

## Introduction

This series of lessons aim to be a guide to programming, exploring various concepts, through the utilization of different programming languages, while assuming a nonexistent background in Computer Science/Engineering.

The target audience is, anyone who has little to no experience with programming and who wishes to learn.

This endeavor begins as a journey rather than a course and will most definitely be a learning experience for everyone involved.

### **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.

## Setting up the environment

What do I mean by “environment”? Well, this may mean different things depending on how you look at it, based on the programming language you are going to write on. But we will keep things simple and begin by just setting up any text editor. You can use just about any editor you like depending on your operating system and preferences (even plain old windows Notepad on Windows10 or gedit on Linux) but we will use Sublime for now because it is available on all popular operating systems. (If you have Linux feel free to use gedit).

You can download Sublime from the official website ( <https://www.sublimetext.com/download> ).

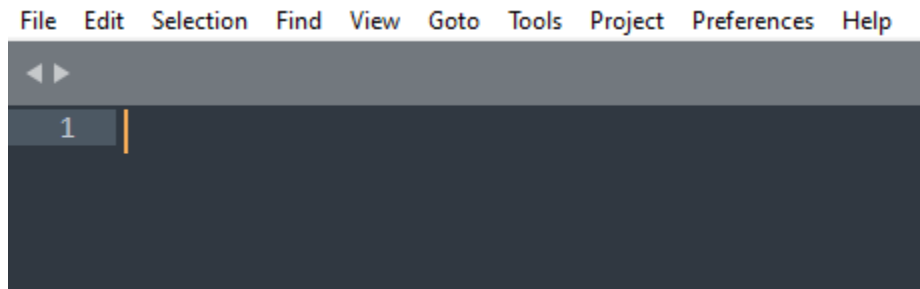
If you are on Windows, you will also need to install gcc (C compiler – more on this later). There are many ways to do this (well, as expected! We are teaching programming here after all! 😊 ) but I suggest you follow these instructions

[https://genome.sph.umich.edu/wiki/Installing\\_MinGW\\_%26\\_MSYS\\_on\\_Windows](https://genome.sph.umich.edu/wiki/Installing_MinGW_%26_MSYS_on_Windows)

Follow the “Installing MinGW” part, although I will assume that you know your way around a computer, feel free to contact me through Facebook if you have any questions or encounter problems at any point of these lessons, I will be happy to help (if I can)!

Now that you have your editor and MinGW installed, go ahead and start the editor.

If you have installed sublime (I will assume this), you should be seeing a blank screen with a welcoming message (there might be other popups/notifications you can go ahead and click “x” on all of them for now).



And so, our journey begins! I hope you are ready for... well the universal start on every known programming language known to man! We are about to tell the computer to tell say "Hello" to the world! Not very exciting? Maybe... But that's where it all begins!

Without further due, let's get to it!

The language we will be using today (and for the first few lessons) will be C. What was that? You wanted python you say? Well, that's totally understandable since it is today's hype. But programming is not about "this" or "that" language, it is a way of thinking, and if you master it, the language that you use won't matter. If you become proficient in let's say... functional programming, you can then use any language that supports functions, you only have to learn the new syntax! (which is not as difficult as it may sound, bear with me and in a few lessons you I will prove it to you!).

Ok, so back to it!

### **Running our first program!**

Let's add these lines of code:

A screenshot of a code editor window. The menu bar at the top includes File, Edit, Selection, Find, View, Goto, Tools, Project, Preferences, and Help. The editor area is dark-themed. A tab at the top is labeled 'hello\_world.c'. The code is as follows:

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

(I will always omit title bars for copyright reasons)

Woah! A lot of jargon aaaand the message! Now let's learn how to run a piece of c code ( a source code file as it is called)!

