CSCI 444/544 - Lab 0
Friday, January 15, 2021
LAB IS DUE BY **FRIDAY JANUARY 22 11:59 PM**!!

This lab will verify your installation is set up correctly and serve as the starting point for A1. The provided code is set to work on the lab machines out of the box. If you are working on a lab machine, continue to Step 0A. If you are working on your personal machine, then continue to Step 0B.

Step 0 – "Hello World"

You can use any text editor/IDE for this lab. You will only need access to gcc. Compile and run as you wish.

Make a new file, main.c – since we're only working with C code.

Part I – C++ Hello World

For C++, we needed to include the iostream library

```
#include <iostream>
```

And then inside main we had access to our standard out stream object and we can print a message.

```
std::cout << "Hello C++ World!" << std::endl;
```

This should seem like old hat. Great, let's do the equivalent in C.

Part II – C Hello World

For C, we just need to include a different library – the standard input output library.

```
#include <stdio.h>
```

And then inside main we have access to our output functions and we can print a message.

```
printf( "Hello C World!\n" );
```

With C strings, we put the entire formatting all in the output, hence the actual new line character.

Step 1 – I/O

Part I - Output

Now that we have a the start of a working C program, let's expand our I/O toolbox. IMHO, I/O in C++ is clunky and cumbersome, especially whenever we have to deal with any type of formatting. I/O in C on the other hand is much cleaner and succinct. With the above example, `printf()` is already going to the standard out.

We can explicitly state which output stream we want to print data to by using `fprintf()`. There exists a pointer to the standard out stream, aptly called stdout. We can rewrite the above `printf()` line to be:

```
fprintf( stdout, "Hello C World!\n" );
```

And we'd get the same result, but we can specify where the output should go. We could write to a file instead by opening the file and changing our destination stream pointer.

```
FILE *file = fopen("output.txt", "w");
if( !file ) {
  fprintf( stderr, "Could not open file for writing\n" );
} else {
  fprintf( file, "Hello C File!\n" );
}
fclose( file );
```

`fprintf()` will work with any output stream pointer by supplying it as the first argument. The second argument is then the format string to output to the destination. The format strings is how the output becomes much cleaner with C. Let's add in some variables.

Declare a set of variables of various types:

```
int year = 2021;
float piApprox = 22.0f / 7.0f;
float radius = 4.5f;
float area = piApprox * radius * radius;
```

To output a variable, we add a format specifier to the format string that will be substituted with additional arguments. To print out the current year, we would use the line:

```
fprintf( stdout, "The current year is %d.\n", year );
```

And the corresponding output would be:

```
The current year is 2021.
```

The format string will replace %d with the first argument that follows. Read the documentation for fprintf(), available online at https://www.cplusplus.com/reference/cstdio/fprintf/, and use the appropriate format string to print out the following exactly:

```
Our value of PI is 3.143.
A circle with radius 4.5" has an area of 63.64 sq. in.
```

Part II – Input

Input works very similarly to output. Instead of using `fprintf()` with a destination stream and a format string, we'll now use `fscanf()` with a source stream and a format string.

When reading a variable for output, the variable name asks the computer to recall the value in memory. However, when we writing a variable for input we need to provide the memory location to place the value in. While `fprintf()` expects the value to read from, `fscanf()` expects the address to write to.

```
int age;
fprintf( stdout, "What is your age? " );
fscanf( stdin, "%d", &age );
```

The file input.csv has been provided with this lab. It contains three floats that are comma separated. With a single line, read in these three floats and store them in variables x, y, z. Print these values back out along with their average to the following format:

```
Read in 1.0 and 3.0 and 4.5 which averages 2.8.
```

Edit the values in the file and verify your program reads in the updated values.

Step 2 – Pointers & Dynamic Memory / Buffers

Instead of passing in the address of a variable for input, we could have been working with pointers directly instead.

```
int* yourAge = NULL;
yourAge = (int*)malloc(sizeof(int));
fprintf( stdout, "What's your age again? " );
fscanf( stdin, "%d", yourAge );
free(yourAge);
```

`malloc` also allocates a given amount of memory on the heap and returns the associated starting address of that amount of memory. `malloc` returns a void pointer, so we will cast it to our appropriate type. This works for single values, but recall we can also use pointers to represent arrays.

```
int* arr = (int*)malloc(5 * sizeof(int)); // create an array of 5 integers
```

We can now access values within the array in one of two fashions:

```
arr[0] // the first value
arr[1] // the second value
```

OR

```
*arr   // dereference the current value
++arr  // advance forward in memory one step size
*arr   // dereference the current value
--arr  // move backwards in memory one step size and return to start
```

The first method asks the computer to do the math for us using the equation

baseAddress + offset * stepSize

In the second method, we are manually performing the offset. Once we begin working in the style of the second method (which is what OpenGL does), we now start refurring to these array blocks of memory as a buffer.

We reference an amount of memory within a buffer by an offset and a length. Once we start packing data in to a buffer, we need to know where to get/put the data and how much data to pull out/put in.

