Object-Oriented Programming Using C++

Fourth Edition

Joyce Farrell

Australia • Brazil • Japan • Korea • Mexico • Singapore • Spain • United Kingdom • United States

COURSE TECHNOLOGY
CENGAGE Learning™

Object-Oriented Programming Using C++, Fourth Edition
Joyce Farrell

Executive Editor: Marie Lee
Acquisitions Editor: Amy Jollymore
Managing Editor: Tricia Coia
Developmental Editor: Lisa Ruffolo
Editorial Assistant: Patrick Frank
Marketing Manager: Bryant Chrzan
Content Project Manager: Erin Dowler
Art Director: Bruce Bond
Manufacturing Coordinator: Julio Esperas
Proofreader: Wendy Benedetto
Cover Designer: Bruce Bond
Cover Photo: © iStockphoto.com/ Storman
Compositor: International Typesetting and Composition

ISBN-13: 978-1-4239-0257-7

ISBN-10: 1-4239-0257-2

Course Technology
25 Thomson Place
Boston, MA 02210
USA

Cengage Learning is a leading provider of customized learning solutions with office locations around the globe, including Singapore, the United Kingdom, Australia, Mexico, Brazil, and Japan. Locate your local office at: **international.cengage.com/region**

Cengage Learning products are represented in Canada by Nelson Education, Ltd.

For your lifelong learning solutions, visit **course.cengage.com**

Purchase any of our products at your local college store or at our preferred online store **www.ichapters.com**

Printed in the United States of America
1 2 3 4 5 6 7 12 11 10 09 08

CHAPTER SEVEN

7

USING CLASSES

In this chapter, you will:

- Create classes
- Learn about encapsulating class components
- Implement functions in a class
- Understand the unusual use of private functions and public data
- Consider scope when defining member functions
- Use static class members
- Learn about the `this` pointer
- Understand the advantages of polymorphism

Classes are similar to structures in that both provide a means to group data and behaviors together so you can create objects. So far in this book, you have worked with structures containing `public` data fields. However, structures and classes can contain much more than `public` data fields; they can contain methods that make their objects operate like many concrete objects in the physical world. That is, objects can *do* things as well as *contain* things, and some of their attributes and operations can be hidden from users. By default, `struct` data fields are `public`; that is why you have been using structures instead of classes up to this point in this book—structures are easier to work with. To create objects, most C++ programmers prefer to use classes most of the time, partly because, by default, their data fields are `private`, or hidden, and cannot be altered without the programmer's permission. You needed to learn how to handle functions, including passing parameters to them, before you could learn to use classes in meaningful and conventional ways. This chapter will show you how to add both hidden and unhidden fields and functions to your classes, and how C++ handles the functions behind the scenes. This chapter will take you into the world of modern object-oriented programming.

CREATING CLASSES

A **class** is a category of objects; it is a new data type you create that is more complex than the basic data types. Classes provide a description of an object—they detail all the data an object has and all the actions the object can perform. Additionally, classes provide a convenient way to group related data and the functions that use the data. One advantage of creating a class is that when you create an object from the class, you automatically create all the related fields. Another advantage is that you gain the ability to pass an object into a function, or receive an object from a function as a returned value, and automatically pass or receive all the individual fields that each object contains. Another advantage to using classes and objects is that you think about them and manipulate them similarly to the way you use real-life classes and objects. For example, you are familiar with the class of objects known as automobiles; you understand what features an automobile has (an engine size, a number of doors, a color) and you understand what an automobile can do (go forward, idle, go in reverse). When you learn that a particular object is an automobile, you know a lot about the object before you even see it. If a friend tells you he is buying an automobile, you know approximately how it looks and works even though you don't know the values of his particular object yet—for example its engine size, color, or number of doors. Programming classes and objects work similarly: when you define a class you describe what it has and can do; when you create specific objects you supply specific values.

NOTE Creating a class is similar to creating a recipe for a cake or a blueprint for a house. It is a plan for which there might or might not eventually be objects.

NOTE Most C++ compilers contain many built-in classes that are ready for you to use. For example, in Chapter 5 you were introduced to the built-in C++ `string` class. Using this class makes it easier for you to handle strings in your programs.

Consider the `Student` class in Figure 7-1. It contains three data fields: `idNum`, `lastName`, and `gradePointAverage`. C++ programmers would say that the `Student` class shown in the figure is an **abstract data type (ADT)**. This term simply indicates that `Student` is a type you

define, as opposed to types like `char` and `int` that are defined by C++. You define a class by using the keyword `class`, the class identifier, and a pair of curly braces. You must include a semicolon following the closing curly brace. Between the curly braces, you define all the data and behaviors that are members of the class.

```
class Student
{
   int idNum;
   string lastName;
   double gradePointAverage;
};
```

Figure 7-1 A `Student` class

NOTE The `Student` class in Figure 7-1 contains data fields, but no behaviors or functions. You will learn how to add these later in the chapter.

NOTE A language to which you can add your own data types, such as C++, is an **extensible language**.

NOTE You cannot assign a value to a field in a class definition. For example, in the `Student` class, you cannot write `idNum = 123;`. A class definition is only a description of a type; you cannot assign values to fields until you create an object.

When you declare a `Student` object, you use the `Student` name just as you use scalar type names. For example, you can declare the following:

```
Student aSophomore;
```

NOTE Recall from Chapter 1 that an object is any thing, and that a class is a category of things. An object is a specific item that belongs to a class; it is called an instance of a class. A class defines the characteristics of its objects and the methods that can be applied to its objects.

NOTE Conventionally, class names begin with an uppercase letter and object names begin with a lowercase letter.

When you refer to `aSophomore`, you are including all the separate attributes that constitute `aSophomore`—an ID number, a name, and a grade point average. However, unlike with a `struct`, you cannot refer to the specific attributes of the object `aSophomore` by using the object's name, a dot, and the attribute's name. For example, to display the ID number of a declared `Student` named `aSophomore`, you might suppose that you could write statements like the following, but you cannot:

```
aSophomore.idNum = 7645;
cout << aSophomore.idNum;
```

If you attempt to compile a program containing a statement that accesses an object's field such as `aSophomore.idNum`, you receive a message similar to "Cannot access private class member declared in class `Student`". You receive the error message because, by default, all members of a class are **`private`**, meaning they cannot be accessed using any statements in any functions that are not also part of the class. In other words, private class members cannot be displayed, assigned a value, manipulated arithmetically, or used directly in any other way.

It is possible to declare class data members to be `public` instead of `private`. For example, Figure 7-2 shows a `Student` class in which the keyword `public` and a colon (shaded) have been inserted prior to the declaration of the fields. In Figure 7-2, the shaded word `public` is an **access modifier**; it assigns accessibility to the declared variables that follow it. Using the keyword `public` means that the fields are now accessible when they are used with a `Student` object in a `main()` function, just as they would be with a `Student struct`. However, although you could avoid error messages when accessing one of the `Student` fields by using this technique, it would be highly unconventional to do so. Additionally, you would violate the object-oriented principle of encapsulation. Fields within classes are `private` by default, because that is what they usually *should* be in object-oriented programs.

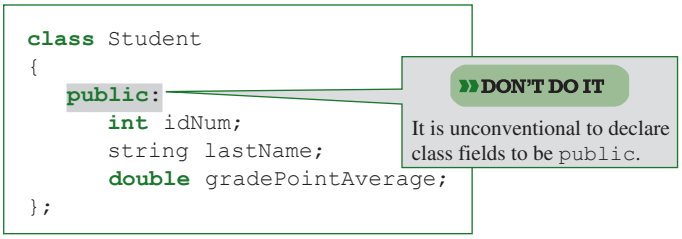

Figure 7-2 An unconventional (and not recommended) version of a `Student` class containing public data fields

»NOTE In the class in Figure 7-2, you can use the `private` access modifier and a colon in place of the shaded `public` access modifier to clearly identify the fields as private in your program. However, if you use no modifier, the fields are still private by default.

T T F

»TWO TRUTHS AND A LIE: CREATING CLASSES

1. A class is a complex data type.
2. You define a class by using the keyword `class`, the class identifier, and a pair of curly braces; you must include a semicolon following the closing curly brace.
3. By default, all members of a class are public, meaning they can be accessed by outside functions.

The false statement is #3. By default, all members of a class are private, meaning they cannot be accessed using any statements in any functions that are not also part of the class.

ENCAPSULATING CLASS COMPONENTS

The first step to creating a class involves determining the attributes of an object, and subsequently dealing with the object as a whole. That's what you are doing when you decide on the field types and names for a class like the `Student` class. When you work with a `Student` object, for example, and pass it to a function, you deal with the object as a whole, and do not

have to think about its internal fields. You think about the fields only when necessary, such as when you need to assign a value to a `Student`'s `idNum` or when you want to display it. A hallmark of object-oriented programming is that you think about program objects in the same way you think about their real-world counterparts.

For example, when you use a real-world object such as a radio, you usually don't think or care about how it works. Sometimes, however, you change some of the states of the radio's attributes, such as volume level or frequency selection. Just as the internal components of a radio are hidden, when you create a class name for a group of associated variables, you contain, or **encapsulate**, the individual components. Sometimes you want to change the state or value of some components, but often you want to think about the entity as a whole and not concern yourself with the details. Programmers sometimes refer to encapsulation as an example of using a "black box." A black box is a device that you can use, but cannot look inside to see how it works. When you use a class, you are using a group name without being concerned with the individual components; the details are hidden.

NOTE
You first learned the phrase "black box" in Chapter 6 when you learned about functions. Like functions, classes hide their internal components from the outside.

When you work with concrete, real-world objects, you think about more than what components they contain or what states they possess; you also think about what the objects can do. For example, a radio's states or values include its present volume and frequency selection, but a radio also possesses a means for *changing* these states. In other words, radios have methods as well as attributes. When you change the volume or frequency, you use an **interface**, such as a dial or button. The interface intercedes between you and the more complicated inner workings of the radio. When you design C++ classes, you should think about what the instantiations of the class (the objects) will *do* and how programmers will make them do it, as well as what the objects *contain*. Therefore your classes will contain fields, and functions that often act as interfaces to the objects. Typically, the fields in your classes will be private, but your interfacing functions will be public.

DESIGNING CLASSES

You can think of the built-in scalar types of C++ as classes. You do not have to define those classes; the creators of C++ have already done so. For example, when the `int` type was first created, the programmers who designed it had to think of the following:

Q: What shall we call it?

A: `int`.

Q: What are its attributes?

A: An `int` is stored in four bytes (or possibly some other number, depending on your system); it holds whole-number values.

Q: What methods are needed by `int`?

A: A method to assign a value to a variable (for example, `num = 32;`).

Q: Any other methods?

A: Some operators to perform arithmetic with variables (for example, `num + 6;`).

Q: Any other methods?

A: Of course, there are even more attributes and methods of an `int`, but these are a good start.

Your job in constructing a new class is similar. If you need a class for students, you should ask:

Q: What shall we call it?
A: `Student`.
Q: What are its attributes?
A: It has an integer ID number, a string last name, and a double grade point average.
Q: What methods are needed by `Student`?
A: A method to assign values to a member of this class (for example, one `Student`'s ID number is 3232, her last name is "Walters", and her grade point average is 3.45).
Q: Any other methods?
A: A method to display data in a member of this class (for example, display one `Student`'s data).
Q: Any other methods?
A: Probably, but this is enough to get started.

For most object-oriented classes, then, you declare both fields and functions.

