

► Lab 2

The Circular Input Queue (v1.0)

Demo: during your lab period between June 6 – June 10, 2022

Purpose:

Program planning is an essential (and easily-overlooked) aspect of any project; this is especially true of C programs. In this lab you are required to write a relatively short program, one that can be implemented quite nicely in under sixteen lines of executable code. Failure to properly plan this program typically results in flawed, bloated code (sometimes by a factor of two or three). Code that executes, but hasn't been optimized for execution, will not receive any marks.

You will have completed this lab when you have:

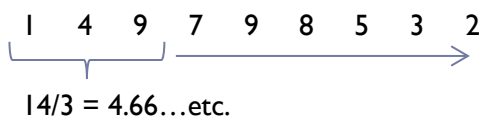
- ☐ Outlined your program's execution using either pseudocode or a flowchart;
- ☐ Implemented your code using the algorithm outlined, having avoided costly work-arounds that only bloat the code without adding to its functionality or clarity;
- ☐ Demonstrated you can debug your code in the VSC editor/debugger;
- ☐ Can adequately answer questions related to the functioning of your code.

Lab 2

The Circular Input Queue (v1.0)

I. Understand the basic concepts related to this lab

- a. A “running average” is the average value over the last n digits in a sequence of numbers. Thus for the sequence 1,4,9,7,9,8,5,3,2, the running average over the last $n = 3$ numbers will be 4.66, 6.67, 8.33, 8.00, 7.33, 5.33, 3.33 i.e. effectively, a window n digits long ‘runs’ over the sequence and averages everything inside.

1 4 9 7 9 8 5 3 2

 $14/3 = 4.66 \dots \text{etc.}$

The running average thus acts as a simple smoothing function, averaging out local fluctuations. Larger values of N will thus produce smoother data.

- b. Consider the effect of optimizing your algorithm for a sequence of $P = 10,000,000$ data points—not at all unusual in the worlds of science and modern finance. If you sum the last $n = 1000$ values in a series and do this for 10,000,000 points each time, the computational cost will increase as $\mathcal{O}(nP)$, or ten billion calculations—a not insignificant number. In modern financial markets decisions are sometimes made in milliseconds, and so the time taken to perform ten billion calculations, even though it amounts to only a second or so on a modern laptop, could be costly.
- c. A smarter way to perform the running average is to notice that for each calculation, most of the work was already done in the previous calculation; all that really changes are the first and last values in the sequence. For example, imagine we wish to average

over the last $n = 100$ numbers in a sequence of one thousand numbers numbered sequentially from 1 to 1000. Thus, for the first 100 digits

1,2,3,4,5,6,7, 8,9,10,...,94,95,96,97,98,99,100

the average is just

$$\frac{1+2+3+4+\dots+96+97+98+99+100}{100} = \text{etc.}$$

For the second average, this is just:

$$\frac{2+3+4+5+\dots+97+98+99+100+101}{100} = \text{etc.}$$

Notice that the sequence $2+3+4+\dots+99+100$ used in the second calculation was already present in the first calculation; all that changed was that the first number (1) was removed and the last number (101) was added. The same idea holds true of the next calculation of the running average:

$$\frac{3+4+5+6+\dots+97+98+99+100+101+102}{100} = \text{etc.}$$

—the first number in the sequence (2) is subtracted, and the next number (102) is added

- d. Assume we know the SUM of the last n numbers being averaged over, along with the FIRSTNUM in the sequence used in calculating that average. Rather than simply perform a new sum over every number in the running average each time (using a for loop), a better strategy is:

```
SUM = SUM - FIRSTNUM
INPUT NEWNUM
SUM = SUM + NEWNUM
AVE = SUM / N
```

- e. The sequence of N numbers to be averaged over will need to be stored in an array—so we can always find the FIRSTNUM. But we don’t require an array larger than size N .

