CS161 Week3 Project Plan Reflection

Ziwei Wu
Student Number: 933296824

July 10, 2017

Understanding. Understanding the problem is the most important part of solving a problem in my opinion. As many have said, programming is 80% thinking and 20% writing the code. When I was given the problem, I try to understand it by analyzing its:

- Purpose: What is motivation behind the problem? Why it needs to be solved?
- Assumptions: What are the assumptions in the problem?
- Constraints: What are the constraints to be taken into consideration?
- Data: What are the expected inputs and outputs? Types of data?

For example, for project 3.a the motive of the program is to ask the user to input some integers, and determine what is the smallest and largest integers among them. The assumptions are:

1. User would enter an integer >= 1 when being asking how many integers he/she would like to enter.
2. User would enter only integers as the numbers to examine.

Since we are aware of these assumptions, we are less worried about user inputing data types like string or float, and so we can narrow down our test cases quite a bit. In terms of constraints, we should be aware of the constraints of our programming language. For example, the max value for int type in C++ is 2147483647, if we use int data type in our program to store input, we should be aware that the program might not work if the user input a value > 2147483647.

Finally, the data is important. A program is like a data manipulating machine. It takes some data, process it and output the data. For 3.a, the input data type is integer, and user can decide how many integers can be used as input. The input data is processed to determine the min and max, then min and max are displayed to the screen as the output. Being consicous of expected inputs and outputs before coding phase will allow us to program much more efficiently.

Testing plan. Testing plan is another important stage before the actual coding. Good testing plan allows us to think deeper about the problem, and to think from a user's perspective. This is crucial because as programmers, we tend to assume that users would know how to use the program and would enter the proper inputs. This is often not the case. For a testing plan, We need to consider:

- Boundary cases: Have we consider both positive and negative inputs?

- Unexpected cases: What if the user enter a string instead of an integer? What if the user did not make a valid selection in the program menu?
- Extreme cases: What if user entered a super large value as input?

For example, in project 3.b, our program is designed to read a text file which contains rows of integers, and output the sum of integers to ”sum.txt”. The input is the text file that a user might provide. Just for the input, there are a lot of things that can occur:

- The file may not contain the integers in a readable format, e.g. in a large list instead of row of numbers.
- The file may not be readable.
- The file may contains extreme values such as a very large number and may cause the sum > 2147483647.

Fortunately, $ifstream$ header provides some ability to handle these cases, for such it would notify the user when the file is not accessable. So it did not affect my project very much. Regardless, I still learned a lot from testing plan. By coming up with a testing plan, we can build more robust program and reduce the chance for bugs.

Design. The design of the programs was drafted by writing pseudocode. I find this approach worked very well because pseudocode is closer our natural language, so it's more easily understood. Another advantage is that it's context free of any specific programming languages. So if I want to implement the program in another language such as Python or Java, I can always have the pseudocode as reference.

A modification that I had to make was for project 3a. After I made my initial submit. I realized my pseudocode only asked the user to enter four integers instead of letting the user choose how many he/she wants to enter. This was because I did not read the problem carefully, and took what appeared in the output demo(only four integers in the demo). The lesson learned here is always read the problem very very carefully. It may take several reads and some thinking to truly understand what is the problem presented.

Implemntation. However writing pseudocode and writing actually code is not the same thing. In project 3.c, I had to implement a number guessing game. The user would choose a number, and ask player to make a guess for the number. Writing pseudocode was fairly easy, but when I write the code I had a problem with looping.

The two cases, “if” statement (player guessed too high), and “else if” statement (player guessed too low) both contained a “$cin >> num$”, where player had to make another guess. The new guess needed to be again evaluated by while loop ($guess! = answer$). I had to find a way for the nested “if” and “else if” to jump back to the start of “while” loop. After some additonal reading from the textbook, I finally found the solution is to use - “continue”.

Therefore, I think while it is important to come up with a sound design before writing the code. It is equally important that we are ready to deal with novel problems and challenges. Sources such as textbook, stack overflow, tutorials videos and blogs can be very handy in this situation.

Improvement. From the experience of problem solving this project, the most important take home message for me is the importance of understanding the problem. For my future works, I need to make sure I understand the problem before writing any actual codes. Writing testing plan and pseudocode can help solidify my understanding of the problem, and also allows me to think about the design and usability of the program.

In terms of design, one of the themes emerged from earlier weeks to this week is that the problems are becoming more complex and involves the use of loop and good use of variables. I learned that the key to solve complex problems is to break the problem down into smaller sub-problems. For example, for project 3.a, the problem can be broken into two pieces:

1. Get the number of integers that the user wants to enter.

2. Finding the largest and the smallest numbers among the integers entered.

Then, by solving each sub-problems and combining them, the more complex problem is easily solved. Finally, in terms of implementation, I learned that it is a good practice to code a little bit and run some tests to see the program is running correctly. This approach allows us to spot any bugs and any unintended results early in the development, so it is easier to fix. I will definitely keep these tips and ideas in mind for my future work.