



Introduction to Classes, Objects and Strings

©1992–2014 by Pearson Education, Inc.
All Rights Reserved.



OBJECTIVES

Topics for this lecture:

- How to define a class and use it to create an object.
- How to implement a class's behaviors as member functions.
- How to implement a class's attributes as data members.
- How to call a member function of an object to perform a task.
- The differences between data members of a class and local variables of a function.
- How to use a constructor to initialize an object's data when the object is created.
- How to engineer a class to separate its interface from its implementation and encourage reuse.
- How to use objects of class `string`.

©1992–2014 by Pearson Education, Inc.
All Rights Reserved.



Introduction
Defining a Class with a Member Function
Defining a Member Function with a Parameter
Data Members, *set* Member Functions and *get* Member Functions
Initializing Objects with Constructors
Placing a Class in a Separate File for Reusability
Separating Interface from Implementation
Validating Data with *set* Functions
Wrap-Up

©1992–2014 by Pearson Education, Inc.
All Rights Reserved.



► Introduction

- ▶ We will be writing programs that employ the basic concepts of *object-oriented programming*.
- ▶ Typically, the programs that we will develop in this class will consist of function `main` and one or more *classes*, each containing *data members* and *member functions*.
- ▶ Today, we will develop a simple, well-engineered framework for organizing object-oriented programs in C++.

©1992–2014 by Pearson Education, Inc.
All Rights Reserved.

► Defining a Class with a Member Function

- ▶ We begin with an example (Fig. 3.1) that consists of class **GradeBook**.
- ▶ Function **main** uses a **GradeBook** object and its **displayMessage** member function to display a message on the screen welcoming the instructor to the grade-book program.

©1992–2014 by Pearson Education, Inc.
All Rights Reserved.

```

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 namespace std;
6
7 // GradeBook class definition
8 class GradeBook
9 {
10 public:
11     // function that displays a welcome message to the GradeBook user
12     void displayMessage() const
13     {
14         cout << "Welcome to the Grade Book!" << endl;
15     } // end function displayMessage
16 }; // end class GradeBook
17
18 // function main begins program execution
19 int main()
20 {
21     GradeBook myGradeBook; // create a GradeBook object named myGradeBook
22     myGradeBook.displayMessage(); // call object's displayMessage function
23 } // end main

```

Fig. 3.1 | Define class **GradeBook** with a member function **displayMessage**, create a **GradeBook** object and call its **displayMessage** function. (Part I of 2.)

©1992–2014 by Pearson Education, Inc.
All Rights Reserved.

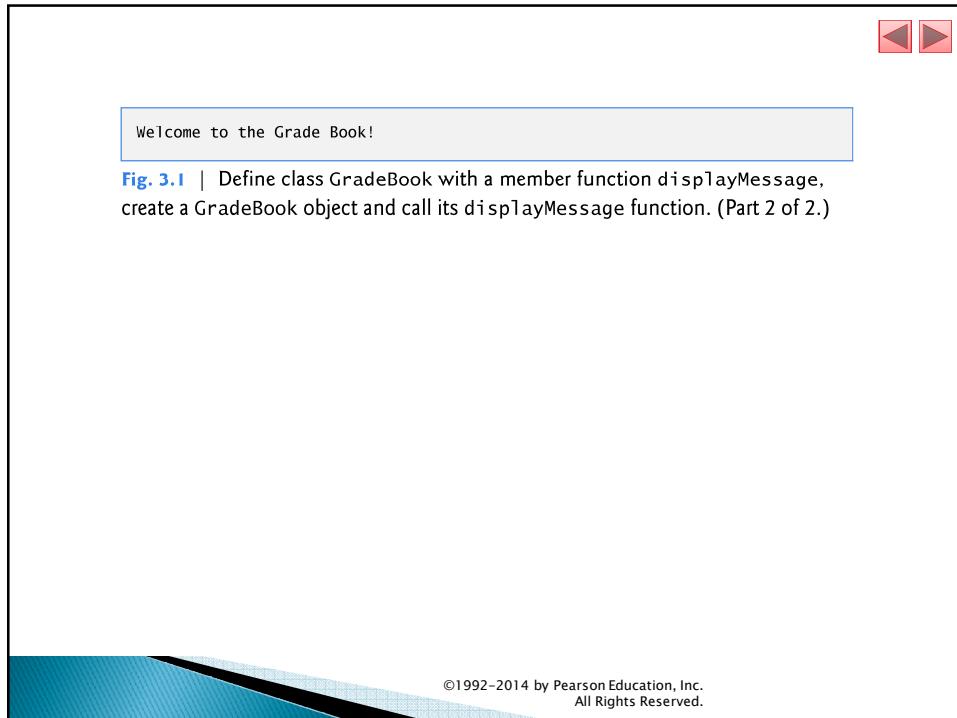


Fig. 3.1 | Define class `GradeBook` with a member function `displayMessage`, create a `GradeBook` object and call its `displayMessage` function. (Part 2 of 2.)

©1992–2014 by Pearson Education, Inc.
All Rights Reserved.

► **Defining a Class with a Member Function (cont.)**

- ▶ The `GradeBook` class definition (lines 8–16) begins with keyword `class` and contains a member function called `displayMessage` (lines 12–15) that displays a message on the screen (line 14).
- ▶ Need to make an object of class `GradeBook` (line 21) and call its `displayMessage` member function (line 22) to get line 14 to execute and display the welcome message.
- ▶ The class definition begins with the keyword `class` followed by the class name `GradeBook`.

©1992–2014 by Pearson Education, Inc.
All Rights Reserved.

► Defining a Class with a Member Function (cont.)

- 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.
 - Often referred to as **Pascal case**.
 - The occasional uppercase letters resemble a camel's humps. More generally, **camel case** capitalization style allows the first letter to be either lowercase or uppercase
- Every class's **body** is enclosed in a pair of left and right braces ({ and }), as in lines 9 and 16.
- The class definition terminates with a semicolon (line 16).

©1992–2014 by Pearson Education, Inc.
All Rights Reserved.



Common Programming Error 3.1

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



©1992–2014 by Pearson Education, Inc.
All Rights Reserved.



► Defining a Class with a Member Function (cont.)

- ▶ Function `main` is always called automatically when you execute a program.
- ▶ Most functions do not get called automatically.
- ▶ You must call member function `displayMessage` explicitly to tell it to perform its task.

©1992–2014 by Pearson Education, Inc.
All Rights Reserved.



► Defining a Class with a Member Function (cont.)

- ▶ The **access-specifier label `public`:** contains the keyword `public`. It is an **access specifier**.
 - Indicates that the function is “available to the public”—that is, it can be called by other functions in the program (such as `main`), and by member functions of other classes (if there are any).
 - Access specifiers are always followed by a colon (:).

©1992–2014 by Pearson Education, Inc.
All Rights Reserved.



► Defining a Class with a Member Function (cont.)

- ▶ Each function in a program performs a task and may *return a value* when it completes its task.
- ▶ 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.
- ▶ Keyword **void** to the left of the function name **displayMessage** is the function's return type.
 - Indicates that **displayMessage** will *not* return any data to its **calling function** when it completes its task.

©1992–2014 by Pearson Education, Inc.
All Rights Reserved.



► Defining a Class with a Member Function (cont.)

- ▶ The name of the member function, **displayMessage**, follows the return type.
- ▶ By convention, our function names use the *camel case* style with a lowercase first letter.
- ▶ The parentheses after the member function name indicate that it is a *function*.

©1992–2014 by Pearson Education, Inc.
All Rights Reserved.

► Defining a Class with a Member Function (cont.)

- ▶ Empty parentheses indicate that a member function does not require additional data to perform its task.
- ▶ The first line of a function definition is commonly called the **function header**.
- ▶ Every function's *body* is delimited by left and right braces ({ and }).
- ▶ The *function body* contains statements that perform the function's task.

©1992–2014 by Pearson Education, Inc.
All Rights Reserved.

► Defining a Class with a Member Function (cont.)