Hence for the following discussion assume an array declared as

`datatype[N]`

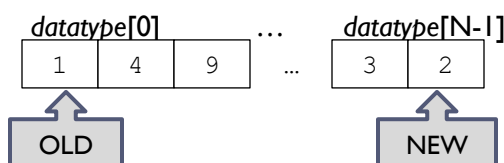
where `datatype` could be an `int`, `float`, etc., and `datatype[i]` will store the first value in the sequence at any time, and `datatype[i+N-1]` will store the last.

- f. To implement our algorithm, we *could* subtract `datatype[0]` and add the new value into `datatype[N-1]` each time. But then we'd need to shift each number in the array every time, setting `datatype[0] = datatype[1]`, `datatype[1] = datatype[2]`, etc. before we finally loaded `datatype[N]` at the end of the array—an operation just as computationally costly as completely re-summing the total each time—exactly what we're trying to avoid.

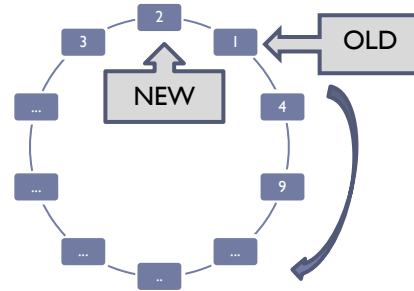
A smarter way to implement the algorithm is to use two indices that point into the array at the two locations of interest: at the location where the first and last values in the sequence are stored. So rather than performing the computationally costly operation of moving each number in the array *down*, we'll increment the two indices *up*. We'll call these two indices **OLD** (i.e. the index where **FIRSTNUM** is stored) and **NEW** (the index of where the **NEWNUM** will be entered.)

NOTE: *DO NOT* attempt to use pointers in this lab, even if you are familiar with them. You should be able to implement your algorithm entirely with arrays, which do not differ from the way you use them in Java.

- g. It is convenient to think of our storage array `datatype[N]` not as a linear sequence like this:



but rather as a circular queue, like this:

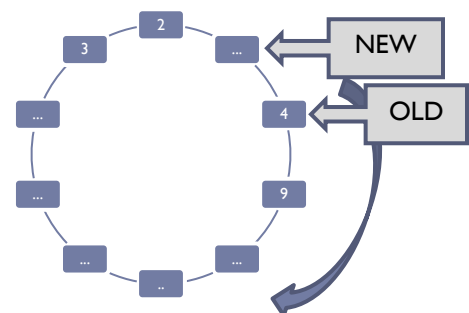


NOTE: the diagram above is based on the example indicated in part I(a), using an average over the last three values.

Now, whenever a new number is added to the queue, it automatically overwrites the old number, which is no longer needed. (But you'll need to subtract the value pointed to by **OLD** from sum before you overwrite the value stored at the **OLD** location.)

REMINDER: **NEW** and **OLD** are *NOT* the numbers being pointed to, they are the indices *that do the pointing*.

So it's the values of **NEW** and **OLD** that move, rather than the numbers in the queue. For example, in the next iteration of the algorithm given above, **NEW** and **OLD** would both advance one location:



Again: Outside of the initialization, at no point in your code is it necessary to use a for loop—or a do, or a while—to sum up the values between **OLD** and **NEW**. The only thing that 'loops' are the values of the indices as they move around the buffer.

II. Implement the algorithm outlined above in pseudocode or a flowchart.

- a. Given this information, write pseudocode or a flowchart to describe the operation of your circular queue.

There are three complications not directly addressed above, which you'll need to solve:

- i. The array has a maximum size corresponding to the value of N , the maximum number of values to be averaged over. Your code needs to 'wrap around' back to index 0 of the array without using any logic statements, such as an `if` statement. You should be able to do this using a simple mathematical operation, rather than using something that looks like: `if (NEW > N) NEW=0...` or its equivalent, in any fashion; there is a single math operation that solves this problem for you.
- ii. Assume the size of the buffer, N , is fixed at 10 for the purposes of this exercise. However, the size of the averaging window, n , should be adjustable, i.e. your algorithm should work for any values of n and N , provided $n \leq N$. You'll need to prompt the user for this value at the start of the code. You can assume that the user is well behaved, so it isn't necessary to check for the case where $n > N$.
- iii. When the program starts up, and before it starts prompting the user to enter numbers to fill the queue, your array has nothing stored in it, except, presumably, 0's. How many numbers should you average over if the number of values actually entered—call it `VALUES`—is less than n ? In this case, you must average over the total number of values entered at that point until `VALUES == n`, after which you can average over n values each time (the number used for the running total). Thus for $n = 4$ and the sequence 1,2,3,4,5,6,7,8, the 'startup' output is

