PREV NEXT

[Page 74]

Chapter 3. Introduction to Classes and Objects

You will see something new. Two things. And I call them Thing One and Thing Two.

Dr. Theodor Seuss Geisel

Nothing can have value without being an object of utility.

Karl Marx

Your public servants serve you right.

Adlai E. Stevenson

Knowing how to answer one who speaks, To reply to one who sends a message.

Amenemope

OBJECTIVES

In this chapter you will learn:

- What classes, objects, member functions and data members are.
- How to define a class and use it to create an object.
- How to define member functions in a class to implement the class's behaviors.
- How to declare data members in a class to implement the class's attributes.
- How to call a member function of an object to make that member function perform its task.
- The differences between data members of a class and local variables of a function.
- How to use a constructor to ensure that an object's data is initialized when the object is created.
- How to engineer a class to separate its interface from its implementation and encourage reuse.

[Page 75]

Outline

PREV PREV NEXT NEXT

[Page 75 (continued)]

3.1. Introduction

In Chapter 2, you created simple programs that displayed messages to the user, obtained information from the user, performed calculations and made decisions. In this chapter, you will begin writing programs that employ the basic concepts of object-oriented programming that we introduced in Section 1.17. One common feature of every program in Chapter 2 was that all the statements that performed tasks were located in function `main`. Typically, the programs you develop in this book will consist of function `main` and one or more classes, each containing data members and member functions. If you become part of a development team in industry, you might work on software systems that contain hundreds, or even thousands, of classes. In this chapter, we develop a simple, well-engineered framework for organizing object-oriented programs in C++.

First, we motivate the notion of classes with a real-world example. Then we present a carefully paced sequence of seven complete working programs to demonstrate creating and using your own classes. These examples begin our integrated case study on developing a grade-book class that instructors can use to maintain student test scores. This case study is enhanced over the next several chapters, culminating with the version presented in Chapter 7, Arrays and Vectors.

PREV PREV NEXT NEXT

[Page 75 (continued)]

3.2. Classes, Objects, Member Functions and Data Members

Let's begin with a simple analogy to help you reinforce your understanding from Section 1.17 of classes and their contents. Suppose you want to drive a car and make it go faster by pressing down on its accelerator pedal. What must happen before you can do this? Well, before you can drive a car, someone has to design it and build it. A car typically begins as engineering drawings, similar to the blueprints used to design a house. These drawings include the design for an accelerator pedal that the driver will use to make the car go faster. In a sense, the pedal "hides" the complex mechanisms that actually make the car go faster, just as the brake pedal "hides" the mechanisms that slow the car, the steering wheel "hides" the mechanisms that turn the car and so on. This enables people with little or no knowledge of how cars are engineered to drive a car easily, simply by using the accelerator pedal, the brake pedal, the steering wheel, the transmission shifting mechanism and other such simple and user-friendly "interfaces" to the car's complex internal mechanisms.

[Page 76]

Unfortunately, you cannot drive the engineering drawings of a carbefore you can drive a car, it must be built from the engineering drawings that describe it. A completed car will have an actual accelerator pedal to make the car go faster. But even that's not enoughthe car will not accelerate on its own, so the driver must press the accelerator pedal to tell the car to go faster.

Now let's use our car example to introduce the key object-oriented programming concepts of this section. Performing a task in a program requires a function (such as `main`, as described in Chapter 2). The function describes the mechanisms that actually perform its tasks. The function hides from its user the complex tasks that it performs, just as the accelerator pedal of a car hides from the driver the complex mechanisms of making the car go faster. In C++, we begin by creating a program unit called a class to house a function, just as a car's engineering drawings house the design of an accelerator pedal. Recall from Section 1.17 that a function belonging to a class is called a member function. In a class, you provide one or more member functions that are designed to perform the class's tasks. For example, a class that represents a bank account might contain one member function to deposit money into the account, another to withdraw money from the account and a third to inquire what the current account balance is.

Just as you cannot drive an engineering drawing of a car, you cannot "drive" a class. Just as someone has to build a car from its engineering drawings before you can actually drive the car, you must create an object of a class before you can get a program to perform the tasks the class describes. That is one reason C++ is known as an object-oriented programming language. Note also that just as many cars can be built from the same engineering drawing, many objects can be built from the same class.

When you drive a car, pressing its gas pedal sends a message to the car to perform a taskthat is, make the car go faster. Similarly, you send **messages** to an objecteach message is known as a **member-function call** and tells a member function of the object to perform its task. This is often called requesting a service from an object.

Thus far, we have used the car analogy to introduce classes, objects and member functions. In addition to the capabilities a car provides, it also has many attributes, such as its color, the number of

doors, the amount of gas in its tank, its current speed and its total miles driven (i.e., its odometer reading). Like the car's capabilities, these attributes are represented as part of a car's design in its engineering diagrams. As you drive a car, these attributes are always associated with the car. Every car maintains its own attributes. For example, each car knows how much gas is in its own gas tank, but not how much is in the tanks of other cars. Similarly, an object has attributes that are carried with the object as it is used in a program. These attributes are specified as part of the object's class. For example, a bank account object has a balance attribute that represents the amount of money in the account. Each bank account object knows the balance in the account it represents, but not the balances of the other accounts in the bank. Attributes are specified by the class's data members.

PREV PREV NEXT NEXT

3.3. Overview of the Chapter Examples

The remainder of this chapter presents seven simple examples that demonstrate the concepts we introduced in the context of the car analogy. These examples, summarized below, incrementally build a `GradeBook` class to demonstrate these concepts:

1. The first example presents a `GradeBook` class with one member function that simply displays a welcome message when it is called. We then show how to create an object of that class and call the member function so that it displays the welcome message.

2. The second example modifies the first by allowing the member function to receive a course name as a so-called argument. Then, the member function displays the course name as part of the welcome message.

3. The third example shows how to store the course name in a `GradeBook` object. For this version of the class, we also show how to use member functions to set the course name in the object and get the course name from the object.

4. The fourth example demonstrates how the data in a `GradeBook` object can be initialized when the object is createdthe initialization is performed by a special member function called the class's constructor. This example also demonstrates that each `GradeBook` object maintains its own course name data member.

5. The fifth example modifies the fourth by demonstrating how to place class `GradeBook` into a separate file to enable software reusability.

6. The sixth example modifies the fifth by demonstrating the good software-engineering principle of separating the interface of the class from its implementation. This makes the class easier to modify without affecting any **clients of the class's objects**that is, any classes or functions that call the member functions of the class's objects from outside the objects.

7. The last example enhances class `GradeBook` by introducing data validation, which ensures that data in an object adheres to a particular format or is in a proper value range. For example, a `Date` object would require a month value in the range 112. In this `GradeBook` example, the member function that sets the course name for a `GradeBook` object ensures that the course name is 25 characters or fewer. If not, the member function uses only the first 25 characters of the course name and displays a warning message.

Note that the `GradeBook` examples in this chapter do not actually process or store grades. We begin processing grades with class `GradeBook` in Chapter 4 and we store grades in a `GradeBook` object in Chapter 7, Arrays and Vectors.

PREV PREV NEXT NEXT

[Page 77 (continued)]

3.4. Defining a Class with a Member Function

We begin with an example (Fig. 3.1) that consists of class `GradeBook`, which represents a grade book that an instructor can use to maintain student test scores, and a `main` function (lines 2025) that creates a `GradeBook` object. This is the first in a series of graduated examples leading up to a fully functional `GradeBook` class in Chapter 7, Arrays and Vectors. Function `main` uses this object and its member function to display a message on the screen welcoming the instructor to the grade-book program.

[Page 78]

Figure 3.1. Defining class `GradeBook` with a member function, creating a `GradeBook` object and calling its member function.

```
 1  // Fig. 3.1: fig03_01.cpp
 2  // Define class GradeBook with a member function displayMessage;
 3  // Create a GradeBook object and call its displayMessage function.
 4  #include <iostream>
 5  using std::cout;
 6  using std::endl;
 7
 8  // GradeBook class definition
 9  class GradeBook
10  {
11  public:
12     // function that displays a welcome message to the GradeBook user
13     void displayMessage()
14     {
15        cout << "Welcome to the Grade Book!" << endl;
16     } // end function displayMessage
17  }; // end class GradeBook
18
19  // function main begins program execution
20  int main()
21  {
22     GradeBook myGradeBook; // create a GradeBook object named myGradeBook
23     myGradeBook.displayMessage(); // call object's displayMessage function
24     return 0; // indicate successful termination
25  } // end main
```

```
Welcome to the Grade Book!
```

First we describe how to define a class and a member function. Then we explain how an object is created and how to call a member function of an object. The first few examples contain function `main` and the `GradeBook` class it uses in the same file. Later in the chapter, we introduce more sophisticated ways to structure your programs to achieve better software engineering.

Class `GradeBook`

Before function `main` (lines 2025) can create an object of class `GradeBook`, we must tell the compiler what member functions and data members belong to the class. This is known as **defining a class**. The `GradeBook` **class definition** (lines 917) contains a member function called `displayMessage` (lines 1316) that displays a message on the screen (line 15). Recall that a class is like a blueprintso we need to make an object of class `GradeBook` (line 22) and call its `displayMessage` member function (line 23) to get line 15 to execute and display the welcome message. We'll soon explain lines 2223 in detail.

The class definition begins at line 9 with the keyword `class` followed by the class name `GradeBook`. By convention, the name of a user-defined class begins with a capital letter, and for readability, each subsequent word in the class name begins with a capital letter. This capitalization style is often referred to as **camel case**, because the pattern of uppercase and lowercase letters resembles the silhouette of a camel.

[Page 79]

Every class's **body** is enclosed in a pair of left and right braces (`{` and `}`), as in lines 10 and 17. The class definition terminates with a semicolon (line 17).

Common Programming Error 3.1

Forgetting the semicolon at the end of a class definition is a syntax error.

Recall that the function `main` is always called automatically when you execute a program. Most functions do not get called automatically. As you will soon see, you must call member function `displayMessage` explicitly to tell it to perform its task.

Line 11 contains the access-specifier label `public:`. The keyword **public** is called an **access specifier**. Lines 1316 define member function `displayMessage`. This member function appears after access specifier `public:` to indicate that the function is "available to the public"that is, it can be called by other functions in the program and by member functions of other classes. Access specifiers are always followed by a colon (`:`). For the remainder of the text, when we refer to the access specifier `public`, we will omit the colon as we did in this sentence. Section 3.6 introduces a second access specifier, `private` (again, we omit the colon in our discussions, but include it in our programs).

Each function in a program performs a task and may return a value when it completes its taskfor example, a function might perform a calculation, then return the result of that calculation. When you define a function, you must specify a **return type** to indicate the type of the value returned by the function when it completes its task. In line 13, keyword **void** to the left of the function name `displayMessage` is the function's return type. Return type `void` indicates that `displayMessage` will perform a task but will not return (i.e., give back) any data to its **calling function** (in this example,

`main`, as we'll see in a moment) when it completes its task. (In Fig. 3.5, you will see an example of a function that returns a value.)

The name of the member function, `displayMessage`, follows the return type. By convention, function names begin with a lowercase first letter and all subsequent words in the name begin with a capital letter. The parentheses after the member function name indicate that this is a function. An empty set of parentheses, as shown in line 13, indicates that this member function does not require additional data to perform its task. You will see an example of a member function that does require additional data in Section 3.5. Line 13 is commonly referred to as the **function header**. Every function's body is delimited by left and right braces (`{` and `}`), as in lines 14 and 16.

The body of a function contains statements that perform the function's task. In this case, member function `displayMessage` contains one statement (line 15) that displays the message `"Welcome to the Grade Book!"`. After this statement executes, the function has completed its task.

Common Programming Error 3.2

Returning a value from a function whose return type has been declared `void` is a compilation error.

Common Programming Error 3.3

Defining a function inside another function is a syntax error.

Testing Class GradeBook

Next, we'd like to use class `GradeBook` in a program. As you learned in Chapter 2, function `main` begins the execution of every program. Lines 2025 of Fig. 3.1 contain the `main` function that will control our program's execution.

In this program, we'd like to call class `GradeBook`'s `displayMessage` member function to display the welcome message. Typically, you cannot call a member function of a class until you create an object of that class. (As you will learn in Section 10.7, `static` member functions are an exception.) Line 22 creates an object of class `GradeBook` called `myGradeBook`. Note that the variable's type is `GradeBook`the class we defined in lines 917. When we declare variables of type `int`, as we did in Chapter 2, the compiler knows what `int` isit's a fundamental type. When we write line 22, however, the compiler does not automatically know what type `GradeBook` isit's a user-defined type. Thus, we must tell the compiler what `GradeBook` is by including the class definition, as we did in lines 917. If we omitted these lines, the compiler would issue an error message (such as "`'GradeBook': undeclared identifier`" in Microsoft Visual C++ .NET or "`'GradeBook': undeclared`" in GNU C++). Each new class you create becomes a new type that can be used to create objects. Programmers can define new class types as needed; this is one reason why C++ is known as an **extensible language**.

Line 23 calls the member function `displayMessage` (defined in lines 1316) using variable `myGradeBook` followed by the **dot operator (.)**, the function name `displayMessage` and an empty set of parentheses. This call causes the `displayMessage` function to perform its task. At the beginning of line 23, "`myGradeBook.`" indicates that `main` should use the `GradeBook` object that was created in line 22. The empty parentheses in line 13 indicate that member function `displayMessage` does not require additional data to perform its task. (In Section 3.5, you'll see how to pass data to a function.) When `displayMessage` completes its task, function `main` continues executing at line 24, which indicates that `main` performed its tasks successfully. This is the end of `main`, so the program terminates.

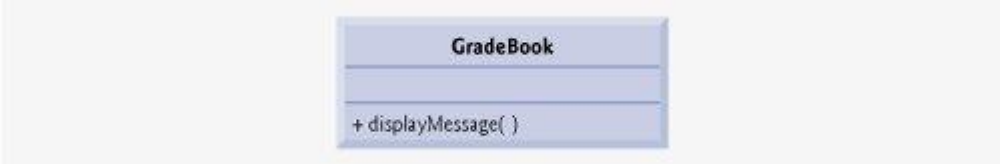
UML Class Diagram for Class `GradeBook`

Recall from Section 1.17 that the UML is a graphical language used by programmers to represent their object-oriented systems in a standardized manner. In the UML, each class is modeled in a class diagram as a rectangle with three compartments. Figure 3.2 presents a **UML class diagram** for class `GradeBook` of Fig. 3.1. The top compartment contains the name of the class, centered horizontally and in boldface type. The middle compartment contains the class's attributes, which correspond to data members in C++. In Fig. 3.2 the middle compartment is empty, because the version of class `GradeBook` in Fig. 3.1 does not have any attributes. (Section 3.6 presents a version of the `GradeBook` class that does have an attribute.) The bottom compartment contains the class's operations, which correspond to member functions in C++. The UML models operations by listing the operation name followed by a set of parentheses. The class `GradeBook` has only one member function, `displayMessage`, so the bottom compartment of Fig. 3.2 lists one operation with this name. Member function `displayMessage` does not require additional information to perform its tasks, so the parentheses following `displayMessage` in the class diagram are empty, just as they are in the member function's header in line 13 of Fig. 3.1. The plus sign (+) in front of the operation name indicates that `displayMessage` is a public operation in the UML (i.e., a `public` member function in C++). We frequently use UML class diagrams to summarize class attributes and operations.