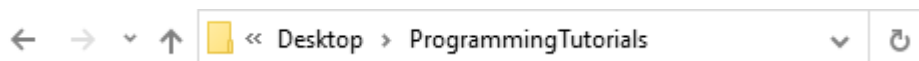
For simplicity's sake I will assume that you are running Windows 10 (imagine if I had to explain this in 20 platforms! Well I don't have 20 partitions on my SSD for all those Operating Systems! But again if you have trouble feel free to contact me and we'll try to sort it out! I'll stop saying this now, and you can just assume it is valid anywhere in these lessons 😊)

So, go to Start -> Search -> cmd and click on “Command Prompt” App. Once cmd opens you should be seeing something similar:

```
Microsoft Windows [Version 10.0.19044.1645]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Perhaps>
```

“C:\Users\Perhaps” is the directory (folder) you are currently in. Let’s “change directory” or “cd” to a folder on the Desktop. Go to your Desktop, create a new folder, name and open it. Click in this dialog box (“ProgrammingTutorials” will be substituted by the name you have given to your folder):



And copy the contents. Then go to cmd again, write “cd ” – with a space after it, right click and paste. Hit enter. You should now be seeing something like this:

```
Microsoft Windows [Version 10.0.19044.1645]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Perhaps>cd C:\Users\Perhaps\Desktop\ProgrammingTutorials
C:\Users\Perhaps\Desktop\ProgrammingTutorials>_
```

If you do, you have successfully changed the “active” folder to the folder you just created. Now, open sublime and type the five lines of code I gave you above. Click File -> Save as... -> save it in the folder you created and give it a (not very original) name of “hello\_world.c”. Now we have to **compile** the *source file* into *machine code*. More jargon? Well what we mean by *compiling* is turning the jargon code from above, into something the computer can understand – zeros and ones – an *executable* file).

To do this, we need to (obviously be in the folder as the .c file and) use the gcc command.

Simply type “gcc hello\_world.c” in the cmd and hit enter. If you have done everything correctly you should be seeing – nothing! :

```
Microsoft Windows [Version 10.0.19044.1645]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Perhaps>cd C:\Users\Perhaps\Desktop\ProgrammingTutorials
C:\Users\Perhaps\Desktop\ProgrammingTutorials>gcc hello_world.c
C:\Users\Perhaps\Desktop\ProgrammingTutorials>_
```

If you now type the command “dir” (which shows the contents of the current folder) you should see an additional file that was just created “a.exe” (or “a.out” if you are on linux)

```
C:\Users\Perhaps\Desktop\ProgrammingTutorials>dir
Volume in drive C has no label.
Volume Serial Number is [REDACTED]

Directory of C:\Users\Perhaps\Desktop\ProgrammingTutorials

28/04/2022  06:03  μμ    <DIR>          .
28/04/2022  06:03  μμ    <DIR>          ..
28/04/2022  05:32  μμ              29.869  0.Setting_Up_The_Environment.docx
28/04/2022  06:00  μμ              40.766  a.exe
28/04/2022  06:03  μμ    <DIR>          Docs
28/04/2022  05:29  μμ              66 hello_world.c
               3 File(s)              70.701 bytes
               3 Dir(s)        6.063.669.248 bytes free

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

Now (finally!) we can run the executable file by typing its’ name (or “./a.out” if you are on linux) :

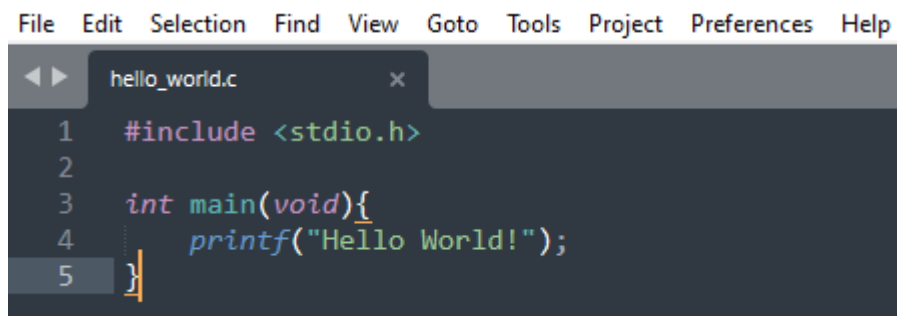
```
C:\Users\Perhaps\Desktop\ProgrammingTutorials>a.exe
Hello World!
C:\Users\Perhaps\Desktop\ProgrammingTutorials>_
```

Hurray! We got the computer to talk to the world! How awesome is that? Not very exciting, probably... But if you think about how much effort has been put into constructing the piece of hardware you are reading this on, you would be as amazed as I am!

And now the *actual learning* begins!

### Understanding what we did!

The code we ran is these five lines:



