PSet: Infix & Postfix Evaluations

Table of Contents

Purposes of this assignment 1
Files provided 1
Overview 2
Step 1: postfix.cpp & postfixDriver.cpp 2
printStack() 3
Step 2: Evaluate Infix Arithmetic Expressions 4
Operator stack and operand stack 4
Infix Arithmetic Expression Examples: 5
Coding: 5
Step 3: infixall.cpp 6
Step 4: Essay on Using MS VS or XCode 8
Submitting your solution 8
Files to submit 9
Due and Grade points 9

Purposes of this assignment

This project seeks to

- Teach you some fundamental algorithms such as postfix, infix, stack, etc.
- Give you experience using C++ template and multiple stacks in C++ STL.
- Give you experience with Visual Studio and debugging – it is very important!

Files provided

- infixpostfix.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
  infixall.cpp – remove this file – a dummy file, but use your infix.cpp instead.
  Copy your infix.cpp into infixall.cpp to begin with, once you finish infix.cpp.
- infixDriver.cpp - Do not change this file. Do not submit it either.
- infixallDriver.cpp – Do not change this file. Do not submit it either.
- infixallx.exe - an executable as a reference

Mac Users: Use **Wine or WineBottler** to run Windows executable on your Mac.

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

This step consists of two parts.

The first part is to implement functions that takes a postfix expression and produces a fully parenthesized infix expression. You must complete the following functions to make it work.

```
string evaluate(string tokens);
void printStack(stack<T> orig);  // a helper function, use function template
```

Once you complete the first part, you implement the following functions as shown below:

```
bool is_numeric(string tokens);
double evaluate_numeric(string tokens);
```

The **`is_numeric()`** returns true if the postfix expression is numerically expressed. If the given postfix is numerically expressed using just numbers and operators, then **`evaluate_numeric()`** is invoked to evaluate the expression numerically. The **`evaluate_numeric()`** would be very similar to **`evaulate()`** code. The difference is to use **`stack<double>`** instead of stack<string> since the arithmetic operation will be performed and its results (numerical values) will be saved and retrieved in stack.

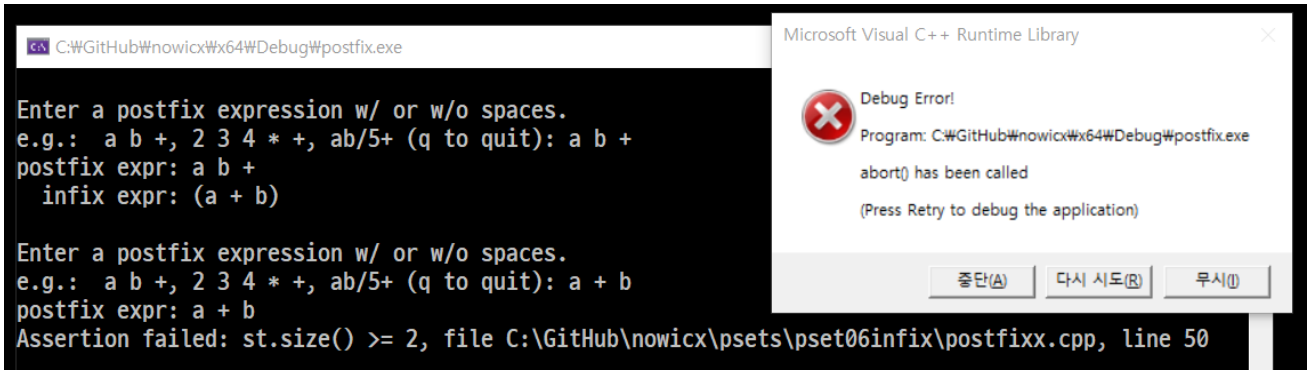
Follow the following instructions and comments in the skeleton code provided in the source file.

