CS 162 LAB #2 – Pointers & Dynamic Memory

Each lab will begin with a brief demonstration by the TAs for the core concepts examined in this lab. As such, this document will not serve to tell you everything the TAs will 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 10 points total. Here's the breakdown:

- Part 1: Understand Pointers (Group work: 1 pts)
- Part 2: Implement Two Pointer Exercises (Individual work: 2 pts)
- Part 3: Understand Dynamic Memory (Individual work: 5 pts)
- Part 4: Implement arrays in assignment 1 (Individual work: 2 pts)

Part 0: Get the start code

Upload/download the start code for this lab onto `flip.engr.oregonstate.edu`:
wget https://classes.engr.oregonstate.edu/eecs/spring2023/cs162-010/labs/lab2.zip

(1 pt) Part 1: Understand Pointers

In part 1, you will be working **in a group** to review the references and pointers in C++.

Files needed for this part: `sentence.cpp`

Revisit pointers:

Suppose our main function is as follows:

```
int main() {
   string sentence;

   get_sentence(sentence);
   cout << sentence << endl;

   return 0;
}
```

In last lab, we know that in order to change the value of the string inside the function, we need to add an ampersand (&) in front of the parameter. This is called pass by reference.

void get_sentence(string &s);

Now, change the function prototype to:

void get_sentence(string *s);

And answer the following questions:

1. Can we change the value of the string inside the function if we change the function prototype to the above?
2. What else needs to be changed when we make the function call? Inside the function?

3. What is the difference between an ampersand (&) and an asterisk (*) added in front of the parameter? Use a diagram to explain.

(2 pts) Part 2: Implement Two Pointer Exercises

In part 2, you will be working **individually** to finish two pointer exercises.

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

(5 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
```

(3 pts) Now, implement the following function(s) to create an array of integers on the heap, which size is determined by the user input. There are three different ways to achieve this goal, the function prototypes are provided:

```
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 three 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**, 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: Implementing arrays in assignment 1

Task one: Create an array to keep track of the shooting result of the player.

- What is the dimension and size of the array?
- How to display the result of each rack using "X", "O", "M"?
- Could you reuse the array to store the shooting result of the other player?

Task two: How would you calculate the total score for all players?

- What are the steps to calculate the total score for one player?
- Do you need a separate array to store the total score for all players?

If any of your functions are more than 15 lines of code, what can you do to make it smaller? If you are having difficulty thinking of how to make it smaller, then ask a TA to help you!!!

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 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 Lab2** from the list of assignments and click "**SUBMIT NOW**"
5. Select your files and click the Submit button.

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