CSI 1430
Week 10 Resource

Colin Burdine

3/21/21

Major Topics:

1. Classes (Part II)
2. Constructors

Keywords: *objects, header file, source file, classes, structs*

Reminder: Group tutoring sessions for this course will be held online every Monday from 7:00pm to 8:00pm CDT. For more information on how to sign up for these sessions, go to: https://www.baylor.edu/tutoring.

1 Notes on Major Topics:

1.1 Classes (Part II)

Last week, we introduced the basic syntax for classes, as well as the role that they play in *Object Oriented Programming* (OOP). Recall that the purpose of OOP is to think of a program or application as a collection of Objects that interact with eachother. These objects not only are able to group primitive data types together, but they also group together a series of *operations* that can be invoked on or by an object. Esentially, we can think of a class as a sort-of blueprint by which we construct *object instances*, which are what we actually work with in our code. We also recall that the *member variables* of an object (the variables declared inside the scope of the class) are particular to each object, meaning they are not (by default) shared. We will go over some exceptions to this rule later on.

Last week, we came up with the following class to represent a Window that a user can manipulate as part of an application's GUI (graphical user interface):

```
class window {

// we declare the associated variables
// in this private to prevent them from
// being accessed or modified incorrectly:
private:
    int width, height;
    bool isMinimized;


// we declare the associated functions to be
// public (typically), so that only the
// functions we allow can be used to modify
// the associated variables:
public:

    // constructor for a window instance:
    window(int w, int h);

    // resizes a window:
    void resizeWindow(int w, int h);

    // minimizes a window:
    void minimize();

    // maximizes a window:
    void maximize();

    // exits from a window:
    void quit();
};
```

Note that there are two keywords in the class body that we haven't discussed yet: `public` and `private`. These keywords are called *access specifiers* and they are useful for denoting which variables and functions can be accessed or called from outside the class body. As you may have guessed, the `private` access specifier denotes the set of functions and variables that can only be accessed from within the class, while the `public` access specifier denotes those that can be accessed from outside. A general rule of thumb in OOP is to keep the variables associated with an object *private*, while making the interface (the set of functions that can be called by/on the object) `public`. This is why we create getters and setters for our member variables instead of making the member variables themselves `public`. This grants us better control of how our data is accessed. By not including a setter, for example, we could make a particular data value *read-only*. Likewise, by not

including a getter, we could make a data value *write-only*.

2 Constructors

When creating and interacting with instances of a class, we may not only want to control what values can be accessed through our *access specifier* keywords; indeed, we might also want to have control over how instances of our class can be created. This is where we can use special functions called *constructors*. Constructors can be defined for both classes and structs in C++, however we will only concern ourselves with classes, since the syntax is identical for both. in our example `window` class, we added the following constructor `window(int w, int h)`, which creates a new window with the given width and height. Constructors are different from regular functions for two reasons:

- They always have the same name as the name of the class.
- They do not have an explicit return type (not even `void`).

For example, the implementation of our window function (in our source file `window.cpp`) might look like:

```
#include "window.h"
...
window::window(int w, int h){
    width = w;
    height = h;
    isMinimized = false;
}
```

Note that we can declare any number of parameters or types in our constructors. In fact there are some special parameter sets that allow for special constructor behavior. The two we will cover this week are *default constructors* and *copy constructors*.

Default constructors are simply constructors declared without any parameters. They are typically used to create objects with meaningful default values. For example, in our `window` class, we could have a default constructor `window()` with the following body in a source file `window.cpp`:

```
#include "window.h"
...

/* default constructor */
window::window(){
    // the window will default to 1024x800 pixels:
    width = 1024;
```

```
    height = 800;
    isMinimized = false;
}
```

We can invoke the default constructor using the calling the default constructor as we would any other constructor. However, the default constructor is special in that if we simply declare an object without initializing it as a default value, C++ will automatically call the default constructor. For example, the following two lines of code are equivalent:

```
// this line explicitly invokes the  default constructor:
window defaultWindowA = window();

// this line implicitly invoked the default constructor:
window defaultWindowB;
```

Another kind of constructor we can create is called a *copy constructor*. This is a special kind of constructor that allows us to construct a class instance as an exact copy of another instance. A copy constructor is recognized by the fact that it only has one parameter, a reference to another instance of the same class. For example, the copy constructor for our window class could be:

```
#include "window.h"
...

/* copy constructor */
window::window(window& other){

// set members variables to be identical to those in other:
    width = other.width;
    height = other.height;
    isMinimized = other.isMinimized;
}
```

We remark briefly that even through the member variables are private in the window class, we can still access the private members of the other window, since we are within the scope of the window class. Like the default constructor, we can invoke this constructor explicitly; however we can also invoke it implicitly using the assignment operator after a new instance is declared. For example, the following are equivalent:

```
// we will assume other is initialized:
window other = ...

// this line explicitly invokes the copy constructor:
window copyWindowA = window(copyWindowA);

// this line implicitly invokes the copy constructor:
```

```
8 window copyWindowB = other;
```

3 Programming Example Problem (Repeat from last week)

To illustrate how we can use classes to solve problems, consider the following example programming problem:

Create both a header file and a source file with a full implementation of the `window` class described above. Use the `window.h` as the header file name and use `window.cpp` as the implementation file name. For the `exit()` function in the window class, you may do nothing or use the C++ function `exit(0)` to kill the program when `exit()` is called (include `<cstdlib>` to use it). As a bonus, provide a constructor, default constructor, and copy constructor function `window(int w, int h)` to allow users to create a new window.

The functions you should expose in this class are:

```
window::window();               // (default constructor for window)
window::window(int w, int h);  // (constructor for window)
window::window(window& other); // (copy constructor for window)

void window::resizeWindow(int w, int h);
void window::minimize();
void window::maximize();
void window::exit();
```

Since this is a repeat of the problem from last week, I will not include an answer key. To check your work, you can reference last week's key for most of the functions. For the different constructors, you can check your work against the implementation given in the notes above.

```h
#ifndef WINDOW_H
#define WINDOW_H

/* window.h
 *
 * This contains the class declaration for the
 * window class.
*/

class window {
private:
    int width, height;
    bool isMinimized;

public:

    // constructor for a window instance:
    window(int w, int h);

    // resizes a window:
    void resizeWindow(int w, int h);

    // minimizes a window:
    void minimize();

    // maximizes a window:
    void maximize();

    // exits from a window:
    void exit();
};

#endif /* WINDOW_H */
```

```cpp
/* window.cpp
 *
 * (This is the implementation of the window class)
*/
#include "window.h"
#include <cstdlib>

// constructor for a window instance:
window::window(int w, int h){
    // initialize the width and height variables:
    width = w;
    height = h;

    // set minimization to false:
    isMinimized = false;
}
```

```
17
18 // resizes a window:
19 void window::resizeWindow(int w, int h){
20     width = w;
21     height = h;
22 }
23
24 // minimizes a window:
25 void window::minimize(){
26     isMinimized = true;
27 }
28
29 // maximizes a window:
30 void window::maximize(){
31     isMinimized = false;
32 }
33
34 // exits from a window:
35 void window::exit(){
36     // perform exiting routine:
37     std::exit(0);
38 }
```