ECE 361 Fall 2022
Homework #4

THIS ASSIGNMENT SHOULD BE SUBMITTED TO CANVAS BY 10:00 PM ON SUN, 20-NOV-2022. THERE IS A 23 HR, 59 MIN GRACE PERIOD FOR THIS ASSIGNMENT BEFORE IT IS CONSIDERED LATE. THE PENALTY FOR BEING LATER THAN THAT IS 2 PTS/DAY. NO HOMEWORK #4 SUBMISSION WILL BE ACCEPTED AFTER 10:00 PM ON TUE, 22-NOV-2022 BECAUSE I WILL BE REVIEWING THE SOLUTION IN CLASS ON WEDNESDAY.

THE ASSIGNMENT IS WORTH 100 POINTS BUT THERE IS AN EXTRA CREDIT OPPORTUNITY WORTH UP TO 5 ADDITIONAL POINTS. FOR THIS ASSIGNMENT YOUR TOTAL SCORE COULD BE UP TO 105 POINTS IF YOU EARNED ALL OF THE POSSIBLE POINTS. SOURCE CODE MUST BE TURNED IN AS .C OR .H FILES. CONSOLE TRANSCRIPTS MUST BE .TXT FILES (SEE NEXT PARAGRAPH). SUBMIT THE PACKAGE AS A SINGLE .ZIP, .7Z,.TGZ, OR .RAR FILE (EX: `ece361f22_rkravitz_hw4.zip`) TO YOUR CANVAS HOMEWORK #4 DROPBOX.

CONSOLE TRANSCRIPTS SHOULD INCLUDE YOUR NAME, USING THE `whoami` COMMAND AND THE STATEMENTS WITHIN YOUR CODE THAT SHOWS THE EXECUTION DIRECTOR. TRANSCRIPTS MUST BE A .TXT FILE SAVED FROM THE GCC COMMAND LINE. PLEASE INCLUDE A NUMBER OF TEST CASES (INCLUDING EDGE CASES) IN EACH TRANSCRIPT TO DEMONSTRATE THE FULL FUNCTIONALITY OF YOUR PROGRAM. THE TRANSCRIPT IS YOUR CHANCE TO SHOW OFF YOUR HARD WORK.

ACKNOWLEDGEMENT: THE RPM CALCULATOR PROBLEM IS BASED ON A PROGRAMMING PROJECTS PROVIDED BY K. N. KING IN *C PROGRAMMING: A MODERN APPROACH*. *STUDENTS FROM PREVIOUS CLASSES FELT THAT THIS WAS ONE OF THE MOST INTERESTING AND FUN*.

FOR THIS ASSIGNMENT, YOU WILL BE TURNING IN THESE FILES (NOTE THAT THE FILE NAMES ARE JUST SUGGESTIONS, BUT PLEASE DO NAME YOUR FILES MEANINGFULLY):

- PROBLEM1_1_TRANSCRIPT.TXT (CONSOLE TRANSCRIPT AND EXPLANATION FOR PART 1)
- PROBLEM1_2.C, .H (SOURCE CODE FOR YOUR STACK ADT)
- PROBLEM1_2_TRANSCRIPT.TXT (CONSOLE TRANSCRIPT AND EXPLANATION FOR PART 2)
- PROBLEM1_3.C, .H (SOURCE CODE FOR YOUR RPN CALCULATOR)
- PROBLEM1_3_TRANSCRIPT.TXT (CONSOLE TRANSCRIPT FOR PART3)
- (OPTIONAL) PROBLEM1_3EC.C, .H (SOURCE CODE FOR YOUR ENHANCED RPN CALCULATOR)
- (OPTIONAL) PROBLEM1_TRANSCRIPT.TXT (CONSOLE TRANSCRIPT FOR YOUR ENHANCED RPN CALCULATOR)
- PROBLEM2_1.MAK (MAKEFILE FOR EITHER YOUR RPN OR ENHANCED RPN CALCULATOR)
- PROBLEM2_2.TXT (CONSOLE TRANSCRIPT SHOWING YOUR MAKEFILE WORKING)

Introduction:

The problems on this homework assignments are related and build on each other. The final application will be a simple Reverse Polish Notation calculator. The calculator will be stack-based (RPN is a LIFO operation) that makes use of a Stack ADT and API that you will implement. Your Stack ADT must be implemented using the API for a Singly Linked List ADT provided in the "starter" code. While this may seem to be inconvenient and maybe even restrictive, I think you will find that most of the "heavy lifting" for your Stack can be accomplished with Linked List API calls. You will manage the build process for this application using the GNU make utility and a `makefile` that you will write.

The Reverse Polish Calculator

Early HP calculators used a system of writing mathematical expressions known as Reverse Polish Notation (RPN). In RPN, operators (+, -, *, /) are placed after their operands instead of between their operands. For example, the expression 1 + 2 would be written as 1 2 + and 1 + 2 * 3 would be written as 1 2 3 * +. RPN (https://en.wikipedia.org/wiki/Reverse_Polish_notation) expressions can be easily evaluated using a stack. The algorithm involves reading the operators and operands in an expression from left to right, performing the following operations:

- When an operand is encountered, push it onto the stack
- When and operator is encountered, pop its operands from the stack, perform the operation on those operands, and then push the result back onto the stack.