Consider the following structure:

```
struct Vertex {
  float position[3];  // denoted as p1, p2, p3
  float normal[3];    // denoted as n1, n2, n3
  char id;            // denoted as i
  float texCoord[2];  // denoted as t1, t2
};
```

In memory, the data would be stored in order as

```
p1 p2 p3 n1 n2 n3 i t1 t2
```

We would state this data is interleaved. If we built an array of our Vertex struct, we would then be working with an Array of Structures (AoS). For instance:

```
p11 p21 p31 n11 n21 n31 i1 t11 t21 p12 p22 p32 n12 n22 n32 i2 t12 t22
```

All of the data for a single vertex is located together in memory. The first vertex starts at location 0 in memory. A float takes up 4 bytes in memory and a character takes up 1 byte of memory. So a single vertex takes up 33 bytes. Below is the layout of the buffer with the starting location of each value in memory:

```
p11 p21 p31 n11 n21 n31 i1 t11 t21 p12 p22 p32 n12 n22 n32 i2 t12 t22
0   4   8   12  16  20  24 25  29  33  37  41  45  49  53  57 58  62
```

With this interleaved style, if we wanted to extract just the normal we would need four pieces of information:

1) The *offset* to the start of first value
2) The *length* of the value to read
3) The *stride* to the start of the next value
4) The number of values to read

For the above configuration in memory, the corresponding values would be:

1) 12 // skip over the first three floats
2) 12 // read three floats worth
3) 33 // the size of an entire vertex's worth of data
4) 2 // we have two vertices in our array

From the second and fourth values, we can calculate that we need 24 bytes to store all the normal values.

An alternative approach to store the same data would be with the following structure:

```
struct VertexData {
  float positions[6];
  float normal[6];
  char ids[2];
  float texCoords[4];
};
```

Now we are working with non-interleaved data and an Structure of Arrays (SoA). Our data in memory now looks like:

```
p11 p21 p31 p12 p22 p32 n11 n21 n31 n12 n22 n32 i1 i2 t11 t21 t12 t22
0   4   8   12  16  20  24  28  32  36  40  44  48 49 50  54  58  62
```

The same amount of memory is still used, but how the data is organized has changed. Now to get all of the normal data, our four values would be:

1) 24 // skip over the first six floats – all of our position data
2) 12 // read three floats worth
3) 12 // skip over a single normal's worth of data
4) 2 // we have two normal in our array

NOTE: In OpenGL, you will sometimes see the stride specified as zero. This signifies that our data is located sequentially so the stride then becomes the length of data last read.

We can only read/write a single block of data at a time. The block is specified by its offset and length.

There is are two additional files provided, data1.txt and data2.txt. The first line states how many records will exist in the file. The following lines then have a single record per line. Each record consists of an integer, a float, and a character. Read in the first file in to an Array of Structures. Read in the second file in to a Structure of Arrays. Then answer the following questions in a README.txt:

1) **What is the size of a single record?**
2) **For the AoS, what is the offset and length for each component of the second record?**
3) **For the SoA, what is the offset and length for each component of the second record?**
4) **When is it more advantageous to use an AoS?**
5) **When is it more advantageous to use a SoA?**

Step 3 – Function Pointers

Just like variables are stored in memory, function definitions are stored in memory as well. Create the following function:

```
void standard(int num) {
  fprintf( stdout, "Num is %d\n", num );
}
```

We can call the function

```
standard(5);
```

But we can also inspect where it is stored in memory:

```
fprintf( stdout, "The standard function is stored at 0x%p\n", standard );
```

We can pass pointers to values around our program and we're essentially just sending memory locations. Therefore, we can send the memory location that refers to a function as well. This is accomplished via function pointers. We've already briefly discussed this is how the callbacks work for our event driven GUI. We will soon see how function pointers are also used in our shader code within GLSL.

To see this in action, create a second function:

```
void external(int num) {
  FILE *file = fopen("output2.txt", "a" );
  fprintf(file, "%d\n", num);
  fclose(file);
}
```

Notice that both the standard function and the external function have the same function signature. They both take an integer is input and return void. We will use this function signature as our data type when creating the function pointer parameter. Create a third function that will accept a function pointer and the argument for the function:

```
void writeToStream( void (*f)(int), int n ) {
  f(n);
}
```

We the data type `void (*)(int)` specifies a function pointer and we provide the name of the pointer after the indirection operator. How do we use this new function? When we call it, we will provide the name of the function to run along with the value to pass to the function.

```
writeToStream( standard, 5 );
writeToStream( external, 5 );
```

Create a third implementation of our function signature that writes out twice the parameter value to the standard output.

Step 4 – Submission

When you've gone through all the steps zip together your main.c and README.txt files and upload to Canvas.

While this lab had very little to do with graphics, you will see a large amount of C code used throughout OpenGL and GLSL. The buffer notation is exactly how our VBOs work, so hopefully this provided a little bit better clarity. If there is still questions about how VBOs work, we will be reviewing them coming up. You'll also see function pointers used in the callbacks and in our shaders after we talk about VBOs.

LAB IS DUE BY **FRIDAY JANUARY 22 11:59 PM**!!