[Page 81]

Figure 3.2. UML class diagram indicating that class `GradeBook` has a public `displayMessage` operation.
(This item is displayed on page 80 in the print version)

[View full size image]



PREV PREV NEXT NEXT

[Page 81 (continued)]

3.5. Defining a Member Function with a Parameter

In our car analogy from Section 3.2, we mentioned that pressing a car's gas pedal sends a message to the car to perform a taskmake the car go faster. But how fast should the car accelerate? As you

know, the farther down you press the pedal, the faster the car accelerates. So the message to the car includes both the task to perform and additional information that helps the car perform the task. This additional information is known as a **parameter**the value of the parameter helps the car determine how fast to accelerate. Similarly, a member function can require one or more parameters that represent additional data it needs to perform its task. A function call supplies valuescalled **arguments**for each of the function's parameters. For example, to make a deposit into a bank account, suppose a `deposit` member function of an `Account` class specifies a parameter that represents the deposit amount. When the `deposit` member function is called, an argument value representing the deposit amount is copied to the member function's parameter. The member function then adds that amount to the account balance.

Defining and Testing Class `GradeBook`

Our next example (Fig. 3.3) redefines class `GradeBook` (lines 1423) with a `displayMessage` member function (lines 1822) that displays the course name as part of the welcome message. The new `displayMessage` member function requires a parameter (`courseName` in line 18) that represents the course name to output.

Figure 3.3. Defining class `GradeBook` with a member function that takes a parameter.
(This item is displayed on page 82 in the print version)

```
 1  // Fig. 3.3: fig03_03.cpp
 2  // Define class GradeBook with a member function that takes a parameter;
 3  // Create a GradeBook object and call its displayMessage function.
 4  #include <iostream>
 5  using std::cout;
 6  using std::cin;
 7  using std::endl;
 8
 9  #include <string> // program uses C++ standard string class
10  using std::string;
11  using std::getline;
12
13  // GradeBook class definition
14  class GradeBook
15  {
16  public:
17     // function that displays a welcome message to the GradeBook user
18     void displayMessage( string courseName )
19     {
20        cout << "Welcome to the grade book for\n" << courseName << "!"
21           << endl;
22     } // end function displayMessage
23  }; // end class GradeBook
24
25  // function main begins program execution
26  int main()
27  {
28     string nameOfCourse; // string of characters to store the course name
29     GradeBook myGradeBook; // create a GradeBook object named myGradeBook
30
31     // prompt for and input course name
32     cout << "Please enter the course name:" << endl;
33     getline( cin, nameOfCourse ); // read a course name with blanks
34     cout << endl; // output a blank line
35
36     // call myGradeBook's displayMessage function
37     // and pass nameOfCourse as an argument
38     myGradeBook.displayMessage( nameOfCourse );
39     return 0; // indicate successful termination
```

```
40  } // end main
```

```
Please enter the course name:
CS101 Introduction to C++ Programming

Welcome to the grade book for
CS101 Introduction to C++ Programming!
```

Before discussing the new features of class `GradeBook`, let's see how the new class is used in `main` (lines 2640). Line 28 creates a variable of type **string** called `nameOfCourse` that will be used to store the course name entered by the user. A variable of type `string` represents a string of characters such as "`CS101 Introduction to C++ Programming`". A `string` is actually an object of the C++ Standard Library class `string`. This class is defined in **header file <string>**, and the name `string`, like `cout`, belongs to namespace `std`. To enable line 28 to compile, line 9 includes the `<string>` header file. Note that the `using` declaration in line 10 allows us to simply write `string` in line 28 rather than `std::string`. For now, you can think of `string` variables like variables of other types such as `int`. You will learn about additional `string` capabilities in Section 3.10.

Line 29 creates an object of class `GradeBook` named `myGradeBook`. Line 32 prompts the user to enter a course name. Line 33 reads the name from the user and assigns it to the `nameOfCourse` variable, using the library function **getline** to perform the input. Before we explain this line of code, let's explain why we cannot simply write

```
cin >> nameOfCourse;
```

[Page 82]

to obtain the course name. In our sample program execution, we use the course name "`CS101 Introduction to C++ Programming`," which contains multiple words. (Recall that we highlight user-supplied input in bold.) When `cin` is used with the stream extraction operator, it reads characters until the first white-space character is reached. Thus, only "`CS101`" would be read by the preceding statement. The rest of the course name would have to be read by subsequent input operations.

[Page 83]

In this example, we'd like the user to type the complete course name and press Enter to submit it to the program, and we'd like to store the entire course name in the `string` variable `nameOfCourse`. The function call `getline( cin, nameOfCourse )` in line 33 reads characters (including the space characters that separate the words in the input) from the standard input stream object `cin` (i.e., the keyboard) until the newline character is encountered, places the characters in the `string` variable `nameOfCourse` and discards the newline character. Note that when you press Enter while typing program input, a newline is inserted in the input stream. Also note that the `<string>` header file must be included in the program to use function `getline` and that the name `getline` belongs to namespace `std`.

Line 38 calls `myGradeBook`'s `displayMessage` member function. The `nameOfCourse` variable in parentheses is the argument that is passed to member function `displayMessage` so that it can perform its task. The value of variable `nameOfCourse` in `main` becomes the value of member function `displayMessage`'s parameter `courseName` in line 18. When you execute this program, notice that member function `displayMessage` outputs as part of the welcome message the course name you type (in our sample execution, `CS101 Introduction to C++ Programming`).

More on Arguments and Parameters

To specify that a function requires data to perform its task, you place additional information in the function's **parameter list**, which is located in the parentheses following the function name. The parameter list may contain any number of parameters, including none at all (represented by empty parentheses as in Fig. 3.1, line 13) to indicate that a function does not require any parameters. Member function `displayMessage`'s parameter list (Fig. 3.3, line 18) declares that the function requires one parameter. Each parameter should specify a type and an identifier. In this case, the type `string` and the identifier `courseName` indicate that member function `displayMessage` requires a `string` to perform its task. The member function body uses the parameter `courseName` to access the value that is passed to the function in the function call (line 38 in main). Lines 2021 display parameter `courseName`'s value as part of the welcome message. Note that the parameter variable's name (line 18) can be the same as or different from the argument variable's name (line 38)you'll learn why in Chapter 6, Functions and an Introduction to Recursion.

A function can specify multiple parameters by separating each parameter from the next with a comma (we'll see an example in Figs. 6.46.5). The number and order of arguments in a function call must match the number and order of parameters in the parameter list of the called member function's header. Also, the argument types in the function call must match the types of the corresponding parameters in the function header. (As you will learn in subsequent chapters, an argument's type and its corresponding parameter's type need not always be identical, but they must be "consistent.") In our example, the one `string` argument in the function call (i.e., `nameOfCourse`) exactly matches the one `string` parameter in the member-function definition (i.e., `courseName`).

Common Programming Error 3.4

Placing a semicolon after the right parenthesis enclosing the parameter list of a function definition is a syntax error.

[Page 84]

Common Programming Error 3.5

Defining a function parameter again as a local variable in the function is a compilation error.

Good Programming Practice 3.1

To avoid ambiguity, do not use the same names for the arguments passed to a function and the corresponding parameters in the function definition.

Good Programming Practice 3.2

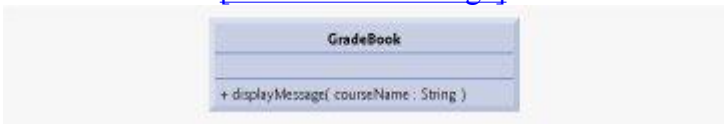
Choosing meaningful function names and meaningful parameter names makes programs more readable and helps avoid excessive use of comments.

Updated UML Class Diagram for Class `GradeBook`

The UML class diagram of Fig. 3.4 models class `GradeBook` of Fig. 3.3. Like the class `GradeBook` defined in Fig. 3.1, this `GradeBook` class contains `public` member function `displayMessage`. However, this version of `displayMessage` has a parameter. The UML models a parameter by listing the parameter name, followed by a colon and the parameter type in the parentheses following the operation name. The UML has its own data types similar to those of C++. The UML is language-independentit is used with many different programming languagesso its terminology does not exactly match that of C++. For example, the UML type `String` corresponds to the C++ type `string`. Member function `displayMessage` of class `GradeBook` (Fig. 3.3; lines 1822) has a `string` parameter named `courseName`, so Fig. 3.4 lists `courseName : String` between the parentheses following the operation name `displayMessage`. Note that this version of the `GradeBook` class still does not have any data members.

Figure 3.4. UML class diagram indicating that class `GradeBook` has a `displayMessage` operation with a `courseName` parameter of UML type `String`.

[View full size image]



PREV PREV NEXT NEXT

[Page 84 (continued)]

3.6. Data Members, set Functions and get Functions

In Chapter 2, we declared all of a program's variables in its `main` function. Variables declared in a function definition's body are known as **local variables** and can be used only from the line of their declaration in the function to the immediately following closing right brace (`}`) of the function definition. A local variable must be declared before it can be used in a function. A local variable cannot be accessed outside the function in which it is declared. When a function terminates, the values of its local variables are lost. (You will see an exception to this in Chapter 6 when we discuss `static` local variables.) Recall from Section 3.2 that an object has attributes that are carried with it as it is used in a program. Such attributes exist throughout the life of the object.

A class normally consists of one or more member functions that manipulate the attributes that belong to a particular object of the class. Attributes are represented as variables in a class definition. Such variables are called **data members** and are declared inside a class definition but outside the bodies of the class's member-function definitions. Each object of a class maintains its own copy of its attributes in memory. The example in this section demonstrates a `GradeBook` class that contains a `courseName` data member to represent a particular `GradeBook` object's course name.

`GradeBook` Class with a Data Member, a set Function and a get Function

In our next example, class `GradeBook` (Fig. 3.5) maintains the course name as a data member so that it can be used or modified at any time during a program's execution. The class contains member functions `setCourseName`, `getCourseName` and `displayMessage`. Member function `setCourseName` stores a course name in a `GradeBook` data membermember function `getCourseName` obtains a `GradeBook`'s course name from that data member. Member function `displayMessage`which now specifies no parametersstill displays a welcome message that includes the course name. However, as you will see, the function now obtains the course name by calling another function in the same class`getCourseName`.

Figure 3.5. Defining and testing class `GradeBook` with a data member and set and get functions.
(This item is displayed on pages 85 - 86 in the print version)

```
1  // Fig. 3.5: fig03_05.cpp
2  // Define class GradeBook that contains a courseName data member
3  // and member functions to set and get its value;
4  // Create and manipulate a GradeBook object with these functions.
5  #include <iostream>
6  using std::cout;
7  using std::cin;
8  using std::endl;
9
10 #include <string> // program uses C++ standard string class
11 using std::string;
12 using std::getline;
13
14 // GradeBook class definition
15 class GradeBook
16 {
17 public:
18    // function that sets the course name
19    void setCourseName( string name )
20    {
21       courseName = name; // store the course name in the object
22    } // end function setCourseName
23
24    // function that gets the course name
25    string getCourseName()
26    {
27       return courseName; // return the object's courseName
28    } // end function getCourseName
29
30    // function that displays a welcome message
31    void displayMessage()
32    {
33       // this statement calls getCourseName to get the
34       // name of the course this GradeBook represents
35       cout << "Welcome to the grade book for\n" << getCourseName() << "!"
36          << endl;
37    } // end function displayMessage
38 private:
```

```
39     string courseName; // course name for this GradeBook
40  }; // end class GradeBook
41
42  // function main begins program execution
43  int main()
44  {
45     string nameOfCourse; // string of characters to store the course name
46     GradeBook myGradeBook; // create a GradeBook object named myGradeBook
47
48     // display initial value of courseName
49     cout << "Initial course name is: " << myGradeBook.getCourseName()
50        << endl;
51
52     // prompt for, input and set course name
53     cout << "\nPlease enter the course name:" << endl;
54     getline( cin, nameOfCourse ); // read a course name with blanks
55     myGradeBook.setCourseName( nameOfCourse ); // set the course name
56
57     cout << endl; // outputs a blank line
58     myGradeBook.displayMessage(); // display message with new course name
59     return 0; // indicate successful termination
60  } // end main
```

```
Initial course name is:

Please enter the course name:
CS101 Introduction to C++ Programming

Welcome to the grade book for
CS101 Introduction to C++ Programming!
```

[Page 86]

Good Programming Practice 3.3

Place a blank line between member-function definitions to enhance program readability.

A typical instructor teaches more than one course, each with its own course name. Line 39 declares that `courseName` is a variable of type `string`. Because the variable is declared in the class definition (lines 1540) but outside the bodies of the class's member-function definitions (lines 1922, 2528 and 3137), line 39 is a declaration for a data member. Every instance (i.e., object) of class `GradeBook` contains one copy of each of the class's data members. For example, if there are two `GradeBook` objects, each object has its own copy of `courseName` (one per object), as we'll see in the example of Fig. 3.7. A benefit of making `courseName` a data member is that all the member functions of the class (in this case, `GradeBook`) can manipulate any data members that appear in the class definition (in this case, `courseName`).

[Page 87]

Access Specifiers `public` and `private`

Most data member declarations appear after the access-specifier label **private:** (line 38). Like `public`, keyword `private` is an access specifier. Variables or functions declared after access specifier `private` (and before the next access specifier) are accessible only to member functions of the class for which they are declared. Thus, data member `courseName` can be used only in member functions `setCourseName`, `getCourseName` and `displayMessage` of (every object of) class `GradeBook`. Data member `courseName`, because it is `private`, cannot be accessed by functions outside the class (such as `main`) or by member functions of other classes in the program. Attempting to access data member `courseName` in one of these program locations with an expression such as `myGradeBook.courseName` would result in a compilation error containing a message similar to

```
cannot access private member declared in class 'GradeBook'
```

Software Engineering Observation 3.1

As a rule of thumb, data members should be declared `private` and member functions should be declared `public`. (We will see that it is appropriate to declare certain member functions `private`, if they are to be accessed only by other member functions of the class.)

Common Programming Error 3.6

An attempt by a function, which is not a member of a particular class (or a `friend` of that class, as we will see in Chapter 10), to access a private member of that class is a compilation error.

The default access for class members is `private` so all members after the class header and before the first access specifier are `private`. The access specifiers `public` and `private` may be repeated, but this is unnecessary and can be confusing.

Good Programming Practice 3.4

Despite the fact that the `public` and `private` access specifiers may be repeated and intermixed, list all the `public` members of a class first in one group and then list all the `private` members in another group. This focuses the client's attention on the class's `public` interface, rather than on the class's implementation.

Good Programming Practice 3.5

If you choose to list the `private` members first in a class definition, explicitly use the `private` access specifier despite the fact that `private` is assumed by default. This improves program clarity.

Declaring data members with access specifier `private` is known as **data hiding**. When a program creates (instantiates) an object of class `GradeBook`, data member `courseName` is encapsulated (hidden) in the object and can be accessed only by member functions of the object's class. In class `GradeBook`, member functions `setCourseName` and `getCourseName` manipulate the data member `courseName` directly (and `displayMessage` could do so if necessary).

[Page 88]

Software Engineering Observation 3.2

We will learn in Chapter 10, Classes: A Deeper Look, Part 2, that functions and classes declared by a class to be `friend`s can access the `private` members of the class.

Error-Prevention Tip 3.1

Making the data members of a class `private` and the member functions of the class `public` facilitates debugging because problems with data manipulations are localized to either the class's member functions or the `friend`s of the class.

