CIT237
Chapter 13: Introduction to Classes – Part 2

October 23, 2019

Reminders

- Quiz 5 will be held at the start of class on
  Wednesday, November 6.
- The material covered on Quiz 5 will be:
  - Lectures of October 21 through October 30.
  - Chapters 13 and 14, and *maybe* some of Chapter 15.
  - Further updates will be provided as the date approaches.
- Programming Project 2:
  - The EXTENDED due date is:
    Monday, October 28
  - If you are having difficulty, talk to me during the Lab portion of today's class.

Recall Previous Lecture

- We introduced the discussion of classes, including
  - Abstract Data Types (ADT)
  - C++ Class Declaration and Definition Syntax
  - Defining an object (an instance of a class)

Example Class Specification

```
class Rectangle
{
private:
     double width;
     double height;

public:
     void setWidth(double);
     void setHeight(double);

     double getWidth() const;
     double getHeight() const;
     double getArea() const;
};
```

Example Class Implementation

```
double Rectangle::getWidth() const
{    return width;  }

double Rectangle::getHeight() const
{    return height;   }

double Rectangle::getArea() const
{    return width * height;  }

void Rectangle::setWidth(double w)
{     width=w;          }

void Rectangle::setHeight(double h)
{     height = h;       }
```

Example Class Invocation

```
int main()
{
    double height1, width1, area1;
    Rectangle r1;
    cout << "Enter the height and width: ";
    cin >> height1;
    r1.setHeight(height1);
    cin >> width1;
    r1.setWidth(width1);
    area1 = r1.getArea();
    cout << "Area of rectangle is "
         << area1 << endl;
    system("pause");
    return 0;
}
```

Individual Member Variable Initialization

- Question: Can we define initial values for member variables?

```
class Rectangle
{
private:
     double width=0; //  Question: Is this OK?
     double height;
     . . .
};
```

- Answer: Yes, if your compiler allows it. (Earlier versions of Visual C++ considered this to be a compiler error.)

Object Initialization

In general, how do you initilize an object?

Use a special function called a “constructor”:

```
class Rectangle
{
private:
    double width;
    double height;

public:
    Rectangle();    // constructor

   . . .
}
```

Constructors

- A member function that is automatically called when an object is created.
- Its purpose is to initialize an object.
- The constructor function name is always the same as the class name.
- Has no return type.

- Example:

```
Rectangle::Rectangle()
{
    width=0;
    height=0;
}
```

Default Constructor

- A “default constructor” is a constructor that can be invoked without an argument list.
  - Either no arguments, or all arguments have “default values”
- If you write a class with no constructor at all, C++ will write a “default constructor” for you:
  - This automatically generated “default constructor” will take no arguments.
  - The automatically generated “default constructor” does not contain any code which we (application programmers) can see.

Simple Object Instantiation

- A simple instantiation of a class (with no arguments) calls the default constructor:

```
Rectangle r;
```

Constructor with Arguments

- To create a constructor that takes arguments:
  - indicate parameter types in prototype:

```
Rectangle(double, double);
```

  - Use parameters in the definition:

```
Rectangle::Rectangle(double w, double len)
{
   width = w;
   length = len;
}
```

Passing Arguments to Constructors

- You can pass arguments to the constructor when you create an object:

```
Rectangle r(10, 5);
```

More About Default Constructors

- If all of a constructor's parameters have default arguments, then it is a default constructor. For example:

```
Rectangle(double = 0, double = 0);
```

- Creating an object and passing no arguments will cause this constructor to execute:

```
Rectangle r;
```

Classes with No Default Constructor

- When all of a class's constructors require arguments, then the class has NO default constructor.

- When this is the case, you must pass the required arguments to the constructor when creating an object.

Destructors

- Member function automatically called when an object is destroyed
- Destructor name is ~*ClassName*, for example:

```
~Rectangle
```

- Has no return type; takes no arguments
- Only one destructor per class, *i.e.*, it cannot be overloaded
- If constructor allocates dynamic memory, destructor should release it
- If a class has no explicitly declared destructor, then it is considered to have a destructor which does nothing.

Constructors, Destructors, and Dynamically Allocated Objects

- When an object is dynamically allocated with the `new` operator, its constructor executes:

```
Rectangle *r = new Rectangle(10, 20);
```

- When the object is destroyed, its destructor executes:

```
delete r;
```

Memory Management

- To prevent “memory leaks” (unreferenced dynamically allocated memory), there must be a one-to-one correspondence between allocation and de-allocation.
- If a constructor allocates memory, then a destructor should de-allocate it.
- If the programmer has a clear idea of what dynamically allocated memory is “owned” by a particular object, then it is clear what memory needs to be de-allocated by the destructor.

Overloading Constructors

- A class can have more than one constructor

- Overloaded constructors in a class must have different parameter lists:

```
Rectangle();
Rectangle(double);
Rectangle(double, double);
```

Only One Default Constructor and One Destructor

- Do not provide more than one default constructor for a class: one that takes no arguments and one that has default arguments for all parameters

```
Square();
Square(int = 0);  // will not compile
```

- Since a destructor takes no arguments, there can only be one destructor for a class

Separating Specification from Implementation

- Place class *declaration* in a header file that serves as the class specification file named `ClassName.h`, for example, `Rectangle.h`
- Place member function *definitions* in a class implementation file named `ClassName.cpp`, for example, `Rectangle.cpp`
- Implementation file should `#include` the class specification file.
- Programs that use the class must `#include` the class specification file, and be compiled and linked with the member function definitions

Specification File: Rectangle.h

```
#ifndef RECTANGLE_H
#define RECTANGLE_H

class Rectangle
{
   private:
      double width;
      double length;
   public:
      void setWidth(double);
      void setLength(double);
      double getWidth() const;
      double getLength() const;
      double getArea() const;
};
#endif
```

Implementation File: Rectangle.cpp

```
#include "Rectangle.h"
#include <iostream>
using namespace std;

void Rectangle::setWidth(double w)
{
    .   .   .
}

void Rectangle::setLength(double len)
{
    .   .   .
}
```

Familiar Preprocessor Directives

- We have seen the use of several preprocessor directives:

```
#include <iostream>
```

(Include library definitions)

```
#define PI 3.14159
```

(Define a "constant" – Note: this is ***not*** the best way to define a constant.)

"Include Guard" Directives

- "Conditional" inclusion of text:

```
#ifndef RECTANGLE_H
#define RECTANGLE_H
    Header File Contents:
        class declaration, plus
        any other #define statements
#endif
```

- This assures that the *Header File Contents* can only be included ***once***.

Lab 13.1: Mortgage Class Specification

The class declaration must be in a file called **`Mortgage.h`**:

```
#ifndef MORTGAGE_H
#define MORTGAGE_H
        //  various #include statements (if needed).
        using namespace std;

        class Mortgage
        {
           private:
                 //  variable declarations
           public:
                 //  function prototypes
        };
#endif
```

Lab 13.1: Mortgage Class Implementation

File name: **`Mortgage.cpp`**:

```
#include "Mortgage.h"
using namespace std;
//   code to implement the member functions.
double Mortgage::getMonthly() const
{
    . . .
}
```