```
File Edit Selection Find View Goto Tools Project Preferences Help
hello_world.c x
1  #include <stdio.h>
2
3  int main(void){
4      printf("Hello World!");
5  }
```

Let’s start simple. Line 4 says “printf” which means print to the standard output (which means cmd for us at this point) the *string* between the two parentheses, i.e., the *argument* “Hello world!” of the

*function* printf. This line is one of the most basic *commands* you will ever use, and which I still use in different languages, for “finding out what is wrong with my program” (de-bug-ging).

Ok, maybe it was not that simple. Recap:

Function: a name followed by parentheses – e.g. *printf( )*

String: the stuff between the <“ ”> which is understood by the compiler as plain text and not source code.

Argument: the stuff between the parentheses – e.g. a string or *void* if the function takes no argument.

A command is terminated by a semicolon “;”. This means that, a semicolon is typed at the end of every command, so that the compiler knows that the command is over, and a new one starts (at the next line, if you want your code to be readable by humans as well as the machine 😊).

What I mean by this is that the following is perfectly OK for the compiler:

```
hello_world.c x
#include <stdio.h>

int main(void){
    printf("Hello World!");printf("What ");printf("a ");printf("nice ");printf("day!");
}
```

```
C:\Users\Perhaps\Desktop\ProgrammingTutorials>gcc hello_world.c
C:\Users\Perhaps\Desktop\ProgrammingTutorials>a.exe
Hello World!What a nice day!
C:\Users\Perhaps\Desktop\ProgrammingTutorials>
```

But compared to this (which will also have the same output if it is run – you can test this on your own!):

```
hello_world.c x
#include <stdio.h>

int main(void){
    printf("Hello World!");
    printf("What ");
    printf("a ");
    printf("nice ");
    printf("day!");
}
```

I’ll let you decide which form is more readable by humans!

(On a side note, every time you change something in the code, you have to save the source file, and re-compile it, as you can see above, in order for the changes to have an effect. The file that you run will still be a.exe, but every time you compile the code, the compiler replaces the a.exe file with a newly generated one!)

Off to the next line! Line 3 it is! Let's ignore the "int" and the "{...}" for now.

Pop quiz! You should be able to identify what main(void) is.

So?

Answer: If you thought "a function", you guessed it right! (and you should have guessed it right, given that you correctly understood the above pages – if not, you might want to re-read pages 4 and 5).

Let's resume! *Main*, is exactly that, the main function that is run when a c source file is executed. I said a "c" source file is executed, but this knowledge is transferrable to every other programming language out there! Whenever you hear the word "main" from now on, you will know that we are talking about a *function* that executes (runs) our code!

But what about the "{...}" (curly brackets) ? Well, these define the *scope* or the *range* of the function i.e. the code that will be executed whenever main is executed (*called*). Every line after the "{" and before the "}" is called the function's *body*.

And "what about the 'int' part?" you say? Well, a function executes line by line from the top to bottom of its' body. And then? How will we know that the execution of the function is done? Well, we know, because the function *returns* a value (an int-eger in this case).

"But I did not see anything being 'returned'!" you might say. Let's fix that.

**Exercise! Write (*define*) a function named "my\_print()" that takes a string argument, prints the string and returns an integer. Then, call it from *main*.**

Daunting? Well it is not, and you will be able to do this in just a few minutes from now!

Let's break it down to smaller problems.

First and foremost, how do you *define* a function? You have already defined main! How about now? Can you solve this smaller problem? Feel free to peek at the code above and try to define the function without looking at the answer in the next page.

Answer: 5 points to Gryffindor if you typed something like this:

```
my_print("Hello from the other... fuuuunction!"){  
}
```

10 points if you made a cheesy joke like mine 😊

But is this correct syntax? Does the compiler understand it? Let's ask! Place the above function in the source file (start typing at line 2 or 3 – location matters we'll talk about this later) and compile it, like before and see the output.

Congratulations, you produced your first error! Oops!

