-std=C11 tells the compiler which C standard to use. The C standard we're using here is C11. That's a modern C standard.
-Wall means to display all warnings. Sometimes the compiler registers a warning because it thinks you did something wrong, but it doesn't break the build command. So in theory, you could still built the executable, but it's safe to always look at these warnings and try to fix them because typically, there's an error in your program.
-fmax-errors=10 tells the compiler well, after 10 errors, stop compiling and throwing more errors because I'll get overwhelmed.
-Wextra means extra warnings that are not included yet in the w all will also be displayed.

gcc -o program program.c
gcc program.c will produce a.out. That is actually our executable program. So we can call that ./a.out.

when we call gcc. The first thing that happens is the preprocessor is called, which means it's a simple textual replacement. For example, the contents of stdio.h is found, which is a text file, and it's actually verbatim copied into your source code. There are other things that the preprocessor does, but we're not going to go into those details. Next, the source code is translated into machine language into a so-called object file. That object file in and of itself cannot be executed. But it is the binary computer readable version of your code.

gcc -c -o program.c program.c : we now, in addition to that, give the compiler an option -c. This flag, -c, tells the compiler I only want to translate into machine language. I do not yet want to create an executable program. So when I run this GCC command, and now I type ls, there is a program.o and along with my program.c. But I can't run that. if I try to look at it, it is a bunch of gibberish. It is the machine language version of my program.

I need to invoke the linker. That is also invoked by simply typing gcc. So gcc is sort of a name for all kinds of things that are happening all at the same time. But I can separate them out. And this is important for separate compilation. I want to specify the name of my output file. This time, I want to create my program. And

gcc -o program(executable) program.o

I already have this object file.

Creating and using a Makefile

How to automate all of the necessary steps to build the final executable program from source code spread out over multiple files.

Now the purpose of a make file is to automate the building of executable programs. We had to compile program.c into an object file program.o. Then we had to separately compile weatherstats.c into weatherstats.o. And then we had to link those two together. So there were really three steps involved. And to have to type all these commands and the compiler flags was kind of annoying.

We can use the program Make-- it's called Make-- to do multiple things all at once. It'll do all of these compiling and linking steps for you. The program Make, which is a program that comes as

part of a Linux distribution, reads instructions from a so-called Makefile And so we're going to create such a file of instructions.

A Makefile has a very specific structure. And the name of the Makefile is also fixed. You can't name it anything else. Here's the fixed structure.
I will first write this in English and then I'll give you an example.

1: First always comes a target. A target is what is to be produced. Then a colon, and then what is needed to do so, then a new line, a tab-- very important, you can't put spaces there, it has to be a tab-- and how to produce what you want to produce.

Producing the executable is my target.
Program(executable): file1.o file2.o : I need to link these two object files together to create my executable.

```
Program: file1.o file2.o
        gcc -std=c11 -Wall -fmax-errors=10 -Wextra file1.o file2.o -o Program
```

My output file is program. So that's why the -o program here in the very end. And the two files that I want to link together are program.o and weatherstats.o. And we're done with this line.

2 : Now we need to also tell the Makefile well, how do you make file1.o. Well, file1.o depends on file1.c as well as file3.h. It needs that h file. Because pound sign includes that h file. And how do I

```
file1.o:  file1.c  file3.h
        gcc -std=c11 -Wall -fmax-errors=10 -Wextra -c file1.c file3.h -o file1.o
```

3: create file2.o :

```
file2.o:  file2.c
        gcc -std=c11 -Wall -fmax-errors=10 -Wextra -c file2.c -o file2.o
```

I simply type Make here at the command prompt, the same thing happens as before. And it even echoes the GCC commands that were executed. So that's nice to see. All three commands were executed.

Now suppose I'm going to only make a little change in file1.c. And that's the only change I make.
And the Make command is actually so smart it realizes that there's nothing to be done. It doesn't have to re-translate other .o extension files. There's nothing to be changed. So when I now type Make here at the command prompt, it only re-translates file1.c into file1.o and then runs the link command. Because otherwise, nothing has changed. Now if I type Make again without making any change, it tells me, there's nothing to make.

So Make automatically checks for dependencies and what things have changed. It only re-translates or re-links those files that actually have to be re-translated. So now you know about these very important Make files that actually allow the automation of the build process of an executable program.

