

Introduction

Last time we talked about a lot of things. The pages just poured on the (digital) paper! Even though our source code was very little, quite a few concepts were introduced. It shows that programming is a philosophy rather than a specific language, but it is also not as boring as reading some ancient Greek's manuscript 😊!

What will we see today? Let's discover it together!

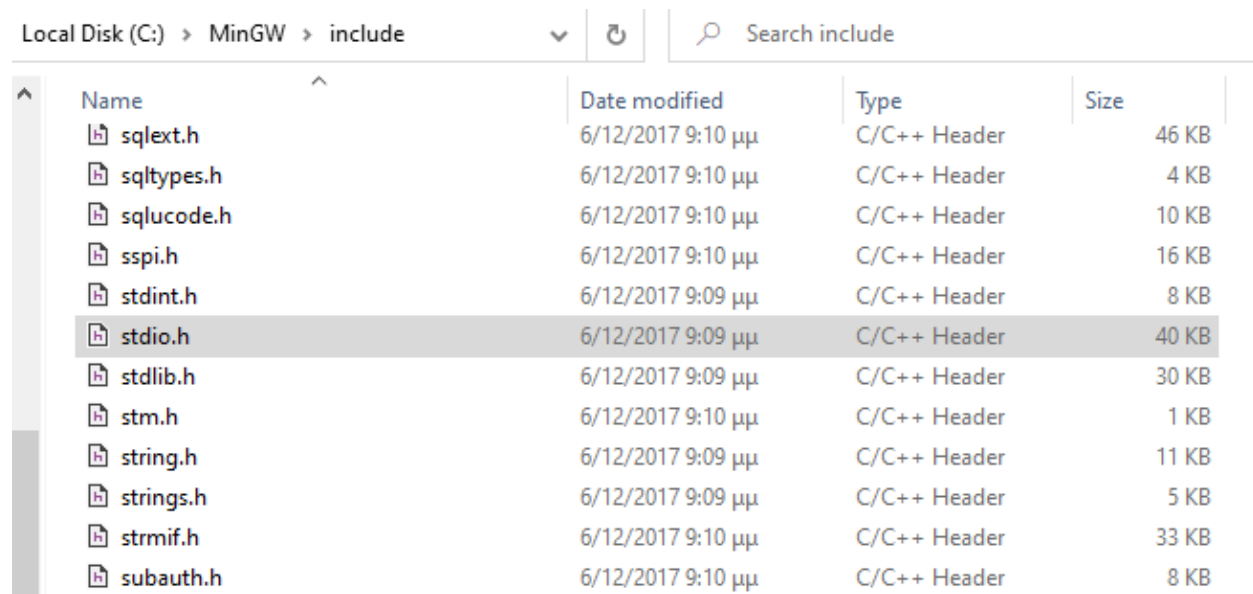
Disclaimer:

The content I post (both here and on GitHub) is a product of my own experience and is shaped according to my knowledge and nothing more.

#include Directives

Let's begin our journey, today, by talking about something that was presented in the last lesson but was not discussed. The first line of code that we (hopefully you too 😊) wrote on our trusty editor was an `#include` statement. These statements are actually called *directives* and they guide the - let's assume - compiler to *include*, i.e. add some code before the rest of our program.

Specifically, the `#include <stdio.h>` directive, includes a *header* file (.h) in our source file. A header file is a file that has the *declarations* of (among other things) variables, constants and functions. Such a function is our trusty `printf()`! Sure enough, if we navigate to `C:\MinGW\include` folder we will find a "stdio.h" file!



Local Disk (C:) > MinGW > include				
Search include				
Name	Date modified	Type	Size	
sqlext.h	6/12/2017 9:10 μμ	C/C++ Header	46 KB	
sqltypes.h	6/12/2017 9:10 μμ	C/C++ Header	4 KB	
sqlucode.h	6/12/2017 9:10 μμ	C/C++ Header	10 KB	
sspi.h	6/12/2017 9:10 μμ	C/C++ Header	16 KB	
stdint.h	6/12/2017 9:09 μμ	C/C++ Header	8 KB	
stdio.h	6/12/2017 9:09 μμ	C/C++ Header	40 KB	
stdlib.h	6/12/2017 9:09 μμ	C/C++ Header	30 KB	
stm.h	6/12/2017 9:10 μμ	C/C++ Header	1 KB	
string.h	6/12/2017 9:09 μμ	C/C++ Header	11 KB	
strings.h	6/12/2017 9:09 μμ	C/C++ Header	5 KB	
strmif.h	6/12/2017 9:10 μμ	C/C++ Header	33 KB	
subauth.h	6/12/2017 9:10 μμ	C/C++ Header	8 KB	

And if we open it with our editor, we will find a lot of declarations. If you look at the file, and thought, "This is blasphemy!! This is Madness!!" – no it isn't, it is just Spartaaaa!!

