CS 162 LAB #3 – Dynamic Memory and Multidimensional Arrays

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 and quiz (Individual: Quiz is worth 5 points)
- Part 2: TA demo
- Part 3: Understand Dynamic Memory (Group work: 3 points)
- Part 4: Two Pointer exercises (Individual work: 2 points)
- Part 5: 2D static arrays (Individual work: 5 points)

(5 pts) Part 1: Worksheet and quiz

This session will be led by your lab TAs. Please follow their instructions, participate, and complete worksheet 3 and the lab 3 quiz.
https://web.engr.oregonstate.edu/~songyip/Teaching/CS162/labs/WS3.docx (pdf version)

Get the starter code

Clone the starter code for lab 3 from GitHub Classroom here:
https://classroom.github.com/a/WP8N-iIT

Part 2: TA demo: 2D static array of booleans

A TA will demonstrate a program that fills a 2D statically allocated array of booleans with trues and falses based on simulated coin tosses. You'll apply some of these techniques in part 5 of this lab.

(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 doubles, we can do this:

```
double nums[10];
```

The size, layout, and lifetime of memory of the array above is decided during compile time. However, what if we don't know how many integers we want to store in the array until runtime? What size should we use for my array?

```
double nums[???];
```

The following is not legal C++ code!

```
int size;
cin >> size;
double nums[size]; // DON'T DO THIS!
```

The above code attempts to create a statically allocated **Variable-Length Array** (VLA). Statically allocated VLAs are not allowed in C++. However, if you try to compile and run the above code, it seemingly works just fine. This is because `g++` actually supports VLAs through a defaultly enabled VLA extension. Even still, you should **NOT** rely on such extensions; they're not portable (not all compilers necessarily support them on all platforms since they're not officially part of the C++ language), they're not protected by the backwards-compatibility of C++ versioning, and other C++ developers who don't know about this VLA extension will have a hard time understanding how your code works.

Moreover, recall that all statically allocated memory is bound to the scope of its associated variable, and hence the lifetime of statically allocated memory is coupled to the call stack. That is, whenever a local variable falls out of scope, all of its memory is freed. In the above code, both **`size`** and the array **`nums`** are local variables, and they will fall out of scope when the containing function ends. What if we need that array's memory to stay alive after the function ends?

To solve both of these problems (creating VLAs and decoupling the lifetime of our array's memory from the call stack), we need to allocate our array's memory dynamically (colloquially, we need to store our array on the "heap" instead of the "stack"). To achieve this, we need to use the **`new`** operator, like so:

```
int *nums = new int[size];
```

Recall from lecture that the **`new`** keyword allocates dynamic memory in a special place (the heap) and returns the base address of the allocated memory. We then store that base address in a pointer. Also, recall that you can use a subscript operator (square brackets, `[]`) on a pointer that stores the base address of an array as if it were an array variable itself. With this knowledge, we can use `nums[i]` to access the i^{th} element of our dynamically allocated array. Moreover, dynamic memory's lifetime is independent of the call stack; even after the `nums` variable falls out of scope, the array's memory will remain allocated. This means that we could (for example) return a copy of the pointer `nums` to the function caller. That copy would still point to the same location in memory (the base address of the dynamic array), and that memory would still be allocated for our array.

(1 pt) Let's take an opportunity to practice dynamic memory, pass-by-pointer, and pass-by-reference. Form groups of 3 and implement the following functions in a new `.cpp` file (e.g., `dynamic_memory.cpp`). Each serves the same purpose—to dynamically allocate an array of integers and send a pointer storing its base address back to the function caller. The size of the array should be determined by an integer parameter. The only difference between the three functions is *how* they send the pointer back to the caller (return it, or store it in an external variable via a pointer or reference parameter). Each group member is tasked with implementing one function, but it is essential for everyone to comprehend all three.

