Introduction into Makefiles

Why do I need make?

At the beginning of a sofware project the build process is easy: Just tell the compiler which files to use, where the output should be written and which options (p.ex. Optimization) you prefere. With time your project will not only grow in filesize but also in the amount of files: Some code snippets which are too big to be kept inside your main program or which might also be needed within some other files will be outsourced into a separate file. This external code definition can be included at any other location in your software project. With that you have to pay more attention when you want to rebuild your work: You have to compile all the additional files at first, then the other files that include the precompiled files and in the end your main program. At some point this might get to complicated, time consuming and error-prone to just *type the compile command into your shell one by one until you're done with all files*. By doing it that way you might acidentally forget to rebuild one of the files which might lead to a time consuming debugging process. To make this whole process easier a simple script which contains the set of instructions you want to run takes the weight off your shoulders. This can be done via a Bash-Script or something simular but in terms of development **Make** is a good way to go. This so called build-management-tool is made to automate your build process. Usually the processed files are source code and the call of the `$make` command creates an executable binary but *make* doesn't care about that. You can use *make* to perform any kind of operations where the call of a command is depending of the files that changed. Withing the configuration for your project, which is written inside the *Makefile*, *make* can even handle recursive definitions of jobs made for files depending on jobs made for some other files. One of the most fanyc features from *Make* is that it can figure out wether a source file is newer than its compiled version (*Make* does this by comparing the *last modification timestamp* of the files) and by using *Make* to rebuild the source file it only performs the task where it is needed.

What is a Makefile?

A *Makefile* (yes, that's the name of the file and it has to be written with an uppercase letter at the beginning) contains the set of instructions that are required to execute some tasks p.ex. Building the last version of your code (running compilers, linkers, ... step by step). The execution of `$make` requires a *Makefile* in the current working directory. Within the makefile you define so called "targets". A target is a task to perform p.ex. Compile some code, clean up temporary build-files, installexecutable binaries on your operating system. The definition of a target follows always the same syntax:

```
# a comment begins with a hash sign

<target-name>: <dependency-target-1> <dependency-target-2>
```

```
<task-1>
<task-2>
```

Comments within the *Makefile* begin with a hash sign `#`. At first: The first target in your *Makefile* is the default target. Normally you can run a single target by typing `$make <name-of-target>` into your command prompt. If you do not specify a target the first one in the file is chosen as defualt. After the name of the target a colon `:` follows. Some targets might be depending on the rerults of other targets. This dependencies can be defined in the same line with the target name after the colon. This parameter is optional and can contain multiple entries separated by a space. When this target is called its dependencies are executed first (and recursive) and after that the tasks specified for this target. The tasks are defined as a set of shell commands and will be executed one after another. Every line has to be indented with a Tab.

Example 1

The document you are currently reading is written as a Markdown file and gets converted into a pdf file using the program `pandoc`. This process can be made easier with the following *Makefile*:

```
# a simple example for a Makefile to compile this Markdown Script about Makefiles into a pdf

pdf:
    pandoc Makefiles.md -o Makefiles.pdf

clean:
    rm -f Makefiles.pdf
```

Becaue `pdf` is defined as the first target it becomes the default target. Therefore it makex no difference wether you call `$make` or `$make pdf` on the command line to start the conversion process. The `clean` target will be used to remove the pdf file. In this case it is not really necessary except for an example purpose. Usually bigger software projects contain a `clean` target to remove files generated during the build process. `pdf` and `clean` have no targets they are depending on.

Example 2