Just kidding, it is probably going to be a little overwhelming but don't fret! Let's search (Ctrl+F) for `printf`. We will find this line:

```
343  extern int __mingw_stdio_redirect__(printf)(const char*, ...);
```

Let's ignore most of it 😊. For now, just notice that there exists the name `printf` and then a parenthesis with a `char` argument which infers the declaration of the function.

OK enough with this madness, let's close the `stdio.h` file from the editor, as we won't need it anymore. I just wanted to show you that the `stdio.h` file is essential if you want to use the `printf` function in your program. If you try to use `printf` without including the header file for it, then, the compiler will whine at you for not doing so. In fact, let's try and *reproduce* this error.

Go ahead and write this code snippet:

```
lesson_2.c
1
2  int main(void){
3      printf("Hello again!");
4  }
```

Time to fire up our trusty command line! Compile and...

```
C:\Users\Perhaps\Desktop\ProgrammingTutorials>gcc lesson_2.c
lesson_2.c: In function 'main':
lesson_2.c:3:2: warning: implicit declaration of function 'printf' [-Wimplicit-function-declaration]
  printf("Hello again!");
  ^~~~~~
lesson_2.c:3:2: warning: incompatible implicit declaration of built-in function 'printf'
lesson_2.c:3:2: note: include '<stdio.h>' or provide a declaration of 'printf'
C:\Users\Perhaps\Desktop\ProgrammingTutorials>
```

Compiler is sad.

Warning! Implicit declaration of built-in function 'printf'! This means that he recognizes the function printf but can't remember what it does! Luckily the compiler also remembers where the function is declared and kindly tells you (with a note) to include '<stdio.h>' or... you can provide a *user defined* function declaration for it. Indeed, as you may have noticed, we are the user! Er, how about we just *link* the built-in declaration with an include statement and save us a lot of time and trouble. Go ahead and add the *include* statement like this:

```
lesson_2.c
1  #include <stdio.h>
2
3  int main(void){
4      printf("Hello again!");
5  }
```

It is important that all include statements go to the top of the file. Now let's compile again.

```
C:\Users\Perhaps\Desktop\ProgrammingTutorials>gcc lesson_2.c
C:\Users\Perhaps\Desktop\ProgrammingTutorials>
```

Compiler no longer sad. Compiler happy! Time for a cup of coffee!

~~~~~

## Header and Source files

Ready? OK! Let's continue.

Now we will talk about how to separate the *declarations* from the *definitions*. For this purpose, we will create our very own header file!

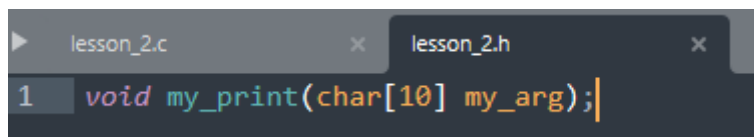
When I talk about the declaration of a function, I mean the place where we declare to the compiler the *name*, the *arguments*, and the *return type* of the function. When I talk about the definition of a function, I mean the place where we define what the function actually does, i.e. we define its' body.

**Exercise:** Write the code for a function named "my\_print" that takes a "string" argument of length 15 and prints it out to the console. Separate the function declaration from its' definition with the help of a header file.

Can you guess how this is going to work? I suggest you pause for a minute or two and give it some thought. You should at least have some idea of how this is going to work by now.

Got it? Great! 😊 Let's try this together now!

In our editor go to File -> New File (or Ctrl+N) and in the page that opens, let's declare our function. Save the file by giving it a name that looks like this: <your\_header\_file\_name>.h



