Compiling Programs with Make

Written by Chris Gregg and Kevin Montag, with modifications by Nick Troccoli

Click here (https://youtu.be/CrwBHSeA05Y) for a walkthrough video.

In CS107, we will use a program called `make` to compile our programs. Make is a program that **dates back to 1976 (https://en.wikipedia.org/wiki/Make_(software)#Origin)**, and it is used to build projects with dependencies such that it only recompiles files that have been changed, and avoids having to type lengthy compile commands. It is a single file that contains all the files and settings to compile a project and link it with the appropriate libraries.

For simple projects with uncomplicated settings, you can build without a makefile by directly invoking a compiler like **GCC (gcc)**, 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=gnu99` (to use the updated GNU99 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.

For our purposes, you will not need to know too much about Make, except how to use it. All CS107 projects will be distributed with a pre-written Makefile which you can usually use as-is. What is very important, however, is that you need to **remember to run `make` after any change to the source code of your programs** -- many students forget to run `make` and wonder why they get unexpected results from their programs, when it is simply that they never re-compiled their code after the changes!

The simplest way to use `make` is by typing `make` in a directory that contains a "Makefile" called, fittingly, `Makefile`:

```
$ make
gcc -g -Og -std=gnu99  -o hello helloWorld.c helloLanguages.c
$ ./hello
Hello World
Hallo Welt
Bonjour monde
```

Here is an example Makefile for the program above with the following files: `hello.h`, `helloLanguages.c`, `helloWorld.c`:

```
#
# A very simple makefile
#

# The default C compiler
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=gnu99  use the GNU99 standard language definition
CFLAGS = -g -Wall -O0 -std=gnu99

hello: helloWorld.c helloLanguages.c hello.h
    $(CC) $(CFLAGS) -o hello helloWorld.c helloLanguages.c

.PHONY: clean

clean:
    rm -f hello *.o
```

Note that lines beginning with '#' are comments, and are ignored when the makefile is processed.

The Makefile has rules that are followed to decide when to compile a program. In particular, the `hello:` line in the Makefile tells Make to re-compile the program if any of the three files (`hello.c`, `helloLanguages.c`, and `hello.h`) have changed. On the following line, which *must begin with a tab and not spaces* (if you don't do this, you'll get a "missing separator" error!), the compilation line runs. There are two variables in this Makefile, `CC` (the compiler), and `CFLAGS` (the flags that we are going to send to the compiler).

A commonly-included second rule, `clean:`, allows you to write `make clean` and remove all files associated with compiling that program. (The first rule listed in the Makefile is executed by default when you type `make`, which is why you don't have to write `make hello` each time).

See **how to compile with gcc (gcc)** for information about how the compilation happens.

Extra: More About Makefiles

While you don't need to know much about Makefiles in order to compile your assignments, read on if you're interested in learning more about the details behind writing a Makefile. Checking out makefiles from some real world projects is another interesting way to see make in action.

Below is a longer Makefile that might be used to build a larger project:

```
#
# A simple makefile for managing build of project composed of C source files.
#


# It is likely that default C compiler is already gcc, but explicitly
# 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=gnu99  use the GNU99 standard language definition
CFLAGS = -g -Wall -O0 -std=gnu99

# 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 of the project.
# If you add/change names of source files, here is where you
# 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 one
# used when make is invoked with no argument. Given the definitions
# above, this Makefile file will build the one named TARGET and
# assume that it depends on all the named OBJECTS files.

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

# Phony means not a "real" target, it doesn't build anything
# The phony target "clean" is used to remove all compiled object files.
# 'core' is the name of the file outputted in some cases when you get a
# crash (SEGFAULT) with a "core dump"; it can contain more information about
# the crash.
.PHONY: clean

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

Let's go through this makefile and see what's there.

Macros

These are the substitutions defined toward 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.

Diving deeper, 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 -O0 -std=gnu99 -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. In large-scale projects, these sorts of optimizations can save hours of compilation time whenever a project is built.

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.

Even More

If you're interested in learning even more about `make`, check out the following resources:

- the full **make manual (http://www.gnu.org/software/make/)** (from GNU)
- Section 2 of this **Stanford Unix Programming Tools (http://cslibrary.stanford.edu/107/UnixProgrammingTools.pdf)** document
- the No-starch press **"Gnu Make Book" (http://proquest.safaribooksonline.com.stanford.idm.oclc.org/book/operating-systems-and-server-administration/9781457189883)** (requires authentication, accesses Stanford's subscription to Safari Books Online).
