EE 355 Lab 3 - Algorithms & Control Structures

1 Introduction
In this lab you will gain experience writing C/C++ programs that utilize loops and conditional structures.

This assignment should be performed INDIVIDUALLY. This is a peer evaluated lab submitted on Vocarium. Your grade will be determined by 2 peer evaluations. Please see the guidelines for peer evaluated labs provided by the instructor.

2 What you will learn
After completing this lab you should be able:
- Find repeated structure/actions in an algorithm and use loops to represent them
- Use for and while loops to control iterations
- Use conditional statements ('if' statements)
- Use the 'gdb' debugger to step through code and

3 Background Information and Notes – GNU Debugger (gdb)

3.1 Starting up GDB
To compile a program with debugger symbols, use the -g flag when you compile. For programs with a Makefile, we'll usually include it for you. Once that is done you can debug the program. The command to run gdb is
gdb EXECUTABLE_NAME
Here you can run the executable using by entering run or r. If you need to add command line arguments, you can add them when you run the executable. For example: run ARG_1 ARG_2 ...

3.2 Breakpoints & Printing Info
If your program has a segmentation fault, memory error, or other program-terminating problem, gdb will halt and let you know that you have hit an error. It is useful to stop the execution of your program at an arbitrary point. This can be accomplished by setting breakpoints. To set a breakpoint, use the break command. The break requires a line number to be specified, and if there are multiple source files, you must specify that too. For example:
- break LINE_NUMBER
Once you hit a breakpoint, you can use the `print` and `backtrace` commands. When the program stops, there are a couple commands that might be helpful.

- `print VARIABLE_NAME` will print the value of a given variable name. This can be used to see what value is incorrect and is usually a good point to start looking for bugs.
- `backtrace` or `bt` will print the stack trace; this will show you the functions that have been called up to the point of the crash.

To proceed with program execution, there are two commands you can use.

- `continue` or `c` to run the program until it encounters the next breakpoint or the end of execution.
- `step` or `s` to step through the program line by line.
- `next` or `n` to step through the program, but skip over function calls (i.e. treat the function call as one line of code rather than stepping through each subsequent line of the function call)

To delete a breakpoint: `delete BREAKPOINT_NUMBER`

To list all breakpoints: `info breakpoints`

To quit gdb: `quit`

### 3.3 Summary

`gdb` is a powerful tool, and we've only outlined the basics. A quick Google search can really help if you want to find some more commands to play with.

### 4 Procedure

#### 4.1 Prime Time

Write a program (must be named `prime23.cpp`) to determine if a natural number (greater than 1) has only 2 and/or 3 as prime factors (but no other prime factors) and how many of each factor (2 and 3) it does have. Write your program from scratch (you can reference other examples to get started with the basic structure of a program) and name it `prime23.cpp`. The program should meet the following requirements:

a. Prompt (print a message to) the user to enter a natural number. [i.e. use `cout`]. The prompt should read `Enter a positive integer:`

b. Receive the integer input from the user. [i.e. use `cin`]

c. Implement your algorithm (using `while` loops and `if` statements).
d. Print either “Yes” on one line and a count of 2 factors and a count of 3 factors (i.e. an input of 24 would print: Twos=3, Threes=1) on the next or just “No” if the number has factors other than 2 or 3.

**Example 1: Input is 30**

| No |

**Example 2: Input is 18**

| Yes |
| Twos=1 Threes=2 |

**Example 3: Input is 8**

| Yes |
| Twos=3 Threes=0 |

**Example 4: Input is 9**

| Yes |
| Twos=0 Threes=2 |

**Example 5: Input is 10**

| No |

Your output messages must be **EXACTLY** in the same format as shown above to get credit. Take care you create such an output.

Compile and test your program. Whether your program works or not, we want you to practice using a debugger like gdb. To practice do the following:

- Start GDB
- Set a breakpoint at whatever line you wrote your first 'while' loop in your code and run the program to that breakpoint. Then type 'run'.
- Step through one iteration (line by line) of the code in the while loop for a few steps
- Print out the current value of some variable while stepping through your code (likely you should print the current value of the number your finding the factors of).

In a file: “gdbanswers.txt” copy down the commands you would use to accomplish each of the bullet points above and submit it with your code files.

### 4.2 ASCII Art

The final product of this assignment will be built up in 3 parts. Name your file 'tri.cpp'.