```
1 void my_print(char[10] my_arg);
```

As you can see, now we have 2 files. A .c file and a .h file. So far so good. But how do we *link* the 2 files together? Can you guess?\*

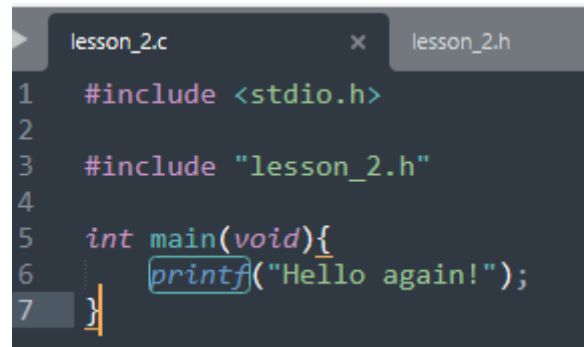
\*Also, if you compile the above snippet, an error is produced. Can you fix it?

If you thought "with an include statement of course!", you are absolutely on the mark! Keep up the good work!

If you didn't know, maybe you should read the previous section again. You had your mind on the coffee all along, didn't you? Well, I don't blame you, I cheated a bit and made my coffee from the beginning of the lesson.

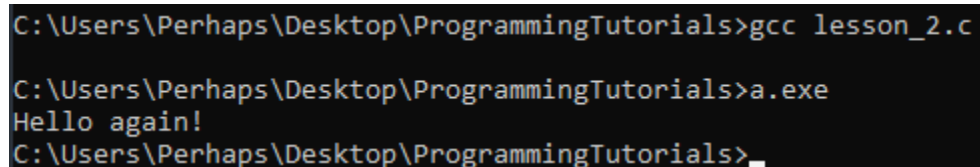
Next time you'll be better prepared, you silly you! 😊

Ok, so on with the lesson! Let's add the include statement. A good question would be "But where do I put the directive? There's 2 files after all!". Short answer is, the *include* directive should be added to the *source* file. So that when you call a function which is defined in the header file, the compiler will know where to look for its' declaration.



```
lesson_2.c x lesson_2.h
1  #include <stdio.h>
2
3  #include "lesson_2.h"
4
5  int main(void){
6      printf("Hello again!");
7  }
```

Did you notice the discrepancy? Indeed. The user defined header file is surrounded by "" instead of <>. This is important, as it tells the compiler where to look for the file. When you use "", you tell the compiler to look in the same directory as the file that is currently processed. While <> suggests that we refer to a library file, that is located somewhere else. Go ahead compile and run the source file.



```
C:\Users\Perhaps\Desktop\ProgrammingTutorials>gcc lesson_2.c
C:\Users\Perhaps\Desktop\ProgrammingTutorials>a.exe
Hello again!
C:\Users\Perhaps\Desktop\ProgrammingTutorials>
```

No pesky errors or warnings! Great!

Now let's actually define and call my\_print() from main().

```

lesson_2.c  x  lesson_2.h
1  #include <stdio.h>
2  #include "lesson_2.h"
3
4  void my_print(char my_arg[15]){
5      printf(my_arg);
6  }
7
8  int main(void){
9      my_print("Hello again!\0");
10 }

```

Compile and Run!

```

C:\Users\Perhaps\Desktop\ProgrammingTutorials>gcc lesson_2.c
C:\Users\Perhaps\Desktop\ProgrammingTutorials>a.exe
Hello again!
C:\Users\Perhaps\Desktop\ProgrammingTutorials>_

```

So far so good. But why bother with the header file? After all, the only code that seems to be run is in the source file. Well, header files are used for *code reusability*. What that means is essentially you write the code once, and then you can reuse it in as many files as you want without much effort. For example, do you remember printf? Somebody wrote the code for that function, and you can just call it with two lines of code. Right now, it may not seem like a big deal, but you will soon realize the significance of this handy work.

~~~~~

Variables: Declaration and Initialization.

Now we will talk about variables. A variable is a *container* (a "bag" if you may) that *holds a value*. The value is *initialized*, usually in the beginning of main (or another function) and *may or may not change* during the execution of a program.

Exercise: let's add to our program, a variable of type `int` and initialize it with the value 5.

To do this we need to know the *type*, the *name* and the *value* of the variable. We know all this information, so at this point all that remains is to show you the syntax:

```
int x = 5;
```

Don't forget about the semicolon! Otherwise the compiler will greet you with a syntax error!

