CSI 1430
Week 9 Resource

Colin Burdine

3/15/21

Major Topics:

1. Object-Oriented Programming
2. Structs
3. Classes

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.

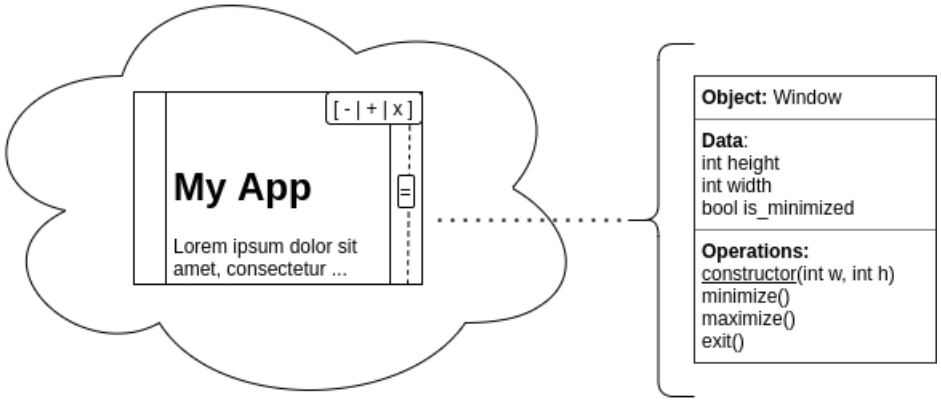
1 Notes on Major Topics:

1.1 Object-Oriented Programming

So far in the semester, we have been writing our code *linearly*, meaning we are given some problem to solve or some task to accomplish and we have produced programs that follow a long sequence of instructions to perform some task. If we were able to break down a problem into sub-problems we might have also written some user-defined functions to help with organizing our code. However as our projects are getting larger and larger, we are starting to wrestle with the problem of *complexity*. At a certain point, you may start to see that writing a set of functions that solve a problem linearly, from start-to-finish, may be too difficult to write, and even more complex to debug. So what is the solution? How

can we handle the added complexity that comes with large programs? One answer is to use *Object-Oriented Programming*.

So, what is *Object-Oriented Programming* (OOP)? OOP is design philosophy for programming, in which we group together both data and operations into *Objects*. C++ is an example of an *Object-Oriented Programming Language*, meaning that it supports structures and keywords for defining Objects and the actions that they perform. Consider, for example, a desktop application that has a window-based user interface (as most modern systems do). We can do several things to a window, such as minimize, maximize, or exit a window. However, a window may also store some information about itself, such as the window size. An illustration of this is given below:



Notice that we have grouped together both the attributes of the window and the operations that are performable on it. This allows us to easily create multiple windows for our application, each with its own associated size and state (i.e. minimized/maximized). In C++ there are two mechanisms by which we can group together data and operations: using *structs* and *classes*.

1.2 Structs

Using a struct (short for "Structure") is one way that C++ allows us to group together data and operations. Typically, structs are used for the creation of simple objects. Suppose, for example, we want to plot a series of data points on a 2D-plane, where each data point has an associated x and y coordinate. We can declare a point structure as follows and create a point instance as follows:

```
1 // This is the declaration of a struct:
2 struct point_t {
3 double x, y
4 };
```

```
// this is an instance of the struct above:
point_t myPoint = point_t{ 1.0, -1.0 };
```

Note that when we initialize instances of structs, the order in which we specify the arguments must correspond to the order in which the values are declared in the struct. We can also omit the curly braces entirely when creating a struct instance. In this case the values will be allocated, but have no predictable value until they are set. We can access any values in a struct using the dot-operator (".") followed by the name of the field in the struct that we are accessing. For example, we could do the following:

```
#include <iostream>
using namespace std;
...

point_t myPoint = point_t { -1.2, 3.4 };

cout << "Point is: (" << myPoint.x << ", " << myPoint.y << ")" << endl;
```

The output of the code above would be:

```
Point is: (-1.2, 3.4)
```

1.3 Classes

In C++ classes are similar to structs, though they are much more powerful, because they allow for *data hiding*, which means that public and private access specifiers can be used to denote which values (or functions) can be accessed from outside the class. Even though we did not show it in the previous example, both structs and classes can have functions associated with them; however, we rarely see functions associated with structs (the only exception to this rule are *constructors*, which we will cover next week). To define a class to represent an application window, as in our illustration above, we can use the following syntax:

```
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 exit();
};
```

2 Example Programming Problem

To illustrate how we can use classes to solve problems, consider the following 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 built-int C++ function `exit(0)` to kill the program when the window's `exit()` is called (include `<cstdlib>` to use it). As a bonus, provide a 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(int w, int h); // (constructor for window)

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

My solution to this problem is on the next page. You can look at my answer if you are stuck, but I would strongly encourage you to sketch out a design first, and then proceed to write your solution in C++. This exercise is intended to check your understanding of the concepts from this week and last week.

```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 }
```