Lab 04: More Memory in C
CSE-031-01 **Points: 20**

Overview

In this lab, we will continue to investigate some of the interesting facts about how C interacts with memory. You will continue to practice the use of pointers to access memory.

You can refer to chapter 5 and 6 of K&R for references on pointers.

Getting Started

Before we begin any activities, create a directory (`Lab_4`) inside the `CSE31` directory we created last week. You will save all your work from this lab here. **Note that all the files shown in green below are the ones you will be submitting for this assignment.**

You must have a clear idea of how to answer the lab activities before leaving lab to receive participation score.

How Type-Casting Works with Memory

From lecture, we have learned that an array is stored in a contiguous memory space. The address of each element of an array is assigned by the computer (operating system).

TPS (Think-Pair-Share) activity 1: Discuss questions 1 – 10 (25 minutes) while paired with your classmates assigned by your TA (you will be assigned to groups of 3-4 students) and record your answers in a text file named `tpsAnswers.txt` under a section labelled "TPS 1" (***you will continue to use this file to record your answers to all the TPS questions that follow in the lab handout***):

1. Open `MemCast.c`, compile and run the program. What do you expect the program to print? (`%x` in `printf` allows an integer to be printed in Hex format).
2. ***Before changing the code***, what do you expect the program to print if you print `four_ints[0]` again at the end of the program?
3. Insert a print statement to output `four_ints[0]` at the end of the program and verify your answer from (2).
4. Now add a print statement to the program so it will print out `four_ints[1]`. What does it print? Are you surprised (or lost) by the results?
5. Let us study the code carefully and investigate what happened. No, the memory did not go crazy.
   a. How many array(s) were allocated in this program?
   b. Are `four_ints` and `four_c` pointing to the same location?
   c. Verify your answer of (b) by printing out the values of `four_ints` and `four_c`.
6. Write a loop to print out the addresses and values (in Hex) of all the elements of `four_ints`. What is the difference in addresses between two consecutive elements? Discuss what this difference means.
7. Use a piece of paper to draw the layout of the array ***horizontally*** so that the smallest address begins from the ***RIGHT-hand-side***. Indicate the ***address*** and ***value*** of each element based on the results of (6). You can draw it digitally.
8. Now, write a loop to print out the same information for `four_c` as you did in (6). What is the difference in addresses between two consecutive elements? Discuss what this difference means.
9. Use the same piece of paper (or file) from (7) to draw a similar structure for `four_c`.
10. By comparing the layout of the array pointed by the two pointers, what do you observe in terms of how C accesses memory when the index of an array is incremented?

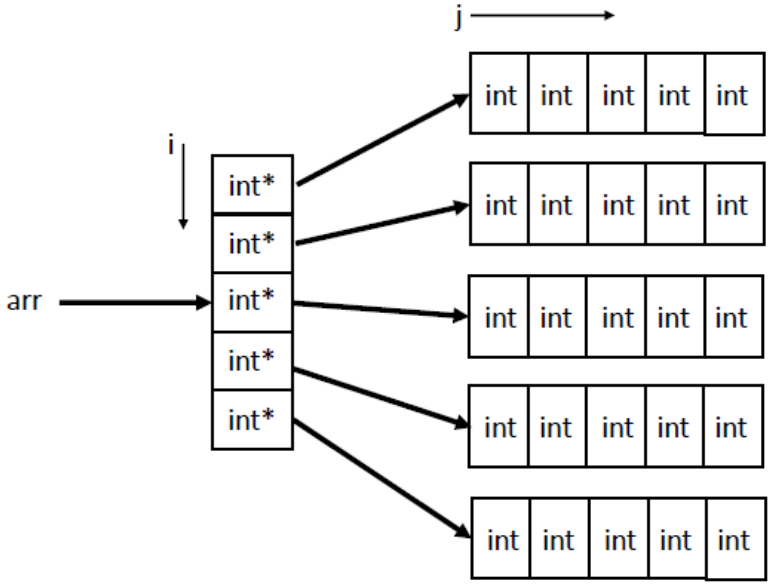
Your TA will "invite" one of you randomly after the activity to share what you have discussed.

2D Arrays with Malloc

In C, a 1-dimensional array can be declared as a pointer pointed to a dynamically allocated memory location:

```
int* array = (int*)malloc(n * sizeof(int)); // n is the size of array
```

We can also use a double pointer (**) to construct a 2D array. In fact, a 2D array is just multiple rows of 1D arrays. You can also view it as an array of arrays as shown below:



In the example above, each row is an array of `int`. Since each row is an array of `int`, we need an `int*` pointing to each row. As a result, we need to have an array of `int*` to point at different rows of `int`.

TPS activity 2: Discuss questions 1 – 7 (25 minutes) with your TPS partners in your assigned group and record your answers in `tpsAnswers.txt` under a section labelled "TPS 2":

1. Open `Array2D.c`. This program will create a `n x n` array of `int`. Explain what line #8 does.
2. Since every array must be allocated in the memory before it can be used, we need to allocate the rows one by one. To do this, we need to be able to access the pointers from the first array (pointed by `arr`). Assuming `i` is the index of that array, how do we access the `ith` value of the array?
3. Without using array notations (`[]`), insert code to complete ***allocating all the rows and initialize all contents to be*** `0`. Your code should work with different values for `n`. **Hint:** if `j` is the index of each row, how do you access `arr[i][j]` in pointer notation?
4. To verify whether you have created your array correctly, we need a function to print out the array. The function `printArray` has been declared. It takes in both the array to be printed and size of array. Why do we need to pass the size as an argument?
5. Complete `printArray` so it prints out the ***content*** and ***layout*** of an array correctly.
6. Now, let us modify the content of the array. Insert code to make the array into a diagonal matrix that looks like the following (***again, do not limit the size to 5***):

1	0	0	0	0
0	2	0	0	0
0	0	3	0	0
0	0	0	4	0
0	0	0	0	5

7. Call `printArray` to print out your new array and verify result.

Individual Assignment 1: Matrix Multiplication

Now we know how 2D arrays work. Let us put them in practical use (you must not use any array notions `[]`) in this assignment):

1. Open `MatrixMult.c`. and define in `main()` two `n x n` matrices (arrays) using `malloc`.
2. Implement `printArray` function so you can call it to print a 2D array.
3. In `main()`, call `printArray` to print out the 2 arrays you have defined in (1).
4. Implement `matMult` so it multiplies the 2 input arrays and return the resulting array. ***Pay attention to the input arguments and return type***.
5. In `main()`, call `matMult` to multiply the 2 arrays you have defined in (1).
6. In `main()`, call `printArray` to print out the resulting array you receive in (5).
7. You will need to declare any variables that are necessary.

Collaboration

You must credit anyone you worked with in any of the following three different ways:

1. Given help to
2. Gotten help from
3. Collaborated with and worked together

What to hand in

When you are done with this lab assignment, submit all your work through CatCourses.

Before you submit, make sure you have done the following:

- Your code compiles and runs on a Linux machine (without the need for special libraries).
- Attached `MemCast.c`, `Array2D.c`, `MatrixMult.c`, and `tpsAnswers.txt`.
- Filled in your collaborator's name (if any) in the "Comments…" textbox at the submission page.

Also, remember to demonstrate your code to the TA or instructor before the end of the grace period.

Scoring:

- TPS activity 1: 4pts (total)
- TPS activity 2: 4pts (total)
- Individual Assignment 1: 12pts (with demo, otherwise 0)