

GNU Makefile Journal

Jason Nett, Daniel Cruz Alonso

4/23/13

Contents

1	Understanding Makefiles	1
1.1	Syntax	2
1.2	Simple Example 1	2
1.2.1	Creating Static Libraries and Compiling	3
1.2.2	Creating Dynamic Libraries and Compiling	3
1.2.3	Compiling this example with <code>makefile</code>	4
1.3	Pattern Rules in a Makefile	6
1.4	Pattern Rules and Variables	6
1.5	Phony Targets	7
1.6	Organizing Large Projects and Makefile	8

1 Understanding Makefiles

I'm following this video: <https://www.youtube.com/watch?v=U3wC1DJsWlc>

-
- Programs can consist of several source files
 - Number can grow quickly for large projects
 - There can be dependencies - Object files can reference each other
 - Change in a source file requires the object file to be recompiled and the executable relinked.
 - Header file changes can require multiple object files to be recompiled.

Knowing which source files have changed and need to be recompiled can quickly become complicated where you have many source files and a header file can be included by many source files.

-
- The programmer needs to track which files need to be recompiled or relinked.
 - This process can become tedious and error prone for complicated projects.
 - *One approach:* recompile the entire project whenever a change is made, but this could require hours of compilation time for large projects.
 - *Alternative:* use `make` to automate the task

-
- Several variants of `make` exist

- This tutorial uses GNU make
- make requires that we specify the dependencies that exist in our project
- ...and also requires commands to obtain object or executables from source
- make uses this information for proper rebuilds

-
- We give make this information as a ‘makefile’
 - It is usually named `makefile` or `Makefile`
 - `makefiles` consist of *rules*, which consist of *targets*, *prerequisites*, and *commands*.

1.1 Syntax

The syntax looks like:

```
target: prereq1 prereq2
    commands
```

NOTE: The whitespace on the second line must be a tab. This command tells make that `target` depends on `prereq1` and `prereq2`. Subsequently, `commands` tells make how to build a target from the prerequisites.

1.2 Simple Example 1

(In the same series of videos on YouTube, the previous video provided the following source code and I will use information from that video in this section.)

Let's create a `makefile` for a library example using source files `test_link.C`:

```
#include <stdio.h>

float cubed(float a);
float powerfour(float a);

int main()
{
    float x = 23.1415;
    float y = cubed(x);
    printf("%f\n", y);
    y= powerfour(x);
    printf("%f\n", y);
    return 0;
}

libcubed.C:

float cubed( float a )
{
    return a*a*a;
}

libpowerfour.C

float powerfour( float a )
{
    return a*a*a*a;
}
```

1.2.1 Creating Static Libraries and Compiling

We see that `test_link.C` has two other dependencies `libcubed.C` and `powerfour.C`. Execute this line to create object files for `libcubed.C` and `powerfour.C`:

```
gcc -c libcubed.C libpowerfour.C
```

This created `libcubed.o` and `powerfour.o`. Note that these are object files, not executables. They still need to be “linked” into an executable with `test_link.C`.

Combine these two object files into an “archive”:

```
jason-> ar rs libmymath.a libcubed.o libpowerfour.o
ar: creating libmymath.a
```

We have just combined the two object files into a single static library file `libmymath.a`.

Now we want to link this library `libmymath.a` to our main program. Note that if we simply tried to compile the main program `test_link.C` alone, we will get some compiler errors:

```
jason-> gcc -o test_link test_link.C
/tmp/ccaskn24.o: In function 'main':
test_link.C:(.text+0x16): undefined reference to 'cubed(float)'
test_link.C:(.text+0x3c): undefined reference to 'powerfour(float)'
collect2: ld returned 1 exit status
```

So to compile the main code with the library `libmymath.a` linked, execute:

```
gcc -o test_link test_link.C libmymath.a
```

with the library file simply added onto the end as an argument. Now the `test_link` executable appears.

```
jason-> ls
Compile.sh  Journal      libcubed.o  libpowerfour.C  test_link
Compile.sh~ libcubed.C  libmymath.a  libpowerfour.o  test_link.C
[~/Documents/Development/Tutorial_makefile]
jason-> ./test_link
12392.946289
286791.375000
```

- Linking with *static* (as opposed to *dynamic*) libraries merges object files to produce an executable
- Disadvantages:
 - Library code is embedded in the executable
 - Large number or large sized libraries
 - Executing a program maps it into memory, producing several copies

A better tool is to use *dynamic* libraries.

