CMPS 12M-02
Data Structures Lab
Lab Assignment 4 (Lab4)
Due: Monday, November 13 by 11:59pm

- **Assignments submitted late will lose 1 point if submitted less than one day (24 hours) late.**
- **Assignments submitted more than one day late will receive no credit.**

The purpose of this lab assignment is to have you program in the C programming language, including standard input-output functions, command line arguments, File IO, and compilation with Makefiles. This file describes the Lab4 Assignment. But to do Lab4, you should first read the file *Some_Helpful_Background_on_C.pdf*. Students already familiar with C are probably familiar with most of that material; students who didn't know C but are taking CMPS 12M will recognize that we're repeating some material in that file.

What to turn in

Write a C program called `balanced.c` that behaves exactly like the program `Balanced.java` that most of you wrote for HW3. (Note the difference in naming convention. For C, the file name starts with a small letter, while for Java, it starts with a capital letter. Admittedly, this is annoying, but is quite common among programming in different languages.) However, you must use an Array-Based implementation of a Stack, not a Reference-Based Implementation of a Stack.

Since a few students taking CMPS 12M-02 passed CMPS 12B in previous quarters, so they are not taking CMPS 12B-02, we'll repeat the explanation of what the program is supposed to do.

Main objective: Determine whether any expression with multiple kinds of brackets, braces and parentheses is balanced using an Array-Based Stack. You must implement the stack directly yourselves. Using a Reference-Based Stack, or a stack library, or JCF stacks, or a list ADT will earn no credit.

Types of brackets: There are four "parenthesis/bracket" pairs:

- **(and)**
- **[and]**
- **{ and }**
- **< and >**

A line may contain other characters (including spaces) besides these 8 characters, but those characters are ignored. An expression is balanced if all the "open parentheses" are closed by a matching "close parentheses", and nested parenthesis match, so that (for example) you can't have "]" closing "{", or a stray extra ")" when there isn't an open "(".

For example, the following lines are all not valid (N):

```
 ab(x]w
alpha{  beta<gamma}delta>
(([[Turing]])
The ( quick [ brown ( fox ( jumped  ) over ) the ] lazy ) dog )
< { }
> html <
```

And the following lines are all valid (Y):

ab[x]w
alpha{ beta<gamma>delta}
([[Turing]])
The (quick [brown (fox (jumped) over) the] lazy) dog
metamorphosis
< html >
([< { } >]) < < ( ) > >

You must implement your solution to Lab4 using an Array-Based Stack. Furthermore, you have to write your own stack from scratch. You cannot use any built-in libraries for stacks or linked lists, nor can you use an ADT for lists or a Reference-Based stack.

Your program `balanced.c` will take two command line arguments naming the input and output files respectively (following the `FileIO.c` example that's in *Some_Helpful_Background_on_C.pdf*). It will read in each line in the input file, then print a separate line in the output file for each line in the input file. If the input line is valid (that is, balanced), then the output line should be the single character Y; if the input line is invalid (that is, not balanced), then the output line should be the single character N. **You may assume that there aren't more than 80 characters on any input line**; this is important, since you're using an Array-Based implementation of Stacks.

Your balanced.c program should contain functions corresponding to the usual stack interface (createStack, push, pop, isEmpty, peek and popall). Place the definition of these functions after all preprocessor directives but before the function `main()`. C requires that the signature of a function be known before the first call to the function. Recall that the signature is the name of the function, along with its return type and the type and number of the formal parameters. See Charlie McDowell's "C for Java Programmers" book (Section 2.1, Declaring Functions) for two approaches showing how to declare functions before using them.

The function definition should take the form of <return type> <function name>(<parameter1>). Your main function will have some similarities to the `FileIO.c` example provided in *Some_Helpful_Background_on_C.pdf*, except that the while loop should contain appropriate calls to the stack functions. **You cannot use built-in libraries for stacks or linked lists to do this assignment. You must write the functions yourself, using an Array-Based Stack**.

One function in the library `string.h` that you will find useful is `strlen()`, which returns the length of its string argument. For example the following program prints out the length of a string entered on the command line.

```
/*
*  charCount.c
*  prints the number of characters in a string on the command line
*/
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

int main(int argc, char* argv[]){
   if(argc<2){
      printf("Usage: %s some-string\n", argv[0]);
      exit(EXIT_FAILURE);
   }
   printf("%s contains %d characters\n", argv[1], strlen(argv[1]) );
   return EXIT_SUCCESS;
}
```

Remember that a string in C is a `char` array, not a separate data type as it is in Java. (This may help you write your Stack implementation in C.) Recall also that arrays in C cannot be queried as to their length. How then does `strlen()` work? Actually a string in C is a little bit more than a `char` array. By convention C strings are terminated by the null character `'\0'`. This character acts as a sentinel, telling the functions in `string.h` where the end of the string is. Function `strlen()` returns the number of characters in its `char*` (`char` array) argument up to (but not including) the null character.

Also recall that arrays in C are passed by reference, not by value. An array name in C is literally the address of (i.e. a pointer to) the first element in the array. Arrays, strings, and pointer variables in C will be discussed in subsequent lab assignments.

To test `balanced` with specific input and output files, you may have to enter:

```
./balanced input_file output_file
```

unless your PATH already includes the current directory (.)

Write a Makefile that creates an executable binary file called `balanced`, and includes a `clean` utility. **Note that you're required to have a single c program file for Lab4, `balanced.c`, not multiple c files**.

The Lab4 folder should contain: `balanced.c`. `Makefile`, and `README` and the 4 test files mentioned in the next paragraph. There is no checking script requirement (or spec file) for Lab4, so you'll have to check that you have the right files by yourselves.

In Piazza under Lab4, there is a Lab4files.zip file that contains the same 4 files, *test-input.txt*, *test-output.txt*, *more-input.txt* and *more-output.txt* that were provided for HW3. The output files contain the correct output for the corresponding input file. You can use these to test your program.

Grading:

You code should terminate within 3 minutes for all runs. If it doesn't, we will not give you credit. There is no checker requirement for this assignment. However, to receive credit, you must also submit your git commit id using the Google Form for Lab4. You can resubmit the Google Form if you want to do so, but we'll grade you based on your latest re-submission, so be careful about this. At this point, everybody should be able to do this correctly!

1. (10 points) Everything was done correctly.
2. You will lose one point for each error that you have when we test your program on *test-input.txt* and on *more-input.txt*, the same files that were used for testing HW3 (which also appear on Piazza under Lab4).
3. (0 points) balanced.c does not compile or you've submitted the wrong files.