files so that they may be reused.

Note

Header files normally contain declarations of variables, functions, and classes, but not the implementation of the functions and classes. This is the case with almost all header files that come with a C++ compiler. However, in some situations, the functions and classes may also be implemented in the same header file. In the next chapter, you will see an example of a class that has the implementation in a file separate from the declarations.

Before continuing, familiarize yourself with the code in Figure 12-2. In the paragraphs which follow, the implementation of the **circle** class will be broken down and examined.

COMPILER DIRECTIVES

At the beginning of the file are the compiler directives **#ifndef** and **#define**. These are used to prevent the class from being defined twice, which can create problems. The **#define** directive instructs the compiler to define a symbol and to remember that the symbol exists. The **#ifndef** directive checks the compiler's symbol table for a specified entry.

```
#ifndef _CIRCLE_H
#define _CIRCLE_H
```

```
#ifndef _CIRCLE_H
#define _CIRCLE_H

const float PI = 3.14159;

class circle
{
  public:
         // constructors
         circle();                        // default constructor
         circle(const circle &);          // copy constructor

         // member functions
         void   SetRadius(float);
         double Area();

  private:
         // data
         float radius;
};

// default constructor
circle::circle()
{
  radius = 0;
}

// copy constructor
circle::circle(const circle & Object)
{
  radius = Object.radius;
}

// Method to set the radius of the circle
void circle::SetRadius(float IncomingRadius)
{
  radius = IncomingRadius;
}

// Method to find the area of the circle
double circle::Area()
{
  return(PI*radius*radius);
}

#endif
```

FIGURE 12-2
The definition and implementation of the circle object. This is the `circle.h` header filer used by the *OOP.CPP* program.

In this case, these compiler directives are used together to make sure that the circle class has not already been defined. The **`#ifndef`** directive checks for the existence of a symbol named **`_CIRCLE_H`**. If the entry is not found, the **`#define`** directive defines the symbol and the source code that follows defines the **`circle`** class. If the **`_CIRCLE_H`** symbol is already defined, it means that this header file has been compiled already and the definition of the **`circle`** class is skipped.

At the end of Figure 12-2 is the compiler directive **#endif** which ends the original **#ifndef** directive. The **#ifndef** compiler directive works with the **#endif** directive to form an if structure similar to what you have worked with in previous chapters. The **#ifndef** directive instructs the compiler to compile everything between the **#ifndef** and the **#endif** if the symbol is not defined.

CLASS DEFINITION

A class definition is made up of several different parts. The definition begins with the keyword **class**, followed by the class name, and an opening brace. The definition ends with a closing brace and a semicolon. Functions and variables that are prototyped and declared in a class definition are called *members*. The syntax is similar to what you have used with structures.

```
class circle
{
  public:
         // constructors
         circle();                        // default constructor
         circle(const circle &);          // copy constructor

         // member functions
         void   SetRadius(float);
         double Area();

  private:
         // data
         float radius;
};
```

PITFALLS

If the semicolon after a class definition is omitted, the compiler will report several errors. Therefore, if multiple errors are encountered when compiling a class, check for the presence of the semicolon at the end of the class definition.

After the opening brace is the keyword **public** followed by a colon. The **public** keyword tells the compiler to let the programmer using the class have access to all the variables and functions between the **public** and **private** keywords. Any variables and functions after the **private** keyword cannot be accessed from outside the object. The **private** keyword is what allows a **circle** object to protect its data. This data protection is known as ***information hiding***, which is an important benefit provided by encapsulation. By using information hiding, objects can protect the integrity of their data.

The constructor prototypes follow the **public** keyword. ***Constructors*** tell the compiler how to create the object in memory and what the initial values of its data will be. Constructors are given the same name as the class.

```
// constructors
circle();                        // default constructor
circle(const circle &);          // copy constructor
```

For the **circle** class there are two constructors. The first constructor is known as the ***default constructor***. The default constructor is used when the object is instanciated with no arguments. The second constructor is known as the ***copy constructor***. A copy constructor receives a reference to another object as an argument, and is used when objects are passed to functions by value. You will learn more about copy constructors in the next chapter.

