Practical Makefiles, by example

Author: John Tsiombikas
Contact: nuclear@mutantstargoat.com
Date: 9 Feb 2015
Last update: 2 June 2022

Contents

1 Rationale

The purpose of this document is to explain how to write practical makefiles for your everyday hacks and projects. I want to illustrate, how easy it is to use make for building your programs, and doing so, dispel the notion that resorting to big clunky graphical IDEs, or makefile generators such as autotools or cmake, is the way to focus on your code faster.

The purpose of this document is certainly *not*, to teach you how to write user-friendly, all-encompassing, release-quality build systems. You can certainly arrive to

that by extending a simple makefile, and I'll give you a few pointers about that sort of thing in the end, but I will not focus on complexity and completeness, but rather on simplicity and practicality.

Finally, it's worth spelling out up front that when I say "make", I'm really referring to the GNU implementation of make, which is packed with useful features, not necessarily found in other make programs out there. It might be interesting to write a follow-up article, providing similar examples for other make implementations out there, such as BSD make, Watcom make, Borland make, or Microsoft's nmake. But I'll leave that for another time.

2 Building C/C++ programs

First let's consider an example C program, and how to build it without make.

In order to escape the trap of an oversimplified toy program, let's assume we're making a small computer game. Our game is made up of the following source and header files:

`main.c`	entry point and system event loop
`game.h` & `game.c`	event handlers and game update/redraw loop
`player.h` & `player.c`	player state and input
`level.h` & `level.c`	game world state
`enemy.h` & `enemy.c`	enemy state and AI
`renderer.h` & `renderer.c`	graphics algorithms / game drawing
`vecmath.h` & `vecmath.c`	vector & matrix math needed for simulation and rendering
`image.h` & `image.c`	image loading for textures
`mesh.h` & `mesh.c`	3D mesh data structures & loading

Also, our game depends on a number of 3rd party libraries:

OpenGL	Low level graphics library
GLUT	Cross-platform OpenGL window setup & event handling
libpng	PNG image file format reader/decoder
zlib	Needed by libpng for lz77 decompression

To build this program we would first have to run the compiler on each source file, generating a corresponding object code file:

```
cc -c main.c
cc -c game.c
cc -c level.c
...
```

Then, we would have to feed all the object files to the linker (possibly still using the C compiler front-end for convenience), instructing it to link all the libraries we need as well:

```
cc -o mygame *.o -lGL -lglut -lpng -lz -lm
```

Obviously, such tediousness could be automated with a shell script. However, that would be a grossly sub-optimal solution, as it would recompile all source files from scratch, every time we try to build with it.

3 Enter make

Make is a much more elegant solution to our building problems. We create a file named `Makefile`, which contains a set of rules describing build products, their dependencies, and which commands are needed to build them. Then when we ask make to build our program binary, it recursively traverses the dependency graph to figure out which parts need to be re-created, based on the last-modified timestamps of the target file, and its dependency files.

Warning: do not run away terrified by the first makefile example. It's only meant to illustrate how make works, so it's needlessly verbose. If you're easily scared, skip to section: "A makefile for 99% of your programs".

Back to our game example, the binary `mygame`, obviously depends on `main.o` (among other object files). `main.o` in turn, is created by compiling `main.c`. So, let's define two make rules for creating these two files:

```
mygame: main.o player.o enemy.o renderer.o vecmath.o image.o mesh.o
    cc -o mygame main.o player.o enemy.o renderer.o vecmath.o image.o mesh.

main.o: main.c
    cc -c main.c

player.o: player.c
    cc -c player.c

# ... and so on, and so forth ...
```

Essentially for each rule, the part before the colon is the filename we want to build when this rule is executed, the part after the colon is the list of dependencies, i.e. what is needed to create this rule's target. And finally, in subsequent lines, we just type the commands needed to make this happen.

Note that each line in the commands part of a rule needs to begin with a leading tab. Spaces will not work, it really has to be a tab character, so pay extra attention to your text editor settings, about how it treats any tabs you enter.

