CSI 1430
Week 15 Resource

Colin Burdine

4/25/21

Major Topics:

1. Course Content Review

Keywords: *final exam review*

Reminder: The last group tutoring session for this course will be this coming Monday (4/26/21) from 7:00pm to 8:00pm CDT. During this session, we will be reviewing for the final exam and answering questions. For more information on how to sign up for this session, go to: https://www.baylor.edu/tutoring.

1 Notes on Major Topics:

1.1 Course Content Review

Because this is the last resource for this course, I have provided a course content outline that indexes some of the major topics in this course. I have also provided references to each of the previous weekly resources below. Hopefully this will be of use when making a study plan for the final exam. While this outline is comprehensive, it is not exhaustive. In addition to reviewing the content in this resource, I would highly encourage you to review your past programming assignments, as they are great examples of the material outlined above.

Most Important Topics:

These are the topics you will want to be sure you understand fully:

1. Basic Input/Output (Week 3)

   - The `cout` and `cin` objects
   - Using the stream operators `<<` and `>>`
   - The `<iostream>` library

   The `cout` and `cin` objects are input and output stream objects supplied by C++. To use them, be sure to use `#include <iostream>` and use the `std` namespace. We can print values to `cout` using the stream insertion (`<<`) operator, which displays them as text in the console. We can also read values from `cin` using the stream extraction (`>>`) operator, which reads a value entered by the user in the console.

2. Variable Types and Arithmetic Operations (Week 3)

   - The types `int`, `double`, `char`, `bool`, `string` (and others).

   The types `int`, `double`, `char`, `bool`, etc. are C++ primitives, meaning they are built into the language. The `string` type behaves like a primitive (it is included in the `std` namespace), but it is actually an object. In order to use the `string` type, sometimes you may need to add `#include <string>`.

   - The `+`, `-`, `*`, `%`, and `/` operators (how do they behave with different types?)

   The `+`, `-`, and `*` operators perform addition, subtraction, and multiplication for all numeric types. The division operator `/` performs standard division on floating-point types like `double` and `float`; however, with integers it truncates the remainder, rounding down to the closest integer with absolute value less than or equal to the quotient. The modulus (`%`) operator returns the integer remainder of division. For example, `7 % 4` would evaluate to `3`.

3. Conditional Branching (Week 3-4)

   - Behavior of `if(...){...}`
   - Behavior of `if(...){...}else{...}`
   - Behavior of `if(...){...}else if(...){...}else{...}`

   In most programming languages, including C++, sequences of `if`/`else` statements are all mutually exclusive, meaning only one block in brackets gets evaluated. The expressions in parentheses following each `if` or `else if` must evaluate

to a boolean value. Each of the expressions are tried in order from top to bottom, until one is found to be true. If all of these expressions evaluate to `false` and there is a final `else` block, that final `else` block will be executed.

4. Boolean values and Logical Operations (Week 4-5)

   - Behavior of `&&`, `||`, and `!` operators.

     The `&&`, `||`, and `!` operators represent a logical AND, OR, and NOT respectively. They most commonly are used to operate on boolean values, but they can also operate on numeric types (recall that a numeric value of 0 is interpreted as `false`, while anything else is interpreted as `true`). The order of evaluation for these boolean operators is first NOT, then AND, then OR, though it is good practice to use parentheses to make boolean expressions unambiguous.

   - DeMorgan's Law: `!(x && y) == (!x) || (!y)`

     DeMorgan's Law (the identity above) is useful for simplifying boolean expressions. From the identity above, it can also be shown that `!(x || y) == (!x) && (!y)`.

5. Variable Scope (Week 6)

   - How do we identify the scope of a variable?

     As a general rule of thumb, we can think of the scope of a variable as being limited to the pair of braces ("{ ... }") within which the variable is declared. Index variables declared in `for` loop are limited to the duration of the for loop.

6. Loops (Week 6)

   - Behavior of `while(...){...}` loops
   - Behavior of `do{...}while(...)` loops
   - Behavior of `for( ; ; ){...}` loops

     Loops are useful for executing code until a boolean expression (the loop condition) is met. The `for` and `while` loops always check the loop condition before executing, while a `do-while` loop always executes the loop body once before checking the loop condition. `for` loops are commonly used for iterating over

data structures like arrays and vectors.

