CSS 332: Programming Issues with Object-Oriented Languages
Lab3: Data representation, C strings, Scope and Lifespan

This is a follow-on lab about more advanced memory management topics. Please pick *one* of the following tasks to complete to start with. There is no need for you to rush to do everything; instead, focus on whatever you're least comfortable with and try to get to the point where you now feel good about that concept. Please pick *one* of the following tasks to start with. There is no need for you to rush to do everything; instead, focus on whatever you're least comfortable with and try to get to the point where you now feel good about that concept. Complete and submit **3 of the 4 exercises** for full credit

`CommandLine.cpp` (Observation only)

For your experimentation out of class, download and exercise with `CommandLine.cpp` a simple program which examines command line arguments. This is also used in one of our videos. Be sure to either use `system("pause");` or use a break point to be able to see the output window.

`SizeofDemo.CPP`

Download `SizeofDemo.CPP` and explore the `sizeof()` operator. Answer the following questions as comments in your code:

1. Consider `word[]` and `word2[].` Which is a valid "C-string"? Why? Why are the two arrays different size?
2. The nums array contains 3 elements, so why is its size 12 (or 24)? Write a line of code to display the actual number of elements in nums array.
3. The size reported for moreNums is 8 (or 4), even though it's supposed to contain 10 ints. Is this some sort of magical compression? Assuming not, why is its size only 8 (or 4), and why does `sizeof()` treat moreNums differently than nums?
4. Did memset do what you expected?

`MorePointers.CPP`

C++ has a string class, just like Java. But it also has a more primitive string representation, "C strings". These are arrays of chars. Download and use the `MorePointers.cpp` code to first check that you understand pointer basics. Then, look at the following points about C strings:

1. We include cstring; why?
2. Why do we use the **`strcpy()`** function? Why would it be safer to use **`strncpy()`** (hint: look them up; what's the difference between them?)?
3. How many elements does a char array need to have to hold a 5-character C string? Why?
4. Is the second argument to **`strcpy()`** a C string or a C++ string object? What exactly is its type?

ScopeEtc.cpp, SuperInt.h, and SuperInt.cpp

Here is some code to use for exploring variable scope, lifespan, and l-values: `ScopeEtc.cpp, SuperInt.h, SuperInt.cpp`

1. Your driver function is located in `ScopeEtc.cpp.`

2. Comment the `ScopeEtc.cpp` file code nicely and explain your new findings. Replace the simple explanation of program on top, with a detailed description of file.

3. Somewhere in `ScopeEtc.cpp`, try putting a `SuperInt` object on the right hand side of an assignment statement using `theValue().` Does it work? Why?

4. Also, in `ScopeEtc.cpp` try putting a `SuperInt` object on the left hand side of an assignment statement using `theValue().` Does it work? What can you do to make it work? Why?

Struct Fraction.cpp

C++ allows us to create our own user-defined aggregate data types. An aggregate data type is a data type that groups multiple individual variables together. One of the simplest aggregate data types is the **struct**.

Create a struct to hold a fraction, name your file `Fraction.cpp`.

1. Create 2 integer members, named numerator and denominator.
2. Code to prompt user and get user input for 2 fractions has already been implemented in `main()`.
3. Write a function called `multiply` that takes 2 fractions, multiplies them together, and prints the result out as a decimal number.
4. How would you prevent `multiply()` from doing integer division?
5. You do not need to reduce the fraction to its lowest terms.
6. Download the `FractionDrive.cpp` file provided to test your struct code.
7. Now, in `FractionDrive.cpp` crate a 3rd Fraction object called `f3`. Create a pointer to point to `f3.` This time use pointers to manually assign values to data members of `f3.`
8. Finally, `call multiply(f1, f3)` and check your output.