ECE 220 – Computer Programming for Engineering – Winter 2017

Laboratory No. 4:
"Modular and Tailored"

Objective

The goal of this lab is to make you familiar with writing modular C programs, pointers, and memory management.

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.
If you are not able to finish your program and/or demonstrate it at the beginning of the 5th lab, you are able to submit it during the lab on the following day(s) (any section). The penalty is 10% for each day of delay. If your section is on Thursday (and there is not lab on Friday, and of course Saturday and Sunday) then to avoid losing too many points, you can send a code (only once) to lab instructors, i.e., Jinbo (jinbo@ualberta.ca) or Ning (ncao@ualberta.ca) or Syed Tauhid Zuhori (zuhori@ualberta.ca).

Pre-Lab Quiz

There is a quiz for this lab on the eClass. You are asked to do it before the lab starts. The lab instructor will check your answers at the beginning of the lab.

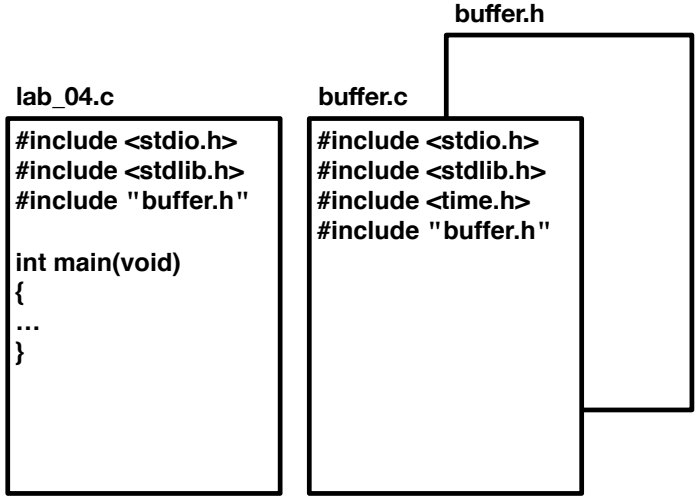
Problem Specification

You work with the C program that reads contents of an input buffer, processes the read data, and stores the results in an output buffer – in other words, you work with the program you wrote in Lab 3.

However, this time your program should be modular, and should use a few memory management functions.

TASK No. 1

The first task is to split your program (from the lab 3) into two modules: **lab_04.c** and **buffer.c**. The program should have a total of three files (on the side).



The file **lab_04.c** should contain only the `main()` function. Its template is shown below.

```
#include <stdio.h>
#include <stdlib.h>
#include "buffer.h"
#define MAX_DATA_SIZE 10

void processing(float local_buffer[]);

int main(void) {

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

}

void processing(float local_buffer[]) {

}
```

The file **buffer.c** should contain the following functions (of course, the names of your functions could be different):

- `void generate_data(…)`
- `int reading(…)`
- `void writing(…)`
- `void display_buffer(…)` // a new function to display the content of the output buffer

The template for buffer.c is below.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h> /* to initialize your random generator */
#include "buffer.h"
#define MAX_DATA_SIZE 10
#define OUTPUT_BUFF_MAX_SIZE 4