- You declare a field using a data type and an identifier.
- You declare a function by writing its prototype, which serves as the interface to the function.

For example, you might want to create a method for a `Student` class that displays all the details of a `Student`'s data. You could create a function with the prototype `void displayStudentData();` and include it in the `Student` class definition shown in Figure 7-3. In this figure, the three data fields are `private` by default, because no access modifier precedes them. However, the shaded `public` access modifier precedes the `displayStudentData()` function declaration, making the function `public`.

```
class Student
{
    private:
       int idNum;
       string lastName;
       double gradePointAverage;
    public:
       void displayStudentData();
};
```

Figure 7-3 `Student` class that includes one function definition

When you declare an object, you create an instance of the class; you **instantiate** the object. When you declare a class with a function definition such as the one shown in Figure 7-3, and you then instantiate an object of that class, the object possesses more than three

fields—it also possesses access to a function. If you declare a `Student` object as `Student aSophomore;`, then you can use the `displayStudentData()` function by referring to `aSophomore.displayStudentData()`. Similarly, another `Student` object, `Student aGraduate;`, could use `aGraduate.displayStudentData()`. Although a `main()` function (or any other function outside the class) cannot use any of the fields defined in the class, the `displayStudentData()` function can use them because it belongs to the same class. Other functions, for example a `main()` function, can't use the private data fields directly, but they *are* allowed to use the `displayStudentData()` function because it is `public`, not `private`.

At this point, you might wonder, if you are going to create a public function that accesses private data, why not just make the data public in the first place, avoiding the function? You create the public function so you, the class creator, can control how the data items are used. For example, if your radio provides an interface that allows you to set the volume only as high as 10, then you cannot set it to 11 even though the internal components of your radio might be able to produce that level of sound. Similarly, if you allow access to only public methods for a function that is a user of your class (often called a **class client**), then you control how the user can manipulate the data. For example, if you never want a class client to be able to see the last digit of a `Student` object's ID number, then you simply do not display that digit from within the `displayStudentData()` function. The client must use your public interface to see any data values; the client cannot just code `cout << aSophomore.idNum;`.

T T F

»TWO TRUTHS AND A LIE: ENCAPSULATING CLASS COMPONENTS

1. When you create a class name for a group of associated variables, you contain, or encapsulate, the individual components.
2. Most classes contain private fields and public functions that often act as interfaces to the fields.
3. When you design and create a class, you instantiate it.

The false statement is #3. You instantiate an object when you declare it; an object is a class instance.

IMPLEMENTING FUNCTIONS IN A CLASS

After you create a function's prototype in a class, you still must write the actual function. When you construct a class, you create two parts. The first part is a **declaration section**, which contains the class name, variables (attributes), and function prototypes. The second part created is an **implementation section**, which contains the functions themselves. For example, Figure 7-4 shows the `Student` class that includes a shaded function implementation for `displayStudentData()`.

```
// declaration section:
class Student
{
   private:
      int idNum;
      string lastName;
      double gradePointAverage;
   public:
      void displayStudentData();
};
// implementation section:
void Student::displayStudentData()
{
   cout << "Student #" << idNum << "'s last name is " <<
      lastName << endl;
  cout << "The grade point average for this student is " <<
      gradePointAverage << endl;
}
```

Figure 7-4 `Student` class that includes one function definition and implementation

In the class definition shown in Figure 7-4, the declaration section includes a function prototype, and the implementation section contains the actual function. The `displayStudentData()` function header is preceded by the class name, `Student`, and the scope resolution operator (::). You must use both the class name and the scope resolution operator when you implement a member function, because they tie the function to this class and allow every instantiated object to use the function name. The statements within the function are similar to those in any other function; in this case, they just display the object's data fields.

NOTE Instead of prototyping a function in the declaration section of a class and implementing that function later, you can implement it in place of the prototype. This causes the function to become an inline function. Usually, however, you should keep the function declaration and implementation separate, as in the example in Figure 7-4.

NOTE You can refer to the scope resolution operator used in a member function header as a binary scope resolution operator, because it requires two operands: the class name to the left and the function name to the right. In Chapter 6, you learned to use the unary scope resolution operator to access a global variable from within a function that contains a local variable with the same identifier.

USING PUBLIC FUNCTIONS TO ALTER PRIVATE DATA

You gain a major advantage when you make a data field private. You might spend a great deal of time creating a class, including writing and debugging all of its member functions. Your carefully crafted methods could be worthless if outside functions over which you have no control could modify or erroneously use the member data of any object in the class. Making data fields private prevents outside manipulation of those fields. When you create and test a class, and store its definition in a file, programs that use the definition cannot use private member data incorrectly. If a

private member of your `Student` class, such as `idNum`, must be a four-digit number, or if you require that the `idNum` always be preceded by a pound sign when it is displayed, functions that are not a member of your class can never change those rules (either intentionally or by accident).

NOTE You can choose to make data public if you have a good reason. Frequently, `public` data is also constant, so it can be used, but not altered. For example, a `MathConstants` class might have `public` data members that hold mathematical constants, such as `PI`. Many object-oriented software developers are opposed to making any data `public`, arguing that data should always be `private` and accessed through a `public` interface.

However, if a program can't assign a value to a field because it is private, then the field is not of much use. You keep data private, yet gain the ability to alter it, by creating additional public functions that *can* assign values to a class's private fields. For example, if the offices at your firm are all private and you cannot communicate with the executives in them, the company won't function for long. Usually the receptionists are public, and you can use them to communicate with the private executives. Similarly, you communicate with the private members of a class by sending messages to the public member functions.

NOTE Classes can contain many functions with many purposes. Usually the first functions you create for a class are those that provide a means for input and output of the class data fields.

Figure 7-5 shows the declaration for a `Student` class that contains three additional functions used to assign values to a `Student`'s data fields. These functions are named `setIdNum()`, `setLastName()`, and `setGradePointAverage()`. Each is used to set the value of one of the fields within the `Student` class.

```
class Student
{
   private:
      int idNum;
      string lastName;
      double gradePointAverage;
   public:
      void displayStudentData();
      void setIdNum(int);
      void setLastName(string);
      void setGradePointAverage(double);
};
```

Figure 7-5 `Student` class with set functions for private data

You can place any lines of code you want within these functions when you implement them. If you want the `setLastName()` function to assign a string to the `Student`'s `lastName`, then you can implement the function as shown in Figure 7-6.

```
void Student::setLastName(string name)
{
   lastName = name;
}
```

Figure 7-6 The `setLastName()` function

The `setLastName()` function shown in Figure 7-6 is a member of the `Student` class. You can determine this because the class header contains the `Student` class name and the scope resolution operator. The `setLastName()` function takes a string parameter that has the local identifier `name`. The function assigns the passed name value to the `lastName` field of `Student` object. When you write a program in which you instantiate a `Student` object named `aJunior`, you can assign a last name to the `aJunior` object with a statement such as the following:

```
aJunior.setLastName("Farnsworth");
```

The string "Farnsworth" is passed into the `aJunior` object's `setLastName()` function, where it is copied to the `aJunior` object's `lastName` field.

If `lastName` were a public field within the `Student` class, then you could assign "Farnsworth" to `lastName` with the statement `aJunior.lastName = "Farnsworth";`. That is, you would not have to use the `setLastName()` method. However, making the `lastName` field public would violate one of the canons of object-oriented programming: Whenever possible, data should be kept private, and access to data should be controlled by public functions.

Assume that a student ID number should not be more than four digits in length or else you want to define a default value to the field. When you implement the `setIdNum()` function of the `Student` class, you can assign the ID number argument to the class `idNum` field only if it is a valid ID; otherwise you can force the `idNum` field to 9999. Figure 7-7 shows a `setIdNum()` function that operates in this way.

```
void Student::setIdNum(int num)
{
   const int MAX_NUM = 9999;
   if(num <= MAX_NUM)
      idNum = num;
   else
      idNum = MAX_NUM;
}
```

Figure 7-7 The `Student` class `setIdNum()` function

NOTE
The `setIdNum()` function in Figure 7-7 allows a negative ID number. You could add more code to further limit the allowed `idNum` values.

When you use the `setIdNum()` function with an object like `aJunior`, you are assured that `aJunior` receives a valid `idNum` according to the rules you have defined. If the `idNum` field were public, a client could write a program and include a statement that assigned an invalid ID, for example `aJunior.idNum = 123456;`. However, when the `idNum` is private, you must use the public `setIdNum()` function. If you try to pass in a value with too many digits, such as with the statement `aJunior.setIdNum(123456);` the result is that the `aJunior`'s `idNum` is set to 9999, not 123456.

Figure 7-8 shows the entire `Student` class, including the implementation of the `setGradePointAverage()` function that accepts a `double` argument, and assures that the argument value is no more than 4.0 before assigning it to any `Student`'s `gradePointAverage`

field. Figure 7-8 also includes a short demonstration `main()` function that assigns values for two `Student` objects. (An interactive program would prompt for the values instead.) Figure 7-9 shows the output of the program.

```
#include<iostream>
#include<string>
using namespace std;
// declaration section
class Student
{
   private:
      int idNum;
      string lastName;
      double gradePointAverage;
   public:
      void displayStudentData();
      void setIdNum(int);
      void setLastName(string);
      void setGradePointAverage(double);
};
// implementation section
void Student::displayStudentData()
{
   cout << "Student #" << idNum << "'s last name is " <<
      lastName << endl;
   cout << "The grade point average for this student is " <<
      gradePointAverage << endl;
}
void Student::setIdNum(int num)
{
   const int MAX_NUM = 9999;
   if(num <= MAX_NUM)
      idNum = num;
   else
      idNum = MAX_NUM;
}
void Student::setLastName(string name)
{
   lastName = name;
}
void Student::setGradePointAverage(double gpa)
{
   const double MAX_GPA = 4.0;
   if(gpa <= MAX_GPA)
      gradePointAverage = gpa;
   else
      gradePointAverage = 0;
}
```

Figure 7-8 The `Student` class and a demonstration `main()` function (*continued*)

```
int main()
{
   Student aStudent;
   Student anotherStudent;
   aStudent.setLastName("Smith");
   aStudent.setIdNum(3456);
   aStudent.setGradePointAverage(3.5);
   aStudent.displayStudentData();
   anotherStudent.setLastName("Wesson");
   anotherStudent.setIdNum(34567);
   anotherStudent.setGradePointAverage(4.5);
   anotherStudent.displayStudentData();
   return 0;
}
```

Figure 7-8 (*continued*)

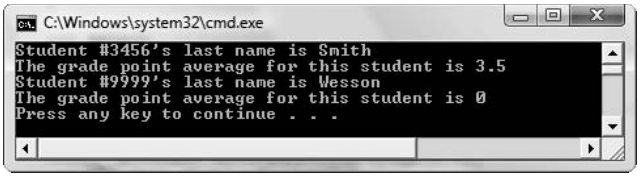

Figure 7-9 Output of the program in Figure 7-8

NOTE Many C++ programmers prefer to place the public section before the private section in a class definition. The reasoning is that other programmers then see and use the public interfaces. Use the style you prefer.

NOTE You can use the access specifiers `public` and `private` as many times as you want in a class definition—for example, two private members, followed by one public member, followed by two more private members. It's better style, however, to group the private and public members together.

In the `Student` class functions in Figure 7-8, notice that the fields `idNum`, `lastName`, and `gradePointAverage` never need to be passed into the functions as arguments. The functions have direct access to these fields because both the fields and functions belong to the same class. Similarly, the functions that provide values for the fields never need to return values anywhere; as members of the same class, the functions have direct access to the fields in the class.