Testing Class GradeBook

- ▶ Typically, you cannot call a member function of a class until you create an object of that class.
- ▶ First, create an object of class **GradeBook** called **myGradeBook**.
 - The variable's type is **GradeBook**.
 - The compiler does not automatically know what type **GradeBook** is—it's a **user-defined type**.
 - Tell the compiler what **GradeBook** is by including the class definition.
 - Each class you create becomes a new type that can be used to create objects.

©1992–2014 by Pearson Education, Inc.
All Rights Reserved.

► Defining a Class with a Member Function (cont.)



- ▶ Call the member function `displayMessage`- by using variable `myGradeBook` followed by the dot operator `(.)`, the function name `display-Message` and an empty set of parentheses.
- ▶ Causes the `displayMessage` function to perform its task.

©1992–2014 by Pearson Education, Inc.
All Rights Reserved.

► Defining a Class with a Member Function (cont.)



*UML Class Diagram for Class **GradeBook***

- ▶ In the UML, each class is modeled in a [UML class diagram](#) as a *rectangle* with three *compartments*.
- ▶ Figure 3.2 presents a class diagram for class **GradeBook** (Fig. 3.1).
- ▶ The *top compartment* contains the class's name centered horizontally and in boldface type.
- ▶ The *middle compartment* contains the class's attributes, which correspond to data members in C++.
 - Currently empty, because class **GradeBook** does not yet have any attributes.

©1992–2014 by Pearson Education, Inc.
All Rights Reserved.

► Defining a Class with a Member Function (cont.)

UML Class Diagram for Class GradeBook

- ▶ 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 plus sign (+) in front of the operation name indicates that `displayMessage` is a public operation in the UML.

©1992–2014 by Pearson Education, Inc.
All Rights Reserved.

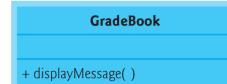


Fig. 3.2 | UML class diagram indicating that class GradeBook has a public `displayMessage` operation.

©1992–2014 by Pearson Education, Inc.
All Rights Reserved.





► Defining a Member Function with a Parameter

- ▶ Car analogy
 - Pressing a car's gas pedal sends a *message* to the car to perform a task—make 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.
 - The message to the car includes *both* the *task to perform* and *additional information that helps the car perform the task*.

- ▶ Additional information that a function needs to perform its task is known as a **parameter**.
- ▶ A function call supplies values—called **arguments**—for each of the function's parameters.

©1992–2014 by Pearson Education, Inc.
All Rights Reserved.



► Defining a Member Function with a Parameter (cont.)

- ▶ Fig. 3.3 redefines class **GradeBook** (lines 9–18) with a **displayMessage** member function (lines 13–17) that displays the course name as part of the welcome message.
 - The new version of **displayMessage** requires a *parameter* (**courseName** in line 13) that represents the course name to output.
- ▶ A variable of type **string** represents a string of characters.
- ▶ A string is actually an object of the C++ Standard Library class **string**.
 - Defined in **header file <string>** and part of namespace **std**.
 - For now, you can think of **string** variables like variables of other types such as **int**.
 - Additional **string** capabilities in Section 3.9.

©1992–2014 by Pearson Education, Inc.
All Rights Reserved.

```

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 #include <string> // program uses C++ standard string class
6 using namespace std;
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( string courseName ) const
14     {
15         cout << "Welcome to the grade book for\n" << courseName << "!"
16         << endl;
17     } // end function displayMessage
18 }; // end class GradeBook
19

```

Fig. 3.3 | Define class GradeBook with a member function that takes a parameter, create a GradeBook object and call its displayMessage function. (Part 1 of 3.)

©1992-2014 by Pearson Education, Inc.
All Rights Reserved.

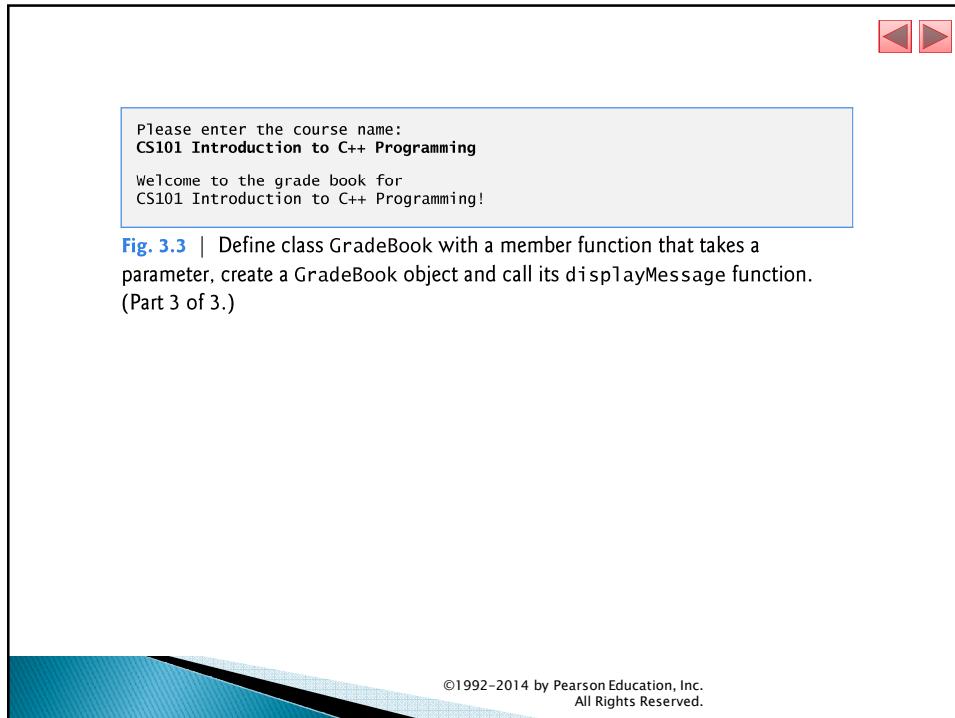
```

20 // function main begins program execution
21 int main()
22 {
23     string nameOfCourse; // string of characters to store the course name
24     GradeBook myGradeBook; // create a GradeBook object named myGradeBook
25
26     // prompt for and input course name
27     cout << "Please enter the course name:" << endl;
28     getline( cin, nameOfCourse ); // read a course name with blanks
29     cout << endl; // output a blank line
30
31     // call myGradeBook's displayMessage function
32     // and pass nameOfCourse as an argument
33     myGradeBook.displayMessage( nameOfCourse );
34 } // end main

```

Fig. 3.3 | Define class GradeBook with a member function that takes a parameter, create a GradeBook object and call its displayMessage function. (Part 2 of 3.)

©1992-2014 by Pearson Education, Inc.
All Rights Reserved.



► Defining a Member Function with a Parameter (cont.)

- ▶ Library function `getline` reads a line of text into a **string**.
- ▶ The function call `getline(cin, nameOfCourse)` 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.
- ▶ When you press *Enter* while entering data, a newline is inserted in the input stream.
- ▶ The `<string>` header file must be included in the program to use function `getline`.

At the bottom of the window, there is a decorative footer with a blue and black diagonal pattern.

► Defining a Member Function with a Parameter (cont.)



- ▶ Line 33 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 13.

©1992–2014 by Pearson Education, Inc.
All Rights Reserved.

► Defining a Member Function with a Parameter (cont.)



- ▶ 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* to indicate that a function does *not* require any parameters.
- ▶ Each parameter must specify a type and an identifier.

©1992–2014 by Pearson Education, Inc.
All Rights Reserved.

► Defining a Member Function with a Parameter (cont.)



- ▶ A function can specify multiple parameters by separating each parameter from the next with a comma.
- ▶ 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.
- ▶ The argument types in the function call must be consistent with the types of the corresponding parameters in the function header.

