CS 103 Lab - Algorithms & Control Structures

1 Introduction

In this lab you will gain experience writing programs that utilize loops and conditional structures.

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 print variable values

3 Background Information and Notes – GNU Debugger (gdb)

3.1 Compiler Options

To debug a program you must tell the compiler to provide extra information for the debugger in the executable file. If you ever directly use a compiler like `clang++` or `g++`, you would pass this flag as a command-line argument to the compiler like this:

```
clang++ -g program.cpp -o program
```

Another useful compiler option is to enable warnings. Save this file as `noinit.cpp`:

```
#include <iostream>
int main() {
   int x;
   return x;
}
```

If you compile `clang++ noinit.cpp`, it succeeds. But we're using `x` without initializing it, which is a bad idea! Wouldn't it be nice if the compiler warned us about this? Try `clang++ -Wall noinit.cpp` and see what happens.

The `compile` command automatically includes `-g` and `-Wall`. So in CSCI 103, you won't have to worry about it. However, as you grow as a developer, you might learn about including these options automatically in `Makefiles` (stay tuned) or `.bashrc` environment variables.

3.2 Starting up GDB

Once a program has been compiled that you want to debug, you start the `gdb` debugger program with the command

`gdb EXECUTABLE_NAME`

Once `gdb` has started, it gives you a new debugger console. You can now 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 ...`

3.3 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. But it can also be useful to stop the execution of your program at an arbitrary point of your choosing. 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 which one. For example:

- `break LINE_NUMBER`
- `break sourcefile.cpp:LINE_NUMBER`

When your program is stopped, here are two commands to extract information, which will help you to identify the bugs in your code:

- `print VARIABLE_NAME` will print the value of a given variable name.
- `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 a few commands you can use.

- `continue` or `c` to run the program until it encounters a breakpoint or finishes
- `next` or `n` to “step over” through the program: go to the next line, skipping any function calls that happen along the way
- `step` or `s` to “step in:” go to the next possible basic instruction, which will enter a function if one is about to be called, or just go to the next line otherwise

To delete a breakpoint: `delete BREAKPOINT_NUMBER`
To list all breakpoints: `info breakpoints`
To quit gdb: `quit`
This is just a summary; see the online documentation or `man gdb` for more commands that will be useful as you develop your programming skills.

4 Procedure

4.1 Develop and Code an Algorithm

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

It can be very useful to plan out pseudocode or a flowchat, to make sure you have a working algorithm, before you start writing your code.

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' to start the program.

- 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).

Demonstrate your program and show your TA/Sherpa your code explaining how it works. Also demonstrate your knowledge of 'gdb' by performing the actions described in the previous paragraph in front of your TA/Sherpa.

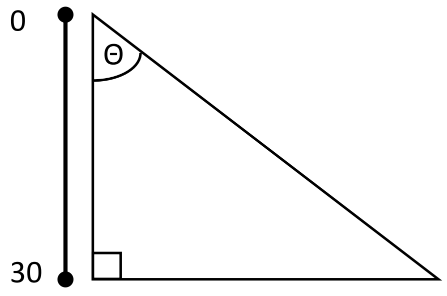
4.2 ASCII Art[i]

The final product of this assignment, `tri.cpp`, will be built up in 3 parts.

[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.). Note: You do not have to put a `'\n'` or `endl` character in every string that you print via `cout`. You should get a triangle similar to the one shown below.

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

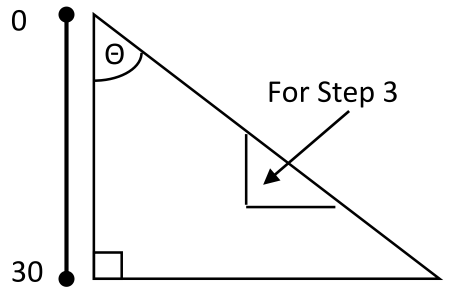
[Step 2] Except for distortions in the proportion of a characters height/width (i.e. the font that Linux uses), the triangle above should be an isosceles triangle (i.e. 45 degrees for each of the non-right angles). We could generalize and pick an arbitrary number of degrees for the angle, Θ, shown below.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



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.

Derive a mathematical expression for the length of the x-axis (i.e. # of *'s) as a function of Θ and the y-coordinates. 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 `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. Since the user is inputting a value in degrees, but the trigonometric functions in cmath use radians, you'll have to convert them. The built-in constant `M_PI` can be useful for this.

[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 ≤ x ≤ 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.



Here are some sample outputs:

```
Enter the angle theta in degrees:
45


*
**
***
****
*****
******
*******
********
*********
**********
***********
************
*************
**************
***************
****************
*****************
******************
*******************
********************
********************
********************
********************
********************
********************
********************
********************
********************
********************
```

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

```
Enter the angle theta in degrees:
60

*
***
*****
******
********
**********
************
*************
***************
*****************
*******************
********************
********************
********************
********************
********************
********************
*******************************
********************************
**********************************
************************************
**************************************
***************************************
*****************************************
*******************************************
*********************************************
**********************************************
************************************************
**************************************************
***************************************************
```

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

Note: Depending on how you code your triangle program, integer rounding, and the chosen angle, the first couple lines may have 0 asterisks.

Demonstrate just this last program to your TA/CP.

[i] Adapted from Michael Locasto while he was at Cornell University