

TASK 1 – Brains On!

Alrighty! Now that we have a basic understanding of C programming and a work environment we can use to compile and execute C code, let's dig a bit deeper.

Before we go any further, however, we need to talk about data types!

Data Types

Python removes the need for any data type specifications. C on the other hand *requires data type specification*. If you want to use an integer in a variable, you first need to specify the data type before declaring the variable. This looks something like this:

```
int someVar;
```

Notice how I didn't set it equal to anything. In C, unlike in Python, we can simply declare the memory space but not set anything in it. To initialize the variable, I can do this:

```
someVar = 2;
```

The main data types you will be using in C will be:

Data Type	Size (bytes)	Numerical Range	Format Specifier
int	4	-2,147,483,648 to 2,147,483,647	%d
unsigned int	4	0 to 4,294,967,295	%u
char	1	-128 to 127	%c
float	4	1.2E-38 to 3.4E+38	%f
double	8	1.7E-308 to 1.7E+308	%lf
pointer	8	0x0 to 0xffffffffffffff	%p

Format Specifier

If you look closely at the chart above, you'll see a column titled *Format Specifier*. This specifier will be used anytime you need to input or output a given variable. We'll cover that in the next section.

Function: printf()

This is the function you will use to print data to the console. While it may seem just like using the *print()* function in Python, it's quite a bit more complicated than that. First off, *printf()* requires the use of a *format string*. The format string is the *first* argument in all *printf()* calls and

it always contains the supporting string characters *and* format specifiers. For instance, look at the following code:

```
printf("Hello world!");
```

While it may seem we're just printing a string from *printf()*, what we're actually doing is feeding *printf()* a *format string*. When the function is called, the *format string* is pushed onto the stack followed by as many variables as you might wish to print. In this case, there are NO variables being pushed. The *format string* then manages *how* each variable is printed. Since this call has not variables being pushed and no format specifiers, it looks rather plain.

Let's try to print an Integer:

```
int val = 5;  
printf("This is the value: %d\n", val);
```

This time, the call looks quite a bit different. Let's pick it apart.

The first argument in the call is the *format string*. It has a leading set of characters that support user comprehension followed by a *%d* format specifier and a *\n* indicating a new line. When we feed *printf()* the *%d*, we're telling it, "Print the first variable after the format as an integer." When we run the program, we'll get the following output:

```
This is the value: 5
```

Let's pile some more variables in there:

```
int val1 = 5;  
int val2 = 10;  
int val3 = 20;  
printf("val1 = %d, val2 = %d, and val3 = %d\n", val1, val2, val3);
```

The print function will print the first variable after the format in the first format specifier, the second variable after the format in the second specifier, and so on. This is called *positional* printing. The variables position in the function call determines where it'll be printed in the format string.

Function: *scanf()*

To take input, we'll start by using the *scanf()* function. Later down the road, we'll stop this as it's an insecure function, but for now it's the easiest way to learn.

The *scanf()* function works much like the *printf()* function (but not exactly). You feed it a *format string* then the variables you want to save input into.

Let's take some input:

```
int val;  
scanf("%d", &val);
```

It should look familiar, but also a bit different. You see, while the *printf()* function can just take the variable as an argument, in order for C to store something at a given variable, it *needs* the address (or pointer) of that variable. To get the address of any variable, you simply use the *&* character just as we did above.

What happens here is that we create a space in memory for the integer called *val*. This creates 4 bytes of space. This 4 bytes of space has a memory address that we'll need to give to *scanf()* so the function knows where to put the input data. To get this we use *&val* and feed that into the *scanf()* function.

At this point there is a distinction that must be made. Any String (char) **does not need to use the &** character when feeding into *scanf()* due to that fact that a char * is *already* a memory address as denoted by the *.

So to take input into a string we would use:

```
scanf("%s", string);
```

I didn't show the variable creation because that's a different conversation that we'll have later. For this lab, you know enough to be dangerous!

Main Syntax

Real quick, let's talk about syntax... C is a strongly typed language. This means the language can also be called "Syntax Heavy". Here are the basics.

All variables need to be prefaced with their data types.

```
int val1;  
float val2;  
char letter1;  
double val3;
```

In C we have both *declaration* (what you see above) and *initialization*. To initialize a variable is to put a value in it. Nothing more. Like this:

```
val1 = 5;  
val2 = 2.5;  
letter1 = 'A';  
val3 = 3.45;
```

And yes, I'll say it... **YOU MUST USE A SEMICOLON AT THE END OF EVERY LINE.**

For Repetition Structures, we use {} instead of indentation. Plus we need to declare our counter variables. Pay close attention here while we write a *for loop*.

```

int i;
int count = 10;
for(i = 0; i < count; i++) {
    printf("%d\n", i);
}

```

We declare the indexing variable, we declare and initialize a count variable, then we write our for loop. In the for loop we first set *i* to 0, then define the stop condition (*i* < 0), and finally we set how much *i* will count up each iteration (++ denotes +1). In the print statement we push a %d\n format string to print an *int* and give it the *i* variable to print.

Here's a while loop:

```

int good = 0;
while(good < 10) {
    printf("NUM: %d\n", good);
    good++;
}

```

Here, we declare an int called good and initialize it to 0. We then setup our while loop giving it the stop condition of < 10. In the while loop we print good then iterate good so that we can actually get out of the loop.

Custom Functions

Okay, last bit here. In C, we need to be able to define and call functions. To do this, we need to first define a function protocol. For a function that will loop over some interval, the prototype could look like this:

```

void loopPrint(int);

```

The header is broken up like this: the return type (void – return nothing), the name (loop_print), and the parameters as *data types only* (int).