```
C:\Users\Perhaps\Desktop\ProgrammingTutorials>gcc hello_world.c  
hello_world.c:3:10: error: expected declaration specifiers or '...' before string constant  
my_print("Hello from the other... fuuuunction!"){  
      ^~~~~~  
C:\Users\Perhaps\Desktop\ProgrammingTutorials>_
```

Now we shall commence debugging! Debugging is the process of trying to find out what went wrong when you don't get the expected output from your program, or in general when compiling returns errors (hopefully, without screaming your lungs out at the screen after many hours of googling).

First of all, google the error. Seriously. Google. The. Error. Then, google some more. Googling will be the powerhouse of your cell. What? I mean Googling will be your bread and butter when trying to solve errors like this. At first it will be difficult but you will get the hang of it eventually.

What was that you said? You tried googling the error and nobody had the answer ready for you? Well it happens. Now try googling “define a function in c”. Just joking 😊 (I mean, you can also, do that to get extra information, but I will explain right away).

Let’s read what the compiler says *carefully*. “Expected blah blah blah before string constant”. What is a *constant* you ask? It is a value (*of type* int, string etc..) that does not change during the execution of the program. It is an *immutable* value – it can’t *mutate* (change). If you notice the output there is a “^” beneath the string argument of your function, which underlines where the error originated. So, the string constant must be the “Hello from the other... fuuuuunction!”. AHA! Getting somewhere. Maybe if we tell the compiler, the type of the argument (string), it will actually do its’ job and compile our program? Let’s type:

```
my_print(String "Hello from the other... fuuuunction!") {  
}
```

And compile:

```
C:\Users\Perhaps\Desktop\ProgrammingTutorials>gcc hello_world.c  
hello_world.c:3:17: error: expected ')' before string constant  
my_print(String "Hello from the other... fuuuunction!") {  
                ^~  
C:\Users\Perhaps\Desktop\ProgrammingTutorials>
```

Tough luck! I’ll save you the looong trip I had in order to find out (when I was first starting out) that C has an immense problem with data of type “string”. Instead, C expects all text to be char-acters i.e. of type “char”.

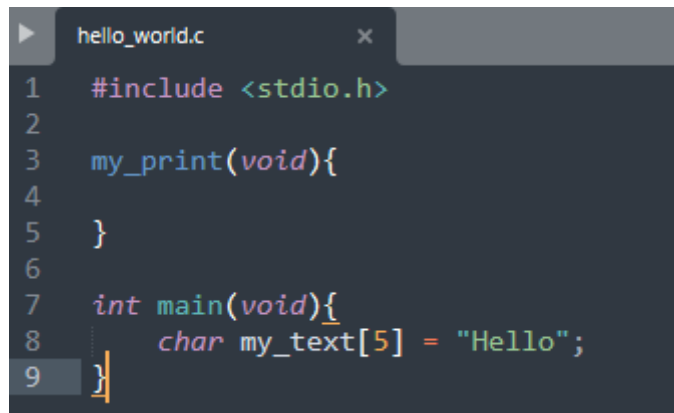
Now might be a good time to pause for five minutes – maybe have a snack?, because we are going to talk about *arrays* next.

~~~~~



### Defining a function - Char data type and Char array

What is a char? It is a single letter (upper or lower case), number or special character like “b”, “v”, “8” and “@”. So if we want to define a “string constant” we have to define a char *array* i.e. a specific number of characters. For example, if we want to store the string “Hello” we need 5 characters, so we type `char[5]` i.e. we *define* a 5 *element array*. Let’s do that inside our main function. We do that by *declaring* (defining) a *data type* (integer, char etc), a *name* so that we can “call” our array wherever we need to, the *size of the array* inside square brackets (e.g. [5]) and a *value* (optional). Also, let’s change the argument of `my_print()` to void for now, like so:



```
hello_world.c x
1  #include <stdio.h>
2
3  my_print(void){
4
5  }
6
7  int main(void){
8      char my_text[5] = "Hello";
9  }
```

Pop Quiz! Can you identify each component of the command at line 8?