T T F

TWO TRUTHS AND A LIE: IMPLEMENTING FUNCTIONS IN A CLASS

1. Fields and function prototypes are stored in a class's implementation section and a class's function bodies are stored in its declaration section.
2. When you implement a function in a class, you precede the function name with the class name and the scope resolution operator (::).
3. You usually keep data private, yet gain the ability to alter it, by creating public functions that assign values to a class's private fields.

The false statement is #1. Fields and function prototypes are stored in a class's declaration section and a class's function bodies are stored in its implementation section.

UNUSUAL USE: USING PRIVATE FUNCTIONS AND PUBLIC DATA

When you create a class, usually you want to make data items private, to control how they are used, and to make functions public to provide a means to access and manipulate the data. However, if you have a reason to do so, you are free to make particular data items public. Similarly, not all functions are public.

When you think of real-world objects, such as kitchen appliances, there are many functions you control through a public interface: adjusting the temperature on a refrigerator or oven, setting a cycle on a dishwasher, and so on. However, there are other functions that appliances encapsulate: a freezer might defrost itself without your help, and a dishwasher automatically switches from the wash to the rinse cycle. With objects you create, functions also can be private if you choose.

For example, Figure 7-10 shows a `Carpet` class that contains data fields that store a `Carpet`'s length, width, and price. Assume that a `Carpet`'s price can be one of three values based on the `Carpet`'s area. The `Carpet` class in Figure 7-10 contains public functions that set the length and width, but because the price is determined by the length and width, it only has one public function that retrieves its value. The value of price is set by a private function that is called any time the length or width changes. The shaded `setPrice()` function is not intended to be used by a client function, such as the `main()` function at the bottom of the figure. Instead, `setPrice()` is used only by the two functions that set carpet dimensions. It makes sense that `setPrice()` should be private because you would not want a client program to change a `Carpet`'s price to one that violated the carpet pricing rules. Figure 7-11 shows a typical execution of the `main()` function in which the user continues to enter various `Carpet` dimensions to observe the changes in price.

```
#include<iostream>
using namespace std;
// declaration section
class Carpet
{
   private:
      int length;
      int width;
      double price;
      void setPrice();
   public:
      int getLength();
      int getWidth();
      double getPrice();
      void setLength(int);
      void setWidth(int);
};
```

Figure 7-10 `Carpet` class and `main()` demonstration function (*continued*)

```
// implementation section
int Carpet::getLength()
{
   return length;
}
int Carpet::getWidth()
{
   return width;
}
double Carpet::getPrice()
{
   return price;
}
void Carpet::setLength(int len)
{
   length = len;
   setPrice();
}
void Carpet::setWidth(int wid)
{
   width = wid;
   setPrice();
}
void Carpet::setPrice()
{
   const int SMALL = 12;
   const int MED = 24;
   const double PRICE1 = 29.99;
   const double PRICE2 = 59.99;
   const double PRICE3 = 89.99;
   int area = length * width;
   if(area <= SMALL)
      price = PRICE1;
   else
      if(area <= MED)
         price = PRICE2;
      else
         price = PRICE3;
}
```

Figure 7-10 (*continued*)

```
int main()
{
   Carpet aRug;
   const char QUIT = 'Q';
   char dim;
   int length;
   int width;
   aRug.setLength(1);
   aRug.setWidth(1);
   cout << "Enter L to enter length or " <<
      "W to enter width or " <<
      QUIT << " to quit > ";
   cin >> dim;
   while(dim != QUIT)
   {
      if(dim == 'L')
      {
         cout << "Enter a length > ";
         cin >> length;
         aRug.setLength(length);
         cout << "Length is " << aRug.getLength() <<
            "  Width is " << aRug.getWidth() << endl <<
            "Price is " << aRug.getPrice() << endl;
      }
         else
         {
            cout << "Enter a width > ";
            cin >> width;
            aRug.setWidth(width);
            cout << "Length is " << aRug.getLength() <<
               "  Width is " << aRug.getWidth() << endl <<
               "Price is " << aRug.getPrice() << endl;
         }
         cout << "Enter L to enter length or W " <<
            "to enter width or " <<
            QUIT << " to quit > ";
         cin >> dim;
    }
    return 0;
}
```

Figure 7-10 (*continued*)

Although it is somewhat unusual to make functions private, it is more unusual to make data public. However, C++ allows you to make data public. For example, the `Carpet` class might contain a public data member that holds the value of PI so it can be used in calculating the area of round carpets. The advantage to this approach is that you avoid having to write a get function to retrieve the data. Almost always, when there is a good reason for data to be public, it also is constant and static. You will learn more about static class members later in this chapter.

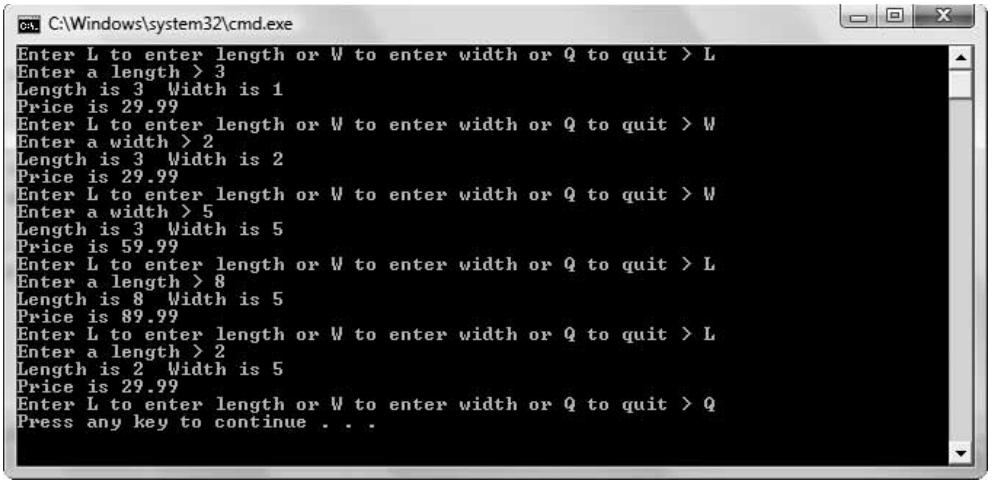

Figure 7-11 Typical execution of the application in Figure 7-10

T T F

»TWO TRUTHS AND A LIE: UNUSUAL USE: USING PRIVATE FUNCTIONS AND PUBLIC DATA

1. When you create a class, usually you want to make data items public so that client programs can use their data.
2. When you create a class, usually you want to make functions public to provide a means to access and manipulate the private data.
3. Although it is unusual, you can make functions private.

The false statement is #1. When you create a class, usually you want to make data items private, to control how they are used.

CONSIDERING SCOPE WHEN DEFINING MEMBER FUNCTIONS

You already know about scope; a local variable is in scope only within the function in which it is declared. The scope resolution operator is a C++ operator that identifies a member function as being in scope within a class. It consists of two colons (::).

For example, the `Customer` class in Figure 7-12 contains two fields (one for the name and another to hold the balance due) and four functions (two that set the field values and two that display them). The headers for the functions signal that they are members of the `Customer` class by using the class name and scope resolution operator in front of the function identifiers.

```
#include<iostream>
#include<string>
using namespace std;
// declaration section
class Customer
{
   private:
      string name;
      double balance;
   public:
      void setName(string);
      void setBalance(double);
      string getName();
      double getBalance();
};
// implementation section
void Customer::setName(string custName)
{
   name = custName;
}
void Customer::setBalance(double amount)
{
   balance = amount;
}
string Customer::getName()
{
   return name;
}
double Customer::getBalance()
{
   return balance;
}
```

Figure 7-12 The `Customer` class

Within a function that is part of a class, when you use field names, you do not need to take any special action; the fields are part of the same class as the function, so the function has access to them. For example, in the `setBalance()` function in Figure 7-12, `balance` refers to the field with that identifier. However, the `setBalance()` function can also be written as shown in Figure 7-13. In this version, the `Customer` class name and scope resolution operator are used with the field name.

```
void Customer::setBalance(double amount)
{
      Customer::balance = amount;
}
```

Figure 7-13 The `setBalance()` function of the `Customer` class, using scope resolution with the field name

Within the `setBalance()` function in the `Customer` class, the class name and scope resolution operator are completely optional when using the field name; the function operates correctly without them because the `balance` field is part of the same class as the function itself. However, the scope resolution operator does no harm, and leaves no doubt as to the origin of the field names; it is equivalent to using your full formal name instead of your nickname. Just as a mother seldom uses the name "Catherine Marie Johnson," opting instead for "Cathy," C++ programmers usually do not bother using the class name and scope resolution operator when referring to field names from functions within the same class.

However, there are circumstances when the scope resolution operator might be required with a field name. Whenever there is a naming conflict between a local variable and a field name, you can use the scope resolution operator to distinguish the field from the local variable. Consider the version of the `Customer` class `setBalance()` function in Figure 7-14. In this version, the parameter passed into the function that holds the `Customer`'s balance is named `balance` (see shading). Within this version of the function, any use of `balance` refers to the passed parameter. To distinguish between the local variable and the object's field, the scope resolution operator is used with the field that belongs to each object. If you placed a statement within the `setBalance()` function, such as `balance = balance;`, you simply would assign the passed parameter's value to itself, and the object's field would never be set. A class's field with the same identifier as a parameter or other local variable is hidden if you do not use the scope resolution operator.

»NOTE
Later in this chapter, you will learn how to use a `this` pointer instead of the class name and scope resolution operator when a local variable and a field name conflict.

```
void Customer::setBalance(double balance)
{
      Customer::balance = balance;
}
```

Figure 7-14 A version of `setBalance()` in which the scope resolution operator is required

T T F

»TWO TRUTHS AND A LIE: CONSIDERING SCOPE WHEN DEFINING MEMBER FUNCTIONS

1. Whenever you use a field name within a function that is part of a class, you have the option of using the class name and the scope resolution operator with the field name.
2. Whenever there is a naming conflict between a local variable and a field name, you can use the scope resolution operator with the local variable to distinguish between the two.
3. A class's field with the same identifier as a parameter is hidden if you do not use the scope resolution operator.

The false statement is #2. Whenever there is a naming conflict between a local variable and a field name, you can use the scope resolution operator with the class's field to distinguish between the two.

USING STATIC CLASS MEMBERS

A C++ object is an instantiation of a class that can contain both data members and methods. When you create an object, a block of memory is set aside for the data members. Just as the declaration `int x;` reserves enough space in your system to hold an integer, the declaration `Student oneStudent;` reserves enough storage in your system to hold a `Student` object. For example, if a `Student` is defined to contain an integer ID number and a `double` grade point average, then each time a `Student` object is declared, enough memory for both the integer and the `double` is reserved. This makes sense, because every `Student` object requires its own ID and grade point average. Fields such as `idNumber` and `gradePointAverage` are **instance fields** or **instance variables** because you store a separate copy of each instance of the class.

When objects are instantiated, each one gets its own block of memory for its data members. If you create two `Student`s, you reserve two blocks of memory; if you create an array of 100 objects, 100 blocks of memory are set aside.

