Lecture 16:
Introduction to
Classes and Objects

Ioan Raicu
Department of Electrical Engineering & Computer Science
Northwestern University

EECS 211
Fundamentals of Computer Programming II
April 23rd, 2010
• Each class can provide a constructor that can be used to initialize an object of the class when the object is created.

• A constructor is a special member function that must be defined with the same name as the class, so that the compiler can distinguish it from the class’s other member functions.

• An important difference between constructors and other functions is that constructors cannot return values, so they cannot specify a return type (not even void).

• Normally, constructors are declared public.
• C++ requires a constructor call for each object that is created, which helps ensure that each object is initialized before it’s used in a program.

• The constructor call occurs implicitly when the object is created.

• If a class does not explicitly include a constructor, the compiler provides a default constructor—that is, a constructor with no parameters.
3.6 Initializing Objects with Constructors (cont.)

```cpp
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    GradeBook( string name )
15    {
16       setCourseName( name ); // call set function to initialize courseName
17    } // end GradeBook constructor
18```

**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.)
3.6 Initializing Objects with Constructors (cont.)

```cpp
19 // function to set the course name
20 void setCourseName( string name )
21 {
22     courseName = name; // store the course name in the object
23 } // end function setCourseName
24
25 // function to get the course name
26 string getCourseName()
27 {
28     return courseName; // return object's courseName
29 } // end function getCourseName
30
31 // display a welcome message to the GradeBook user
32 void displayMessage()
33 {
34     // call getCourseName to get the courseName
35     cout << "Welcome to the grade book for\n" << getCourseName()
36         << "!" << endl;
37 } // end function displayMessage
38 private:
39     string courseName; // course name for this GradeBook
40 }; // end class GradeBook
```

**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.)
3.6 Initializing Objects with Constructors (cont.)

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

**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.)
• Any constructor that takes no arguments is called a default constructor.

• A class gets a default constructor in one of two ways:
  – 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.
  – 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.
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.
• 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.
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`).
3.7 Placing a Class in a Separate File for Reusability (cont.)

// Fig. 3.9: GradeBook.h
// GradeBook class definition in a separate file from main.
#include <iostream>
#include <string> // class GradeBook uses C++ standard string class
using namespace std;

// GradeBook class definition
class GradeBook
{
public:
    // constructor initializes courseName with string supplied as argument
    GradeBook( string name )
    {
        setCourseName( name ); // call set function to initialize courseName
    } // end GradeBook constructor

    // function to set the course name
    void setCourseName( string name )
    {
        courseName = name; // store the course name in the object
    } // end function setCourseName

Fig. 3.9 | GradeBook class definition in a separate file from main. (Part 1 of 2.)
3.7 Placing a Class in a Separate File for Reusability (cont.)

```c++
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" << 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.)
// Fig. 3.10: fig03_10.cpp
// Including class GradeBook from file GradeBook.h for use in main.
#include <iostream>
#include "GradeBook.h" // include definition of class GradeBook
using namespace std;

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

    // display initial value of courseName for each GradeBook
    cout << "gradeBook1 created for course: " << gradeBook1.getCourseName() << "\ngradeBook2 created for course: " << gradeBook2.getCourseName() << endl;
}

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

• 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.

• 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.
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.

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.
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.
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.
• 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.
• 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).
• 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.
3.8 Separating Interface from Implementation (cont.)

```cpp
// Fig. 3.11: GradeBook.h
// GradeBook class definition. This file presents GradeBook's public
// interface without revealing the implementations of GradeBook's member
// functions, which are defined in GradeBook.cpp.
#include <string> // class GradeBook uses C++ standard string class
using namespace std;

// GradeBook class definition
class GradeBook
{
    public:
        GradeBook( string ); // constructor that initializes courseName
        void setCourseName( string ); // function that sets the course name
        string getCourseName(); // function that gets the course name
        void displayMessage(); // function that displays a welcome message
    private:
        string courseName; // course name for this GradeBook
}; // end class GradeBook
```