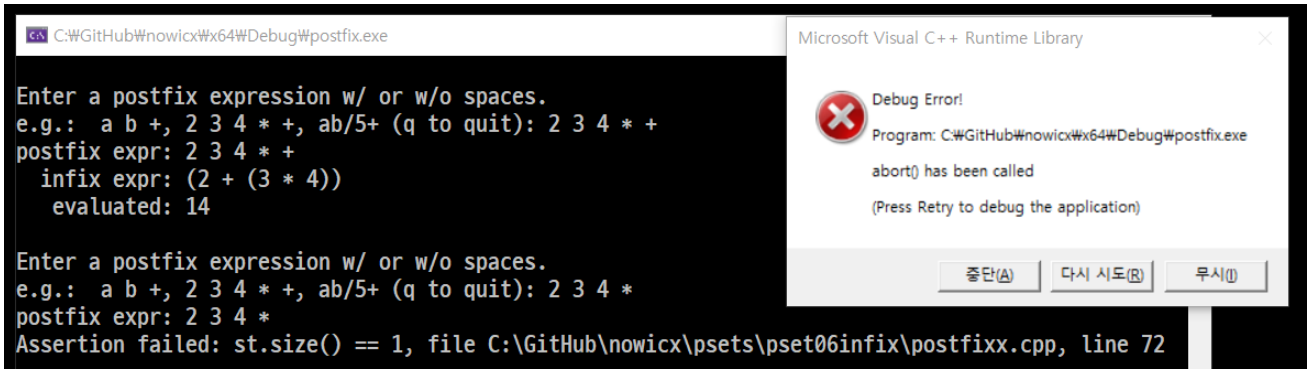
- The function **`evaluate()`** returns a fully parenthesized infix expression in string after postfix evaluation. It uses **`stack<string>`** during the evaluation process.
- The function **`evaluate_numeric()`** returns a numeric value after evaluating the postfix expression. It uses **`stack<double>`** during the evaluation process.
- The function **`is_numeric()`** returns true if the postfix expression is expressed numerically, not symbolically. .

- Using C++ STL stack class, implement **`printStack(stack<T> st).`** It prints the stack elements starting from bottom. It is useful for your debugging.
- For simplicity of coding, the postfix expression consists of **single character operands and operators only** and may have some spaces.
- Use **`assert()`** macro a couple of places properly to check the size of stack in **`evaluate().`** If the condition is true, the program continues normally and if the condition is false, the program is terminated and an error message is displayed. A assert has the following general syntax.

```
assert (bool_conditional_expression)
```

Sample runs:





printStack()

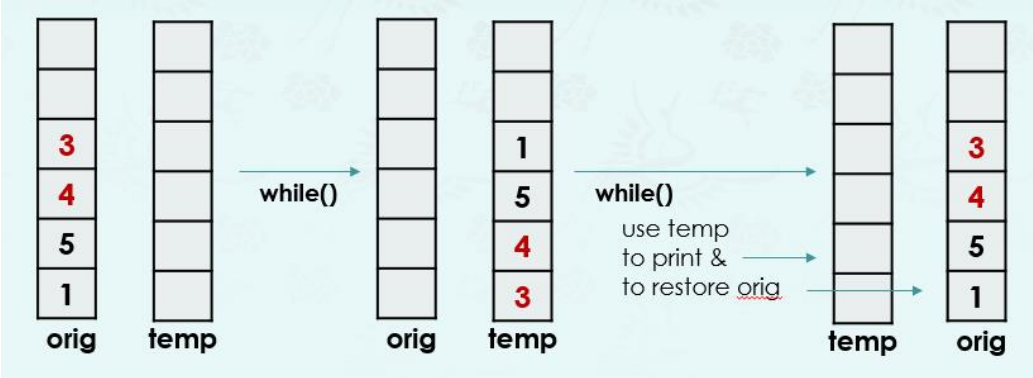
While working on this pset, you want to know the current status of the stacks. You are required to write a help function that prints contents of a stack from the bottom to top. Since you have implemented this function using recursion algorithm before, you will implement it using iteration this time.

Implement **printStack(stack<T> st)** for this step as shown below. You may use the stack class provided in C++/STL. You will implement a recursion version of this function later of this pset.

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**.

