Project 4

Programming and Algorithms II CSCI 211

Note: You may use the Dstack class you created in Lab 6 for this project.

Objectives

- Learn postfix notation
- Use Dstack to implement an RPN (“Reverse Polish Notation”) calculator
- Practice using cin.peek()
- Practice handling errors

Overview

The standard form of equations, 2 + 4, is called infix because the operator (+ in this example) is “in” the middle of the operands (2 and 4 in this example). This form has the drawback of not being able to control the order in which the operators are applied without using parentheses. For example, if you want 2 + 4 multiplied by 5, you have to write (2+4)*5, or be aware of the order of operations. While this is easy to do in a programming language, it is difficult to do on a standard calculator.

Postfix notation (also called “Reverse Polish Notation” or RPN) is much better suited for calculators. In postfix notation, the operator goes **after** the operand. For example, the equation 2 4 + means 2 plus 4, and the expression 2 4 + 5 * means to first apply + to 2 and 4 and then apply * to the result of 2 + 4 and 5. The result is (2 + 4) * 5 and you did not have to type in the “(” and “)”.

Postfix calculators are easily implemented using a stack data structure. When the user enters an operand, it is pushed onto the stack. When the user enters a binary operator, the two operands on top of the stack are removed (via pop), the operator is applied to them, and the result is pushed back onto the stack.

If at the end of the input, there is one and only one operand on the stack, then the input was a valid expression and the value on the stack is the result of evaluating that expression.

Implement an RPN calculator that can evaluate any legal postfix expression, and catch all illegal input. As soon as illegal input is encountered, print an error message (see below) and call exit(1).

Program Requirements

Implementation

Recommended: Start by coping your dstack.cpp and dstack.h files from Lab 6 into your Project 4 directory.

Create file calc.cpp which contains function main().

Keep the functionality of the calculator completely separate from the Dstack class. Implement the parts of the program that have to do with the calculator (including the error messages) in calc.cpp.

Input

Your program must read a single equation from standard input. The operators are + - * / and ^ (where ^ is power, e.g., 2 4 ^ is 2 raised to the power of 4). Operands can be any floating point number, for example 4, 4.2, .2, 0.2, 0000.233. You can assume that each operand will fit into a standard "double" type variable (i.e., the input will not contain an operand with 200 digits).

Your program must ignore all white space (spaces, tabs, newlines). If two numbers are in a row, there will be whitespace between them. For example, 42 is the number forty-two, and 4 2 are the numbers four and two.

There do not have to be spaces between operands and operators. For example, 10 10+ is a legal equation.

There do not have to be spaces between two operators. For example, 10 10 10++ is a legal equation.

All operands are positive and do not specify a sign.

Program Output

If the input is a correct post-fix expression, your program must print the result.

For example, if the input is 10 10+, your program must output 20 followed by a newline, and nothing else.

Arithmetic Overflow

You may assume that all intermediate results and the final result are small enough to fit into a standard "double" type variable.

Errors and Error Messages

All errors must be detected in calc.cpp. Recall that the Dstack::pop() method must return false when the stack is empty.

All errors must be output in calc.cpp (you cannot output an error message in any of the stack functions).

If the input is not a valid post-fix expression, your program should output the following to standard error (cerr): "Error: Invalid expression"…followed by a newline (endl) and then terminate the program. Programs can be terminated by calling the exit() function, but you may only call exit() from calc.cpp.

Your program must check for all possible errors (except numbers that are too large). This includes all illegal mathematical operations such as divide by zero. The power function is especially problematic. Consider it closely.

Specific "Invalid Expression" cases you must handle to pass all tests:

- Invalid operand values like 1.2.3 (to detect this, peek to see if next char is a '.' after each cin of an operand).

Tips & Tricks

Here is the main RPN algorithm in pseudocode:

```
while (not the end of the input)
   If the next input is a number
      read it and push it on the stack
      elseIf the next input is an operator
         read it
         pop 2 operands off of the stack
         apply the operator
         push the result onto the stack
   When you reach the end of the input:
      if there is one number on the stack, print it else error
```

You cannot just use 'getline()' or 'cin >> <variable>' for this program because you do not always know if an operator or an operand (a number) will be the next thing in the input stream. You will need to use cin.peek() to look at the next character in the input. cin.peek() returns the next character without actually reading it. If it is whitespace (space, tab, newline) you can skip it using cin.ignore(). If it is a legal operator, you can read the single character using cin >> char variable. If the next character is a digit, you can use cin >> double variable to read the entire number. cin.peek() returns EOF when there are no more characters in the input.

Tip: The cctype library has many useful functions to identify punctuation, numbers, letters, etc. For example, you may use cin.peek() in conjunction with the library function isspace() to determine if the next character is a space.

General Requirements

1. Comment your code:
   - The top of each file has our usual comment block with your name, the file name, and your Blackboard user id.
   - Above each method and function implementation is a descriptive comment block.
   - There are in-line comments in your function and method implementation code every several lines of code.
2. Free all dynamically allocated (heap) memory prior to program exit.
3. No function or method contains more than 30 to 40 lines, including white-space and comments.
4. Avoid code duplication.
5. Use a **single** return statement for all functions & methods that return a value.
6. Follow these variable naming rules: Local variables are all lowercase, with underbars between words, e.g. my_array. Class member variables begin with m_ and are all lowercase, with underbars between words, e.g. m_count. Class names and Struct names are "camel case", starting with an uppercase letter, e.g. IntStack. Constants are ALL CAPS.
7. Use descriptive names for all classes, structs, functions, methods, parameters, variables and types.
8. Do not declare any non-constant **global** variables.
9. Do not submit commented-out code.
10. Do not submit unused / un-called code.
11. Code blocks are properly aligned.
12. No more than three submissions to TurnIn (use run_tests to validate locally). Come see me if you are having trouble with your TurnIn submissions (e.g. all local tests are passing but not all TurnIn tests are not passing).

Testing

Use the tests and Makefile for Lab 4 from the starter pack on Blackboard Learn, using the usual "run_tests" script.

<..continued on next page…>

Submission

Just prior to final submission, be sure to re-check the grading criteria in Blackboard under Course Content > Projects > Project 4, and re-read the requirements listed above.

Submit files calc.cpp dstack.h dstack.cpp to Turn-in.

Each student will complete and submit this assignment individually. Make steady progress, work daily.