Member Functions `setCourseName` and `getCourseName`

Member function `setCourseName` (defined in lines 1922) does not return any data when it completes its task, so its return type is `void`. The member function receives one parameter`name`which represents the course name that will be passed to it as an argument (as we will see in line 55 of `main`). Line 21 assigns `name` to data member `courseName`. In this example, `setCourseName` does not attempt to validate the course namei.e., the function does not check that the course name adheres to any particular format or follows any other rules regarding what a "valid" course name looks like. Suppose, for instance, that a university can print student transcripts containing course names of only 25 characters or fewer. In this case, we might want class `GradeBook` to ensure that its data member `courseName` never contains more than 25 characters. We discuss basic validation techniques in Section 3.10.

Member function `getCourseName` (defined in lines 2528) returns a particular `GradeBook` object's `courseName`. The member function has an empty parameter list, so it does not require additional data to perform its task. The function specifies that it returns a `string`. When a function that specifies a return type other than `void` is called and completes its task, the function returns a result to its calling function. For example, when you go to an automated teller machine (ATM) and request your account balance, you expect the ATM to give you back a value that represents your balance. Similarly, when a statement calls member function `getCourseName` on a `GradeBook` object, the statement expects to receive the `GradeBook`'s course name (in this case, a `string`, as specified by the function's return type). If you have a function `square` that returns the square of its argument, the statement

```
int result = square( 2 );
```

returns `4` from function `square` and initializes the variable `result` with the value `4`. If you have a function `maximum` that returns the largest of three integer arguments, the statement

```
int biggest = maximum( 27, 114, 51 );
```

returns 114 from function maximum and initializes variable biggest with the value 114.

Common Programming Error 3.7

Forgetting to return a value from a function that is supposed to return a value is a compilation error.

Note that the statements at lines 21 and 27 each use variable courseName (line 39) even though it was not declared in any of the member functions. We can use courseName in the member functions of class GradeBook because courseName is a data member of the class. Also note that the order in which member functions are defined does not determine when they are called at execution time. So member function getCourseName could be defined before member function setCourseName.

[Page 89]

Member Function displayMessage

Member function displayMessage (lines 3137) does not return any data when it completes its task, so its return type is void. The function does not receive parameters, so its parameter list is empty. Lines 3536 output a welcome message that includes the value of data member courseName. Line 35 calls member function getCourseName to obtain the value of courseName. Note that member function displayMessage could also access data member courseName directly, just as member functions setCourseName and getCourseName do. We explain shortly why we choose to call member function getCourseName to obtain the value of courseName.

Testing Class GradeBook

The main function (lines 4360) creates one object of class GradeBook and uses each of its member functions. Line 46 creates a GradeBook object named myGradeBook. Lines 4950 display the initial course name by calling the object's getCourseName member function. Note that the first line of the output does not show a course name, because the object's courseName data member (i.e., a string) is initially emptyby default, the initial value of a string is the so-called **empty string**, i.e., a string that does not contain any characters. Nothing appears on the screen when an empty string is displayed.

Line 53 prompts the user to enter a course name. Local string variable nameOfCourse (declared in line 45) is set to the course name entered by the user, which is obtained by the call to the getline function (line 54). Line 55 calls object myGradeBook's setCourseName member function and supplies nameOfCourse as the function's argument. When the function is called, the argument's value is copied to parameter name (line 19) of member function setCourseName (lines 1922). Then the parameter's value is assigned to data member courseName (line 21). Line 57 skips a line in the output; then line 58 calls object myGradeBook's displayMessage member function to display the welcome message containing the course name.

Software Engineering with Set and Get Functions

A class's private data members can be manipulated only by member functions of that class (and by

"friends" of the class, as we will see in Chapter 10, Classes: A Deeper Look, Part 2). So a client of an objectthat is, any class or function that calls the object's member functions from outside the objectcalls the class's `public` member functions to request the class's services for particular objects of the class. This is why the statements in function `main` (Fig. 3.5, lines 4360) call member functions `setCourseName`, `getCourseName` and `displayMessage` on a `GradeBook` object. Classes often provide `public` member functions to allow clients of the class to set (i.e., assign values to) or get (i.e., obtain the values of) `private` data members. The names of these member functions need not begin with set or get, but this naming convention is common. In this example, the member function that sets the `courseName` data member is called `setCourseName`, and the member function that gets the value of the `courseName` data member is called `getCourseName`. Note that set functions are also sometimes called **mutators** (because they mutate, or change, values), and get functions are also sometimes called **accessors** (because they access values).

Recall that declaring data members with access specifier `private` enforces data hiding. Providing `public` set and get functions allows clients of a class to access the hidden data, but only indirectly. The client knows that it is attempting to modify or obtain an object's data, but the client does not know how the object performs these operations. In some cases, a class may internally represent a piece of data one way, but expose that data to clients in a different way. For example, suppose a `Clock` class represents the time of day as a `private int` data member `time` that stores the number of seconds since midnight. However, when a client calls a `Clock` object's `getTime` member function, the object could return the time with hours, minutes and seconds in a `string` in the format `"HH:MM:SS"`. Similarly, suppose the `Clock` class provides a set function named `setTime` that takes a `string` parameter in the `"HH:MM:SS"` format. Using `string` capabilities presented in Chapter 18, the `setTime` function could convert this `string` to a number of seconds, which the function stores in its `private` data member. The set function could also check that the value it receives represents a valid time (e.g., `"12:30:45"` is valid but `"42:85:70"` is not). The set and get functions allow a client to interact with an object, but the object's `private` data remains safely encapsulated (i.e., hidden) in the object itself.

[Page 90]

The set and get functions of a class also should be used by other member functions within the class to manipulate the class's `private` data, although these member functions can access the `private` data directly. In Fig. 3.5, member functions `setCourseName` and `getCourseName` are `public` member functions, so they are accessible to clients of the class, as well as to the class itself. Member function `displayMessage` calls member function `getCourseName` to obtain the value of data member `courseName` for display purposes, even though `displayMessage` can access `courseName` directlyaccessing a data member via its get function creates a better, more robust class (i.e., a class that is easier to maintain and less likely to stop working). If we decide to change the data member `courseName` in some way, the `displayMessage` definition will not require modificationonly the bodies of the get and set functions that directly manipulate the data member will need to change. For example, suppose we decide that we want to represent the course name as two separate data members`courseNumber` (e.g., `"CS101"`) and `courseTitle` (e.g., `"Introduction to C++ Programming"`). Member function `displayMessage` can still issue a single call to member function `getCourseName` to obtain the full course to display as part of the welcome message. In this case, `getCourseName` would need to build and return a `string` containing the `courseNumber` followed by the `courseTitle`. Member function `displayMessage` would continue to display the complete course title `"CS101 Introduction to C++ Programming,"` because it is unaffected by the change to the class's data members. The benefits of calling a set function from another member function of a class will become clear when we discuss validation in Section 3.10.

Good Programming Practice 3.6

Always try to localize the effects of changes to a class's data members by accessing and manipulating the data members through their get and set functions. Changes to the name of a data member or the data type used to store a data member then affect only the corresponding get and set functions, but not the callers of those functions.

Software Engineering Observation 3.3

It is important to write programs that are understandable and easy to maintain. Change is the rule rather than the exception. Programmers should anticipate that their code will be modified.

Software Engineering Observation 3.4

The class designer need not provide set or get functions for each `private` data item; these capabilities should be provided only when appropriate. If a service is useful to the client code, that service should typically be provided in the class's `public` interface.

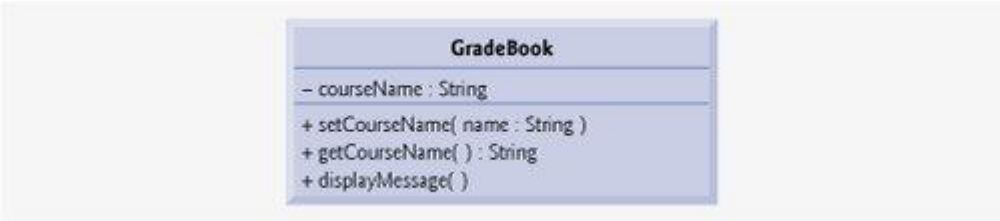
[Page 91]

`GradeBook`'s UML Class Diagram with a Data Member and set and get Functions

Figure 3.6 contains an updated UML class diagram for the version of class `GradeBook` in Fig. 3.5. This diagram models class `GradeBook`'s data member `courseName` as an attribute in the middle compartment of the class. The UML represents data members as attributes by listing the attribute name, followed by a colon and the attribute type. The UML type of attribute `courseName` is `String`, which corresponds to `string` in C++. Data member `courseName` is `private` in C++, so the class diagram lists a minus sign (-) in front of the corresponding attribute's name. The minus sign in the UML is equivalent to the `private` access specifier in C++. Class `GradeBook` contains three `public` member functions, so the class diagram lists three operations in the third compartment. Recall that the plus (+) sign before each operation name indicates that the operation is `public` in C++. Operation `setCourseName` has a `String` parameter called `name`. The UML indicates the return type of an operation by placing a colon and the return type after the parentheses following the operation name. Member function `getCourseName` of class `GradeBook` (Fig. 3.5) has a `string` return type in C++, so the class diagram shows a `String` return type in the UML. Note that operations `setCourseName` and `displayMessage` do not return values (i.e., they return `void`), so the UML class diagram does not specify a return type after the parentheses of these operations. The UML does not use `void` as C++ does when a function does not return a value.

Figure 3.6. UML class diagram for class `GradeBook` with a private `courseName` attribute and public operations `setCourseName`, `getCourseName` and `displayMessage`.

[View full size image]



PREV PREV NEXT NEXT

[Page 91 (continued)]

3.7. Initializing Objects with Constructors

As mentioned in Section 3.6, when an object of class `GradeBook` (Fig. 3.5) is created, its data member `courseName` is initialized to the empty string by default. What if you want to provide a course name when you create a `GradeBook` object? Each class you declare can provide a **constructor** that can be used to initialize an object of the class when the object is created. A constructor is a special member function that must be defined with the same name as the class, so that the compiler can distinguish it from the class's other member functions. An important difference between constructors and other functions is that constructors cannot return values, so they cannot specify a return type (not even `void`). Normally, constructors are declared `public`. The term "constructor" is often abbreviated as "ctor" in the literaturewe prefer not to use this abbreviation.

C++ requires a constructor call for each object that is created, which helps ensure that the object is initialized properly before it is used in a programthe constructor call occurs implicitly when the object is created. In any class that does not explicitly include a constructor, the compiler provides a **default constructor**that is, a constructor with no parameters. For example, when line 46 of Fig. 3.5 creates a `GradeBook` object, the default constructor is called, because the declaration of `myGradeBook` does not specify any constructor arguments. The default constructor provided by the compiler creates a `GradeBook` object without giving any initial values to the object's data members. [Note: For data members that are objects of other classes, the default constructor implicitly calls each data member's default constructor to ensure that the data member is initialized properly. In fact, this is why the `string` data member `courseName` (in Fig. 3.5) was initialized to the empty stringthe default constructor for class `string` sets the `string`'s value to the empty string. In Section 10.3, you will learn more about initializing data members that are objects of other classes.]

[Page 92]

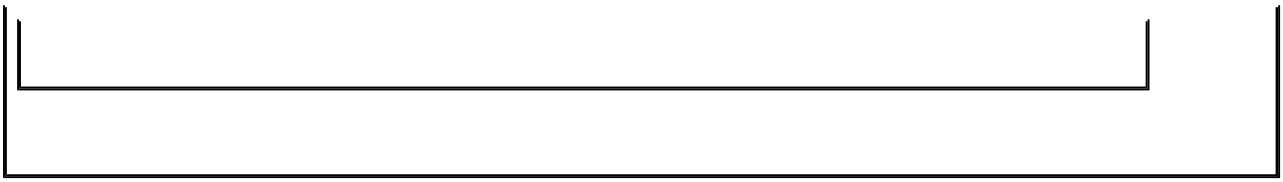
In the example of Fig. 3.7, we specify a course name for a `GradeBook` object when the object is created (line 49). In this case, the argument `"CS101 Introduction to C++ Programming"` is passed to the `GradeBook` object's constructor (lines 1720) and used to initialize the `courseName`. Figure 3.7 defines a modified `GradeBook` class containing a constructor with a `string` parameter that receives the initial course name.

Figure 3.7. Instantiating multiple objects of the `GradeBook` class and using the `GradeBook` constructor to specify the course name when each `GradeBook` object is created.

(This item is displayed on pages 92 - 93 in the print version)

```
1  // Fig. 3.7: fig03_07.cpp
2  // Instantiating multiple objects of the GradeBook class and using
3  // the GradeBook constructor to specify the course name
4  // when each GradeBook object is created.
5  #include <iostream>
6  using std::cout;
7  using std::endl;
8
9  #include <string> // program uses C++ standard string class
10 using std::string;
11
12 // GradeBook class definition
13 class GradeBook
14 {
15 public:
16    // constructor initializes courseName with string supplied as argument
17    GradeBook( string name )
18    {
19       setCourseName( name ); // call set function to initialize courseName
20    } // end GradeBook constructor
21
22    // function to set the course name
23    void setCourseName( string name )
24    {
25       courseName = name; // store the course name in the object
26    } // end function setCourseName
27
28    // function to get the course name
29    string getCourseName()
30    {
31       return courseName; // return object's courseName
32    } // end function getCourseName
33
34    // display a welcome message to the GradeBook user
35    void displayMessage()
36    {
37       // call getCourseName to get the courseName
38       cout << "Welcome to the grade book for\n" << getCourseName()
39          << "!" << endl;
40    } // end function displayMessage
41 private:
42    string courseName; // course name for this GradeBook
43 }; // end class GradeBook
44
45 // function main begins program execution
46 int main()
47 {
48    // create two GradeBook objects
49    GradeBook gradeBook1( "CS101 Introduction to C++ Programming" );
50    GradeBook gradeBook2( "CS102 Data Structures in C++" );
51
52    // display initial value of courseName for each GradeBook
53    cout << "gradeBook1 created for course: " << gradeBook1.getCourseName()
54       << "\ngradeBook2 created for course: " << gradeBook2.getCourseName()
55       << endl;
56    return 0; // indicate successful termination
57 } // end main
```

```
gradeBook1 created for course: CS101 Introduction to C++ Programming
gradeBook2 created for course: CS102 Data Structures in C++
```



[Page 93]

Defining a Constructor

Lines 1720 of Fig. 3.7 define a constructor for class `GradeBook`. Notice that the constructor has the same name as its class, `GradeBook`. A constructor specifies in its parameter list the data it requires to perform its task. When you create a new object, you place this data in the parentheses that follow the object name (as we did in lines 4950). Line 17 indicates that class `GradeBook`'s constructor has a `string` parameter called `name`. Note that line 17 does not specify a return type, because constructors cannot return values (or even `void`).

Line 19 in the constructor's body passes the constructor's parameter `name` to member function `setCourseName`, which assigns a value to data member `courseName`. The `setCourseName` member function (lines 2326) simply assigns its parameter `name` to the data member `courseName`, so you might be wondering why we bother making the call to `setCourseName` in line 19the constructor certainly could perform the assignment `courseName = name`. In Section 3.10, we modify `setCourseName` to perform validation (ensuring that, in this case, the `courseName` is 25 or fewer characters in length). At that point the benefits of calling `setCourseName` from the constructor will become clear. Note that both the constructor (line 17) and the `setCourseName` function (line 23) use a parameter called `name`. You can use the same parameter names in different functions because the parameters are local to each function; they do not interfere with one another.