Sometimes every instantiation of a class requires the same value. For example, you might want every `Student` object to contain a data member that holds the student athletic fee—a value that is the same for all `Student`s. If you declare 100 `Student` objects, all `Student`s need their own ID and grade point average, but not all `Student`s need their own copy of the athletic fee figure. If each `Student` object contains a copy of the athletic fee, you repeat the same information 100 times, wasting memory. To avoid this, you can declare the athletic fee variable as **static**, meaning that only one memory location is allocated, no matter how many objects of the class you instantiate. In other words, all members of the class share a single storage location for a `static` data member of that same class. A class variable that you declare to be `static` is the same for all objects that are instantiations of the class. Each instantiation of the class *appears* to be storing its own copy of the same value, but each actually just has access to the same memory location. The fields, such as the ID number, that are separate memory locations for each instance are **non-static**.

NOTE When you create a non-static variable within a function, a new variable is created every time you call that function. The variable might have a different address and different initial value from the previous time you called the function. When you create a static variable, the variable maintains its memory address and previous value for the life of the program.

DEFINING STATIC DATA MEMBERS

Because it uses only one memory location, a `static` data member is defined (given a value) in a single statement outside the class definition. Most often this statement appears just before the class implementation section. Consider the class and program in Figure 7-15. Note that in this example, the small `Student` class contains just two fields. In this program, every object of the class that you ever instantiate receives its own copy of the non-static `idNum` field, but the same static `athleticFee` value applies to every class member that is ever instantiated. (The first shaded statement in Figure 7-15 shows the declaration of the field.) When the `main()` function declares two `Student` objects, `aFreshman` and `aSophomore`, they each possess their own `idNum` value, but they share the `athleticFee` value. In the output shown in Figure 7-16, each object displays the same athletic fee.

```
#include<iostream>
using namespace std;
// declaration section:
class Student
{
   private:
      int idNum;
      static double athleticFee;
   public:
      void setIdNum(int);
      int getIdNum();
      double getAthleticFee();
};
// implementation section:
double Student::athleticFee = 45.25;
void Student::setIdNum(int num)
{
   idNum = num;
}
int Student::getIdNum()
{
   return idNum;
}
double Student::getAthleticFee()
{
   return athleticFee;
}
int main()
{
   Student aFreshman, aSophomore;
   aFreshman.setIdNum(1234);
   aSophomore.setIdNum(2345);
   cout << "Freshman: " << aFreshman.getIdNum() <<
      " Athletic fee: " << aFreshman.getAthleticFee() <<
      endl;
   cout << "Sophomore: " << aSophomore.getIdNum() <<
      " Athletic fee: " << aSophomore.getAthleticFee() <<
      endl;
   return 0;
}
```

Figure 7-15 A class that contains a static `athleticFee` field and a program that demonstrates its use

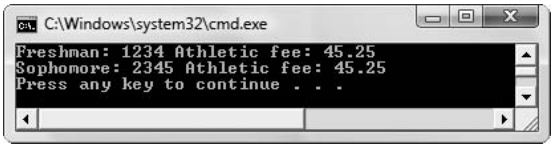

Figure 7-16 Output of application in Figure 7-15

Even though each `Student` declared in the program shown in Figure 7-15 has a unique ID number, all `Student` objects share the athletic fee, which is assigned a value just once (in the second shaded statement in the figure). You must assign a value to each `static` field outside the class in which it is declared. (You can also change the `static` field value later, if it is not a constant field.) In Figure 7-15, notice that the keyword `static` is used when the `athleticFee` is declared. The keyword `static` is not used in the assignment statement in which `athleticFee` is assigned a value.

You cannot use the `Student` class `idNum` field unless you have created a `Student` object; no `idNum` exists unless a `Student` exists. However, a `static` class member exists, even when you have not instantiated any objects of the class. However, for a non-class member function (such as `main()`) to use the static field directly, it cannot be private. Figure 7-17 shows a revised `Student` class in which the static field is made public. Now the `main()` function can access the fee without using an object. In the last shaded statement in Figure 7-17, you can see that you can use `athleticFee` with just the class name and the scope resolution operator. Static variables are sometimes called **class variables**, **class fields**, or **class-wide fields** because they don't belong to a specific object; they belong to the class, and you can use them even if you never instantiate an object. Figure 7-18 shows the program execution.

```
#include<iostream>
using namespace std;
// declaration section:
class Student
{
   private:
      int idNum;
   public:
      static double athleticFee;
      void setIdNum(int);
      int getIdNum();
      double getAthleticFee();
};
// implementation section:
double Student::athleticFee = 45.25;
void Student::setIdNum(int num)
{
   idNum = num;
}
int Student::getIdNum()
{
   return idNum;
}
```

Figure 7-17 A class that contains a static `athleticFee` field and a program that demonstrates its use (*continued*)

```
double Student::getAthleticFee()
{
   return athleticFee;
}
int main()
{
  Student aFreshman, aSophomore;
  aFreshman.setIdNum(1234);
  aSophomore.setIdNum(2345);
  cout << "Freshman: " << aFreshman.getIdNum() <<
     " Athletic fee: " << aFreshman.getAthleticFee() << endl;
   cout << "Sophomore: " << aSophomore.getIdNum() <<
      " Athletic fee: " << aSophomore.getAthleticFee() <<
      endl;
   cout << "The athletic fee for all students is " <<
     Student::athleticFee << endl;
    return 0;
}
```

Figure 7-17 (*continued*)

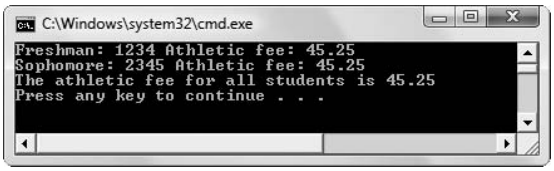

Figure 7-18 Output of application in Figure 7-17

The class in Figure 7-17 violates one of the principles of object-oriented programming in that usually, you want a class's data to be unalterable by outside functions. In the version of the `Student` class in Figure 7-17, the athletic fee is public. That allows you to use the fee without an object, but it also means that any program could change the value of the fee. Therefore, static fields frequently are also defined to be constant. For example, a real student athletic fee is most likely a constant that should always have the same value for every student. The field `athleticFee` in Figure 7-15 is not defined as a constant field. You could make it constant in the same way you make other fields constant—by inserting the word `const` in front of its name when you declare it. The declaration, using conventional constant identifier capitalization, would be as follows:

```
const static double ATHLETIC_FEE;
```

The definition would be written as follows:

```
const double ATHLETIC_FEE = 45.25;
```

Static fields frequently are constant. However, `static` does not mean the same thing as constant. Each of the following is true:

- Constant fields are never changeable.
- Non-constant fields are changeable.

- Static fields hold the same value for every object instantiated from a class because there is only one copy that all instances share.
- Non-static fields can hold different values for every object of a class; there is a separate copy for each instantiation.
- Non-constant, non-static fields can hold different values for every object and can be changed.
- Constant, non-static fields cannot be changed, but a copy is made for every instantiation.
- Non-constant, static fields hold the same value for every instantiation; there is just one memory location that all instances share but it can be altered.
- *Constant* static fields hold one unchangeable value for every object.

Because the `athleticFee` field in the `Student` class in Figure 7-17 was not declared constant, you could insert a statement like `aFreshman.athleticFee = 139.95;` into the `main()` function. From that point on, every `Student` object (both the `aFreshman` and the `aSophomore`) would have an `athleticFee` of 139.95 instead of 45.25. Alternately, and more logically, you could also make the following statement:

```
Student::athleticFee = 139.95;
```

This statement also would change every `Student`'s `athleticFee` field from that point on. Again, you can see that you do not need to create any objects to use a `static` class variable.

NOTE Although you can use static class members with an object name or by using them with the class name, some programmers prefer to use them only with the class name. That's because they perceive static members to truly belong to the class as a whole.

NOTE Although static fields frequently are also constant, it is not always so. For example, in game programming you might create an `Alien` class. As each `Alien` enters the game, you might want each object to "know" how many of its kind exist because you allow an `Alien` to act more daring if more comrades are in play. You could create a non-constant static field for the class with a value that increased as more `Alien`s entered the game, and to which all `Alien` objects had access.

USING STATIC FUNCTIONS

In C++, a `static` field can be accessed with an object and also without one. (Fields that are not `static` are instance fields and always need a declared object when you use them.) In the program shown in Figure 7-17, the `static athleticFee` field is `public`, which is why you can access it directly, as in the last shaded output in the figure. If `athleticFee` were `private`, and you wanted to access it without using an object, you would have to use a `public` function to access the value, as you do with any other `private` variable. Additionally, the function would have to be a `static` function. A **`static` function** is one you can use with or without a declared object. On the other hand, a **non-`static` function** requires an object.

When an object uses a non-`static` function, as in `aFreshman.getAthleticFee()`, the function receives a pointer to the object. The pointer is known as the `this` pointer, and you will learn about it in the next section.

NOTE Non-static functions can access static variables (provided there is an object) as well as non-static ones. Static functions can access only static members; they cannot access non-static variables.

Static fields do not need to be associated with an object; they are class-wide fields. When you do not want to declare an object, you still can access a `static`, class-wide field only by using a function that is also `static`. In other words, a field that is not associated with an object can be accessed only by a function that is not associated with an object—you need a `static` function in order to access a `static` field when you choose not to use an object.

You use a `static` function by prefixing the function name with the class name and the scope resolution operator. Figure 7-19 shows a modified `Student` class that holds a `private`, `static` athletic fee. Because the field is `static`, it can be used without an object. The `static public` function `getAthleticFee()` requires no object; in the shaded statement in `main()` it is used without an object. Figure 7-20 shows the output.

```
#include<iostream>
using namespace std;
// declaration section:
class Student
{
   private:
      int idNum;
      static double athleticFee;
   public:
      void setIdNum(int);
      int getIdNum();
      static double getAthleticFee();
};
// implementation section:
double Student::athleticFee = 45.25;
void Student::setIdNum(int num)
{
   idNum = num;
}
int Student::getIdNum()
{
   return idNum;
}
double Student::getAthleticFee()
{
   return athleticFee;
}
int main()
{
   cout << "Everyone's athletic fee is " <<
      Student::getAthleticFee(); << endl;
   return 0;
}
```

Figure 7-19 A `Student` class containing a static field and function and a program that demonstrates their use

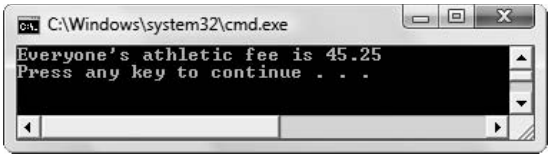

Figure 7-20 Output of application in Figure 7-19

So, in summary, in C++ a class's members can be organized in a large number of ways:

- A class can contain any number of non-static fields, static fields, non-static functions and static functions, and any of them might be `public` or not.
- A non-class function (like a `main()` function) can never directly access a class's `private` fields or functions whether they are static or not.
- If you use an object in a non-class function (like a `main()` function), it can access a class's public fields and functions, whether they are static or non-static.
- If you do not use an object in a non-class function, then you can use the class name to access a class's `public static` fields and functions, but not its instance fields and functions.
- If you want to use a class's function to access a static field, then the function must be `static`.
- A non-`static` function can use a static field, but a `static` function cannot use a non-static field.

T T F

TWO TRUTHS AND A LIE: USING STATIC CLASS MEMBERS

1. If a class has a non-static field, and you instantiate 10 objects, 10 separate fields are stored in memory.
2. If a class has a static field, and you instantiate 10 objects, only one copy of the field resides in memory.
3. In C++, you must use a `static` function without an object.

The false statement is #3. In C++, you can use a `static` function with or without an object.