[Step 1] A right triangle could be drawn using the ‘*’ ASCII character and the cout function by using two for loops (one nested inside the other with the outer loop counting lines and the inner loop counting characters per line). Write a program to print out the *’s to form an isosceles right triangle 31 rows high (i.e. 1 * on row 0, 2 *’s on row 1, etc.).

You should get a triangle similar to the one shown below. **Hint:** You do not have to put a
'\n' or endl character in every string that you print via cout but can print a newline only when you have printed "enough" asterisks

*  
**  
***  
Y axis ****  
******  
*******  
*********  
...  
X axis

[Step 2] Except for distortions in the proportion of a characters height/width (i.e. the font that Linux uses), the triangle above should be an isosceles triangle (i.e. 45 degrees for each of the non-right angles). We could generalize and pick an arbitrary number of degrees for the angle, Θ, shown below. Assume $15 \leq \Theta \leq 75$ and that the **height of the triangle will be 31 text lines running from a y-axis value of 0 to 30** (inclusive).

[Step 2] Modify your program from step 1 to now **query the user for a value of Θ** between 15 and 75 (you can assume they will always comply and not give you a bad value) then print a triangle given a particular value of Θ. To solve this problem, try to derive a mathematical expression for the length of the x-axis (i.e. # of *'s) as a function of Θ and the y-coordinates. Use the C++ constant $\pi$ in place of π. Then iterate through every line (y-coordinate) and calculate the appropriate x-coordinate (round down to get an integral value using the floor() function in math.h / cmath, if desired). The x-coordinate you compute will directly determine how many *'s are printed. Your program should prompt the user for the value of theta they would like to use.

[Step 3] Modify the program in step 2 so that if the x-coordinate (the number of *'s you print on a line) for any line falls within the range $20 \leq x \leq 30$ then only print a line with exactly 20 *'s rather than the calculated value of x. This will make a sharp vertical cliff in the triangle for some number of lines (maybe none if theta is small) and then resume along the original diagonal line. Compile and test your program.

**Note: DO NOT print SPACES between the *'s.**
Enter the angle \( \theta \) in degrees:

\[
\begin{align*}
&45 \\
&* \\
&** \\
&*** \\
&**** \\
&***** \\
&****** \\
&******* \\
&******** \\
&********* \\
&********** \\
&*********** \\
&************ \\
&************* \\
&************** \\
&*************** \\
&**************** \\
&***************** \\
&****************** \\
&******************** \\
&******************* \\
&************************ \\
&******************************** \\
&******************************* \\
\end{align*}
\]

45-degree Sample Output
(Note the 2 blank lines at the top of the triangle)

Enter the angle \( \theta \) in degrees:

\[
\begin{align*}
&60 \\
&* \\
&*** \\
&****** \\
&******* \\
&******** \\
&********** \\
&*********** \\
&************ \\
&************* \\
&************** \\
&*************** \\
&**************** \\
&***************** \\
&****************** \\
&******************** \\
&******************* \\
&************************ \\
&************************** \\
&*************************** \\
&**************************** \\
&***************************** \\
\end{align*}
\]

60-degree Sample Output
(Note the 1 blank line at the top of the triangle)

Submit your final (step 3) version of tri.cpp on Vocareum.

5 Evaluation Criteria

1. [1 pt.] Your prime number program produces the correct output for a number with only 2’s as prime factors.
2. [1 pt.] Your prime number program produces the correct output for a number with only 3’s as prime factors.
3. [2 pt.] Your prime number program produces the correct output for a number with at least one factor of 2, 3, and some other prime number (e.g. 5, 7, etc. where the answer is No)
4. [1 pt.] Your prime number program produces the correct output for a number with no 2’s and 3’s as prime factors.
5. [1 pt.] GDB commands were submitted and are reasonable
6. [1 pt.] Your triangle program produces the correct output with a square cutout for the appropriate lines for 45 degrees.
7. [1 pt.] Your triangle program produces the correct output with a square cutout for the appropriate lines for 30 degrees.
8. [1 pt.] Your triangle program produces the correct output with a square cutout for the appropriate lines for 60 degrees.
9. [1 pt.] Both programs were appropriately indented when opened in ‘gedit’ on the Linux VM.

---

1 Adapted from Michael Locasto while he was at Cornell University