So, if we modify `main.c`, and then type `make` (which means: build the first rule in the makefile), make will try to build the file named `mygame`. In order to build it, make will have to use the declared dependency files, so it will execute each rule corresponding to those files in turn, starting from `main.o`.

To build `main.o`, make sees that it needs `main.c`, which having been modified more recently than `main.o`, is deemed out of date and must be re-created by executing the command specified in this rule.

Similarly, returning to the initial rule, since now `main.o` is newer than the old `mygame` binary, the binary will be rebuilt by running the linking command, after visiting each object file rule in turn and determining that none of them needs rebuilding, since we didn't touch any of their dependencies.

4 Simpler makefile

Obviously, having to type rules for each of our source files is tedious, and thankfully unnecessary. Make actually knows how to create object code from C source files, so we can skip the object file rules, and also provides some handy variables for referring to the target or dependency files in rule commands without having to re-type everything. So here is a slightly simpler makefile for the same task:

```
myprog: main.o game.o level.o player.o enemy.o renderer.o vecmath.o image.o
    cc -o $@ $^ -lGL -lglut -lpng -lz -lm
```

`@` is a built-in make variable containing the target of each rule (myprog in this case), and `^` is another built-in variable containing all the dependencies of each rule (so the list of object files in this case). Similarly, although not useful for this particular rule, there is also the `<` variable, which contains only the first element of the dependencies (so `main.o` if we used it in the rule above).

We substitute the value of any variable (built-in or not) by prepending a dollar sign to its name. However, one peculiarity of the make syntax, as opposed to say the bourne shell syntax, is that only the first character following the dollar sign is considered to be the variable name. If we want to use longer names, we have to parenthesize the name before applying the dollar sign to extract its value. So for instance if we had defined a variable: `obj = main.o game.o level.o ... (etc) ...`, then to substitute its contents we have to use parentheses: `$(obj)`. If we instead tried to use `$obj` then make would try to expand a variable named "o", and then append the string "bj" to the result.

5 A makefile for 99% of your programs

There's one last thing bugging me in the last example. I hate having to type manually the list of object files needed to build the program. Thankfully it turns out we don't need to do that either. See the following example:

```
src = $(wildcard *.c)
obj = $(src:.c=.o)

LDFLAGS = -lGL -lglut -lpng -lz -lm

myprog: $(obj)
    $(CC) -o $@ $^ $(LDFLAGS)

.PHONY: clean
clean:
    rm -f $(obj) myprog
```

The first line of this example collects all source files in the current directory in the `src` variable. Then, the second line transforms the contents of the `src` variable, changing all file suffixes from `.c` to `.o`, thus constructing the object file list we need.

I also used a new variable called `LDFLAGS` for the list of libraries required during linking (`LDFLAGS` is conventionally used for this usage, while similarly `CFLAGS` and `CXXFLAGS` can be used to pass flags to the C and C++ compilers respectively).

Finally, I added a new rule for cleaning up every target, in order to rebuild the whole program from scratch. The `clean` rule is marked as *phony*, because its target is not an actual file that will be generated, but just an arbitrary name that we wish to use for executing this rule. In order to run any rule other than the first one, the user needs to pass the target name as a command-line argument to make. So, for instance, to clean everything in this example, just type `make clean`.

6 Further improvements

The makefile listed above is sufficient for most small programs you'll ever write. In this section, I'll go over a few improvements for larger projects.

6.1 Multiple source directories

Let's say you have some source files under `src/`, other source files under, `src/engine`, and some more source files under `src/audio`. Also, to make it more interesting, let's assume that your code is a mix of C and C++ in any of these directories. Here's a very simple modification which handles this situation:

```
csrc = $(wildcard src/*.c) \
       $(wildcard src/engine/*.c) \
       $(wildcard src/audio/*.c)
ccsrc = $(wildcard src/*.cc) \
        $(wildcard src/engine/*.cc) \
        $(wildcard src/audio/*.cc)
obj = $(csrc:.c=.o) $(ccsrc:.cc=.o)

LDFLAGS = -lGL -lglut -lpng -lz -lm

mygame: $(obj)
    $(CXX) -o $@ $^ $(LDFLAGS)
```

The rest is exactly the same as previously. Note that I used the `CXX` built-in variable, which defaults to `c++` instead of `cc`, to invoke the linker in the correct way for C++ programs, automatically linking `libstdc++` with it. Also, backslashes at the end of lines serve to merge multiple lines together by escaping the newline character.