UNDERSTANDING THE `THIS` POINTER

When you define a class, you include fields and functions. If the class has two non-static fields and one static field such as the `Employee` class shown in Figure 7-21, and you then declare two `Employee` objects (see shading in `main()` function), you reserve storage for two versions of the non-static fields. However, you store only one version of the static field that every object can use. Figure 7-22 shows how the data fields are stored for the two objects declared in the `main()` function in Figure 7-21. The `clerk` and the `driver` each have their own ID number and `payRate`, but only one `companyIdNum` is stored in memory. In the `main()` function, each object is created and passed to a `display()` function that accepts an `Employee` parameter. Figure 7-23 shows the program output.

```
#include<iostream>
using namespace std;
class Employee
{
   private:
      int idNum;
      double payRate;
      const static int COMPANY_ID_NUM;
   public:
      void setIdNum(int);
      void setPayRate(double);
      int getIdNum();
      double getPayRate();
      static int getCompanyIdNum();
};
const int Employee::COMPANY_ID_NUM = 12345;
void Employee::setIdNum(int id)
{
   idNum = id;
}
void Employee::setPayRate(double rate)
{
   payRate = rate;
}
int Employee::getIdNum()
{
   return idNum;
}
double Employee::getPayRate()
{
   return payRate;
}
int Employee::getCompanyIdNum()
{
   return COMPANY_ID_NUM;
}
int main()
{
   Employee clerk, driver;
   void display(Employee);
   clerk.setIdNum(777);
   clerk.setPayRate(13.45);
   driver.setIdNum(888);
   driver.setPayRate(18.79);
   display(clerk);
   display(driver);
   return 0;
}
```

NOTE
In Figure 7-21, notice that when each `Employee` object is passed to the `display()` function, it is known by the local identifier `emp` within the function.

Figure 7-21 `Employee` class and `main()` function that creates two `Employee` objects (*continued*)

```
void display(Employee emp)
{
   cout << "Employee #" << emp.getIdNum() <<
      "\'s pay rate is " << emp.getPayRate() <<
      ". The company ID is #" << emp.getCompanyIdNum() <<
      endl;
}
```

Figure 7-21 (*continued*)

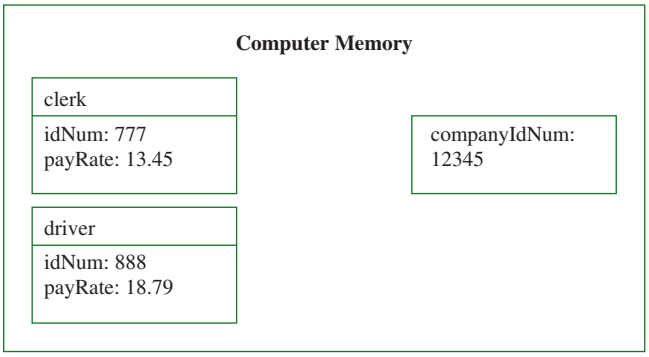

Figure 7-22 How non-`static` and `static` fields are stored in the program in Figure 7-21

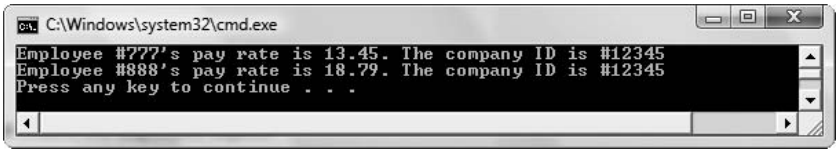

Figure 7-23 Output of program in Figure 7-21

Each `Employee` needs a unique ID number, so that field is not made `static`. Because all `Employee` objects use the same company ID, it would be wasteful to store multiple copies of the same value. Additionally, if the `static` value required changing (if, for example, the company acquired a new company ID number), then if there was a separate field for each `Employee`, each version associated with each object would have to be updated. As a `static` field, `COMPANY_ID_NUM` is stored once.

Just as it would be wasteful to store a shared company ID number separately for each declared `Employee`, it also would waste space if you stored the code for the member functions `setIdNum()`, `getIdNum()`, `setPayRate()`, and `getPayRate()` separately for each

Employee. The Employee class in Figure 7-21 contains only a few member functions. Imagine an Employee class with 20 member functions, and a program that declares 1,000 Employee objects. If the same function code were stored repeatedly for every instance of a class, the storage requirements would be enormous.

Luckily, C++ does not store member functions separately for each instance of a class. Instead, one copy of each member function is stored, and each instance of a class uses the same function code. Whether you use the expression clerk.getIdNum() or driver.getIdNum(), you call the same copy of the getIdNum() function. This does not mean the output results are identical, as you can see from Figure 7-23. Even though each call to the function uses the same programming instructions, each call to getIdNum() uses a different memory location to access the appropriate idNum.

Because only one copy of each function exists, when you call a non-static member function, it needs to know which object to use. To ensure that the function uses the correct object, you use the object's name, such as clerk or driver, with the dot operator. The address of the correct object, clerk or driver, is then automatically passed to the non-static function.

Within a function like getIdNum(), the address of the calling object is stored in a special pointer called the **this pointer**. The this pointer holds the memory address of the current object that is using the function; that's why it is named this—it refers to "this object" as opposed to any other object. The this pointer is automatically supplied every time you call a non-static member function of a class. For example, when you make the function call clerk.getIdNum(), the actual function call made by the compiler is getIdNum(&clerk). Similarly, when you make the function call driver.getIdNum(), the actual function call is getIdNum(&driver). In other words, when you use an object, the dot operator, and a non-static function of the object's class, you actually pass the specific object's address to the function as an unseen argument. The actual argument list used by the compiler for the getIdNum() function is getIdNum(Employee *this). The getIdNum() function receives a pointer to an Employee object; the automatically supplied object address is the means by which the function knows which Employee's data to retrieve.

NOTE Don't explicitly send the address of the object when you call a non-static member function; if you do, the function receives two address values—the one you typed and the one automatically sent.

NOTE The this pointer also is passed to non-static member functions when they receive additional, explicitly stated arguments. Figure 7-21 shows that the Employee class setIdNum() function receives an integer argument. In addition, it receives a pointer to the object it manipulates. When you make the statement clerk.setIdNum(345);, the actual parameter list used by the compiler for setIdNum() is setIdNum(Employee *this, const int id);. The actual function call used by the compiler is setIdNum(&clerk, 345);. When you call a non-static member function using an object, the first (and sometimes only) argument passed to the function is a pointer that holds the address of the object to which you are referring.

NOTE The this pointer is a constant pointer. You cannot modify it, as in the following:

```
this = &someOtherObject;  // illegal!
```

USING THE this POINTER EXPLICITLY

Within any member function, you can prove that the `this` pointer exists and that it holds the address of the current object. You do so by explicitly using the `this` pointer to access the object's data fields. In Figure 7-24, the four non-static `Employee` class member functions have been rewritten to explicitly refer to the `this` pointer when an object's field is used. Notice in the shaded sections of the figure that the `this` pointer can precede any field. If you substitute the functions in Figure 7-24 for their counterparts in the original class definition in Figure 7-21, there is no difference in the execution of the program; there would be no difference in any program that uses the `Employee` class. The versions of the functions shown in Figure 7-24 simply illustrate that the `this` pointer exists, and that you can refer to the contents stored in the `this` pointer like you can with any other pointer—by placing an asterisk in front of the pointer's name. (The parentheses are required in `(*this).employeeIdNum` because only `this` is a pointer. Without the parentheses, the compiler would treat `this.employeeIdNum` as a pointer, which it is not—it is an integer, and you cannot dereference it using the asterisk.)

```
void Employee::setIdNum(int id)
{
   (*this).idNum = id;
}
void Employee::setPayRate(double rate)
{
   (*this).payRate = rate;
}
int Employee::getIdNum()
{
   return (*this).idNum;
}
double Employee::getPayRate()
{
   return (*this).payRate;
}
```

Figure 7-24 `Employee` non-static member functions explicitly using the `this` pointer

USING THE POINTER-TO-MEMBER OPERATOR

Figure 7-25 shows yet another way to use the `this` pointer. The functions operate like those in Figure 7-24, but they use the C++ **pointer-to-member operator**, which looks like an arrow and is constructed by using a dash followed by a right-angle bracket (or greater-than sign). Any pointer variable that is declared to point to an object can be used to access individual fields or functions of that class by using the parentheses and the asterisk shown in Figure 7-24. Or, the pointer-to-member operator shown in Figure 7-25 can be used.

```
void Employee::setIdNum(int id)
{
   this->idNum = id;
}
void Employee::setPayRate(double rate)
{
   this->payRate = rate;
}
int Employee::getIdNum()
{
   return this->idNum;
}
double Employee::getPayRate()
{
   return this->payRate;
}
```

Figure 7-25 Explicitly using the `this` pointer with the pointer-to-member operator

Programmers usually do not code their class' member functions as shown in Figure 7-24 or Figure 7-25. The functions work fine without typing the `this` references. Often, you do not have to be concerned with the `this` pointer. Although it's helpful to understand how the function knows which object to use, the reference is made automatically, and you don't have to think about it. There are, however, occasions when you might want to use the `this` pointer explicitly within member functions, and other occasions when you will see the `this` pointer used in programs. The most common occurrence is when you want to return the pointer that was sent to a function (for example, `return(*this);`). As another example, you could use the `this` pointer when an argument to a function has the same identifier as a field. Then you can use the `this` pointer to identify the object's field. In the version of the `Employee` class `setIdNum()` function in Figure 7-26, if you did not include the shaded `this` pointer, then the value of the local argument `idNum` would be assigned to itself and the object's `idNum` would never receive an appropriate value.

NOTE
The only type of class member function that does not have a `this` pointer is a static member function.

```
void Employee::setIdNum(const int idNum)
{
   this->idNum = idNum;
}
```

Figure 7-26 Using the `this` pointer to differentiate between an object's field and a local variable with the same identifier

T T F

TWO TRUTHS AND A LIE: UNDERSTANDING THE `this` POINTER

1. When a client program calls a non-`static` function that is the member of a class, you must explicitly pass a `this` pointer to the function.
2. Within a class member function, a `this` pointer holds the memory address of the current object that is using the function.
3. Assume a class contains a non-static field named `product`. Within a member function, the following two expressions are equivalent:

```
(*this).product
this->product
```

The false statement is #1. When a client program calls a non-`static` function that is the member of a class, a `this` pointer is passed to the function automatically.

UNDERSTANDING POLYMORPHISM

Polymorphism is the object-oriented program feature that allows the same operation to be carried out differently depending on the object. When you speak English, you use the same instructions with different objects all the time. For example, you interpret a verb differently depending on the object to which it is applied. You catch a ball, but you also catch a cold. You run a computer, a business, a race, or a stocking. The meanings of "catch" and "run" are different in each case, but you can understand each operation because the combination of the verb and the object makes the command clear.

When you write C++ (and other object-oriented) programs, you can send the same message to different objects and different types of objects. Suppose you have two members of the `Employee` class, `clerk` and `driver`. You can display the ID number of each member with the following statements:

```
cout <<  clerk.getId() << endl;
cout <<  driver.getId();
```

No confusion arises as to which member's ID number should be displayed, because the correct pointer is passed as a `this` pointer each time you call the `getId()` function.

