

COS 135 Individual Assignment 9

Due: Monday 04/12/23 End of the day

This assignment is based on the concepts we covered on dynamic memory allocation, you must use dynamic memory to implement C programs for parts (a), (b), and (c).

Write C programs for following tasks and submit your source codes (you must submit .c files without compilation warning or errors). Sample Program inputs/outputs are given.

What to submit:

- Please **submit three separate .c source codes**. Your programs must produce similar outputs if the same inputs are provided.

[-5 pts each] Source codes should be able to compile and execute without errors or warnings:

- Always compile your source codes with -Wall (enable all warnings) switch to find all the compiler warnings and fix them (see example below).

```
$ gcc mycode.c -o myprogram -Wall
```

[-10 pts each source code] **Comments** are required in the following locations:

- **[-2 pts]** At the top of the source code, comment your name and insert a short program description
- **[-4 pts]** Comment the purpose of each variable.
- **[-4 pts]** Comment major sections of your code such as inputs, functions, and outputs.

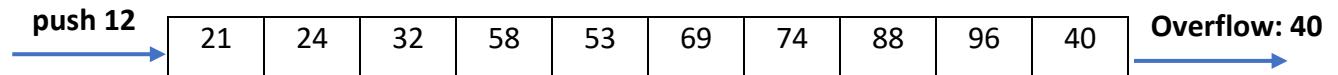
Program Design: Your program is a professional document and must be neat and easy to read.

All programs should follow these specifications.

- Comments should be aligned and entered in a consistent fashion
- Blank lines should be added to aid readability
- Code within blocks should be indented
- Comments should not contain spelling mistakes
- Variable names should be meaningful
- Define functions where necessary
- **Take time to design your code (program flow and functions) before start coding**

(a) (30pts): Implement a Dynamic FIFO (First-In First-Out) Queue

FIFO Queues are a type of data structures, specifically designed to operate in a FIFO context (first-in first-out), where elements are inserted into one end of the queue and extracted from the other. Queues should support push command to insert new data elements.



push: Inserts a new element at the start of the queue. For example, user enters “push 5”: the number 5 should be inserted as the first element. Before inserting, shift all the elements in the queue. If the array overflows, remove last element and prints it to the screen. After shifting all the existing elements, insert the new value to the start of the queue.

Write a C program to implement this algorithm, you may use following guidelines:

1. Pass the queue size as a command line argument (`./fifo_queue 5` to create 5 elements)
(See sample code attached to demonstrate receiving command line arguments)
2. Implement the push function (a separate function) as explained to insert data to the queue
(you may need to save the last elements location in a variable for reference)
3. A user should be able to issue ‘push’ command (for example, push 50) to insert new values
4. A user should be able to issue ‘print’ command to print the current elements in the queue
to the terminal
5. Implement the shift algorithm to shift data to the right (tip: you may recall the swap
function we discussed or implement your own approach)
6. Insert the new value into the first element after shifting the data
7. If FIFO queue overflows, print the overflowing value to the terminal

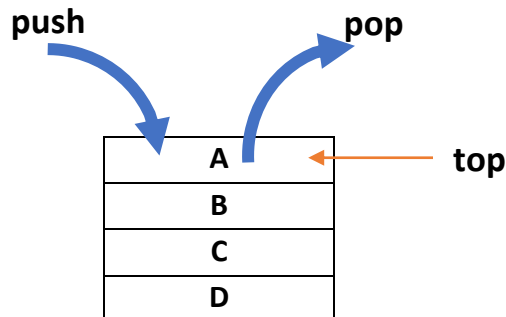
Design and develop a C program to fulfil above requirements. Define functions where necessary.
You need to use dynamic memory allocation to create the data structure (i.e. FIFO Queue).

