

Original Source: <http://mrbook.org/blog/tutorials/make/>

A tutorial by example

Compiling your source code files can be tedious, especially when you want to include several source files and have to type the compiling command everytime you want to do it.

Well, I have news for you... Your days of command line compiling are (mostly) over, because YOU will learn how to write Makefiles. Makefiles are special format files that together with the *make* utility will help you to automagically build and manage your projects.

The make utility

If you run

```
make
```

at the command line prompt, this program will look for a file named *makefile* in your directory, and then execute it.

If you have several makefiles, then you can execute them with the command:

```
make -f MyMakefile
```

There are several other switches to the `make` utility. For more info, `man make`.

Build Process

1. Compiler takes the source files and outputs object files
2. Linker takes the object files and creates an executable

Compiling by hand

The trivial way to compile the files and obtain an executable, is by running the command:

```
g++ main.cpp hello.cpp factorial.cpp -o hello
```

The basic Makefile

The basic makefile is composed of:

```
target: dependencies
```

```
[tab] system command
```

This syntax applied to our example would look like:

```
all:
```

```
    g++ main.cpp hello.cpp factorial.cpp -o hello
```

To run this makefile on your files, type:

```
make -f Makefile
```

On this first example we see that our target is called *all*. This is the default target for makefiles. The *make* utility will execute this target if no other one is specified.

We also see that there are no dependencies for target *all*, so *make* safely executes the system commands specified.

Finally, make compiles the program according to the command line we gave it.

Using dependencies

Although this is not necessary for project submission, it is sometimes useful to use different targets. This is because if you modify a single file in your project, you don't have to recompile everything, only what you modified.

Here is an example. Assume the following is contained in a file named Makefile:

```
all: p1
```

```
p1 : main.o factorial.o hello.o
```

```
    g++ main.o factorial.o hello.o -o hello
```

```
main.o: main.cpp
```

```
    g++ -c main.cpp
```

```
factorial.o: factorial.cpp
```

```
g++ -c factorial.cpp

hello.o: hello.cpp

g++ -c hello.cpp

clean:

rm hello.o factorial.o main.o
```

NOTE: ONLY INCLUDE THE OBJ FILES IN THE CLEAN OPTION! YOU DO NOT WANT TO DELETE YOUR .CPP FILES!!!! NEVER USE *rm* *

Now we see that the target *all* has only dependencies, but no system commands. In order for *make* to execute correctly, it has to meet all the dependencies of the called target (in this case *all*).

Each of the dependencies are searched through all the targets available and executed if found.

Thus to compile your project you need only type ***make p1*** or ***make all***.

In this example we see a target called *clean*. It is useful to have such target if you want to have a fast way to get rid of all the object files and executables. **NOTE: ONLY INCLUDE THE OBJ FILES IN THE CLEAN OPTION! YOU DO NOT WANT TO DELETE YOUR .CPP FILES!!!! NEVER USE *rm* ***

Thus, if you wish to remove the .o files and recompile, you can type ***make clean*** and then ***make p1***.

NOTE: ONLY INCLUDE THE OBJ FILES IN THE CLEAN OPTION! YOU DO NOT WANT TO DELETE YOUR .CPP FILES!!!! NEVER USE *rm* *

NOTE: ONLY INCLUDE THE OBJ FILES IN THE CLEAN OPTION! YOU DO NOT WANT TO DELETE YOUR .CPP FILES!!!! NEVER USE *rm* *