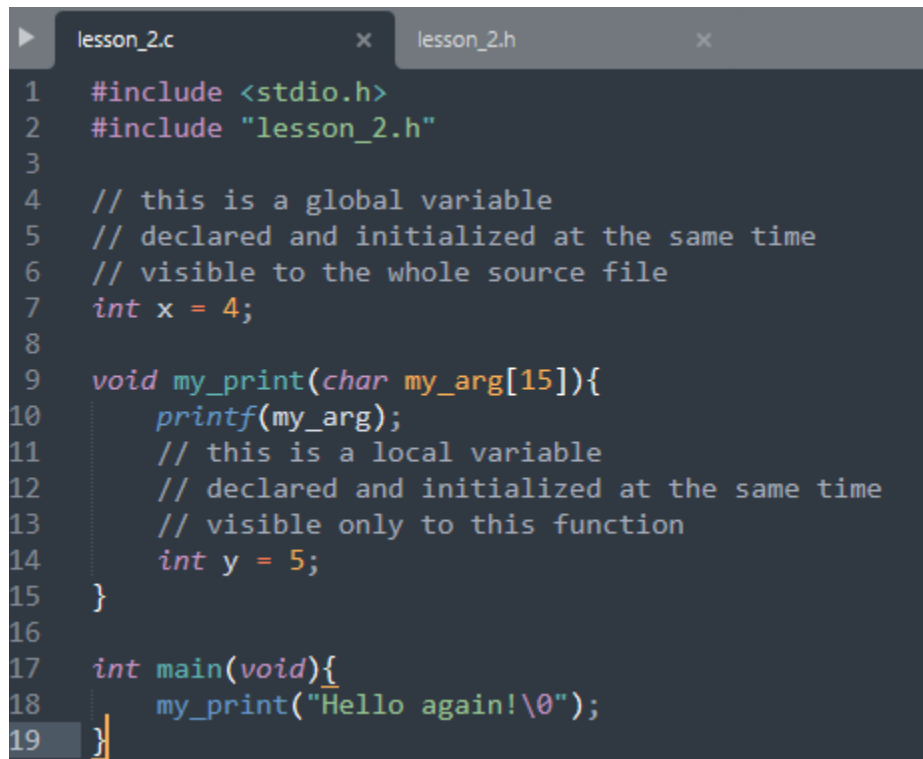
Notice that we *declare* and *initialize* the variable in a single line of code.

Global and local variables

Now, a question arises. Where do I put this line of code? The answer is: *it depends*.

Do you want the variable to be *visible* by the entirety of the source file? In other words, do you want the variable to be a *global*? Or perhaps you just need the variable for a *local* calculation in some function?

If it is the first case and you need a *global* variable, then you should put this line of code "outside" of every function, and if it is the second you should define the variable in the scope of the function that needs it. For example:



```
1  #include <stdio.h>
2  #include "lesson_2.h"
3
4  // this is a global variable
5  // declared and initialized at the same time
6  // visible to the whole source file
7  int x = 4;
8
9  void my_print(char my_arg[15]){
10     printf(my_arg);
11     // this is a local variable
12     // declared and initialized at the same time
13     // visible only to this function
14     int y = 5;
15 }
16
17 int main(void){
18     my_print("Hello again!\0");
19 }
```

Side note: ignore the `///` for now, these are called *comments* and we will talk about them after the break. For now, you just need to know that the GCC compiler, completely ignores every line starting with `///`

Pop Quiz! Can you devise a test scenario for this use case?

Pop Quiz (continued): What will happen if we try to print the values of these 2 variables in main?

```

lesson_2.c  x  lesson_2.h  x
1  #include <stdio.h>
2  #include "lesson_2.h"
3
4  // this is a global variable
5  // declared and initialized at the same time
6  // visible to the whole source file
7  int x = 4;
8
9  void my_print(char my_arg[15]){
10     printf(my_arg);
11     // this is a local variable
12     // declared and initialized at the same time
13     // visible only to this function
14     int y = 5;
15 }
16
17 int main(void){
18     my_print("Hello again!\0");
19     printf("%d", x);
20     printf("%d", y);
21 }

