```
48
49     // display initial value of courseName for each GradeBook
50     cout << "gradeBook1 created for course: " << gradeBook1.getCourseName()
51        << "\ngradeBook2 created for course: " << gradeBook2.getCourseName()
52        << endl;
53  } // end main
```

```
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 2 of 2.)

Defining a Constructor

Lines 14–17 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 46–47). Line 14 indicates that class `GradeBook`'s constructor has a `string` parameter called `name`. Line 14 does not specify a return type, because constructors cannot return values (or even `void`).

Line 16 in the constructor's body passes the constructor's parameter `name` to member function `setCourseName` (lines 20–23), which simply assigns the value of its parameter to data member `courseName`. You might be wondering why we bother making the call to `setCourseName` in line 16—the constructor certainly could perform the assignment `courseName = name`. In Section 3.9, 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. Both the constructor (line 14) and the `setCourseName` function (line 20) 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.

Testing Class `GradeBook`

Lines 43–53 of Fig. 3.7 define the `main` function that tests class `GradeBook` and demonstrates initializing `GradeBook` objects using a constructor. Line 46 creates and initializes a `GradeBook` object called `gradeBook1`. When this line executes, the `GradeBook` constructor (lines 14–17) is called (implicitly by C++) with the argument `"CS101 Introduction to C++ Programming"` to initialize `gradeBook1`'s course name. Line 47 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 50–51 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 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 typically contains a "garbage" value.
2. 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 a constructor *with* arguments, C++ will not implicitly create a default constructor for that class. 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, 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.*

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

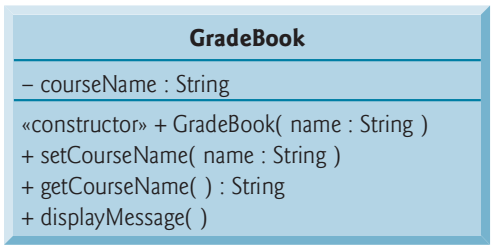

Fig. 3.8 | UML class diagram indicating that class `GradeBook` has a constructor with a `name` parameter of UML type `String`.

3.7 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. For example, we can reuse C++

Standard Library type `string` in any C++ program by including the header file `<string>` (and, as we'll see, by being able to link to the library's object code).

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 baggage—our `main` function—and their programs will then have two `main` functions. Attempting to compile a program with two `main` functions in Microsoft Visual C++ produces an error such as

```
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'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 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`.

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 8–38), the appropriate header files and a `using` declaration. 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 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'll 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.

```
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  using namespace std;
6
7  // GradeBook class definition
8  class GradeBook
9  {
```

Fig. 3.9 | `GradeBook` class definition in a separate file from `main`. (Part 1 of 2.)

```
10 public:
11    // constructor initializes courseName with string supplied as argument
12    GradeBook( string name )
13    {
14       setCourseName( name ); // call set function to initialize courseName
15    } // end GradeBook constructor
16
17    // function to set the course name
18    void setCourseName( string name )
19    {
20       courseName = name; // store the course name in the object
21    } // end function setCourseName
22
23    // function to get the course name
24    string getCourseName()
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()
31    {
32       // call getCourseName to get the courseName
33       cout << "Welcome to the grade book for\n" << getCourseName()
34          << "!" << endl;
35    } // end function displayMessage
36 private:
37    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.)

```
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
```

Fig. 3.10 | Including class `GradeBook` from file `GradeBook.h` for use in `main`. (Part 1 of 2.)

```
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`. (Part 2 of 2.)

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++ 2008 produces the linker error message:

```
error LNK2001: unresolved external symbol _mainCRTStartup
```

To compile and link with GNU C++ on Linux, you must first include the header file in a `.cpp` source-code file, then GNU C++ 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.

The compiler does not know what a `GradeBook` is because it's a user-defined type. In fact, the compiler doesn't 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 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.

To create `GradeBook` objects `gradeBook1` and `gradeBook2` in lines 11–12 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 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 4, we give the compiler access to the information it needs (Fig. 3.9, line 37) to determine the size of a `GradeBook` object and to determine whether objects of the class are used correctly (in lines 11–12 and 15–16 of Fig. 3.10).

Line 4 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 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. 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 3.8, 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, and
3. the clients do *not* know how the class's member functions are implemented.

3.8 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 differently—some 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 `public` 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'll 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's 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 you do not write client code that depends on the class's implementation details. If you were to do so, the client code would be more likely to "break" if the class's implementation changed.

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. 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 programmer—we'll explain this figure in detail.

`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 9–18). 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 12–15) 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. Also, 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 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.

```
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 namespace std;
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
```

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

The function prototype in line 12 (Fig. 3.11) 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 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 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.

