

Homework #2

Oct. 16th (Wed.) ~ Nov. 1st (Fri.) 23:59

* Be sure to read the note on the last page.

P1 (25pt). Complete the linked list-based Stack class in HW2_Stack.h to satisfy the following conditions by referring to the given HW2_Stack.h. Each element of the stack is stored in the form of linked list.

Files to submit : HW2_Stack.h

(※ Do NOT modify the inside of the Stack class and do NOT write main() function)

(※ As the data type of *T*, basic C++ data types (int, double, char, etc.) are assumed)

(※ To avoid errors in the template when the header and the source are separated, implement all within the header)

Stack Class

- **Definition**
 - Stack : A data structure following LIFO (Last-In-First-Out) principle
- **Node Class** (define as **Private**)
 - Same as the *Node* class as defined in Lab 3
- **Member Variable** (define as **Private**)
 - **Node*** **top** : A pointer pointing to the top of the stack
- **Constructor**
 - **Stack()** : Constructor; initialize *top* to nullptr
- **Destructor**
 - **~Stack()** : Destructor; delete all nodes in the stack
- **Member Function**
 - **bool isEmpty() const** : Return true if the stack is empty, otherwise, return false
 - **T getTop() const** : Return the data the *top* points in the stack;
return T() if the stack is empty
 - **int getSize() const** : Return the number of data in the stack
 - **void push(const T& x)**
 - ◆ Insert new node with *x* as data into the stack

- **T pop()**
 - ◆ If the stack is not empty, return the *data* at the location to which the *top* points, delete the node *top* points, and update the *top* to point the next node
 - ◆ Return T() if the stack is empty
- **void print() const**
 - ◆ Print all data stored in the stack as follows
e.g.) Print [] if the stack is empty
Print [1, 2, 3] when the stack contains 1, 2, and 3 from the top

When HW2_Stack_Test.cpp is executed, the result is as follows.

```

C:\Windows\system32\cmd.exe
# initialization
Top : 0
[]

# Push 10 times
Push 1
Push 2
Push 3
Push 4
Push 5
Push 6
Push 7
Push 8
Push 9
Push 10
Top : 10
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]

# Pop 4 times
Pop 10
Pop 9
Pop 8
Pop 7
Top : 6
[6, 5, 4, 3, 2, 1]

# Push 2 times
Push : 100
Push : 200
Top : 200
[200, 100, 6, 5, 4, 3, 2, 1]

# Pop 6 times
Pop 200
Pop 100
Pop 6
Pop 5
Pop 4
Pop 3
Top : 2
[2, 1]

계속하려면 아무 키나 누르십시오 . . .

```

P2 (35pt). Complete the array-based Queue class in HW2_Queue.h to satisfy the following conditions by referring to the given HW2_Queue.h. Each element of the queue is stored in the array.

Files to submit : HW2_Queue.h

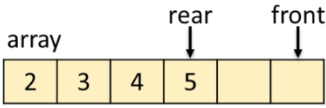
(※ Do NOT modify the inside of the Queue class and do NOT write main() function)

(※ As the data type of *T*, basic C++ data types (int, double, char, etc.) are assumed)

(※ To avoid errors in the template when the header and the source are separated, implement all within the header)

(※ shift() function is not a basic function of the queue, but is added for the assignment)

Queue Class

- **Definition**
 - Queue : A data structure following FIFO (First-In-First-Out) principle;
in this assignment, it is implemented in the form of a circular queue
 - **Member Variable** (define as **Private**)
 - **int front** : An index pointing to the front of the queue;
the queue data starts immediately after the location pointed to by *front*
 - **int rear** : An index pointing to the rear of the queue;
the same as *front* when the queue is empty;
the queue data ends the location pointed to by *rear*
 - 
 - **T* array** : A pointer pointing to the array that stores data of the queue
 - **int capacity** : A size of the array that stores data of the queue
- **Constructor**
 - **Queue(int c)** : Constructor; create *array* of *c* (>0) size,
and initialize *front* and *rear* to 0 and *capacity* to *c*
 - **Destructor**
 - **~Queue()** : Destructor; delete *array* of the queue
 - **Member Function**
 - **bool isEmpty() const** : Return true if the queue is empty, otherwise, return false
 - **T getFront() const** : Return the front data in the queue; return T() if the queue is empty
 - **T getRear() const** : Return the rear data in the queue; return T() if the queue is empty
 - **int getSize() const** : Return the number of data in the queue
 - **int getCapacity() const** : Return the *capacity*