[Page 94]

Testing Class `GradeBook`

Lines 4657 of Fig. 3.7 define the `main` function that tests class `GradeBook` and demonstrates initializing `GradeBook` objects using a constructor. Line 49 in function `main` creates and initializes a `GradeBook` object called `gradeBook1`. When this line executes, the `GradeBook` constructor (lines 1720) is called (implicitly by C++) with the argument `"CS101 Introduction to C++ Programming"` to initialize `gradeBook1`'s course name. Line 50 repeats this process for the `GradeBook` object called `gradeBook2`, this time passing the argument `"CS102 Data Structures in C++"` to initialize `gradeBook2`'s course name. Lines 5354 use each object's `getCourseName` member function to obtain the course names and show that they were indeed initialized when the objects were created. The output confirms that each `GradeBook` object maintains its own copy of data member `courseName`.

Two Ways to Provide a Default Constructor for a Class

Any constructor that takes no arguments is called a default constructor. A class gets a default constructor in one of two ways:

1. The compiler implicitly creates a default constructor in a class that does not define a constructor. Such a default constructor does not initialize the class's data members, but does call the default constructor for each data member that is an object of another class. [Note: An uninitialized variable typically contains a "garbage" value (e.g., an uninitialized `int` variable

might contain `-858993460`, which is likely to be an incorrect value for that variable in most programs).]

2. The programmer explicitly defines a constructor that takes no arguments. Such a default constructor will perform the initialization specified by the programmer and will call the default constructor for each data member that is an object of another class.

If the programmer defines a constructor with arguments, C++ will not implicitly create a default constructor for that class. Note that for each version of class `GradeBook` in Fig. 3.1, Fig. 3.3 and Fig. 3.5 the compiler implicitly defined a default constructor.

Error-Prevention Tip 3.2

Unless no initialization of your class's data members is necessary (almost never), provide a constructor to ensure that your class's data members are initialized with meaningful values when each new object of your class is created.

Software Engineering Observation 3.5

Data members can be initialized in a constructor of the class or their values may be set later after the object is created. However, it is a good software engineering practice to ensure that an object is fully initialized before the client code invokes the object's member functions. In general, you should not rely on the client code to ensure that an object gets initialized properly.

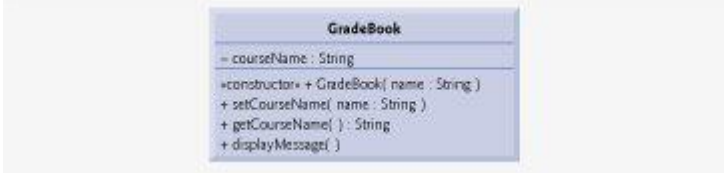
Adding the Constructor to Class `GradeBook`'s UML Class Diagram

The UML class diagram of Fig. 3.8 models class `GradeBook` of Fig. 3.7, which has a constructor with a `name` parameter of type `string` (represented by type `String` in the UML). Like operations, the UML models constructors in the third compartment of a class in a class diagram. To distinguish a constructor from a class's operations, the UML places the word "constructor" between guillemets (« and ») before the constructor's name. It is customary to list the class's constructor before other operations in the third compartment.

[Page 95]

Figure 3.8. UML class diagram indicating that class `GradeBook` has a constructor with a `name` parameter of UML type `String`.

[View full size image]



PREV PREV NEXT NEXT

[Page 95 (continued)]

3.8. Placing a Class in a Separate File for Reusability

We have developed class `GradeBook` as far as we need to for now from a programming perspective, so let's consider some software engineering issues. One of the benefits of creating class definitions is that, when packaged properly, our classes can be reused by programmerspotentially worldwide. For example, we can reuse C++ Standard Library type `string` in any C++ program by including the header file `<string>` in the program (and, as we will see, by being able to link to the library's object code).

Unfortunately, programmers who wish to use our `GradeBook` class cannot simply include the file from Fig. 3.7 in another program. As you learned in Chapter 2, function `main` begins the execution of every program, and every program must have exactly one `main` function. If other programmers include the code from Fig. 3.7, they get extra baggageour `main` functionand their programs will then have two `main` functions. When they attempt to compile their programs, the compiler will indicate an error because, again, each program can have only one `main` function. For example, attempting to compile a program with two `main` functions in Microsoft Visual C++ .NET produces the error

```
error C2084: function 'int main(void)' already has a body
```

when the compiler tries to compile the second `main` function it encounters. Similarly, the GNU C++ compiler produces the error

```
redefinition of 'int main()'
```

These errors indicate that a program already has a `main` function. So, placing `main` in the same file with a class definition prevents that class from being reused by other programs. In this section, we demonstrate how to make class `GradeBook` reusable by separating it into another file from the `main` function.

Header Files

Each of the previous examples in the chapter consists of a single `.cpp` file, also known as a **source-code file**, that contains a `GradeBook` class definition and a `main` function. When building an object-oriented C++ program, it is customary to define reusable source code (such as a class) in a file that by convention has a `.h` filename extensionknown as a **header file**. Programs use `#include` preprocessor directives to include header files and take advantage of reusable software components, such as type `string` provided in the C++ Standard Library and user-defined types like class `GradeBook`.

[Page 96]

In our next example, we separate the code from Fig. 3.7 into two files`GradeBook.h` (Fig. 3.9) and `fig03_10.cpp` (Fig. 3.10). As you look at the header file in Fig. 3.9, notice that it contains only the `GradeBook` class definition (lines 1141) and lines 38, which allow class `GradeBook` to use `cout`, `endl` and type `string`. The `main` function that uses class `GradeBook` is defined in the source-code file `fig03_10.cpp` (Fig. 3.10) at lines 1021. To help you prepare for the larger programs you will

encounter later in this book and in industry, we often use a separate source-code file containing function `main` to test our classes (this is called a driver program). You will soon learn how a source-code file with `main` can use the class definition found in a header file to create objects of a class.

Figure 3.9. `GradeBook` class definition.

(This item is displayed on pages 96 - 97 in the print version)

```
1  // Fig. 3.9: GradeBook.h
2  // GradeBook class definition in a separate file from main.
3  #include <iostream>
4  using std::cout;
5  using std::endl;
6
7  #include <string> // class GradeBook uses C++ standard string class
8  using std::string;
9
10 // GradeBook class definition
11 class GradeBook
12 {
13 public:
14    // constructor initializes courseName with string supplied as argument
15    GradeBook( string name )
16    {
17       setCourseName( name ); // call set function to initialize courseName
18    } // end GradeBook constructor
19
20    // function to set the course name
21    void setCourseName( string name )
22    {
23       courseName = name; // store the course name in the object
24    } // end function setCourseName
25
26    // function to get the course name
27    string getCourseName()
28    {
29       return courseName; // return object's courseName
30    } // end function getCourseName
31
32    // display a welcome message to the GradeBook user
33    void displayMessage()
34    {
35       // call getCourseName to get the courseName
36       cout << "Welcome to the grade book for\n" << getCourseName()
37          << "!" << endl;
38    } // end function displayMessage
39 private:
40    string courseName; // course name for this GradeBook
41 }; // end class GradeBook
```

Including a Header File That Contains a User-Defined Class

A header file such as `GradeBook.h` (Fig. 3.9) cannot be used to begin program execution, because it does not contain a `main` function. If you try to compile and link `GradeBook.h` by itself to create an executable application, Microsoft Visual C++ .NET will produce the linker error message:

```
error LNK2019: unresolved external symbol _main referenced in
function _mainCRTStartup
```

[Page 97]

Running GNU C++ on Linux produces a linker error message containing:

```
undefined reference to 'main'
```

This error indicates that the linker could not locate the program's `main` function. To test class `GradeBook` (defined in Fig. 3.9), you must write a separate source-code file containing a `main` function (such as Fig. 3.10) that instantiates and uses objects of the class.

Figure 3.10. Including class `GradeBook` from file `GradeBook.h` for use in `main`.

```
// Fig. 3.10: fig03_10.cpp
// Including class GradeBook from file GradeBook.h for use in main.
#include <iostream>
using std::cout;
using std::endl;

#include "GradeBook.h" // include definition of class GradeBook

// function main begins program execution
int main()
{
   // create two GradeBook objects
   GradeBook gradeBook1( "CS101 Introduction to C++ Programming" );
   GradeBook gradeBook2( "CS102 Data Structures in C++" );

   // display initial value of courseName for each GradeBook
   cout << "gradeBook1 created for course: " << gradeBook1.getCourseName()
      << "\ngradeBook2 created for course: " << gradeBook2.getCourseName()
      << endl;
   return 0; // indicate successful termination
} // end main
```

```
gradeBook1 created for course: CS101 Introduction to C++ Programming
gradeBook2 created for course: CS102 Data Structures in C++
```

Recall from Section 3.4 that, while the compiler knows what fundamental data types like `int` are, the compiler does not know what a `GradeBook` is because it is a user-defined type. In fact, the compiler does not even know the classes in the C++ Standard Library. To help it understand how to use a class, we must explicitly provide the compiler with the class's definitionthat's why, for example, to use type `string`, a program must include the `<string>` header file. This enables the compiler to determine the amount of memory that it must reserve for each object of the class and ensure that a program calls the class's member functions correctly.

[Page 98]

To create `GradeBook` objects `gradeBook1` and `gradeBook2` in lines 1314 of Fig. 3.10, the compiler must know the size of a `GradeBook` object. While objects conceptually contain data members and member functions, C++ objects typically contain only data. The compiler creates only one copy of the class's member functions and shares that copy among all the class's objects. Each object, of course, needs its own copy of the class's data members, because their contents can vary among objects (such as two different `BankAccount` objects having two different `balance` data members). The member function code, however, is not modifiable, so it can be shared among all objects of the class. Therefore, the size of an object depends on the amount of memory required to store the class's data members. By including `GradeBook.h` in line 7, we give the compiler access to the information it needs (Fig. 3.9, line 40) to determine the size of a `GradeBook` object and to determine whether objects of the class are used correctly (in lines 1314 and 1718 of Fig. 3.10).

Line 7 instructs the C++ preprocessor to replace the directive with a copy of the contents of `GradeBook.h` (i.e., the `GradeBook` class definition) before the program is compiled. When the source-code file `fig03_10.cpp` is compiled, it now contains the `GradeBook` class definition (because of the `#include`), and the compiler is able to determine how to create `GradeBook` objects and see that their member functions are called correctly. Now that the class definition is in a header file (without a `main` function), we can include that header in any program that needs to reuse our `GradeBook` class.

How Header Files Are Located

Notice that the name of the `GradeBook.h` header file in line 7 of Fig. 3.10 is enclosed in quotes (`" "`) rather than angle brackets (`< >`). Normally, a program's source-code files and user-defined header files are placed in the same directory. When the preprocessor encounters a header file name in quotes (e.g., `"GradeBook.h"`), the preprocessor attempts to locate the header file in the same directory as the file in which the `#include` directive appears. If the preprocessor cannot find the header file in that directory, it searches for it in the same location(s) as the C++ Standard Library header files. When the preprocessor encounters a header file name in angle brackets (e.g., `<iostream>`), it assumes that the header is part of the C++ Standard Library and does not look in the directory of the program that is being preprocessed.

Error-Prevention Tip 3.3

To ensure that the preprocessor can locate header files correctly, `#include` preprocessor directives should place the names of user-defined header files in quotes (e.g., `"GradeBook.h"`) and place the names of C++ Standard Library header files in angle brackets (e.g., `<iostream>`).

Additional Software Engineering Issues

Now that class `GradeBook` is defined in a header file, the class is reusable. Unfortunately, placing a class definition in a header file as in Fig. 3.9 still reveals the entire implementation of the class to the class's clients`GradeBook.h` is simply a text file that anyone can open and read. Conventional software engineering wisdom says that to use an object of a class, the client code needs to know only what member functions to call, what arguments to provide to each member function and what return type to expect from each member function. The client code does not need to know how those functions are implemented.

If client code does know how a class is implemented, the client code programmer might write client code based on the class's implementation details. Ideally, if that implementation changes, the class's clients should not have to change. Hiding the class's implementation details makes it easier to change the class's implementation while minimizing, and hopefully eliminating, changes to client code.

In Section 3.9, we show how to break up the `GradeBook` class into two files so that

1. the class is reusable
2. the clients of the class know what member functions the class provides, how to call them and what return types to expect
3. the clients do not know how the class's member functions are implemented.

PREV PREV NEXT NEXT

[Page 99 (continued)]

3.9. Separating Interface from Implementation

In the preceding section, we showed how to promote software reusability by separating a class definition from the client code (e.g., function `main`) that uses the class. We now introduce another fundamental principle of good software engineering**separating interface from implementation**.

Interface of a Class

Interfaces define and standardize the ways in which things such as people and systems interact with one another. For example, a radio's controls serve as an interface between the radio's users and its internal components. The controls allow users to perform a limited set of operations (such as changing the station, adjusting the volume, and choosing between AM and FM stations). Various radios may implement these operations differentlysome provide push buttons, some provide dials and some support voice commands. The interface specifies what operations a radio permits users to perform but does not specify how the operations are implemented inside the radio.

Similarly, the **interface of a class** describes what services a class's clients can use and how to request those services, but not how the class carries out the services. A class's interface consists of the class's `public` member functions (also known as the class's **public services**). For example, class `GradeBook`'s interface (Fig. 3.9) contains a constructor and member functions `setCourseName`, `getCourseName` and `displayMessage`. `GradeBook`'s clients (e.g., `main` in Fig. 3.10) use these functions to request the class's services. As you will soon see, you can specify a class's interface by writing a class definition that lists only the member function names, return types and parameter types.

Separating the Interface from the Implementation

In our prior examples, each class definition contained the complete definitions of the class's `public` member functions and the declarations of its `private` data members. However, it is better software engineering to define member functions outside the class definition, so that their implementation details can be hidden from the client code. This practice ensures that programmers do not write client code that depends on the class's implementation details. If they were to do so, the client code would be more likely to "break" if the class's implementation changed.

The program of Figs. 3.113.13 separates class GradeBook's interface from its implementation by splitting the class definition of Fig. 3.9 into two filesthe header file GradeBook.h (Fig. 3.11) in which class GradeBook is defined, and the source-code file GradeBook.cpp (Fig. 3.12) in which GradeBook's member functions are defined. By convention, member-function definitions are placed in a source-code file of the same base name (e.g., GradeBook) as the class's header file but with a .cpp filename extension. The source-code file fig03_13.cpp (Fig. 3.13) defines function main (the client code). The code and output of Fig. 3.13 are identical to that of Fig. 3.10. Figure 3.14 shows how this three-file program is compiled from the perspectives of the GradeBook class programmer and the client-code programmerwe will explain this figure in detail.

[Page 100]

GradeBook.h: Defining a Class's Interface with Function Prototypes

Header file GradeBook.h (Fig. 3.11) contains another version of GradeBook's class definition (lines 918). This version is similar to the one in Fig. 3.9, but the function definitions in Fig. 3.9 are replaced here with **function prototypes** (lines 1215) that describe the class's public interface without revealing the class's member function implementations. A function prototype is a declaration of a function that tells the compiler the function's name, its return type and the types of its parameters. Note that the header file still specifies the class's private data member (line 17) as well. Again, the compiler must know the data members of the class to determine how much memory to reserve for each object of the class. Including the header file GradeBook.h in the client code (line 8 of Fig. 3.13) provides the compiler with the information it needs to ensure that the client code calls the member functions of class GradeBook correctly.