More elaborate Makefile

Creating a static Library

Create your library

Libraries typically consist of a number of object files bundled together into an archive with the extension .a on Linux.

More precisely, this is how you create a so-called static library.

1 : You first compile the files containing the functions into object files, with the exception .o.

2 : Then you bundle these object files together into one archive, and on Linux, that'll have the extension .a.

3 : Such libraries can then be linked to your source code containing your main function to create an executable program.

The linker will only extract those functions from the archive of object files that are necessary and used in your code.

Suppose we wanted to create a weather library. We should probably add more functionality and more .o files

weatherstats.c file contains all the functions we use., and we also have a header file with headers for those functions.

1 : compile weatherstats.c into weatherstats.o.

```
gcc -c  weatherstats.c -o  weatherstats.o
```

2 : To bundle together all the .o files that make up my library, I use the ar command as an archive, and it gets us parameters r, c, and s.

Those are standard parameters, I'll just very briefly mention what they stand for.
R stands for replace, in case a certain .o file is already part of your library. You could replace it if you wanted to.
C stands for creative, doesn't exist yet.
S means create an index for faster access.
We will always choose these parameters, they're very standard.

Next comes the name of the library that I wanted to create or modify. In libraries, typically start with the three letters lib, as in library, and I want my library to be named weather. And the extension is .a, as in a for this archive command.

```
ar rcs  libweather.a weatherstats.o
```

3 : I want to create the executable. I want to invoke the linker.

```
gcc -o out program.o  libweather.a
```

Simply type libweather.a right here, because the library happens to be located in the exact same directory where I find myself.

Different way for step 3.

```
gcc -o result program.o -L. -lweather
```

(-L specifies the library's directory, and . means the current directory)

There is another type of library, namely a so-called dynamic or shared library. What we have created so far, the library with the extension .a, is a so-called static library. Here are the main differences between a static and dynamic library in c.

In a static library, the linker is responsible for finding all used library functions, such as printf, or square root, or such functions, or functions from your own library. And then, to copy all of these into your executable file. So your executable file contains the actual binary code for all functions that it requires.

static library	dynamic library
The linker is responsible for finding all used library functions, such as printf, or square root, or such functions, or functions from your own library. And then, to copy all of these into your executable file. So your executable file contains the actual binary code for all functions that it requires.	The linking happens dynamically at runtime through the operating system. So every program that accesses the library uses the same copy of the library that sits somewhere else in memory, and is only accessed via a link by your code. So your executable code does not contain the actual binaries of the functions it uses, it only contains the names and links of those functions.
Static libraries typically have the extension .a in Linux, as in archive, or .lib under Windows.	Dynamic libraries have the extension .so in Linux, as in shared object, or .dll, as in dynamic link library in Windows.
Since a statically linked library means that the actual function binary is copied into your executable, the executable is automatically a larger file, needing more disk space, and also more space and main memory at runtime.	Dynamically linked library, the executable only contains the name of the library, or a link to the library. So that makes the files smaller. At runtime, the library is only in existence in one location in memory, and your program links to it.
if a library changes, in the static case, the executable does not automatically update. And so, it will keep using the old code, until it's being recompiled and relinked.	In the dynamic case, if a library changes, the executable will automatically use the new library code. But if a library becomes incompatible with your code, so a library updated and updated and at some point, doesn't work with your code anymore.
One big advantage of static libraries that the access is much faster when it is statically linked into your executable program.	Dynamic querying of symbols takes time, and so dynamic library access is slower.

Library	static	Dynamic
Compile library files	gcc -c part1.c -o part1.o gcc -c part1.c -o part1.o ...	gcc -c -fpic part1.c -o part1.o gcc -c -fpic part1.c -o part1.o ...
Create library	ar rcs libmylib.a part1.o part1.o ..	gcc -shared -o libmylib.so part1.o part1.o ..
Compile the main program	gcc -c program.c -o program.o	gcc -c program.c -o program.o
Link library to main program	gcc -o program program.o -L. -lmylib	gcc -o program program.o -L. -lmylib add the library pass to environment variable: export LD_LIBRARY_PATH=$PWD:LD_LIBRARY_PATH

Modify your library

Update library:

ar rcs libweather.a weatherio.o weatherstats.o

link program:

gcc -std=c11 -Wall -fmax-errors=10 -Wextra -o program program.o -L. -lweather

Ultimate makefile