Similarly, suppose you have three different objects that are members of different classes: a `clerk` who is a member of the `Employee` class, a `shirt` that is a member of the `Inventory` class, and `XYZCompany`, a member of the `Supplier` class. Each object can certainly have different types of ID numbers that are constructed using different rules, and some ID numbers might be different data types. (For example, many ID numbers are stored as strings.) Even though objects instantiated from classes are different, in each case it seems appropriate to call a function that returns an ID number `getId()`. C++ allows you to create three very different member functions, one for each of the classes in question, but use the same identifier for each. One might return a simple integer, one might return an integer with a number sign (#) preceding it, and one might return a string composed of digits and letters.

Similarly, each class might need a `setId()` function. One version might impose a three-digit limit on the ID, another might impose a five-digit limit, and a third might add a final check digit calculated from a mathematical formula that employs the first digits. The functions can contain different numbers of statements as well as statements with different content, but they all can have the same identifier and use a unique, appropriate ID number format. The `this` pointer is sent to each function holding the appropriate address so each function call uses the correct object to do its work.

NOTE The concept of polymorphism includes far more than using the same function name for different function bodies. In Chapter 6, you learned about overloading functions where functions with the same name were differentiated by their argument lists. That form of polymorphism is **ad hoc polymorphism**. In Chapters 9 and 10 you will learn how a single function implementation can be used with more than one data type—a form called **pure polymorphism**.

When you can apply the same function name to different objects, your programs become easier to read and make more sense. It takes less time to develop a project; you also can make later changes more rapidly. C++ programmers often can write programs faster than programmers who use non-object-oriented programming languages, and polymorphism is one of the reasons why.

T T F

TWO TRUTHS AND A LIE: UNDERSTANDING POLYMORPHISM

1. Multiple classes can contain methods with the same identifier.
2. Polymorphism is the object-oriented program feature that allows the same object to use multiple methods.
3. Polymorphism helps make a program's method calls more natural to use.

The false statement is #2. Polymorphism is the object-oriented program feature that allows the same operation to be carried out differently depending on the object.

YOU DO IT

CREATING AND USING A CLASS

In the next set of steps, you create a class that implements a function. Then you write a short demonstration program to instantiate an object of the class, and use the object's fields and functions.

1. Open a new file. Type a comment identifying the program's purpose, which is to create and demonstrate a class that holds information about a college course. Then type the first statements you need:

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

2. Next type the `CollegeCourse` class declaration section. It includes three private fields that hold the three-letter code of the department in which the course is offered, the course number, and the number of seats available for which students can enroll. It also contains the definition for three functions—one that sets the course department and number, one that sets the number of students allowed in the course, and one that displays the course data. Don't forget the class-ending semicolon, which is easy to omit.

```
class CollegeCourse
{
   private:
      string department;
      int courseNum;
      int seats;
   public:
      void setDepartmentAndCourse(string, int);
      void setSeats(int);
      void displayCourseData();
};
```

3. Next, in the implementation section, write the `setDepartmentAndCourse()` function. Notice the class name and scope resolution operator before the function name in the header. The function takes two arguments. The first is the string representing the college department, for example CIS, ENG, or SOC. The second is the course number, for example 101 or 200. The two are assigned to fields without any validation, although in a more sophisticated program you might want to ensure that the department was an existing department in the college, or that the course number fell between specific values.

```
void CollegeCourse::setDepartmentAndCourse(string dept, int num)
{
   department = dept;
   courseNum = num;
}
```

4. The `setSeats()` function accepts a number of students allowed to enroll in a course. The function assures that the number is not negative, assigning 0 if it is. The function assigns the local seats variable to the class member variable. Notice that because the function parameter and the object's field have the same identifier, the `this` pointer must be used with the class member to distinguish it.

```
void CollegeCourse::setSeats(int seats)
{
   if(seats < 0)
      seats = 0;
   this->seats = seats;
}
```

5. The `displayCourseData()` function that is a member of the `CollegeCourse` class contains a single statement that displays the values of the `CollegeCourse`'s data fields.

```
void CollegeCourse::displayCourseData()
{
   cout << department << courseNum <<
      " accommodates " << seats <<
      " students" << endl;
}
```

6. Start a `main()` function that declares a `CollegeCourse` object and prompts the user for values. Notice how the second two prompts make use of the data the user has already entered, to make the prompts more personalized.

```
int main()
{
   CollegeCourse myMondayClass;
   string dept;
   int num;
   int students;
   cout << "Enter the department that hosts the class," <<
      endl;
   cout << "for example 'CIS' >>  ";
   cin >> dept;
   cout << "Enter the course number, for example, for " <<
      dept << "101, enter 101 >>  ";
   cin >> num;
   cout << "Enter the number of students that are allowed" <<
      endl;
   cout << "to enroll in " << dept << num << " >>  ";
   cin >> students;
```

7. Add the code to assign the user-entered values to the declared `CollegeCourse` object and then display the results. End the `main()` function with a `return` statement and a closing curly brace.

```
   myMondayClass.setDepartmentAndCourse(dept, num);
   myMondayClass.setSeats(students);
   myMondayClass.displayCourseData();
   return 0;
}
```

8. Save the file as **CollegeCourse.cpp**. Compile and run the program. The output should look like Figure 7-27.

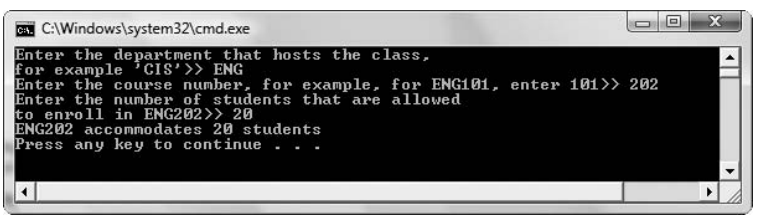

Figure 7-27 Output of the `CollegeCourse` application

USING A static FIELD

In the following set of steps, you use a `private static` class variable to keep track of how many objects have been initialized. You will use a `public static` function to display the `private static` variable.

1. Open a new file in your C++ editor. Type the necessary `include` and `using` statements, and begin a class that holds information about business letters you write. For simplicity, this class contains just three `private` fields: the title and last name of the recipient and a count of how many letters have been sent. (A real-life class might also include the recipient's mailing address data.)

```
#include<iostream>
#include<string>
using namespace std;
class Letter
{
   private:
      string title;
      string recipient;
      static int count;
```

2. Include three functions in the public section of the class—one to set the recipient data, one to display the letter greeting, and one to display the count of letters that have been sent. The `displayCount()` function is `static` because it will be used without an object to display the `static` field.

```
  public:
     void setRecipient(string, string);
     void displayGreeting();
     static void displayCount();
};
```

3. Initialize the `Letter` count to zero. This statement is placed outside the body of the class definition.

```
int Letter::count = 0;
```

4. Write the `setRecipient()` function. This function is a member of the `Letter` class, so you use the class name and scope resolution operator in the function heading. The function sets the title and name of the `Letter` recipient with parameter values that are passed into it. Because the identifier for the name argument is the same as a field in the class, the `this` pointer is used to refer to the object's field. The `setRecipient()` function also adds 1 to the count of letters sent.

```
void Letter::setRecipient(string title, string name)
{
   this->title = title;
   recipient = name;
   ++count;
}
```

5. Write the `displayGreeting()` function that displays a full letter salutation including a title, name, and comma, for example, "Dear Mr. Johnson,".

```
void Letter::displayGreeting()
{
    cout << "Dear " << title << ". " <<
       recipient << "," << endl;
}
```

6. The `displayCount()` function displays the count of the number of `Letter` objects that have been created.

```
void Letter::displayCount()
{
   cout << "Current count is " << count << endl << endl;
}
```

7. Next, begin a `main()` function that declares a `Letter` object as well as two strings into which the user can enter a title and name. Also prototype a `displayLetter()` function to which you will pass a `Letter` recipient.

```
int main()
{
   Letter aLetter;
   string title;
   string name;
   void displayLetter(Letter);
```

8. Next, declare a loop control variable named `more`. The variable is initialized to 'y'. A loop continues while the value of `more` is not 'n'. At the end of the loop, the user provides a new value for `more`, perhaps entering 'n' to stop the loop's execution. Within the loop, prompt the user for a title and name, then use these values to set the `Letter`'s values. Pass the `Letter` to the `displayLetter()` function before displaying the count of the number of letters created and before asking the user whether more letters should be created. The `displayCount()` function is not used with an object (although it could be); instead, it is used without an object which can be done because it is `static`. Include a closing curly brace for the `while` loop and a `return` statement and closing brace for the `main()` function.

```
   char more = 'y';
   while(more != 'n')
   {
      cout << "Enter title for recipient, " <<
         "for example 'Mr' or 'Ms'  >>  ";
      cin >> title;
      cout <<  "Enter last name of recipient >>  ";
      cin >> name;
      aLetter.setRecipient(title, name);
      displayLetter(aLetter);
      Letter::displayCount();
```

```
        cout << "Do you want to send another - y or n? ";
        cin >> more;
    }
    return 0;
}
```

9. Finally, write the `displayLetter()` function. This function is not a member of the `Letter` class; it is a function called by `main()`. The `Letter` that contains the recipient data is passed to this function in which a business letter is created. The function uses the `displayGreeting()` function that is part of the `Letter` class to display a greeting, then displays the letter body.

```
void displayLetter(Letter letter)
{
   letter.displayGreeting();
   cout << "Thank you for your recent order. We look" <<
      endl << "forward to serving you again soon." <<
      endl << "                    Sincerely," <<
      endl << "                    ABC Company" <<
      endl << endl;
}
```

10. Save the program as **Letter.cpp**, and compile and run the program. A typical run is shown in Figure 7-28. Notice how each `Letter` passed to the `displayLetter()` function has a unique greeting. Also notice that the `static count` continues to increase as each new `Letter` is created; this is possible because all `Letter` objects share the same `count` field.

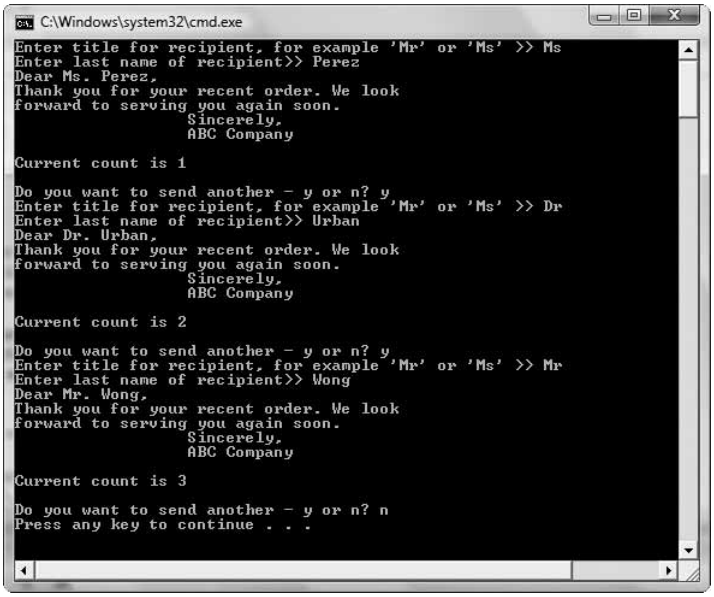

Figure 7-28 Typical execution of the `Letter` application

UNDERSTANDING HOW `static` AND NON-`static` FIELDS ARE STORED

When you create an object that contains non-`static` data fields, the fields are stored together in adjacent memory locations. However, `static` fields do not belong to any one object; they belong to the class as a whole. Therefore, it makes sense that any `static` data fields are not stored with any one object. In the next steps, you will prove this for yourself by adding a function to the `Letter` class that displays the memory locations of an object's fields.

