Object Oriented Programming Techniques

Lab 12 – May 3, 2024

Exercise 1

Use singly linked list to implement a stack of `char` values. Use smart pointers instead of raw pointers in your code. Implement the following functions in your `stack` class.

- `push()` that pushes a value onto the stack
- `pop()` that pops a value off the stack
- `top()` that returns the top value of the stack without popping the stack
- `empty()` that determines if the stack is empty
- `print()` that prints the stack

Also implement the default constructor that initializes an empty stack and a destructor to clean up the list.

Exercise 2

Evaluating postfix expression: Humans generally write expressions like `3 + 4` and `7 / 9` in which the operator (`+` or `/` here) is written between its operands—this is called *infix* notation. Computers "prefer" *postfix* notation in which the operator is written to the right of its two operands. The preceding infix expressions would appear in postfix notation as `3 4 +` and `7 9 /`, respectively.

Write a program that evaluates a valid postfix expression such as `6 2 + 5 * 8 4 / -`

The following arithmetic operations are allowed in an expression:

`+`	addition	`*`	multiplication
`-`	subtraction	`/`	division
`^`	exponentiation	`%`	remainder

The program should read a postfix expression consisting of digits and operators into a `string`. Using `stack` implemented earlier, the program should scan the expression and evaluate it. The algorithm is as follows:

1. While you have not reached the end of the string, read the expression from left to right.
   - If the current character is a digit,
     - Push its integer value onto the stack (the integer value of a digit character is its value in the computer's character set minus the value of '0' in the computer's character set).
   - Otherwise, if the current character is an operator,
     - Pop the two top elements of the stack into variables x and y.
     - Calculate y operator x.
     - Push the result of the calculation onto the stack.
2. When you reach the end of the string, pop the top value of the stack. This is the result of the postfix expression.

[Note: In Step 2 above, if the operator is `'/'`, the top of the stack is `2` and the next element in the stack is `8`, then pop `2` into `x`, pop `8` into `y`, evaluate `8 / 2` and push the result, `4`, back onto the stack. This note also applies to operator `'-'`.]

Implement the following functions to complete the exercise:

a) `evaluatePostfixExpression()` that evaluates the postfix expression
b) `calculate()` that evaluates the expression `op1 operator op2`

Exercise 3

Infix to Postfix notation: Write a program that converts an ordinary infix arithmetic expression (assume a valid expression is entered) with single-digit integers such as

```
(6 + 2) * 5 - 8 / 4
```

to a postfix expression. The postfix version of the preceding infix expression is

```
6 2 + 5 * 8 4 / -
```

The program should read the expression into string **`infix`** and use stack functions implemented in previous exercise to help create the postfix expression in string **`postfix`**. The algorithm for creating a postfix expression is as follows:

1. If the first and last characters in **`infix`** are not **`'('`** and **`')'`** respectively then append them to **`infix`**.
2. While the stack is not empty, read **`infix`** from left to right and do the following:
   - If the current character in **`infix`** is a digit, copy it to the next element of **`postfix`**.
   - If the current character in **`infix`** is a left parenthesis, push it onto the stack.
   - If the current character in **`infix`** is an operator,
     - Pop operators (if there are any) at the top of the stack while they have equal or higher precedence than the current operator, and insert the popped operators in **`postfix`**.
     - Push the current character in **`infix`** onto the stack.
   - If the current character in **`infix`** is a right parenthesis
     - Pop operators from the top of the stack and insert them in **`postfix`** until a left parenthesis is at the top of the stack.
     - Pop (and discard) the left parenthesis from the stack.

Implement the following functions to complete the exercise.

a) **`convertToPostfix()`** that converts the infix expression to postfix notation
b) **`isOperator()`** that determines whether **`c`** is an operator
c) **`precedence()`** that determines whether the precedence of **`operator1`** is greater than or equal to the precedence of **`operator2`**, and, if so, returns **`true`**