CSCI205 Computer Science III
Lab Assignment
Stacks, Queues, Deques

Notes: All code must build error and warning free. Use the command line arguments **-Wall** and **-pedantic** to ensure you are warning free

Documentation Requirements: All external sources must be documented and cited. If you use any AI tool for any part of this lab you must document and cite it. If you use Stack Overflow or any othert online source you must document and cite. Failure to do so will be considered a violation of academic integrity.

If you write logic yourself also document this. Any lack of attribution instance will be penalized.

Lab Description: You are going to reinvent the wheel a bit here in an effort to really grasp the underlying mechanics of **abstract data type** implementation. C++ has templated Stack, Deque and Queue classes already defined but you are not allowed to use C++ library objects for this assignment. ***You are going to build them yourself as an exercise***. All problems in this lab will be solved using structures that define yourself ***other than an array***. Envision this as a learning exercise, not as a "solve the problem as quickly as possible" exercise. Your goal is to develop your understanding and critical thinking, not to simply arrive at the answer as quickly as possible.

You will use C++ templates to define structures that are ***type agnostic*** and not restricted to single data type support. This will allow programmers to specify the **data types** at compilation time, whatever our context requires. In one instance we may be stacking simple types like ints or doubles, and other times we may be stacking complex object types like Fractions or any other type we may define in the future.

Avoid placing type specific code in your abstract data types

Templates allow us to define **how** a collection should be managed without worrying about **what** the specific type actually is. You have seen this in practice with Java ArrayLists and C++ Vectors, where the type is specified at compile time

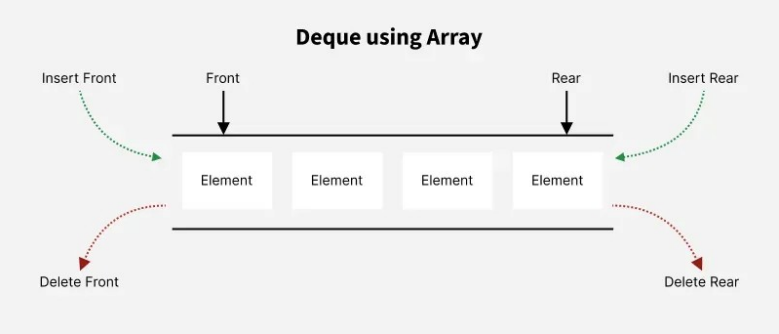
Examples:

- ArrayList<String>, ArrayList<Fraction>, ArrayList<Customer>
- vector<String>, vector<Fraction>, vector<Card>

READ THIS: C++ Class Templates: https://www.learncpp.com/cpp-tutorial/133-template-classes/

Task One: Study the following example operations on a Deque. Deques allow for insertion and removal on both ends. You want to consider this a generalized linear data structure that can also be re-used in the context of LIFO stacks and FIFO queues.

The Deques, Stacks and Queues that we work with in this lab will be built using ***partially filled arrays as the underlying memory managemen***t. Keep this in mind moving forward because we want all of our insertions and removals to maintain O(1) efficiency.



Take note of the challenges that arrays face in this situation. This will help frame our discussion of alternate memory management approaches.

Initial empty deque with 2 integer index "pointers" front and rear

Value										
Index										
Refs	Front									Rear

Notes:

- Front ref can begin pointing to the position "to be inserted at" or -1. This will determine how and when you increment the reference
- Back ref can begin pointing to the position "to be inserted at" or capacity – 1. This will determine how and when you decrement the reference
- These index values will move towards each other as the deque fills. If they cross paths, the deque is full

deque after 6 push operations

Value	**42**	**86**	**53**					**92**	**27**	**12**
Index	0	1	2	3	4	5	6	8	9	10
Refs			Front					Rear		

1) deque.push_front(42);
2) deque.push_back(12);
3) deque.push_back(27);
4) deque.push_back(92);
5) deque.push_front(86):
6) deque.push_front(53);

x = deque.pop_front( );

Value	42	86	~~53~~					92	27	12
Index	0	1	2	3	4	5	6	8	9	10
Refs		Front						Rear		

- x = 53
- no need to over write 53. Simply moving the Front ref marks position 2 as available

x = deque.pop_back( );

Value	42	86	~~53~~					~~92~~	27	12
Index	0	1	2	3	4	5	6	8	9	10
Refs		Front							Rear	

- x = 92
- no need to overwrite 92. Simply moving Rear marks position 8 as available

x = deque.back( );

Value	42	86	~~53~~					~~92~~	27	12
Index	0	1	2	3	4	5	6	8	9	10
Refs		Front							Rear	

- x = 27
- value stays in list and Rear does not move

deque.push_front(19);

Value	42	86	19					~~92~~	27	12
Index	0	1	2	3	4	5	6	8	9	10
Refs			Front						Rear	

- 19 is inserted at the front, Front advances

x = deque.pop_back( );

Value	42	86	19					~~92~~	~~27~~	12
Index	0	1	2	3	4	5	6	8	9	10
Refs			Front							Rear

