

RPN Calculator

For this project, you will design and implement a simple integer calculator, which interprets reverse Polish notation (RPN) expressions. There is no graphical interface. Calculator input will be provided in an input script file and feedback from the calculator application will be written to an output file.

The RPN Calculator will use an internal stack to manage interpreting RPN expressions. How to interpret RPN expressions will be discussed in class. The RPN Calculator will also have an internal register (memory location), called the Accumulator, which will be used to store the last value that was computed. When the Calculator is “turned on”, the Accumulator will store zero, and its internal stack will be empty. The Calculator may have any additional internal features you find useful or necessary.

For our purposes, the input to the Calculator will consist only of integer operands and operator symbols, where:

- an integer operand is a sequence of one or more base 10 digits, optionally preceded by a plus or minus sign, and containing no embedded spaces or tabs.
- an operator symbol is one of the characters +, -, *, or /. All operators are binary.

Data Structures:

The primary data structures element of this project is the stack used to parse the RPN expressions. Your implementation is under the following specific requirements:

- You must use a stack, encapsulated as a C++ template. (Templated structs are an abomination.)
- The underlying stack structure must be linked, not array-based. The list nodes must also be implemented via a C++ template.
- The stack must protect against overflow and underflow, with some sensible means for the user to detect when such an error has occurred. You may make good use of an internal error state data member for this; whereas my implementation uses exceptions. You may use either approach, or another.

Your design must make appropriate use of classes. The problem specification implies the existence of additional classes besides those involved in the implementation of the stack. Aside from list nodes used only within an encapsulating class, data members of classes must be private.

Your design should incorporate some degree of error handling. As noted below, the input file will be syntactically correct, so syntax checking of input is not required. On the other hand, stack errors may occur (e.g., invalid RPN input), and illegal numerical operations (division by zero) could be specified. Your program must deal with those situations without crashing. Any such error should produce a meaningful error message.

Script File Description:

Your program **must** read its input from a file named `CalculatorIn.txt` — use of another input file name will almost certainly result in a score of 0. Lines beginning with a semicolon (`;`) character are comments; your program will ignore comments. An arbitrary number of comment lines may occur in the script file.

The following commands may occur in the script file:

enter [<integer operand> | <operator symbol>]

This will cause the given token to be passed to the Calculator (probably as a string, but that’s your decision). The Calculator should determine whether it has been passed an operand or operator, and take the appropriate action. The token will be syntactically valid, so you are not required to check it for errors; however, a truly robust implementation would detect an error during parsing and generate a useful error message.

display value

This will cause the Calculator to display the current value in the Accumulator to the output file. See the output section for formatting instructions.

display stack

This will cause the Calculator to display a list of the values that are currently stored in the RPN stack to the output file. See the output section for formatting instructions.

clear

This resets the Accumulator **and memory locations** to zero, and clears the Calculator's internal stack.

exit

This causes the Calculator application to terminate. The input file is guaranteed to end with an exit command.

You may assume that the input file will conform to the given syntax, so syntactic error checking is not required. However, you should be careful about logical errors, such as division by zero, in the RPN evaluation.

Here is a sample script file:

```
; Sample script for RPN Calculator project.
;
; Enter some operands:
enter 17
enter 42
enter -3
; Multiply the last two operands:
enter *
; Check the last computed value:
display value
; Check the calculator stack:
display stack
; Add that to the first operand:
enter +
; Check the value and stack again:
display value
display stack
; Clear the calculator and check:
clear
display value
display stack
; Try an illegal operation:
enter +
; Load another operand and try an operation:
enter 100
enter /
display value
display stack
; Try an illegal operation:
enter 0
enter /
; Quit:
exit
```

There will be a single tab character after each command (whether or not it takes a parameter). There will be no other "extra" characters in lines that contain commands.

Output File Description:

Your program **must** write its output to a file named `CalculatorOut.txt` — use of another output file name will undoubtedly result in a score of 0. Since your output for this assignment will be graded automatically, you must adhere to the specified format exactly. The first two lines should contain your name, course and project title, followed by a blank line, as shown. You must echo the commands (but not the comments) to the output file, labeled as shown. The output, if any, from processing each command must be written to the output file immediately after the command echo, as shown. Each command and its accompanying output must be delimited in some way (e.g., the lines of hyphens I used).

Be sure you read the description of scoring in the *Student Guide* to the Curator System. It is generally important that you use the same labels, including spelling and capitalization and punctuation.

<pre>William D McQuain CS 2704 RPN Calculator ----- Command: enter 17 ----- Command: enter 42 ----- Command: enter -3 ----- Command: enter * ----- Command: display value -126 ----- Command: display stack 0: -126 1: 17 ----- Command: enter + ----- Command: display value -109 ----- Command: display stack 0: -109 ----- Command: clear</pre>	<pre>----- Command: display value 0 ----- Command: display stack stack is empty ----- Command: enter + Error: not enough operands ----- Command: enter 100 ----- Command: enter / Error: not enough operands ----- Command: display value 0 ----- Command: display stack 0: 100 ----- Command: enter 0 ----- Command: enter / Error: attempt to divide by zero ----- Command: exit -----</pre>
--	---

The output above was produced by my solution, executed on the sample input file given earlier. Memory and stack values should have an index label, as shown. You are not required to use this exact horizontal spacing.

What to turn in and how:

Submit your C++ source code file to the Curator System. Instructions for submitting to the Curator are given in the *Student Guide* at the Curator website: <http://ei.cs.vt.edu/~eags/Curator.html>. Be sure to follow those instructions carefully. You will submit your assignments via the URL:

<http://eags.cs.vt.edu:8080/curator/>

You will be allowed to submit your solution up to five times. Your highest score will be counted.

Evaluation:

Your submitted program will be assigned a score based upon the runtime testing performed by the Curator System. After that, your program will be given a brief evaluation by one of the GTAs, who will consider:

- whether your implementation makes appropriate use of data structures (in particular, the use of a well-designed stack for the RPN parsing), and
- whether your design shows a good object-oriented decomposition of the given problem, and
- whether the internal documentation of your code is acceptable.

This evaluation will produce a deduction (possibly zero, of course) that will be applied to your runtime testing score to produce your final score for the project.

You should document your implementation in accordance with the Standards Page on the course website. It is possible that your program will be evaluated for documentation as well as for correctness of results. If that is done, your submission that achieved the highest score will be evaluated by one of the TAs, who will assess a deduction (ideally zero) against your score from the Curator.

Note well: if you make two or more submissions that are tied for the highest score, the earliest of those will be graded. There will be absolutely no exceptions to this policy!

Moral: code and document to meet requirements from the beginning, rather than planning to retrofit documentation into a finished program.

Pledge:

Each of your program submissions must be pledged to conform to the Honor Code requirements for this course. Specifically, you **must** include the following pledge statement in the header comment for your program:

```
// On my honor:  
//  
// - I have not discussed the C++ language code in my program with  
// anyone other than my instructor or the teaching assistants  
// assigned to this course.  
//  
// - I have not used C++ language code obtained from another student,  
// or any other unauthorized source, either modified or unmodified.  
//  
// - If any C++ language code or documentation used in my program  
// was obtained from another source, such as a text book or course  
// notes, that has been clearly noted with a proper citation in  
// the comments of my program.  
//  
// - I have not designed this program in such a way as to defeat or  
// interfere with the normal operation of the Curator System.  
//  
// <Student Name>
```

Failure to include this pledge in a submission is a violation of the Honor Code.