ENSF 337: Programming Fundamentals for Software and Computer
Lab 6, Fall 2018 – Thursday October 18h

M. Moussavi

Department of Electrical & Computer Engineering
University of Calgary

Objective:

The main objective of this lab is to practice the subjects of dynamic allocation of memory, and file I/O manipulation in C.

Due Date:

- **In-lab exercises:** that must be handed in on paper **by the end of your scheduled lab period**, in the hand-in boxes. The hand-in boxes are on the second floor of the ICT building, in the hallway on the west side of the building.

- **Post-lab exercises:** must be submitted electronically using the D2L Dropbox feature. All of your work should be in a single PDF file that is easy for your TA to read and mark.
  Due date and times for post-lab exercises is: Thursday Oct 25, before 2:00 PM.

Important Notes:

- Some post-lab exercises may ask you to draw a diagram that most of the students prefer to hand-draw them. In these cases you need to **scan** your diagram with an appropriate device and insert the scanned picture of your diagram into your PDF file (the post-lab report).

- 20% marks will be deducted from the assignments handed in up to 24 hours after the due date. It means if your mark is X out of Y, you will only gain 0.8 times X. There will be no credit for assignments turned in later than 24 hours after the due dates; they will be returned unmarked.

Marking scheme: 37 marks

- Exercise A (in-lab): 8 marks
- Exercise B (in-lab): 8 marks
- Exercise C (Post-lab): 5 marks
- Exercise D (Post-lab): 12 marks
- Exercise E (Post-lab): 4 marks

Exercise A: Allocation of Memory on the Heap

This is an in-lab exercise

Read This First:

Before doing this exercise, you should review lecture notes or a C textbook to be sure you know exactly how C library function `malloc` works.

What to Do:

Download the file `lab6exe_A.c` and `lab6exe_A.h` from D2L. Read these files carefully and draw a memory diagram for point one, assuming that all the calls to the library function `malloc` succeed.

What to Submit:

As part of your in-lab report submit your AR diagram.

Exercise B: Draw AR Diagrams

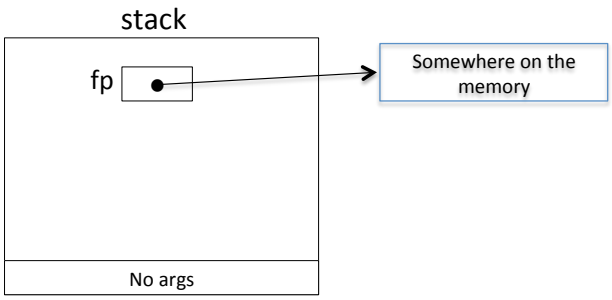
This is an in-lab exercise

What to Do:

Download file `lab6exe_B.c, lab6exe_B.h,` and `lab6exe_B.txt,` from D2L. This exercise receives its input from text file, `lab6EXB.txt`, and populates an array of structure called Bits_Pattern with the available data in this file. Study the content of the downloaded files carefully to understand what the program does. Then draw a memory when the program reaches point one **for the second time**.

Note: in your AR diagram you don't need to show where exactly a FILE pointer points. Just mention it points to "somewhere on the memory" as following figure shows:

```
FILE * fp = fopen("a_file_name", "r");
```



What to Submit:

As part of your in-lab report submit your AR diagram.

Exercise C: Writing into a Text File:

This is a post-lab exercise

What to Do:

Download file `lab6exe_C.c, lab6exe_C.h`, and `lab6exe_C.txt`, from D2L. If you read the given files carefully you will notice that this program reads the content of the file `lab6exe_C.txt` into an array of integers that is declared within the body of a structure called `IntVector`. Then, the program displays the stored values in the array on the screen in a single column format. Your task in this exercise is to complete the definition of the function called `display_multiple_column`. Please see the function interface comment for more details.

What to Submit:

As part of your post-lab report submit the definition of your function `display_multiple_column`, and content of the produced text called `lab6exe_C_output.txt` (you don't need to submit the file itself, just copy and paste the content of the file into your post-lab report).

Exercise D: String Manipulation Using Dynamic Allocation

This is a post lab exercise.

The main objective of this exercise is to give you the opportunity to practice dynamic allocation of memory for the purpose of string operations such as copying a string into another string, appending a string to the end of another string, or truncating the size of a string. The other objective of this exercise is to provide you with more practice on C structure type, and finally to take a very small step towards the concept of data abstraction.

Read This First:

Creating a kind of wrapper data type that encapsulates several related data is a common practice to build data structures such as vectors, string, linked list, trees in object oriented programming languages such as C++. We will discuss the proper way of applying these concepts later in the C++ part of the course.

In this exercise, we define a structure type called String that simply contains two pieces of related data. First one is a pointer that is supposed to point to a dynamically allocated array of characters, used as an storage for a null-terminated c-string. Second one is an integer number that represents the length of the stored string in the first data member (count of character up to and excluding the '\0').

Here is the definition of this struct:

```
typedef struct String{
    char *dynameic_storage;
    int length;
} String;
```

What To Do:

Download the files `lab6exe_D.c` and `lab6exe_D.h` from D2L. If you read these files carefully, you will notice that the existing code contains the following functions plus main and a three other functions for testing operations: copy, append, and truncate:

`void create_empty_string (String *str);`
Creates an empty string that its dynamically allocated storage has onle one element holding a '\0' and length of zero.

`void String_cpy(String *dest, const char* src);`
Copies a c-string from src into the dynamically allocated storage of String object that dest points to.