The function prototype in line 12 (Fig. 3.12) indicates that the constructor requires one string parameter. Recall that constructors do not have return types, so no return type appears in the function prototype. Member function setCourseName's function prototype (line 13) indicates that setCourseName requires a string parameter and does not return a value (i.e., its return type is void). Member function getCourseName's function prototype (line 14) indicates that the function does not require parameters and returns a string. Finally, member function displayMessage's function prototype (line 15) specifies that displayMessage does not require parameters and does not return a value. These function prototypes are the same as the corresponding function headers in Fig. 3.9, except that the parameter names (which are optional in prototypes) are not included and each function prototype must end with a semicolon.

[Page 101]

Figure 3.11. GradeBook class definition containing function prototypes that specify the interface of the class.
(This item is displayed on page 100 in the print version)

```
 1  // Fig. 3.11: GradeBook.h
 2  // GradeBook class definition. This file presents GradeBook's public
 3  // interface without revealing the implementations of GradeBook's member
 4  // functions, which are defined in GradeBook.cpp.
 5  #include <string> // class GradeBook uses C++ standard string class
 6  using std::string;
 7
 8  // GradeBook class definition
 9  class GradeBook
10  {
11  public:
12     GradeBook( string ); // constructor that initializes courseName
```

```
13     void setCourseName( string ); // function that sets the course name
14     string getCourseName(); // function that gets the course name
15     void displayMessage(); // function that displays a welcome message
16  private:
17     string courseName; // course name for this GradeBook
18  }; // end class GradeBook
```

Common Programming Error 3.8

Forgetting the semicolon at the end of a function prototype is a syntax error.

Good Programming Practice 3.7

Although parameter names in function prototypes are optional (they are ignored by the compiler), many programmers use these names for documentation purposes.

Error-Prevention Tip 3.4

Parameter names in a function prototype (which, again, are ignored by the compiler) can be misleading if wrong or confusing names are used. For this reason, many programmers create function prototypes by copying the first line of the corresponding function definitions (when the source code for the functions is available), then appending a semicolon to the end of each prototype.

`GradeBook.cpp`: Defining Member Functions in a Separate Source-Code File

Source-code file `GradeBook.cpp` (Fig. 3.12) defines class `GradeBook`'s member functions, which were declared in lines 1215 of Fig. 3.11. The member-function definitions appear in lines 1134 and are nearly identical to the member-function definitions in lines 1538 of Fig. 3.9.

Notice that each member function name in the function headers (lines 11, 17, 23 and 29) is preceded by the class name and `::`, which is known as the **binary scope resolution operator**. This "ties" each member function to the (now separate) `GradeBook` class definition, which declares the class's member functions and data members. Without "`GradeBook::`" preceding each function name, these functions would not be recognized by the compiler as member functions of class `GradeBook`the compiler would consider them "free" or "loose" functions, like `main`. Such functions cannot access `GradeBook`'s `private` data or call the class's member functions, without specifying an object. So, the compiler would not be able to compile these functions. For example, lines 19 and 25 that access variable `courseName` would cause compilation errors because `courseName` is not declared as a local variable in each functionthe compiler would not know that `courseName` is already declared as a data member of class `GradeBook`.

Common Programming Error 3.9

When defining a class's member functions outside that class, omitting the class name and binary scope resolution operator (`::`) preceding the function names causes compilation errors.

To indicate that the member functions in `GradeBook.cpp` are part of class `GradeBook`, we must first include the `GradeBook.h` header file (line 8 of Fig. 3.12). This allows us to access the class name `GradeBook` in the `GradeBook.cpp` file. When compiling `GradeBook.cpp`, the compiler uses the information in `GradeBook.h` to ensure that

---

[Page 102]

1. the first line of each member function (lines 11, 17, 23 and 29) matches its prototype in the `GradeBook.h` filefor example, the compiler ensures that `getCourseName` accepts no parameters and returns a `string`.

2. each member function knows about the class's data members and other member functionsfor example, lines 19 and 25 can access variable `courseName` because it is declared in `GradeBook.h` as a data member of class `GradeBook`, and lines 13 and 32 can call functions `setCourseName` and `getCourseName`, respectively, because each is declared as a member function of the class in `GradeBook.h` (and because these calls conform with the corresponding prototypes).

Figure 3.12. `GradeBook` member-function definitions represent the implementation of class `GradeBook`.

```
1  // Fig. 3.12: GradeBook.cpp
2  // GradeBook member-function definitions. This file contains
3  // implementations of the member functions prototyped in GradeBook.h.
4  #include <iostream>
5  using std::cout;
6  using std::endl;
7
8  #include "GradeBook.h" // include definition of class GradeBook
9
10 // constructor initializes courseName with string supplied as argument
11 GradeBook::GradeBook( string name )
12 {
13    setCourseName( name ); // call set function to initialize courseName
14 } // end GradeBook constructor
15
16 // function to set the course name
17 void GradeBook::setCourseName( string name )
18 {
19    courseName = name; // store the course name in the object
20 } // end function setCourseName
21
22 // function to get the course name
23 string GradeBook::getCourseName()
24 {
25    return courseName; // return object's courseName
26 } // end function getCourseName
27
28 // display a welcome message to the GradeBook user
29 void GradeBook::displayMessage()
30 {
31    // call getCourseName to get the courseName
32    cout << "Welcome to the grade book for\n" << getCourseName()
33       << "!" << endl;
```

```
34  } // end function displayMessage
```

Testing Class `GradeBook`

Figure 3.13 performs the same `GradeBook` object manipulations as Fig. 3.10. Separating `GradeBook`'s interface from the implementation of its member functions does not affect the way that this client code uses the class. It affects only how the program is compiled and linked, which we discuss in detail shortly.

[Page 103]

Figure 3.13. `GradeBook` class demonstration after separating its interface from its implementation.

```
 1  // Fig. 3.13: fig03_13.cpp
 2  // GradeBook class demonstration after separating
 3  // its interface from its implementation.
 4  #include <iostream>
 5  using std::cout;
 6  using std::endl;
 7
 8  #include "GradeBook.h" // include definition of class GradeBook
 9
10  // function main begins program execution
11  int main()
12  {
13     // create two GradeBook objects
14     GradeBook gradeBook1( "CS101 Introduction to C++ Programming" );
15     GradeBook gradeBook2( "CS102 Data Structures in C++" );
16
17     // display initial value of courseName for each GradeBook
18     cout << "gradeBook1 created for course: " << gradeBook1.getCourseName()
19        << "\ngradeBook2 created for course: " << gradeBook2.getCourseName()
20        << endl;
21     return 0; // indicate successful termination
22  } // end main
```

```
gradeBook1 created for course: CS101 Introduction to C++ Programming
gradeBook2 created for course: CS102 Data Structures in C++
```

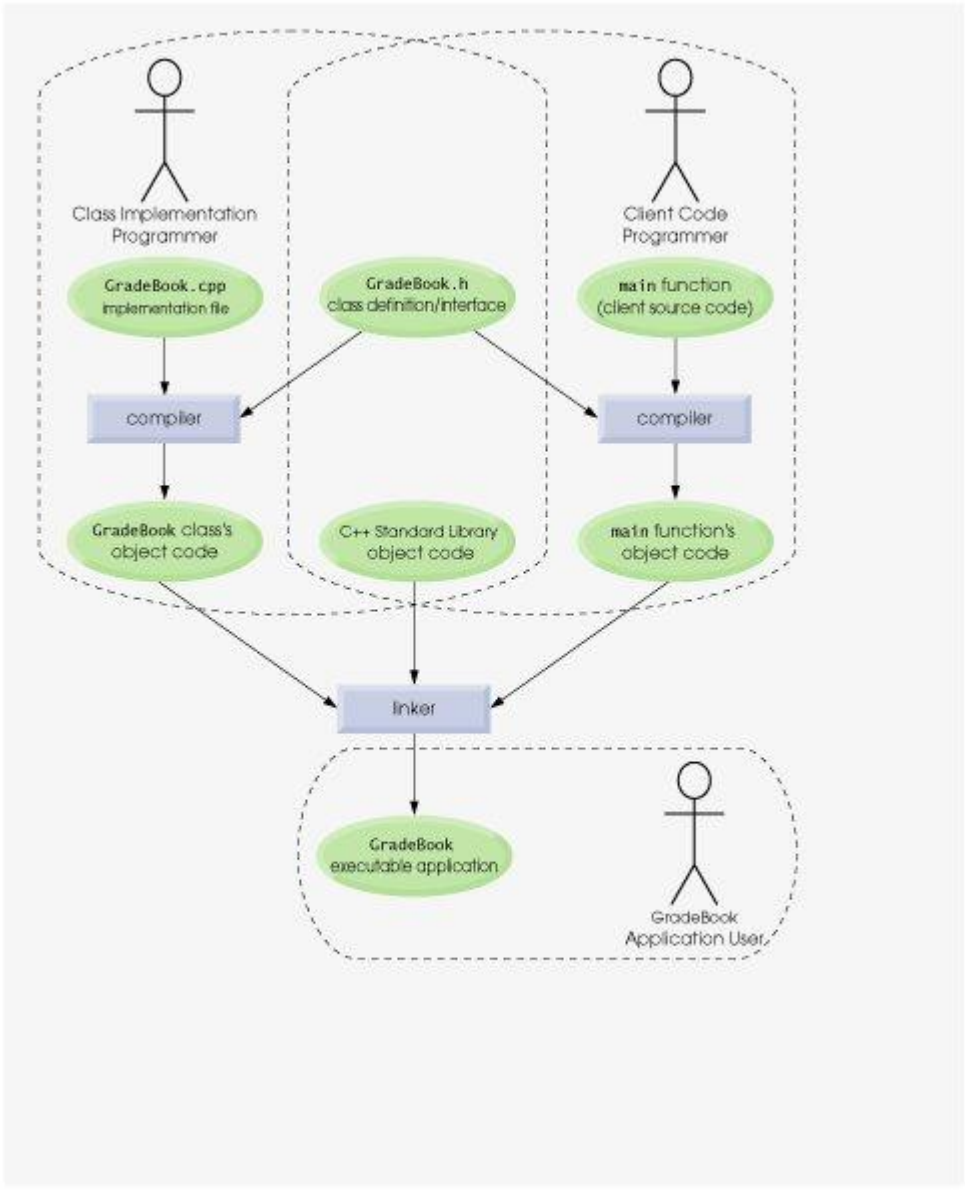
As in Fig. 3.10, line 8 of Fig. 3.13 includes the `GradeBook.h` header file so that the compiler can ensure that `GradeBook` objects are created and manipulated correctly in the client code. Before executing this program, the source-code files in Fig. 3.12 and Fig. 3.13 must both be compiled, then linked togetherthat is, the member-function calls in the client code need to be tied to the implementations of the class's member functionsa job performed by the linker.

The Compilation and Linking Process

The diagram in Fig. 3.14 shows the compilation and linking process that results in an executable `GradeBook` application that can be used by instructors. Often a class's interface and implementation will be created and compiled by one programmer and used by a separate programmer who implements the class's client code. So, the diagram shows what is required by both the class-implementation programmer and the client-code programmer. The dashed lines in the diagram show the pieces required by the class-implementation programmer, the client-code programmer and the `GradeBook` application user, respectively. [Note: Figure 3.14 is not a UML diagram.]

Figure 3.14. Compilation and linking process that produces an executable application.
(This item is displayed on page 104 in the print version)

[View full size image]



A class-implementation programmer responsible for creating a reusable `GradeBook` class creates the header file `GradeBook.h` and source-code file `GradeBook.cpp` that `#includes` the header file, then compiles the source-code file to create `GradeBook`'s object code. To hide the implementation details of `GradeBook`'s member functions, the class-implementation programmer would provide the client-code programmer with the header file `GradeBook.h` (which specifies the class's interface and data members) and the object code for class `GradeBook` which contains the machine-language instructions that represent `GradeBook`'s member functions. The client-code programmer is not given

`GradeBook`'s source-code file, so the client remains unaware of how `GradeBook`'s member functions are implemented.

[Page 104]

[Page 105]

The client code needs to know only `GradeBook`'s interface to use the class and must be able to link its object code. Since the interface of the class is part of the class definition in the `GradeBook.h` header file, the client-code programmer must have access to this file and `#include` it in the client's source-code file. When the client code is compiled, the compiler uses the class definition in `GradeBook.h` to ensure that the `main` function creates and manipulates objects of class `GradeBook` correctly.

To create the executable `GradeBook` application to be used by instructors, the last step is to link

1. the object code for the `main` function (i.e., the client code)
2. the object code for class `GradeBook`'s member function implementations
3. the C++ Standard Library object code for the C++ classes (e.g., `string`) used by the class implementation programmer and the client-code programmer.

The linker's output is the executable `GradeBook` application that instructors can use to manage their students' grades.

For further information on compiling multiple-source-file programs, see your compiler's documentation or study the Dive-Into™ publications that we provide for various C++ compilers at www.deitel.com/books/cpphtp5.

PREV PREV NEXT NEXT

[Page 105 (continued)]

3.10. Validating Data with set Functions

In Section 3.6, we introduced set functions for allowing clients of a class to modify the value of a `private` data member. In Fig. 3.5, class `GradeBook` defines member function `setCourseName` to simply assign a value received in its parameter `name` to data member `courseName`. This member function does not ensure that the course name adheres to any particular format or follows any other rules regarding what a "valid" course name looks like. As we stated earlier, suppose that a university can print student transcripts containing course names of only 25 characters or less. If the university uses a system containing `GradeBook` objects to generate the transcripts, we might want class `GradeBook` to ensure that its data member `courseName` never contains more than 25 characters. The program of Figs. 3.153.17 enhances class `GradeBook`'s member function `setCourseName` to perform this **validation** (also known as **validity checking**).

`GradeBook` Class Definition

Notice that `GradeBook`'s class definition (Fig. 3.15)and hence, its interfaceis identical to that of Fig. 3.11. Since the interface remains unchanged, clients of this class need not be changed when the

definition of member function `setCourseName` is modified. This enables clients to take advantage of the improved `GradeBook` class simply by linking the client code to the updated `GradeBook`'s object code.

Validating the Course Name with `GradeBook` Member Function `setCourseName`

The enhancement to class `GradeBook` is in the definition of `setCourseName` (Fig. 3.16, lines 1831). The `if` statement in lines 2021 determines whether parameter `name` contains a valid course name (i.e., a `string` of 25 or fewer characters). If the course name is valid, line 21 stores the course name in data member `courseName`. Note the expression `name.length()` in line 20. This is a member-function call just like `myGradeBook.displayMessage()`. The C++ Standard Library's `string` class defines a member function **length** that returns the number of characters in a `string` object. Parameter `name` is a `string` object, so the call `name.length()` returns the number of characters in `name`. If this value is less than or equal to `25`, `name` is valid and line 21 executes.

[Page 107]

Figure 3.15. `GradeBook` class definition.

(This item is displayed on page 106 in the print version)

```
1  // Fig. 3.15: GradeBook.h
2  // GradeBook class definition presents the public interface of
3  // the class. Member-function definitions appear in GradeBook.cpp.
4  #include <string> // program uses C++ standard string class
5  using std::string;
6
7  // GradeBook class definition
8  class GradeBook
9  {
10 public:
11    GradeBook( string ); // constructor that initializes a GradeBook object
12    void setCourseName( string ); // function that sets the course name
13    string getCourseName(); // function that gets the course name
14    void displayMessage(); // function that displays a welcome message
15 private:
16    string courseName; // course name for this GradeBook
17 }; // end class GradeBook
```

Figure 3.16. Member-function definitions for class `GradeBook` with a set function that validates the length of data member `courseName`.

