CS 162 LAB #4 – Arrays, Pointers, and Structs

Each lab will begin with a brief demonstration for the core concepts examined in this lab. As such, this document will not serve to tell you everything in the demo. It is highly encouraged that you ask questions and take notes.

In order to get credit for the lab, you need to be checked off by the end of lab. You can earn a maximum of 3 points for lab work completed outside of lab time, but you must finish the lab before the next lab. For extenuating circumstances, contact your lab TAs and the instructor.

This lab is worth 15 points total. Here's the breakdown:

• Part 1: Worksheet	(5 pts)
• Part 2: TA demo	
• Part 3: Understand Dynamic Memory	(Group work: 3 points)
• Part 4: Two Pointer exercises	(Individual work: 2 points)
• Part 5: Arrays, Structs, Pointers	(Individual work: 5 points)

(5 pts) Part 1: Worksheet

This session will be led by your lab TAs. Please follow their instructions, participate, and complete worksheet 4:

https://classes.engr.oregonstate.edu/eecs/winter2024/cs162-001/labs/WS4.docx (pdf version)

Get the start code

Upload/download the start code for this lab onto `flip.engr.oregonstate.edu`:
wget https://classes.engr.oregonstate.edu/eecs/winter2024/cs162-001/labs/lab4.zip

Part 2: TA demo: 1D array of struct objects

In lecture we've learned how to create 1D/2D dynamic array using functions. We've also learned the concept of structs, which is a user-defined data type. In this part, we will be combining these concepts together. First, your lab TAs will give you a demo on 1D array of struct objects.

First, we define a struct for a Person:

```
struct Person {
      string name;
      int age;
};
```

A `Person` a data type, which has two members, `name` and `age`. Just like other types (`int`, `char`, etc.), we can instantiate variables of type `Person`. Instead of calling them variables, we call them objects. Similarly, we can create an array of `Person` objects, initialize them, and print them out. Please refer to the provided code **`demo.cpp`**, use the comments to help you understand the program.

(3 pts) Part 3: Understand Dynamic Memory

We've learned how 1D static arrays work in C++. For example, if we want to create an array to store 10 integers, we can do:

```
int nums[10];
```

The memory of the array above is created on stack during compile time and can store up to 10 integers. However, what if we don't know how many integers to store until the user inputs it? What size should we use for my array?

```
int nums[???];
```

The following is not supported by all C/C++ compilers and is considered bad practice!!

```
int size;
cin >> size;
int nums [size]; //NEVER DO THIS!!!
```

Here, both `size` and the array `nums` are local variables, and they will be out of scope when a function ends. What if we need that array to keep alive after the function ends? We need dynamically-allocated (runtime) memory! That is, the memory allocated on the heap during runtime. To achieve this, we need `new` operator:

```
int *nums = new int [size]; //new returns an address on the heap
```

(1 pt) Now, form groups of 3 and proceed to implement the following function(s) for creating an array of integers on the heap. The size of the array should be determined by the user input. There are three different ways to achieve this goal, and the function prototypes are provided. Each group member is tasked with implementing one function, but it is essential for everyone to comprehend all three.

```
1. // take an int argument, represent the size of the array,
   // and return the address of the array
   int* create_array1(int size);

2. // take a reference to an int pointer to allocate the array,
   // and an int argument, represent the size of the array
   void create_array2(int *& array, int size);

3. // take the address of an int pointer, dereference it to
   allocate,
   // and an int argument, represent the size of the array
   void create_array3 (int ** array, int size);
```

(1 pt) Next, in your main(), write the function calls to use the functions that you created above.

(1 pt) Lastly, we need to manually free/delete the heap memory allocated to avoid memory leaks. Use delete operator to delete the memory off the heap. **Make sure you set your pointer(s) back to NULL/nullptr**, since it is not supposed to be pointing anywhere anymore.