- **void push(const T& x)**
 - ◆ If there is free space in the queue, add x to the queue and update *rear*
 - ◆ If the queue is full (only one space remains in the *array* when new data is inserted, e.g. 4/5 \rightarrow 5/10 (size / capacity)), double the *array* size of queue and perform push process
- **T pop()**
 - ◆ If the queue is not empty, return the front data of the queue and update *front*
 - ◆ Return T() if the queue is empty
- **void print() const**
 - ◆ Print all data stored in the queue as follows
e.g.) Print [] if the queue is empty
Print [1, 2, 3] when the queue contains 1, 2, and 3 from the front to rear
- **void shift(int amount)**
 - ◆ If *amount* is negative, shift the queue to the left to the absolute value of *amount*
 - ◆ If *amount* is positive, shift the queue to the right to the absolute value of *amount*
e.g.) The queue *a* contains [1, 2, 3, 4, 5, 6, 7];
if shift amount is 2, the result of a.print() is [6, 7, 1, 2, 3, 4, 5];
if shift amount is -3, the result of a.print() is [4, 5, 6, 7, 1, 2, 3]

When HW2_Queue_Test.cpp is executed, the result is as follows.

```

C:\Windows\system32\cmd.exe
# Initialization
Front : 0
Rear : 0
[]

# Push 7 times
Push 0 <1/5>
Push 1 <2/5>
Push 2 <3/5>
Push 3 <4/5>
Push 4 <5/10>
Push 5 <6/10>
Push 6 <7/10>
Front : 0
Rear : 6
[0, 1, 2, 3, 4, 5, 6]

# Pop 4 times
Pop : 0 <6/10>
Pop : 1 <5/10>
Pop : 2 <4/10>
Pop : 3 <3/10>
Front : 4
Rear : 6
[4, 5, 6]

# Push 9 times
Push 100 <4/10>
Push 200 <5/10>
Push 300 <6/10>
Push 400 <7/10>
Push 500 <8/10>
Push 600 <9/10>
Push 700 <10/20>
Push 800 <11/20>
Push 900 <12/20>
Front : 4
Rear : 900
[4, 5, 6, 100, 200, 300, 400, 500, 600, 700, 800, 900]

# Pop 6 times
Pop : 4 <11/20>
Pop : 5 <10/20>
Pop : 6 <9/20>
Pop : 100 <8/20>
Pop : 200 <7/20>
Pop : 300 <6/20>
Front : 400
Rear : 900
[400, 500, 600, 700, 800, 900]

# Shift -17
[900, 400, 500, 600, 700, 800]
계속하려면 아무 키나 누르십시오 . . .
  
```

P3 (40pt). There are three methods of arithmetic expression.

- **Prefix Notation** : operator is written ahead of operands; e.g.) +ab
- **Infix Notation** : operators are used in between operands; e.g.) a+b
- **Postfix Notation** : operator is written after the operands; e.g.) ab+

For example, the expression '(5-6)*7' in infix notation can be written as '*(-56)7' or '*-567' in prefix notation, or can be written as '56-7*' in postfix notation. However, it is difficult for a computer to interpret the infix notation immediately because this notation has the disadvantage of changing the calculation order according to the operator's priority. Therefore, the expression should be converted into prefix notation or postfix notation that can be calculated in order regardless of the operator's priority. In this problem, we would like to implement a calculator that (1) **converts the expression written in infix notation into postfix notation** and (2) **derives the calculation results** using the Stack and Queue classes implemented above.

※ In this problem, only one-digit integer (0~9) operands are used.

(1) Converting from infix notation into postfix notation

Stack is required to convert infix notation to postfix notation. For example, when the expression '1+2' is entered in infix notation, operand 1 can be output immediately, but operator + should be output after operand 2. In this case, the operator + is stored in the stack and operand 2 is printed first. Also, consider the case where the input expression is '5-2*3'. After 5 and 2 are printed and operator - is stored in the stack, operator * should be output first because operator * has a higher priority than operator -. Thus, for incoming operators to be output first, the LIFO structure, stack, is suitable as the data structure for operator storage.

However, operators that come later are not always output first. In the case of '5*2-3', '2-3' is calculated first using the above method, so the operator has to be prioritized. Basically, open and closed brackets ('(', ')') have the lowest operator priority, followed by addition and subtraction, multiplication and division, and power. Since the expression in parentheses is always calculated first, if the open bracket '(' appears, add it to the stack, and if the closed bracket ')' appears, pop must be performed until the open bracket '(' is found in the stack.

The following pseudo code shows the method of converting infix notation to postfix notation.

```

exp_infix ← expression in infix notation (no space)
exp_postfix ← empty expression
stack ← empty stack
for (character ch in exp_infix) { // repeat from exp_infix[0] to exp_infix[size-1]
    if (ch is an operand) { add ch to exp_postfix }
    else if (stack is empty) { push operator ch to stack }
    else if (top node of stack has lower priority than ch)
        { push operator ch to stack }
    else if (top node of stack has higher priority than ch) {
        pop operators in stack until the top node of stack has lower
        priority than ch and add the results to exp_postfix
        push operator ch to stack }
}
pop all operators in stack and add the results to exp_postfix
return exp_postfix

```

The pseudo code above does not take parenthesis into account, so you must consider parenthesis when implementation.

