A Tutorial on Portable Makefiles

August 20, 2017
nullprogram.com/blog/2017/08/20/

In my first decade writing Makefiles, I developed the bad habit of liberally using GNU Make's extensions. I didn't know the line between GNU Make and the portable features guaranteed by POSIX. Usually it didn't matter much, but it would become an annoyance when building on non-Linux systems, such as on the various BSDs. I'd have to specifically install GNU Make, then remember to invoke it (i.e. as `gmake`) instead of the system's make.

I've since become familiar and comfortable with make's official specification[1], and I've spend the last year writing strictly portable Makefiles. Not only has are my builds now portable across all unix-like systems, my Makefiles are cleaner and more robust. Many of the common make extensions — conditionals in particular — lead to fragile, complicated Makefiles and are best avoided anyway. It's important to be able to trust your build system to do its job correctly.

This tutorial should be suitable for make beginners who have never written their own Makefiles before, as well as experienced developers who want to learn how to write portable Makefiles. Regardless, in order to understand the examples you must be familiar with the usual steps for building programs on the command line (compiler, linker, object files, etc.). I'm not going to suggest any fancy tricks nor provide any sort of standard starting template. Makefiles should be dead simple when the project is small, and grow in a predictable, clean fashion alongside the project.

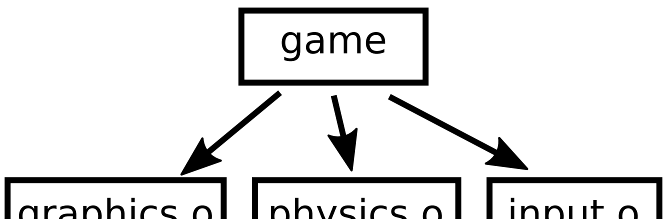
I'm not going to cover every feature. You'll need to read the specification for yourself to learn it all. This tutorial will go over the important features as well as the common conventions. It's important to follow established conventions so that people using your Makefiles will know what to expect and how to accomplish the basic tasks.

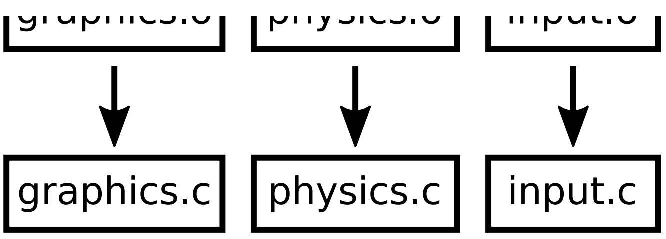
If you're running Debian, or a Debian derivative such as Ubuntu, the `bmake` and `freebsd-buildutils` packages will provide the `bmake` and `fmake` programs respectively. These alternative make implementations are very useful for testing your Makefiles' portability, should you accidentally make use of a GNU Make feature. It's not perfect since each implements some of the same extensions as GNU Make, but it will catch some common mistakes.

What's in a Makefile?

I am free, no matter what rules surround me. If I find them tolerable, I tolerate them; if I find them too obnoxious, I break them. I am free because I know that I alone am morally responsible for everything I do. —Robert A. Heinlein

At make's core are one or more dependency trees, constructed from *rules*. Each vertex in the tree is called a *target*. The final products of the build (executable, document, etc.) are the tree roots. A Makefile specifies the dependency trees and supplies the shell commands to produce a target from its *prerequisites*.





In this illustration, the ".c" files are source files that are written by hand, not generated by commands, so they have no prerequisites. The syntax for specifying one or more edges in this dependency tree is simple:

```
target [target...]: [prerequisite...]
```

While technically multiple targets can be specified in a single rule, this is unusual. Typically each target is specified in its own rule. To specify the tree in the illustration above:

```
game: graphics.o physics.o input.o
graphics.o: graphics.c
physics.o: physics.c
input.o: input.c
```

The order of these rules doesn't matter. The entire Makefile is parsed before any actions are taken, so the tree's vertices and edges can be specified in any order. There's one exception: the first non-special target in a Makefile is the *default target*. This target is selected implicitly when make is invoked without choosing a target. It should be something sensible, so that a user can blindly run make and get a useful result.

