# ELEC 278 – Data Structures and Algorithms

## Lab 2: Stacks, Queues and Recursion

Start: September 28, 2022  Due: Day 7 of Lab Day

## Contents

# 1. Basic Lab Information

You should review the Lab Overview document prior to reading the material for this lab exercise.

If you have already installed a C compiler on your computer (or have one available), then you can continue with the lab. If you have not installed the C programming tools, please stop working on this lab, and review Lab Overview document Appendix A.

You should also review Appendix B of the Lab Overview document. This contains important information about how to submit your assignment.

# 2. Lab Resources

Specific to this lab:

1. Lab Instruction sheet (this document)
2. A zip file containing source code (in a folder called Lab02Src). Check the file "PACKINGLIST.txt" for a list of files that should be included.

Learner provided:

1. Programming environment selected by the learner. The Lab Overview document provides a list of possible C programming tools that you may use.

This lab has a lot of code for you to read. The purpose of the example code is to reinforce what you have seen in the class material. It also provides you with code to cut-and-paste into the code you write to solve the assigned problems. This lab has multiple programming problems to solve.

## 2.1 List of Files provided with this Lab

The files are listed in the folder called Lab02Src.

# 3. Lab Objective

There are three objectives for lab 2:

a) Look at the stack data structure and how to implement stacks using either arrays or linked lists.
b) Look at the queue data structure and how to implement queues using either arrays or linked lists.
c) Look at recursion.

## 3.1 Topics covered in this lab

- Stacks – implemented using arrays and implemented using linked lists
- Queues – implemented using arrays and using linked lists
- Recursion and its equivalent to iteration

### 3.2 Topics that were covered in Lab 1

- Arrays, pointers, using pointer as if it were an array name.
- Dynamic memory allocation – malloc() and free().
- Reading files – whole lines as strings, individual numbers, using scanf() to read data, flushing input after using scanf()
- Using command-line arguments.
- Structures and linked lists.

## 4. Lab Steps

There are four steps in this lab. The first three steps are reading or study steps – you have some code that illustrates programming concepts and you are to develop an understanding as to how the code works. The last step is work you must complete – by using materials that you studied in the earlier parts. Some of you may find it easier if you tackle one or two steps and then leave the lab for something else, coming back to it later. You can nibble on pieces over the two-week period – or have a large feast the night before the lab is due. Figure out which technique works for you.

### 4.1 Part 1 – Stacks

There are seven sample stack programs for you to read in the Lab 2 Materials. Two of the programs show how to implement a stack using an array. The first program is as simple as it can be. Notice that it uses a fixed size array to hold the stack. Notice also that the first item pushed on the stack goes in array location 0, the next item pushed goes in location 1, etc. The second program is almost identical to the first, except that all the constants relating to the stack – size, index value for top when stack is empty, index value for top when stack is full – are now handled by defines. A define is a way of using a symbol that has meaning to the human reading the code ("STACK_SIZE") rather than just using a number ("100"). Read the second program and consider how easy it is to change the size and then recompile.

The third program to look at – **Stack-MultiInstance.c** – shows how to take the concept one step further and allow your program to have multiple stacks. There are now routines that let you create and destroy stacks. In this case, the stacks are still stacks of integers.

The fourth program to look at is **Stack-Strings.c**. This is one step beyond **Stack-MultiInstance**. This code creates a generic stack functionality – capable of stacking integers or pointers to strings, as necessary. The stack code works regardless of the stack contents – but the user must do some odd casts to make it work. A cast, you may recall, is a way of telling the C compiler that even though it looks like you are making a mistake and using a variable of an inappropriate type, you really are aware of what you are doing. (Warning: This code was meant to be an illustration – real code would have a few more checks to make sure a programmer is not making mistakes while using the code.)

All the previous code used arrays for the stack. Arrays have a limitation – they are fixed in size. (There is a way to grow arrays when they need to be bigger, but we will leave that for another time. Stack-LinkedListVersion00.c gets around this issue by using a linked list for the stack. Every new item pushed on the stack gets its own node (structure) to stay in. The stack can grow "forever" – or at least until the memory limitation on the computer is reached.

The final piece of code is **Stack-BracketChecker.c.** This code checks to see if brackets in a C program are perfectly balanced – that is, it records opening brackets as they are encountered while reading the C program. When a closing bracket is encountered, it should match the last opening bracket. So, the top item is popped from the stack and if it matches the type of the closing bracket just found, then it is discarded, and processing continues. If it does not match, then an error message is printed, indicating there is a problem. Try the program on the test files **testbrkch.txt** and **testbrkch2.txt**

### 4.2 Part 2 – Queues

There are four programs showing queues – **Queue-ArrayVersion1.c**, **Queue-ArrayVersion2.c**, **Queue-ArrayVersion3.c** and **Queue-LinkListVersion.c.**

The first array version is the simple version. It allows for one queue of integers. Version 2 allows for multiple queues to exist at the same time. Version 2 stores the array for the queue within the queue structure. Version 3 is like version 2, but the array holding the queue data is separately malloc()ed.

The linked list version is the last segment of queue code. This version only implements a single queue.

### 4.3 Part 3 - Recursion

As noted in class, a recursive definition is a definition that self-references. That is, we can define a function as follows: (This is the definition of factorial)

For all values of n > 0

F(1) = 1

F(n) = F(n-1) * n for n > 1

This could be implemented in C as follows:

```
int  fact  (int n)
{
      if (n < 1)      return -1;    // not valid for factorial
      if (n == 1)     return 1;
      return  fact(n-1) * n;
 }
```

As an exercise **THAT IS NOT TO BE SUBMITTED**, implement the following function using recursion:

For all values of n = > 0

F(0) = 1

F(1) = 2

F(n) = F(n-1)*F(n-2) for n > 1

Add code to your implementation to count the number of times your function is called. Can you suggest a way to reduce the number of function calls?

## 4.4 Part 4 – Deliverables

### 4.4.1 Stack Problem

Create a program that reads in characters one at a time, and when the end of line is reached, prints the characters in reverse order.

**Detailed specifications:**

- The source file name shall be stackproblem.c.
- The input shall be one line of characters. You should handle an input of up to 80 characters (plus the newline character at the end of the line).
- The output shall be one line of characters – in opposite order to the input.
- The first line of your program shall conform to the course standard for headlines, that is, it should be
  // stackproblem.c – Lab 02 – FirstName, LastName

### 4.4.2 Queue program

Implement a multi-instance version of the linked-list queue. In your program, create 4 queues. Read numbers until you encounter a negative integer input. For every number input, if the number modulo 4 is 0, put the number in the first queue; if the number modulo 4 is 1, put the number in the second queue; if the number modulo 4 is 2, put it in the third queue, and if the number modulo 4 is 3, put it in the fourth queue. When the negative number is input, output all the numbers in the first queue, then all the numbers in the second queue, then all the numbers in the third queue, and then all the numbers in the fourth queue.

**Detailed specifications:**

- The source file name shall be queueproblem.c.
- The input shall be integer numbers separated by spaces or newlines. There may be as many as 400 inputs, and it is possible that all input numbers modulo 4 will evaluate to the same value.
- The output shall be integer numbers each printed with a %5d format, 12 per output line.
- The first line of your program shall conform to the course standard for headlines, that is, it should be
  // queueproblem.c – Lab 02 – FirstName, LastName