CSCI 201 – Computer Science 1

Homework Assignment 3

Objectives. The objectives of this assignment are:
1. Learn to design a `do while` loop and implement it in C++.
2. Learn to create functions in C++.
3. Learn how functions can be employed to encapsulate a part of the process, and thereby simplify the code in the main program.

In this homework, we modify and extend the program completed in Lab 3 to incorporate some additional features.

Part I. Due date: Monday Jan 29.
Using a do-while loop. Recall that in a post-test loop, the loop body must execute once before the exit condition is checked. Therefore, to use a post-test loop, *we have to change the way we implement the strategy.* This can be done as follows:

```
initialize total as zero;
initialize operator as '+';
read num;
process num (add or subtract from total, depending on operator);
read next operator;
If (operator is '=') exit and print total.
read num;
...
```

Question 1: Find the repetition in the above sequence and construct a flowchart with a post-test loop (the exit condition should be checked at the end of the loop) using Raptor.
Question 2: Translate this into C++ code, compile and test in a script session.
Question 3 (reflection): Compare the pretest and post-test versions. In this case, which one feels more natural and why? What are the pros and cons of using one over the other?

What to submit: Raptor file for Q 1, script file, .cpp file and answer to Q 3 in the `Hwork3Part1` folder within your CourseFiles folder.

Part II. Due date: Thursday, February 1.

Checking for bad input. There are several kinds of errors that can occur when the program is being used, and the mechanism to handle each one is different. In this assignment we look at two

kinds of errors:
1. Input errors when entering the operator (character input).
2. Input errors when entering the operand (number input).

Dealing with bad characters. Consider a situation where the user inputs some character other

than + or - for an operator. Since the input will always be an `ASCII` character, the program can read it into a variable of type `char` and check it with an `if` statement. We would like that the program should continue operating properly in the event of bad input, and allow the user to correct the mistake as often as needed. That is accomplished by printing an error message and asking the user to give the input again. A sample run looks something like this:

```
Welcome to your friendly neighborhood accumulator! Please input
your expression, one token at a time, starting with an operand and
type in  ‘‘=’’  when completed.
3.0
*
Bad character. Please input +, - or =.
)
Bad character. Please input +, - or =.
+
2.0
-
-5.0
=
10.0
Thank-you for using your friendly neighborhood accumulator!
```

Question 4: Write a `C++` function with the following header
`char getOperator()`
that performs just the task of reading an operator, allowing the user to make any number of mistakes. Create a test program for this function. In `CourseInfo/Hwork3/` in CourseFiles, you will find the file `getCodeTest.cpp` that gives you a example of how such a function can be written and tested separately. Create a script file showing the source code, compilation and testing.

Question 5: Copy the above function into either one of your accumulator programs (with the `while` loop or the `do-while` loop) and replace the `cin` statement(s) that reads the operator with a call to this function. In `CourseInfo/Hwork3/` in CourseFiles, you will find the file `colorCounter.cpp` that gives you a example of how this is done. Create a script file showing the source code, compilation and testing.

Dealing with bad numbers. In the previous error, even when a bad operator is typed in, we can still read it because it is a character. A more difficult situation arises when the user types in bad input when an operand was expected. Here, the program is expecting a `C++ float constant` but finds something else, say, an alphabet. A sample run looks like this:

```
Welcome to your friendly neighborhood accumulator! Please input
your expression, one token at a time, starting with an operand and
type in  ‘‘=’’  when completed.
3.0
*
Bad character. Please input +, - or =.
```

```
)
Bad character. Please input +, - or =.
+
-
Need a float value; please try again.
a
Need a float value; please try again.
2.0
-
-5.0
=
10.0
Thank-you for using your friendly neighborhood accumulator!
```

Using the cin object to catch bad input. When `cin` finds unexpected input, i.e., an alphabet instead of a real number, it does the following:

- The program continues as if the read statement was not executed, i.e., the program will continue to execute with whatever value was stored in the variable.
- The special boolean variable, also called `cin`, becomes false.

Our strategy therefore is to check boolean variable `cin` after the read operation, to detect bad input. After detecting the bad input the `clear` operation is invoked on `cin` to fix the input stream and the `ignore` operation is used to flush out the input pipeline. See the file `UsingClearCin.cpp` in `CourseInfo/Hwork3/` in CourseFiles for an example. In that program, the command
`cin.ignore(200, '\n');`
tells the system to ignore the first 200 characters or the end-of-line(denoted by `'\n'`), whichever comes first. (This is usually enough, but we can still get some strange output; try, for instance, the input "25a" at the first prompt when you run `UsingClearCin.cpp`. We cannot fix everything!)

Question 6: Write a `C++` function with the following header
`float getOperand()`
that performs just the task of reading an operand, and allowing the user to make any number of mistakes. This function will return a float value.Create a test program for this function. Create a script file showing the source code, compilation and testing.
Question 7: Copy the above function into your program for Question 5 of this homework, and replace the `cin` statement with a call to this function. Create a script file showing the source code, compilation and testing.
What to submit: Script files for Questions 4, 5, 6 and 7 and the final .cpp file in the `Hwork3Part2` folder, inside your CourseFiles folder.