A target can be specified more than once. Any new prerequisites are appended to the previously-given prerequisites. For example, this Makefile is identical to the previous, though it's typically not written this way:

```
game: graphics.o
game: physics.o
game: input.o
graphics.o: graphics.c
physics.o: physics.c
input.o: input.c
```

There are six *special targets* that are used to change the behavior of make itself. All have uppercase names and start with a period. Names fitting this pattern are reserved for use by make. According to the standard, in order to get reliable POSIX behavior, the first non-comment line of the Makefile *must* be `.POSIX`. Since this is a special target, it's not a candidate for the default target, so `game` will remain the default target:

```
.POSIX:
game: graphics.o physics.o input.o
graphics.o: graphics.c
physics.o: physics.c
input.o: input.c
```

In practice, even a simple program will have header files, and sources that include a header file should also have an edge on the dependency tree for it. If the header file changes, targets that include it should also be rebuilt.

```
.POSIX:
game: graphics.o physics.o input.o
graphics.o: graphics.c graphics.h
physics.o: physics.c physics.h
input.o: input.c input.h graphics.h physics.h
```

Adding commands to rules

We've constructed a dependency tree, but we still haven't told make how to actually build any targets from its prerequisites. The rules also need to specify the shell commands that produce a target from its prerequisites.

If you were to create the source files in the example and invoke make, you will find that it actually *does* know how to build the object files. This is because make is initially configured with certain *inference rules*, a topic which will be covered later. For now, we'll add the `.SUFFIXES` special target to the top, erasing all the built-in inference rules.

Commands immediately follow the target/prerequisite line in a rule. Each command line must start with a tab character. This can be awkward if your text editor isn't configured for it, and it will be awkward if you try to copy the examples from this page.

Each line is run in its own shell, so be mindful of using commands like `cd`, which won't affect later lines.

The simplest thing to do is literally specify the same commands you'd type at the shell:

```
.POSIX:
.SUFFIXES:
game: graphics.o physics.o input.o
    cc -o game graphics.o physics.o input.o
graphics.o: graphics.c graphics.h
    cc -c graphics.c
physics.o: physics.c physics.h
    cc -c physics.c
input.o: input.c input.h graphics.h physics.h
    cc -c input.c
```

Invoking make and choosing targets

I tried to walk into Target, but I missed. —Mitch Hedberg

When invoking make, it accepts zero or more targets from the dependency tree, and it will build these targets — e.g. run the commands in the target's rule — if the target is *out-of-date*. A target is out-of-date if it is older than any of its prerequisites.

```
# build the "game" binary (default target)
$ make

# build just the object files
$ make graphics.o physics.o input.o
```

This effect cascades up the dependency tree and causes further targets to be rebuilt until all of the requested targets are up-to-date. There's a lot of room for parallelism since different branches of the tree can be updated independently. It's common for make implementations to support parallel builds with the `-j` option. This is non-standard, but it's a fantastic feature that doesn't require anything special in the Makefile to work correctly.

Similar to parallel builds is make's `-k` ("keep going") option, which *is* standard. This tells make not to stop on the first error, and to continue updating targets that are unaffected by the error. This is nice for fully populating Vim's quickfix list[2] or Emacs' compilation buffer[3].

It's common to have multiple targets that should be built by default. If the first rule selects the default target, how do we solve the problem of needing multiple default targets? The convention is to use *phony targets*. These are called "phony" because there is no corresponding file, and so phony targets are never up-to-date. It's convention for a phony "all" target to be the default target.

I'll make `game` a prerequisite of a new "all" target. More real targets could be added as necessary to turn them into defaults. Users of this Makefile will also expect `make all` to build the entire project.

Another common phony target is "clean" which removes all of the built files. Users will expect `make clean` to delete all generated files.

```
.POSIX:
.SUFFIXES:
all: game
game: graphics.o physics.o input.o
    cc -o game graphics.o physics.o input.o
graphics.o: graphics.c graphics.h
    cc -c graphics.c
physics.o: physics.c physics.h
    cc -c physics.c
input.o: input.c input.h graphics.h physics.h
    cc -c input.c
clean:
    rm -f game graphics.o physics.o input.o
```

Customize the build with macros

So far the Makefile hardcodes `cc` as the compiler, and doesn't use any compiler flags (warnings, optimization, hardening, etc.). The user should be able to easily control all these things, but right now they'd have to edit the entire Makefile to do so. Perhaps the user has both `gcc` and `clang` installed, and wants to choose one or the other without changing which is installed as `cc`.

