How *make* and *Makefiles* work in C

Make is an useful CLI tool which is used to speed up the development process in C. More particularly it is used to speed up compiling.

As you may know the usual way to compile a .c source file would be something like:

```
clang -o hello helloworld.c
```

But this is a very simple scenario and in most cases you also want to use some compiler flags and so it might look like this:

```
clang -ggdb3 -O0 -std=c99 -Wall -Werror -o hello helloworld.c
```

Woah, that is a lot to type in the terminal every single time you need to compile that program. To make matters even worse, imagine if you are working on a larger project which contains more than one source file (and maybe even some header files)!

Thankfully *make* can help us with this, so instead of typing the above you could simply type:

```
make hello
```

And the program *make* would compile your program using all the flags above. But how does *make* know how to make your program (which flags to use etc.)? By looking at a *Makefile*. A Makefile is simply a text document used by *make* which contains information about how to compile a program. This means that you can create a text document where you tell *make* what to do when you run it.

Creating and using a Makefile

Let us start with a simple example: We have the source file ”helloworld.c” and want to *make* a program from it called ”hello”. In order to create a Makefile just open up your favorite text editor and save a new text file as ”makefile” or ”Makefile” in the same directory (folder) as your source code (, that is where you always put Makefiles). Make sure that there is no file ending to the name (no ”.txt” or similar). So now we have the files ”helloworld.c” and ”Makefile” in the same directory.

Now starts the fun part - telling *make* what to do! First we need to figure out what we actually want to happen. What we want is to have helloworld.c compiled into the executable file hello using a couple of compiler flags. In a Makefile we would call the executable file ”hello” a *target*, the ”helloworld.c” a *prerequisite* or *dependency* and the way to compile our source code to an executable a *recipe*. The dependancies are simply the files needed to create the target, like .c .h or .o files

In general it looks like this:

```
1.  target: prerequisites
2.      recipe
```

In our Makefile it would look like this:

```
1.  hello: helloworld.c
2.      clang -ggdb3 -O0 -std=c99 -Wall -Werror -o hello helloworld.c
```

Let’s break it down. This means that whenever we run ”make hello” at the CLI, *make* will check if target (the executable file) does not exist - if it doesn’t it will compile the program to ensure that it does exist. Compilation will also occur if there are any dependancies that are newer than the target (meaning you have updated your code). So if no dependancies are newer than the target the recipe will not be executed (- it is not necessary to recompile your program if you have not changed the source code).

Another usual thing to include in a Makefile is a *clean* rule. It is used to get rid of everything that running *make* created and can in our case look like this:

```
1.  hello: helloworld.c
2.      clang -ggdb3 -O0 -std=c99 -Wall -Werror -o hello helloworld.c
3.
4.  clean:
5.      rm -f hello
6.
```

If we want to use multiple files in our project the solution would be somewhat different - here is the first approach:

```
1.  hello: helloworld.c hellofunctions.c
2.      clang -ggdb3 -O0 -std=c99 -Wall -Werror -o hello helloworld.c
    hellofunctions.c
3.
4.  clean:
5.      rm -f hello
```

As you can see we simply added another dependency and modified the recipe to use ”hellofunctions.c” as well as ”helloworld.c” when compiling with clang. Let’s clean this up a bit by using variables:

```
1.  SRCS = helloworld.c hellofunctions.c
2.  CFLAGS = -ggdb3 -O0 -std=c99 -Wall -Werror
3.
4.  hello: $(SRCS)
5.      clang $(CFLAGS) -o hello $(SRCS)
6.
7.  clean:
8.      rm -f hello core
```

Looks better doesn’t it? Now we don’t have to type our dependencies in two places! There is still one thing that could be better though: If we update one of our dependencies we don’t really need all of the other dependencies to be recompiled aswell - it’s a waste of time. A better approach would be to work with object files. Each source file can be compiled to an object file and then all of the object files can be linked together to form the target. That way only one source file has to be recompiled (to an object file) in case it is newer than the target. This is how it would work for an example:

1. We have ”helloworld.c” and ”hellofunctions.c” and run *make hello*.
2. *Make* looks at the dependencies for ”hello” which have been defined in the Makefile to be ”helloworld.o” and ”hellofunctions.o”. Each of these object files have dependencies themselves; ”helloworld.o” depends on ”helloworld.c” and ”hellofunctions.o” depends on ”hellofunctions.c”. *Make* checks if any of the .c files are newer than the target and if so compiles that .c file to a .o file. (E.g: ”helloworld.c” is newer than ”hello”, therefore ”helloworld.c” is compiled to ”helloworld.o”)
3. ”helloworld.o” and ”hellofunctions.o” are linked together - the executable is created.

Here is the what the example would look like in code, the '#' is used for comments:

```
1.   # variables
2.   SRCS = helloworld.c hellofunctions.c
3.
4.   # this is equal to "OBJS = helloworld.o hellofunctions.o"
5.   OBJS = $(SRCS:.c=.o)
6.
7.   CFLAGS = -ggdb3 -O0 -std=c99 -Wall -Werror
8.
9.   # default target, depends on object files
10.  hello: $(OBJS)
11.      clang $(CFLAGS) -o hello $(OBJS)
12.
13.  # recompile .o files using CFLAGS if outdated, dependencies are implicit
14.  $(OBJS):
15.
16.  clean:
17.      rm -f hello *o
```

The way *make* would execute this Makefile would be the following:

A. Initialize all the variables on line 2-7

B. On line 10, look at the dependencies for "hello"

C. Apparently the dependencies "$(OBJS)" for "hello" have their own dependencies as *make* sees on line 14. Line 14 only consists of targets (a list of object files) - there are no dependencies or recipe, how is this? Well, when not typing any recipe and when using .o files as targets *make* will automatically assume that the .o file comes from a .c file with the same name and will compile that .c file using CFLAGS to the .o target. *Make* will go through all object files listed in $(OBJS) and if necessary compile them.

D. *Make* jumps back to line 10 and now sees that the dependencies for "hello" are newer than the target, this means that the recipe for the target should be run.

E. The recipe on line 11 for "hello" is run - the object files are linked together and the executable "hello" is ready to be run!

Now that you hopefully know a lot more about *make* and Makefiles than before - *make* some awesome programs!