(2) Calculating using postfix notation

Since the operators are arranged in sequence as the infix notation is converted to the postfix notation, the following calculations can be repeated until all operations are performed using the stack.

- 1) Push it to the stack when meeting operand
- 2) Pop two operands from the stack when meeting operator,
and push the results to the stack after performing the operation

e.g.) The expression '(2+5)*3^(2+1)' is converted to '25+321+^*' in postfix notation, and the process for calculating this is as follows.

- | | |
|---------------------------------|-----------------------------------|
| 1) push(2), push(5) | 2) pop() 2 and 5, and push(2+5) |
| 3) push(3), push(2), push(1) | 4) pop() 2 and 1, and push(2+1) |
| 5) pop() 3 and 3, and push(3^3) | 6) pop() 7 and 27, and push(7*27) |
| 7) pop() the final result 189 | |

Then, complete the Calculator class in HW2_Calculator.cpp to satisfy the following conditions by referring to the given HW2_Calculator.h.

Files to submit : HW2_Calculator.cpp

(※ Use Stack class in HW2_Stack.h and do NOT change the Stack class)

(※ In Calculator class, you CANNOT add or modify member variables but you can add member functions if necessary)

Calculator Class

- **Member Variable** (define as **Private**)
 - **string exp_infix** : Expression using infix notation
 - **int w_add** : Operator priority for addition(+) operator
 - **int w_sub** : Operator priority for subtraction(-) operator
 - **int w_mult** : Operator priority for multiplication(*) operator
 - **int w_div** : Operator priority for division(/) operator
 - **int w_pow** : Operator priority for power(^) operator
- **Constructor**
 - **Calculator(string str)** : Constructor; initialize *exp_infix* to *str*;
initialize the weights in order of $w_{pow} > w_{mult} = w_{div} > w_{add} = w_{sub}$
- **Member Function**
 - **void setInfixExp(string str)** : set *exp_infix* as *str*
 - **void setWeight(char op, int w)**
 - ◆ Set the priority of operator *op* (one of +, -, *, /, ^) as *w*
 - **int getWeight(char op) const**
 - ◆ Return the priority of each operators (+, -, *, /, ^)
 - **string getPostfixExp() const**
 - ◆ Convert *exp_infix* to postfix notation using the stack and *getWeight()*, and return it
 - **int calcTwoOperands(int operand1, int operand2, char op) const**
 - ◆ Return the operation result of 'operand1 op operand2'
 - e.g.) If operand1 = 2, operand2 = 3 and op = '*', it returns 6 (=2*3)
 - ◆ In the case of division (/), the calculation result is truncated e.g.) 3/2=1
 - **int calculate()**
 - ◆ Convert *exp_infix* to postfix notation using *getPostfixExp()* and return the operation result using the stack and *calcTwoOperand()*
 - ◆ If there are operations of the same priority, perform the operation on the left first e.g.) 3/2*6=1*6=6, 3*6/2=18/2=9

When HW2_Calculator_Test.cpp is executed, the result is as follows.

```
C:\Windows\system32\cmd.e...  
Calculate (2+5)*3^(2+1)  
Postfix : 25+321+^*  
Result : 189  
  
Calculate (2+5)*3^(2+1)  
Postfix : 25+3*21+^  
Result : 9261  
  
Calculate 3/2*6  
Postfix : 32/6*  
Result : 6  
  
Calculate 3*6/2  
Postfix : 36*2/  
Result : 9  
  
Calculate (3*(6/2))  
Postfix : 362/*  
Result : 9  
  
계속하려면 아무 키나 누르십시오 . . .
```


- **Note**

- **Scoring will be based on execution results in Microsoft Visual Studio 2019.**
- **Do NOT modify given names of functions and variables! Otherwise score will be deducted.**
- **Submit the files with the exact file names given! Otherwise score will be deducted.**
- **Output should be in the same form as the example given! Otherwise score will be deducted.**
- **Scoring will be done with more complex case than the example test case given.**
- **No plagiarism! If plagiarism is detected, 0 points will be given for the assignment and it will be notified to the professor.**
- **If you have any question, ask questions via e-mail only if they are not resolved after sufficient search has been done.**

- **How to submit**

- Write code for each problem in each .h / .cpp file.
- Files should be saved **with exact file names** given in each problem.
- Compress your code into **one compressed file**.
 - ◆ The compressed file name should be "HW2_(name)_(student ID).zip" using zip compression.
 - ◆ e.g.) "HW2_김태환_2017-11111.zip"
- Submit the compressed file to the "Assignment 2" on the eTL course page.

- **Deadline for submission**

- **By Friday, November 1st, 11:59 pm.**
- Submit the assignment **via e-mail within the deadline** if there is any problem when you submit it on eTL.
E-mail : ds@snucad.snu.ac.kr
- **No delay submission (both eTL / e-mail). 0 points for late submission or no submission.**
- **Make sure that the file is attached and submitted! 0 points for submission without attachment.**