To solve this, make has *macros* that expand into strings when referenced. The convention is to use the macro named `CC` when talking about the C compiler, `CFLAGS` when talking about flags passed to the C compiler, `LDFLAGS` for flags passed to the C compiler when linking, and `LDLIBS` for flags about libraries when linking. The Makefile should supply defaults as needed.

A macro is expanded with `$(...)`. It's valid (and normal) to reference a macro that hasn't been defined, which will be an empty string. This will be the case with `LDFLAGS` below.

Macro values can contain other macros, which will be expanded recursively each time the macro is expanded. Some make implementations allow the name of the macro being expanded to itself be a macro, which is turing complete[4], but this behavior is non-standard.

```
.POSIX:
.SUFFIXES:
CC     = cc
CFLAGS = -W -O
LDLIBS = -lm

all: game
game: graphics.o physics.o input.o
    $(CC) $(LDFLAGS) -o game graphics.o physics.o input.o $(LDLIBS)
graphics.o: graphics.c graphics.h
    $(CC) -c $(CFLAGS) graphics.c
physics.o: physics.c physics.h
    $(CC) -c $(CFLAGS) physics.c
input.o: input.c input.h graphics.h physics.h
    $(CC) -c $(CFLAGS) input.c
clean:
    rm -f game graphics.o physics.o input.o
```

Macros are overridden by macro definitions given as command line arguments in the form `name=value`. This allows the user to select their own build configuration. This is one of make's most powerful and under-appreciated features.

```
$ make CC=clang CFLAGS='-O3 -march=native'
```

If the user doesn't want to specify these macros on every invocation, they can (cautiously) use make's `-e` flag to set overriding macros definitions from the environment.

```
$ export CC=clang
$ export CFLAGS=-O3
$ make -e all
```

Some make implementations have other special kinds of macro assignment operators beyond simple assignment (=). These are unnecessary, so don't worry about them.

Inference rules so that you can stop repeating yourself

The road itself tells us far more than signs do. —Tom Vanderbilt, Traffic: Why We Drive the Way We Do

There's repetition across the three different object files. Wouldn't it be nice if there was a way to communicate this pattern? Fortunately there is, in the form of *inference rules*. It says that a target with a certain extension, with a prerequisite with another certain extension, is built a certain way. This will make more sense with an example.

In an inference rule, the target indicates the extensions. The `$<` macro expands to the prerequisite, which is essential to making inference rules work generically. Unfortunately this macro is not available in target rules, as much as that would be useful.

For example, here's an inference rule that teaches make how to build an object file from a C source file. This particular rule is one that is pre-defined by make, so you'll never need to write this one yourself. I'll include it for completeness.

```
.c.o:
    $(CC) $(CFLAGS) -c $<
```

These extensions must be added to .SUFFIXES before they will work. With that, the commands for the rules about object files can be omitted.

```
.POSIX:
.SUFFIXES:
CC     = cc
CFLAGS = -W -O
LDLIBS = -lm

all: game
game: graphics.o physics.o input.o
    $(CC) $(LDFLAGS) -o game graphics.o physics.o input.o $(LDLIBS)
graphics.o: graphics.c graphics.h
physics.o: physics.c physics.h
input.o: input.c input.h graphics.h physics.h
clean:
    rm -f game graphics.o physics.o input.o

.SUFFIXES: .c .o
.c.o:
    $(CC) $(CFLAGS) -c $<
```

The first empty .SUFFIXES clears the suffix list. The second one adds .c and .o to the now-empty suffix list.

Other target conventions

Conventions are, indeed, all that shield us from the shivering void, though often they do so but poorly and desperately. —Robert Aickman

Users usually expect an "install" target that installs the built program, libraries, man pages, etc. By convention this target should use the PREFIX and DESTDIR macros.

The PREFIX macro should default to /usr/local, and since it's a macro the user can override it to install elsewhere, such as in their home directory[5]. The user should override it for both building and installing, since the prefix may need to be built into the binary (e.g. -DPREFIX=$(PREFIX)).