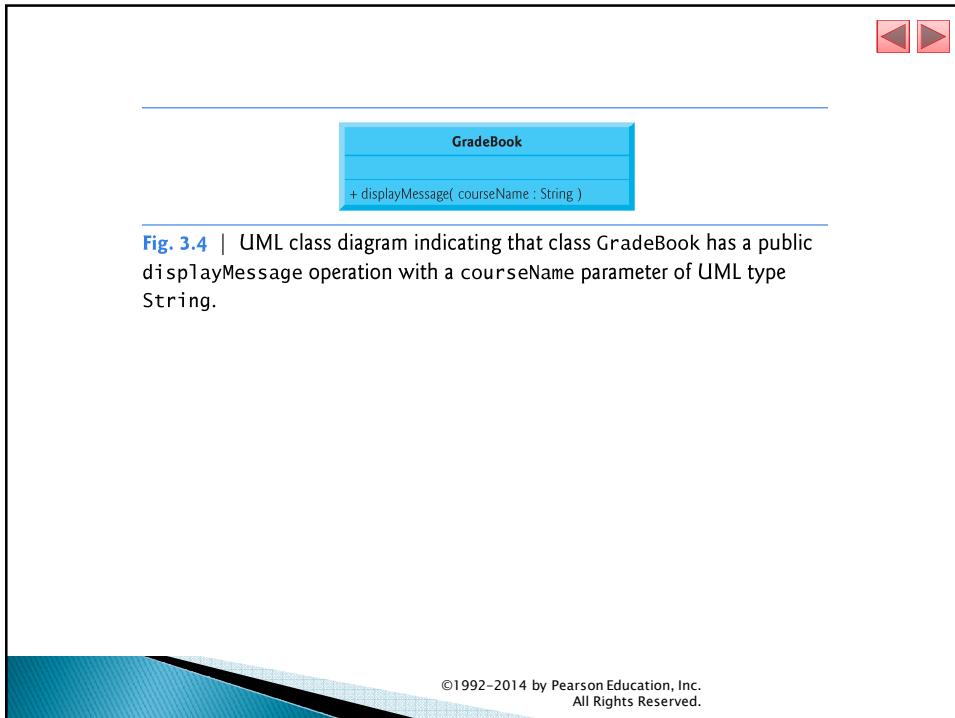
©1992–2014 by Pearson Education, Inc.
All Rights Reserved.

► Defining a Member Function with a Parameter (cont.)



- ▶ The UML class diagram of Fig. 3.4 models class **GradeBook** of Fig. 3.3.
- ▶ 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 independent*—it's used with many different programming languages—so its terminology does not exactly match that of C++.

©1992–2014 by Pearson Education, Inc.
All Rights Reserved.



► Data Members, set Member Functions and get Member Functions

- ▶ 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 closing right brace (**{**) of the block in which they're declared.
 - 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's declared.
 - *When a function terminates, the values of its local variables are lost.*

©1992–2014 by Pearson Education, Inc.
All Rights Reserved.

► Data Members, set Member Functions and get Member Functions (Cont.)

- ▶ An object has attributes that are carried with it as it's 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 attributes in memory.

©1992–2014 by Pearson Education, Inc.
All Rights Reserved.

► Data Members, set Member Functions and get Member Functions (Cont.)

- ▶ A typical instructor teaches several courses, each with its own course name.
- ▶ A variable that is declared in the class definition but outside the bodies of the class's member-function definitions is a *data member*.
- ▶ Every instance (i.e., object) of a class contains each of the class's data members.
- ▶ A benefit of making a variable a data member is that all the member functions of the class can manipulate any data members that appear in the class definition.

©1992–2014 by Pearson Education, Inc.
All Rights Reserved.

```

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 #include <string> // program uses C++ standard string class
7 using namespace std;
8
9 // GradeBook class definition
10 class GradeBook
11 {
12 public:
13     // function that sets the course name
14     void setCourseName( string name )
15     {
16         courseName = name; // store the course name in the object
17     } // end function setCourseName
18
19     // function that gets the course name
20     string getCourseName() const
21     {
22         return courseName; // return the object's courseName
23     } // end function getCourseName

```

Fig. 3.5 | Defining and testing class GradeBook with a data member and *set* and *get* member functions. (Part I of 3.)

©1992-2014 by Pearson Education, Inc.
All Rights Reserved.

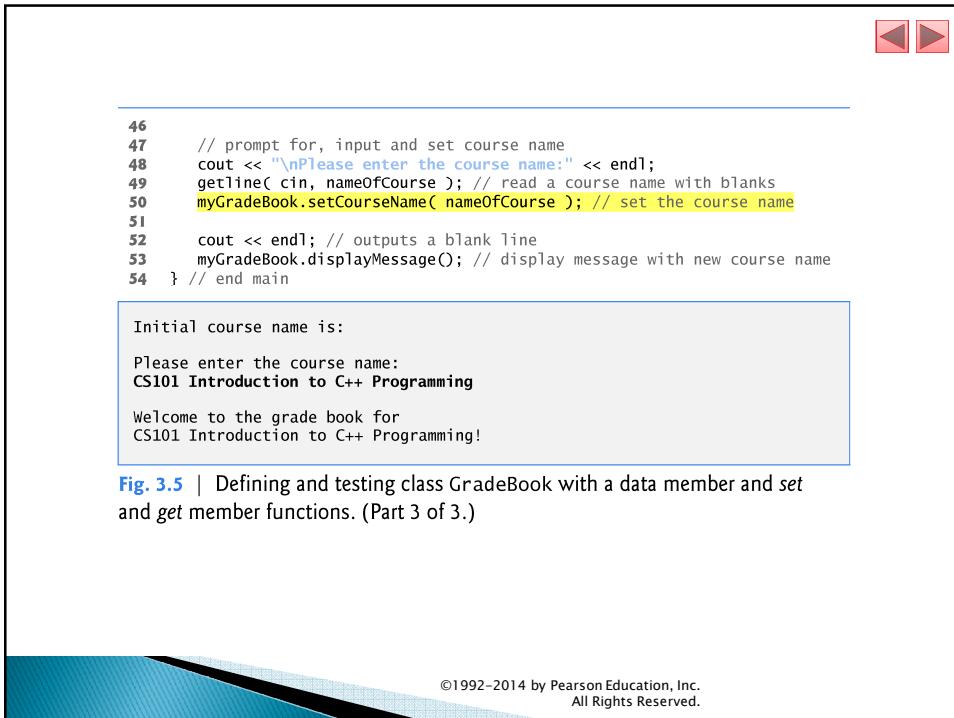
```

24
25     // function that displays a welcome message
26     void displayMessage() const
27     {
28         // this statement calls getCourseName to get the
29         // name of the course this GradeBook represents
30         cout << "Welcome to the grade book for\n" << getCourseName() << "!"
31         << endl;
32     } // end function displayMessage
33 private:
34     string courseName; // course name for this GradeBook
35 }; // end class GradeBook
36
37 // function main begins program execution
38 int main()
39 {
40     string nameOfCourse; // string of characters to store the course name
41     GradeBook myGradeBook; // create a GradeBook object named myGradeBook
42
43     // display initial value of courseName
44     cout << "Initial course name is: " << myGradeBook.getCourseName()
45     << endl;

```

Fig. 3.5 | Defining and testing class GradeBook with a data member and *set* and *get* member functions. (Part 2 of 3.)

©1992-2014 by Pearson Education, Inc.
All Rights Reserved.



The screenshot shows a C++ code editor with the following code:

```

46 // prompt for, input and set course name
47 cout << "\nPlease enter the course name:" << endl;
48 getline( cin, nameOfCourse ); // read a course name with blanks
49 myGradeBook.setCourseName( nameOfCourse ); // set the course name
50
51 cout << endl; // outputs a blank line
52 myGradeBook.displayMessage(); // display message with new course name
53
54 } // end main

```

Below the code, the terminal window displays the following output:

```

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!

```

Fig. 3.5 | Defining and testing class GradeBook with a data member and *set* and *get* member functions. (Part 3 of 3.)

©1992–2014 by Pearson Education, Inc.
All Rights Reserved.

► Data Members, *set* Member Functions and *get* Member Functions (Cont.)

- ▶ Most data-member declarations appear after the access-specifier label **private**:
- ▶ 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're declared.