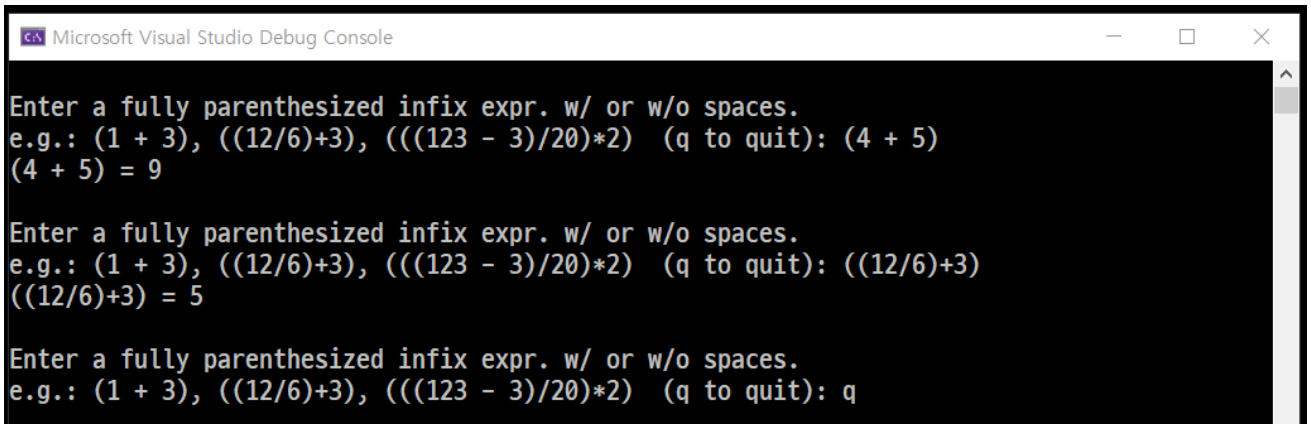


.

Step 2: Evaluate Infix Arithmetic Expressions

The second part is to evaluate an infix expression. The user may enter in either an infix expression, 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.

Sample run:



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 (+, -, *, /)

Use the printStack(stack<T> that you implement previously. This time, this function will work for both stack<int> and stack<char> data types.

Infix Arithmetic Expression Examples:

For example, an infix arithmetic expression consists of:

```
1 - 3
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 power is 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.

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. Copy a help function, **printStack(stack<T>)** from postfix.cpp.
2. Allow multiple digits of integer values (operands).
3. Fix a bug in **compute()** function.
4. Use **`assert()`** macro in a couple of place to check the stack size properly.
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
```

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, 1+2^3+2 returns 11.
- Improve the code **by removing the limitation so-called "fully parenthesized"** in the given infix expression.
- Add **precedence()** function returning 0, 1, 2 ... depending on an operator.
- **Using recursion**, rewrite `Template` version of printStack() such that it uses the system stack instead of defining your own stack.

  The idea is to top/pop the element of **the user's stack** and call the recursive function printStack() until it reaches the bottom of the stack. During recursive calls, top/popped elements saved in **the system stack** automatically. Once the user's stack becomes empty, start printing the element which was popped last. Thus elements will be printed from bottom to top. Then push back the element that was printed, this will preserve the order of the elements in the stack. Its pseudo code is listed below:

```
void printStack(stack<T> st)
   if stack is empty, return
   get the item at top from stack and remove it from the original stack
   printStack()      // recursive calls until it reaches the bottom of stack
   print the item    // recursive calls ended, recover item from system stack
   push the item     // reconstruct the original stack
```

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

```
(1 + 2^ 3 ) * 5-1 = 44
2*(21-6)/5 = 6
2 * (3 - 7 + 46) = 84
(2 + 4) * 100 - 2 *5 + 1= 591
```

