ECE 220 – Computer Programming for Engineering – Winter 2017

Laboratory No. 3: "Divide and Conquer"

Objective

The goal of this lab is to make you familiar with functions in C programs, and to gain understanding how buffers work. Again, the lab will give you a chance to learn and use debugging – a special process that allows you to execute program line by line and see values of all variables. The program called debugger is a part of Xcode, Eclipse, and Visual Studio.
This lab could be done in groups of two.

Submission

Demonstrate your program to a TA of your section. It can happen during the same lab or the following lab (your section). **The first hour of the next lab** will dedicated to this purpose. For the rest of the lab, you should focus on the lab 3.

Problem Specification

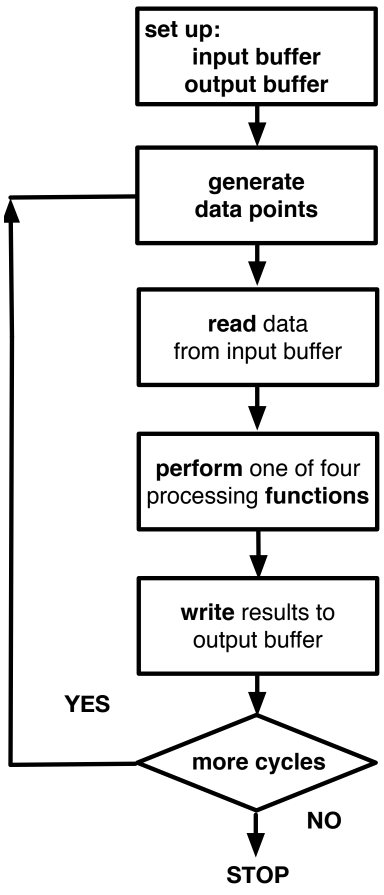
You are asked to write a C program that reads contents of an input buffer, processes the read data, and stores the results in an output buffer. Each of these actions is performed by a different function. Additionally, this program should also generate data and put it in the input buffer (you are simulating a process of filling out an input buffer with data)

In reality, a program you are about to write runs in an infinite loop constantly checking, processing, and outputting data. However, to make it simpler and doable the user will control how many loops of reading an input buffer should execute, i.e., the program should ask the user after each loop if he/she wants to continue.

A very simple flowchart of the program is shown besides.
Pre-lab task:

1. Develop a detailed flowchart using this one as a starting point.
2. Execute (manually) the algorithm.

Please submit the PRE_LAB (.pdf file) with the results of the task 2 (execution of the algorithm) using **eClass**. The deadline is **the noon of the lab day of your section** (e.g., for Section H32, the deadline is Mar 1st, 12:00PM).


set up:
input buffer
output buffer
generate data points
read data from input buffer
perform one of four processing functions
write results to output buffer
more cycles
YES
NO
STOP

As mentioned earlier your program should **generate** data. We want this to be done differently every time the program executes the loop (see flowchart, Page 1). In order to do this we will use a random generation process. It allows us to mimic randomness of a size of data that should to be read from the input buffer, and a type of function that should be performed on this data. Multiple random numbers should be generated to accomplish this.

A simple algorithm **DATA_INIT** that describe the above-mentioned process is:

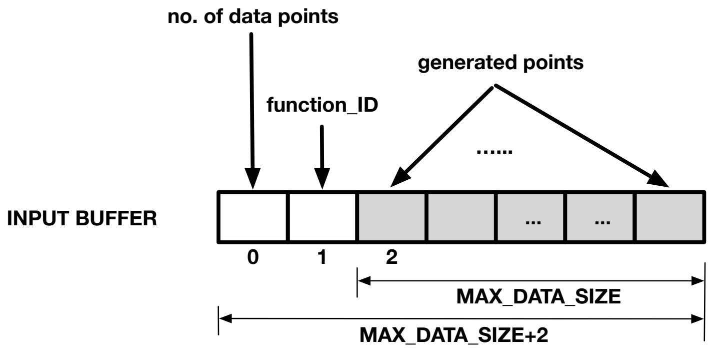
Step1: generate a number that will indicate how many data points of the input buffer should be created (generated) later; the range for this number should from 2 (minimum two data points should be read from the input buffer) to a constant (define this as MAX_DATA_SIZE, so it can be easily changed at any time); let us call this number `no_of_data_points`;

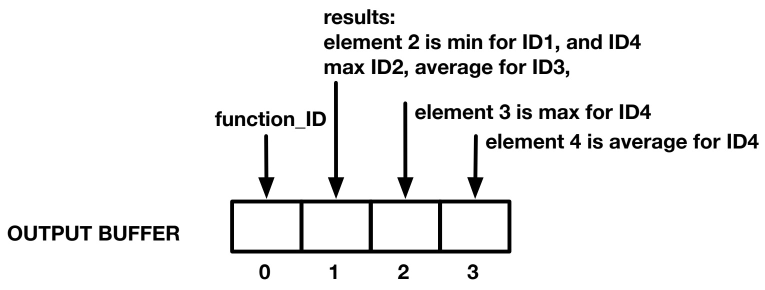
Step2: generate `no_of_data_points` numbers in the range from 0 to 1; each number should be put into a single byte of the input buffer; (useful information, page 6)