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're 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 the names used do not match those used in the function definition. 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 12–15 of Fig. 3.11. The definitions appear in lines 9–32 and are nearly identical to the member-function definitions in lines 12–35 of Fig. 3.9.

```
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 {
11    setCourseName( name ); // call set function to initialize courseName
12 } // end GradeBook constructor
13
14 // function to set the course name
15 void GradeBook::setCourseName( string name )
16 {
17    courseName = name; // store the course name in the object
18 } // end function setCourseName
19
20 // function to get the course name
21 string GradeBook::getCourseName()
22 {
23    return courseName; // return object's courseName
24 } // end function getCourseName
25
26 // display a welcome message to the GradeBook user
27 void GradeBook::displayMessage()
28 {
29    // call getCourseName to get the courseName
30    cout << "Welcome to the grade book for\n" << getCourseName()
31       << "!" << endl;
32 } // end function displayMessage
```

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

Notice that each member-function name in the function headers (lines 9, 15, 21 and 27) 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 (Fig. 3.11), 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`. These are also called global functions. 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 17 and 23 that access variable `courseName` would cause compilation errors

because courseName is not declared as a local variable in each function—the 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 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

1. the first line of each member function (lines 9, 15, 21 and 27) matches its prototype in the GradeBook.h file—for example, the compiler ensures that getCourseName accepts no parameters and returns a string, and that
2. each member function knows about the class's data members and other member functions—for example, lines 17 and 23 can access variable courseName because it's declared in GradeBook.h as a data member of class GradeBook, and lines 11 and 30 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).

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.

```
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
```

Fig. 3.13 | GradeBook class demonstration after separating its interface from its implementation. (Part 1 of 2.)

```
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. (Part 2 of 2.)

As in Fig. 3.10, line 5 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 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 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 client code that uses the class. 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.]

A class-implementation programmer responsible for creating a reusable `GradeBook` class creates the header file `GradeBook.h` and the 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 class's member-function implementation details, 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 `GradeBook` object code (i.e., the machine-language instructions that represent `GradeBook`'s member functions). The client-code programmer is not given `GradeBook.cpp`, so the client remains unaware of how `GradeBook`'s member functions are implemented.

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 must `#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, 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 and
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. Compilers and IDEs typically invoke the linker for you after compiling your code.

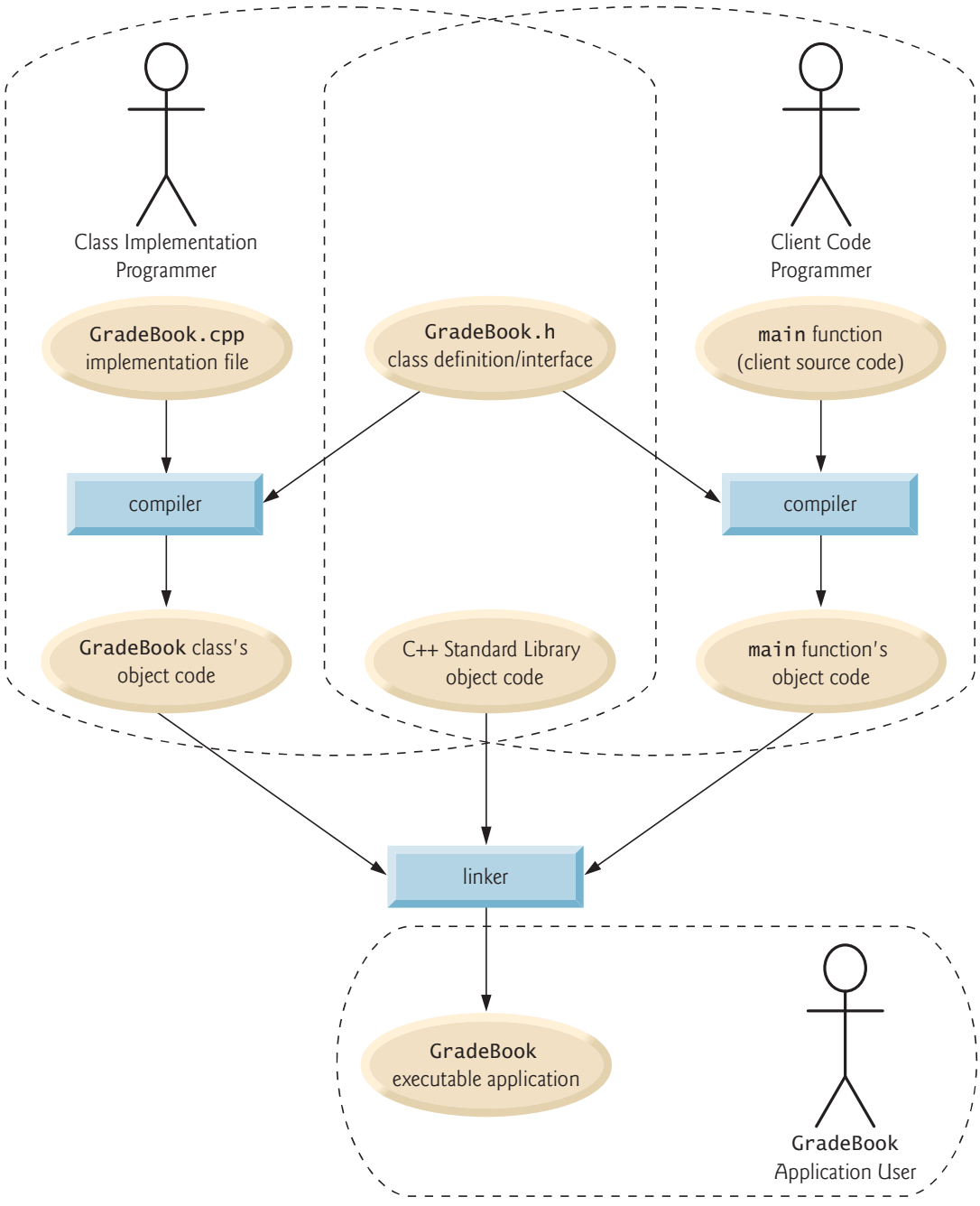

Fig. 3.14 | Compilation and linking process that produces an executable application.

For further information on compiling multiple-source-file programs, see your compiler's documentation. We provide links to various C++ compilers in our C++ Resource Center at `www.deitel.com/cplusplus/`.

3.9 Validating Data with *set* Functions

In Section 3.5, 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 `set-`

CourseName 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. 3.15–3.17 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 interface—is 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.