1. If it is not still open, open the **Letter.cpp** file in your text editor.

2. In the public section of the `Letter` class, add a declaration for the function that will display the memory locations of the object's fields:

```
void showMemoryLocations();
```

3. Below the `displayCount()` function but above `main()`, add the implementation of the `showMemoryLocations()` function. The function uses the address operator (`&`) to display the memory address of each field in a `Letter` object.

```
void Letter::showMemoryLocations()
{
   cout << "Memory addresses:" << endl;
   cout << "title:     " << &title << endl;
   cout << "recipient  " << &recipient << endl;
   cout << "count      " << &count << endl;
}
```

4. In the `while` loop within the `main()` function, just after the call to the `displayCount()` function, add a call to the `showMemoryLocation()` function as follows:

```
aLetter.showMemoryLocations();
```

5. Save the modified file as **Letter2.cpp**, then compile and execute the program. Figure 7-29 shows the start of a typical execution. After the user enters a title and name, the letter and the count display. The function that displays the field memory addresses executes. From the output, you can see that the memory addresses of the title and recipient fields are very close to each other, but the static count field is in a very different memory location. It makes sense that an object's fields are stored adjacently; it also makes sense that a static field is not stored with any individual object.

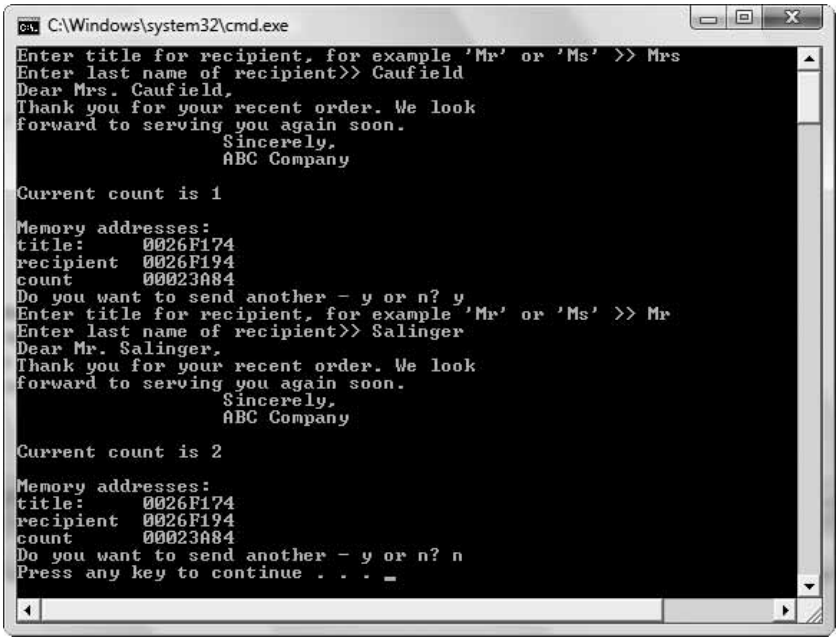

Figure 7-29 Execution of the `Letter2` application

CHAPTER SUMMARY

- A class is a category of objects; they provide a convenient way to group related data and functions. By default, all members of a class are `private`, meaning they cannot be accessed using any statements in any functions that are not also part of the class.
- When you create a class name for a group of associated variables, you hide, or encapsulate, the individual components. When you design C++ classes, you think about what the class instantiations (the objects) will *do* and how programmers will make them do it, as well as what the objects *contain*. Therefore your classes will contain fields, and functions that often act as interfaces to the fields contained in the objects. Typically, the fields in your classes will be private, but your interfacing functions will be public.
- When you construct a class, you create two parts. The first part is a declaration section, which contains the class name, variables (attributes), and function prototypes. The second part created is an implementation section, which contains the functions themselves.
- When you create a class, usually you want to make data items private, to control how they are used, and to make functions public, to provide a means to access and manipulate the data. However, if you have a reason to do so, you are free to make particular data items public. Similarly, not all functions are public; some are private.

- The scope resolution operator is a C++ operator that identifies a member function as being in scope within a class. It consists of two colons (::).
- When it is created, each object gets its own block of memory for its data members. When you declare a field as static within a class, only one memory location is allocated, no matter how many objects you instantiate. In other words, all members of the class share a single storage location for a static data member of that same class. Because it uses only one memory location, a static data member is defined (given a value) in a single statement outside the class definition. Static fields frequently are also defined to be constant, although they do not have to be.
- When you do not want to declare an object, you still can access a `static`, class-wide field only by using a function that is also `static`.
- One copy of each member function in a class is stored no matter how many objects exist, and each instance of a class uses the same function code. When you call a member function, it knows which object to use because you use the object's name, such as `clerk` or `driver`, with the dot operator. The address of the correct object is stored in the `this` pointer and automatically passed to the function.
- Within any member function, you can explicitly use the `this` pointer to access the object's data fields. You can use the C++ pointer-to-member operator, which looks like an arrow and is constructed by using a dash followed by a right angle bracket (or greater-than sign).
- Polymorphism is the object-oriented program feature that allows the same operation to be carried out differently depending on the object. When you can apply the same function name to different objects, your programs become easier to read and make more sense. It takes less time to develop a project; you also can make later changes more rapidly.

KEY TERMS

A **class** is a category of objects; it is a new data type you create that is more complex than the basic data types.

An **abstract data type (ADT)** is a type you define, as opposed to types that are defined by C++.

An **extensible language** is a language to which you can add your own data types.

The access modifier **`private`** means a class member cannot be accessed using any statements in any functions that are not also part of the class.

An **access modifier** assigns accessibility to the declared variables that follow it.

To **encapsulate** components is to contain them.

An **interface** intercedes between you and the inner workings of an object.

To **instantiate** an object is to declare or create it.

A function that is a user of your class is a **class client**.

The **declaration section** of a class contains the class name, variables (attributes), and function prototypes.

The **implementation section** of a class contains the functions.

Instance fields or **instance variables** are the fields that are stored uniquely for each class instance.

When a field in a class is **static**, only one memory location is allocated, no matter how many objects you instantiate. All members of the class share a single storage location for a `static` data member of that same class. You can use a static field with or without an object instantiation.

Non-static fields are instance fields for which a separate memory location is used for each instantiated object.

Static variables are sometimes called **class variables**, **class fields**, or **class-wide fields** because they don't belong to a specific object; they belong to the class, and you can use them even if you never instantiate an object.

A **`static` function** is one you can use with or without a declared object.

A **non-`static` function** requires an object.

The **`this` pointer** holds the memory address of the current object that is using a non-static function. It is automatically supplied every time you call a non-static member function of a class.

The **pointer-to-member operator** looks like an arrow and is used between an object's pointer and an object's field. It is constructed by using a dash followed by a right angle bracket (or greater-than sign).

Ad hoc polymorphism is the type of polymorphism that occurs when functions are overloaded.

Pure polymorphism is the type of polymorphism that occurs when a single function implementation can be used with more than one data type.

REVIEW QUESTIONS

1. C++ programmers refer to a type you define as an ADT, or an _____ .

   a. abstract data type

   b. alternative data theory

   c. adaptable data type

   d. anonymous default test

2. You have defined a class named `Invoice` that contains two non-static private fields, `invoiceNumber` and `amount`. When you write a `main()` function and declare one `Invoice` object named `anInvoice`, you can display the object's amount field with the statement _____ .

   a. `cout << anInvoice(amount);`

   b. `cout << Invoice.amount;`

   c. `cout << anInvoice.amount;`

   d. None of the above

3. You have defined a class named `Invoice` that contains two non-static private fields, `invoiceNumber` and `amount`. The class also contains a non-static public function that displays the amount; the function's prototype is `void showAmount();`. When you write a `main()` function and declare one `Invoice` object named `anInvoice`, you can display the object's amount field with the statement _____ .

   a. `cout << Invoice::showAmount();`

   b. `cout << Invoice.showAmount();`

   c. `cout << anInvoice.showAmount();`

   d. None of the above

4. When you encapsulate class components, you _____ them.

   a. destroy
   c. display
   b. hide
   d. format

5. The word interface is most closely associated with _____ .

   a. prototypes of public functions in classes

   b. declarations of private variables in classes

   c. class descriptions

   d. public function implementations

6. You create a class in two sections called _____ .

   a. declaration and implementation

   b. typical and atypical

   c. common and protected

   d. abstract and concrete

7. A class named `Apartment` contains a non-static public function named `showRent()` that neither takes nor receives arguments. The correct function header for the `showRent()` function is _____ .

   a. `showRent()`

   b. `Apartment::showRent()`

   c. `void Apartment::showRent()`

   d. `void Apartment.showRent()`

8. The operator you use in a function header that ties a function name to a class is the _____ operator.

   a. pointer-to-member
   c. scope resolution
   b. `this`
   d. binary

9. A technique that programmers use to provide object encapsulation is to usually make objects' data _____ .

   a. `private`
   c. accessible
   b. polymorphic
   d. `static`

10. The most common arrangement for a class is that _____ .

   a. data members are `static` and functions are not
   b. data members are not `static` and functions are
   c. data members are `public` and functions are not
   d. data members are not `public` and functions are

11. You control how data is stored in classes by using _____ .

   a. polymorphism
   b. well-written public functions
   c. anonymous classes
   d. private methods that clients cannot access

12. Which is true?

   a. All functions in classes are private.
   b. Most functions in classes are private.
   c. Most functions in classes are public.
   d. Most classes contain no functions.

13. A program that uses a class is known as the class's _____ .

   a. patient
   c. consumer
   b. superclass
   d. client

14. You create a class named `Car` with private non-static data fields named `year`, `make`, and `price`. The `Car` class contains a public non-static function named `setYear()` whose header is `void Car::setYear(int year)`. Within this function, which statement correctly sets an object's field to the value of the parameter?

   a. `year = year;`
   b. `Car::year = year;`
   c. `year = Car::year;`
   d. Two of the above are correct.

15. You create a class named `Car` with private non-static data fields named `year`, `make`, and `price`. The `Car` class contains a public non-static function named `setMake()` whose header is `void Car::setMake(string carMake)`. Within this function, which statement correctly sets an object's field to the value of the parameter?

    a. `make = carMake;`

    b. `this->make = carMake;`

    c. `carMake = make;`

    d. Two of the above are correct.

16. To create just one memory location for a field no matter how many objects you instantiate, you should declare the field to be _____ .

    a. private
    c. static
    b. anonymous
    d. stagnant

17. A function that you use to display a static variable without creating an object must be _____ .

    a. private
    c. static
    b. anonymous
    d. stagnant

18. The pointer that is automatically supplied when you call a non-static class member function is the _____ pointer.

    a. `public`
    c. implicit
    b. `this`
    d. reference

19. When you call a public static function from outside its class, you _____ .

    a. can use an object

    b. must use an object

    c. must not use an object

    d. Static functions cannot be called from outside the class.

20. The feature in object-oriented programs that allows the same operation to be carried out differently, depending on the object, is _____ .

    a. encapsulation
    c. pointer creation
    b. inheritance
    d. polymorphism

EXERCISES

1. Define a class named `Movie`. Include private fields for the title, year, and name of the director. Include three public functions with the prototypes `void Movie::setTitle(string);`, `void Movie::setYear(int);`, and `void setDirector(string);`. Include another function that displays all the information about a `Movie`. Write a `main()` function that declares a movie object named `myFavoriteMovie`. Set and display the object's fields. Save the file as **Movie.cpp**.