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

generate_data() {

}

void reading(float local _buffer[]) {

}

void writing(float local_buffer[]) {

}

void display_buffer() {

}
```

The file **buffer.h** (header) should contain prototypes of all functions. The rest of your program should be in **lab_04.c**.

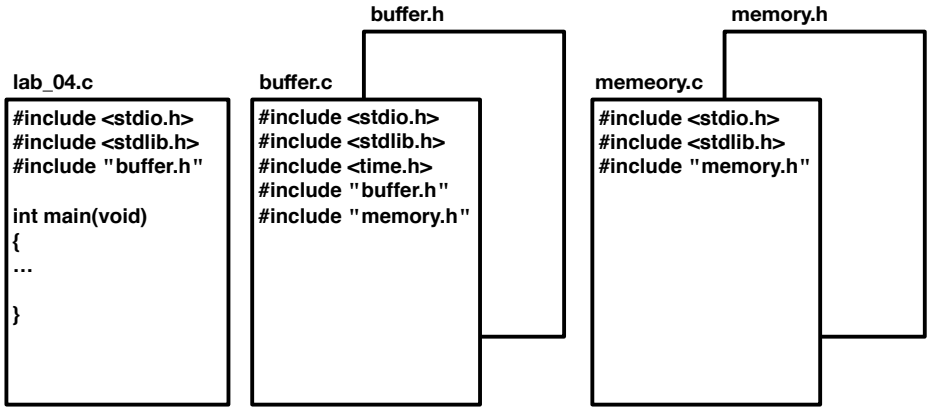
Once you split your program and have these three files, run it and make sure that it performs correctly.

TASK No. 2

The second task is to add memory management module to your 'new' program and take advantage of the memory management functions, i.e., you can ask for exact amount of memory needed for both input and output buffers, as well as your local buffer.

The first activity is to download two files: **memory.c** and **memory.h** from the eClass. Familiarize with the two functions: `my_malloc` and `my_free`. There is something special about them: they allow you to track how many bytes you have requested/freed from the system. Please analyze both functions, and try to figure out how it is done.

IMPORTANT: Please read and understand all material provided in the section **Useful information (memory management)**, page 6 of this manual.
At this point you can add memory management part to your program, and have all these files:



Add `#include "memory.h"` to your **buffer.c** file. Run your program, and be sure it works.

The second activity is to utilize both functions: `my_malloc` and `my_free`. Use `my_malloc` every time when you ask the system (operating) for memory – to create a buffer, and use `my_free` every time you return the memory to the system. You should have three buffers in your program (all of them should have dynamically allocated memory):

- **input buffer** (used to store generated data), its size it determined by a random generator; this buffer should be created (memory allocated) in the function `generation()`, and freed in the function `reading()`; the size of this buffer is

equal to a number of data points plus two, and a number of data points is randomly generated and should be in the range form 2 to BUFF_SIZE;
- **local buffer** (processing buffer) used to store data read from the input buffer, used during data processing; this buffer should be created (memory allocated) in the function `reading()`, and freed in the function `writing()`;
- **output buffer** (used to store results); this buffer should be created (memory allocated) in the function `writing()`, and freed in the function `display_buffer()`.

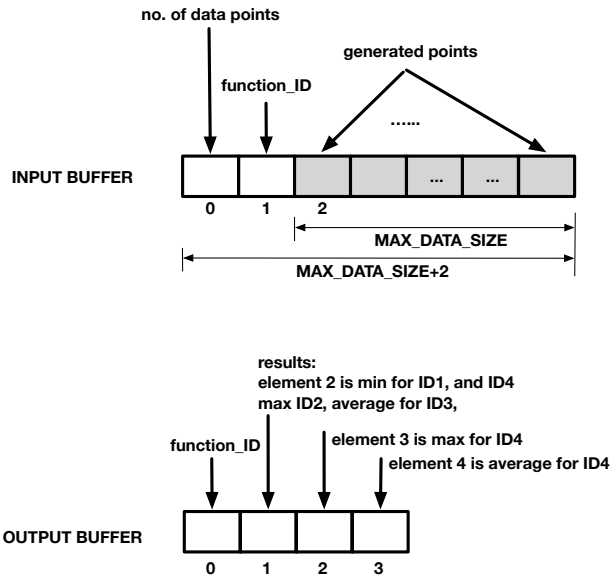
As in the Lab 03 and Task no. 1, the whole program should include the following parts (functions) called from the loop of the `main()` function:

- generate data
- read data
- process data
- write data
- display buffer

The loop should be executed as long as the user indicates otherwise.

IMPORTANT: every time you use the `my_malloc()` and `my_free()` functions you will see a number of bytes that you "own" to the system. To be sure that you do not have a memory leakage – it means you return less memory than you asked for, and eventually you accumulated memory. You will not have memory leakage when at the end of each loop you have zero bytes own to the system.

You can use the same (similar) buffers as in the lab 3. Their format is shown below.



The file **lab_04.c** should contain two functions: `main()` and `processing(…)`.

The template is shown below.

```
#include <stdio.h>
#include <stdlib.h>
#include "buffer.h"

void processing(float local_buffer[]);

int main(void) {

   float *local_buffer;

…

}

void processing(float local_buffer[]) {
…
}
```

The template for **buffer.c** is below.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h> /* to initialize your random generator */
#define MAX_DATA_SIZE 10
#include "buffer.h"
#include "memory.h"

float *input_buffer;
float *output_buffer;

void generate_data() {
…
}

float *reading(){
// see NOTE 3 (page 9 of this document) for an explanation
…
}

void writing(float local_buffer[]) {
…
}

void display_buffer() {
…
}
```

Implementation Details

Design your program first !!!! Draw a diagram or a flowchart of the program. Be clear when you allocate memory for buffers and where you free it. Show it to a TA before you proceed further.

Hints

1. Start with a simple version of the program – task no. 1, and then task no. 2, (the first and second activities).
   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.

