Introduction

Welcome to lesson 3 of our series! I hope that by now you have become accustomed to the basic syntax of C. If you master this, then you will have little to no difficulty adapting to new syntaxes (I mean other programming languages). C’s explicit insanity helps you understand more of the things that you are doing. For example, when you type:

```
Int x = 5;
```

You know where the line of code starts, and where it ends. You know that *x* is a *variable* (I say this, because it *could* be something other than a variable, if you omit the data type – not in our trusty C of course) of type *integer* and that it has the value *5*.

Other languages can be sort of obfuscated by omitting much of this information. For example, in python you would write x = 5. You could assume that x’s data type will be integer, but not without diving through python’s documentation about what the interpreter does, when it gets to that line, but that’s a tale for a different day.

Enough with my rumblings! Let’s get to the interesting stuff!

We will begin by introducing the concept of *branches*. What is a “branch” you say? Well, let’s pause for a minute to visualize this. Picture that the main flow of the program (let’s assume it’s the function main) was a tree, then what would a *branch* be? A detour to somewhere else!

Kind of confusing? Here is a visual aid!

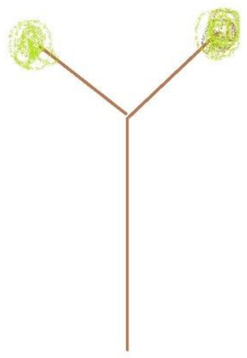
Here is a gorgeous tree!

And here is the result of my bad editing skills! 😊

If we assume that the program starts executing the first command at the roots, and then *slowly* (depending on the program as well as the hardware that you have) makes its' way to the leaves, you can hopefully "see" why this *detour* is called a "branch".

But hang on a second. Are we going to be building such gorgeous trees today??

Well, this tree right here is more like what we will be building today:



I could hear your laughter from here.

You shouldn’t forget that as small as it may be, it is still a plant! So don’t forget to water it regularly and one day it will grow up to match its’ most appealing cousins!

How much the tree will grow, is up to you!

How about a little watering?

The conditional statement

Let’s say we have an integer. And we want to write a user defined function that accepts this integer as an argument and returns the absolute value. How might we do this? Can you think of a way?

Well, the simplest thing I can think of, is this pseudocode:

```
#include <stdio.h>

int get_abs(int x){
      // if x > 0 then return x
      // if x < 0 then return -x
      // if x == 0 then return 0
}

int main(void){

}
```

And sure enough, it is almost this easy to translate these comments into usable code! The general syntax is this:

```
if (<one or more conditions>){
	// code to execute when conditions are met
}else if(<one or more conditions>) {
	// code to execute when conditions are met
}// we can add here as many else if's as we need
else {
	// code to execute if you got to this point
}
```

The code execution begins at the top. First, the conditions are evaluated. If they are “true” (i.e., the conditions are met), then the code in the first curly brackets (everything between the first “{“, and “}”) gets executed. If the conditions of “if” are false (i.e., the conditions are not met), then the execution moves to the next “else if” it can find. Then the new conditions get evaluated, so on and so forth. If it runs out of “else if”s then it executes the code in “else” block. If there is no “else” block, then nothing gets executed.

Caution! Only *one* block of code gets executed.

Let’s see the above use case in action.

```
#include <stdio.h>

int get_abs(int x){
	// if x > 0 then return x
	if (x > 0){
		return x;
	}
	// if x < 0 then return -x
	else if(x < 0){
		return -x;
	}
	// if x == 0 then return 0
	else if (x == 0){
		return 0;
	}


}

int main(void){
	printf("%d\n", get_abs(-4));
	printf("%d\n", get_abs(6));
```

```c
      printf("%d\n", get_abs(0));

      return 0;
}
```

These are 3 new operators! They are called *logical* operators:

1. “Less than” or “<”
2. “Greater than” or “>” and
3. “Equals” or “==” (notice that there is also the assignment operator “=” which is different)

Can you guess what will be the program’s output?

If you guessed “4\n6\n0\n” you are correct!

Notice that I omitted the “else” block and instead used only “else if”s. Can you think of a different way to write this program, so that it produces the same output?

I can think of these alternatives:

```c
//---------------Variation #1---------------

// if x > 0 then return x
if (x > 0){
      return x;
}
// if x < 0 then return -x
else if(x < 0){
      return -x;
}
// if x == 0 then return 0
else{
      return 0;
}
```

```
//---------------Variation #2---------------

// if x > 0 then return x
if (x > 0){
     return x;
}
// if x == 0 then return 0
else if(x == 0){
     return 0;
}
// if x < 0 then return -x
else{
     return -x;
}
```

```
//---------------Variation #3---------------

// if x > 0 then return x
if (x > 0){
     return x;
}

// if x < 0 then return -x
if(x < 0){
     return -x;
}

// if x == 0 then return 0
if(x == 0){
     return 0;
}
```

And there are probably a few dozen (or more!) ways of doing this.