You can check to see if you have any memory leaks using valgrind.

```
%valgrind program_exe
```

(2 pts) Part 4: Two Pointer Exercises

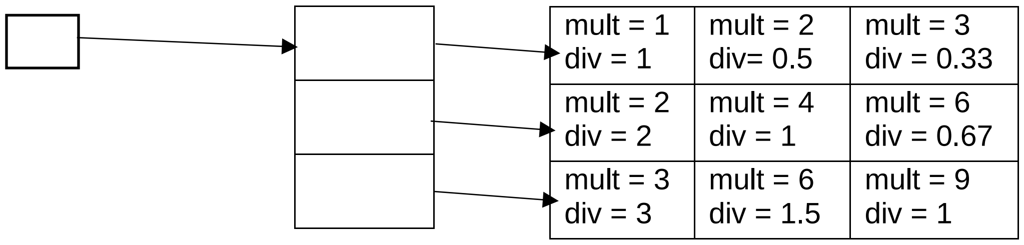
In part 3, you will be working **individually** to finish two pointer exercises. This is to reflect your understanding of pointers.

1. Files needed for this part: `Q1.cpp`, `Q2.cpp`
2. Use the instructions in the comments to finish the programs.

(5 pts) Part 5: Arrays, Structs, Pointers, etc.

Now, let's combine the concepts of pointers, structs, arrays together. Write a program that creates a **dynamic 2D array of `multdiv_entry structs`**. The 2D array will be used to store and print the multiplication and division tables for the values of row and column specified by the user. Here, instead of creating two 2D arrays to store the multiplication and division tables, we will only create one 2D array. Each element of the 2D array is the multdiv_entry type, which is a struct defined by the programmer. See below for the memory layout. Note that the table will start at 1 instead of zero. This prevents us from causing a divide by zero error in the division table! Specifically, follow these steps to create the program.

Example: a 3X3 2D array of `multdiv_entry` structs:



1. Open the file named `multdiv.cpp`. This is the file in which you will write your program.
2. Set your program up to read the number of rows and columns from the user. No error handling is needed here.
3. Understand the provided struct. The following definition of the struct stores a single entry in both the multiplication table and the division table. The data type is `multdiv_entry`
```
        struct multdiv_entry {
                int mult;
                float div;
        };
```
4. Write a program that uses `row`, `col`, and your `struct` to generate and then print the multiplication and division tables. For example, if the user runs your program with row = 5, and col = 5, your program should create a 5 by 5 matrix of structs and assign the multiplication values to the `mult` variable in the struct and the division of the indices to the `div` variable in the struct. Then, print out both tables.

Your program needs to be well modularized with functions. Specifically, you should write and use the following functions:

- `multdiv_entry** create_table(int row, int col);`
  This function should allocate space for a `row` x `col` array of your `struct` and fill it with the appropriate multiplication and division values. It should return the 2D array it

creates.

- `void print_table(multdiv_entry** tables, int row, int col);`
  This function should take a 2D array as created by `create_table()` and its sizes and print out the multiplication and division tables like above.

- `void delete_table(multdiv_entry** tables, int row);`
  This function should delete all of the memory allocated to a 2D array created by `create_table()`, given the array and its size as arguments. ***This function is important.*** Your program should not have a memory leak. (Test your program with `valgrind` tool)

  At the end of the program, prompt the user if they want to see this information for a different size matrix. Make sure you do not have a memory leak.

5. Now, compile your program:

```
g++ multdiv.cpp -o multdiv
```

once your program is compiled, test it with a few different values of row and col to make sure it works.

Example Run (User inputs are highlighted):

```
Enter an integer greater than 0 for row: 5
Enter an integer greater than 0 for col: 5
Multiplication Table:
1   2   3   4   5
2   4   6   8   10
3   6   9   12  15
4   8   12  16  20
5   10  15  20  25

Division Table:
1.00  0.50  0.33  0.25  0.20
2.00  1.00  0.67  0.50  0.40
3.00  1.50  1.00  0.75  0.60
4.00  2.00  1.33  1.00  0.80
5.00  2.50  1.67  1.25  1.00

Would you like to see a different size matrix (0-no, 1-
yes)? 0
```

Show your completed work and answers to the TAs for credit. You will not get points if you do not get checked off!

Submit your work to TEACH for our records **(Note: you will not get points if you don't get checked off with a TA!!!)**

1. Transfer all files you've created in this lab (.cpp, .txt,) from the ENGR server to your local laptop.
2. Go to TEACH.
3. In the menu on the right side, go to **Class Tools → Submit Assignment**.

4. Select **CS162 Lab4** from the list of assignments and click “**SUBMIT NOW**”
5. Select your files and click the Submit button.