©1992–2014 by Pearson Education, Inc.
All Rights Reserved.

► Data Members, set Member Functions and get Member Functions (Cont.)

- ▶ 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.

©1992–2014 by Pearson Education, Inc.
All Rights Reserved.



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 friends of the class.



©1992–2014 by Pearson Education, Inc.
All Rights Reserved.



Common Programming Error 3.2

An attempt by a function, which is not a member of a particular class (or a friend of that class) to access a **private** member of that class is a compilation error.

©1992–2014 by Pearson Education, Inc.
All Rights Reserved.



► Data Members, set Member Functions and get Member Functions (Cont.)

- ▶ Declaring data members with access specifier **private** is known as **data hiding**.
- ▶ When a program creates (instantiates) an object, its data members are encapsulated (hidden) in the object and can be accessed only by member functions of the object's class.

©1992–2014 by Pearson Education, Inc.
All Rights Reserved.

► Data Members, set Member Functions and get Member Functions (Cont.)

- ▶ In this example, `setCourseName` does not attempt to validate the course name—i.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.
- ▶ When a function that specifies a return type other than `void` is called and completes its task, the function uses a `return statement` to return a result to its calling function.

©1992–2014 by Pearson Education, Inc.
All Rights Reserved.

► Data Members, set Member Functions and get Member Functions (Cont.)

- ▶ Member function `displayMessage` (lines 26–32) 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.

©1992–2014 by Pearson Education, Inc.
All Rights Reserved.

► Data Members, set Member Functions and get Member Functions (Cont.)

- ▶ Line 30 calls member function `getCourseName` to obtain the value of `courseName`.
 - Member function `displayMessage` could also access data member `courseName` directly, just as member functions `setCourseName` and `getCourseName` do.
- ▶ By 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.

©1992–2014 by Pearson Education, Inc.
All Rights Reserved.

► Data Members, set Member Functions and get Member Functions (Cont.)

- ▶ A **client of an object**—that is, any class or function that calls the object’s member functions from *outside* the object—calls the class’s `public` member functions to request the class’s services for particular objects of the class.
 - This is why the statements in `main` call member functions `setCourseName`, `getCourseName` and `displayMessage` on a `GradeBook` object.

©1992–2014 by Pearson Education, Inc.
All Rights Reserved.

► Data Members, `set` Member Functions and `get` Member Functions (Cont.)

- ▶ 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.
 - These member function names need not begin with `set` or `get`, but this naming convention is common.
- ▶ 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).

©1992–2014 by Pearson Education, Inc.
All Rights Reserved.



Good Programming Practice 3.1

Always try to localize the effects of changes to a class's data members by accessing and manipulating the data members through their corresponding `get` and `set` functions.



©1992–2014 by Pearson Education, Inc.
All Rights Reserved.



Software Engineering Observation 3.1

Write programs that are clear and easy to maintain.
Change is the rule rather than the exception. You should anticipate that your code will be modified, and possibly often.

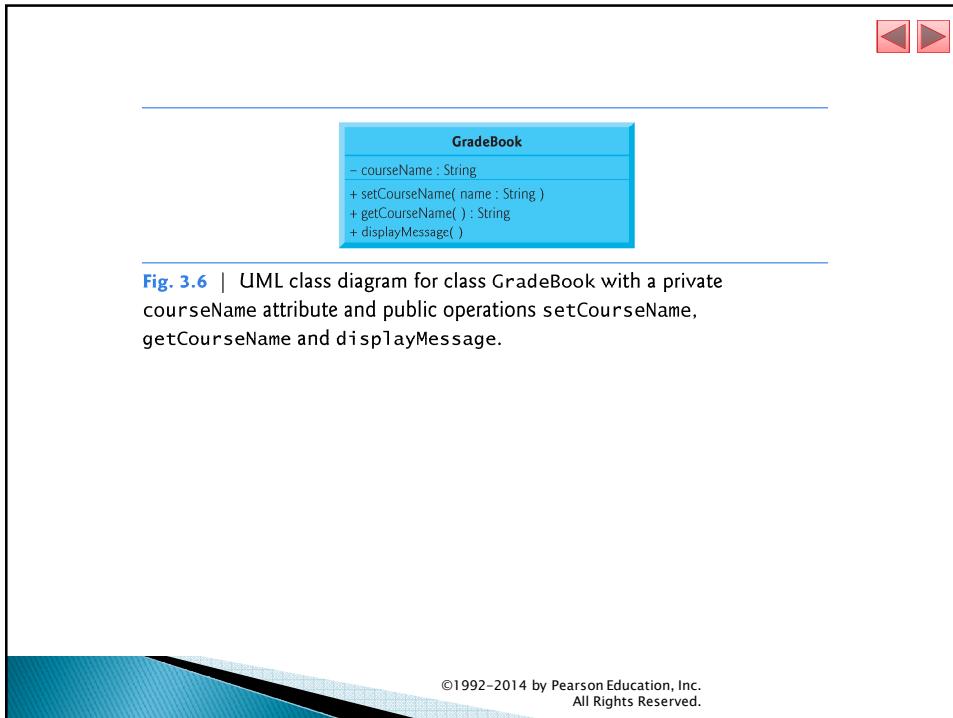


©1992–2014 by Pearson Education, Inc.
All Rights Reserved.

3.4 Data Members, set Member Functions and get Member Functions (Cont.)

- ▶ Figure 3.6 contains an updated UML class diagram for the version of class **GradeBook** in Fig. 3.5.
- ▶ The UML represents data members as attributes by listing the attribute name, followed by a colon and the attribute type.

©1992–2014 by Pearson Education, Inc.
All Rights Reserved.



5 Initializing Objects with Constructors

- ▶ Each class can provide one or more **constructors** 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.

©1992–2014 by Pearson Education, Inc.
All Rights Reserved.



5 Initializing Objects with Constructors

- ▶ 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`.

©1992–2014 by Pearson Education, Inc.
All Rights Reserved.



5 Initializing Objects with Constructors (cont.)

- ▶ C++ automatically calls a constructor for each object that is created, which helps ensure that objects are initialized properly before they're used in a program.
- ▶ The constructor call occurs when the object is created.
- ▶ If a class does not *explicitly* include constructors, the compiler provides a `default constructor` with *no* parameters.

©1992–2014 by Pearson Education, Inc.
All Rights Reserved.

```

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 #include <string> // program uses C++ standard string class
7 using namespace std;
8
9 // GradeBook class definition
10 class GradeBook
11 {
12 public:
13     // constructor initializes courseName with string supplied as argument
14     explicit GradeBook( string name )
15         : courseName( name ) // member initializer to initialize courseName
16     {
17         // empty body
18     } // end GradeBook constructor
19

```

Fig. 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. (Part 1 of 3.)

©1992-2014 by Pearson Education, Inc.
All Rights Reserved.