```
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 namespace std;
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
```

Fig. 3.15 | GradeBook class definition.

Validating the Course Name with GradeBook Member Function setCourseName

The enhancement to class GradeBook is in the definition of setCourseName (Fig. 3.16, lines 16–29). The if statement in lines 18–19 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 19 stores it in data member courseName. Note the expression name.length() in line 18. 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 19 executes.

The if statement in lines 21–28 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

```
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
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.length() <= 25 ) // if name has 25 or fewer characters
19       courseName = name; // store the course name in the object
20
21    if ( name.length() > 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       cout << "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()
33 {
34    return courseName; // return object's courseName
35 } // end function getCourseName
36
37 // display a welcome message to the GradeBook user
38 void GradeBook::displayMessage()
39 {
40    // call getCourseName to get the courseName
41    cout << "Welcome to the grade book for\n" << getCourseName()
42       << "!" << endl;
43 } // end function displayMessage
```

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

`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 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 24 (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 copied—the 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 24 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 24 assigns the substring returned by `substr` to data member `courseName`. In this way, `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 26–27 display a warning message.

The `if` statement in lines 21–28 contains two body statements—one to set the `courseName` to the first 25 characters of parameter `name` and one to print an accompanying message to the user. Both statements should 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'll learn more about placing multiple statements in a control statement's body in Chapter 4.

The statement in lines 26–27 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 26–27 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. 3.15–3.16) featuring validation. Line 12 creates a `GradeBook` object named `gradeBook1`. Recall that the `GradeBook` constructor calls `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 12 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 12). The output in Fig. 3.17 contains the warning message output by lines 26–27 of Fig. 3.16 in member function `setCourseName`. Line 13 creates another `GradeBook` object called `gradeBook2`—the valid course name passed to the constructor is exactly 25 characters.

Lines 16–19 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 22 calls

```
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 myGradeBook's courseName (with a valid-length string)
22    gradeBook1.setCourseName( "CS101 C++ Programming" );
23
24    // display each GradeBook's courseName
25    cout << "\ngradeBook1's course name is: "
26       << gradeBook1.getCourseName()
27       << "\ngradeBook2's course name is: "
28       << gradeBook2.getCourseName() << endl;
29 } // 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
```

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

`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 25–28 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, and so on

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`*—you must provide appropriate validity checking and report the errors.*

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 can test the return value of a *set* function to determine whether the 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. 3.15–3.17 simple at this early point in the book, `setCourseName` in Fig. 3.16 just prints an appropriate message.

3.10 Wrap-Up

In this chapter, you created user-defined classes, and created and used objects of those classes. 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 the object 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 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 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. 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.

Summary

Section 3.2 Classes, Objects, Member Functions and Data Members

- 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.
- 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's used in a program. These attributes are specified as data members in the object's class.

Section 3.3 Defining a Class with a Member Function

- A class definition contains the data members and member functions that define the class's attributes and behaviors, respectively.