Guide to makefiles

Written by Kevin Montag, CS107 TA

Makefiles are the Unix programmer's way of managing the build process for a project. Programming IDEs like Visual Studio store obscure configuration files based on selections in a series of disjointed and convoluted settings

menus. The Unix programmer, on the other hand, keeps things clean, storing a lightweight and easily-modified text file (with an admittedly disjointed and convoluted syntax) directly in the project directory. A well-written makefile describes all the files and settings used to compile a project and link it with the appropriate libraries, and makes compilation trivial from the command line.

How/why to use a makefile

For simple projects with uncomplicated settings, you can build without a makefile by directly invoking the compiler, e.g. `gcc file1.c file2.c file3.c` compiles three files and links them together into an executable named `a.out`. You could add flags such as `-Wall` (for warnings) or `-std=c99` (to use the updated C99 specification), or `-o [name]` to set the name of the resulting executable. However, manually re-typing these compilation commands quickly becomes tedious as projects get even slightly complex, and it is easy to mistype or be inconsistent. Managing the build with a makefile is much more convenient and less error-prone and far simpler after a one-time setup cost.

Using makefiles is simple. The command `make` invokes the

make program which reads in the file `Makefile` from the current directory and executes the build commands necessary to build the default target. You can also name just the specific target you want to build, such as `make reassemble` or `make myprogram`. The special target `make clean` will remove any previously built products and let you start fresh. The makefile itself contains the "recipe" information that specifies the steps required to build a given target, including what dependencies it has, what subproducts must be built in what sequence, what flags to pass to their various tools, and so on.

A makefile with a proper dependency graph will only rebuild a target when one or more of the components it depends on has changed. This allows separating the components of a project which can be built independently. For example, some program you're working on might make use of a vector and a linked-list that you've implemented, neither of which makes any reference to the other in its implementation. Good design sense places the code for these structures in separate files, say `vector.c/h` and `list.c/h` . If you make a change to `vector.c` , nothing about `list.c/h` changes with it - the content of these files stays the same, and in a deeper sense, the *code generated* by `list.c/h` will be no different after modifying the vector implementation. Thus, if we've

previously compiled the project, it's possible to set things up so that we can recompile the project (encompassing our changes to `vector.c`) without recompiling the code in `list.c/h`. In this toy example, the performance gain is minimal, but in large-scale real-world projects, these sorts of optimizations can save literally hours of compilation time whenever a project is built. Makefiles help us to automatically separate generated code for distinct components of a project, and automatically decide which portions of a project need to be built when we want to compile.

All CS107 projects will be distributed with a pre-written Makefile which you will typically use as-is, but occasionally you will have reason to poke around in it and make minor changes. You don't need to read the rest of this guide to be a happy and productive user of `make`, but if you are curious about how it works and someday aspire to construct your own Makefiles, read on.

How to write a makefile

The makefile syntax is known for having something of a learning curve, and it is common practice to copy and modify old makefiles, rather than generating new ones

from scratch. The best way to get a sense for what's going on is to dive right in with an example. Below is a simple makefile that might be used to build a project:

```
#
# A simple makefile for managing build of project co
#


# It is likely that default C compiler is already gc
# set, just to be sure
CC = gcc

# The CFLAGS variable sets compile flags for gcc:
#  -g         compile with debug information
#  -Wall      give verbose compiler warnings
#  -O0        do not optimize generated code
#  -std=c99   use the C99 standard language definitio
CFLAGS = -g -Wall -O0 -std=c99

# The LDFLAGS variable sets flags for linker
#  -lm   says to link in libm (the math library)
LDFLAGS = -lm

# In this section, you list the files that are part
# If you add/change names of source files, here is w
# edit the Makefile.
SOURCES = demo.c vector.c map.c
OBJECTS = $(SOURCES:.c=.o)
TARGET = demo


# The first target defined in the makefile is the on
# used when make is invoked with no argument. Given
# above, this Makefile file will build the one named
# assume that it depends on all the named OBJECTS fi

$(TARGET) : $(OBJECTS)
    $(CC) $(CFLAGS) -o $@ $^ $(LDFLAGS)

# Phony means not a "real" target, it doesn't build
# The phony target "clean" is used to remove all com
```

```
.PHONY: clean

clean:
    @rm -f $(TARGET) $(OBJECTS) core
```

Let's go through this makefile and see what's there. Note that lines beginning with '#' are comments, and are ignored when the makefile is processed.

Macros

These are the substitutions defined twoard the top of the makefile (lines that look like `CFLAGS = -g -Wall`). They are similar to `#define` statements in C, and should be used for any expression which is likely to be used repeatedly in a makefile. Once a macro has been assigned, we can reference it later using `$(MACRO_NAME)` (e.g. `$(CFLAGS)` in the example above). When we type `make` in a terminal, the file parser will simply replace these macro references with the assigned content.