7. Arrays and Vectors (Week 7-8)

   - Array declaration syntax (e.g: `int[3] myArr = { 1, 2, 3 };`)
   - Vector declaration syntax (e.g: `vector<int> myVec = vector<int>(3,42);`)

     We note that arrays are built in to C++, while vectors are part of the C++ Standard Template Library (STL). To use vectors, we must add `#include <vector>`.
   - How are vectors different from arrays (aside from using `#include<vector>`)?
   - When should we use vectors instead of arrays (and vice versa)?

     Vectors are different from arrays because (a) they can be resized, and (b) they also keep track of their own size. Arrays are not resizable, and must be created with a fixed size. Vectors are great when the number of items to be added to the array is unknown; however, arrays are generally faster and take up less memory.

8. Functions (Week 8-9)

   - How to write user-defined functions

     A used-defined function is written in two parts. First there is the prototype, which contains the return type, function name, and parameters (e.g. `double doSomething(int x);`). Second, there is the implementation, which contains the actual code that is executed (e.g. `double doSomething(int x){...}`).

   - Passing variables by reference (using `&` in the parameters)
   - Scope of variable inside of functions

     The default behavior of passing parameters in functions is by making *copies* of the parameters. The scope of these copies is limited to the function itself, so modifying these copied parameters will not change the original values. However, if a parameter is passed by reference (e.g. `func(int& x)`), then the value is not copied, and *can* be modified within the function.

9. Code Organization (Week 9)

   - Header files (`.h`) versus source files (`.cpp`).

- When do we include header files? Can we include source files?

  Never under any circumstances include a `.cpp` file. Just don't do it.

10. Object-Oriented Programming- OOP (week 10)

    - What is the difference between a class and an object?
    - What do the `public`, `private`, and `protected` access specifiers do?

    A *class* can be thought of as a blueprint for an objects. In other words, we create instances of classes, and these instances are what we manipulate when our program executes. For example, when we write `string s = "hello"`, the value `"hello"` is an instance of the `string` class. Within a class we can have `public`, `private`, and `protected` member variables and member functions. `public` members are accessible from outside the class (using the `.` operator). `private` members cannot be accessed from outside the class. `protected` members are essentially like private members but are accessible from withing inheriting classes.

11. Classes and Structs (Week 10-11)

    - Creation of objects (e.g. `myObject instance = myObject(...)`)
    - Creation of structs (e.g. `myStruct instance = {...}`)

12. Constructors and Class functions (Week 11-12)

    - Constructors (in both classes and structs)
    - Copy Constructors and Default Constructors

    Constructors are how we create instances of classes. They are special functions because they have the same name as a class and have no return type (the return type is implicitly an instance of the class). Default constructors have no parameters and assign meaningful default values to the member variables of an instance. Regular constructors set the member variables according to the values passed as parameters.

13. Streams (Week 11)

    - Standard I/O (`cin` and `cout`) with `<iostream>`
    - File I/O (`ofstream`, `ifstream`, and `fstream`) with `<fstream>`

- String I/O (`ostringstream`, `istringstream`, and `stringstream`) with `<sstream>`

  Streams are useful for reading and writing formatted data. To write data to an output stream, we use the stream insertion (`<<`) operator. To read data from an input stream we use the stream extraction (`>>`) operator. For more information on these streams, take a look at the resource from week 11.

14. Searching Algorithms (Week 12)

    - Linear search
    - Binary search (For binary search to work, what must be true about a list?)
    - If we are searching a list of size N for a value, how many elements must we check at most for each algorithm?

      There are two primary methods of searching, linear searching and binary searching. Linear searching iterates through a (possibly unsorted) array or vector. It works by checking one element at a time. Binary search uses bisection to quickly find a value in an array or vector of sorted values. Binary search is much faster than linear search, but requires the values to be sorted.

15. Sorting Algorithms (Week 12)

    - Selection Sort, Insertion Sort, and Bubble Sort

      I cannot give a brief summary of these sorting algorithms in this outline, as these algorithms are quite involved. I would encourage you to look at the resource for week 12 and try to write code for these algorithms on your own. The best way to know that you understand an algorithm is to code it up for yourself!

Supplemental Topics:

These are the topics that are good to know and be familiar, but tend to be more peripheral than the other course material:

1. STL Containers (Week 13)
2. Operations on Objects (Week 13)
3. Pointers (Week 14)