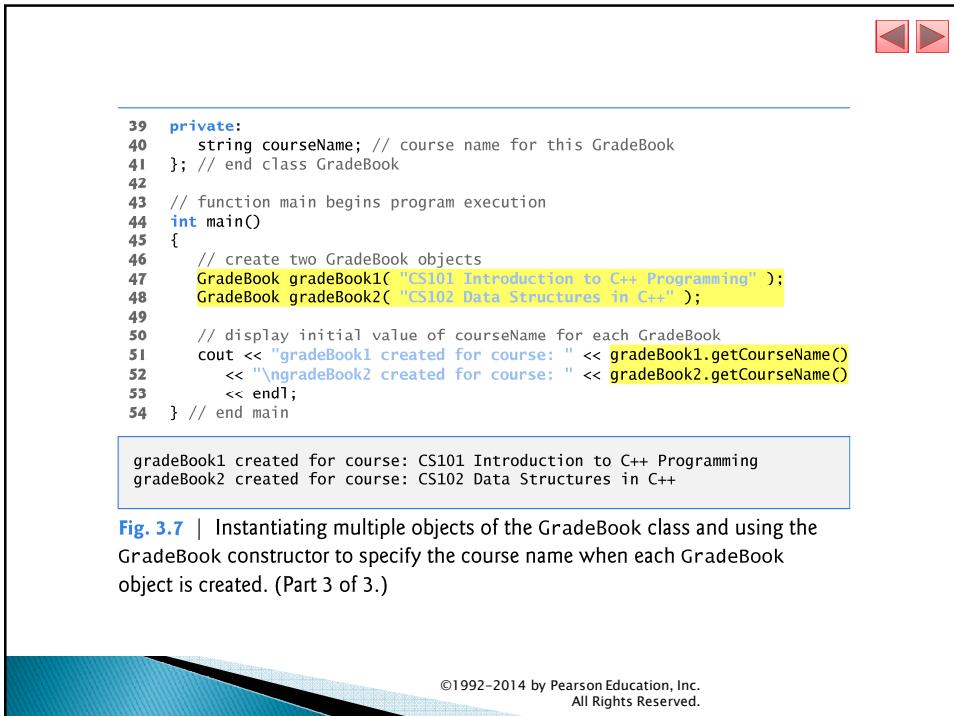
```

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() const
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() const
34     {
35         // call getCourseName to get the courseName
36         cout << "Welcome to the grade book for " << getCourseName()
37         << "!" << endl;
38     } // end function displayMessage

```

Fig. 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. (Part 2 of 3.)

©1992-2014 by Pearson Education, Inc.
All Rights Reserved.



The screenshot shows a C++ code editor with the following code:

```

39 private:
40     string courseName; // course name for this GradeBook
41 } // end class GradeBook
42
43 // function main begins program execution
44 int main()
45 {
46     // create two GradeBook objects
47     GradeBook gradeBook1( "CS101 Introduction to C++ Programming" );
48     GradeBook gradeBook2( "CS102 Data Structures in C++" );
49
50     // display initial value of courseName for each GradeBook
51     cout << "gradeBook1 created for course: " << gradeBook1.getCourseName()
52         << "\ngradeBook2 created for course: " << gradeBook2.getCourseName()
53         << endl;
54 } // end main

```

A yellow box highlights the constructor calls in lines 47 and 48. A blue box highlights the output text in the terminal window below:

```

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

```

Fig. 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. (Part 3 of 3.)

©1992-2014 by Pearson Education, Inc.
All Rights Reserved.

5 Initializing Objects with Constructors (cont.)

- ▶ 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.
- ▶ The constructor uses a **member-initializer list** (line 15) to initialize the **courseName** data member with the value of the constructor's parameter name.

©1992-2014 by Pearson Education, Inc.
All Rights Reserved.

5 Initializing Objects with Constructors (cont.)



- ▶ *Member initializers* appear between a constructor's parameter list and the left brace that begins the constructor's body.
- ▶ The member initializer list is separated from the parameter list with a *colon* (:).

©1992–2014 by Pearson Education, Inc.
All Rights Reserved.

5 Initializing Objects with Constructors (cont.)



- ▶ A member initializer consists of a data member's *variable name* followed by parentheses containing the member's *initial value*.
- ▶ In this example, `courseName` is initialized with the value of the parameter `name`.
- ▶ If a class contains more than one data member, each data member's initializer is separated from the next by a comma.
- ▶ The member initializer list executes *before* the body of the constructor executes.

©1992–2014 by Pearson Education, Inc.
All Rights Reserved.

5 Initializing Objects with Constructors (cont.)



- ▶ Line 47 creates and initializes a **GradeBook** object called **gradeBook1**.
 - When this line executes, the **GradeBook** constructor (lines 14–18) is called with the argument "**CS101 Introduction to C++ Programming**" to initialize **gradeBook1**'s course name.

- ▶ Line 48 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.

©1992–2014 by Pearson Education, Inc.
All Rights Reserved.

5 Initializing Objects with Constructors (cont.)



- ▶ Any constructor that takes *no* arguments is called a default constructor.

- ▶ A class gets a default constructor in one of several ways:
 - The compiler *implicitly* creates a default constructor in every class that does *not* have any user-defined constructors. The 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. An uninitialized variable contains an undefined ("garbage") value.

 - You *explicitly* define a constructor that takes no arguments. Such a default constructor will call the default constructor for each data member that is an object of another class and will perform additional initialization specified by you.

 - *If you define any constructors with arguments, C++ will not implicitly create a default constructor for that class.*

©1992–2014 by Pearson Education, Inc.
All Rights Reserved.

5 Initializing Objects with Constructors (cont.)



- ▶ 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's customary to list the class's constructor before other operations in the third compartment.

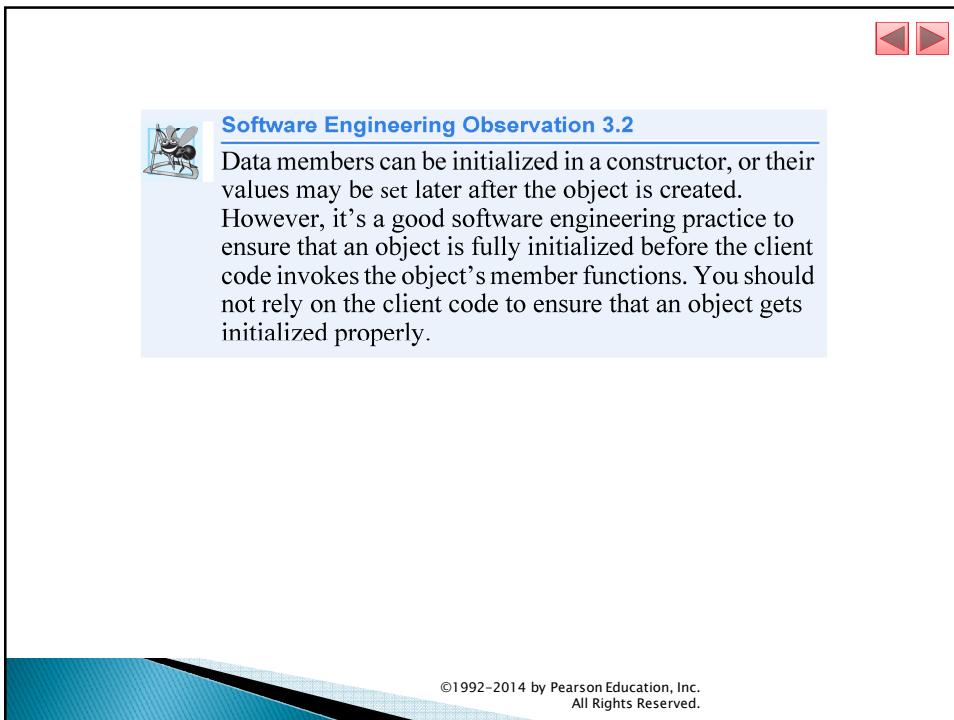
©1992–2014 by Pearson Education, Inc.
All Rights Reserved.



Error-Prevention Tip 3.2

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

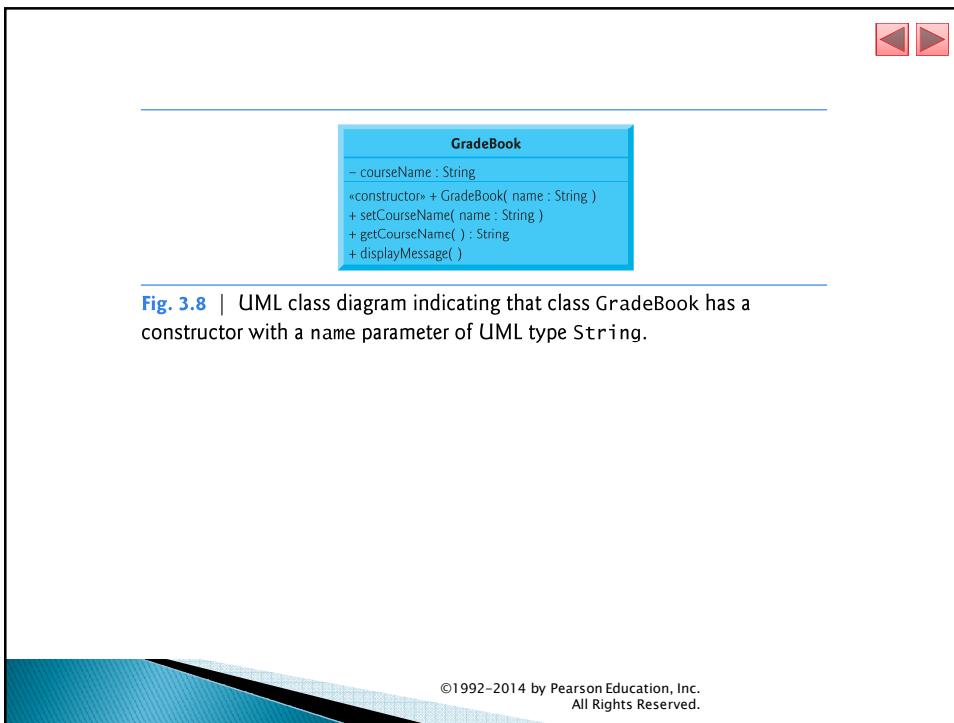
©1992–2014 by Pearson Education, Inc.
All Rights Reserved.



