COMP220 — Project 1

Fall 2017

Abstract

For your first project you'll be writing a modified version of the Reverse Polish Notation (RPN) calculator discussed in Chapter 5 section 2 of the text. It combines elements of exercises 8, 9, and 11 from chapter 6. The end result of your efforts is a fairly robust Rational number class, a class for tokenizing RPN expressions, and a basic command-line interface-based program for evaluating RPN expressions.

CLI-RPNRC

The program you'll be writing is a command-line interface (CLI) based reverse polish notation (RPN) calculator for rational numbers. The user will pass a RPN expression involving rational operands and the operators $+$, $*$, $/$, $-$, $<$, $>$, $<=$, $>=$, $==$, and $!=$. When comparisons are combined with arithmetic, then boolean true should be interpreted as 1/1 and false as 0/1. For example, the expression

20 5 42 57 == + /

should yield 1/4.

Rational Operands

Rational numbers are numerical values which can be written as fractions with integer valued numerators and denominators. You'll be extending the Rational class defined and discussed in chapter 6 section 3 of the text by completing exercise 8 from chapter 6. In addition to the operators listed in exercise 8, you should implement operator<< and operator>> to enable easy streaming I/O with a Rational datum. The streaming I/O operators also get used by gTests for better error reporting. Note that you must implement all the operators specified in exercise 8, even though operators that assign values ($+=$, $-=$, $*=$, $/=$, $++$, $--$) are not valid in our RPN expressions.

CLI Tokenizing

Your program will read the entire expression as a command-line argument, *tokenize* the expression, then carry out the calculation specified by the expression or indicate there is something wrong with the expression. Tokenizing is the act of splitting the sequence of raw characters into a sequence of logical tokens. In this case tokens will be strings that represent operators or rational numbers. The text discusses a class-based solution to tokenizing in section 6.4 (covered in class). Your tokenizer class will be similar, so you should start with that structure. This class should not only tokenize the expression, but provide a means to recognize properly formated tokens. For example, using this class we should be able to recognize if a string can be read as a rational or not. This allows our calculator to avoid errors that would occur when something like `hello world +` is encountered.

Rationals can be expressed as any two integers separated by a /. No spaces should be found on either side of the / character. This means 123/456 is a valid rational expression but 123 /456, 123 / 456, and 123/ 456 are not. The Users are allowed to express negative numerators and denominators and unsimplified rationals, but we will always store them in simplest form and associate negative values with the numerator only.

Operands and operators in full expressions **must** be separated by whitespace but the type and amount of whitespace doesn't matter. This means tokenizing a complete expression means breaking up the string by whitespace and discarding all the whitespace.

Command Line Input

Just to review command line arguments, remember that argc contains the number of arguments (which is always at least one, the name of the program itself), and argv contains an array of C-style strings (which are really just arrays of chars).

The input to the program should be exactly one string, like so:

```
> ./rpncalc "2 2 +"
```

which yields 4. To get a C++ string out of the above input, use the following template:

```
int main(int argc, char* argv[]) {
  if (argc != 2) {
    // handle this case, print message and exit
  }

  std::string input(argv[1]);
  // tokenize and evaluate expression
}
```

If the user provides no arguments (beyond the program itself), your program should print out a helpful message about how it should be used. If the user provides more than the one string argument (such as passing the above input without quotes), you may also print out the helpful message and quit. However, for slightly more challenge you can also handle this input, treating all the arguments together as a single expression.

Error Handling

Once an expression has been tokenized, the process of evaluating the expression occurs. Several things can go wrong during this process:

- Improperly formatted operands and zero denominator operands
- Unknown operators
- Too many or too few operands (you can detect having too many operands when the stack has more than one number on it at the end of the computation)
- Divide by zero

We'll catch all these problems dynamically, while we're evaluating the expression, rather than statically, prior to evaluation as we're tokenizing or parsing the expression. The evaluation procedure should detect the errors as they occur and throw exceptions specific to the errors. Since we don't yet know how to define our own exception classes, use std::runtime_error with a descriptive string passed to the constructor. The program's main procedure, or some other caller to the evaluation procedure, can then catch the error, report the problem to the user, and end the program gracefully.

Logistics

Event	Date
Project Lab Assignment	10/19
Free Lab	10/26
Project Due	11/2 (2 PM)

Grades

Grades are based on the completeness of the program and the design. Whereas previous assignments have been completely (more or less) specified, there is some leeway here for designing the program as you see fit, provided it meets the above expectations. I am even open to changing the above expectations if you can thoroughly justify your reasons for changing them.

Complete programs have a fully functioning calculator program that includes robust error handling. Complete designs have a complete set of documentation and tests. Programs that fail to compile on either the Debug/Release or Test build will not receive not better than a D. Good programs will have complete, elegant designs with code that is "polished" — clean, documented, indented, consistent, and well-tested.

Lab Sessions

During the first lab I suggest starting with the Rational class from the book, writing gtests for all the existing methods, and beginning work on the extensions required for this class. You may want to implement operator== sooner rather than later in order to more easily test the arithmetic operators. Next week there will be no lab assignment; this is time for you to work on the project. I will be in the lab during this time to help answers any questions you may have. You should submit the project as `proj1` using handin by the due date specified above.