# A Simple Makefile Tutorial

Makefiles are a simple way to organize code compilation. This tutorial does not even scratch the surface of what is possible using *make*, but is intended as a starters guide so that you can quickly and easily create your own makefiles for small to medium-sized projects.

## A Simple Example

Let's start off with the following three files, hellomake.c, hellofunc.c, and hellomake.h, which would represent a typical main program, some functional code in a separate file, and an include file, respectively.

**hellomake.c**

```
#include <hellomake.h>

int main() {
  // call a function in another file
  myPrintHelloMake();

  return(0);
}
```

**hellofunc.c**

```
#include <stdio.h>
#include <hellomake.h>

void myPrintHelloMake(void) {

  printf("Hello makefiles!\n");

  return;
}
```

**hellomake.h**

```


/*
example include file
*/

void myPrintHelloMake(void);
```

Normally, you would compile this collection of code by executing the following command:

```
gcc -o hellomake hellomake.c hellofunc.c -I.
```

This compiles the two .c files and names the executable hellomake. The `-I.` is included so that gcc will look in the current directory (.) for the include file hellomake.h. Without a makefile, the typical approach to the test/modify/debug cycle is to use the up arrow in a terminal to go back to your last compile command so you don't have to type it each time, especially once you've added a few more .c files to the mix.

Unfortunately, this approach to compilation has two downfalls. First, if you lose the compile command or switch computers you have to retype it from scratch, which is inefficient at best. Second, if you are only making changes to one .c file, recompiling all of them every time is also time-consuming and inefficient. So, it's time to see what we can do with a makefile.

The simplest makefile you could create would look something like:

Makefile 1

```
hellomake: hellomake.c hellofunc.c
	gcc -o hellomake hellomake.c hellofunc.c -I.
```

If you put this rule into a file called `Makefile` or `makefile` and then type `make` on the command line it will execute the compile command as you have written it in the makefile. Note that `make` with no arguments executes the first rule in the file. Furthermore, by putting the list of files on which the command depends on the first line after the `:`, make knows that the rule `hellomake` needs to be executed if any of those files change. Immediately, you have solved problem #1 and can avoid using the up arrow repeatedly, looking for your last compile command. However, the system is still not being efficient in terms of compiling only the latest changes.

One very important thing to note is that there is a tab before the gcc command in the makefile. There must be a tab at the beginning of any command, and `make` will not be happy if it's not there.

In order to be a bit more efficient, let's try the following:

Makefile 2

```
CC=gcc
CFLAGS=-I.

hellomake: hellomake.o hellofunc.o
     $(CC) -o hellomake hellomake.o hellofunc.o
```

So now we've defined some constants `CC` and `CFLAGS`. It turns out these are special constants that communicate to `make` how we want to compile the files hellomake.c and hellofunc.c. In particular, the macro `CC` is the C compiler to use, and `CFLAGS` is the list of flags to pass to the compilation command. By putting the object files--hellomake.o and hellofunc.o--in the dependency list and in the rule, `make` knows it must first compile the .c versions individually, and then build the executable hellomake.

Using this form of makefile is sufficient for most small scale projects. However, there is one thing missing: dependency on the include files. If you were to make a change to hellomake.h, for example, `make` would not recompile the .c files, even though they needed to be. In order to fix this, we need to tell `make` that all .c files depend on certain .h files. We can do this by writing a simple rule and adding it to the makefile.

Makefile 3

```
CC=gcc
CFLAGS=-I.
DEPS = hellomake.h

%.o: %.c $(DEPS)
        $(CC) -c -o $@ $< $(CFLAGS)

hellomake: hellomake.o hellofunc.o
        $(CC) -o hellomake hellomake.o hellofunc.o
```

This addition first creates the macro DEPS, which is the set of .h files on which the .c files depend. Then we define a rule that applies to all files ending in the .o suffix. The rule says that the .o file depends upon the .c version of the file and the .h files included in the DEPS macro. The rule then says that to generate the .o file, `make` needs to compile the .c file using the compiler defined in the CC macro. The -c flag says to generate the object file, the `-o $@` says to put the output of the compilation in the file named on the left side of the `:`, the `$<` is the first item in the dependencies list, and the CFLAGS macro is defined as above.

As a final simplification, let's use the special macros `$@` and `$^`, which are the left and right sides of the `:`, respectively, to make the overall compilation rule more general. In the example below, all of the include files should be listed as part of the macro DEPS, and all of the object files should be listed as part of the macro OBJ.

Makefile 4

```
CC=gcc
CFLAGS=-I.
DEPS = hellomake.h
OBJ = hellomake.o hellofunc.o

%.o: %.c $(DEPS)
        $(CC) -c -o $@ $< $(CFLAGS)

hellomake: $(OBJ)
        $(CC) -o $@ $^ $(CFLAGS)
```

So what if we want to start putting our .h files in an include directory, our source code in a src directory, and some local libraries in a lib directory? Also, can we somehow hide those annoying .o files that hang around all over the place? The answer, of course, is yes. The following makefile defines paths to the include and lib directories, and places the object files in an obj subdirectory within the src directory. It also has a macro defined for any libraries you want to include, such as the math library `-lm`. This makefile should be located in the src directory. Note that it also includes a rule for cleaning up your source and object directories if you type `make clean`. The .PHONY rule keeps `make` from doing something with a file named clean.

Makefile 5

```
IDIR =../include
CC=gcc
CFLAGS=-I$(IDIR)

ODIR=obj
LDIR =../lib

LIBS=-lm

_DEPS = hellomake.h
DEPS = $(patsubst %,$(IDIR)/%,$(_DEPS))

_OBJ = hellomake.o hellofunc.o
OBJ = $(patsubst %,$(ODIR)/%,$(_OBJ))


$(ODIR)/%.o: %.c $(DEPS)
	$(CC) -c -o $@ $< $(CFLAGS)

hellomake: $(OBJ)
	$(CC) -o $@ $^ $(CFLAGS) $(LIBS)

.PHONY: clean

clean:
	rm -f $(ODIR)/*.o *~ core $(INCDIR)/*~
```

So now you have a perfectly good makefile that you can modify to manage small and medium-sized software projects. You can add multiple rules to a makefile; you can even create rules that call other rules. For more information on makefiles and the `make` function, check out the GNU Make Manual, which will tell you more than you ever wanted to know (really).