Infix Expression Evaluation

Table of Contents

Purposes of this assignment

This project seeks to

- give you experience using multiple stacks in C++ STL.
- teach you one of the fundamental algorithms of computer science
- give you more experience with Visual Studio and debugging – it is very important!

Files provided

- infix.pdf – this file
- infix.cpp – a skeleton code to begin with
- infixDriver.cpp, infixallDriver.cpp – a driver to test infix.cpp and infixall.cpp
- **infixx.exe, infixall.exe** – a solution for pc
- **infixx, infixall** – a solution for mac

Overview

This pset evaluates an expression represented by a string. Expression can contain parentheses; you can assume parentheses are well-matched. For simplicity, you can assume only binary operations allowed are +, -, *, and /.

- **Infix notation**: Operators are written between the operands they operate on, e.g. 3 + 4 .
- **Prefix notation**: Operators are written before the operands, e.g. + 3 4
- **Postfix notation**: Operators are written after operands, e.g. 3 4 +

Infix notation is how expressions are written and recognized by humans and, generally, input to programs. Infix expressions are harder for computers to evaluate because of the

addional work needed to decide precedence. However, there is a very well-known algorithm suggested by **Edgar Dijkstra** for evaluating an infix notation using two stacks.

Your assignment is to debug and add more functionality in the infix expression evaluation code provided. The user types in either an expression or an assignment, and the code computes and displays the result. Infix expressions are harder for computers to evaluate because of the addional work needed to decide precedence. His idea was using two stacks instead of one, one for operands and one for operators.

Examples:

For example, an arithmetic expression consists of:

```
1 - 3
1 - ( 2 * 5 )

1 + (234 – 5)
123 - ( 21 * 5 )

2 * (( 3 - 7 ) + 46)
(12 + (4 * 100)) - (2* 5)
(((2 + 4) * 100) - ( 2 *5 )) + 1

12 + 4 * 100 - 2* 5
(2 + 4) * 100 - 2 *5 + 1
```

- Numbers:
  All numbers will be represented internally by integer values. The division or power will be carried out as an integer division.
- Parentheses:
  They have their usual meaning. Only **(** and **)** will be used; don't use **{ } [ ]**.
- Operators:
  - \+ for addition; used only as a binary operator.
  - \- for subtraction; used only as a binary operator.
  - * for multiplication.
  - / for division.
  - ^ for power. (It handles in Task B – later part of this pset)

Operator stack and operand stack

This program we are about to code takes an infix expression in a string and returns its result. Each operator in the expression is represented as a single char and each operand is represented as an integer value. The numbers can be a multiple digit.

During this process, it uses two stacks, one for operators and the other for operands.

- Operand stack: This stack will be used to keep track of numbers.
- Operator stack: This stack will be used to keep operations (+, -, *, /, ^)

You will use the stack class provided in C++/STL.

Task A: infix.cpp

We are going to approach this problem with two tasks or steps.

Implement the algorithm as simple as possible using two stacks. For simplicity of coding, we assume that the expression is **fully parenthesized**. This approach does not require the precedence of operators during implementation, but the user must put all the parenthesis necessary to have it evaluated correctly. For example, the following expression should work properly and produce the result correctly with your complete code:

```
1 - 3 = -2
1 - ( 2 * 5 ) = -9
( ( 3 - 1 ) * 5 ) - 4 = 6

2*((21-6)/5) = 6
(12 + (4 * 100)) - (2* 5) = 402
(((2 + 4) * 100) - ( 2 *5 )) + 1 = 591
```

The skeleton code, infix.cpp, provided for this task may works partially, and you are asked to complete the code by implementing the following items:

1. The expression may have with/without some spaces between operator and operand.
2. The values (operands) may be multiple-digit integers.

The algorithm is roughly as follows.

1 While there are still tokens to be read in,
  1.1 Get the next token.
  1.2 If the token is:
    1.2.1 A space: ignore it
    1.2.2 A left brace: ignore it
    1.2.3 A number:
      1.2.3.1 read the number (it could be a multiple digit.)
      1.2.3.2 push it onto the value stack
    1.2.4 A right parenthesis:
      1.2.4.1 Pop the operator from the operator stack.
      1.2.4.2 Pop the value stack twice, getting two operands.
      1.2.4.3 Apply the operator to the operands, in the correct order.
      1.2.4.4 Push the result onto the value stack.
    1.2.5 An operator
      1.2.5.1 Push the operator to the operator stack
2 (The whole expression has been parsed at this point.
  Apply remaining operators in the op stack to remaining values in the value stack)
  While the operator stack is not empty,
  2.1 Pop the operator from the operator stack.
  2.2 Pop the value stack twice, getting two operands.
  2.3 Apply the operator to the operands, in the correct order.
  2.4 Push the result onto the value stack.
3 (At this point the operator stack should be empty, and the value stack should have only one value in it, which is the result.)
  Return the top item in the value stack.

Task B: infixall.cpp

Once you finish all the functionality in Task A, you make a copy of **infix.cpp** into **infixall.cpp**.

Step 1. Add the exponential operator ^. For example, 2^3 returns 8.

Step 2. Now, you are asked to improve the code **by removing the limitation so-called "fully parenthesized"** in the given infix expression.