Software Engineering Observation 3.2

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

©1992–2014 by Pearson Education, Inc.
All Rights Reserved.



GradeBook

– courseName : String
«constructor» + GradeBook(name : String)
+ setCourseName(name : String)
+ getCourseName() : String
+ displayMessage()

©1992–2014 by Pearson Education, Inc.
All Rights Reserved.

6 Placing a Class in a Separate File for Reusability



- ▶ One of the benefits of creating class definitions is that, when packaged properly, our classes can be reused by programmers—potentially worldwide.
- ▶ Programmers who wish to use our **GradeBook** class cannot simply include the file from Fig. 3.7 in another program.
 - Function **main** begins the execution of every program, and every program must have exactly one **main** function.

©1992–2014 by Pearson Education, Inc.
All Rights Reserved.

6 Placing a Class in a Separate File for Reusability (cont.)



- ▶ 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's customary to define *reusable* source code (such as a class) in a file that by convention has a **.h** filename extension—known as a **header**.
- ▶ Programs use **#include** preprocessing directives to include header files and take advantage of reusable software components.

©1992–2014 by Pearson Education, Inc.
All Rights Reserved.

6 Placing a Class in a Separate File for Reusability (cont.)

- ▶ Our next example separates 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 7–38) and the headers on which the class depends.
 - The **main** function that *uses* class **GradeBook** is defined in the source-code file **fig03_10.cpp** (Fig. 3.10) in lines 8–18.
- ▶ To help you prepare for the larger programs you'll encounter later in industry, we often use a separate source-code file containing function **main** to test our classes (this is called a **driver program**).

©1992–2014 by Pearson Education, Inc.
All Rights Reserved.

```

1 // Fig. 3.9: GradeBook.h
2 // GradeBook class definition in a separate file from main.
3 #include <iostream>
4 #include <string> // class GradeBook uses C++ standard string class
5
6 // GradeBook class definition
7 class GradeBook
8 {
9 public:
10    // constructor initializes courseName with string supplied as argument
11    explicit GradeBook( std::string name )
12    : courseName( name ) // member initializer to initialize courseName
13    {
14        // empty body
15    } // end GradeBook constructor
16
17    // function to set the course name
18    void setCourseName( std::string name )
19    {
20        courseName = name; // store the course name in the object
21    } // end function setCourseName
22

```

Fig. 3.9 | GradeBook class definition in a separate file from main. (Part I of 2.)

©1992–2014 by Pearson Education, Inc.
All Rights Reserved.

```

23 // function to get the course name
24 std::string getCourseName() const
25 {
26     return courseName; // return object's courseName
27 } // end function getCourseName
28
29 // display a welcome message to the GradeBook user
30 void displayMessage() const
31 {
32     // call getCourseName to get the courseName
33     std::cout << "Welcome to the grade book for\n" << getCourseName()
34     << "!" << std::endl;
35 } // end function displayMessage
36 private:
37     std::string courseName; // course name for this GradeBook
38 }; // end class GradeBook

```

Fig. 3.9 | GradeBook class definition in a separate file from main. (Part 2 of 2.)

©1992-2014 by Pearson Education, Inc.
All Rights Reserved.

```

1 // Fig. 3.10: fig03_10.cpp
2 // Including class GradeBook from file GradeBook.h for use in main.
3 #include <iostream>
4 #include "GradeBook.h" // include definition of class GradeBook
5 using namespace std;
6
7 // function main begins program execution
8 int main()
9 {
10    // create two GradeBook objects
11    GradeBook gradeBook1( "CS101 Introduction to C++ Programming" );
12    GradeBook gradeBook2( "CS102 Data Structures in C++" );
13
14    // display initial value of courseName for each GradeBook
15    cout << "gradeBook1 created for course: " << gradeBook1.getCourseName()
16    << "\ngradeBook2 created for course: " << gradeBook2.getCourseName()
17    << endl;
18 } // end main

```

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

Fig. 3.10 | Including class GradeBook from file GradeBook.h for use in main.

©1992-2014 by Pearson Education, Inc.
All Rights Reserved.



6 Placing a Class in a Separate File for Reusability (cont.)

- ▶ Throughout the header (Fig. 3.9), we use `std::` when referring to `string` (lines 11, 18, 24 and 37), `cout` (line 33) and `endl` (line 34).
- ▶ Headers should never contain `using` directives or `using` declarations (Section 2.7).
- ▶ 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.

©1992–2014 by Pearson Education, Inc.
All Rights Reserved.



6 Placing a Class in a Separate File for Reusability (cont.)

- ▶ To help the compiler understand how to use a class, we must explicitly provide the compiler with the class's definition
 - That'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.

©1992–2014 by Pearson Education, Inc.
All Rights Reserved.

6 Placing a Class in a Separate File for Reusability (cont.)



- ▶ 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 data members, because their contents can vary among objects.
- ▶ The member-function code, however, is *not modifiable*, so it can be shared among all objects of the class.

©1992–2014 by Pearson Education, Inc.
All Rights Reserved.

6 Placing a Class in a Separate File for Reusability (cont.)



- ▶ 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 4, we give the compiler access to the information it needs to determine the size of a **GradeBook** object and to determine whether objects of the class are used correctly.

©1992–2014 by Pearson Education, Inc.
All Rights Reserved.

6 Placing a Class in a Separate File for Reusability (cont.)

- ▶ A `#include` directive instructs the C++ preprocessor to replace the directive with a copy of the contents of `GradeBook.h` 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.

©1992–2014 by Pearson Education, Inc.
All Rights Reserved.

6 Placing a Class in a Separate File for Reusability (cont.)

- ▶ Notice that the name of the `GradeBook.h` header file in line 4 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, it attempts to locate the header file in the same directory as the file in which the `#include` directive appears.

©1992–2014 by Pearson Education, Inc.
All Rights Reserved.

6 Placing a Class in a Separate File for Reusability (cont.)



- 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.

©1992–2014 by Pearson Education, Inc.
All Rights Reserved.



Error-Prevention Tip 3.3

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



©1992–2014 by Pearson Education, Inc.
All Rights Reserved.

6 Placing a Class in a Separate File for Reusability (cont.)



- ▶ Placing a class definition in a header file reveals the entire implementation of the class to the class's clients.
- ▶ 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.

©1992–2014 by Pearson Education, Inc.
All Rights Reserved.

6 Placing a Class in a Separate File for Reusability (cont.)



- ▶ 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.

©1992–2014 by Pearson Education, Inc.
All Rights Reserved.



7 Separating Interface from Implementation

- ▶ **Interfaces** define and standardize the ways in which things such as people and systems interact with one another.
- ▶ 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 **public** interface consists of the class's **public** member functions (also known as the class's **public services**).

©1992–2014 by Pearson Education, Inc.
All Rights Reserved.



