Infix & Postfix 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
- postfixDriver.cpp – Do not change this file. Do not submit it either.
- postfix.cpp – a skeleton code to begin with
- postfix.exe – an executable as a reference
- infix.cpp – a skeleton code to begin with
- infixDriver.cpp - a driver to test infix.cpp
- infixallDriver.cpp – a driver to test infixall.cpp
- infixallx.exe - an executable as a reference

Overview

The first part of this pset is to evaluate a postfix expression to an infix expression.

The second part of this pset is to evaluate an infix arithmetic expression represented by a string and produce a value. This 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 +

Step 1: postfix.cpp & postfixDriver.cpp

The first part of this assignment is to create a function called **`evaluate()`** that takes an postfix expression and produces a fully parenthesized infix expression.

- For simplicity of coding, the postfix expression consists of single character operands and operators only and may have spaces.
- Implement **`printStack(stack<string> s)`** that prints the stack elements from bottom. It is useful for your debugging for this step. This function is also required for the latter part of this assignment.
- Set **`assert()`** function properly in **`evaluate()`.**

A detail algorithm for this assignment is described in the **StackApps** lecture video.

A sample run:

```
PS C:\GitHub\nowicx\src> g++ postfixDriver.cpp postfix.cpp; ./a
Example: a b +, 2 3 4 * +, ab/5+
Enter a postfix expression(q to quit): q
postx: 2 3 4 * +
infix: (2 + (3 * 4))
postx: a b * 5 +
infix: ((a * b) + 5)
postx: 1 2 + 7 *
infix: ((1 + 2) * 7)
postx: a b c - d + / e a - * c *
infix: (((a / ((b - c) + d)) * (e - a)) * c)
Happy Coding~~
PS C:\GitHub\nowicx\src>
```

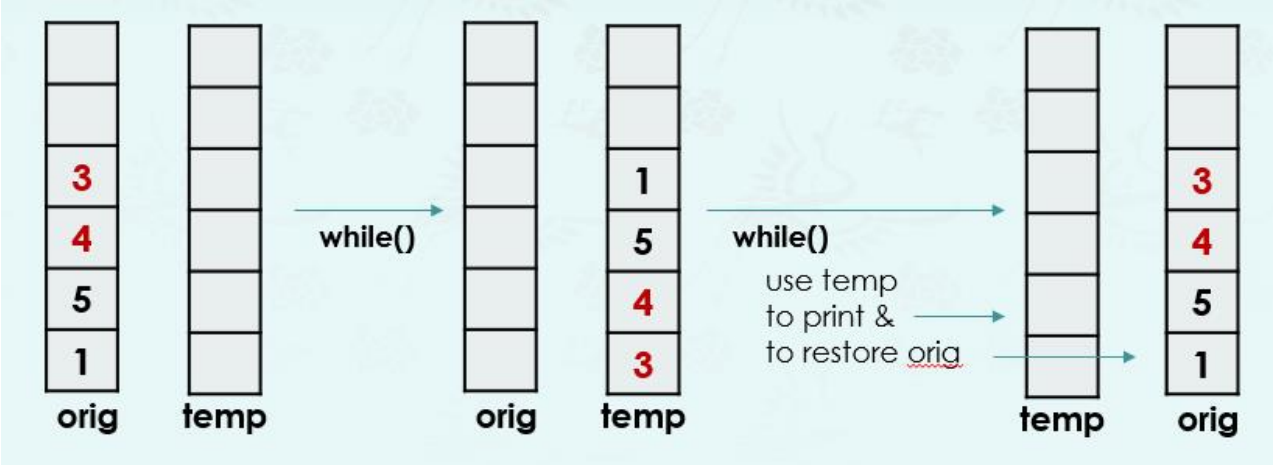
printStack()

While working on this pset, you want to know the current status of the stacks. You are required to write a help function that prints contents of a stack from the bottom to top. Since you have implemented this function using recursion algorithm before, you will implement it using iteration this time.

Implement **printStack(stack<string> s)** for this step. You may use the stack class provided in C++/STL.