```

Answer: If you guessed "compiler throws an error" you are right!

```

C:\Users\Perhaps\Desktop\ProgrammingTutorials>gcc lesson_2.c
lesson_2.c: In function 'main':
lesson_2.c:20:15: error: 'y' undeclared (first use in this function)
    printf("%d", y);
                  ^
lesson_2.c:20:15: note: each undeclared identifier is reported only once for each function it appears in
C:\Users\Perhaps\Desktop\ProgrammingTutorials>_

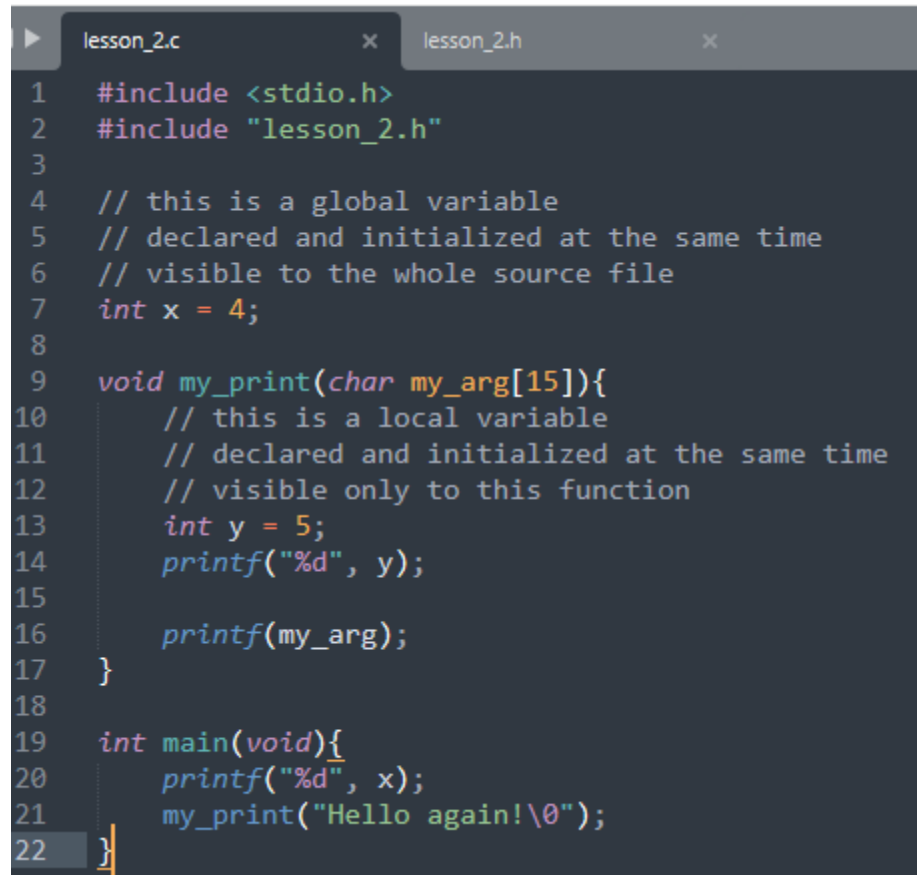
```

As you can see, `y` is declared in function `my_print()` and not `main()`, so when the compiler tries to print the value of `y`, it can't find it in the section of the memory associated with `main()` and therefore throws this error.

Exercise: Can you think of a way to change the source code, so that it prints:

First, the value of x and then the value of y, while x remains a global variable and y local to my_print()?

Here is *one* solution to this problem:



```
1  #include <stdio.h>
2  #include "lesson_2.h"
3
4  // this is a global variable
5  // declared and initialized at the same time
6  // visible to the whole source file
7  int x = 4;
8
9  void my_print(char my_arg[15]){
10     // this is a local variable
11     // declared and initialized at the same time
12     // visible only to this function
13     int y = 5;
14     printf("%d", y);
15
16     printf(my_arg);
17 }
18
19 int main(void){
20     printf("%d", x);
21     my_print("Hello again!\0");
22 }
```

And here is another, a bit more advanced solution, but one that you could totally think of with what you've learned so far from these lessons.

```
lesson_2.c  x  lesson_2.h  x
1  #include <stdio.h>
2  #include "lesson_2.h"
3
4  // this is a global variable
5  // declared and initialized at the same time
6  // visible to the whole source file
7  int x = 4;
8
9  int my_print(char my_arg[15]){
10     // this is a local variable
11     // declared and initialized at the same time
12     // visible only to this function
13     int y = 5;
14
15     printf(my_arg);
16
17     return y;
18 }
19
20 int main(void){
21
22     int main_y = my_print("Hello again!\0");
23     printf("%d", x);
24     printf("%d", main_y);
25 }
```

Notice that I change the return type of my_print(), in both the .c and .h files, (i.e. I modified the declaration of the function) if you don't do this, the compiler will throw a conflict error.