- *Static Libraries*: References are resolved during the linking stage. “Static linking”
- *Dynamic Libraries*: References are resolved during run-time (execution). “Dynamic linking”
However, in dynamic linking, the executable does NOT have the library code embedded within it. So those libraries *must* be available. With static linking, all you need is the executable because the library info is already inside it. Additionally, with dynamic linking, the operating system keeps only one copy of a library in memory.

1.2.2 Creating Dynamic Libraries and Compiling

First, generate PIC (“position independent code”) object files:

```
jason-> gcc -fPIC -c libcubed.C libpowerfour.C
[~/Documents/Development/Tutorial_makefile]
jason-> ls
Compile.sh  Journal      libcubed.o  libpowerfour.C  test_link
Compile.sh~ libcubed.C  libmymath.a  libpowerfour.o  test_link.C
```

This outputs `libcubed.o` and `libpowerfour.o`.

Second, combine the object files into a dynamic library with `gcc`.

```
jason-> gcc -shared -Wl,-soname,libmymath.so -o libmymath.so libcubed.o libpowerfour.o
[~/Documents/Development/Tutorial_makefile]
jason-> ls
Compile.sh  Journal      libcubed.o  libmymath.so  libpowerfour.o  test_link.C
Compile.sh~ libcubed.C  libmymath.a  libpowerfour.C  test_link
```

- `-shared`: produce a shared library
- `-Wl, -soname, libmymath.so`: (*Notice the lack of spaces after commas.*) The `-Wl` calls the linker and the rest assign a name to the shared library.
- `-o libmymath.so`: Output the shared library.

Recompile `test_link.C` while linking in the dynamic library `libmymath.so`:

```
gcc -o test_link test_link.C libmymath.so
```

However, when I try to run this:

```
jason-> ./test_link
./test_link: error while loading shared libraries: libmymath.so: cannot open shared object file: No such file
```

The compiler looks for shared libraries in some specific locations and is not finding the shared library in this case. We need to tell the dynamic linker where to find the dynamic libraries.

How do we do this?

- Environmental variable `LD_LIBRARY_PATH`
- Colon separated list of directories
- We add the current working directory to the list

Add the current directory “.” to `LD_LIBRARY_PATH`, assuming that it already has some paths listed.

```
jason-> echo $LD_LIBRARY_PATH

[~/Documents/Development/Tutorial_makefile]
jason-> export LD_LIBRARY_PATH=./$LD_LIBRARY_PATH
[~/Documents/Development/Tutorial_makefile]
jason-> echo $LD_LIBRARY_PATH
.:
```

Now try compiling and running again:

```
jason-> gcc -o test_link test_link.C libmymath.so
[~/Documents/Development/Tutorial_makefile]
jason-> ./test_link
12392.946289
286791.375000
```

1.2.3 Compiling this example with `makefile`

Try this Makefile:

```
test_link: test_link.C libmymath.so
    gcc -o test_link test_link.C libmymath.so
```

Recall the syntax from earlier:

```
target: prereq1 prereq2
    commands
```

We are specifying that executable `test_link` depends on source code `test_link.C` as well as the functions it calls in shared library `libmymath.so`. Those are the “dependencies” of the executable. Notice how the `commands` line of the `Makefile` is the same compilation command that we used previously for dynamic linking.