6.2 Handling cross-platform differences

So far, in all the examples, I've linked OpenGL and GLUT by passing the `-lGL -lglut` flags to the linker. That's how reasonable UNIX systems do it, but what if we want to build our project on vaguely UNIX-like abominations like MacOSX, where you have to use the following command-line arguments: `-framework OpenGL -framework GLUT` (since OpenGL and GLUT they are "*frameworks*" and not *just* libraries ... *sigh*) ? Simple; use `uname -s` to figure out if we're building on MacOSX, and modify the `LDFLAGS` variable accordingly:

```
LDFLAGS = $(libgl) -lpng -lz -lm

ifeq ($(shell uname -s), Darwin)
```

```
    libgl = -framework OpenGL -framework GLUT
else
    libgl = -lGL -lglut
endif
```

The order of defining `libgl` and defining `LDFLAGS` doesn't matter, as long as they're both before the first rule which uses either of them, since `LDFLAGS` won't be evaluated (and thus require substitution of the `libgl` variable) until it's used in a rule.

A slightly more elegant, but not exactly equivalent way of doing the above would be the following instead:

```
LDFLAGS = $(libgl_$(shell uname -s)) -lpng -lz -lm

libgl_Linux = -lGL -lglut
libgl_Darwin = -framework OpenGL -framework GLUT
```

I said not exactly equivalent, because in this way we would have to enumerate all possible UNIX systems we'd like to support instead of just handling them in the `else` part of the conditional statement.

6.3 Wildcard rules

As I mentioned above, make knows how to compile C and C++ source files to create object code, and you don't have to explicitly write rules of the form:

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

But let's say you want to compile code for a new language, which make knows nothing about. There is a mechanism to write generic rules, so that you won't have to laboriously type specific rules for each file you wish to compile.

If you wish to inform make, on the generic form of rules for building object files out of source files in a fictitious *foo* language, which are commonly in files with a `.foo` suffix, and compiled by running `fooc -c`, you can write the following wildcard rule:

```
%.o: %.foo
    fooc $(FOOFLAGS) -o $@ -c $<
```

Then, whenever make needs to build `xyzzy.o`, and there is a file named `xyzzy.foo`, it will use that rule to compile it.

6.4 Automatic #include dependency tracking

> Update: it turns out there's a better way to do this, make sure to also read the next section for an improved version. Thanks to Dan Raymond for pointing it out. I'm keeping this section in the document for posterity, and because I don't want to reformulate the explanations of how this works and why it's necessary, which are still valid.

The built-in rule for compiling C source files, is similar to:

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

Which means, that if we change `foo.c`, `foo.o` will be out of date and will be rebuilt automatically. There are, however, usually many more dependencies that need to be considered to determine if `foo.o` needs rebuilding, which fall through the cracks in this simple but generic scheme.

Specifically, if `foo.c` also includes `foo.h` and `bar.h`, then if we fail to rebuild `foo.o` when either of these header files change, will result in a potentially incorrect program, and depending on the nature of the modifications, possibly memory corruption, and hopefully segmentation faults at runtime.

For simple hacks with a couple of source and header files, we can afford to just clean and rebuild any time we change header files significantly, for larger programs however, we'd really like to have our build system track dependencies due to header file inclusion too.

Make can't possibly know what constitutes a dependency, and include a syntactic analyzer to detect it for each and every language, which is why this doesn't happen automatically. However with the help of our compiler's pre-processor, we can create the necessary rules to do it.