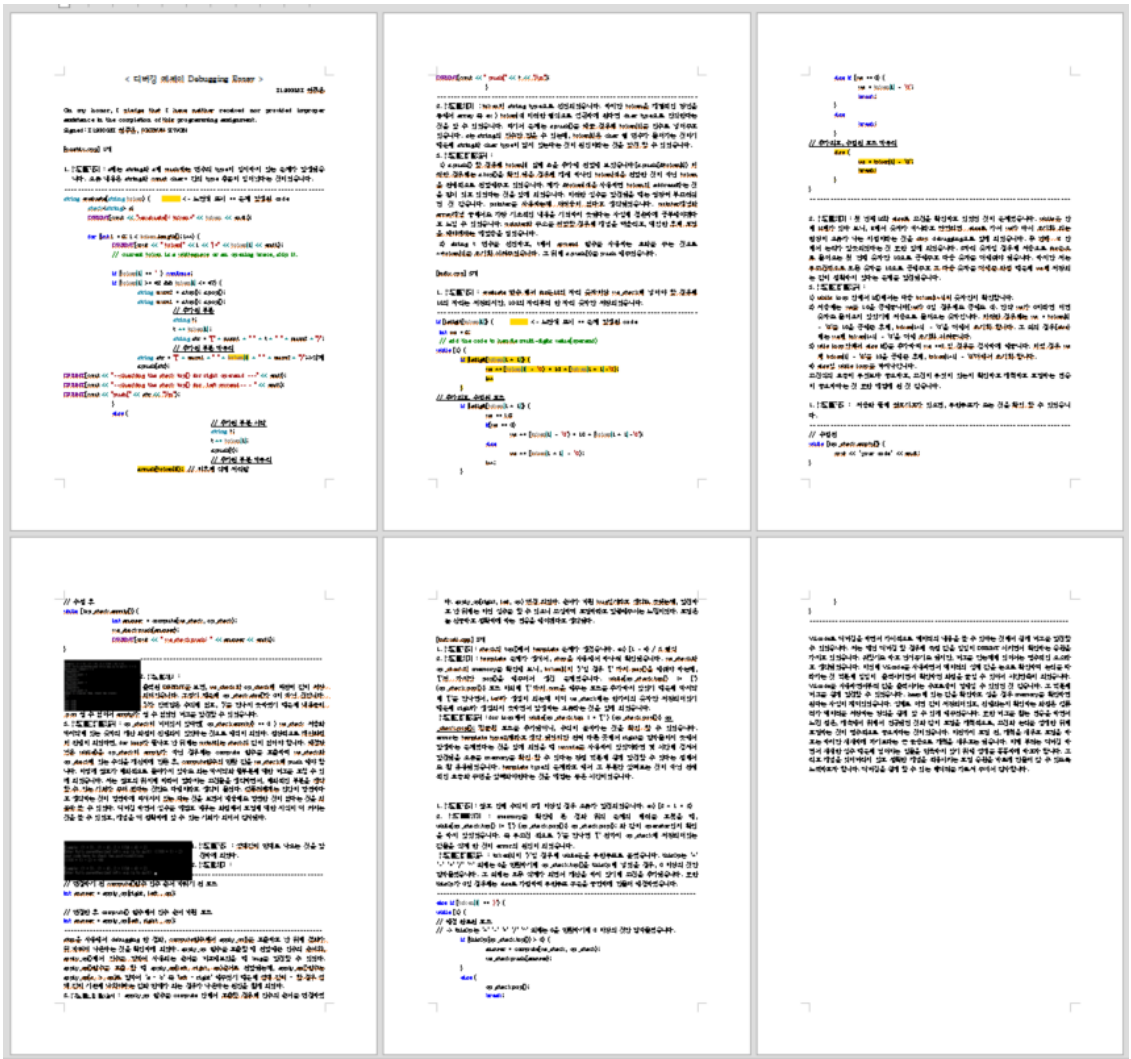
The algorithm is roughly as follows. Note that no error checking is done explicitly; you should add that yourself.

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.

Step 4: Essay on Using MS VS or XCode

Write **at least three pages** long essay (excluding cover-page) about your experiences and your findings while using either VS or XCode.

- The purpose of this essay is to show that you know how to use the tool and **how you applied it for this coding**.
- Your writing may include screen captures and code, but they should **not be over 50%.**
- Only a few good papers may get 1.5 points.
  Expect 0.5 ~ 1.0 point if the minimum requirement is fulfilled.
- An outline of an essay is shown below:



Submitting your solution

- **On my honour, I pledge that I have neither received nor provided improper assistance in the completion of this assignment.**
  Signed: _____ **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
- Grade:
  - Step 1: 1 point
  - Step 2: 1 point
  - Step 3: 1 point
  - Step 4: 0.25 ~ 1.0 point
    (A few exceptionally good essays may get up to 1.5 points)