- x = 27
- Rear moves

x = deque.pop_back( );

Value	42	86	19					~~92~~	~~27~~	~~12~~
Index	0	1	2	3	4	5	6	8	9	10
Refs	Rear		Front							

- x = 12
- Rear ***wraps around***

x = deque.pop_back( );

Value	~~42~~	86	19					~~92~~	~~27~~	~~12~~
Index	0	1	2	3	4	5	6	8	9	10
Refs		Rear	Front							

- x = 42
- Rear moves

x = deque.push_back(67);

Value	67	86	19					~~92~~	~~27~~	~~12~~
Index	0	1	2	3	4	5	6	8	9	10
Refs	Rear		Front							

- 67 added
- Rear moves

x = deque.push_back(117);

Value	67	86	19					~~92~~	~~27~~	117
Index	0	1	2	3	4	5	6	8	9	10
Refs			Front							Rear

- 117 added
- Rear ***wraps around***

TASK TWO: Define and implement a templated Deque.

1. You have been provided with a skeleton ***templated Deque class*** definition. This class consists of a series of method definitions marked with **|| TO DO**. Your first task is to finish all of these off. Be sure that you read the linked article above and begin to develop an understanding of type agnostic code using templates.

   You are allowed to add data or make sensible changes to the provided code **as long as you include comments about your decision process.**

   Ask questions!

2. After finishing and testing step 1, implement a ***templated stack*** in C++ by using the ***composition pattern*** and including a private reference to a ***deque*** object. Using composition over inheritance allows us to expose the behaviors we choose to the users of our stack, while hiding those that aren't "stack appropriate".

   Be sure you understand this design decision to choose composition over inheritance

   a. **Include:** push, pop and peek. you must use calls to deque methods where appropriate. Example: implement **push** by calling the deque method **push_front**. All you are doing here is defining an ***abstract interface*** for stack behaviors onto a more general deque data structure.

   b. all methods must maintain O(1) complexity

3. Implement a ***templated queue*** in C++ in the same fashion described above.

   a. **Include**: enqueue and dequeue. Use calls to deque methods where appropriate.

      Example: implement **enqueue** by calling **push_front.**
      Example: implement **deque** by calling **pop_back.**

      All you are doing here is defining an ***abstract interface*** for queue behaviors onto a more general deque data structure.

   b. all methods must maintain O(1) complexity

4. Also mentioned in the linked article on templating in C++ is an unexpected issue that exists when using generic types and separating the class interface from the implementation (.h files and .cpp files). Essentially you cannot do it without some modifications, and the easiest solution is to

a. Combine the interface and implementation into the header file, placing the implementation inside the class definition and changing the file extension to **hpp**. This file extension simply communicates the fact that there is code inside the header file.

   Check **Deque.hpp** for an example of this.

5. Write a simple **main** file and prove that your class behaviors function as expected. Use the provided ***main.cpp*** as an example but be more thorough and you are required to include descriptive messages in your output. Failure to do so will be penalized.

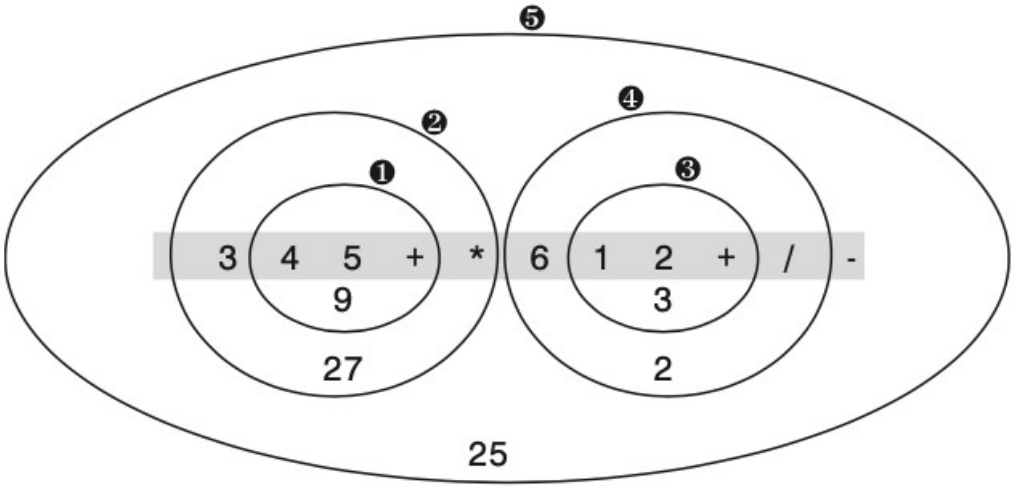
TASK THREE: Postfix Notation. This concept provides insight into how compilers manage to parse complex mathematical expressions contained in source code.

https://www.free-online-calculator-use.com/postfix-evaluator.html

Convert the following infix expressions to postfix notation. Feel free to use an infix/postfix calculator to check your work but make sure you can do it by hand. This will show up on an exam. Include your answers in a text document as part of your submission

INFIX EXPRESSION	**POSTFIX EXPRESSION**
A + B – C	
A * B / C	
A + B * C	
A * B + C	
A * (B + C)	
A * B + C * D	
(A + B) * (C – D)	
((A + B) * C) – D	
A + B * (C – D / (E + F))	

The following figure shows how a human can evaluate a postfix expression using visual inspection and a pencil. **Postfix Expression: 345+*612+/–**



Start with the first operator on the left and draw a circle around it and the two operands to its immediate left. Then apply the operator to these two operands— performing the actual arithmetic—and write down the result inside the circle. In the figure, evaluating 4+5 gives 9.