Sample output:

```
./fifo_queue 4
Enter command (push, print, or quit): push 12
Enter command (push, print, or quit): push 4
Enter command (push, print, or quit): print
4, 12
Enter command (push, print, or quit): push 7
Enter command (push, print, or quit): push 5
Enter command (push, print, or quit): push 3
Overflow: 12
```

(b) (50pts): Implement a dynamic stack

A **stack** is a container of objects that are inserted and removed according to the last-in first-out (**LIFO**) principle. In the pushdown stacks only two operations are allowed: **push** the item into the stack and **pop** the item out of the stack. A stack is a limited access data structure - elements can be added and removed from the stack only at the top. push adds an item to the top of the stack, pop removes the item from the top. A helpful analogy is to think of a stack of books; you can remove only the top book, also you can add a new book on the top.



push: Inserts a new element from the top (note: if this is the first element, it will be saving at the top, when new data is pushed this top element should go down (one level at a time) – similar to put books into a box; second and subsequent elements will be on top of the previous element).
An example command: push N

pop: Removes the topmost element from the stack and output to the terminal.

Write a C program to implement this algorithm using dynamic memory, use following guidelines:

1. First, define a **dynamic char array** of size 1 (this array should grow as new data is pushed or shrink when the data is popped).
2. Implement the push function (a separate function) as explained to insert new data to the stack. A user should be able to issue 'push' command (for example, push R) to insert a new value. New value R should be added to the top and all the existing values should shift down.
3. A user should be able to issue 'pop' command to remove the topmost element from the stack and print it to the terminal. All the remaining values should shift up and delete the bottom most array element.
4. A user should be able to issue 'print' command to print all the elements in the current stack to the terminal.
5. A user should be able to issue 'quit' command to quit from the application
6. You should test your code for maximum 10 elements in the array.

Design and develop a C program using dynamic memory to fulfil above requirements.

Define functions where necessary.

Sample output #1:

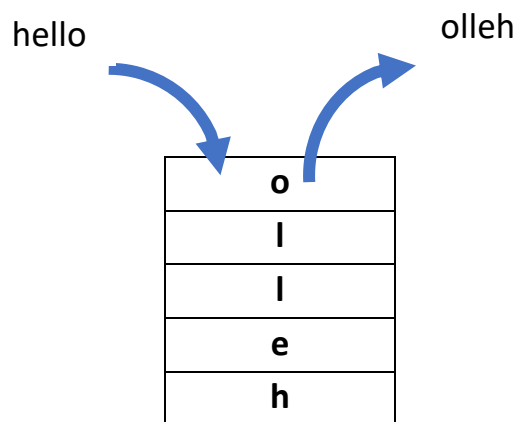
```
$/stack
Enter command (push, pop, print, or quit): push A
Enter command (push, pop, print, or quit): push B
Enter command (push, pop, print, or quit): push C
Enter command (push, pop, print, or quit): push D
Enter command (push, pop, print, or quit): push E
Enter command (push, pop, print, or quit): push F
Enter command (push, pop, print, or quit): push G
Enter command (push, pop, print, or quit): push H
Enter command (push, pop, print, or quit): push I
Enter command (push, pop, print, or quit): push J
Enter command (push, pop, print, or quit): print
J
I
H
G
F
E
D
C
B
A
Enter command (push, pop, print, or quit): push K
Error: Maximum elements in the stack
Enter command (push, pop, print, or quit): pop
Popped: J
Enter command (push, pop, print, or quit): push K
Enter command (push, pop, print, or quit): print
K
I
H
G
F
E
D
C
B
A
Enter command (push, pop, print, or quit): quit
Bye
```

Sample output #2:

```
$/stack
Enter command (push, pop, print, or quit): push N
Enter command (push, pop, print, or quit): push R
Enter command (push, pop, print, or quit): print
R
N
Enter command (push, pop, print, or quit): pop
popped: R
Enter command (push, pop, print, or quit): quit
Bye
```

(c) (20pts) Implement a **dynamic** stack (application of a Stack)

Write a C program to implement a simple application of a **dynamic stack** to reverse a word or a sentence (*you may develop this program based on the program developed in part (b)*). First, obtain user's input, create a dynamic char array, then push user's input to stack - letter by letter - then pop letters from the stack and print to the terminal. You must use dynamic memory allocation to create the array based on number of characters in the user's input (memory optimization).



Sample output #1:

```
$/stack_reverse
Enter a phrase: COS135 @ UMaine
Reversed output is "eniaMU @ 531SOC"
```