3.5 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.

```
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.)

```
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\n" << 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.)

```
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
```

```
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.)

3.5 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.
- *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 (:)*.

3.5 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.

3.5 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

3.5 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.*

3.5 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.

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.

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.

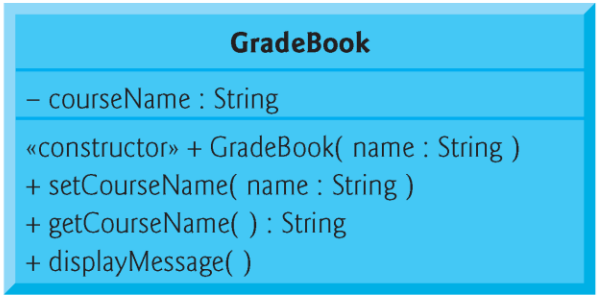

Fig. 3.8 | UML class diagram indicating that class `GradeBook` has a constructor with a `name` parameter of UML type `String`.

3.6 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.
  - As you learned in Chapter 2, function `main` begins the execution of every program, and every program must have exactly one `main` function.

3.6 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

3.6 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 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).