Extra for Experts

If no default constructor is defined, the compiler will actually create a default one for you. A class can have several different constructors that accept different arguments. Though not required, most every class should at least have a default constructor. For example, in addition to the default constructor the **circle** *class could include a constructor that allows you to pass the radius of the circle when the object is instanciated.*

After the constructors are the ***member function*** prototypes. Member functions allow programmers using an object to send information to it and receive information from it. Member functions are the messages used for communication in object-oriented programming.

```
// member functions
void   SetRadius(float);
double Area();
```

For the **circle** object, two member functions are needed, one to set the radius, and one to retrieve the area. Member function prototypes are written just like normal function prototypes with a return type, a name, and an argument list. Recall from Chapter 9 that it is not necessary for a prototype to include the names of the parameters, just the type. The implementation of the member functions follow the definition of the class.

The **private** keyword comes after the member function prototypes. The only data required for the **circle** object is the radius, so it is declared as a float with the identifier **radius**.

```
private:
          // data
          float radius;
```

Because **radius** is after the **private** keyword, a programmer using a **circle** object cannot access the radius directly, so member functions must be used. After the **radius** variable, the definition of the **circle** class is closed with the closing brace and a semicolon.

IMPLEMENTING MEMBER FUNCTIONS

To implement a member function, a special syntax must be used. The function is implemented like a normal C++ function except that the class name and the ***scope-resolution operator*** (::) precede the function name. Constructors are slightly different from other member functions because they do not have any return type—not even void.

Constructor implementation in the **circle** class is very simple. The default constructor sets the radius equal to zero. The copy constructor sets the radius equal to the passed object's radius.

```
// default constructor
circle::circle()
{
  radius = 0;
}

// copy constructor
circle::circle(const circle & Object)
{
  radius = Object.radius;
}
```

By initializing all data in an object, errors can be avoided later in the program. For example, if the integer data type were implemented as a class, the constructors could set its value to zero.

Note

The basic C++ data types (such as `int`*) are not classes. Because* `int` *is a primitive data type, it does not have constructors, destructors, or any other properties of a class.*

The implementation of the other member functions is also simple. Figure 12-3 shows a template for implementing a member function.

```
return_type class_name::function_name(parameters)
{
  // necessary code
}
```

FIGURE 12-3
The proper way to implement a member function.

The **SetRadius** function sets the radius equal to the value it is passed.

```
// Method to set the radius of the circle
void circle::SetRadius(float IncomingRadius)
{
  radius = IncomingRadius;
}
```

The **Area** function returns the area of the circle using the standard formula, Area = πr^2.

```
// Method to find the area of the circle
double circle::Area()
{
  return(PI*radius*radius);
}
```

EXERCISE 12-2 MODIFYING A CLASS

1. Load the *CIRCLE.H* header file.
2. Add a member function to the **`circle`** class which will return the radius. Remember to add the prototype to the definition and to implement the function at the end of the header file.
3. Add the necessary code to the *OOP.CPP* program to get the radius from **`Circle_One`**, then print it to the screen.
4. Save and close the *CIRCLE.H* and *OOP.CPP* files.

On the Net

Through this chapter, classes have been implemented completely in header files. Classes can also be defined in header files and actually implemented in **`cpp`** *files. This is done by including the header file at the top of the implementation source file with the* **`#include`** *directive. To learn more about dividing the source code for a class, see http://www.ProgramCPP.com. See topic 12.3.2.*

SECTION 12.3 QUESTIONS

1. Describe one guideline to follow when designing a class.
2. Why are classes normally defined in header files?
3. What is the purpose of a constructor?
4. What is the purpose of a member function?
5. What are members?
6. What are methods?

PROBLEM 12.3.1

Using the **`circle`** class definition as a model, create a file called *SQUARE.H* and implement a class which models a square and includes an Area method.

PROBLEM 12.3.2

Modify the class you implemented in Problem 12.3.1 to create a rectangle class which models a rectangle and includes an Area method.