(This item is displayed on pages 106 - 107 in the print version)

```
1  // Fig. 3.16: GradeBook.cpp
2  // Implementations of the GradeBook member-function definitions.
3  // The setCourseName function performs validation.
4  #include <iostream>
5  using std::cout;
6  using std::endl;
7
8  #include "GradeBook.h" // include definition of class GradeBook
9
10 // constructor initializes courseName with string supplied as argument
11 GradeBook::GradeBook( string name )
12 {
13    setCourseName( name ); // validate and store courseName
```

```
14  } // end GradeBook constructor
15
16  // function that sets the course name;
17  // ensures that the course name has at most 25 characters
18  void GradeBook::setCourseName( string name )
19  {
20     if ( name.length() <= 25 ) // if name has 25 or fewer characters
21        courseName = name; // store the course name in the object
22
23     if ( name.length() > 25 ) // if name has more than 25 characters
24     {
25        // set courseName to first 25 characters of parameter name
26        courseName = name.substr( 0, 25 ); // start at 0, length of 25
27
28        cout << "Name \"" << name << "\" exceeds maximum length (25).\n"
29           << "Limiting courseName to first 25 characters.\n" << endl;
30     } // end if
31  } // end function setCourseName
32
33  // function to get the course name
34  string GradeBook::getCourseName()
35  {
36     return courseName; // return object's courseName
37  } // end function getCourseName
38
39  // display a welcome message to the GradeBook user
40  void GradeBook::displayMessage()
41  {
42     // call getCourseName to get the courseName
43     cout << "Welcome to the grade book for\n" << getCourseName()
44        << "!" << endl;
45  } // end function displayMessage
```

The `if` statement in lines 2330 handles the case in which `setCourseName` receives an invalid course name (i.e., a name that is more than 25 characters long). Even if parameter `name` is too long, we still want to leave the `GradeBook` object in a **consistent state**that is, a state in which the object's data member `courseName` contains a valid value (i.e., a `string` of 25 characters or less). Thus, we truncate (i.e., shorten) the specified course name and assign the first 25 characters of `name` to the `courseName` data member (unfortunately, this could truncate the course name awkwardly). Standard class `string` provides member function **substr** (short for "substring") that returns a new `string` object created by copying part of an existing `string` object. The call in line 26 (i.e., `name.substr ( 0, 25 )`) passes two integers (`0` and `25`) to `name`'s member function `substr`. These arguments indicate the portion of the string `name` that `substr` should return. The first argument specifies the starting position in the original `string` from which characters are copiedthe first character in every string is considered to be at position 0. The second argument specifies the number of characters to copy. Therefore, the call in line 26 returns a 25-character substring of `name` starting at position 0 (i.e., the first 25 characters in `name`). For example, if `name` holds the value `"CS101 Introduction to Programming in C++"`, `substr` returns `"CS101 Introduction to Pro"`. After the call to `substr`, line 26 assigns the substring returned by `substr` to data member `courseName`. In this way, member function `setCourseName` ensures that `courseName` is always assigned a string containing 25 or fewer characters. If the member function has to truncate the course name to make it valid, lines 2829 display a warning message.

Note that the `if` statement in lines 2330 contains two body statementsone to set the `courseName` to the first 25 characters of parameter `name` and one to print an accompanying message to the user. We want both of these statements to execute when `name` is too long, so we place them in a pair of braces,

{ }. Recall from Chapter 2 that this creates a block. You will learn more about placing multiple statements in the body of a control statement in Chapter 4.

[Page 108]

Note that the cout statement in lines 2829 could also appear without a stream insertion operator at the start of the second line of the statement, as in:

```
cout << "Name \"" << name << "\" exceeds maximum length (25).\n"
   "Limiting courseName to first 25 characters.\n" << endl;
```

The C++ compiler combines adjacent string literals, even if they appear on separate lines of a program. Thus, in the statement above, the C++ compiler would combine the string literals "\" exceeds maximum length (25).\n" and "Limiting courseName to first 25 characters.\n" into a single string literal that produces output identical to that of lines 2829 in Fig. 3.16. This behavior allows you to print lengthy strings by breaking them across lines in your program without including additional stream insertion operations.

Testing Class GradeBook

Figure 3.17 demonstrates the modified version of class GradeBook (Figs. 3.153.16) featuring validation. Line 14 creates a GradeBook object named gradeBook1. Recall that the GradeBook constructor calls member function setCourseName to initialize data member courseName. In previous versions of the class, the benefit of calling setCourseName in the constructor was not evident. Now, however, the constructor takes advantage of the validation provided by setCourseName. The constructor simply calls setCourseName, rather than duplicating its validation code. When line 14 of Fig. 3.17 passes an initial course name of "CS101 Introduction to Programming in C++" to the GradeBook constructor, the constructor passes this value to setCourseName, where the actual initialization occurs. Because this course name contains more than 25 characters, the body of the second if statement executes, causing courseName to be initialized to the truncated 25-character course name "CS101 Introduction to Pro" (the truncated part is highlighted in red in line 14). Notice that the output in Fig. 3.17 contains the warning message output by lines 2829 of Fig. 3.16 in member function setCourseName. Line 15 creates another GradeBook object called gradeBook2the valid course name passed to the constructor is exactly 25 characters.

[Page 109]

Figure 3.17. Creating and manipulating a GradeBook object in which the course name is limited to 25 characters in length.

(This item is displayed on pages 108 - 109 in the print version)

```
 1  // Fig. 3.17: fig03_17.cpp
 2  // Create and manipulate a GradeBook object; illustrate validation.
 3  #include <iostream>
 4  using std::cout;
 5  using std::endl;
 6
 7  #include "GradeBook.h" // include definition of class GradeBook
 8
 9  // function main begins program execution
10  int main()
11  {
```

```
12     // create two GradeBook objects;
13     // initial course name of gradeBook1 is too long
14     GradeBook gradeBook1( "CS101 Introduction to Programming in C++" );
15     GradeBook gradeBook2( "CS102 C++ Data Structures" );
16
17     // display each GradeBook's courseName
18     cout << "gradeBook1's initial course name is: "
19        << gradeBook1.getCourseName()
20        << "\ngradeBook2's initial course name is: "
21        << gradeBook2.getCourseName() << endl;
22
23     // modify myGradeBook's courseName (with a valid-length string)
24     gradeBook1.setCourseName( "CS101 C++ Programming" );
25
26     // display each GradeBook's courseName
27     cout << "\ngradeBook1's course name is: "
28        << gradeBook1.getCourseName()
29        << "\ngradeBook2's course name is: "
30        << gradeBook2.getCourseName() << endl;
31     return 0; // indicate successful termination
32  } // end main
```

```
Name "CS101 Introduction to Programming in C++" exceeds maximum length (25).
Limiting courseName to first 25 characters.

gradeBook1's initial course name is: CS101 Introduction to Pro
gradeBook2's initial course name is: CS102 C++ Data Structures

gradeBook1's course name is: CS101 C++ Programming
gradeBook2's course name is: CS102 C++ Data Structures
```

Lines 1821 of Fig. 3.17 display the truncated course name for `gradeBook1` (we highlight this in red in the program output) and the course name for `gradeBook2`. Line 24 calls `gradeBook1`'s `setCourseName` member function directly, to change the course name in the `GradeBook` object to a shorter name that does not need to be truncated. Then, lines 2730 output the course names for the `GradeBook` objects again.

Additional Notes on Set Functions

A `public` set function such as `setCourseName` should carefully scrutinize any attempt to modify the value of a data member (e.g., `courseName`) to ensure that the new value is appropriate for that data item. For example, an attempt to set the day of the month to 37 should be rejected, an attempt to set a person's weight to zero or a negative value should be rejected, an attempt to set a grade on an exam to 185 (when the proper range is zero to 100) should be rejected, etc.

Software Engineering Observation 3.6

Making data members `private` and controlling access, especially write access, to those data members through `public` member functions helps ensure data integrity.

Error-Prevention Tip 3.5

The benefits of data integrity are not automatic simply because data members are made `private`the programmer must provide appropriate validity checking and report the errors.

[Page 110]

Software Engineering Observation 3.7

Member functions that set the values of `private` data should verify that the intended new values are proper; if they are not, the set functions should place the `private` data members into an appropriate state.

A class's set functions can return values to the class's clients indicating that attempts were made to assign invalid data to objects of the class. A client of the class can test the return value of a set function to determine whether the client's attempt to modify the object was successful and to take appropriate action. In Chapter 16, we demonstrate how clients of a class can be notified via the exception-handling mechanism when an attempt is made to modify an object with an inappropriate value. To keep the program of Figs. 3.153.17 simple at this early point in the book, `setCourseName` in Fig. 3.16 just prints an appropriate message on the screen.

PREV PREV NEXT NEXT

[Page 110 (continued)]

3.11. (Optional) Software Engineering Case Study: Identifying the Classes in the ATM Requirements Document

Now we begin designing the ATM system that we introduced in Chapter 2. In this section, we identify the classes that are needed to build the ATM system by analyzing the nouns and noun phrases that appear in the requirements document. We introduce UML class diagrams to model the relationships between these classes. This is an important first step in defining the structure of our system.

Identifying the Classes in a System

We begin our OOD process by identifying the classes required to build the ATM system. We will eventually describe these classes using UML class diagrams and implement these classes in C++. First, we review the requirements document of Section 2.8 and find key nouns and noun phrases to help us identify classes that comprise the ATM system. We may decide that some of these nouns and noun phrases are attributes of other classes in the system. We may also conclude that some of the nouns do not correspond to parts of the system and thus should not be modeled at all. Additional classes may become apparent to us as we proceed through the design process.

Figure 3.18 lists the nouns and noun phrases in the requirements document. We list them from left to right in the order in which they appear in the requirements document. We list only the singular form

of each noun or noun phrase.

We create classes only for the nouns and noun phrases that have significance in the ATM system. We do not need to model "bank" as a class, because the bank is not a part of the ATM systemthe bank simply wants us to build the ATM. "Customer" and "user" also represent entities outside of the systemthey are important because they interact with our ATM system, but we do not need to model them as classes in the ATM software. Recall that we modeled an ATM user (i.e., a bank customer) as the actor in the use case diagram of Fig. 2.18.

We do not model "$20 bill" or "deposit envelope" as classes. These are physical objects in the real world, but they are not part of what is being automated. We can adequately represent the presence of bills in the system using an attribute of the class that models the cash dispenser. (We assign attributes to classes in Section 4.13.) For example, the cash dispenser maintains a count of the number of bills it contains. The requirements document does not say anything about what the system should do with deposit envelopes after it receives them. We can assume that simply acknowledging the receipt of an envelopean operation performed by the class that models the deposit slotis sufficient to represent the presence of an envelope in the system. (We assign operations to classes in Section 6.22.)

[Page 111]

Figure 3.18. Nouns and noun phrases in the requirements document.

Nouns and noun phrases in the requirements document		
bank	money / fund	account number
ATM	screen	PIN
user	keypad	bank database
customer	cash dispenser	balance inquiry
transaction	$20 bill / cash	withdrawal
account	deposit slot	deposit
balance	deposit envelope	

In our simplified ATM system, representing various amounts of "money," including the "balance" of an account, as attributes of other classes seems most appropriate. Likewise, the nouns "account number" and "PIN" represent significant pieces of information in the ATM system. They are important attributes of a bank account. They do not, however, exhibit behaviors. Thus, we can most appropriately model them as attributes of an account class.

Though the requirements document frequently describes a "transaction" in a general sense, we do not model the broad notion of a financial transaction at this time. Instead, we model the three types of transactions (i.e., "balance inquiry," "withdrawal" and "deposit") as individual classes. These classes possess specific attributes needed for executing the transactions they represent. For example, a withdrawal needs to know the amount of money the user wants to withdraw. A balance inquiry, however, does not require any additional data. Furthermore, the three transaction classes exhibit unique behaviors. A withdrawal includes dispensing cash to the user, whereas a deposit involves receiving deposit envelopes from the user. [Note: In Section 13.10, we "factor out" common features of all transactions into a general "transaction" class using the object-oriented concepts of abstract classes and inheritance.]

We determine the classes for our system based on the remaining nouns and noun phrases from Fig. 3.18. Each of these refers to one or more of the following:

- ATM
- screen
- keypad
- cash dispenser
- deposit slot
- account
- bank database
- balance inquiry

[Page 112]

- withdrawal
- deposit

The elements of this list are likely to be classes we will need to implement our system.

We can now model the classes in our system based on the list we have created. We capitalize class names in the design processa UML conventionas we will do when we write the actual C++ code that implements our design. If the name of a class contains more than one word, we run the words together and capitalize each word (e.g., `MultipleWordName`). Using this convention, we create classes `ATM`, `Screen`, `Keypad`, `CashDispenser`, `DepositSlot`, `Account`, `BankDatabase`, `BalanceInquiry`, `Withdrawal` and `Deposit`. We construct our system using all of these classes as building blocks. Before we begin building the system, however, we must gain a better understanding of how the classes relate to one another.

Modeling Classes

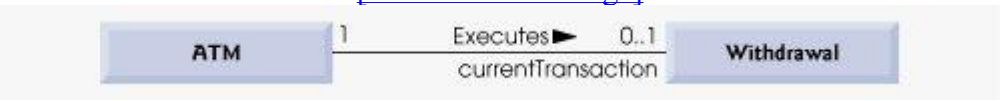
The UML enables us to model, via class diagrams, the classes in the ATM system and their interrelationships. Figure 3.19 represents class `ATM`. In the UML, each class is modeled as a rectangle with three compartments. The top compartment contains the name of the class, centered horizontally and in boldface. The middle compartment contains the class's attributes. (We discuss attributes in Section 4.13 and Section 5.11.) The bottom compartment contains the class's operations (discussed in Section 6.22). In Fig. 3.19 the middle and bottom compartments are empty, because we have not yet determined this class's attributes and operations.

Figure 3.19. Representing a class in the UML using a class diagram.

Class diagrams also show the relationships between the classes of the system. Figure 3.20 shows how our classes `ATM` and `Withdrawal` relate to one another. For the moment, we choose to model only this subset of classes for simplicity. We present a more complete class diagram later in this section. Notice that the rectangles representing classes in this diagram are not subdivided into compartments. The UML allows the suppression of class attributes and operations in this manner, when appropriate, to create more readable diagrams. Such a diagram is said to be an elided diagramone in which some information, such as the contents of the second and third compartments, is not modeled. We will place information in these compartments in Section 4.13 and Section 6.22

Figure 3.20. Class diagram showing an association among classes.

[View full size image]



In Fig. 3.20, the solid line that connects the two classes represents an associationa relationship between classes. The numbers near each end of the line are multiplicity values, which indicate how many objects of each class participate in the association. In this case, following the line from one end to the other reveals that, at any given moment, one `ATM` object participates in an association with either zero or one `Withdrawal` objectszero if the current user is not currently performing a transaction or has requested a different type of transaction, and one if the user has requested a withdrawal. The UML can model many types of multiplicity. Figure 3.21 lists and explains the multiplicity types.

[Page 113]

An association can be named. For example, the word `Executes` above the line connecting classes `ATM` and `Withdrawal` in Fig. 3.20 indicates the name of that association. This part of the diagram reads "one object of class `ATM` executes zero or one objects of class `Withdrawal`." Note that association names are directional, as indicated by the filled arrow-headso it would be improper, for example, to read the preceding association from right to left as "zero or one objects of class `Withdrawal` execute one object of class `ATM`."

