Faculty of Arts and Sciences
Department of Computer Science
CMPS 212 – Intermediate Programming with Data Structures
Assignment 3
Due Date: Sunday July 3rd at midnight

Problem 1. Evaluating Boolean Expressions

You should create a program that is capable of evaluating an arbitrary Boolean expression and returning the proper result. Specifically:

- Your program should prompt for an expression and then read it in from keyboard. It should then evaluate the expression and print the result to screen. This process should repeat until the user indicates that they want to stop.

- In your expressions, you should assume that a value of '0' is 'false', while any non-zero value is 'true'. Your calculator should be able to evaluate the following cases:

  a. A value by itself (i.e. '(1)')
  b. A simple expression containing two values and a binary operator (i.e. '(1 && 0)'). The binary operators that your program must support are '&&' (logical AND), '||' (logical OR), '==' (equals operator), '!=' (not-equals operator), '>' (greater than), and '<' (less than).
  c. A simple expression containing one value and a unary operator (i.e. '(!0)'). The only unary operator you need to worry about is '!' (unary NOT).
  d. A properly parenthesized compound expression containing any combination of the above expressions (i.e. (((3 < 4) && (4 > 3)) || 0).

- A good technique for solving this problem is to have your program manage three stacks, one for operands, one for operators, and one for keeping track of parenthesis. You may implement your solution however you wish, though you may find the following high-level algorithm helpful:

  1. While the user has more data to enter, display a prompt and read in the next expression (read the whole line at once).
  2. For each character in the input string:
     a. Examine the character, and if it is a left parenthesis, push it onto the parenthesis stack, If it is a value, then push it onto the operand stack (also called the "value" stack…you will need to add some additional processing to handle values that are more than 1 character long). If it is an operator, push it onto the operator stack (you will need to add some processing to differentiate operators that are represented in 1 character from those represented in 2 characters). If it is a right parenthesis, then pop the parenthesis stack, otherwise continue from the next input character.
     b. If the pop of the parenthesis stack does not return a left parenthesis, the input is malformed, and this is an error condition. Otherwise pop the operator stack, and if you get no operator, then check to make sure the value stack is not empty. If it is, then you were given an empty expression (i.e. something like '()') and this can be considered an error condition, otherwise if it is not empty, continue from the next character in the input.
     c. If the pop of the operator stack *did* return something, examine the operator you received, and pop one item off the value stack in the case of a unary operator, or two items in the case of a binary operator. Apply the appropriate operation to the

value(s), and then push the result back onto the value stack. If at any point the value stack is empty when you try to pop from it, then your input was invalid and this is an error condition. Assuming no errors have occurred, continue from the next character in the string.

3. Once you have iterated across the entire input string, the result of the expression should be at the top of the value stack, and the operator and parenthesis stacks should be empty (if they are not you have an error condition). Pop this value off the stack and display it as your result. Continue from step 1 as long as the user has more input.

Sample Input/Output:

This program will evaluate boolean expressions and return a value of 'true' if the expression is true, or 'false' if it if false. Your boolean expressions must be fully parenthesized. Enter a blank line at the prompt to quit the program.

Enter the boolean expression to evaluate now: (!0)
Your expression is TRUE

Enter the boolean expression to evaluate now: (!1)
Your expression is FALSE

Enter the boolean expression to evaluate now: (!6734)
Your expression is FALSE

Enter the boolean expression to evaluate now: (5 > 9)
Your expression is FALSE

Enter the boolean expression to evaluate now: (5 < 9) Your expression is TRUE

Enter the boolean expression to evaluate now: (5 == 4)
Your expression is FALSE

Enter the boolean expression to evaluate now: (1 || 1)
Your expression is TRUE

Enter the boolean expression to evaluate now: (1 || 0)
Your expression is TRUE

Enter the boolean expression to evaluate now: (0 || 0)
Your expression is FALSE

Enter the boolean expression to evaluate now: ((34 > 100) || (34 < 100))
Your expression is TRUE

Enter the boolean expression to evaluate now: (1 && 0)
Your expression is FALSE

Enter the boolean expression to evaluate now: (1 && 1)
Your expression is TRUE

Enter the boolean expression to evaluate now: (0 && 0)
Your expression is FALSE

Enter the boolean expression to evaluate now: ((34 > 100) || (!(34 < 100)))
Your expression is FALSE

Enter the boolean expression to evaluate now: ((5 > 6) == 0)
Your expression is TRUE

Problem 2 (STL Containers and iterators)

1. Write a function *product* that takes a list<int> as an argument and that returns the product of the values in the list. You should use an iterator. Write a tester program to test your function.

2. Write a function *hasNegativeSum* that takes a list<int > as a parameter and that returns true if the list has a negative sum and that returns false otherwise. In particular, the method should compute a running sum of values starting with the first value and should determine whether the running sum ever goes negative. For example, if a variable called list stores the following list:

   (6, 11, -20, 7, 14, 93)

   then the call *hasNegativeSum*(list) should return true because the sum of the first three elements is negative (6 + 11 + -20 = -3). If the list had instead stored these values:

   (6, 15, -20, 7, -4, 5, -9)

   then *hasNegativeSum*(list) would return false because even though the list contains some negative numbers, the running sum starting with the first value never goes negative.

   You are not allowed to call any other methods of the list<int> container to solve this problem. You must solve it using an iterator.

   Write a tester program to test your function

3. Write a function *mirrorHalves* that accepts a queue of integers as a parameter and replaces each half of that queue with itself plus a mirrored version of itself (the same elements in the opposite order). For example, suppose a variable q stores the following elements (each half is underlined for emphasis):

   front [10, 50, 19, 54, 30, 67] back

   After a call of *mirrorHalves*(q);, the queue would store the following elements (new elements in bold):

   front [10, 50, 19, **19, 50, 10**, 54, 30, 67, **67, 30, 54**] back

   If your method is passed an empty queue, the result should be an empty queue. If your method is passed a null queue or one whose size is not even, your method should throw an exception called *EmptyStackException* (How to create and throw an exception? that's part of your homework).

You may use one stack or one queue (but not both) as an auxiliary storage to solve this problem. You may not use any other auxiliary data structures to solve this problem, although you can have as many simple variables as you like. You may not use recursion to solve this problem. For full credit your code must run in O(n) time where n is the number of elements of the original queue.

Write a tester program to test your function.