2. a. Define a class named `Customer` that holds private fields for a customer ID number, last name, first name, and credit limit. Include four public functions that each set one of the four fields. Do not allow any credit limit over $10,000. Include a public function that displays a `Customer`'s data. Write a `main()` function in which you declare a `Customer`, set the `Customer`'s fields, and display the results. Save the file as **Customer.cpp**.

   b. Write a `main()` function that declares an array of five `Customer` objects. Prompt the user for values for each `Customer`, and display all five `Customer` objects. Save the file as **Customer2.cpp**.

3. a. Define a class named `GroceryItem`. Include private fields that hold an item's stock number, price, quantity in stock, and total value. Write a public function named `dataEntry()` that calls four private functions. Three of the private functions prompt the user for keyboard input for a value for one of the data fields stock number, price, and quantity in stock. The function that sets the stock number requires the user to enter a value between 1000 and 9999 inclusive; continue to prompt the user until a valid stock number is entered. The functions that set the price and quantity in stock require non-negative values; continue to prompt the user until valid values are entered. Include a fourth private function that calculates the `GroceryItem`'s total value field (price times quantity in stock). Write a public function that displays a `GroceryItem`'s values. Finally, write a `main()` function that declares a `GroceryItem` object, assigns values to its fields, and uses the display function. Save the file as **Grocery.cpp**.

   b. Write a `main()` function that declares an array of 10 `GroceryItem` objects. Assign values to all 10 items and display them. Save the file as **Grocery2.cpp**.

4. Write the class definition for a `Dog`. Private data fields include name, breed, and age, and a constant static field for the license fee, which is $12.25. Create public member functions to set and display the data. Write a `main()` function that demonstrates the class operates correctly. Save the file as **Dog.cpp**.

5. a. Write a class definition for an `Order` class for a nightclub that contains a table number, a server's name, and the number of patrons at the table. Include a private static data member for the table minimum charge, which is $4.75. Write a `main()` function that declares no `Order` objects, but that uses a static member function to display the table minimum charge. Save the file as **Order.cpp**.

b. Using the same `Order` class, write a `main()` function that declares an `Order` object, assigns appropriate values, and displays the `Order` data, including the minimum charge for the table (the minimum charge times the number of patrons at the table). Save the file as **Order2.cpp**.

6. Write a class definition for a `Die` class that represents a six-sided playing die. Include a field to hold the current value (the die side facing up), and functions to get and set the value. Also include a static, constant field that holds the maximum die value (6). Write a program that plays a game of "21" by instantiating a `Die` object and using it to take turns "rolling" for the computer and for the player using a random number. (See Appendix E for information on generating random numbers in a specified range.) Keep a running total for the computer and for the player. The goal is to get as close to a total of 21 without going over. The player can choose to stop rolling at any time. The computer will stop rolling when its total reaches 19 or more. Declare a winner when one opponent goes over 21 or both opponents quit. If both opponents quit with the same total (19, 20, or 21), then declare the player to be the winner. Save the program as **TwentyOneWithDice.cpp**.

7. Computer games often contain different characters or creatures. For example, you might design a game in which alien beings possess specific characteristics such as color, number of eyes, or number of lives. Design a character for a game, creating a class to hold at least three attributes for each character. Include methods to get and set each of the character's attributes. Then write an application in which you create at least two characters each of which has a random age up to 100, a random number of eyes up to 10, and a random number of legs, up to 12. In turn, pass each character to a display method that displays the character's attributes. Save the program as **MyCharacters.cpp**.

8. a. Write the class definition for a `Date` class that contains three private integer data members: `month`, `day`, and `year`. Create a static member to hold a slash. Create two public member functions, `setDate()` and `showDate()`. You will use the static slash in the `showDate()` function. The `setDate()` function accepts three integer arguments and passes them on to three private functions, `setMonth()`, `setDay()`, and `setYear()`. If a month is greater than 12, then set it to 12. If a day is greater than 31, then set it to 31. Write a `main()` function that instantiates several objects of the `Date` class and tests the member functions. Save the file as **Date.cpp**.

   b. Improve the `setDay()` function by ensuring the day is not higher than allowed for a month—31 for January, March, May, July, August, October, or December; 30 for April, June, September, or November; and 28 or 29 for February. February usually contains 28 days. If a year is evenly divisible by 4, but not by 100, February contains 29 days. (For example, February has only 28 days in 1900 and 2100.) However, if a year is evenly divisible by 400, then February has 29 days (as in the year 2000). Write a `main()` function that instantiates several objects of the `Date` class and tests the member functions. Save the file as **Date2.cpp**.

   c. Add a public function named `increaseDay()` to the `Date` class. Its purpose is to add 1 to the `day` field of a `Date` object. When the day is increased to more than the highest day number allowed for a month, the month increases by 1 and the day is set to 1. Write a `main()` function that declares several `Date` objects that will thoroughly test all the

functions; in other words, make sure some are close to the end of a month. Loop to call the `increaseDay()` function five times for each `Date`, displaying each version of the `Date` as soon as it is increased. Write a `main()` function that instantiates several objects of the `Date` class and tests the member functions. Save the file as **Date3.cpp**.

d. Add instructions to the `increaseDay()` function so that when the month becomes more than 12, the year increases by 1, and the month and day are set to 1. Write a `main()` function that declares a `Date` object. Set the `Date` to 12/29/2007 and call the `increaseDay()` function 400 times, displaying each version of the `Date` as soon as it is increased. Examine the output to make sure each month and year changes correctly. Save the file as **Date4.cpp**.

9. a. Define a class named `CoffeeOrder`. Declare a private static field that holds the price of a cup of coffee as $1.25. Include private integer fields that you set to a flag value of 1 or 0 to indicate whether the order should have any of the following: cream, milk, sugar, or artificial sweetener. Include a public function that takes a user's order from the keyboard and sets the values of the four fields in response to four prompts. If the user indicates both milk and cream, turn off the milk flag to allow only cream. If the user indicates both sugar and artificial sweetener, turn off the artificial sweetener flag, allowing only sugar. Include another function that displays the user's completed order. Write a `main()` function that declares a `CoffeeOrder` object and calls the data entry and display methods. Save the file as **Coffee.cpp**.

b. Using the `CoffeeOrder` class, write a `main()` function that continues to ask a user for an order in a loop until the user indicates the order is complete or 10 orders have been placed, whichever comes first. After the user indicates that ordering is complete, display a recap of all the coffee orders, including the cream, milk, sugar, and sweetener status of each, as well as a count of the number of coffees ordered and the total price. Save the file as **Coffee2.cpp**.

10. a. Define a class named `TaxReturn` that contains a tax ID number, last name, first name, annual income, number of dependents, and amount of tax owed for a taxpayer. Include constant static fields that hold the tax rates for the situations shown in the following table.

Income ($)	0 dependents	1 dependent	2 or more dependents
0–10,000	0.10	0.08	0.07
10,001–30,000	0.12	0.11	0.09
30,001–60,000	0.18	0.15	0.13
60,001 and up	0.25	0.22	0.19

Include a static function that displays the tax table. Write a `main()` function that contains a single statement that displays the tax table. Save the file as **TaxReturn.cpp**.

b. Add two public functions to the `TaxReturn` class: `setAll()`, which can set the field values and `display()`, which can display them. The `setAll()` function accepts arguments

for the tax ID number, last and first names, annual income, and number of dependents. The `setAll()` function should then call a private function that computes the tax owed. Create an array of 10 `TaxReturn` objects. In a loop, prompt the user for ID number, first and last name, annual income, and number of dependents, then use these values to set a `TaxReturn`'s fields. At the end of the loop, display all the values for all 10 `TaxReturn` objects, including the tax owed. Save the file as **TaxReturn2.cpp**.

11. Each of the following files in the Chapter07 folder of the data files provided with your book contains syntax and/or logical errors. Determine the problem in each case and fix the program. Save your solutions by adding "Fixed" to the file name, as in **DEBUG7-1Fixed.cpp**.

    a. DEBUG7-1.cpp

    b. DEBUG7-2.cpp

    c. DEBUG7-3.cpp

    d. DEBUG7-4.cpp

CASE PROJECT 1

In previous chapters, you have been developing a `Fraction` structure for Teacher's Pet Software. Now you will develop a class that contains the fields and functions that a `Fraction` needs. Create a `Fraction` class with three private data fields for whole number, numerator, and denominator. Also create a constant static public field to hold the symbol that separates a numerator and denominator when a `Fraction` is displayed—the slash. Create three public member functions for the class, as follows:

- An `enterFractionValue()` function that prompts the user to enter values for the `Fraction`. Do not allow the user to enter a value of 0 for the denominator of any `Fraction`; continue to prompt the user for a denominator value until a valid one is entered.
- A `reduceFraction()` function that reduces a `Fraction` to proper form. For example, a `Fraction` with the value 0 2/6 would be reduced to 0 1/3 and a `Fraction` with the value 4 18/4 would be reduced to 8 1/2.
- A `displayFraction()` function that displays the `Fraction` whole number, numerator, slash, and denominator.

Add any other functions to the `Fraction` class that will be useful to you. Create a `main()` function that declares a `Fraction` object, and continues to get `Fraction` values from the user until the user enters a `Fraction` with value 0 (both the whole number and numerator are 0). For each `Fraction` entered, display the `Fraction`, reduce the `Fraction`, and display the `Fraction` again.

CASE PROJECT 2

In previous chapters, you have been developing a `BankAccount` structure for Parkville Bank. Now you will develop a class that contains the fields and functions that a `BankAccount` needs. Create a class containing private fields that hold the account number and the account balance. Include a constant static field that holds the annual interest rate (3 percent) earned on accounts at Parkville Bank. Create three public member functions for the class, as follows:

- An `enterAccountData()` function that prompts the user for values for the account number and balance. Allow neither a negative account number nor one less than 1000, and do not allow a negative balance; continue to prompt the user until valid values are entered.
- A `computeInterest()` function that accepts an integer argument that represents the number of years the account will earn interest. The function displays the account number, then displays the ending balance of the account each year, based on the interest rate attached to the `BankAccount`.
- A `displayAccount()` function that displays the details of the `BankAccount`.

Create a `main()` function that declares an array of 10 `BankAccount` objects. After the first `BankAccount` object is entered, ask the user if he or she wants to continue. Prompt the user for `BankAccount` values until the user wants to quit or enters 10 `BankAccount` objects, whichever comes first. For each `BankAccount` entered, display the `BankAccount` details. Then prompt the user for a term with a value between 1 and 40 inclusive. Continue to prompt the user until a valid value is entered. Then pass the array of `BankAccount` objects, the count of valid objects in the array, and the desired term to a function that calculates and displays the values of each of the `BankAccount`s after the given number of years at the standard interest rate.

UP FOR DISCUSSION

1. In this chapter, you learned that instance data and methods belong to objects (which are class members), but that static data and methods belong to a class as a whole. Consider the real-life class named `StateInTheUnitedStates`. Name some real-life attributes of this class that are static attributes and instance attributes. Create another example of a real-life class and discuss what its static and instance members might be.
2. Think of some practice or position to which you are opposed. For example, you might have objections to organizations on the far right or left politically. Now suppose that such an organization offered you twice your annual salary to create Web sites for them. Would you do it? Is there a price at which you would do it? What if the organization was not so extreme, but featured products you found distasteful? What if the Web site you designed was not objectionable, but the parent company's policies were objectionable? For example, if you are opposed to smoking, would you design a Web site for a tobacco company? At what price? What if the site just displayed sports scores without promoting smoking directly?