$$\begin{array}{l}
 1.00, 1.50, 2.00, 2.50, 3.50, 4.50, 5.50, 6.50 \\
 \underbrace{\hspace{1.5cm}} \quad 1 / 1 = 1.00 \\
 \underbrace{\hspace{1.5cm}} \quad (1+2) / 2 = 1.50 \\
 \underbrace{\hspace{1.5cm}} \quad (1+2+3) / 3 = 2.00 \\
 \underbrace{\hspace{1.5cm}} \quad 1+2+3+4 / 4 = 2.50 \\
 \underbrace{\hspace{1.5cm}} \quad 2+3+4+5 / 4 = 3.50
 \end{array}$$

- b. You will require one loop to control the overall execution of your code. You may assume that your code executes until terminated by pressing Ctl-C, hence you should NOT check for an explicit termination command or character. So your outer pseudocode loop will look something like this:

```
while(true) {
    ...
}
```

Before the loop, your pseudocode should prompt the user for any values needed (e.g. the value of n). Inside this loop you'll need to figure out the sequence of commands needed to implement the code correctly, based on the preceding discussion.

Note: there are many ways to approach this problem, and many possible implementations. There are perhaps just two or three 'optimum' solutions, and you should strive for these. What is 'optimum'? If your finished C program results in more than 20 lines of code, then your code is certainly less than optimum: it can be tightened up. On the other hand, if your C code is 11 lines long or less, then you've very probably compressed your code beyond the limits of human comprehensibility: consider fleshing things out in a bit more detail.

- c. When you have written your pseudocode, show it to the professor for feedback before proceeding to Part III of this lab.

III. Implement your pseudocode in C and demonstrate it

- a. Implement your pseudocode in C and test it. Since your code needs to 'wrap' around the array successfully, be sure to enter more than 10 values as part of your testing.

When the code starts up, the average will be over the number of values entered, up until n values; test that your code does the math correctly, averaging over the right number of values, especially during the loading stage of the operation;

Sample Output:

- a. See sample output, below
- b. You should display the information stored in the entire buffer, i.e. all N values, not just the n values entered. (For this, the display of the contents, you can use a *for* loop)
- c. When your code has been properly debugged, demonstrate it to the professor, and be prepared to answer questions about your program.

HINTS:

- (1) NEW advances each time a number is entered, so NEW *always* steps forward; OLD only moves after $VALUES == n$. But each must wrap back to 0 when its value is equal to the array size, N .
- (2) If you code this correct, you should not need to code the 'startup' portion as a separate piece of code.

Marking

Requirements	Mark
Pseudocode/Flowchart includes all features of the program described above	12
Pseudocode has been translated into optimized C Code	10
Student has demonstrated their code in the VSC debugger	3
Total:	25

Enter the number of values to be averaged over (N): 4									
Enter a value: 5									
Buffer contents:	5	0	0	0	0	0	0	0	0
Number of values entered:	1	Average over: 1				Average: 5.00			
Enter a value: 3									
Buffer contents:	5	3	0	0	0	0	0	0	0
Number of values entered:	2	Average over: 2				Average: 4.00			
Enter a value: 1									
Buffer contents:	5	3	1	0	0	0	0	0	0
Number of values entered:	3	Average over: 3				Average: 3.00			
Enter a value: -5									
Buffer contents:	5	3	1	-5	0	0	0	0	0
Number of values entered:	4	Average over: 4				Average: 1.00			
Enter a value: 16									
Buffer contents:	5	3	1	-5	16	0	0	0	0
Number of values entered:	5	Average over: 4				Average: 3.75			