If you got a completely different answer to the pop quiz, don't fret! Well done! The beauty of programming is that there is not a single key that unlocks every door. Usually, you can solve a problem in many ways. Of course, some of the solutions will be "better" (i.e. either are executed faster or use less memory) than others, but that's a subject for another day.

~~~~~

### Clarification

I believe I haven't talked at all about the execution order of the program. In Functional Programming, the order of execution is sequential.

We know that `main()` is the first function that gets called whenever we run `a.exe`. But what happens next? Well, here is a simplified explanation. In our program above, `main` (line 20) gets called and then the function's execution begins. Then, (line 21 is empty so it is ignored by the compiler) line 22 gets executed, then line 23, then 24 etcetera, until it reaches the `"}`" of the function. Can you guess (by looking at the code above) which line of code that is?

Well, it is line 25 of course! When the execution reaches this point, the execution of the function ends, and if it is (like our case) the end of the `main` function, then the program (`a.exe`) ends.

You can think of this process as a car that travels down a straight road and then at some point reaches its' destination and stops.

You might observe that a car can also turn right, left or even go backwards. An analogy for this also exists for a programming concept called *branches*, but we'll talk about this a bit later on.

Side note: I don't know what happens if the car starts to fly, or if you think about an amphibious vehicle. Who knows, maybe you'll learn this, and then tell me about it someday 😊

## Comments

We touched on this subject before the break. Comments are denoted with the `"/"` characters at the start of the line. Whenever the compiler reaches one such line that begins with these characters, it completely ignores it and goes to the next line.

There can also be multiline comments. A multiline comment starts with `"/"` on the first line and ends with a `*/`.

This is a multiline comment:

```
/*  
this is a global variable  
declared and initialized at the same time  
visible to the whole source file  
*/
```

Comments are usefully to other coders when they read a piece of code. Their purpose is to help other people understand what your code does, and/or how it does it. You should get acquainted with them as they are very useful to others. It is a good programming practice to use as little of them, given that their length is enough to explain what needs explaining 😊.

## Data Types: Integers and Chars

Next up, we'll talk about data types. There are many different data types, but we'll talk about two of them\*.

\*For more detailed information you can refer to this site: [https://www.tutorialspoint.com/cprogramming/c\\_data\\_types.htm](https://www.tutorialspoint.com/cprogramming/c_data_types.htm)

We have already seen a few things about chars and integers. Now we will dive a little deeper into this subject. First thing you should know is that for C, everything is represented by an integer, i.e. a whole number at a specific interval. For example, char is a whole number in the interval [-127, 127] which has a length of:

$$\text{Length(char)} = 127 + 1(\text{for zero}) + 127 = 255$$

Also,  $2^8 = 256$  and  $256 > 255$ , so 8 bits (= 1 byte) are enough for representing 1 char.

An integer is 4 bytes =  $4 * 8 = 32$  bits  $\rightarrow 2^{32} = 4.294.967.296$  if we split this in a symmetric interval around 0 we get this interval: [-2.147.483.648, 2.147.483.647].

You may have noticed at this point that  $2.147.483.647 * 2 = 4.294.967.294$  which is not equal to 4.294.967.296. But that's because in some binary representations, 0 has 2 representations. So  $4.294.967.294 + 2 = 4.294.967.296$ .

But enough of that jargon. Let's put what we've learned to the test!

## Overflow: Inf and NaN

Firstly, let's take a look at the integers. Let's make a new file called "lesson\_2b.c" and try to assign a bigger number than the maximum of the above interval and then print the value:

```
lesson_2.c x lesson_2b.c x
1  #include <stdio.h>
2
3  int main(void){
4      int x = 2147483648;
5      printf("%d", x);
6  }
```

Pop Quiz: Can you guess what will happen?

```
C:\Users\Perhaps\Desktop\ProgrammingTutorials>gcc lesson_2b.c
C:\Users\Perhaps\Desktop\ProgrammingTutorials>a.exe
-2147483648
C:\Users\Perhaps\Desktop\ProgrammingTutorials>
```

If you guessed that it "wraps around" to the minimum number that can be represented in this interval, you probably have some background in computer science 😊

This is called overflow. In C *this* is the *predefined behavior* to handle the overflow for integers. In other languages you might encounter *+Inf* (infinity), *-Inf*, or *NaN* (Not a Number) values whenever the value of the variable exceeds the permitted interval.

~~~~~