7 Separating Interface from Implementation (cont.)

- ▶ 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.
- ▶ It's better software engineering to define member functions outside the class definition, so that their implementation details can be hidden from the client code.
 - Ensures that you do not write client code that depends on the class's implementation details.
- ▶ The program of Figs. 3.11–3.13 separates class **GradeBook**'s interface from its implementation by splitting the class definition of Fig. 3.9 into two files—the 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.

©1992–2014 by Pearson Education, Inc.
All Rights Reserved.



7 Separating Interface from Implementation (cont.)

- ▶ 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.
- ▶ Figure 3.14 shows how this three-file program is compiled from the perspectives of the **GradeBook** class programmer and the client-code programmer—we'll explain this figure in detail.

©1992–2014 by Pearson Education, Inc.
All Rights Reserved.



7 Separating Interface from Implementation (cont.)

- ▶ Header file **GradeBook.h** (Fig. 3.11) is similar to the one in Fig. 3.9, but the function definitions in Fig. 3.9 are replaced here with **function prototypes** (lines 11–14) 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.
- ▶ Including the header file **GradeBook.h** in the client code (line 5 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.

©1992–2014 by Pearson Education, Inc.
All Rights Reserved.

```

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
7 // GradeBook class definition
8 class GradeBook
9 {
10 public:
11     explicit GradeBook( std::string ); // constructor initialize courseName
12     void setCourseName( std::string ); // sets the course name
13     std::string getCourseName() const; // gets the course name
14     void displayMessage() const; // displays a welcome message
15 private:
16     std::string courseName; // course name for this GradeBook
17 };

```

Fig. 3.11 | GradeBook class definition containing function prototypes that specify the interface of the class.

©1992–2014 by Pearson Education, Inc.
All Rights Reserved.

Good Programming Practice 3.2

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

©1992–2014 by Pearson Education, Inc.
All Rights Reserved.

7 Separating Interface from Implementation (cont.)

- ▶ Source-code file **GradeBook.cpp** (Fig. 3.12) *defines* class **GradeBook**'s member functions, which were declared in lines 11–14 of Fig. 3.11.
- ▶ Each member-function name (lines 9, 16, 22 and 28) is preceded by the class name and `::`, which is known as the **scope resolution operator**.
- ▶ This “ties” each member function to the (now separate) **GradeBook** class definition (Fig. 3.11), which declares the class’s member functions and data members.

©1992–2014 by Pearson Education, Inc.
All Rights Reserved.

```

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 #include "GradeBook.h" // include definition of class GradeBook
6 using namespace std;
7
8 // constructor initializes courseName with string supplied as argument
9 GradeBook::GradeBook( string name )
10 : courseName( name ) // member initializer to initialize courseName
11 {
12     // empty body
13 } // end GradeBook constructor
14
15 // function to set the course name
16 void GradeBook::setCourseName( string name )
17 {
18     courseName = name; // store the course name in the object
19 } // end function setCourseName
20

```

Fig. 3.12 | GradeBook member-function definitions represent the implementation of class GradeBook. (Part I of 2.)

©1992–2014 by Pearson Education, Inc.
All Rights Reserved.

```
21 // function to get the course name
22 string GradeBook::getCourseName() const
23 {
24     return courseName; // return object's courseName
25 } // end function getCourseName
26
27 // display a welcome message to the GradeBook user
28 void GradeBook::displayMessage() const
29 {
30     // call getCourseName to get the courseName
31     cout << "Welcome to the grade book for\n" << getCourseName()
32     << "!" << endl;
33 } // end function displayMessage
```

Fig. 3.12 | GradeBook member-function definitions represent the implementation of class GradeBook. (Part 2 of 2.)

©1992–2014 by Pearson Education, Inc.
All Rights Reserved.



Common Programming Error 3.3

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

©1992–2014 by Pearson Education, Inc.
All Rights Reserved.



7 Separating Interface from Implementation (cont.)

- ▶ To indicate that the member functions in **GradeBook.cpp** are part of class **GradeBook**, we must first include the **GradeBook.h** header file (line 5 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
 - the first line of each member function matches its prototype in the **GradeBook.h** file, and that
 - each member function knows about the class's data members and other member functions

©1992-2014 by Pearson Education, Inc.
All Rights Reserved.

```

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

```

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

Fig. 3.13 | GradeBook class demonstration after separating its interface from its implementation.

©1992-2014 by Pearson Education, Inc.
All Rights Reserved.

7 Separating Interface from Implementation (cont.)

- Before executing this program, the source-code files in Fig. 3.12 and Fig. 3.13 must both be compiled, then linked together—that is, the member-function calls in the client code need to be tied to the implementations of the class's member functions—a job performed by the linker.
- 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.

©1992–2014 by Pearson Education, Inc.
All Rights Reserved.

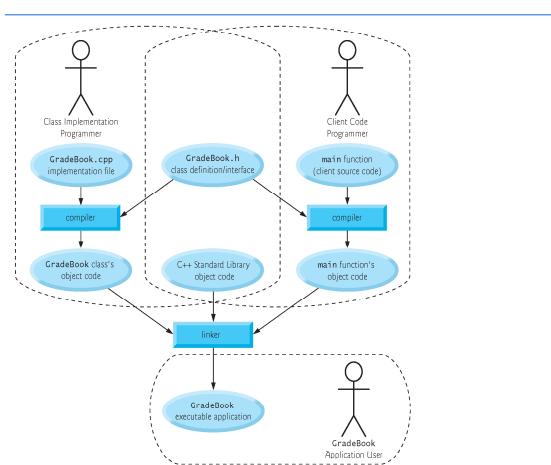


Fig. 3.14 | Compilation and linking process that produces an executable application.

©1992–2014 by Pearson Education, Inc.
All Rights Reserved.

8 Validating Data with *set Functions*

- ▶ The program of Figs. 3.15–3.17 enhances class **GradeBook**'s member function **setCourseName** to perform **validation** (also known as **validity checking**).
- ▶ Since the interface of the class 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.

©1992–2014 by Pearson Education, Inc.
All Rights Reserved.

```

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 <iostream> // program uses C++ standard string class
5
6 // GradeBook class definition
7 class GradeBook
8 {
9 public:
10    explicit GradeBook( std::string ); // constructor initializes courseName
11    void setCourseName( std::string ); // sets the course name
12    std::string getCourseName() const; // gets the course name
13    void displayMessage() const; // displays a welcome message
14 private:
15    std::string courseName; // course name for this GradeBook
16 }; // end class GradeBook

```

Fig. 3.15 | GradeBook class definition presents the public interface of the class.

©1992–2014 by Pearson Education, Inc.
All Rights Reserved.

8 Validating Data with *set Functions* (cont.)



- ▶ The C++ Standard Library's **string** class defines a member function **length** that returns the number of characters in a **string** object.
- ▶ A **consistent state** is a state in which the object's data member contains a valid value.
- ▶ 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 first argument specifies the starting position in the original **string** from which characters are copied.
 - The second argument specifies the number of characters to copy.

©1992–2014 by Pearson Education, Inc.
All Rights Reserved.

8 Validating Data with *set Functions* (cont.)



- ▶ Figure 3.17 demonstrates the modified version of class **GradeBook** (Figs. 3.15–3.16) featuring validation.
- ▶ 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.

©1992–2014 by Pearson Education, Inc.
All Rights Reserved.

```

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

```

Fig. 3.16 | Member-function definitions for class GradeBook with a *set* function that validates the length of data member courseName. (Part 1 of 3.)

©1992-2014 by Pearson Education, Inc.
All Rights Reserved.

```

14 // function that sets the course name;
15 // ensures that the course name has at most 25 characters
16 void GradeBook::setCourseName( string name )
17 {
18     if ( name.size() <= 25 ) // if name has 25 or fewer characters
19         courseName = name; // store the course name in the object
20
21     if ( name.size() > 25 ) // if name has more than 25 characters
22     {
23         // set courseName to first 25 characters of parameter name
24         courseName = name.substr( 0, 25 ); // start at 0, length of 25
25
26         cerr << "Name '" << name << "' exceeds maximum length (25).\\n"
27             << "Limiting courseName to first 25 characters.\\n" << endl;
28     } // end if
29 } // end function setCourseName
30
31 // function to get the course name
32 string GradeBook::getCourseName() const
33 {
34     return courseName; // return object's courseName
35 } // end function getCourseName

```

Fig. 3.16 | Member-function definitions for class GradeBook with a *set* function that validates the length of data member courseName. (Part 2 of 3.)

