`Makefiles`

Using the `make` utility requires a programmer to create a special file named `Makefile`, from which the `make` program reads information. A `Makefile` consists of pairs of lines (each pair governs the updating of one file).

Example: Suppose that the source file for an executable binary file named `project` is stored in a file named `project.cc`. The program `#include`s a library's header file named `lib.h`, and the library's implementation file is named `lib.cc`.

Upon what files does `project` depend?
→ `project.o` and `lib.o`

Upon what files does `project.o` depend?
→ `project.cc` and `lib.h`

Upon what files does `lib.o` depend?
→ `lib.cc` and `lib.h`

A `Makefile` has one pair of lines for each file to be "made", so our example `Makefile` will have three line-pairs (one for `project`, one for `project.o` and one for `lib.o`).

The first line of each line-pair in a `Makefile` has the form:

TargetFile: *DependencyFile*$_1$ *DependencyFile*$_2$ ... *DependencyFile*$_n$

where *TargetFile* is the file that needs to be updated, and each Dependency$_i$ is a file upon which *TargetFile* depends.

The second line of the pair is a UNIX command to make *TargetFile*. The command must be preceded by a TAB and end with a Return.

To illustrate: The first line-pair in our `Makefile` appears as follows:

```
project: project.o lib.o
	g++ project.o lib.o -o project
```

Note that the first line specifies the dependencies of `project`, and the second line is the UNIX command to make `project`.

Of course, `project.o` won't exist the first time we compile, so we should specify a line-pair for it, too:

```
project.o: project.cc lib.h
	g++ -c project.cc
```

We should then do the same thing for `lib.o`:

```
lib.o: lib.cc lib.h
	g++ -c lib.cc
```

The Makefile thus appears as follows:

```
project: project.o lib.o
         g++ project.o lib.o -o project

project.o: project.cc lib.h
         g++ -c project.cc

lib.o: lib.cc lib.h
         g++ -c lib.cc
```

Now, when a user types

```
make
```

the program reads the Makefile, and

1. Sees that project depends upon project.o, and
   a. Checks project.o, which depends on project.cc and lib.h;
   b. Determines whether project.o is out of date;
   c. If so, it executes the command to make project.o:
   ```
   g++ -c project.cc;
   ```
2. Sees that project also depends upon lib.o, and
   a. Checks lib.o, which depends on lib.cc and lib.h;
   b. Determines whether lib.o is out of date;
   c. If so, it executes the command to make lib.o:
   ```
   g++ -c lib.cc;
   ```
3. Sees that everything on which project depends is now up to date, and so executes the command to make project:
   ```
   g++ project.o lib.o -o project
   ```

Details About Make.

1. While a Makefile usually consists of pairs of lines, there can in fact be any number of commands after the line specifying the dependencies.

   Example: We could write:

   ```
   project: project.o lib.o
         g++ project.o lib.o -o project
         rm project.o
         rm lib.o
   ```

This would automatically remove the object files after project is made.

2. The make utility also allows a user to specify what is to be made:

   uname% <u>make lib.o</u>

   will operate using lib.o as its primary *TargetFile* instead of project.

3. A *TargetFile* need not have any dependencies. This, combined with (1) and (2) allows make to be used for all kinds of non-compilation activities.

   Example: Suppose our Makefile contains the following lines:

   ```
   clean:
         rm -f project *.o *~ *#
   ```

   and the user types

   uname% <u>make clean</u>

   What happens? (This is a fool-proof way to clean up a messy directory.)

4. make is coordinated with emacs. When an emacs user types the command:

```
M-x compile
```

emacs responds with

```
Compile command: make -k
```

If a Makefile is in the directory containing the file on which you are working, then pressing the Return key will execute
make using that Makefile.

Summary

The make utility eliminates the complexity of separate compilation by determining what files are out of date and re-making them. Learning to use it effectively can save a great deal of time, especially on projects that have several files.

Observation: There can be only one file named Makefile in a directory.

Conclusion: Since a Makefile coordinates the translation of one project, each project should be stored in its own dedicated directory, with a separate Makefile to coordinate its translation. Doing so allows you to remove the object files, binary executables, etc., because to remake the project, you need only cd to the directory and type make.

An added benefit is that all the files for one project are confined within one directory, making it easier to port the project to a different machine. (Just copy the directory to the new machine, cd to the directory, and type make).