CPS222 - DATA STRUCTURES AND ALGORITHMS

Project #2 - Due Wednesday, Feb. 24, at the start of class

Purpose: To give you experience using stacks and working with various forms of expressions

Note: This project must be done in teams of 2. One person doing the project alone will be allowed if the number of students is odd - advance permission from the professor required! It is desirable that students of similar C++ background work together.

Introduction

For this project, you are to complete a program which processes any number of lines of data, each containing an infix expression. Each infix expression will be constructed from one-digit integers, the operators +, -, *, and /, and parentheses - with no intervening spaces. The program will echo the line of data and then convert it to an equivalent postfix expression, printing the postfix on the following line. It will then evaluate the postfix expression and print its value on a third line. Optionally, it will also convert it to prefix and print the prefix form as well. The program will keep reading input lines from standard input until end-of-file (control-D on the console).

Example:

If the input data is:

```
1+2+3+4
3-2*5+4
```

Then the output should be:

```
Infix:   1 + 2 + 3 + 4          <-- note that the program inserts one blank
Postfix: 1 2 + 3 + 4 +          <-- space between tokens for readability
Value:   10
Prefix:  + + + 1 2 3 4

Infix:   3 - 2 * 5 + 4
Postfix: 3 2 5 * - 4 +
Value:   -3
Prefix:  + - 3 * 2 5 4
```

Requirements

Create a directory ~/cps222/project2 on the workstations. The professor has created an Expression class and a main program which you should copy to this directory. Completing the program will require implementing three private methods of class Expression (convertToPostfix(), evaluate(), and convertToPrefix()). You will probably also find it useful to create a function that these methods call (which would be entirely contained in the implementation file - see the Implementation notes below.)

The following command can be used to copy the initial code, assuming you have changed to the correct directory with cd:

```
cp /home/cps222/project2/* .
```

This will copy the following files to your directory:

project2.cc - main program (use as-is)
`expression.h` - declaration for class Expression (use as-is)
`expression.cc` - implementation for class Expression (implement as noted in comments)
`Makefile` - commands for compiling program; please read this to learn how it works;
`proj2.in` - file of test data w/o errors or unary minus (tests minimum requirements & option 3)

`proj2e.in` - file of test data with syntax and divide by zero errors (tests option 1)
`proj2u.in` - file of test data with unary minus (tests option 2)

All your work will be done in `expression.cc`, and will consist of implementing the three methods listed above. You must <u>not</u> change any of the already-written code in `expression.cc`, or any of the code in `expression.h or project2.cc`.

The following commands can be used to compile, link, and run the program. Note that they will work with the files as initially copied, though nothing will be printed for the postfix and prefix forms of the expression, and the value will be 0, if you run the program without adding your code.

```
make
./project2
```

The program will read its input from standard input and write its output to standard output. You can make the program read from a text file of previously-prepared test data by running it as follows:

```
./project2 < whatever file you want to use
```

The test data files you copied (`proj2.in, proj2e.in` and `proj2u.in`) contain test data to thoroughly exercise your program. (It's up to you to examine the output for correctness) The latter two files should only be used if you have implemented error checking or unary minus as the case may be - otherwise the errors / use of unary minus they contain will likely crash your program.

Your program must fulfill the minimum requirements stated below, for which the maximum possible grade is “C”; additional credit may be earned by doing one or more of the options listed. **If you choose to do one or more options, you should first get the minimal requirements working correctly using the test data furnished, and should save a copy of the file containing the working program. Likewise, if you attempt more than one option you should get one working before moving on to the next, and should save a copy of each working version.**

The options do not have to be done in the order listed. In fact, if you do any two (your choice)you can earn an "A level" grade on the project (95), and if you do all three you can earn extra credit.

<u>Minimum</u> (up to 65 points for successful operation and good methodology + quiz: up to 10 = 75 max)

Implement only the methods convertToPostfix() and evaluate() which handle infix to postfix conversion and evaluation of the postfix form. Leave convertToPrefix() unimplemented (just { return string("") } as in the initial file.) (The program will not attempt to output the prefix form if the prefix is "".) For this option, you may assume that no characters other than digits, operators, or parentheses will appear on the input line, and that the infix expressions in the data file are well-formed. You may also assume that the expression can be evaluated - i.e. division by 0 does not occur. Your program must produce correct results when tested with `proj2.in`.

<u>Option 1</u> (up to 10 additional points for successful operation and good methodology)

Modify your program to detect and report syntax errors in the infix expression when converting it to postfix, and division by zero during evaluation. Syntax errors you should detect include occurrence of an operator where an operand was expected or vice versa, an expression beginning or ending with an operator (including a subexpression inside parentheses), unbalanced parentheses, and invalid characters (e.g. spaces, letters, or punctuation marks). When an error is detected, your code should throw the appropriate exception (`SyntaxError` or `DivideByZeroError`).

The code for dealing with a syntax error will look like this (the example assumes a syntax error occurred at position p and the description was "Operator Expected" - your code will need to specify the actual position and error, of course.)

```
        throw SyntaxError(p, "Operator expected");
```

Of course, no further description is needed for a `DivideByZeroError`. The code for doing this will look like this (the example assumes a division by 0 occurred at position p in the postfix - your code will need to specify the actual position, of course.)

```
        throw DivideByZeroError(p);
```

The following are examples of what the output might look like for specific cases. (The carat is written by the main program based on the value of the position parameter to the error constructor (which should be the position of the offending character), and takes into account formatting done by the tester - e.g. the position that was specified when the SyntaxError constructor was called in the first case was 3 and the second case was 2.)

```
Infix:   1 + 2 3
               ^Operator expected

Infix:   3 / 0
Postfix: 3 0 /
             ^Division by Zero
```

You **<u>must</u>** use the error-checking approach that was presented in class. Your error-checking code **<u>must</u>** be integrated smoothly with your conversion and evaluation code - you **<u>must not</u>** attempt to do it as a separate step. Your program must produce correct results when tested with `proj2e.in`.