Now go to the next operator to the right, and draw a circle around it, the circle you already drew, and the operand to the left of that. Apply the operator to the previous circle and the new operand and write the result in the new circle. Here 3*9 gives 27. Continue this process until all the operators have been applied: 1+2 is 3, and 6/3 is 2. The answer is the result in the largest circle: 27–2 is 25.

Convert the following postfix expressions to infix notation.

POSTFIX EXPRESSION	**INFIX EXPRESSION**
xy*5z*/10+	
xy*2+12/y*x+	
ABCD/+*	
ABC+D/*	
2ABC+*2-*D/	
2A*B*C+2D/-	
2AB*BC-/+CD/+	
ABBC-/+CD/+2+	
ABC–DEF+/*+	

Infix to Postfix Algorithm

1. Scan the infix expression from left to right.
2. If the scanned character is an operand, add it to the postfix string.
3. Else,
    a. If the precedence of the scanned operator is greater than the precedence of the operator on the stack, push it.
    b. Else, Pop all the operators from the stack which are greater than or equal in precedence to that of the scanned operator.
    c. Push the scanned operator on the stack. (If you encounter parenthesis while popping, stop there and push the scanned operator on the stack.
4. If the scanned character is an (, push it on the stack.
5. If the scanned character is an), pop the stack and concatenate it until a (is encountered, and discard both the parenthesis.
6. Repeat steps 2-6 until infix expression is scanned.
7. Print the output
8. Pop and output from the stack until it is not empty.

Example: Translating Infix A+B*(C–D) to Postfix

Character from Infix	**Infix Expression**	**Postfix Expression**	**Stack Contents**	**Rule**
A	A	A		write A to postfix
+	A+	A	+	push + on stack
B	A+B	AB	+	write B to postfix
*	A+B*	AB	+*	push * on stack
(	A+B*(	AB	+*(	push (on stack
C	A+B*(C	ABC	+*(	write C to postfix
–	A+B*(C–	ABC	+*(–	push - on stack
D	A+B*(C–D	ABCD	+*(–	write D to postfix
)	A+B*(C–D)	ABCD–	+*(	pop stack to postfix
	A+B*(C–D)	ABCD–	+*(	quit popping when (
	A+B*(C–D)	ABCD–	+*	pop stack to postfix
	A+B*(C–D)	ABCD–*	+	pop stack to postfix
	A+B*(C–D)	ABCD–*+		

Programmatically Evaluating Postfix Notation

1. Parse postfix expression from left to right
2. Push operands onto the stack
3. For each operator
    a. pop two operands from the Stack
    b. apply the operator
    c. push the result onto stack
4. When finished, pop the stack for the result

Example: abc*+d+ a = 2, b = 3, c = 4, d = 5

stack	**operator**	**rule**
a		push a
a, b		push b
a, b, c		push c
pop c	*	operator found, pop two operands from stack
pop b	4*3=12	perform operation
a, 12		push result on to stack
a, 12	+	operator found, pop two operands from stack
pop 12		
pop a	12+2	perform operation
14		push result on to stack
14, d		push d
pop d	+	operator found, pop two operands from stack
pop 14	5+14	perform operation
19		push result to stack
19		expression complete, pop stack from result

TASK FOUR: Using the Stack that you created above to complete the following exercises. These are modifications of algorithms demonstrated in the readings. You may use these as a basis for your work here.

- **Ex 1:** Modify the ***infix-to-postfix*** algorithm so that it can detect errors in the parenthesis pairs. Use the bracket matching algorithm and stack to manage this process.
- **Ex 2:** Implement a direct infix evaluator that combines the functionality of ***infix-to-postfix*** conversion and the ***postfix evaluation*** algorithm. Your evaluator should process infix tokens from left to right and use two stacks, one for operators and one for operands, to perform the evaluation. It should also include the ability to determine if there are parenthesis issues.

SUBMISSION: Push all files to your repository. You should expect to push

- Deque.hpp
- Stack.hpp
- Queue.hpp
- Any associated cpp files to go with your classes (if using)
- main.cpp
- infix_calculator.cpp
- Infix to postfix answers
- Anything else I may have forgotten to mention in this list.

  Please do not push executables, .vscode folders, lab instruction documents or any other files that aren't relevant to me when assessing your work.

You do not need to perform any graphing for this assignment, but I will be analyzing your methods to ensure they conform to the efficiency requirements mentioned above.