1. ```
   // Takes an int argument representing the size of the array,
   // and returns the base address of the dynamically allocated
   // array
   int* create_array1(int size);
   ```

2. ```
   // Takes a reference to an int pointer, and an int argument
   // representing the size of the array. Stores the base address
   // of the dynamically allocated array in the int pointer
   // that's supplied via the reference parameter
   void create_array2(int*& array, int size);
   ```

3. ```
   // Takes a pointer to an int pointer, and an int argument
   // representing the size of the array. Stores the base address
   // of the dynamically allocated array in the int pointer
   // that's supplied via the pointer parameter (i.e., stores the
   // base address in the int pointer that the int double-pointer
   // parameter points to)
   void create_array3 (int** array, int size);
   ```

(1 pt) Next, in your main() function, write the function calls to use the functions that you created above. The size argument supplied to the function calls should be received from the user via `cin`.

(1 pt) Lastly, since dynamic memory's lifetime is decoupled from the call stack, *we* need to decide when to free/delete it. Failure to free dynamically allocated memory is known as a **memory leak**. To free dynamically allocated memory, use the delete operator. Refer to the lecture notes for a refresher on how to use it. Also, **it's a good idea to assign your pointer(s) the value of nullptr after freeing their underlying dynamic memory** since that's the only way to keep track of the fact that their memory has been freed (e.g., to avoid accidentally deleting them a second time in the future, which results in undefined behavior). Don't forget to do this.

Check to see if your program has any memory leaks by using valgrind (installed on the ENGR servers) with the following steps:

1. Recompile your program using `g++`, but supply the `-g` flag somewhere in the command. For example, `g++ -g main.cpp`
2. Run your executable through valgrind like so (replacing `a.out` with the name of your executable as appropriate):

   ```
   valgrind --leak-check=full a.out
   ```

Valgrind should report 0 bytes in 0 blocks at exit. If it doesn't, then you've forgotten to free some of your dynamic memory, or your program crashed before it got a chance to free some of it. In such a case, Valgrind should report where in your code that dynamic memory was allocated (assuming you compiled with the `-g` flag). Unfortunately, Valgrind can't tell you where you
   ```

should free your memory—only where it was allocated. Determining when and where to free your dynamic memory is your responsibility. Just remember not to free it until you're done with it.

(2 pts) Part 4: Two Pointer Exercises

In this part of the lab, you will be working **individually** to finish two pointer-related exercises.

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

(5 pts) Part 5: 2D static arrays

Create a new `.cpp` file (e.g., `2d_arrays.cpp`). In this `.cpp` file, write a program that:

1. (1 pt) Contains a function with the following header / prototype:

   ```
   void populate_multiplication_table(int table[12][12]);
   ```

   When called, this function's purpose is to populate the given 2D array with the values of a 12x12 multiplication table.

   To clarify, you get one point here just for prototyping / having this function, even if it's not fully implemented.

   Recall from lecture that the size of the first dimension in a multidimensional array parameter is optional, but the sizes of subsequent dimensions are required. Here we supply both of them, just to be explicit that this function is strictly designed for 12x12 arrays.
2. (1 pt) Statically allocates a 12x12 2D array of integers
3. (2 pts) Appropriately implements and uses the `populate_multiplication_table` function prototyped above to populate the statically allocated 2D array
4. (1 pt) Prints the multiplication table to the terminal. The first row should contain values 1-12; the second row should contain the multiples of 2 from 2 to 24; the third row should contain the multiples of 3 from 3 to 36; and so on. When printed to the terminal, each row should be printed on its own line, and each column should be separated by a tab. A tab character can be printed like so:

   ```
   cout << "\t";
   ```

   A backslash (not a forward slash) can be used with various other characters in strings to form **escape sequences** that represent special characters or character sequences. Using a backslash with a t (as above) forms an escape sequence for a tab character.

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

For backup purposes, please stage, commit, and push your work for this lab (including all documents/text files for group work, and programs) to your remote repo on GitHub Classroom.