Programming Project 2

Due: Wednesday, 9/26/18 at 11:59 pm

Balanced Symbol Checker

For this lab, write a C program that will determine whether input is given with properly balanced symbols. We will often use symbols together to specify the beginning and ending of an item, such as the use of parentheses in a mathematic expression or the use of curly braces in a C, C++ or Java program. For this program, we will be checking the following symbol pairs:

- parentheses: ( )
- curly braces: { }
- square brackets: [ ]
- angle brackets: < >

This program will require the use of a stack implemented in a dynamic array. This dynamic array is to grow to a larger size when a push operation would be done to a full array causing an array overflow. For this program, your dynamic array MUST start with 2 positions in the array. When the array needs to grow, it size MUST grow by 2 additional positions each time (note the array to grow in size from 2 to 4 to 6 to 8 to 10 to …).

The **push** operation is now defined as follows:

```
if (the stack array if full)
        grow the array
add the value to the stack array
increment the top-of-stack value
```

The **grow** operation is defined as follows:

```
Create a temporary pointer to the current stack array
Allocate a new dynamic array of the larger size and Have the stack array variable refer/point
   to the new dynamic array
Copy the existing values from the current stack array to the new dynamic array
Deallocate the current stack array
Update the maximum stack size variable
```

Input

The input for this program will come from standard input. Each line of input will be a single expression that is to be checked for balanced symbols. You may assume that each line of input is less than 300 characters long. The program must loop to read in multiple lines of input. If the input on the line contains only the letter q or Q, quit the program.

Since we have limited the length of the input and are trying to process one line of input at a time, the best way to read the input is the **fgets()** function in the <stdio.h> library. Since we are reading from standard input, you are to use the value of **stdin** for the third parameter of **fgets().** This causes fgets() to read input from the standard input. **You MUST use fgets() for this programming project to read in the input**. See the base code to see how this is done. If you are not familiar with the fgets() functions, do a Google search and read. The cplusplus.com website has a good reference page on fgets().

Stack Use Algorithm

To check for balance symbols in an expression, the expression is inspected from left to right after the entire line is read in. The algorithm only cares about opening symbols, closing symbols and the end of the expression. Any other input is ignored. The stack must be empty at the start of the expression.

When an opening symbol is encountered, this symbol is pushed onto the stack. The opening symbols are: ({ [and <.

When a closing symbol is encountered, check the symbol at the top of the stack.

- If the symbol on the top of the stack is the corresponding opening symbol, pop the stack and continue.
- If the symbol on the top of the stack is NOT the corresponding opening symbol, the expression is NOT balanced and the wrong closing symbol was encountered. (Error #1)
- If the stack is empty, the expression is NOT balanced and there is a missing opening symbol. (Error #2)

When the end of the expression is encountered (i.e. the end of the input line), check to see if the stack is empty.

- If the stack is empty, then the expression was balanced.
- If the stack is NOT empty, the expression was not balanced and there is a missing closing symbol. (Error #3)

Since the only input we really care about are the 8 characters that form the 4 symbol pairs and determining when the "end of the expression" is reached, any other input on the line can be ignored.

Output

For each line of input, your program should display the input and specify whether the expression

- is balanced
- is unbalanced because it is **expecting a different closing symbol** (the wrong closing symbol is on the top of the stack) (Error #1)
- is unbalanced because it is **missing an opening symbol** (stack is empty when a closing symbol is encountered) (Error # 2)
- is unbalanced because it is **missing a closing symbol** (line ends while the stack is not empty) (Error # 3)

For the unbalanced expression, print the "up arrow" character at the place where the unbalanced error occurred with a message. Only report the first error encountered in any line of input. The following are some examples output showing the 4 possible outcomes:

```
( ( a a ) < > [ [ [ { [ x ] } ]]] <>)
Expression is balanced

( ( a a ) < > [ [ [ { [ x ] ]]] <>)
                            ^ expecting }

( ( a a ) ) < > > [ [ [ { [ x ] } ]]] <>)
                ^ missing <

( ( a a ) < > [ [ [ { [ x ] } ]]]
                                  ^ missing )
```

Use of C struct and C functions

When writing your code, you MUST place all of the data items needed for the stack in a C struct called "stack". These data items must include the following (and may include others if needed).

- the pointer to the dynamic array that actually holds the stack
- the integer variable specifying the current size of the dynamic array
- the integer variable specifying the top of the stack

The instance of this struct MUST be declared locally in main(). It may NOT be global. (Note: If you want it to be declared locally in some other function other than main(), that is also OK.)

In your program, you MUST write functions for:

- initializing the stack (typically named: init( )),
- checking if the stack is empty (typically named: is_empty( )),
- pushing/adding an element onto the stack (typically named: push( )),
- popping/removing an element off of the stack (typically named: pop( )),
- accessing/returning the top element on the stack (typically named: top( )), and
- clear the stack so that it is empty and ready to be used again (typically named: clear( )).

All of these functions MUST take **as their first parameter** a pointer to the struct that contains the instance of the stack that is being used. The only exception to this is that the initializing function may return a newly created instance and return a pointer to this instance.

Command Line Argument: Debug Mode

Your program is to be able to take one optional command line argument, the -d flag. When this flag is given, your program is to run in "debug" mode. When in this mode, your program is to display a message whenever an item is pushed or popped from the stack. This message must include the character being pushed or popped. Also, when the stack grows, you are to explicitly state the old and new size of the dynamic array for the stack as well as indicate the number of values copies from the current to the new dynamic array.

When the flag is not given, this debugging information should not be displayed. One simple way to set up a "debugging" mode is to use a "Boolean" variable which is set to true when debugging mode is turned on but false otherwise. This debugging mode variable can be set up as a global variable if desired. Then using a simple if statement controls whether information should be output or not.

```
if ( debugMode == TRUE )
        printf (" Debugging Information \n");
```

Program Submission

You are to submit the programs for this lab via the **Information** page in Blackboard.

To help the TA, name your file with your net-id and the assignment name, like:

- ptroy1projX.c

Where X is the number of the project assignment (and ptroy1 is replaced with your own net-id).