

CS 215 - Fundamentals of Programming II

Very Basic make

"**make** is a command generator."¹ This document explains a small portion of the functionality of **make** sufficient for most courses. If you want to know more, see the books cited in the footnote.

make generally is used for managing software projects (usually in UNIX environments). It allows a programmer to specify the dependencies between various program files and attach commands that generate these files. This can save time and effort since only the files that are affected by a change need to be regenerated, rather than the entire project.

To use **make** you create a *makefile* that describes the dependency relationships among the files in your project and states the commands for generating each file that are not source files. The name of this file is **Makefile** or **makefile**, unless there are to be multiple makefiles in one directory. For the CS 215 class, each project requiring a makefile must be put into its own subdirectory with a makefile named **Makefile**.

A makefile is a plain text file with the following syntax.

```
# Comments start with hash character to the end of line
```

```
target1 : dependencies1
<tab>    command1
```

```
target2 : dependencies2
<tab>    command2
```

```
...
```

A target represents something that is to be generated. The dependencies are those files which are used to create the target. The command is used to create the target when needed.

The target must start in the first column of a line. There can be more than one target listed in a line separated by spaces. This would mean that all of the targets depend on the same files. The dependencies are a list of files separated by spaces upon which the target depends. There may be more than one command to perform for each target, each on a separate line. Each command line must start with the TAB character (**not** spaces). For example, in file named **Makefile** for a video store application, we might have:

```
vidstore : vidstore.o store.o tape.o
<tab>    clang++ vidstore.o store.o tape.o -Wall -o vidstore
```

This says that the target **vidstore** depends on the files **vidstore.o**, **store.o**, and **tape.o**. The command **clang++ vidstore.o store.o tape.o -Wall -o vidstore** is performed to create **vidstore**. This command links together the **.o** object files into an executable file named **vidstore**.

A **.o** object file is the result of compiling a single source file, so it usually depends on a **.cpp** source file and some **.h** header files. Thus it also should be listed as a target with dependencies. For example,

¹ Andrew Oram and Steve Talbott, *Managing Projects with make*, O'Reilly Media, Inc. 1991 (out of print). The current reference book for make is Richard Stallman, *GNU Make Reference Manual*, Samurai Media Limited, 2015.

```
vidstore.o : vidstore.cpp store.h tape.h
<tab>      clang++ -Wall -std=c++11 -c vidstore.cpp
```

could be the next target and says that **vidstore.o** depends on the files **vidstore.cpp**, **store.h**, and **tape.h**. To complete the makefile for this project, there would also be targets for **store.o** and **tape.o**.

To run **make** you just type

```
$ make
```

at the shell prompt. By default, it looks for a file in the current directory named **makefile** or **Makefile** in that order, though most people recommend **Makefile** so that it will be distinctive among lowercase file names. By default, **make** will try to create the first target. You also can ask it to create a particular target explicitly by giving it as a command-line argument. Thus, for our example makefile, typing just **make** will try to create **vidstore**, which may create other files, too, while typing

```
$ make vidstore.o
```

will try to create just **vidstore.o**. Making intermediate targets is useful if you are trying to get a file like **vidstore.cpp** to compile when the rest of your project is not done yet.

When you run **make** the following happens: If the target has dependencies, **make** first checks if the dependencies are up-to-date, before checking the original target. If they are, then it checks if the target is older than the dependencies. If so, it runs the command to create a new version of the target. If not, everything is up-to-date and nothing happens.

If the dependencies are not up-to-date, then they are recreated first. Thus a target that depends on one file that depends on another file that has been modified eventually will cause the first target to be updated. Suppose for the above example, you modify **store.h**. Then when **make** is invoked, it assumes you want to make **vidstore**. Since dependency **vidstore.o** is also a target, **make** checks if the files **vidstore.cpp**, **store.h**, or **tape.h** are newer. Since we've modified **store.h**, **make** will compile a new **vidstore.o**. (This probably also will happen with **store.o**.) Since **vidstore.o** (and **store.o**) is now newer than any existing **vidstore** file, **make** will then also link a new executable.

Choosing targets and dependencies is something of an art. When you write a target, it clearly depends on anything that is listed in the command to bring it up-to-date, but you only need to list the ones that can be changed by you. For example, in code files, any user-defined included header files should also be listed as a dependency. For C++ programs, this generally means you'll have a makefile of the form:

```

# Makefile for program

program : program.o class1.o class2.o ...
<tab> clang++ -Wall -o program program.o class1.o class2.o ...

program.o : program.cpp class1.h class2.h ...
<tab> clang++ -Wall -std=c++11 -c program.cpp

class1.o : class1.cpp class1.h
<tab> clang++ -Wall -std=c++11 -c class1.cpp

...

```

Since typing **make** by itself assumes that you want to check the first target, generally the first target in a makefile should generate the entire program. Anything that is listed as a dependency of the first target that is generated by a program, rather than something you wrote explicitly, also should be a target with its own dependencies and a command line that will generate it, and so forth. (For most programs, there's usually only two steps, compiling and linking, but an example of where a C++ source file could be generated is when a special-purpose code generator like **flex** or **bison** is used.)

In addition to code/program targets, you can also specify *dummy* targets that are never created. These targets can be used to do things explicitly, since an absent target causes the corresponding commands to be executed. For example, most makefiles specify a target **clean**, which will remove all of the generated files. (The purpose of doing this would be to allow the building of a program from a clean state the next time you call **make**.) For the above example, we might have:

```

# dummy target to create a clean slate
# *~ are emacs backup files
clean:
<tab> rm vidstore *.o *~

```

To use this target, we would type:

```
$ make clean
```

and it would delete the **vidstore** executable, all of the object files, and all of the emacs backup files (the files ending in ~).

These special targets are usually placed at the end of the makefile, though occasionally if the makefile is responsible for creating more than one program, there will be a dummy target at the beginning such as:

```

all: program1 program2 ...

program1: program1.o class1.o ...
<tab> clang++ -Wall -o program1 program1.o class1.o ...

program2: program2.o class1.o ...
<tab> clang++ -Wall -o program2 program2.o class1.o ...

...

```

Since the check for the dependencies of **all** will automatically check the dependencies for each program and recompile them automatically, there is no separate command associated with this target.

If you have multiple makefiles or would rather give one a more descriptive name (e.g., **Makefile.vidstore**), you can have **make** read a specified file using the **-f** option. For example, if the makefile for the video store application is named **Makefile.vidstore**, then typing

```
$ make -f Makefile.vidstore
```

will create **vidstore**. This can be combined with the explicitly named target as follows:

```
$ make -f Makefile.vidstore vidstore.o
```