Step3: generate a number in the range from 1 to 4, it will represent an ID of a function that will process data after it is read from the input buffer; each number indicates a different operation; let us call this number `function_ID`;

The input buffer should have the size of MAX_DATA_SIZE+2: one byte for `no_of_data_points`, one byte for `function_ID`; and MAX_DATA_SIZE bytes for data;

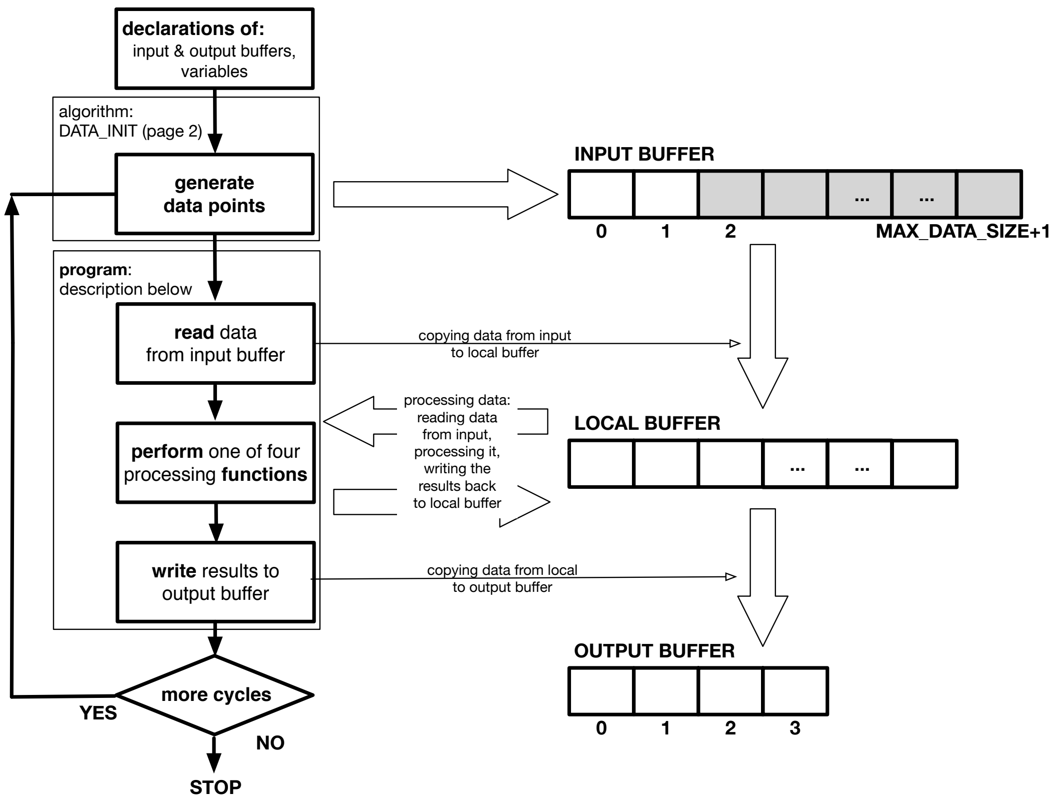
The output buffer should have the size of 4 should have the format as below.




results:
element 2 is min for ID1, and ID4
max ID2, average for ID3,
function_ID
element 3 is max for ID4
element 4 is average for ID4
OUTPUT BUFFER
0
1
2
3

A program you suppose to write, should read data from the input buffer, perform operations on this data – identified by the read `function_ID`, and write the results into the output buffer.

In order to make the time of reading and writing as minimal as possible (in reality, other process would want access to these buffers too), it is quite common to have another buffer – called a local buffer. This buffer is "fully owned" by the program. The data read from the input buffer are store there, all calculations are performed on the content of this buffer, and the results are copied from this buffer to the output buffer. The whole scenario is illustrated in the figure below. NOTE: the organization of the local buffer, i.e., its name, size are decided by you.


declarations of: input & output buffers, variables
algorithm: DATA_INIT (page 2)
generate data points
program: description below
read data from input buffer
perform one of four processing functions
write results to output buffer
more cycles
YES
NO
STOP
INPUT BUFFER
0
1
2
...
MAX_DATA_SIZE+1
copying data from input to local buffer
processing data: reading data from input, processing it, writing the results back to local buffer
LOCAL BUFFER
copying data from local to output buffer
OUTPUT BUFFER
0
1
2
3

The individual steps of the program are described below (each of them is a single function).

The **reading** of data from the input buffer involves:

- reading a number from the first element of the input buffer, this number represents a number of data points to read (`no_of_data_points`)
- reading a number from the second element of the input buffer – this is ID of a processing function (`function_ID`)
- reading all data points

the ID of a data processing function and all points are written into the local buffer.

