Erik Sjoberg  
[esjoberg@cmu.edu](mailto:esjoberg@cmu.edu)

# Assignent 3: Postfix Equation Solver

## ADT Design

a. Describe the design of your various ADTs, the data structures you selected and why you selected them, what information and operations did you hid and your reasoning for your decisions. Describe how a program uses your ADTs.

**ADT Choices**

For this assignment, I chose to base my code on an implementation of the Stack ADT. I chose to use a stack because the characteristics of a stack fit the requirements of a postfix equation solver perfectly. It is necessary to store values and retrieve them in a first-in-last-out fashion, and this is the only access pattern required.

Because my postfix equation solver (RPN) algorithm as implemented with a stack requires only a single pass over the input string, I chose not to implement a more dynamic list structure to store the input. This choice was reinforced because the C standard library provides the strtok() function explicitly for the purpose of iterating once over a set of delimited tokens in a string, and replicating this functionality with my own custom implementation of a list would reduce the simplicity, performance, and usability of the code.

**ADT Design**

I implemented my stack ADT with a struct containing a static array of floats (1000 in this case), as well as an index pointer to the current “top” of the stack. Pushing is achieved by incrementing the index, and writing the appropriate value in that location of the array, while popping is implemented similarly. An initialization routine and assertions help ensure that the stack invariants hold (index >= 0 always, >= 1 on pop, index <= size of array always, etc.).

This design is extremely simple and performant, and meets the needs of this assignment perfectly. The downside of this approach is the allocation of a chunk of memory, most of which will never be used, which also limits the length of expression that can be calculated. However, this limitation is relatively insignificant due to the memory management of modern operating systems and the large pools of memory in modern computers as compared to the requirements of this program.

**Information Hiding**

My stack ADT hides the details of the internal memory structure and associated indices from the user, and also defines a type to hide the details of the struct-based implementation. The user interacts through provided functions only, and does not need to know any details of the implementation besides the maximum size limit (which is included in the C file as the #DEFINE STACK\_SIZE)

**How my ADT is used**

I provide the user with standard stack functions for initialization, push, pop, and peek. These functions are used by providing a pointer to the stack object as the first argument. Although the user defines a FloatStack object, then uses init\_stack() to prepare it for use.

## Program Analysis

b. Show the analysis of your application in terms of correctness and complexity (performance and memory) as described in class. You may use any representation and analysis techniques presented in class (or any others you may have found appealing… if in doubt, check with the instructor).

**Correctness**The key invariants in my system take place in the computeRPN() function, and are:

1. Items on stack >= 0
2. Items on stack <= STACK\_SIZE = 1000
3. Input string is valid (0-terminated)
4. Within the main loop,
   1. Number of tokens remaining on input string is always decreasing (convergence)
   2. For each number, stack size increases by one
   3. For each operator, stack size decreases by one
5. At end of main loop, stack will contain one number (the solution)

Because the string of tokens is only iterated over once, the program can be shown to always terminate provided a valid string.

**Complexity: Performance**O(n)

This can be seen in the inner loop of the computeRPN() function, which iterates over a fixed-size string input only once as can be seen in the pseudocode below:

* For token in string:
  + IF token is number:
    - Push on stack
  + Else
    - Pop twice, calculate, then push result on stack

In no cases will the program perform more than N calculations of fixed cost, where N is the number of tokens in the string.

**Complexity: Memory**  
O(n) (on fancy operating systems)

This can also be seen in the inner loop of comuteRPN(). The worst case event is a long string of number-tokens followed by a sequence of operators (+, -, /, \*) which will result in every number token being pushed onto the stack before the stack size starts to shrink again due to each operator removing one net item from the stack.

O(1) (on simple operating systems)

Due to the array-based implementation of my stack ADT, the memory usage on simple operating systems with no fancy memory management is static, depending only on the size of the pre-allocated stack.

## Build / Run Instructions

1. Change path to the folder containing the source code
   1. cd /path/to/sourcecode/folder/
2. Compile with gcc (use MinGW if on Windows)
   1. gcc -Wall solver.c -o calc
3. Run calc.exe (from console, or double click)
4. Input equation in RPN, then press enter
5. The solution will be shown at the bottom of the window
   1. Additional output is also shown indicating the process of calculating the answer, and any errors or warnings that may have occurred
6. Enter another equation if desired.
7. Exit by typing “exit”