As a sidenote, I should mention that these logical operators also exist:

1. “Greater or equal” or “>=”
2. “Lesser or equal” or “<=”
3. “Not equal” or “!=”

For a complete list check this link out: https://www.tutorialspoint.com/cprogramming/c_operators.htm

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

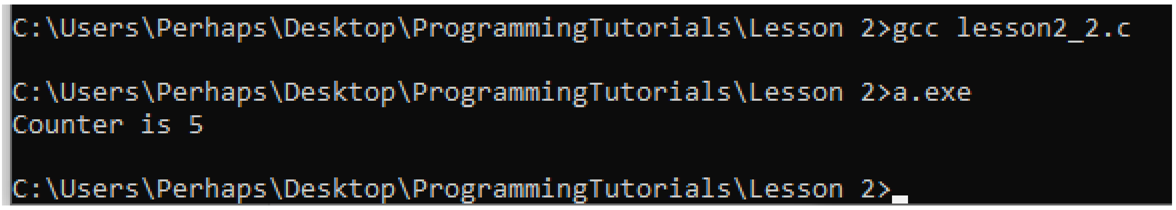
And now, of to the next section of our lesson!

For loops

What if (pun intended) you needed to do a specific operation, let’s say incrementing a counter, 5 times? Okay, this is manageable, you would just write something like this:

```c
#include <stdio.h>

int main(void){
      int counter = 0;

      counter++;
      counter++;
      counter++;
      counter++;
      counter++;

      printf("Counter is %d\n", counter);
}
```

Which prints out to the console:



```
C:\Users\Perhaps\Desktop\ProgrammingTutorials\Lesson 2>gcc lesson2_2.c

C:\Users\Perhaps\Desktop\ProgrammingTutorials\Lesson 2>a.exe
Counter is 5

C:\Users\Perhaps\Desktop\ProgrammingTutorials\Lesson 2>
```

Oh silly me, I haven’t formally introduced you to the “++” operator!

There are 2 more arithmetic operators, “++” and “--".

The following statements are equivalent:

1. i++ and i = i + 1
2. i-- and i = i -1

For a complete list of operators, you can always consult your favorite search engine!
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Now back to our example. It is simple enough. But what if you needed to count to 1.000.000? 10.000.000? 1.000.000.000.000.000?

It would probably take you hours, days or even years to write all these lines of code, incrementing the counter one step at a time.

Turns out there is a simpler way of achieving this. Loops!

There are 3 types of loops. We will discuss *for loops* next:

The syntax of a for loop goes like so:

```
for( int i = start; i < end; i++){
     // usually we use i, j, k etc as the counting variable's name
     // but you can use any name, same as declaring any other variable
}
```

Can you solve the problem of counting to 1000, utilizing a for loop without looking any further?

Hint: use a for loop! Then, print the value in the console!

Simple Solution:

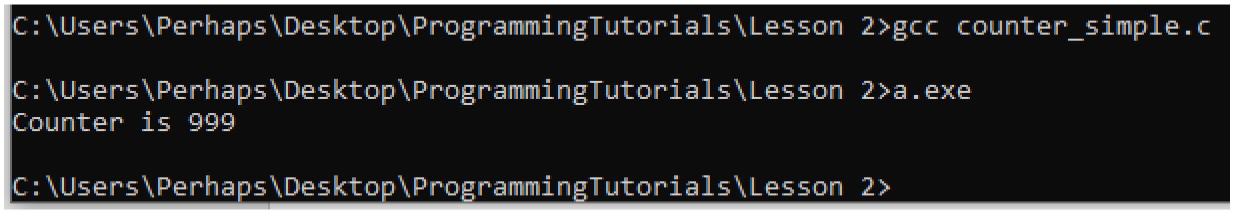
```
#include <stdio.h>

int main(void){
     int counter;
     for( int i = 0; i < 1000; i++){
          counter = i;
     }

     printf("Counter is %d\n", counter);

     return 0;
}
```

More advanced solution:

```c
#include <stdio.h>

int main(void){
	// if you want the variable "i" to be available outside
	// of the for loop, you need to declare it like so:
	int i = 0;
	// and then, omit the "int i = start" statement like this:
	for( ; i < 1000; i++){

	}

	printf("Counter is %d\n", i);
}
```

Pop Quiz! Did you run it? If so, what does it print to the console?

Answer:



```
C:\Users\Perhaps\Desktop\ProgrammingTutorials\Lesson 2>gcc counter_simple.c

C:\Users\Perhaps\Desktop\ProgrammingTutorials\Lesson 2>a.exe
Counter is 999

C:\Users\Perhaps\Desktop\ProgrammingTutorials\Lesson 2>
```

Do you understand why it counts to only 999, and not 1000?

Hint: check the for loop’s conditional statement!

Solution: The conditional statement is this: “I < 1000” so, when “I” has the value of 999 the execution of the loop will stop.

Well not exactly. The code execution would be like this:

```c
//...
//i = 998;
i = i + 1; // -> i = 999
if( i < 1000 ) // -> True
counter = i //-> counter = 999
i = i + 1; // -> i = 1000
if (i < 1000) // -> False
printf(…)
return 0;
```

~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Next time we will cover the rest of the loop types, and then we'll put what we learned to the test, once again, with some exercises! I hope you found this lesson as interesting as I did! Enjoy the rest of your coffee!

P.S.

Don't forget to water your tree!
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~