We'll write a wildcard rule to generate makefile fragments with an explicit rule for each source file in our project, which also includes header files in the dependency list. Then we'll instruct make to include these makefile fragments in our makefile as if we wrote them by hand. Any missing makefile fragments will be automatically generated with our wildcard rule during this inclusion. Finally we'll add a rule to clean all these dependency files (the makefile fragments).

```
src = $(wildcard *.c)
obj = $(src:.c=.o)
dep = $(obj:.o=.d)  # one dependency file for each source

LDFLAGS = -lGL -lglut -lpng -lz -lm

mygame: $(obj)
    $(CC) -o $@ $^ $(LDFLAGS)

-include $(dep)   # include all dep files in the makefile

# rule to generate a dep file by using the C preprocessor
# (see man cpp for details on the -MM and -MT options)
%.d: %.c
    @$(CPP) $(CFLAGS) $< -MM -MT $(@:.d=.o) >$@

.PHONY: clean
clean:
    rm -f $(obj) mygame

.PHONY: cleandep
cleandep:
    rm -f $(dep)
```

The added lines are marked with comments in the example above. To see exactly

what the C preprocessor outputs when instructed to write dependency information, try running the following in a file with a single include statement:

```
cpp foo.c -MM -MT foo.o
```

That's right, it just outputs a single makefile rule! Which is exactly what we need for make to track all the dependencies correctly.

6.5 Improved automatic dependency tracking

The dependency generation mechanism described in the previous section, is slightly more complicated than necessary, and requires a separate (although automatic) pass through the source files to generate the dependency files, which consumes significant time on older machines or huge projects.

There's a way to instruct the compiler (GCC and clang supports this, more compilers might do so too), to generate the necessary makefile fragment (`.d` file) at the same time as it compiles each source file. This is done by passing the `-MMD` option to the compiler.

The absence of the `.d` files during the first make will not be detrimental, because at that point we're doing a full build anyway, and subsequent make invocations will have the necessary makefile fragments included to aid in deciding which source files need to be rebuilt because a header file has changed.

This approach, apart from reducing initial build times, also simplifies our makefile from the previous section slightly, since there's no need for a custom `%.d: %.c` rule for generating the dependency files:

```
src = $(wildcard *.c)
obj = $(src:.c=.o)
dep = $(obj:.o=.d)  # one dependency file for each source

CFLAGS = -MMD     # option to generate a .d file during compilation
LDFLAGS = -lGL -lglut -lpng -lz -lm

mygame: $(obj)
    $(CC) -o $@ $^ $(LDFLAGS)

-include $(dep)   # include all dep files in the makefile

.PHONY: clean
clean:
    rm -f $(obj) mygame

.PHONY: cleandep
cleandep:
    rm -f $(dep)
```

The only changes from the previous example, is the removal of the wildcard rule, and the addition of `-MMD` in `CFLAGS`.

6.6 Building sub-projects

Consider the following case: our game has source files in `src/`, but we also need to use an external library called `libfoo`, which is not a system-wide installed library, but one we want to carry with our source tree under `libs/foo`. How would we integrate that in our build system?

Very simply, instruct make to first build `libfoo`, by running `make` after changing to the `libs/foo` directory, and then link it as usual:

```
src = $(wildcard src/*.c)
obj = $(src:.c=.o)

LDFLAGS = -Llibs/foo -lfoo -lGL -lglut -lpng -lz -lm

# list the libfoo rule as a dependency
mygame: $(obj) libfoo
    $(CC) -o $@ $^ $(LDFLAGS)

.PHONY: clean
    rm -f $(obj) mygame

# this is the new rule for recursively building libfoo
.PHONY: libfoo
libfoo:
    $(MAKE) -C libs/foo
```

By adding the phony `libfoo` rule as a dependency to our binary linking rule, we essentially force make to always change into `libs/foo` and run make before linking our binary. If there is no need to actually rebuild `libfoo` because it has no changes, that makefile will do nothing and return immediately, so there's no harm in trying every time.

Arbitrarily complex programs can be built by using a similar approach, to build various modules as separate libraries. So this little snippet really scales our simple build system to easily handle huge projects as well.

7 Going the extra mile for release

I promised in the beginning that I'd give a few pointers on how to move from a practical makefile, to a release-quality build system. First of all, what do I mean by that. Surely in the last section we've seen many improvements that can be used to handle pretty much anything a project might need. We are missing however, some things that end-users expect; mainly "install" and "uninstall" rules, and build options.