The word `currentTransaction` at the `Withdrawal` end of the association line in Fig. 3.20 is a role name, which identifies the role the `Withdrawal` object plays in its relationship with the `ATM`. A role name adds meaning to an association between classes by identifying the role a class plays in the context of an association. A class can play several roles in the same system. For example, in a school personnel system, a person may play the role of "professor" when relating to students. The same person may take on the role of "colleague" when participating in a relationship with another professor, and "coach" when coaching student athletes. In Fig. 3.20, the role name `currentTransaction` indicates that the `Withdrawal` object participating in the `Executes` association with an object of class `ATM` represents the transaction currently being processed by the ATM. In other contexts, a `Withdrawal` object may take on other roles (e.g., the previous transaction). Notice that we do not specify a role name for the `ATM` end of the `Executes` association. Role names in class diagrams are often omitted when the meaning of an association is clear without them.

In addition to indicating simple relationships, associations can specify more complex relationships, such as objects of one class being composed of objects of other classes. Consider a real-world automated teller machine. What "pieces" does a manufacturer put together to build a working ATM?

Our requirements document tells us that the ATM is composed of a screen, a keypad, a cash dispenser and a deposit slot.

Figure 3.21. Multiplicity types.

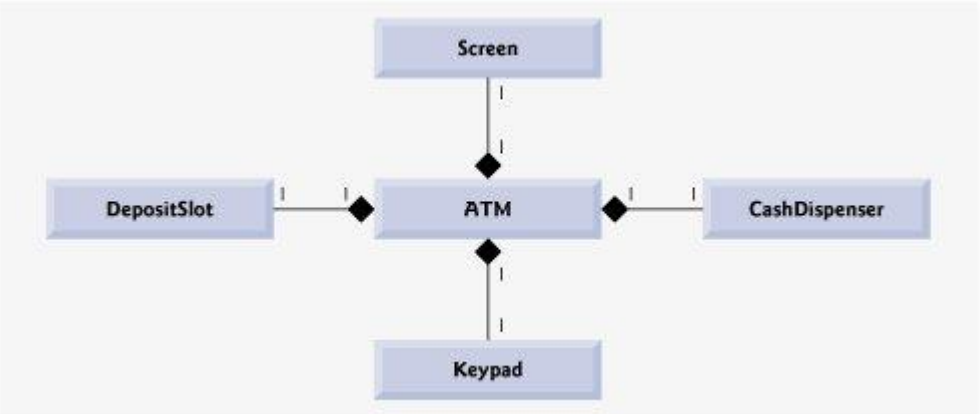
Symbol	Meaning
0	None
1	One
m	An integer value
0..1	Zero or one
m, n	m or n
m..n	At least m, but not more than n
*	Any nonnegative integer (zero or more)
0..*	Zero or more (identical to *)
1..*	One or more

[Page 114]

In Fig. 3.22, the solid diamonds attached to the association lines of class `ATM` indicate that class `ATM` has a composition relationship with classes `Screen`, `Keypad`, `CashDispenser` and `DepositSlot`. Composition implies a whole/part relationship. The class that has the composition symbol (the solid diamond) on its end of the association line is the whole (in this case, `ATM`), and the classes on the other end of the association lines are the partsin this case, classes `Screen`, `Keypad`, `CashDispenser` and `DepositSlot`. The compositions in Fig. 3.22 indicate that an object of class `ATM` is formed from one object of class `Screen`, one object of class `CashDispenser`, one object of class `Keypad` and one object of class `DepositSlot`. The ATM "has a" screen, a keypad, a cash dispenser and a deposit slot. The "has-a" relationship defines composition. (We will see in the "Software Engineering Case Study" section in Chapter 13 that the "is-a" relationship defines inheritance.)

Figure 3.22. Class diagram showing composition relationships.

[View full size image]



According to the UML specification, composition relationships have the following properties:

1. Only one class in the relationship can represent the whole (i.e., the diamond can be placed on only one end of the association line). For example, either the screen is part of the ATM or the ATM is part of the screen, but the screen and the ATM cannot both represent the whole in the relationship.

2. The parts in the composition relationship exist only as long as the whole, and the whole is responsible for the creation and destruction of its parts. For example, the act of constructing an ATM includes manufacturing its parts. Furthermore, if the ATM is destroyed, its screen, keypad, cash dispenser and deposit slot are also destroyed.

3. A part may belong to only one whole at a time, although the part may be removed and attached to another whole, which then assumes responsibility for the part.

The solid diamonds in our class diagrams indicate composition relationships that fulfill these three properties. If a "has-a" relationship does not satisfy one or more of these criteria, the UML specifies that hollow diamonds be attached to the ends of association lines to indicate aggregationa weaker form of composition. For example, a personal computer and a computer monitor participate in an aggregation relationshipthe computer "has a" monitor, but the two parts can exist independently, and the same monitor can be attached to multiple computers at once, thus violating the second and third properties of composition.

[Page 115]

Figure 3.23 shows a class diagram for the ATM system. This diagram models most of the classes that we identified earlier in this section, as well as the associations between them that we can infer from the requirements document. [Note: Classes `BalanceInquiry` and `Deposit` participate in associations similar to those of class `Withdrawal`, so we have chosen to omit them from this diagram to keep it simple. In Chapter 13, we expand our class diagram to include all the classes in the ATM system.]

Figure 3.23. Class diagram for the ATM system model.

[View full size image]

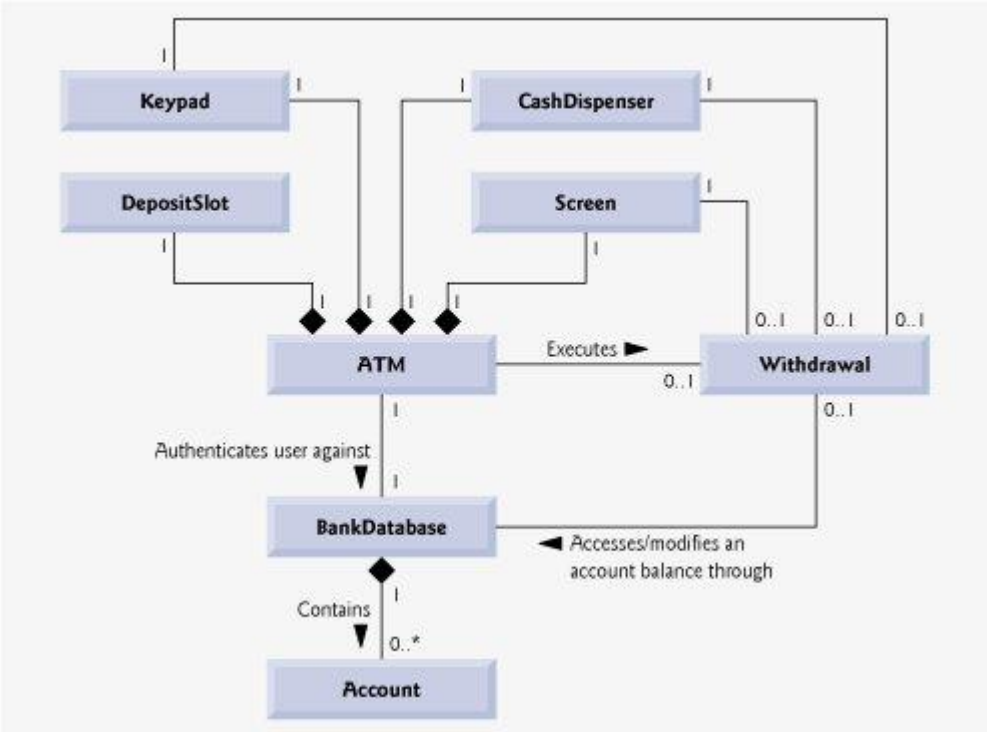


Figure 3.23 presents a graphical model of the structure of the ATM system. This class diagram includes classes `BankDatabase` and `Account`, and several associations that were not present in either Fig. 3.20 or Fig. 3.22. The class diagram shows that class `ATM` has a one-to-one relationship with class `BankDatabase`one `ATM` object authenticates users against one `BankDatabase` object. In Fig. 3.23, we also model the fact that the bank's database contains information about many accountsone object of class `BankDatabase` participates in a composition relationship with zero or more objects of class `Account`. Recall from Fig. 3.21 that the multiplicity value 0..* at the `Account` end of the association between class `BankDatabase` and class `Account` indicates that zero or more objects of class `Account` take part in the association. Class `BankDatabase` has a one-to-many relationship with class `Account`the `BankDatabase` stores many `Account`s. Similarly, class `Account` has a many-to-one relationship with class `BankDatabase`there can be many `Accounts` stored in the `BankDatabase`. [Note: Recall from Fig. 3.21 that the multiplicity value * is identical to 0..*. We include 0..* in our class diagrams for clarity.]

[Page 116]

Figure 3.23 also indicates that if the user is performing a withdrawal, "one object of class `Withdrawal` accesses/modifies an account balance through one object of class `BankDatabase`." We could have created an association directly between class `Withdrawal` and class `Account`. The requirements document, however, states that the "ATM must interact with the bank's account information database" to perform transactions. A bank account contains sensitive information, and systems engineers must always consider the security of personal data when designing a system. Thus, only the `BankDatabase` can access and manipulate an account directly. All other parts of the system must interact with the database to retrieve or update account information (e.g., an account balance).

The class diagram in Fig. 3.23 also models associations between class `Withdrawal` and classes `Screen`, `CashDispenser` and `Keypad`. A withdrawal transaction includes prompting the user to choose a withdrawal amount and receiving numeric input. These actions require the use of the screen and the keypad, respectively. Furthermore, dispensing cash to the user requires access to the cash

dispenser.

Classes `BalanceInquiry` and `Deposit`, though not shown in Fig. 3.23, take part in several associations with the other classes of the ATM system. Like class `Withdrawal`, each of these classes associates with classes `ATM` and `BankDatabase`. An object of class `BalanceInquiry` also associates with an object of class `Screen` to display the balance of an account to the user. Class `Deposit` associates with classes `Screen`, `Keypad` and `DepositSlot`. Like withdrawals, deposit transactions require use of the screen and the keypad to display prompts and receive input, respectively. To receive deposit envelopes, an object of class `Deposit` accesses the deposit slot.

We have now identified the classes in our ATM system (although we may discover others as we proceed with the design and implementation). In Section 4.13, we determine the attributes for each of these classes, and in Section 5.11, we use these attributes to examine how the system changes over time. In Section 6.22, we determine the operations of the classes in our system.

Software Engineering Case Study Self-Review Exercises

3.1 Suppose we have a class `Car` that represents a car. Think of some of the different pieces that a manufacturer would put together to produce a whole car. Create a class diagram (similar to Fig. 3.22) that models some of the composition relationships of class `Car`.

3.2 Suppose we have a class `File` that represents an electronic document in a stand-alone, non-networked computer represented by class `Computer`. What sort of association exists between class `Computer` and class `File`?

a. Class `Computer` has a one-to-one relationship with class `File`.

b. Class `Computer` has a many-to-one relationship with class `File`.

c. Class `Computer` has a one-to-many relationship with class `File`.

d. Class `Computer` has a many-to-many relationship with class `File`.

3.3 State whether the following statement is true or false, and if false, explain why: A UML diagram in which a class's second and third compartments are not modeled is said to be an elided diagram.

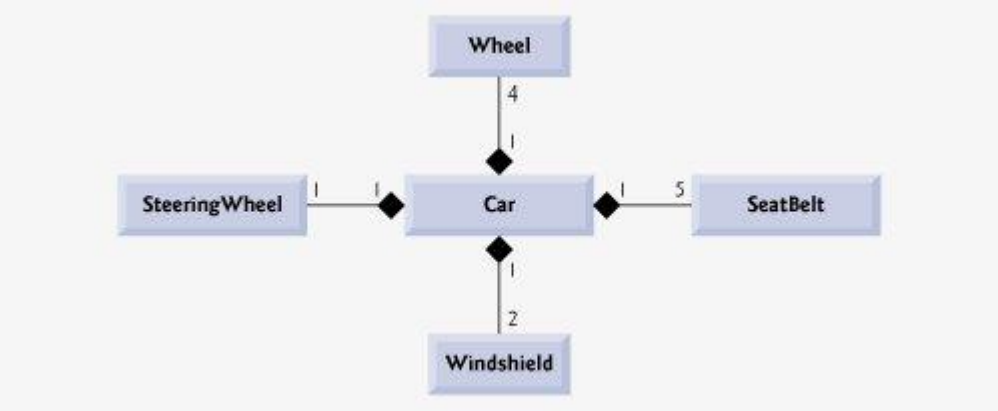
3.4 Modify the class diagram of Fig. 3.23 to include class `Deposit` instead of class `Withdrawal`.

Answers to Software Engineering Case Study Self-Review Exercises

3.1 [Note: Student answers may vary.] Figure 3.24 presents a class diagram that shows some of the composition relationships of a class `Car`.

Figure 3.24. Class diagram showing composition relationships of a class `Car`.
(This item is displayed on page 117 in the print version)

[View full size image]



3.2 c. [Note: In a computer network, this relationship could be many-to-many.]

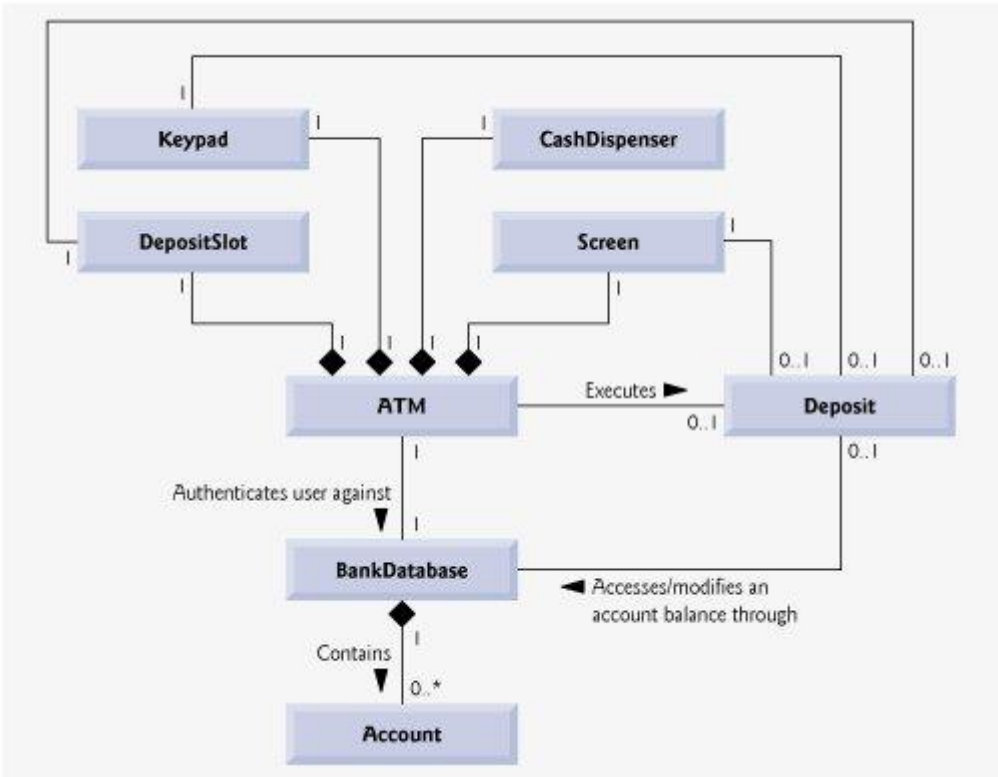
[Page 117]

3.3 True.

3.4 Figure 3.25 presents a class diagram for the ATM including class `Deposit` instead of class `Withdrawal` (as in Fig. 3.23). Note that `Deposit` does not access `CashDispenser`, but does access `DepositSlot`.