**Fig. 3.11** GradeBook class definition containing function prototypes that specify
the interface of the class.
3.8 Separating Interface from Implementation (cont.)

```cpp
// Fig. 3.12: GradeBook.cpp
// GradeBook member-function definitions. This file contains
// implementations of the member functions prototyped in GradeBook.h.
#include <iostream>
#include "GradeBook.h" // include definition of class GradeBook
using namespace std;

// constructor initializes courseName with string supplied as argument
GradeBook::GradeBook( string name )
{
    setCourseName( name ); // call set function to initialize courseName
} // end GradeBook constructor

// function to set the course name
void GradeBook::setCourseName( string name )
{
    courseName = name; // store the course name in the object
} // end function setCourseName

Fig. 3.12 | GradeBook member-function definitions represent the implementation of class GradeBook. (Part 1 of 2.)
```
3.8 Separating Interface from Implementation (cont.)

```cpp
20  // function to get the course name
21  string GradeBook::getCourseName()
22  {
23      return courseName; // return object's courseName
24  } // end function getCourseName

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**. (Part 2 of 2.)
• 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
```cpp
// Fig. 3.13: fig03_13.cpp
// GradeBook class demonstration after separating
// its interface from its implementation.
#include <iostream>
#include "GradeBook.h" // include definition of class GradeBook
using namespace std;

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

    // display initial value of courseName for each GradeBook
    cout << "gradeBook1 created for course: " << gradeBook1.getCourseName() << "\ngradeBook2 created for course: " << gradeBook2.getCourseName() << endl;
} // 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.)
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.
Fig. 3.14 | Compilation and linking process that produces an executable
3.9 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 clas 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.
3.9 Validating Data with set Functions

```cpp
// Fig. 3.15: GradeBook.h
// GradeBook class definition presents the public interface of
// the class. Member-function definitions appear in GradeBook.cpp.
#include <string> // program uses C++ standard string class
using namespace std;

// GradeBook class definition
class GradeBook
{
public:
  GradeBook( string ); // constructor that initializes a GradeBook object
  void setCourseName( string ); // function that sets the course name
  string getCourseName(); // function that gets the course name
  void displayMessage(); // function that displays a welcome message
private:
  string courseName; // course name for this GradeBook
}; // end class GradeBook
```

Fig. 3.15 GradeBook class definition.
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.
3.10 Validating Data with set Functions

```cpp
// Fig. 3.16: GradeBook.cpp
// Implementations of the GradeBook member-function definitions.
// The setCourseName function performs validation.
#include <iostream>
#include "GradeBook.h" // include definition of class GradeBook
using namespace std;

// constructor initializes courseName with string supplied as argument
GradeBook::GradeBook( string name )
{
    setCourseName( name ); // validate and store courseName
} // end GradeBook constructor

// function that sets the course name;
// ensures that the course name has at most 25 characters
void GradeBook::setCourseName( string name )
{
    if ( name.length() <= 25 ) // if name has 25 or fewer characters
        courseName = name; // store the course name in the object
```

**Fig. 3.16** | Member-function definitions for class GradeBook with a set function that validates the length of data member courseName. (Part 1 of 2.)
3.10 Validating Data with set Functions

```cpp
if ( name.length() > 25 ) // if name has more than 25 characters
{
    // set courseName to first 25 characters of parameter name
    courseName = name.substr( 0, 25 ); // start at 0, length of 25

    cout << "Name \"" << name << "\" exceeds maximum length (25).\n" << "Limiting courseName to first 25 characters.\n" << endl;
}
} // end function setCourseName

// function to get the course name
string GradeBook::getCourseName()
{
    return courseName; // return object's courseName
} // end function getCourseName

// display a welcome message to the GradeBook user
void GradeBook::displayMessage()
{
    // call getCourseName to get the courseName
    cout << "Welcome to the grade book for\n" << getCourseName()
     << "!" << endl;
} // 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 2 of 2.)
3.10 Validating Data with \textit{set} Functions

\footnotesize

\begin{verbatim}
// Fig. 3.17: fig03_17.cpp
// Create and manipulate a GradeBook object; illustrate validation.
#include <iostream>
#include "GradeBook.h" // include definition of class GradeBook
using namespace std;

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

    // display each GradeBook's courseName
    cout << "gradeBook1's initial course name is: "
         << gradeBook1.getCourseName()
         << "\ngradeBook2's initial course name is: "
         << gradeBook2.getCourseName() << endl;

    // modify myGradeBook's courseName (with a valid-length string)
    gradeBook1.setCourseName("CS101 C++ Programming");
\end{verbatim}

\textbf{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.)
3.10 Validating Data with set Functions

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