7.1 Writing install/uninstall rules

Writing installation rules for standalone programs is very easy. Simply make your install rule depend on the binary, and copy it to the appropriate place. The appropriate place on UNIX systems is usually `/usr/local/bin`, but it may vary from system to system, and based on administrator preferences, so it's customary to use a `PREFIX` variable for the prefix to the path of the `bin` directory:

```
PREFIX = /usr/local

.PHONY: install
install: mygame
    mkdir -p $(DESTDIR)$(PREFIX)/bin
    cp $< $(DESTDIR)$(PREFIX)/bin/mygame

.PHONY: uninstall
uninstall:
    rm -f $(DESTDIR)$(PREFIX)/bin/mygame
```

Special care should be taken when writing install and uninstall rules, because these are necessarily executed with root privileges by the user, when installing system-wide. Avoid wildcards in `rm` commands like the plague.

Note that I also concatenated another variable in front of the `PREFIX`, called `DESTDIR`. This variable is most of the time undefined (and thus the empty string), and has no effect whatsoever; its use is to allow for staging installations to temporary directories before manually moving them to their actual place, and it's commonly required by software packaging and distribution systems.

So for instance running this with: `make DESTDIR=/tmp/stage install` will result in our program being installed under `/tmp/stage/usr/local/bin`, where it can be inspected and then moved safely by the packaging software to the correct place. This becomes much more important when dealing with installing libraries, which generally contain a lot more of files going many different places in the filesystem.

7.2 Installing libraries

Correctly building and installing shared libraries on various systems, is a big topic and definitely requires its own article. I may write that article at some point if there is any demand for it, but for now, let's stick to static libraries.

Static libraries are basically archives of a bunch of object files, conventionally using the `.a` suffix. The archive itself goes under `$(PREFIX)/lib`, while any public API header files it provides need to end up in `$(PREFIX)/include`. Here's a short makefile fragment, illustrating how to build and install libraries:

```
alib = libfoo.a

$(alib): $(obj)
    $(AR) rcs $@ $^

.PHONY: install
install: $(alib)
    mkdir -p $(DESTDIR)$(PREFIX)/lib
    mkdir -p $(DESTDIR)$(PREFIX)/include
    cp $(alib) $(DESTDIR)$(PREFIX)/lib/$(alib)
    cp foo.h $(DESTDIR)$(PREFIX)/include/

.PHONY: uninstall
uninstall:
```

```
	rm -f $(DESTDIR)$(PREFIX)/lib/$(alib)
	rm -f $(DESTDIR)$(PREFIX)/include/foo.h
```

7.3 Build options

Build options can clearly be specified by setting make variables on the command-line as I've shown previously in the `DESTDIR` example. Users, however really expect a *configure* script with common command-line options, which then generates the makefile.

One way to achieve that is to move the whole makefile, minus the optional parts, to a file called `Makefile.in`, and write a `configure` shell script similar to this:

```
#!/bin/sh

prefix=/usr/local
debugsym=true

for arg in "$@"; do
    case "$arg" in
    --prefix=*)
        prefix=`echo $arg | sed 's/--prefix=//'`
        ;;

    --enable-debug)
        debugsym=true;;
    --disable-debug)
        debugsym=false;;

    --help)
        echo 'usage: ./configure [options]'
        echo 'options:'
        echo '  --prefix=<path>: installation prefix'
        echo '  --enable-debug: include debug symbols'
        echo '  --disable-debug: do not include debug symbols'
        echo 'all invalid options are silently ignored'
        exit 0
        ;;
    esac
done

echo 'generating makefile ...'
echo "PREFIX = $prefix" >Makefile
if $debugsym; then
    echo 'dbg = -g' >>Makefile
fi
cat Makefile.in >>Makefile
echo 'configuration complete, type make to build.'
```

Then `Makefile.in` can contain things like `CFLAGS = $(dbg)`, and also use the `PREFIX` variable as usual. The user can then use the conventional build & install commands, and our makefile will be able to act accordingly:

```
./configure --prefix=/usr/local --disable-debug
make
sudo make install
```