Figure 3.25. Class diagram for the ATM system model including class `Deposit`.

[View full size image]



[Page 118]

3.12. Wrap-Up

In this chapter, you learned how to create user-defined classes, and how to create and use objects of those classes. In particular, we declared data members of a class to maintain data for each object of the class. We also defined member functions that operate on that data. You learned how to call an object's member functions to request the services it provides and how to pass data to those member functions as arguments. We discussed the difference between a local variable of a member function and a data member of a class. We also showed how to use a constructor to specify the initial values for an object's data members. You learned how to separate the interface of a class from its implementation to promote good software engineering. We also presented a diagram that shows the files that class-implementation programmers and client-code programmers need to compile the code they write. We demonstrated how set functions can be used to validate an object's data and ensure that objects are maintained in a consistent state. In addition, UML class diagrams were used to model classes and their constructors, member functions and data members. In the next chapter, we begin our introduction to control statements, which specify the order in which a function's actions are performed.

[Page 118 (continued)]

Summary

- Performing a task in a program requires a function. The function hides from its user the complex tasks that it performs.
- A function in a class is known as a member function and performs one of the class's tasks.
- You must create an object of a class before a program can perform the tasks the class describes. That is one reason C++ is known as an object-oriented programming language.
- Each message sent to an object is a member-function call that tells the object to perform a task.
- An object has attributes that are carried with the object as it is used in a program. These attributes are specified as data members in the object's class.
- A class definition contains the data members and member functions that define the class's attributes and behaviors, respectively.
- A class definition begins with the keyword `class` followed immediately by the class name.
- By convention, the name of a user-defined class begins with a capital letter and, for readability, each subsequent word in the class name begins with a capital letter.
- Every class's body is enclosed in a pair of braces (`{` and `}`) and ends with a semicolon.

- Member functions that appear after access specifier `public` can be called by other functions in a program and by member functions of other classes.
- Access specifiers are always followed by a colon (`:`).
- Keyword `void` is a special return type which indicates that a function will perform a task but will not return any data to its calling function when it completes its task.
- By convention, function names begin with a lowercase first letter and all subsequent words in the name begin with a capital letter.
- An empty set of parentheses after a function name indicates that the function does not require additional data to perform its task.
- Every function's body is delimited by left and right braces (`{` and `}`).
- Typically, you cannot call a member function until you create an object of its class.

[Page 119]

- Each new class you create becomes a new type in C++ that can be used to declare variables and create objects. This is one reason why C++ is known as an extensible language.
- A member function can require one or more parameters that represent additional data it needs to perform its task. A function call supplies arguments for each of the function's parameters.
- A member function is called by following the object name with a dot operator (`.`), the function name and a set of parentheses containing the function's arguments.
- A variable of C++ Standard Library class `string` represents a string of characters. This class is defined in header file `<string>`, and the name `string` belongs to namespace `std`.
- Function `getline` (from header `<string>`) reads characters from its first argument until a newline character is encountered, then places the characters (not including the newline) in the `string` variable specified as its second argument. The newline character is discarded.
- A parameter list may contain any number of parameters, including none at all (represented by empty parentheses) to indicate that a function does not require any parameters.
- The number of arguments in a function call must match the number of parameters in the parameter list of the called member function's header. Also, the argument types in the function call must be consistent with the types of the corresponding parameters in the function header.
- Variables declared in a function's body are local variables and can be used only from the point of their declaration in the function to the immediately following closing right brace (`}`). When a function terminates, the values of its local variables are lost.
- A local variable must be declared before it can be used in a function. A local variable cannot be accessed outside the function in which it is declared.
- Data members normally are `private`. Variables or functions declared `private` are accessible only to member functions of the class in which they are declared.

- When a program creates (instantiates) an object of a class, its `private` data members are encapsulated (hidden) in the object and can be accessed only by member functions of the object's class.
- When a function that specifies a return type other than `void` is called and completes its task, the function returns a result to its calling function.
- By default, the initial value of a `string` is the empty stringi.e., a string that does not contain any characters. Nothing appears on the screen when an empty string is displayed.
- Classes often provide `public` member functions to allow clients of the class to set or get `private` data members. The names of these member functions normally begin with set or get.
- Providing `public` set and get functions allows clients of a class to indirectly access the hidden data. The client knows that it is attempting to modify or obtain an object's data, but the client does not know how the object performs these operations.
- The set and get functions of a class also should be used by other member functions within the class to manipulate the class's `private` data, although these member functions can access the `private` data directly. If the class's data representation is changed, member functions that access the data only via the set and get functions will not require modificationonly the bodies of the set and *get* functions that directly manipulate the data member will need to change.
- A `public` set function should carefully scrutinize any attempt to modify the value of a data member to ensure that the new value is appropriate for that data item.
- Each class you declare should provide a constructor to initialize an object of the class when the object is created. A constructor is a special member function that must be defined with the same name as the class, so that the compiler can distinguish it from the class's other member functions.
- A difference between constructors and functions is that constructors cannot return values, so they cannot specify a return type (not even `void`). Normally, constructors are declared `public`.

[Page 120]

- C++ requires a constructor call at the time each object is created, which helps ensure that every object is initialized before it is used in a program.
- A constructor that takes no arguments is a default constructor. In any class that does not include a constructor, the compiler provides a default constructor. The class programmer can also define a default constructor explicitly. If the programmer defines a constructor for a class, C++ will not create a default constructor.
- Class definitions, when packaged properly, can be reused by programmers worldwide.
- It is customary to define a class in a header file that has a `.h` filename extension.
- If the class's implementation changes, the class's clients should not be required to change.
- Interfaces define and standardize the ways in which things such as people and systems interact.

- The interface of a class describes the `public` member functions (also known as `public` services) that are made available to the class's clients. The interface describes what services clients can use and how to request those services, but does not specify how the class carries out the services.
- A fundamental principle of good software engineering is to separate interface from implementation. This makes programs easier to modify. Changes in the class's implementation do not affect the client as long as the class's interface originally provided to the client remains unchanged.
- A function prototype contains a function's name, its return type and the number, types and order of the parameters the function expects to receive.
- Once a class is defined and its member functions are declared (via function prototypes), the member functions should be defined in a separate source-code file
- For each member function defined outside of its corresponding class definition, the function name must be preceded by the class name and the binary scope resolution operator (`::`).
- Class `string`'s `length` member function returns the number of characters in a `string` object.
- Class `string`'s member function `substr` (short for "substring") returns a new `string` object created by copying part of an existing `string` object. The function's first argument specifies the starting position in the original `string` from which characters are copied. Its second argument specifies the number of characters to copy.
- In the UML, each class is modeled in a class diagram as a rectangle with three compartments. The top compartment contains the class name, centered horizontally in boldface. The middle compartment contains the class's attributes (data members in C++). The bottom compartment contains the class's operations (member functions and constructors in C++).
- The UML models operations by listing the operation name followed by a set of parentheses. A plus sign (`+`) preceding the operation name indicates a `public` operation in the UML (i.e., a `public` member function in C++).
- The UML models a parameter of an operation by listing the parameter name, followed by a colon and the parameter type between the parentheses following the operation name.
- The UML has its own data types. Not all the UML data types have the same names as the corresponding C++ types. The UML type `String` corresponds to the C++ type `string`.
- The UML represents data members as attributes by listing the attribute name, followed by a colon and the attribute type. Private attributes are preceded by a minus sign () in the UML.
- The UML indicates the return type of an operation by placing a colon and the return type after the parentheses following the operation name.
- UML class diagrams do not specify return types for operations that do not return values.
- The UML models constructors as operations in a class diagram's third compartment. To distinguish a constructor from a class's operations, the UML places the word "constructor" between guillemets (« and ») before the constructor's name.

PREV PREV NEXT NEXT

[Page 121]

Terminology

access specifier

accessor

argument

attribute (UML)

binary scope resolution operator (: :)

body of a class definition

calling function (caller)

camel case

class definition

class diagram (UML)

class-implementation programmer

client-code programmer

client of an object or class

compartment in a class diagram (UML)

consistent state

constructor

data hiding

data member

default constructor

default precision

defining a class

dot operator (.)

empty string

extensible language

function call

function header

function prototype

get function

`getline` function of `<string>` library

guillemets, « and » (UML)

header file

implementation of a class

instance of a class

interface of a class

invoke a member function

`length` member function of class `string`

local variable

member function

member-function call

message (send to an object)

minus (-) sign (UML)

mutator

object code

operation (UML)

operation parameter (UML)

parameter

parameter list

plus (+) sign (UML)

precision

`private` access specifier

`public` access specifier

`public` services of a class

return type

separate interface from implementation

set function

software engineering

source-code file

`string` class

`<string>` header file

`substr` member function of class `string`

UML class diagram

validation

validity checking

`void` return type

PREV PREV NEXT NEXT

[Page 121 (continued)]

Self-Review Exercises

3.1 Fill in the blanks in each of the following:

a. A house is to a blueprint as a(n) _____ is to a class.

b. Every class definition contains keyword _____ followed immediately by the class's name.

c. A class definition is typically stored in a file with the _____ filename extension.

d. Each parameter in a function header should specify both a(n) _____ and a (n) _____.

e. When each object of a class maintains its own copy of an attribute, the variable

that represents the attribute is also known as a(n) _____.

f. Keyword `public` is a(n) _____.

g. Return type _____ indicates that a function will perform a task but will not return any information when it completes its task.

h. Function _____ from the `<string>` library reads characters until a newline character is encountered, then copies those characters into the specified `string`.

[Page 122]

i. When a member function is defined outside the class definition, the function header must include the class name and the _____, followed by the function name to "tie" the member function to the class definition.

j. The source-code file and any other files that use a class can include the class's header file via an _____ preprocessor directive.

3.2 State whether each of the following is true or false. If false, explain why.

a. By convention, function names begin with a capital letter and all subsequent words in the name begin with a capital letter.

b. Empty parentheses following a function name in a function prototype indicate that the function does not require any parameters to perform its task.

c. Data members or member functions declared with access specifier `private` are accessible to member functions of the class in which they are declared.

d. Variables declared in the body of a particular member function are known as data members and can be used in all member functions of the class.

e. Every function's body is delimited by left and right braces (`{` and `}`).

f. Any source-code file that contains `int main()` can be used to execute a program.

g. The types of arguments in a function call must match the types of the corresponding parameters in the function prototype's parameter list.

3.3 What is the difference between a local variable and a data member?

3.4 Explain the purpose of a function parameter. What is the difference between a parameter and an argument?

PREV PREV NEXT NEXT

[Page 122 (continued)]

Answers to Self-Review Exercises

3.1 a) object. b) `class`. c) `.h` d) type, name. e) data member. f) access specifier. g) `void`. h) `getline`. i) binary scope resolution operator (`::`). j) `#include`.

3.2 a) False. By convention, function names begin with a lowercase letter and all subsequent words in the name begin with a capital letter. b) True. c) True. d) False. Such variables are called local variables and can be used only in the member function in which they are declared. e) True. f) True. g) True.

3.3 A local variable is declared in the body of a function and can be used only from the point at which it is declared to the immediately following closing brace. A data member is declared in a class definition, but not in the body of any of the class's member functions. Every object (instance) of a class has a separate copy of the class's data members. Also, data members are accessible to all member functions of the class.

3.4 A parameter represents additional information that a function requires to perform its task. Each parameter required by a function is specified in the function header. An argument is the value supplied in the function call. When the function is called, the argument value is passed into the function parameter so that the function can perform its task.

PREV PREV NEXT NEXT

[Page 122 (continued)]

Exercises

3.5 Explain the difference between a function prototype and a function definition.

3.6 What is a default constructor? How are an object's data members initialized if a class has only an implicitly defined default constructor?

3.7 Explain the purpose of a data member.

3.8 What is a header file? What is a source-code file? Discuss the purpose of each.

3.9 Explain how a program could use class `string` without inserting a `using` declaration.

[Page 123]

3.10 Explain why a class might provide a set function and a get function for a data member.

3.11 (Modifying Class `GradeBook`) Modify class `GradeBook` (Figs. 3.113.12) as follows:

a. Include a second `string` data member that represents the course instructor's name.

b. Provide a set function to change the instructor's name and a get function to retrieve it.

c. Modify the constructor to specify two parametersone for the course name and one for the instructor's name.

d. Modify member function `displayMessage` such that it first outputs the welcome message and course name, then outputs `"This course is presented by: "` followed by the instructor's name.

Use your modified class in a test program that demonstrates the class's new capabilities.

3.12 (`Account` Class) Create a class called `Account` that a bank might use to represent customers' bank accounts. Your class should include one data member of type `int` to represent the account balance. [Note: In subsequent chapters, we'll use numbers that contain decimal points (e.g., 2.75)called floating-point valuesto represent dollar amounts.] Your class should provide a constructor that receives an initial balance and uses it to initialize the data member. The constructor should validate the initial balance to ensure that it is greater than or equal to 0. If not, the balance should be set to 0 and the constructor should display an error message, indicating that the initial balance was invalid. The class should provide three member functions. Member function `credit` should add an amount to the current balance. Member function `debit` should withdraw money from the `Account` and should ensure that the debit amount does not exceed the `Account`'s balance. If it does, the balance should be left unchanged and the function should print a message indicating `"Debit amount exceeded account balance."` Member function `getBalance` should return the current balance. Create a program that creates two `Account` objects and tests the member functions of class `Account`.

3.13 (`Invoice` Class) Create a class called `Invoice` that a hardware store might use to represent an invoice for an item sold at the store. An `Invoice` should include four pieces of information as data membersa part number (type `string`), a part description (type `string`), a quantity of the item being purchased (type `int`) and a price per item (type `int`). [Note: In subsequent chapters, we'll use numbers that contain decimal points (e.g., 2.75)called floating-point valuesto represent dollar amounts.] Your class should have a constructor that initializes the four data members. Provide a set and a get function for each data member. In addition, provide a member function named `getInvoiceAmount` that calculates the invoice amount (i.e., multiplies the quantity by the price per item), then returns the amount as an `int` value. If the quantity is not positive, it should be set to `0`. If the price per item is not positive, it should be set to `0`. Write a test program that demonstrates class `Invoice`'s capabilities.

3.14 (`Employee` Class) Create a class called `Employee` that includes three pieces of information as data membersa first name (type `string`), a last name (type `string`) and a monthly salary (type `int`). [Note: In subsequent chapters, we'll use numbers that contain decimal points (e.g., 2.75)called floating-point valuesto represent dollar amounts.] Your class should have a constructor that initializes the three data members. Provide a set and a get function for each data member. If the monthly salary is not positive, set it to `0`. Write a test program that demonstrates class `Employee`'s capabilities. Create two `Employee` objects and display each object's yearly salary. Then give each `Employee` a 10 percent raise and display each

`Employee`'s yearly salary again.

3.15 (`Date` Class) Create a class called `Date` that includes three pieces of information as data membersa month (type `int`), a day (type `int`) and a year (type `int`). Your class should have a constructor with three parameters that uses the parameters to initialize the three data members. For the purpose of this exercise, assume that the values provided for the year and day are correct, but ensure that the month value is in the range 112; if it is not, set the month to 1. Provide a set and a get function for each data member. Provide a member function `displayDate` that displays the month, day and year separated by forward slashes (`/`). Write a test program that demonstrates class `Date`'s capabilities.

PREV NEXT