Makefile Tutorial

Eric S. Missimer

December 6, 2013

1 Basic Elements of a Makefile

1.1 Explicit Rules

A the major part of a Makefile are the explicit rules (a.k.a. recipes) that *make* certain files. Below is the general layout:

```
target: list-of-dependencies
        commands-to-generate-target
```

target will be something such as the program you wish to make or perhaps an object file necessary to make the final program. *list-of-dependencies* is a list of files necessary to build the target. *commands-to-generate-target* are one or more commands that should be executed to generate *target*. There is a tab before each command. The tab is important and it must be a tab, not consecutive spaces.

Below is a an example Makefile to generate a program `hello` from a single c file `hello.c`:

```
hello: hello.c
        gcc hello.c -o hello
```

We will continue to use this example of building `hello` to show the various features of Makefiles.

One of the nice features about Makefiles is the ability to specify dependencies. In the previous example `hello` has a dependency on `hello.c`. This means that if `hello` already exists but has a timestamp that occurs before `hello.c` `make` will recognize that `hello` is out of date and run the commands to remake it. In the more general sense if a target is older than any of its dependencies or any of it's dependencies do not exist, the commands associated with the target will be executed, which in theory will update the target so it has a timestamp after its dependencies. In case you were wondering at this point, if a target has no dependencies the commands to generate the target will only be executed if the target does not exist.

Right now we have only one target in our example Makefile so when you type `make` into a shell `hello` is created. Typing `make hello` has the same effect. When a target is not specified to `make` then the first target in the file is used. This target is typically `all` but that is getting ahead of ourselves.

1.2 Dependencies as Targets

The dependencies of some targets can in themselves be targets. Back to the `hello` example, lets say we wanted to generate the object file `hello.o` before generating the final program.[1] Our Makefile now looks like:

```
hello: hello.o
        gcc hello.o -o hello

hello.o: hello.c
        gcc hello.c -c -o hello.o
```

Since `hello` is our first target executing `make` will create `hello` but only if `hello.o` is out of date. `hello.o` is only out of date if it doesn't exist or if `hello.c` has a newer timestamp than `hello.o`. As you can see there is a recursive nature to the Makefile dependencies. If `hello.c` is updated even though `hello` does not directly depend on `hello.c` it will still be recreated, after `hello.o` is recreated, because `hello.o` will have a newer timestamp after it is recreated.

Now if we had a header file `hello.h` that was included by `hello.c`, if we updated `hello.h` we would want to rebuild `hello.o`. We can specify that `hello.o` depends on `hello.h`:

```
hello: hello.o
        gcc hello.o -o hello

hello.o: hello.c hello.h
        gcc hello.c -c -o hello.o
```

Specifying header file dependencies for small projects by hand is not too bad but if a project is larger than a few files the dependencies list of object files can become time consuming to do by hand. There is an automated approach that will be discussed later which is useful for larger projects or just the lazy (which includes myself).

[1] In this example generating a single object file before generating the executable seems pointless but if you had a large number of object files you might wish to generate each object file before linking to avoid recompiling all the source files when a single source file is changed

1.3 PHONY targets

Phony targets are targets that do not generate a file. A phony target will always have the commands associated with it executed and will always have it's dependencies brought up to date if they are out of date.

Typical phony targets are:

- `all` - this is typically the first target so it is also executed by executing `make` with no target. This target, as the name implies, is usually used to generate everything, or at least everything in a typical build.
- `clean` - this target typically removes what is created by `all` so that only the source files remain.
- `install` - as the name implies this target will install the program. While you might not have an `install` target it is good to know about it because when you download and build software the last step will typically by to install it via `make install`.
- `uninstall` - will remove the program.

A target is declared as phony with the following syntax:

```
.PHONY: all clean install uninstall
```

Our updated Makefile for `hello` is below:

```
.PHONY: all clean

all: hello

hello: hello.o
	gcc hello.o -o hello

hello.o: hello.c hello.h
	gcc hello.c -c -o hello.o

clean:
	-rm hello.o hello
```

You will notice I also introduced another Makefile trick. For the `clean` target a '-' was placed before the `rm` command. What this says is if rm fails, which would happen if the files to remove did not exist, the error reported by rm should be ignored. Typically if an error is encountered by a command invoked by `make`, `make` will stop executing commands. Since we only care about removing the files if they exist we say to ignore the error from `rm`. Note this

does not matter so much for our current `clean` target since there are no commands after the `rm` but if there we more commands after it they would not be executed if `rm` reported an error. Also sometimes you will not see the '-' before `rm` but you will see the following:

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

The `-f` is the *force* flag for `rm` which will make `rm` ignore nonexistent files and never prompt before removing a file, which is typically the desired behavior. I will use the `-f` flag from now on.

One last thing to remember about phony targets: Phony targets in general should not be dependencies of real targets. If a phony target is the dependency of a real target then that real target will be rebuilt every time the real target is told to be updated.

1.4 Variables

Makefiles can have variables which can be used to make your life easier. Lets go back to our `hello` example:

```
PROGRAM=hello

.PHONY: all clean

all: $(PROGRAM)

$(PROGRAM): $(PROGRAM).o
        gcc $(PROGRAM).o -o $(PROGRAM)

$(PROGRAM).o: $(PROGRAM).c $(PROGRAM).h
        gcc $(PROGRAM).c -c -o $(PROGRAM).o

clean:
        rm -rf $(PROGRAM).o $(PROGRAM)
```

Now if we rename `hello.c` and `hello.h` we only need to change the value of the variable `PROGRAM`. Note when we declare (or append to) a variable it is just the variable name but when it is used we surround the variable name with a $(and). Variables can be used as targets, dependencies, substrings with something else appended to it (as in `$(PROGRAM).o`) and in the commands to build targets.

Variables can also be lists of values, but to leverage a lot of power with list variables you will need to understand Makefile macros and/or wildcard matching.

1.5 End of the Basics

That is the basics of Makfiles. With this knowledge you know how to read and write basic Makefiles. You will probably also be able to understand more advance concepts if you find them in a Makefile by filing in the gaps with information you can find on the internet.

A few final notes. A lot of students will write their code, finish their assignment and then create a Makefile. This is the backwards way to do things. The first thing you should create is a Makefile. When you are trying to figure out which compile flags you need, test them in the Makefile not in the console, that way when you figure out the right flags you don't need to copy them or anything. Also the Makefile is there to make your life easier when you are developing the program, that is one of the nice features of the dependencies, you can update a single file and execute `make` and `make` will figure out which intermediate files need to be updated to build the program. As a general rule: if you are working on your program and are typing anything more than `make <target>` you are doing something wrong.

2 More Advance Makefile Elements

Coming soon...