De Anza College
Summer 2016
CIS 22C: Data Abstraction and Structures
Programming Project 1 (Evaluation of Expressions)

For this programming project, you will be implementing two abstract data types, stack and queue and you will use them for evaluating simple mathematical expressions entered in infix notation. The following lists our assumptions on the mathematical expression entered by the user:

1. The mathematical expression in infix notation is a valid one (e.g. 1 + 2*3 is valid while 1 +*2 is not; or 5 is valid while +3*4 / is not).

2. The operands of the given mathematical expression are integers from 0 to 9 only.

3. The only allowable operators are + (for addition), - (for subtraction), * (for multiplication), / (for division), and ^ (for exponentiation). Negative sign is not allowed.

4. Division by zero is not allowed.

5. The exponentiation operator has the highest precedence followed by the multiplication and division operators and finally the addition and subtraction operators.

6. The exponentiation operator is right-associative while all the other operators are left-associative.

7. Parentheses can appear in the mathematical expression.

8. Parentheses can be nested.

9. Assume that the parentheses are well-balanced.

10. Blank spaces can appear in the mathematical expression.

11. There is no limit as to the length of the mathematical expression.

12. The input is terminated when the user hits the `ENTER` key.

The first task for this project is to implement your own version of stack and queue as class templates. Your stack class template should support the following operations: `push`, `pop`, `getTop`, `isEmpty`, and `print`. The following provides the signature and description for each of the stack operations:

- **`bool push(const T item)`** - returns `true` if `item` was successfully pushed on top of the stack; otherwise, returns `false`
- **`bool pop(T & item)`** - returns `true` if the topmost element was successfully assigned to `item` and removed from the stack; otherwise, returns `false`
- **`bool getTop(T & item) const`** - similar to `pop` except that the topmost element is not removed from the stack
- **`bool isEmpty(void) const`** - returns `true` if the stack is empty; otherwise, returns `false`
- **`void print(void) const`** - prints the elements of the stack from top to bottom separated by a single space

Your queue class template should support the following operations: `enqueue`, `dequeue`, `isEmpty`, and

`print`. The following provides the signature and description for each of the queue operations:

- **`bool enqueue(const T & item)`** - returns `true` if item was successfully inserted at the back of the queue; otherwise, returns `false`
- **`bool dequeue(T & item)`** - returns `true` if the front-most element was successfully assigned to `item` and removed from the queue; otherwise, returns `false`
- **`bool isEmpty(void) const`** - returns `true` if the queue is empty; otherwise, returns `false`
- **`void print(void) const`** - prints the elements of the queue from front to back separated by a single space

Once you have implemented your stack and queue, the second task is to use them to convert the given mathematical expression in infix notation to postfix notation. Assuming you have already enqueued the tokens (i.e. operands, operators, and parentheses) that make up the given mathematical expression in the order that they are entered by the user into some queue, say Q1, the following algorithm describes how to convert the infix expression stored in Q1 to postfix expression that will be stored in a second queue, say Q2 :

```
Create an empty stack S1
Create an empty queue Q2
Enqueue the end-of-input marker '#' to Q1
Push the end-of-input marker '#' to S1
Dequeue an element from Q1 and store it to c

While c is not the end-of-input marker '#' do
        If c is an operand then
                Enqueue c to Q2
        Else
          If c is a closing parenthesis ')' then
            While S1 is not empty and the topmost element of S1 is not an opening parenthesis '(' do
                Pop an element from S1 and store it to op
                Enqueue op to Q2
            Pop an element from S1 and store it to op
          Else
            While S1 is not empty and the isp of the topmost element of S1 is >= to the icp of c do
                Pop an element from S1 and store it to op
                Enqueue op to Q2
            Push c to S1
    Dequeue an element from Q1 and store it to c

If S1 is not empty then
  While S1 is not empty and the topmost element of S1 is not the end-of-input marker '#' do
    Pop an element from S1 and store it to op
    Enqueue op to Q2
```

The following table shows the isp (i.e. in-stack-priority) and icp (i.e. in-coming-priority) of the operators:

Operator	ISP	ICP
^	3	4
*, /	2	2
+, −	1	1
(	0	4
Others	-1	-1

Once you have converted the mathematical expression in infix notation to postfix notation (assuming the postfix notation is stored in Q2), your last task is to evaluate it. The following algorithm shows how the evaluation can be done:

```
Create an empty stack S2

While Q2 is not empty do
   Dequeue an element from Q2 and store it to op
   If op is an operand then
         Push the numerical value of op to S2
   Else
         Pop an element from S2 and store it to op2
         Pop an element from S2 and store it to op1
         Perform the operation op1 op op2 and push the result to S2
```

S2 should contain the result of the evaluation. Your program's output should look like the one shown below.

```
Please enter an expression in infix: ( ( 2 – 1 ) * 2 ^ 2 ^ 3 + 8 / 4 )
You typed: ( (2 – 1 ) * 2 ^ 2 ^ 3 + 8 /4 )
Contents of queue: ( ( 2 – 1 ) * 2 ^ 2 ^ 3 + 8 / 4 ) #
Expression in postfix: Contents of queue: 2   1 – 2   2   3   ^   ^   *   8   4   /   +
Contents of queue: 1 – 2 2 3 ^ ^ * 8 4 / +
Contents of stack: 2
Contents of queue: – 2 2 3 ^ ^ * 8 4 / +
Contents of stack: 1 2
Contents of queue: 2 2 3 ^ ^ * 8 4 / +
Contents of stack: 1
Contents of queue: 2 3 ^ ^ * 8 4 / +
Contents of stack: 2 1
Contents of queue: 3 ^ ^ * 8 4 / +
Contents of stack: 2 2 1
Contents of queue: ^ ^ * 8 4 / +
Contents of stack: 3 2 2 1
Contents of queue: ^ * 8 4 / +
Contents of stack: 8 2 1
Contents of queue: * 8 4 / +
Contents of stack: 256 1
Contents of queue: 8 4 / +
Contents of stack: 256
Contents of queue: 4 / +
Contents of stack: 8 256
Contents of queue: / +
Contents of stack: 4 8 256
Contents of queue: +
Contents of stack: 2 256
Contents of queue:
Contents of stack: 258
The result is 258.
Bye, press any key to exit.
```

The following are the additional instructions for this programming project:

- This programming project is to be done individually.
- Create a readme.txt file that describes exactly how to compile and execute your program.
- Collect your source codes, readme file, and other files needed to compile and execute your program into one ZIP file called YourFirstName_YourLastName_prog1.zip. **Please DO NOT include any executable files in your ZIP file.**
- Make sure you follow good object-oriented programming approach, good coding style, and proper documentation.

The grading for this programming project will be based not only on the correctness of the program, but also on the program's overall design, coding style, and documentation.

Submitting Assignments:

- Submit the ZIP file on Catalyst