Answer:

Data type: char

Array name: my\_text

Array size: 5

Array value: Hello

Compile! And...

```
C:\Users\Perhaps\Desktop\ProgrammingTutorials>gcc hello_world.c
hello_world.c:3:1: warning: return type defaults to 'int' [-Wimplicit-int]
my_print(void){
^~~~~~

C:\Users\Perhaps\Desktop\ProgrammingTutorials>a.exe

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

Congratulations, you produced your first warning! But more on that later. Hooray! The program runs! Doesn't do anything (are you sure...?) yet though. But does it? What do you think?

The answer is the program *does* something. You just don't see it output anything in the console. The program defines 2 functions. Our function and main. Also, it defines an array of 5 char elements and stores the value "Hello" to it. Woah that's a lot of *nothing* isn't it?

But what does it mean to *define* something in a program? It means to *allocate memory for it*. But let's take it a step at a time.

Taking a step back, and looking at what we have accomplished: We have defined our function and stored our "Hello" in a char array.

We know the compiler generates a warning, but let's ignore that for now.

Let's alter the code to make our function accept a "string" (more like a char array) as an argument. Can you do that? ...

Let's think about it for a moment. What do we need? A char array. What do we need in order to define an array? A type, a name, a size and a (optional) value. Let's forget about the value for now. Can you alter the code now?

Yes? Awesome! You should have something like this:

```

hello_world.c x
1  #include <stdio.h>
2
3  my_print(char my_arg[5]){
4
5  }
6
7  int main(void){
8      char my_text[5] = "Hello";
9  }

```

It compiles!... with a warning. Let's fix the warning and we will have successfully defined our function!

Let's read the warning first "warning: return type defaults to 'int'". and there is our function (my\_print) underlined. What might that mean?

Let's take a look at our code. Our function is defined at line 3. Let's compare our function definition with main's definition. Is something missing? Oh my! We omitted the return type of our function! As we said before, every function should return a value of a specific type. And as the compiler informs us, given that we didn't specify a return type, it defaults to int. For now, let's make it so our function returns nothing. "nothing", in C, is specified by "void" so can you guess what we have to change, in order to have our function return "nothing"?

We just add void to the definition of our function:

```

hello_world.c x
1  #include <stdio.h>
2
3  void my_print(char my_arg[5]){
4
5  }
6
7  int main(void){
8      char my_text[5] = "Hello";
9  }

```

And compile!

```

C:\Users\Perhaps\Desktop\ProgrammingTutorials>gcc hello_world.c
C:\Users\Perhaps\Desktop\ProgrammingTutorials>

```

Voila! Compiles with no errors or warnings! Oh, what joy!

We have now successfully defined our function! Defining functions varies a bit from language to language (for example python does not need to define a return type, while at the same time, one such function can return a value) but once you understand the “explicit” (verbose) defining of eeeeeverything in C, adapting to other languages won’t be as difficult ever again!

Time to take a breather and take pride in what we learned! Let’s take another 5 minute break, to give ourselves some time to adjust to the new information! From now on, every time you see this sign, you will know it is *that* time again:


~~~~~

## Calling a user defined function and passing arguments

Now it is time to make our code execute our function. How might we do that?

Well, by calling our function from the main function of course! You have seen how we called `printf()`, can you make the generalization and figure out how to call our *user defined* function?

Here is the solution:



```
hello_world.c x
1  #include <stdio.h>
2
3  void my_print(char my_arg[5]){
4
5  }
6
7  int main(void){
8      char my_text[5] = "Hello";
9      my_print(my_text);
10 }
```

Pretty straight forward right? Notice how I wrote `my_text` *inside* the parentheses. I *passed the array as an argument* to our function. Which means that the value of `my_text` is passed to `my_print()` and is now *visible* to it. This is called *call by value* because you pass the *value* of the *variable* (array) to a function.