Algorithm:

- Given a stack called **orig**.
- Create an empty stack called **temp.**
- While **orig** is not empty,
  - Top/Pop push an item from **orig** to **temp**.
- While **temp** is not empty,
  - Top/Pop an item from **temp**, print it and push it **orig**.



.

Step 2: Evaluate Infix Arithmetic Expressions

The second part of your assignment is to 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 additional work needed to decide precedence. There is a very well-known algorithm suggested by **Edgar Dijkstra** for evaluating an infix notation using two stacks. His idea was using **two stacks** instead of one, one for operands and one for operators.

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.

Modify the printStack(stack<string> and implement both **printStack(stack<int> va)** and **printStack(stack<char> op)** as well. Stay tuned since we are going to use C++ Template to make them into one, a generic function.

Infix Arithmetic Expression Examples:

For example, an infix arithmetic expression consists of:

```
1 - 3
1 - ( 2 * 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 – latter part of this pset)

Coding:

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. Write a help function, **printStack(stack<char>)** and **printStack(stack<int>)** that print a stack from bottom to top.
2. Allow multiple digits of integer values (operands).
3. Fix a bug in **compute()** function.
4. Set **`assert()`** function properly in **`evaluate()`.**
5. Complete the code in "Your code here". Most of them exist in **evaluate().**

For simplicity of coding, we assume that the expression is **fully parenthesized except the first and last parenthesis**. 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
( ( 3 - 1 ) * 5 ) – 4 = 6

1 + ( 234 - 5 ) = 230
123 - ( 21 * 5 ) = 18
( 12 - 8 ) * ( 45 / 3 ) = 60

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

The algorithm to evaluate an infix expression 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.

Step 3: infixall.cpp

Once you finish all the functionality in Step 2, you make a copy of **infix.cpp** into **infixall.cpp**. Code the following specifications.

- Add the exponential operator ^. For example, 2^3 returns 8.
- Now, you are asked to improve the code **by removing the limitation so-called "fully parenthesized"** in the given infix expression.
- Use `Template` such that we can have just one version of printStack() instead of two depending on input data types, `int` or `char`.

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 Function Templates -

In this step, replace two printStack() functions into one using function template in C++.

Two printStack() functions are exactly the same except its argument type. There is a wonderful technology that turns two function into one. It is called a generic programming and C++ supports this through function templates. For details, you may refer to here where the following section is excerpted from.

Function templates are special functions that can operate with **generic types**. This allows us to create a function template whose functionality can be adapted to **more than one type** or class without repeating the entire code for each type.

In C++ this can be achieved using **template parameters**. A template parameter is a special kind of parameter that can be used to pass a type as argument: just like regular function parameters can be used to pass values to a function, template parameters allow to pass also types to a function. These function templates can use these parameters as if they were any other regular type.

The format for declaring function templates with type parameters is:

```
template <class identifier> function_declaration;
template <typename identifier> function_declaration;
```

The only difference between both prototypes is the use of either the keyword `class` or the keyword `typename`. Its use is indistinct, since both expressions have exactly the same meaning and behave exactly the same way.

For example, to create a template function that returns the greater one of two objects we could use:

```
1 template <typename myType>
2 myType GetMax (myType a, myType b) {
3  return (a>b?a:b);
4 }
```

Here we have created a template function with `myType` as its template parameter. This template parameter represents a type that has not yet been specified, but that can be used in the template function as if it were a regular type. As you can see, the function template `GetMax` returns the greater of two parameters of this still-undefined type.
http://www.cplusplus.com/doc/oldtutorial/templates/

Using ~Driver.cpp files

Each driver file is provided to make your testing easy. Make sure that your ~.cpp file works with its own driver file before submitting your code.

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

Do not submit driver files. Each file in the following should work its own driver without modifications, respectively.

- postfix.cpp
- infix.cpp
- infixall.cpp

Due and Grade points

- Due: 11:55 pm
- Step 1: Finish this first to get points for Step 2 & 3
- Step 2: No credit if Step 1 is not complete
- Step 3: No credit if Step 1 & 2 are not complete