Makefile Tutorial

A makefile is a separate file that you have in the directory of your project.
The basic syntax for a Makefile is the following:

```
target: dependencies
[tab] <command to execute>
```

The “target” is the name for a group of commands. They act much like gotos would in other programming languages. If you have a target named “foo”, and you type in `make foo` at a terminal, Make will instantly go to the target named “foo” and execute the commands there. Keep the target that will link the project together as the first target in the Makefile. That way you won't have to specify a target when calling Make from the terminal.

The dependencies are a list of files that the target requires. This will include source and header files (and perhaps even .o files). Make uses this list to check and see if there have been any changes to these files since the last time it was run. If there are no changes, then there's no reason for the target associated with these changes to be run. Make will output a message stating this.

All lines that do not start with a target *must* start with a tab. 4 (or 3 or 8) spaces will not count as a tab in this case. It must start with the tab character.

The part after the tab character is the command that you wish to run for the target. In most cases, it will be something along the lines of `gcc -c main.c`, but it can be whatever you want.

Here is a simple example makefile which will compile a source file:

```
mfTut: main.c
	gcc main.c -o mfTut
```

Here, the target is “mfTut”, the dependency is “main.c”, and the command is “gcc main.c -o mfTut”, which will compile and link main.c, and then create an executable called “mfTut”. Note that when you don't specify an output file with the -o option to gcc, the default is “a.out”.

Note that you can have multiple targets per makefile. Here's an example makefile that does the same thing as the above one, but with two targets:

```
mfTut: main.o
	gcc main.c -o mfTut

main.o: main.c
	gcc -c main.c
```

Here, the target “main.o” has “main.c” as a dependency, so it first checks to see if main.c has been changed since the last time Make was run on this makefile. If it has, then the commands associated with the target “main.o” are executed. The command in this case will compile, but not link, main.c, and give the default output file of “main.o” (or “foo.o” if the source code file was “foo.c”). When compiling multiple source files into one project, you need to have the -c flag on all gcc commands that don't

actually create the executable.
Here's a slightly more complex example with multiple source files:

```
prog2: main.o msgQ.o
   gcc -g main.o msgQ.o -o prog2 -lpthread

main.o: main.c msgQ.h
   gcc -c -g -Wall -pedantic  main.c

msgQ.o: msgQ.c msgQ.h
   gcc -c -g -Wall -pedantic msgQ.c
```

Here, the target “prog2” asks “main.o” and “msgQ.o” if they are up to date, and if not, to run their commands to produce up to date files.

Finally, here's an example of a makefile with a target that runs a command other than gcc:

```
mfTut: main.o
   gcc main.c -o mfTut

main.o: main.c
   gcc -c main.c

#clean up all the object files we made
clean:
   rm *.o
```

If we call `make clean`, we'll have all the object files that we've made deleted. In this case, the .o file generated by the “main.o” target will be deleted. This example also shows how to have comments.

In the previous example, you could also have put the `rm *.o` right under the gcc command for the target “mfTut”, and this will delete the object files after fully compiling and linking them. This is not recommended, since Make will have to recompile and link everything every time you call it.

Another thing you can do is use variables to shorten what you have to type.

```
CC=gcc
CFLAGS=-c -g -Wall -pedantic

prog2: main.o msgQ.o
   $(CC) -g main.o msgQ.o -o prog2 -lpthread

main.o: main.c msgQ.h
   $(CC) $(CFLAGS) main.c

msgQ.o: msgQ.c msgQ.h
   $(CC) $(CFLAGS) msgQ.c
```

This tutorial can be found at the UPL website. Go to http://www.upl.cs.wisc.edu to find it.