

## 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 parsing RPN expressions. Parsing RPN is described in detail in Kruse/Ryba, and will be discussed in class. The RPN Calculator will also contain a number of memory locations that are user-accessible, for storing results for later use. These memory locations will be referred to by sequential integer labels, beginning with zero.

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 and internal memory locations will store zeros, and its internal stack will be empty.

The Calculator may have any additional internal features you find useful or necessary.

For our purposes, an RPN expression will contain only operands and operators, in valid RPN sequence, where:

- an operand is a sequence of one or more decimal digits, optionally preceded by a plus or minus sign, and containing no embedded spaces or tabs.
- an operator is one of the characters +, -, \*, or /.

## 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.
- The internal memory locations maintained by the calculator must be allocated dynamically; whether an array or a linked structure is used is a design decision you must make. (However, in this case, one alternative is clearly superior.)

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 (although only if there's a flaw in the input or in your implementation), 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.

If an error occurs during the evaluation of an RPN expression, that will mean that there's an error in your code.

## Driver 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 input file. The first non-comment line will consist of a text label, followed by a tab character and then a positive integer value which specifies how many user-accessible memory locations the RPN calculator will have, such as:

Memory size	10
-------------	----

Each remaining line of the input file will specify one of the commands described below. Each line consists of a sequence of “tokens” which will be separated by single tab characters. A newline character will immediately follow the final “token” on each line.

**enter**      <token>

This will cause the given token to be passed to the Calculator (probably as a string, but that’s your decision). The Calculator will parse the token and determine whether it is 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 robust implementation would detect an error during parsing and generate a useful error message.

**accumulator**

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

**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.

**memory**

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

**store**      <index>

If `index` is a logically valid memory location, this will cause the Calculator to store a copy of the current Accumulator value to the specified memory location. If the specified location is invalid, the command has no effect.

**recall**    <index>

If `index` is a logically valid memory location, this will cause the Calculator to push a copy of value at that memory location onto the Calculator’s internal stack. If the specified location is invalid, the command has no effect.

**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.

**Legend:** in the commands above:

<token>                    a string representing either an integer operand or operator  
<index>                    a non-negative integer

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 input file:

```
; CS 2604 Fall 2000
; Minor Project 1
; Sample Input File 0
;
Memory locations: 5
; Evaluate a trivial RPN expression:
enter 4
enter -7
enter +
; ... and show it:
accumulator
; ... and save it:
store 3
; Evaluate a more complex RPN expression:
enter 5
enter 3
stack
enter *
enter 2
enter 4
stack
enter +
stack
enter /
; ... and show it:
accumulator
; ...and save it:
store 1
; ... and display the memory state:
memory
; Put the first result back on the stack:
recall 3
; ... and multiply it by 14:
enter 14
enter *
; ...and show it:
accumulator
; ...and save it:
store 0
; ... and display the memory state:
memory
; ... and clear the calculator
clear
; ...and exit:
exit
```

**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.

```
William D McQuain
CS 2604 RPN Calculator
```

```
-----
Command: enter    4
-----
```

```
Command: enter   -7
-----
```

```
Command: enter    +
-----
```

```
Command: accumulator
        -3
-----
```

```
Command: store    3
-----
```

```
Command: enter    5
-----
```

```
Command: enter    3
-----
```

```
Command: stack
0:  3
1:  5
2: -3
-----
```

```
Command: enter    *
-----
```

```
Command: enter    2
-----
```

```
Command: enter    4
-----
```

```
Command: stack
0:  4
1:  2
2: 15
3: -3
-----
```

```
Command: enter    +
-----
```

```
Command: stack
0:  6
1: 15
2: -3
-----
```

```
Command: enter    /
-----
```

```
Command: accumulator
        2
-----
```

```
Command: store    1
-----
```

```
-----  
Command: memory  
  1:      2  
  3:     -3  
-----  
Command: recall  3  
-----  
Command: enter   14  
-----  
Command: enter   *  
-----  
Command: accumulator  
          -42  
-----  
Command: store   0  
-----  
Command: memory  
  0:     -42  
  1:      2  
  3:     -3  
-----  
Command: clear  
-----  
Command: exit  
-----
```

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.

### Submitting Your Program:

You will submit this assignment to the Curator System (read the *Student Guide*), and it will be graded automatically. Instructions for submitting, and a description of how the grading is done, are contained in the *Student Guide*.

Note that the automated grading system requires that your program be submitted as a single source file. For this project you should develop your implementation using separate compilation, and then manually concatenate the code into a single .cpp file, being careful to get the order right and remove obsolete #include directives.

You will be allowed up to five submissions for this assignment. Use them wisely. Test your program thoroughly before submitting it. If you do not get a perfect score, analyze the problem carefully and test your fix before submitting again. The highest score you achieve will be counted.

The Student Guide can be found at: <http://ei.cs.vt.edu/~eags/Curator.html>

The submission client can be found at: <http://spasm.cs.vt.edu:8080/curator/>

**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.

**Programming Standards:**

While we won't be carefully evaluating your source code on this assignment for programming style, you should still observe good practice. See the Programming Standards page on the course website for specific requirements that should be observed in this course.

**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.**