But what do I mean that the array is now visible to the function? Let's back up one step. Remember when I talked about the *scope* of a function? All the *variables* (i.e. named types - `<type> <name> = <value>;` - ) that are defined inside the body (scope) of a function are all the *memory* that is *visible (available)* to that function.

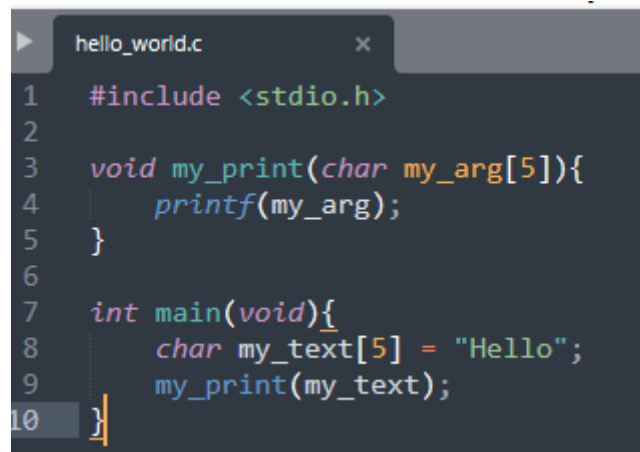
To be a little clearer, in our example, we have defined a variable (array) of type `char` (`my_text`) in the main function. This array is visible to *main only*. If we want to do something with `my_text`'s value in (let's say) our function `my_print()`, such as printing it to the console, we have to pass it as an argument to that function, utilizing *call by value* (or *call by reference*, but this is a bit more advanced concept – we'll talk about this in the next lesson) i.e. passing the value of our array to an array that is visible to `my_print()` – none other than `my_arg` as is defined in our example\*.

\*Not really. In C everything is a pointer, but for educational purposes this explanation will suffice for now, we'll talk more about pointers later on. If you already know about pointers, you can take a look at this: <https://stackoverflow.com/questions/4774456/pass-an-array-to-a-function-by-value>

If you compile the above program, you will see that the compiler returns no errors. So far so good.

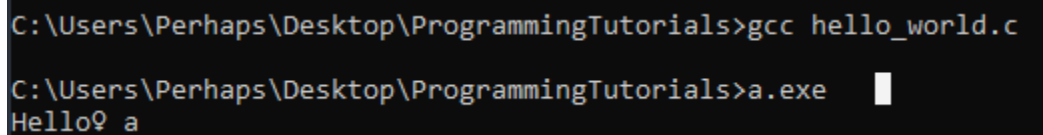
Now we must actually print the “string” argument from inside our function. Can you guess how we might do that?

The solution:



```
hello_world.c
1  #include <stdio.h>
2
3  void my_print(char my_arg[5]){
4      printf(my_arg);
5  }
6
7  int main(void){
8      char my_text[5] = "Hello";
9      my_print(my_text);
10 }
```

Compile, run aaaand...



```
C:\Users\Perhaps\Desktop\ProgrammingTutorials>gcc hello_world.c
C:\Users\Perhaps\Desktop\ProgrammingTutorials>a.exe
Hello? a
```

Weirdness ensues! “Hello” is printed, but what are these weird characters at the end? Well, short answer is that C expects that a “string” is terminated by a ‘\0’ character, which we have not included in our char array definition, therefore, it starts printing but doesn’t know where to stop. Sort-of. Actually there is a limit but this is beyond the scope of the current lesson. For now, we will ignore that this ever happened and just modify our array definitions to include the termination character like so:

```

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

```

Compile, run and:

```

C:\Users\Perhaps\Desktop\ProgrammingTutorials>gcc hello_world.c
C:\Users\Perhaps\Desktop\ProgrammingTutorials>a.exe
Hello
C:\Users\Perhaps\Desktop\ProgrammingTutorials>

```

Problem solved!

As a side note, notice that '\0' is actually 2 characters, but it counts as 1, because the '\' is actually an *escape* character. This means that whatever character follows means "something else". For now, observe that '\0' means that we want to indicate, to the compiler, the end of a string and not the value 0.