©1992-2014 by Pearson Education, Inc.
All Rights Reserved.

```

36 // display a welcome message to the GradeBook user
37 // void GradeBook::displayMessage() const
38 {
39     // call getCourseName to get the courseName
40     cout << "Welcome to the grade book for\n" << getCourseName()
41     << "!" << endl;
42 } // end function displayMessage

```

Fig. 3.16 | Member-function definitions for class GradeBook with a *set* function that validates the length of data member courseName. (Part 3 of 3.)

©1992–2014 by Pearson Education, Inc.
All Rights Reserved.

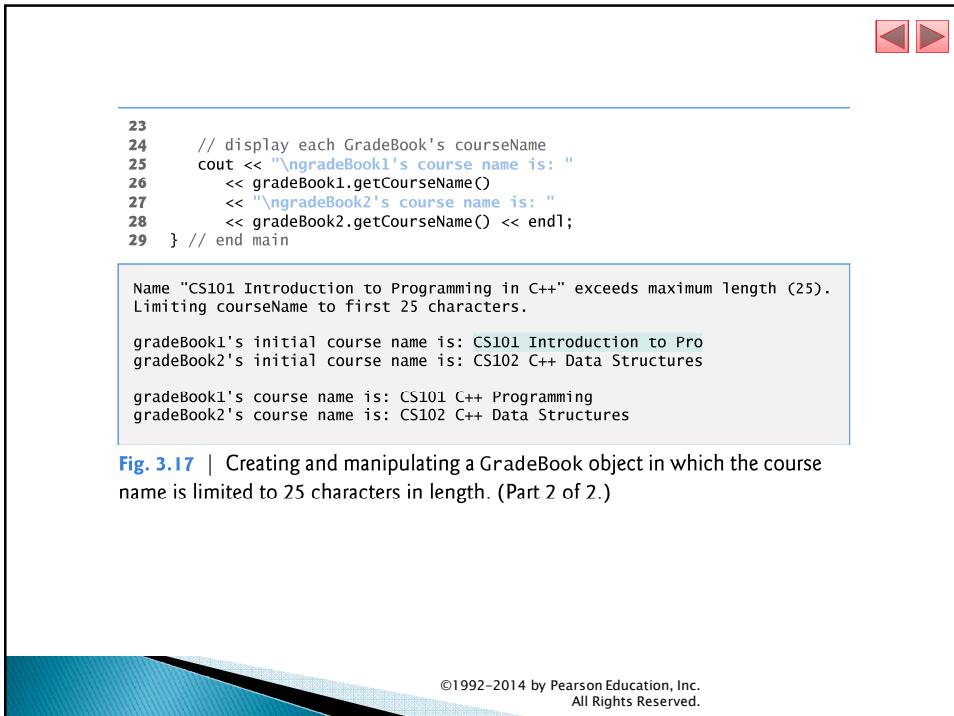
```

1 // Fig. 3.17: fig03_17.cpp
2 // Create and manipulate a GradeBook object; illustrate validation.
3 #include <iostream>
4 #include "GradeBook.h" // include definition of class GradeBook
5 using namespace std;
6
7 // function main begins program execution
8 int main()
9 {
10    // create two GradeBook objects;
11    // initial course name of gradeBook1 is too long
12    GradeBook gradeBook1( "CS101 Introduction to Programming in C++" );
13    GradeBook gradeBook2( "CS102 C++ Data Structures" );
14
15    // display each GradeBook's courseName
16    cout << "gradeBook1's initial course name is: "
17    << gradeBook1.getCourseName()
18    << '\ngradeBook2's initial course name is: '
19    << gradeBook2.getCourseName() << endl;
20
21    // modify gradeBook1's courseName (with a valid-length string)
22    gradeBook1.setCourseName( "CS101 C++ Programming" );

```

Fig. 3.17 | Creating and manipulating a GradeBook object in which the course name is limited to 25 characters in length. (Part 1 of 2.)

©1992–2014 by Pearson Education, Inc.
All Rights Reserved.



The screenshot shows a C++ code editor with the following code:

```

23     // display each GradeBook's courseName
24     cout << "\ngradeBook1's course name is: "
25         << gradeBook1.getCourseName()
26     << "\ngradeBook2's course name is: "
27         << gradeBook2.getCourseName() << endl;
28 } // end main

```

A callout box highlights the line `cout << "\ngradeBook1's course name is: "` with the text:

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

The code output window shows:

```

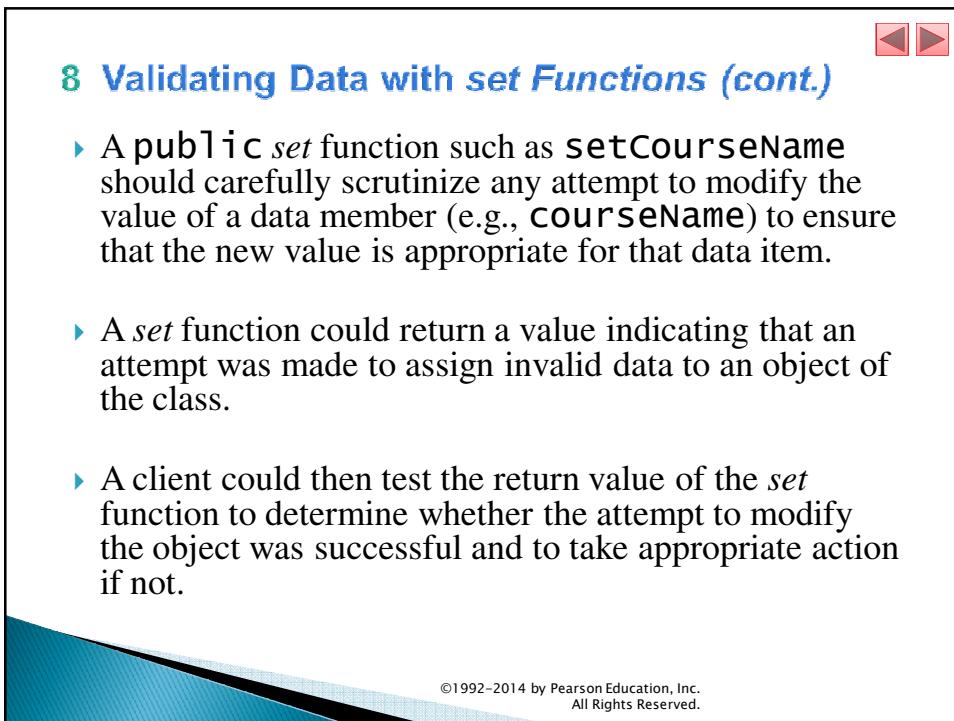
gradeBook1's initial course name is: CS101 Introduction to Pro
gradeBook2's initial course name is: CS102 C++ Data Structures

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

```

Fig. 3.17 | Creating and manipulating a GradeBook object in which the course name is limited to 25 characters in length. (Part 2 of 2.)

©1992–2014 by Pearson Education, Inc.
All Rights Reserved.



8 Validating Data with *set* Functions (cont.)

- ▶ 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.
- ▶ A *set* function could return a value indicating that an attempt was made to assign invalid data to an object of the class.
- ▶ A client could then test the return value of the *set* function to determine whether the attempt to modify the object was successful and to take appropriate action if not.

©1992–2014 by Pearson Education, Inc.
All Rights Reserved.



Software Engineering Observation 3.3

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



©1992–2014 by Pearson Education, Inc.
All Rights Reserved.



Error-Prevention Tip 3.4

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



©1992–2014 by Pearson Education, Inc.
All Rights Reserved.