Now, below the main method, let's write the function.

```

void loopPrint(int loopNum) {
    int i;
    for(i = 0; i < loopNum; i++) {
        printf("%d\n", i);
    }
}

```

To make it value-returning, we just indicate the type it will return. Like this:

```

int sumRange(int);

```

Then we define the full function:

```
int sumRange(int upperBound) {
    int i;
    int accum = 0;
    for(i = 0; i < upperBound + 1; i++) {
        accum += i;
    }
    return accum;
}
```

To call these from the *main()* method, you'd do this:

```
int main() {
    int accum = sumRange(3);
    printf("%d\n", accum)
}
```

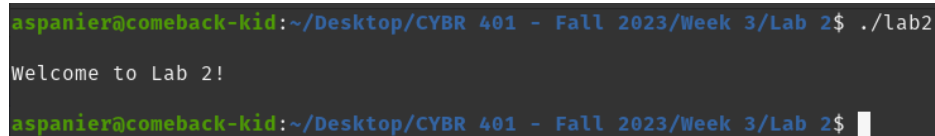
Whew! That's a lot to nail down. So let's write some code. In this lab, **I'm going to lean into you a bit more to think it through, so be sure you read everything above.**

TASK 2 – Make a source file

Make a new .c source file, call it lab2.c. Write all your code here and submit this at the end of this lab. Be sure you include *stdio.h* and *stdlib.h* and write a *main* method. Print "Hello World" and make sure the code compiles and runs before continuing on.

TASK 3 – Print a Welcome Message

First off, let's welcome our user by printing, "Welcome to Lab 2!" Give it a single newline *before* the line and two newlines *after*. Your console should look like this:



```
aspanier@comeback-kid:~/Desktop/CYBR 401 - Fall 2023/Week 3/Lab 2$ ./lab2
Welcome to Lab 2!
aspanier@comeback-kid:~/Desktop/CYBR 401 - Fall 2023/Week 3/Lab 2$
```

TASK 4 – Prompt the User

Define a method called *getUserInput()* that asks a user, "Please enter a number from 1 to 10." Store the value and return it back to the main method. In the *main()* method, print a newline followed by, "You chose the value: X"

Return the proper data type and use a function prototype. Here's a hint, use the *printf()* function to print the prompt, then use the *scanf()* function to read input.

Call the function from main, catch the result, and print from *main()*.

Your console output should look like this:

```
aspanier@comeback-kid:~/Desktop/CYBR 401 - Fall 2023/Week 3/Lab 2$ ./lab2
Welcome to Lab 2!

Please enter a number from 1 to 10: 3

You chose the value: 3
aspanier@comeback-kid:~/Desktop/CYBR 401 - Fall 2023/Week 3/Lab 2$
```

TASK 5 – Loop and Print the User’s Input

Define a method called *calcAvg()*. This function will take an integer *userInput* as input and it will loop from 0 to that value, printing up during each iteration. Return zero from the function.

Be sure to define your function prototype and define the proper return value.

Call the function from main and catch the result.

Your console output should look like this:

```
aspanier@comeback-kid:~/Desktop/CYBR 401 - Fall 2023/Week 3/Lab 2$ ./lab2
Welcome to Lab 2!

Please enter a number from 1 to 10: 5

You chose the value: 5
0
1
2
3
4
aspanier@comeback-kid:~/Desktop/CYBR 401 - Fall 2023/Week 3/Lab 2$
```

TASK 6 – Sum the values in the calcAvg() function and divide by total to get the mean

This is pretty self explanatory. Sum *i* as it counts up into an accumulator. This means you’ll need to declare an accumulator and set it to 0. After summing all *i* into the accumulator, divide the accumulator by the *userInput* count and return the value to main.

Change your return type for the *calcAvg()* to float if you already haven’t. Your accumulator will need to be of type *float*, your mean variable will need to be *float*, and your variable in the *main* method will need to be *float*. All you’re doing is finding *mean*.

Call from the main method, catch the result as a float, and print like below.

Your console output should look like this:

```
aspanier@comeback-kid:~/Desktop/CYBR 401 - Fall 2023/Week 3/Lab 2$ ./lab2

Welcome to Lab 2!

Please enter a number from 1 to 10: 7

You chose the value: 7
The mean is: 3.000000
aspanier@comeback-kid:~/Desktop/CYBR 401 - Fall 2023/Week 3/Lab 2$ ./lab2

Welcome to Lab 2!

Please enter a number from 1 to 10: 8

You chose the value: 8
The mean is: 3.500000
aspanier@comeback-kid:~/Desktop/CYBR 401 - Fall 2023/Week 3/Lab 2$ █
```

TASK 7 – Print a final message!

Great job! Now let's print a farewell to our user. Define a method called *farewell()*. In the function print, "Farewell! Thank you for using this program!" Print a newline before and two newlines after.

Call from the main method. Be sure to create the proper function prototype and return type.

Your console should look like this:

```
aspanier@comeback-kid:~/Desktop/CYBR 401 - Fall 2023/Week 3/Lab 2$ ./lab2

Welcome to Lab 2!

Please enter a number from 1 to 10: 5

You chose the value: 5
The mean is: 2.000000

Farewell! Thank you for using this program!

aspanier@comeback-kid:~/Desktop/CYBR 401 - Fall 2023/Week 3/Lab 2$ █
```

TASK 8 – Save and Submit

Now that you are done, be sure the output prompt is exactly like above. Submit your completed C source file.