CHAPTER 12, SECTION 4

Reusability, Containment, and Inheritance

You have seen some of the advantages of using object-oriented programming. Three more advantages of the paradigm are reusability, containment, and inheritance.

REUSABILITY

Among the greatest advantages that object-oriented programming offers is ***reusability.*** After an object is designed and coded into a class, that class can be reused in any program. This means that productivity may be increased because less code has to be written. In addition, because less code is written, fewer errors can occur. Although procedural code can be reused, object-oriented code is often easier to use, especially when using more advanced techniques of data handling. In a later chapter, you will see how the same class can be reused to handle different kinds of data without the need to change code in the class.

For example, a class which holds names and addresses could be used in a multitude of programs. An address-book program, a mail-list program, and a point-of-sale program could all use the class to keep track of people. After the class is designed and coded, the coding required for the application programs will be reduced greatly.

CONTAINMENT

One of the features of objects that make them so reusable is that objects can contain other objects and use them to implement another object. For example, suppose you want to create a class that defines a car. If you already have a wheel object that defines the properties of a wheel, your car object can use the wheel class to instanciate four wheel objects. This type of relationship among objects is called ***containment*** because the car object *contains* four wheel objects. The relationship is also referred to as a ***has-a relationship*** because a car *has a* wheel (four wheels in this case).

Note

Containment is also sometimes called **composition.** *The example of the car and the wheels would be called a compositional relationship.*

INHERITANCE

Inheritance is the ability of one object to inherit the properties of another object. For example, you might have a building class that defines the properties of a building. The building class could define attributes such as the dimensions of the building, the number of floors, and the type of materials used to construct the building. Suppose you have debugged and perfected your building class, but what you need is a house object. A house has all of the attributes of a building as well as additional attributes such as number of bedrooms, number of bathrooms, and size of garage.

Note

Note that the relationship goes in only one direction. A house is a building, but a building is not necessarily a house. The building might be a skyscraper instead.

Rather than write a new class from scratch or modify the building class, we can create a house class that inherits the properties of the building and then extends those properties with properties which describe the house. The house object and the building object have what is called an ***is-a relationship***, meaning the house *is a* building.

When one class inherits the properties of another, the class from which the properties are inherited is known as the ***parent class***. The class which inherits the properties of another is the ***child class*** or ***derived class***. In the example above, the

The is-a/has-a Rule

How do you know when to use containment or inheritance? Apply the is-a/has-a rule. If the object is a type or kind of another object, use inheritance. For example, a house is a building, so inheritance is used. If the object has another object as part of the object, use containment. For example, a house has a kitchen, so containment is used.

building class is the parent class and the **house** class is the child class. The **building** class can also be described as a ***base class*** upon which other classes are built.

An object created from the derived class can call a parent class's member functions as if they were members of the derived class. Users of the class do not need to know what members are implemented in each class. In fact, the users of the class do not need to know that the class is derived, as long as they know what members are available to them.

Extras for Experts

Recall from Section 3 that a class definition has **public** *and* **private** *sections. All members in the* **public** *section of a class can be accessed by any function outside or inside the class. Members in the* **private** *section of the class can only be accessed by member functions in the class. There is an additional section called* **protected** *that can be part of the definition. Members in the* **protected** *section can only be accessed by member functions in the class and member functions of derived classes.*

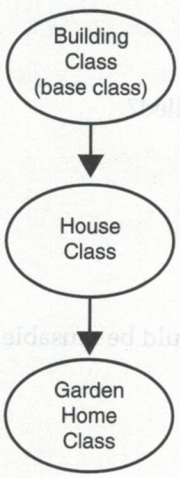

FIGURE 12-4
Inheritance can continue for multiple levels.

MULTILEVEL INHERITANCE

Inheritance can be multilevel. For example, a class called **garden_home** could inherit properties of the **house** class which inherits properties of the **building** class (see Figure 12-4). ***Multilevel inheritance*** is one of the features that makes the work done in an object-oriented program more reusable.