Useful information (memory management)

Analyze the memory management program functions. Become familiar with them, and write the following simple program which uses functions `my_malloc()` and `my_free()` provided in `memory.c` and `memory.h`.

```
#include <stdio.h>
#include "memory.h"

int main(void) {

    int *p;
    p = (int *) my_malloc (sizeof(int));
    if (p == NULL) {
        printf("Memory has not been allocated");
        exit(1);
    }

    *p = 10;

    my_free(p);
    return 0;
}
```

The process of doing is as follows:

- create a project under Xcode or Visual Studio (name it anyway you want, for example `my_own()`)
- create an additional C source file named `memory.c` in the project, copy the contents from the provided file `memory.c` into your `memory.c`

- create a header file names `memory.h` in your project, copy the contents of the provided file `memory.h` into your file `memory.h`

Look at the section regarding Dynamic Memory Allocation (below) how it should be done. NOTE: the examples there contain reference to `malloc()` and `free()`.
In our case we use equivalent functions `my_malloc()` and `my_free()`. Try to understand way.

Show it to a TA before you proceed further.

DYNAMIC MEMORY ALLOCATION:
Any time we declare a variable, the system provides our program with memory, and each variable is stored in this memory allocated to the program. This means that when we start our program, a chunk of memory is allocated to the program, and all our variables have their place there.
However, quite often we do not know how many variables we need, or how large some arrays are. To overcome this issue C allows us to dynamically allocate memory, i.e., to ask the operating system for memory needed to "create" a variable or/and array.

All this is possible thanks to pointers and two C functions from `stdlib.h`: `malloc()` and `free()`.

When we declare a pointer:

```
int *p;
```

C allocates space for the pointer (4 bytes), and the memory location occupied by `p` is not initialized. When `p` is not initialized its value can point to any memory location, and this can cause a run-time error and the program crashes. So, unless we assign to `p` a value representing an address of some variable, we cannot really use it.
But, this gives us a chance to ask for memory and assign address of this memory to `p`, and use `p` (or rather memory pointed by `p`) to store some data. This can be done in the following way.

The first step is to ask for memory. Here, we use a function `malloc()` which takes as a parameter a number of bytes we need. For example:

```
p = (int *) malloc (sizeof(int));
```

Let's take a closer look at this statement:
- `sizeof()` gives us a number of bytes needed to store data when we use specific type, in our case – integer
- `malloc()` gives us back an address to a block of memory (4 bytes in our case – the size of a single integer), this address is of a type `void *` (`malloc()` "does not" know what we want to use this memory for – it "tries" to be flexible)

- `(int *)` indicates that the memory block allocated to us will be used to store an integer value (we are casting `int *` on the original `void *)`

Now we can use this newly allocated memory to store data:

```
*p = 23;
```

Once we are done, we have to return the previously allocated memory back to the system. We do this using a function called `free()` in the following way:

```
free(p);
```

NOTE 1:
It can happen that an operating system cannot allocate memory; therefore it is always good to follow the following steps:
- initialize pointer to NULL (zero address)

```
int *p = NULL;
```

- check if returned address from `malloc()` is not NULL (system did its work and assign a memory block)

```
p = (int *) malloc (sizeof(int));
if (p == NULL) {
   printf("Memory has not been allocated");
   exit(1);
}
```

NOTE 2:
You need to be aware that you can allocate memory block inside a function, and return pointer to this memory to the main program, i.e., you can do the following:

```
char * copy_str(char *p)
{
 char *q = (char *)malloc(strlen(p)+1);

 strcpy(q, p);

 return q;
}
```

and this function can be called from the following `main()`:

```
int main(void)
{
  char *a = NULL;
  a = copy_str("Hi there …");
  free(a);
}
```

However, the **FOLLOWING FUNCTION** is **NOT** good – it will lead to disastrous results:

```
char * copystr(char *p) {

 char q[100];

 strcpy(q, p);

 return q;
}
```

The array q is a local variable, and it disappears when function is done. So, returning the pointer to it, means returning the pointer to the memory that is not yours.

NOTE 3:
A local buffer should be created inside the function `reading()`.

The template for this function is:

```
float * reading(){

    float *buffer;

…
    buffer = (float *)my_malloc(…);
…

    return buffer;
}
```

Such a template will allow you to allocate memory inside the function, and use this memory outside the function. **For explanation,** see **NOTE 2** above.

LAB 04: 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
Pre-Lab Quiz	
Quiz (1.5 pts for each question, 1pt for doing the quiz)	/10
Execution of your program	
Proper splitting of the program (five files: three .c and two .h)	/15
Proper usage of memory management functions (no memory leakage)	/20
Main loop (in lab_04.c file) (nice and clean loop)	/15
Asking for more cycles and displaying the results	/10
Subtotal	**/70 points**
Quality of code (easiness to read/comprehend your program)	
Naming and usage of variables	/10
Documentation (commenting)	/10
Layout (indentation/spacing)	/10
Subtotal	**/30 points**
Total	**/100 points**