Lecture 04 Notes

Derek Harter

CSci 515 Spring 2014 *<2015-02-06 Fri>*

1 First Session (11 - 11:40)

1.1 Functional Programming

- 6.1 To promote software reusability, every function should be limited to performing a single well define task, and the name of the function should express the task effectively.
- 6.2 If you cannot choose a concise name that expresses a function's task, your function might be attempting to perform too many diverse tasks. It's usually best to break such a function into several smaller functions.

Functions serve several purposes from the point of view of the computer science and the application developer:

1. Functions encapsulate an algorithm or procedure.
2. Functions can be black boxes that help reduce complexity. You can build more complex algorithms on top of simple functions, without worrying about the details of how the functions you use are implemented.
3. Functions allow for reuse of code, and help to avoid code repetition.

For example, it can be very complex to write an algorithm to compute mathematical functions, such as the sin() trigonometric function, on a digital computer. However, there are standard libraries of math functions that you can use that implement the sin() function for you. When you use a library that someone else has written, you are using the functions as black boxes. You know what the function should take as input, what it does, and what its result should be. But you do not need to concern yourself with the messy details of how it achieves its result. Likewise, using standard libraries are examples of a type of reuse. The functions are written one time by experts, but reused over and over again in many programs.

As a programmer you can get the same benefits by writing functions yourself. You can write a function once, and reuse it in many locations in your code. This is an example of a good programming practice known as Don't Repeat Yourself (the DRY principle). You should (almost) never have code repeated in 2 or more places. Once you have the urge to repeat a piece of code, you have probably identified a small procedure or task that should be encapsulated inside of a function, that can be called when needed.

Likewise user defined functions help manage complexity when writing complex algorithms. There is a design strategy known as functional decomposition. Basically, when approaching a problem in this manner, you try and break down the ultimate task into smaller sub-tasks. If those sub-tasks are complex, you further break them down into sub-sub-tasks. You repeat this decomposition until you have defined problems that are simple enough to understand and code directly as small functions. Then you solve the larger problems by stringing together and reusing your functions.

It takes skill and practice to learn how best to do such decomposition, but it is fundamental to learning how to solve real problems that are often complex and messy. Functional decomposition allows you to encapsulate the messiness of the details in black boxes, as functions, then forget about that messiness to solve larger issues, on up till you address your ultimate problem.

Being able to break down a problem into good sub-problems, that can be easily solved individually and then combined into larger solutions is an extremely useful skill. One of the benefits of learning math and computer science, is that you become trained in thinking about problems using these methods and tools. For example, being able to write functions, both mathematical and for computer programs, involves precisely defining the inputs that the function works on, and the outputs that it produces. And, in both cases, we need to formally define the operations that occur, so that the function always deterministically produces the same correct results for the given inputs. This is one reason why thinking about and carefully documenting your input parameters and return results for a function are very important.

1.2 Standard Libraries of Functions

- The math library

- Need to include the header to use a standard library #include <cmath>
- Headers for libraries of standard functions basically define the function signature. We will go into more details about functions signatures, also called function prototypes, in a bit.
- Use some of these functions.
- Show Fig 6.7 (pg 219), list of some of the standard libraries in C/C++.

1.3 Defining a Function

- Function name (choose meaningful name)
- Function Input (the parameters to the function)
- Function Output (the return value)
- Always include a function header documentation. Document purpose, parameters and return value.
- Implement pow(), fabs(), maybe ceil() and floor()

2 Second Session (11:45 - 12:30)

2.1 Function prototypes

- The signature of a function is its prototpye.
- To use a function, all you need is the signature.
- This is important conceputally. The signature represents the "black box" idea of functions. The details of the implementation are not so important, at least to those who want to use the function to solve larger problems. They are only concerned about what it does, how you use it, and that it performs it stated task correctly.
- The actual implementation can be somewhere else.
- In fact, put signatures in a header file, implementations in a separate source file, and (re)use.
- This is the basic method used for the standard libraries of functions.

2.2 Variable and Argument Coercion

- C promotes certain numeric values in mixed-type expressions. Type is promoted to the "highest" type in the expression.
- **BEWARE** of the so called integer division error.
- Passing a parameter to a function can also be automatically coerced in this way.

3 Third Session (12:40 - 1:40)

3.1 Scope Rules

- Variables declared in functions are local to the function.
- Variables (and constants) declared outside of functions are global.
- You probably should NOT be using global variables. Global variables allow state information to leak between functions.
- Global constants, however, are often a good idea and useful.
- Understanding the scope of variables is important. Limiting scope of variables is important.

3.2 Passing Parameters by Reference

- Variables are passed to functions by value by default.
- This means, the value is copied, and if you change value in the function, the value is not changed in the caller.
- The only way to get a value back to the caller, is to return it as the return result from the function.
- However, sometimes we need to return more than 1 value, or sometimes for efficiency reasons (e.g. need to return 1 million values, we probably don't really want to copy them all back).
- In this case, we can pass in values by reference.
- A reference parameter, if changed in the function, will be changed for the caller who provided it. In effect, a reference parameter is not a copy, but it IS the actual variable from the caller, so changes to it will be accessible to the caller when the function returns.

3.3 Function Call Stack

- Example, functions A(), B() and C()
- Recursive
- Some advice/examples on using the Visual Studio debugger.