In our sample makefile, there are also a few macros whose values may not be obvious. The line `OBJECTS = $(SOURCES:.c=.o)` defines the `OBJECTS` macro to be the same as the `SOURCES` macro, except that every instance of '.c' is replaced with '.o' - that is, this assignment

is equivalent to `OBJECTS = demo.o vector.o map.o`. There are also two built-in macros used by the makefile, `$@` and `$^`; these evaluate to `demo` and `demo.o vector.o map.o`, respectively, but we will need to learn a bit about targets before we find out why.

For clarity, it may be worth looking at the content of the makefile as the parser "sees" it, with comments removed and macros fully expanded. In this form, our sample makefile looks like:

```
demo : demo.o vector.o map.o
    gcc -g -Wall -o demo demo.o vector.o map.o -lm

.PHONY: clean

clean:
    @rm -f demo demo.o vector.o map.o core
```

Targets

Following our makefile's macro definitions, we see a number of targets. Targets and their associated actions are written in the form:

```
target-name : dependencies
    action
```

The target name is generally the name of the file that will be produced when this target is built. The first target listed in a makefile is the default target, meaning that it is the target which is built when `make` is invoked with no arguments; other targets can be built using `make [target-name]` at the command line. It is also worth mentioning at this point that the Make utility recognizes a number of implicit targets, and in particular that each of our object files has an associated implicit target equivalent to:

```
[filename].o : [filename].c
    $(CC) $(CFLAGS) -o [filename].o [filename].c
```

Much of the power of the Make utility comes from its handling of dependencies. The dependencies of a target are the files which need to exist and be up to date before the target itself can be built. In the example above, the `demo` target depends on three object files (each of which can be built with its own implicit target as specified). Make processes dependencies recursively; if particular dependency has an associated target, the Make utility will

(re)build the dependency's target before processing the parent target, ensuring that all dependencies are up to date before the parent target is processed. Thus, for our sample makefile, the command `make demo` actually behaves more like `make demo.o ; make vector.o ; make map.o ; make demo` (the recursion ends at dependencies which don't have an associated target; this occurs if, for example, we're depending on a source file like `demo.c`, as is the case with the `demo.o` target). The Make utility will then examine the timestamps of each file on which the parent target depends, and will build the parent target if any of these files have been changed more recently than the parent file (or if the parent file does not yet exist). In our case, this means that if the `demo` executable already exists in our directory, `make demo` will not do anything unless the directory's object files need to be rebuilt during recursive dependency processing, which in turn will only occur if any of our source files (`demo.c`, `vector.c`, `map.c`) have been modified more recently than their associated object file was built. Thus if we haven't modified any of our source files, invoking `make demo` repeatedly will only build the `demo` executable once. Furthermore, if we modify just one of our source files, we will only rebuild the associated object file, rather than all three object files.

Finally, each target has an associated command, which will be run in the shell in order to build the target. Generally, this is a command which invokes the compiler, but technically it can be any command which creates a file with the target's name. When defining the command for a target, we also have access to a number of special macros, such as `$@` and `$^` above. We can see now that these macros evaluate, respectively, to the name of the current target and its list of dependencies. Other such target-dependent macros exist, and information on them is available in the Make documentation.

Phony targets Note that the `clean` target in our sample Makefile doesn't actually create a file named 'clean', and thus doesn't fit the pattern which we've been describing for targets. Rather, the `clean` target is used as a shortcut for running a command which clears out the project's build files (the '@' at the beginning of the command tells Make not to print it to the terminal when it is being run). We flag targets like this by listing them as "dependencies" of `.PHONY`, which is a pseudo-target that we'll never actually build. When the Make utility encounters a phony target, it will run the associated command automatically, without performing any dependency checks.

That's it!

There's plenty more that you can do with a makefile, and a few more advanced syntactic elements. An inexhaustible source of make wisdom is the full manual for GNU Make which will tell you more that you could ever want to know. Checking out makefiles from some real world projects is another interesting way to see make in action.

Frequently asked questions about make

Make is failing with a cryptic error about `Makefile: *** missing separator.` Huh?

In what is widely considered one of the dumber decisions in the history of computing, make distinguishes between tabs and spaces in a Makefile. The command lines for a target must begin with a tab and an equivalent number of spaces just won't do. Edit your makefile and replace those errant spaces with a tab to restore Makefile joy.

The compiler warnings from my build seem a bit garbled, is there something I can do to get readable messages? See sample warning below complete with funny characters:

```
reassemble.c: In function â:
reassemble.c:48:5: error: expected â before â token
```

This mismatch is due to gcc getting all fancy and printing identifiers enclosed in curly "smart quotes", but those unusual characters are not meshing well with the language configuration of your terminal. If you set the environment variable `LC_ALL` to `C`, gcc will dumb down its output. Set in your shell configuration file to make the change permanent.