COSC 2P95 – Lab 2 – Section 01 – Procedures, functions, and arrays

As with all other labs for this course, you'll still be using Linux for this lab. As such, reboot and log back in if necessary (refer to Lab 1 for details).

In this lab, you'll be practicing using procedures, functions and arrays. The former two will help you to abstract behaviours and write more readable code; the latter will help you store data for later use.

Procedures, and functions, and bears*, oh my!

*bears may not be implemented until the 2023 C++ standard

Let's just get right to it, shall we?
First, do we remember what `#include` does?
To be safe, let's start with one of our class examples:
http://www.cosc.brocku.ca/~efoxwell/2P95/code/03/slide10/funky.cpp
http://www.cosc.brocku.ca/~efoxwell/2P95/code/03/slide10/funky.h
Use wget, or just download via your browser. Up to you.

Ensure that you can compile it:

```
g++ -o f funky.cpp
```

As we saw in class, note that you don't need to tell the compiler about the .h file; the .cpp file does that for you.

Try writing a procedure that reads from user into into a *local* variable, and then just parrots it back.
Now, try the same thing, but read into a variable at the *file scope* (i.e. don't declare it inside the procedure).
Try printing that variable's value from within another procedure.
Feel free to play with the values and variables a bit. Once you're comfortable with scope, we'll continue.

Now, we want to write a function. Just for ha-has, try writing a function that receives an array of values (and a length of said array), and returns the *largest* of those values.
e.g.

```
int largest(int arr[], int size);
```

How would you go about doing that? Give it a shot.

Pointers

We've already looked at reasonable examples of using pointers in the code examples, so let's try something silly.
Let's write a recursive procedure to display the contents of an array, but handling it the same way we'd write a tail recursion.
i.e.

```
void tail(int *arr, int size) {
	if (size<=0) return;
	cout<<arr[0];
	tail(arr+1,size-1);
}
```

So, here's a question: how can we reverse the sequence of the output, by making the most minimal change possible?

While we're on pointers, what would you expect from the following?

```
int a=37;
int *b=&a;
b=40;
cout<<a<<endl;
```

Note: do ***not*** just skim through this part! Really look at it!
In the hypothetical situation that maybe there's a problem, can you spot what it is?

Reference Letter

Since we already have a basic understanding of what references are and how they work, let's go straight into why we need to be amazingly careful with references, k?
Consider the following function:

```
int &huh(int &sure, int &maybe) {
     return maybe>sure?maybe:sure;
}
```

First, because it uses references, be sure to use variables (e.g. instead of literal values).
What does it actually do?
We can try invoking it:

```
int oh=14,geez=10;
int kwyjibo=huh(oh,geez);
cout<<kwyjibo<<endl;
kwyjibo=0;
cout<<oh<<endl;
```

The output might not be what you'd expect.
Try changing *kwyibo* to also be a reference (note: not a pointer!).

Recursion

Let's try writing a very simple recursive function, for approximating the *square root* of a value.
The basic approach is simple:

- Assume that the root is within some intentionally low guess, and some artificially high guess
  - e.g. between zero and the actual squared value)
- Calculate a guess as being the midpoint between the underestimate and the overestimate
- Square the guess, and compare it against the actual squared target
- If your squared guess is within some reasonable *threshold* of the actual target, you're done
- If the guess is too high (squared guess is greater than the target), then the guess becomes your new upper-bound; if it's too low (squared guess is lower than the target), then the guess becomes your new lower bound
  - Either way, repeat

Try implementing this yourself.
After you're satisfied with your own solution, take a look at this… special version:
http://www.cosc.brocku.ca/~efoxwell/2P95/code/03/extra/roots.cpp

Note three things:

- The abs function is *inlined*
- You *can* provide a default argument if you wish
- That is an incredibly bad way to take the absolute value of a double

Submission Task

Your task is simple: write a function to compute the n^th fibonacci number.
This *n* will be specified via the command-line as the sole parameter.

- The 0^th fibonacci number is 0
- The 1^th fibonacci number is 1
- The n^th fibonacci number is the sum of the n-1^th and the n-2^th

To be clear, your program must behave as such:

- It doesn't matter what you call your source file, but compile it to an executable called `f`
- Its sole output is either a message telling the user that they didn't run it correctly, or just the number corresponding to the requested fibonacci number
- Allowable values for *n* are 1 to 90, inclusive. If the user entered 0 (or lower), or if the user entered a number greater than 90, it must display the aforementioned error message, and then terminate on an error (i.e. the main function should return `1`)
- It should go without saying, but the calculated fibonacci number must be *correct*
    - Beyond that, you are free to calculate it however you choose

Since you're specifying the number via command-line, don't forget to include `cstdlib`, check the number of arguments (the first parameter of the main function, traditionally declared as `int argc`), and use the `atoi` function on the second argument (index 1).

Submission

To demonstrate that you understand the content from this lab, simply demonstrate your command-line fibonacci program. Your demonstrator may wish to try it on different numbers.
Hopefully you realized that 90 is actually a pretty big number.

Note: this is a pretty easy lab, but your assignment or next lab exercise will rely on your completely understanding this week's material, so definitely be careful and pay close attention to what's been provided.

Sample Execution:

```
$ ./f
Invalid usage! Please provide a single argument n, where n is in (0..90]

$ ./f 0
Invalid usage! Please provide a single argument n, where n is in (0..90]

$ ./f 10
55
```

Note: You won't really be able to tell if it's reporting an error or not unless you make use of its return in the Bash shell. There's one really easy way to do this:

```
./f 10 && echo "Success!"
55
Success!
```

Separating two commands with a `;` says to finish the first process before starting the second. However, using `&&` tells it to wait, and to only start the second process if the first process ends without error.