<u>Option 2</u> (up to 10 additional points for successful operation and good methodology).

In ordinary algebraic notation, the operator '-' actually stands for two different operations: negation and subtraction. We differentiate these roles by context. If a '-' occurs in a context where an operator is expected (e.g. just after an operand or ')') it stands for subtraction; if it occurs in a context where an operand is expected, it stands for negation (and an operand is still needed).

Modify your program to handle the use of '-' for negation, as well as its use for subtraction. Hint: since the context information is lost when you convert to postfix, you will need to replace the use of '-' for negation by some other symbol in the postfix expression - e.g. you might use '#' for this. Note that when a '#' is encountered in the postfix, your evaluate routine should pop just one operand from the stack and negate it.

Important: note that unary operators are right associative and take precedence over binary ones - e.g.

```
        ---2*2+--3
```

should be interpreted as if it were parenthesized as

```
        (-(-(-2)))*2+(-(-3))
```

The following is an example of what the output should look like:

```
Infix:   2 * - 3
Postfix: 2 3 # *
```

Your program must produce correct results when tested with `proj2u.in`.

Option 3 (up to 10 additional points for successful operation and good methodology).

Implement the convertToPrefix() method. This task is most easily done by using a recursive auxiliary function that converts a given string to prefix - initially called with the whole expression, and called recursively with subexpressions. See discussion below. Your program must produce correct results when tested with `proj2.in`.

Implementation Notes

1. Use the STL stack template for the two stacks you need. Note that you will need a stack of char for infix to postfix conversion, and a stack of int for evaluation of the postfix. The mechanics of creating a stack, and the methods available on a stack, were presented in lecture (see online notes and sample programs.)

2. Expressions are represented by strings (objects of class string). The following string methods and operators will probably prove useful.

   For the examples that follow, assume the following declaration:

   ```
   string s = "testing";
   ```

   - operator [] can be used to access a specific character - e.g. `s[1]` is 'e' and `s[1] = 'a'` changes the string to "tasting".
   - length() returns the length of a string - e.g. `s.length()` is 7.
   - operator + can be used to concatenate two strings, or to concatenate a character and a string (in either order). Note that using + for concatenation is similar to the Java usage, but less flexible - only other strings and characters can be used. For example `s + '!'` is `"testing!"`
   - operator += can be used to append to a string - for example `s += '!'`changes s to `"testing!"`. (Note that + creates a new string; += changes an existing string.)
   - substr(position, length) extracts a substring - e.g. `s.substr(4, 2)` is `"in"`.

3. You will probably find it useful to define the auxiliary function `int precedence(char c)`, which returns the precedence value of an operator.

4. For conversion from postfix to prefix, the following approach can be used. The key in each case is to look at the last token in the postfix expression. Some examples will explain how this should work.

   To convert a postfix expression ending in a digit, observe that the digit must be the entire postfix expression. (A valid postfix expression must end in an operator unless it consists of just one operand.) The prefix is the same as the postfix in this case - e.g.

   ```
   Postfix:   3
   Prefix:    3
   ```

   To convert a postfix expression ending in a unary operator (unary '-', perhaps changed to '#'), observe that the operator is the first token in the resultant prefix, and the rest of the prefix is the prefix equivalent of the rest of the postfix (computed by a recursive call) - e.g.

   ```
   Postfix:   7 2 - #
   Prefix:    # < prefix equivalent of 7 2 - >
   ```

If the postfix expression ends in a binary operator (+, -, *, or /), observe that the operator is the first token in the resultant expression, and we must find where the rest of the postfix splits into its two operands. Each of these parts must be converted to prefix by a separate recursive call - e.g.

Postfix: `7 2 - 4 3 2 + + *`
Prefix: `*< prefix equivalent of 7 2 - >< prefix equivalent of 4 3 2 + + >`

The trick, in this case is to find where the two operands split. This can be done by using the following strategy, based on keeping track of how many operands are needed to complete the current subexpression.

- Set operandsNeeded to 1.
- Scan the expression right to left starting with the character just to the left of the operator.
- Whenever you see an operand, decrease operandsNeeded (you've found one).
- Whenever you see a unary operator, leave operandsNeeded alone (the operator needs an operand, but produces a result which counts as an operand.).
- Whenever you see a binary operator, increase operandsNeeded (the operator needs two operands, but produces a result which counts as an operand - a net increase of 1).
- When operandsNeeded drops from 1 to 0, you have found the split point.
-

Example: for the postfix expression `7 2 - 4 3 2 + + *`, if we want to find the dividing line between the two operands of *, we proceed as follows.

Initial	operandsNeeded = 1
Scan rightmost +	operandsNeeded = 2
Scan next +	operandsNeeded = 3
Scan 2	operandsNeeded = 2
Scan 3	operandsNeeded = 1
Scan 4	operandsNeeded drops to 0

The two operands of * are `7 2 -` and `4 3 2 + + *`. (Each must now be converted to prefix recursively.)

5. I will be happy to help you with your program - however, I will not look at a program that is not properly commented, and I will not look at code for an option until you have the minimum requirements working properly.