All that is left, now, is to make our function return an integer and we are done! Firstly, let's change the definition of my\_print() like so:

```

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

```

If we compile this the usual way, everything seems fine, but if we enable all warnings during compilation with the -Wall argument to gcc (similarly to passing an argument to a function) we get this output:

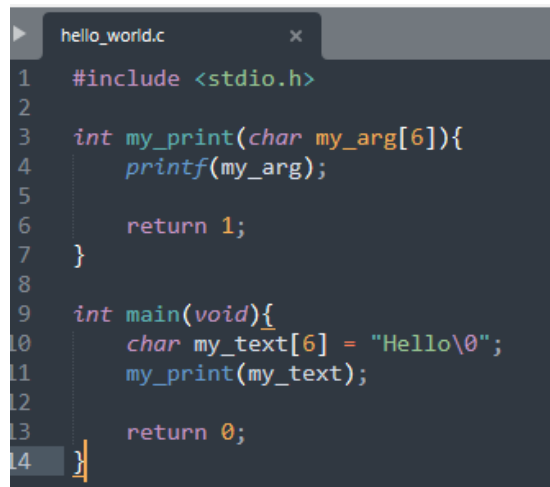
```

C:\Users\Perhaps\Desktop\ProgrammingTutorials>gcc hello_world.c

C:\Users\Perhaps\Desktop\ProgrammingTutorials>gcc hello_world.c -Wall
hello_world.c: In function 'my_print':
hello_world.c:5:1: warning: control reaches end of non-void function [-Wreturn-type]
    }
    ^
C:\Users\Perhaps\Desktop\ProgrammingTutorials>

```

This warning means that while we have defined the function `my_print()` with a return type of `int`, it does not *return* an integer, and the compiler notifies us of that fact. So how do we *return a value*? Well with the *return* statement naturally, like this:



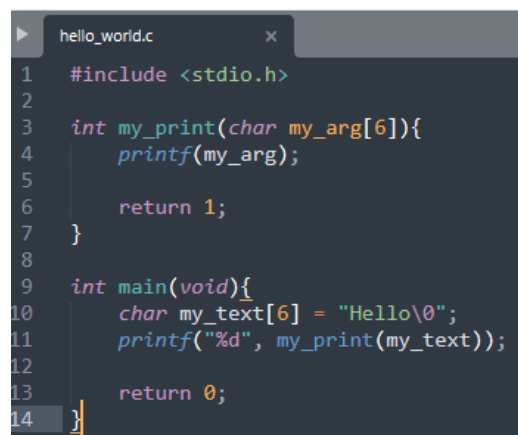
```

hello_world.c
1  #include <stdio.h>
2
3  int my_print(char my_arg[6]){
4      printf(my_arg);
5
6      return 1;
7  }
8
9  int main(void){
10     char my_text[6] = "Hello\0";
11     my_print(my_text);
12
13     return 0;
14 }

```

Notice that I added a `return 0` statement to `main` function. The convention is that if a function returns 0, it is assumed that it has completed its' execution successfully.

To better understand the return value concept let's print the return value of `my_print()` to the console and see what happens!



```

hello_world.c
1  #include <stdio.h>
2
3  int my_print(char my_arg[6]){
4      printf(my_arg);
5
6      return 1;
7  }
8
9  int main(void){
10     char my_text[6] = "Hello\0";
11     printf("%d", my_print(my_text));
12
13     return 0;
14 }

```

Notice that to print an integer with `printf()` we have to pass the argument `"%d"` first and then the integer we want to print.



```
C:\Users\Perhaps\Desktop\ProgrammingTutorials>gcc hello_world.c -Wall
C:\Users\Perhaps\Desktop\ProgrammingTutorials>a.exe
Hello1
C:\Users\Perhaps\Desktop\ProgrammingTutorials>
```

The final thing for this lesson is to notice that `printf` does not terminate the output to the console with a “newline” character. Thus, “Hello” and the return value “1” seem to be joined as one, but in fact are two different things.

If you are reading this, you’ve reached the end of the first lesson! Hooray to you! Now is the time to enjoy the rest of the beverage of your choice that you’ve been sipping on during the 5 minute breaks!

Till next time, keep coding and most importantly have fun doing it! 😊