The DESTDIR is macro is used for *staged builds*, so that it gets installed under a fake root directory for the sake of packaging. Unlike PREFIX, it will not actually be run from this directory.

```
.POSIX:
CC     = cc
CFLAGS = -W -O
LDLIBS = -lm
PREFIX = /usr/local

all: game
install: game
	mkdir -p $(DESTDIR)$(PREFIX)/bin
	mkdir -p $(DESTDIR)$(PREFIX)/share/man/man1
	cp -f game $(DESTDIR)$(PREFIX)/bin
	gzip < game.1 > $(DESTDIR)$(PREFIX)/share/man/man1/game.1.gz
game: graphics.o physics.o input.o
	$(CC) $(LDFLAGS) -o game graphics.o physics.o input.o $(LDLIBS)
graphics.o: graphics.c graphics.h
physics.o: physics.c physics.h
input.o: input.c input.h graphics.h physics.h
clean:
	rm -f game graphics.o physics.o input.o
```

You may also want to provide an "uninstall" phony target that does the opposite.

```
make PREFIX=$HOME/.local install
```

Other common targets are "mostlyclean" (like "clean" but don't delete some slow-to-build targets), "distclean" (delete even more than "clean"), "test" or "check" (run the test suite), and "dist" (create a package).

Complexity and growing pains

One of make's big weak points is scaling up as a project grows in size.

Recursive Makefiles

As your growing project is broken into subdirectories, you may be tempted to put a Makefile in each subdirectory and invoke them recursively.

Don't use recursive Makefiles[6]. It breaks the dependency tree across separate instances of make and typically results in a fragile build. There's nothing good about it. Have one Makefile at the root of your project and invoke make there. You may have to teach your text editor how to do this.

When talking about files in subdirectories, just include the subdirectory in the name. Everything will work the same as far as make is concerned, including inference rules.

```
src/graphics.o: src/graphics.c
src/physics.o: src/physics.c
src/input.o: src/input.c
```

Out-of-source builds

Keeping your object files separate from your source files is a nice idea. When it comes to make, there's good news and bad news.

The good news is that make can do this. You can pick whatever file names you like for targets and prerequisites.

```
obj/input.o: src/input.c
```

The bad news is that inference rules are not compatible with out-of-source builds. You'll need to repeat the same commands for each rule as if inference rules didn't exist. This is tedious for large projects, so you may want to have some sort of "configure" script, even if hand-written, to generate all this for you. This is essentially what CMake is all about. That, plus dependency management.

Dependency management

Another problem with scaling up is tracking the project's ever-changing dependencies across all the source files. Missing a dependency means the build may not be correct unless you `make clean` first.

If you go the route of using a script to generate the tedious parts of the Makefile, both GCC and Clang have a nice feature for generating all the Makefile dependencies for you (`-MM`, `-MT`), at least for C and C++. There are lots of tutorials for doing this dependency generation on the fly as part of the build, but it's fragile and slow. Much better to do it all up front and "bake" the dependencies into the Makefile so that make can do its job properly. If the dependencies change, rebuild your Makefile.

For example, here's what it looks like invoking gcc's dependency generator against the imaginary `input.c` for an out-of-source build:

```
$ gcc $CFLAGS -MM -MT '$(BUILD)/input.o' input.c
$(BUILD)/input.o: input.c input.h graphics.h physics.h
```

Notice the output is in Makefile's rule format.

Unfortunately this feature strips the leading paths from the target, so, in practice, using it is always more complicated than it should be (e.g. it requires the use of `-MT`).

Microsoft's Nmake

Microsoft has an implementation of make called Nmake, which comes with Visual Studio[7]. It's *nearly* a POSIX-compatible make, but necessarily breaks from the standard in some places. Their cl.exe compiler uses `.obj` as the object file extension and `.exe` for binaries, both of which differ from the unix world, so it has different built-in inference rules. Windows also lacks a Bourne shell and the standard unix tools, so all of the commands will necessarily be different.

There's no equivalent of `rm -f` on Windows, so good luck writing a proper "clean" target. No, `del /f` isn't the same.

So while it's close to POSIX make, it's not practical to write a Makefile that will simultaneously work properly with both POSIX make and Nmake. These need to be separate Makefiles.

May your Makefiles be portable

It's nice to have reliable, portable Makefiles that just work anywhere. Code to the standards[8] and you don't need feature tests or other sorts of special treatment.

tags [tutorial c posix]

1. http://pubs.opengroup.org/onlinepubs/9699919799/utilities/make.html
2. http://vimdoc.sourceforge.net/htmldoc/quickfix.html
3. https://www.gnu.org/software/emacs/manual/html_node/emacs/Compilation.html
4. https://nullprogram.com/blog/2016/04/30/
5. https://nullprogram.com/blog/2017/06/19/
6. http://aegis.sourceforge.net/auug97.pdf
7. https://nullprogram.com/blog/2016/06/13/
8. https://nullprogram.com/blog/2017/03/30/