```
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 1 of 2.)

```
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.)

```
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.

3.6 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.

3.6 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.

3.6 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.
- 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.

3.6 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

3.6 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.
  - 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 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>`).

3.6 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.
- 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.

3.7 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).

3.7 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.

3.7 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.

3.7 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.

```
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 }; // end class GradeBook
```

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