The **processing** of data involves:

- based on the read function ID (`function_ID`) specific calculations are performed, four different processing can take place:
- if ID is 1: minimum of data points is determined
- if ID is 2: maximum of data points is determined
- if ID is 3: average of data points is determined
- if ID is 4: minimum, maximum and average of data points are determined

the results should be placed into a local buffer (the data points can be overwritten, we do not need them anymore)

The **writing** of the results into the output buffer involves:

- writing `function_ID` into the output buffer
- writing the result, and this depends what functions it was:
  - if ID is 1: only one value is written, i.e., the minimum
  - if ID is 2: only one value is written, i.e., the maximum
  - if ID is 3: only one value is written, i.e., the average
  - if ID is 4: three values are written, i.e., min, max and average

Additionally, this function should display the results on the screen.

In each new cycle, you have to clear the used buffers before using them again, you can use 0.0 for it.

Important:
It is mandatory to follow the signature of the functions in the program template.
It is also mandatory do print the content of the input, local and output buffers at the end of each function.

Implementation Details
Design your program first!!! Draw a diagram or a flowchart of the program. Use the provided flowchart as a starting point.

The template of the program and prototypes of the functions are below.

Hints
1. Start with a simple version of the program: at the beginning just generation of data and filling out the input buffer, then reading the buffer, and so on … Such an approach is called "incremental development".
2. DO NOT DO EVERYTHING with the FIRST ATTEMPT.
3. Compile your program after adding few statements. DO NOT wait till the end!!!
4. **Use debugger**!!! It is the best tool to verify correctness of your program.

Template for your program

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h> /* to initialize your random generator */

#define MAX_DATA_SIZE 10
#define OUTPUT_BUFF_MAX_SIZE 4


// a function for generating data size, function ID and
// all data, and putting all of it into input_buffer
void generate_data(float input_buffer[]);

// a function for copying the content of input_buffer into
// local buffer
void reading(float input_buffer[], float local_buffer[]);

// a function for processing the content of local buffer;
// it reads function_ID and number of data points from
// local_buffer, and saves the results also into local_buffer
void processing(float local_buffer[]);

// a function for copying the content of local_buffer to the
// output_buffer (according to the patter explained earlier)
void writing(float local_buffer[], float output_buffer[]);



int main(void) {

float input_buffer[MAX_DATA_SIZE+2] = {0};
float output_buffer[OUTPUT_BUFF_MAX_SIZE] = {0};

float local_buffer[MAX_DATA_SIZE+2] = {0};

/* body of your main function */

}


/* place for your definitions of functions*/
```

Useful information

Random number generator:
C language has a special function called `rand()` to generate random numbers. The `rand()` returns an integer value from the range `0` to `RAND_MAX` (whatever this number is – it depends on the system, in **Xcode** and **Eclipse** it is `2147483647`, and in **Visual Studio** it is `32767`). You can "make" a number from `0` to `1` in a very simple way:

```
value = (float)rand()/RAND_MAX;
```

Note, we need (`float`) to make the result of `rand()` type `float`, otherwise we divide `int` by `int` what results in zero (if numerator is larger than denominator). There is another function that should be call at the beginning of the program – it is

```
srand(time(NULL));
```

This function initializes a random generator – it requires a number called seed for an initialization. Very often, a time function `time(NULL)` is used to initialize the generator with a different value every time you run the program.
If you want to start generator always from the same number – so generated numbers are the same every time you run your program (maybe for testing purposes) you can put as a seed any integer number you want. For example:

```
srand(17);
```

A simple code using a random generator is shown below (NOTE: there are two header files required):

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main(void)
{
 float value_float = 0.0;
 int value_int = 0;
 …
 srand(time(NULL));
 …
 value_float = 0.4 + (1 - 0.4) *(float)rand()/RAND_MAX;
 value_int = 5 + rand() % (11-5+1)
 …
}
```

This program generates a float number in the range 0.4 to 1.0, and an integer number in the range 5 to 11.
In the case you have problems with the `rand()` function, you can use a random generator algorithm. An implementation of one of them and an example of its usage can be found on eClass.

LAB 03: Marking Scheme

Student Name: ________________ **ID:** ________________

Student Name: ________________ **ID:** ________________

Marking Scheme

This assignment is worth 6% of your final mark. A total number of points you can obtain is 100. The marking of the lab is done according to the following schema:

TASK	POINTS
Flowchart	
Pre-lab Task: Quality of flowchart (flowchart's ability to convey the main steps of the program, easiness of its understanding).	/5
Proper content of buffers [2nd PRE_LAB Task, file in **eClass**]	/5
Execution of your program	
Function for generating input data (randomness, filling of input buffer)	/15
Function for reading input data (reading all data, storing read data, passing values)	/15
Function for processing data (calculations, results, storing them)	/15
Function for writing data to output buffer (writing all required data)	/15
Asking for more cycles and displaying the results	/5
Subtotal	**/65 points**
Quality of code (easiness to read/comprehend your program)	
Naming and usage of variables	/10
Design (logic structure)	/10
Successful termination of the running program	/10
Documentation (commenting)	/5
Subtotal	**/35 points**
Total	**/100 points**