In the base (required) implementation of the calculator operands are single-digit integers (`0`, `1`, `2`…, 9). The operators are `+`, `-`, `*`, `/`, and `=`. The `=` operator causes the top stack item to be displayed. The stack is cleared after the = operator is executed, and the user is prompted to enter another expression. The process continues until the user enters a character that is not a valid operator or operand. For example:

```
Enter an RPN expression: 1 2 3 * + =
Value of expression: 7
Enter an RPN expression: 5 8 * 4 9 - / =
Value of expression: -8
Enter an RPN expression: q
```

There is, however, an extra credit opportunity for this problem. With some changes to the input handling, it does not take much additional coding to support single and multiple digit, positive and negative integers as operands. *Hint: I made good use of the C-library functions `strtok()` and `strtol()` to implement a general purpose RPN calculator.*

Problem 1 (80 pts) Application programming with stacks

1. (10 pts) Study the code for the `SLLinkedList` ADT provided in the `starters` folder. This ADT provides a Singly Linked List API that can be put to good use in building your Linked List-based Stack. This ADT is based on Karumanchi's singly linked list implementation, but I have modified it to better hide (at least I think so) the internal architecture and to make it easier to build an application with more than one Linked List (not the case for this application, but I considered the reusability of the ADT when I first constructed it).
    a. Build the Linked list ADT using the `gcc` compiler chain on the command line, along with the `test_LinkedList.c` application. Run the application and produce a transcript of a successful run. Add an explanation (I'm expecting a paragraph or two) to why you think the Linked List ADT is working correctly.
    b. Use `gcc` to generate an object (`.o`) file for the Link List ADT. You will use it when you build your Stack ADT and RPN application.

2. (30 pts) Create a Linked List Stack ADT that implements the functions shown in `starters/stackADT/stackADT_LL.h`. Your Stack ADT should be implemented using the API functions in `SLLinkedList`. I have provided a `stackADT_LL.c` file in the `starters` folder that provides a template for your ADT.
    a. Build your Stack ADT using the `gcc` compiler chain on the command line along with `test_stack_LL.c` or (better yet) a test program of your creation. Run the application and produce a transcript of a successful run. Add an explanation (I'm expecting a paragraph or two) to why you think your Stack ADT is working correctly.

3. (40 pts) Write an application that evaluates RPN expressions using the Stack ADT implementation that you created in Part 2. The functionality is described earlier in this document. A stack is an optimal data structure for an RPN calculator because operands are pushed onto the stack as they are entered and the top two entries on the stack are popped off the stack and the operation performed with the result of the operation pushed back onto the stack. For example, consider the sequence `5 9 + =` will perform the following operations:
    - Push 5 on the stack (stack: 5)
    - Push 9 on the stack (stack: 9 5)
    - Pop 9, Pop 5, ADD, Push 14 (stack: 14)
    - Pop 14, Display the result, destroy the stack, create the stack anew (stack:)

a. Compile and execute your RPN application using the `gcc` command line. Include the source code and a transcript showing the output from several test cases. Your test cases should include these test cases and several your own:
    - 1 2 3 * + = (Result: 7)
    - 5 8 * 4 9 - / = (Result: -8)

Extra credit opportunity (up to 5 points):

As specified, the RPN calculator operates on single digit positive numbers. Enhance your application to successfully handle positive and negative integers of one or more digits. For example, `8, 508, +27, -3, -127` would all be valid operands.

a. Create a new version of your RPN calculator that implements the enhanced input processing. Compile and execute your new RPN application using the `gcc` command line. Include the source code and a transcript showing the output from several test cases. Your test cases should include all your test cases from your RPN calculator, these additional cases, and several your own:
    - -50 125 + = (Result: 75)
    - 345 -10 * = (Result: -3450)

Problem 2 (20 pts) Application programming with stacks

1. (10 pts) Write a `make` file to handle the build process for either of your RPN calculator solutions (you only need to pick one). Your `makefile` should include a target to "clean" (delete all but the source code files) your code. To make this problem more interesting treat the Linked List ADT as a "black box"; that is, do not include the targets necessary to rebuild the Linked List ADT. Instead, include the `LinkedList.o` file you generated in Problem 1, Part 1. Since the Stack ADT is your code, include targets to rebuild your Stack ADT in the `makefile`.

2. (5 pts) Build your application using the `makefile` you wrote. Execute your program one last time. Include a transcript (log) that shows your `makefile` at work to build the application. The transcript should also demonstrate that the `makefile`-generated program is working correctly. Identify your transcript with your user name and working directory.

3. (5 pts) Push your source code (.c and .h and make file) files and transcript(s) to your GitHub repository for the assignment. Upload an archive of your repository to your Canvas Homework #4 dropbox. GitHub classroom will take a snapshot of your archive at the deadline, but you can push updates after that.

IMPORTANT: If you push updates after the deadline, please let Emily and me know so that we are sure we are looking at your final submission.