`void String_copy(String *dest, const String* source);`
Copies a c-string from source into the dynamically allocated storage of String object that dest points to.

First you should compile and run the program and pay attention that how functions `String_cpy` and `String_copy` work. Please read the comments in the file `lab6exe_D.c`, for each call to the function `display_String`. The comments tells you what this function is expected to display if functions `String_cpy` and `String_copy` work properly. Here is a snapshot of the program output:

```
Testing String_cpy and String_copy started:
String is empty      0
String is empty      0
String is empty      0
String is empty      0
William Shakespeare      19
Aaron was Here!!!!      18
But now he is in Italy      22
String is empty      0
String is empty      0
But now he is in Italy      22
String is empty      0

Testing String_cpy and String_copy finished...
```

To better understand how this program works, you can also draw AR diagrams for the points that program reaches at the end to the functions: `create_empty_string`, `String_cpy`, and `String_copy`. **These diagrams are not for marking and you shouldn't submit them as part of your post-lab report.**

Your next job in this exercise is to write the implementation of two missing functions `String_truncate` and String_append, according to the given interface comments in the file `Lab6exe_D.h`. You are advised to write and test function `String_truncate` first, and then once this function works with no errors, start working and testing the other function, `String_append`.

To complete this task please take exactly the following steps:

1. In the main function, change the conditional-compilation-directive for the call to test_copying from `#if 1` to `#if 0`.
2. Read carefully the function interface comment for function `String_truncate` in the header file `Lab6-ExD.h` and write the implementation of this function.
3. Change the conditional-compilation-directive for the call to `test_truncating` from `#if 0` to `#if 1`.
4. Compile and run the program and make sure the test results matches as comments in the function `test_truncating` states for each call to the function `display_String`.
5. If you function `String_truncate` works fine (no errors), repeat steps 2 to 4 for function `test_appending`.

What To Submit:

In your post-lab PDF file, include listings of the final versions of your `Lab6exe_D.c`, and the output of the final version of the program, showing the results of calls to the functions `test_truncating`, and `test_appending`.

Exercise E: Fixing a Defective C Program

This is a post-lab lab exercise.

Read This First – Common Mistake With Dynamic Allocation of Memory

Developing programs that use dynamic allocation of memory needs special attention and additional care to avoid runtime errors. Although dynamic allocation of memory may provide some flexibility to write programs that manage their required memory spaces more efficiently, but on the other hands it puts more responsibilities on the programmer's shoulder to avoid malicious errors.

Here is the list of some of the common and possible mistakes that programmers can make when allocating memory dynamically in a C program:

1. Loosing the track of an allocated memory, which results in memory leak.
2. Allocating memory in a function that doesn't return a reference to the calling function.
3. Loosing track of the begging of the allocated memory.
4. Trying to free a portion of the allocated memory. For example starting to release memory from third element of a dynamically allocated block to the end of the block.

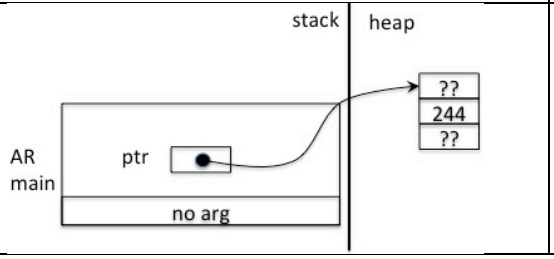
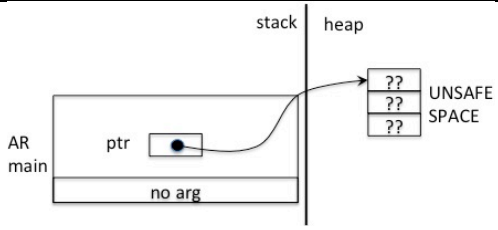
Note: Please notice that even dynamic allocation of memory in a program may allow more efficient use of memory spaces, but on the other hand it will involve addition overhead for extra time for the operations such as searching for available space, etc. This may not be a major concern for many programs, as most of the modern computers have very fast processors hence the overhead time may not be an issue.

Also Important to Notice:

- You can only use C library function **free** to deallocate a block of memory that is allocated by one of the C library functions: malloc, calloc or realloc.

- Pointer that is passed to the function free() MUST be pointing to the first byte of the allocated memory, otherwise it causes "*undefined behavior"*.
- Notice that function free does not change the of value the pointer that is passed to it. In other words the pointer that is passed to free, still points to the same (now invalid) location.
  For the purpose of AR diagrams in ENCM 339, the de-allocated memory must be labeled with "UNSAFE SPACE" phrase. Please see the following code and diagrams at points one and two:

```
int main(void) {
   int* ptr = malloc (3 * sizeof(int));
   ptr[1] = 244;
   // Point 1
   free(ptr);
   // Point 2
   return 0;
}
```

Point 1	Point 2
	

What to Do:

Download file `Lab6exe_E.c` from D2L, and study the code to find out how the program works. If you compile and run the program you will notice that it doesn't do anything useful -- only prints the following messages:

```
Program started...
Program terminated...
```

If you fix the defective parts of the code, which needs some changes to the function `create_array` and some changes to the `main` function, the program output should be as follows:

```
Program started...
100.000000
200.000000
300.000000
400.000000
500.000000
600.000000
700.000000
800.000000
900.000000
1000.000000
Program terminated...
```

What To Submit:

Submit your source code (`Lab6exe_E.c`) and the output of your program.