On the Net

Many modern user interfaces are implemented with objects which use multilevel inheritance. For example, a dialog box object might inherit properties from a window object which in turn inherits properties from a primitive frame or rectangle object. To learn more about how programs are developed using object-oriented user interfaces, see http://www.ProgramCPP.com. See topic 12.4.1.

MULTIPLE INHERITANCE

In C++, objects can also inherit properties from multiple objects. With ***multiple inheritance***, the object has two parent objects and inherits properties from both (see Figure 12-5). Multiple inheritance is rarely necessary and is not allowed in some pure OOP languages.

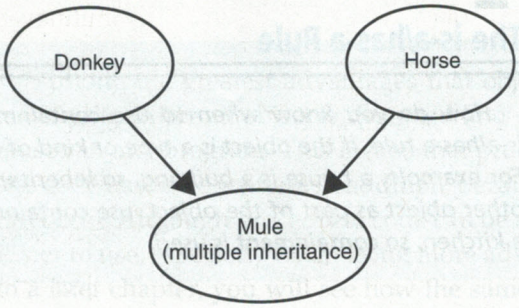

FIGURE 12-5
Multiple inheritance is allowed in C++, but should be used sparingly.

Note

Multiple inheritance is not the same thing as containment. Multiple inheritance involves inheriting the properties of more than one object. It is perfectly acceptable to create a class which includes several other objects as members.

SECTION 12.4 QUESTIONS

1. How does reusability improve productivity and reduce errors?
2. Give an example of an is-a relationship.
3. Give an example of a has-a relationship.
4. What is a class from which properties are inherited called?
5. Give an example of multilevel inheritance.

PROBLEM 12.4.1

List three reusable classes and explain why the classes would be reusable.

PROBLEM 12.4.2

List five everyday objects that contain is-a relationships and five objects that contain has-a relationships.

KEY TERMS

base class
child class
class
class-member operator
composition
constructor
containment
copy constructor
default constructor
derived class

encapsulation
has-a relationship
information hiding
inheritance
instance
instanciated
is-a relationship
member
member function
member selection operator
message
method
multilevel inheritance
multiple inheritance
object
object-oriented paradigm
object-oriented program
object-oriented programming (OOP)
paradigm
parent class
procedural paradigm
reusability
scope-resolution operator

SUMMARY

- In the procedural paradigm, the functions and algorithms are the focus, with data viewed as something for the functions to manipulate.
- The object-oriented paradigm dictates that data should be placed inside of objects and that these objects should communicate with each other in the form of messages.
- Object-oriented programming (OOP) is the process of developing programs using the object-oriented paradigm.
- The design of a class is as important as its implementation.
- Constructors allow all data encapsulated within an object to be initialized to preset values so that errors can be avoided.
- Member functions provide a way for a programmer to pass data to and get data from an object.
- Reusability is a major benefit of object-oriented programming.
- Containment is the term used to describe an object that contains one or more other objects as members.
- Inheritance is the term used to describe an object that inherits properties from another object.
- The class from which an object inherits properties is called a parent class or base class. The class that inherits the properties is called a child class or derived class.
- Inheritance can be multilevel.
- Objects can inherit properties from multiple objects. Multiple inheritance, however, should be used sparingly.

PROJECTS

PROJECT 12-1

You know about the procedural and object-oriented paradigms; research the functional and logic paradigms. Write a report describing the uses for these paradigms.

PROJECT 12-2

Select an object-oriented programming language other than C++ (such as Smalltalk or Java) and compare the way that language uses the object-oriented paradigm to C++.

PROJECT 12-3

Add a method to the circle class that returns the circumference of the circle. Modify *OOP.CPP* to report the circumference.

PROJECT 12-4

Create a class of your own design which has at least two private variables and two member functions. Explain why your class is well designed and has only the required functions.

PROJECT 12-5

Give the source code file for the class you created in Project 12-4 to another student as if he or she is a programmer who wants to use the class. Give the programmer a list of your public member functions and see if your class can be successfully used without opening your source code file to see how it works.