For example, the following expression should work properly and produce the result correctly with your complete code:

```
( 1 + 2 ) - 4 = -1
1 - ( 2 * 5 ) = -9
(3 - 1) * 5 - 4 = 6
(1 + 2^ 3 ) * 5 - 4 = 41

1 + (24 * 5) = 121

2*(21-6)/5 = 6
2 * (3 - 7 + 46) = 84
12 + 4 * 100 - 2* 5 = 402
(2 + 4) * 100 - 2 *5 + 1= 591
```

The algorithm is roughly as follows. Note that no error checking is done explicitly; you should add that yourself.

1 While there are still tokens to be read in,
  1.1 Get the next token.
  1.2 If the token is:
    1.2.1 A space: ignore it
    1.2.2 A left brace: push it onto the operator stack.
    1.2.3 A number:
      1.2.3.1 read the number (it could be a multiple digit.)
      1.2.3.2 push it onto the value stack
    1.2.4 A right parenthesis:
      1.2.4.1 While the item on top of the operator stack is not a left brace,
        1.2.4.1.1 Pop the operator from the operator stack.
        1.2.4.1.2 Pop the value stack twice, getting two operands.
        1.2.4.1.3 Apply the operator to the operands, in the correct order.
        1.2.4.1.4 Push the result onto the value stack.
      1.2.4.2 Pop the left brace from the operator stack and discard it.
    1.2.5 An operator (let's call it **thisOp**)
      1.2.5.1 While the operator stack is not empty, and the top item on the operator stack has the same or greater precedence as thisOp,
        1.2.5.1.1 Pop the operator from the operator stack
        1.2.5.1.2 Pop the value stack twice, getting two values
        1.2.5.1.3 Apply the operator to two values in the correct order
        1.2.5.1.4 Push the result on the value stack
      1.2.5.2 Push the operator (**thisOp**) onto the operator stack
2 (The whole expression has been parsed at this point.
  Apply remaining operators in the op stack to remaining values in the value stack)
  While the operator stack is not empty,

2.1 Pop the operator from the operator stack.
2.2 Pop the value stack twice, getting two values.
2.3 Apply the operator to two values, in the correct order.
2.4 Push the result onto the value stack.

3 (At this point the operator stack should be empty, and the value stack should have only one value in it, which is the result.)
Return the top item in the value stack.

Using infixDriver.cpp, infixallDriver.cpp files

These driver files are provided to make your testing easy. The infixDriver, for example, has four steps of testing as shown below. However, your skeleton code, **infix.cpp**, will work only for Step 1 and the rest of Steps are commented out. Your complete code should work with the rest of Steps at least.

```
int main() {
	setvbuf(stdout, nullptr, _IONBF, 0);  // prevents the output from buffered on console.

	string exp;
	while (true) {
		cout << "\nExample: 2 * ((34 - 4) * 2)";
		cout << "\nEnter an infix expression(fully parenthesized, q to quit): \n";
		getline(cin, exp);
		if (exp[0] == 'q') break;
		cout << exp << " = " << evaluate(exp) << endl;
	};

	cout << "Step 1: Simple cases" << endl;
	cout << "   [-2] = " << evaluate("1 - 3") << endl;
	cout << "   [-9] = " << evaluate("1 - ( 2 * 5 )") << endl;
	cout << "    [6] = " << evaluate("( ( 3 - 1 ) * 5 ) - 4") << endl;
	/*************************************************************************
	cout << "Step 2: Multi-digits operand(or value)" << endl;
	cout << "  [230] = " << evaluate("1 + ( 234 - 5 )") << endl;
	cout << "   [18] = " << evaluate("123 - ( 21 * 5 )") << endl;
	cout << "   [60] = " << evaluate("( 12 - 8 ) * ( 45 / 3 )") << endl;

	cout << "Step 3: Space toleration" << endl;
	cout << " [1000] = " << evaluate("(  1 + 249 ) *4") << endl;
	cout << "  [120] = " << evaluate("(1+23)*5") << endl;
	cout << "  [121] = " << evaluate("1 + (24 * 5)") << endl;

	cout << "Step 4: Final Testing" << endl;
	cout << "    [6] = " << evaluate("2*((21-6)/5)") << endl;
	cout << "   [84] = " << evaluate("2 * (( 3 - 7 ) + 46)") << endl;
	cout << "  [402] = " << evaluate("(12 + (4 * 100)) - (2* 5)") << endl;
	cout << "  [591] = " << evaluate("(((2 + 4) * 100) - ( 2 *5 )) + 1") << endl;
	*************************************************************************/

	cout << "Happy Coding~~";
	return EXIT_SUCCESS;
}
```

Submitting your solution

- **On my honour, I pledge that I have neither received nor provided improper assistance in the completion of this assignment.**
  Signed: _____ **Section:** _____ **Student Number:** _____

- Make sure your code **compiles** and **runs** right before you submit it. Don't make "a tiny last-minute change" and assume your code still compiles. You will not receive sympathy for code that "almost" works.
- If you only manage to work out the Project problem partially before the deadline, you still need to turn it in. However, don't turn it in if it does not compile and run.
- Place your source files in the folder you and I are sharing.
- After submitting, if you realize one of your programs is flawed, you may fix it and submit again as long as it is **before the deadline**. You will have to resubmit any related files together, even if you only change one. You may submit as often as you like. **Only the last version** you submit before the deadline will be graded.

Files to submit

- infix.cpp, infixall.cpp

Due and Grade points

- Due: 11:55 pm, Oct. 9
- 5 points