Now perform the compilation with just `make`.

```
jason-> make
gcc -o test_link test_link.C libmymath.so
```

We could also specify a target with `make [TARGET]`.

Let’s go back a step and troubleshoot a common error. Remove file `libmymath.so`. Now the compilation fails with a common error message:

```
jason-> make
make: *** No rule to make target 'libmymath.so', needed by 'test_link'. Stop.
```

We need to make a rule for this shared library if it does not exist. It’s looking for a rule whose `[TARGET]` is `libmymath.so`. Currently, none exists in `Makefile`.

Edit `Makefile` to:

```
test_link: test_link.C libmymath.so
    gcc -o test_link test_link.C libmymath.so

libmymath.so: libcubed.o libpowerfour.o
    gcc -shared -Wl,-soname,libmymath.so -o libmymath.so libcubed.o libpowerfour.o
```

Notice that the target is shared library `libmymath.so` and the two prerequisites that are combined into it are the two listed object files. Then, the command is the very same shared library compilation command from earlier.

```
jason-> make
gcc -shared -Wl,-soname,libmymath.so -o libmymath.so libcubed.o libpowerfour.o
gcc -o test_link test_link.C libmymath.so
```

Just to try again, suppose I remove all the `*.o` and `*.so` files and recompile with `make`:

```
jason-> make
g++ -c -o libcubed.o libcubed.C
g++ -c -o libpowerfour.o libpowerfour.C
gcc -shared -Wl,-soname,libmymath.so -o libmymath.so libcubed.o libpowerfour.o
gcc -o test_link test_link.C libmymath.so
```

They get generated.

- `libmymath.so` requires `libcubed.o` and `libpowerfour.o`
- There are no rules for the two object files
- But we don’t need any because of implicit rules for `*.o` files.
- We need `-fPIC` flag, so we can make our own explicit rules for those object files.

Add to `Makefile`:

```
test_link: test_link.C libmymath.so
    gcc -o test_link test_link.C libmymath.so

libmymath.so: libcubed.o libpowerfour.o
    gcc -shared -Wl,-soname,libmymath.so -o libmymath.so libcubed.o libpowerfour.o

libcubed.o: libcubed.C
    gcc -fPIC -c libcubed.C

libpowerfour.o: libpowerfour.C
    gcc -fPIC -c libpowerfour.C
```

Try editing `libcubed.C` and running `make` again. Only `libcubed` has a new object file made for it, then the executable is relinked. Unnecessary compilations were avoided.

1.3 Pattern Rules in a Makefile

Notice that the last two sections of the Makefile have the same pattern. If our project expanded to include even more such dependencies, this Makefile could quickly become very long.

We use a “pattern rule”. Replace

```
libcubed.o: libcubed.C
gcc -fPIC -c libcubed.C

libpowerfour.o: libpowerfour.C
gcc -fPIC -c libpowerfour.C
```

with

```
%.o: %.c
gcc -fPIC -c $<
```

This is a “pattern rule” for generating object files from C files.

- `%.o: %.c`
 - Means for a target that ends with `*.o`
 - The prerequisite is the same stem followed by `*.C`
- `gcc -fPIC -c $<` where `$<` is an automatic variable that expands to the name of the first prerequisite.

1.4 Pattern Rules and Variables

- *Assignment:* Variable assignment uses “:=”
- *Reference:* Variables are referenced by enclosing it in “\$()”

To the previous example, let’s say we have a couple more functions that get their own object files `libpowerfive.o` and `libsquareroot.o`. Don’t forget:

```
jason-> touch libpowerfive.C
jason-> touch libsquareroot.C
```

Then the Makefile looks like:

```
test_link: test_link.C libmymath.so
gcc -o test_link test_link.C libmymath.so

libmymath.so: libcubed.o libpowerfour.o libpowerfive.o libsquareroot.o
gcc -shared -Wl,-soname,libmymath.so -o libmymath.so libcubed.o libpowerfour.o \
libpowerfive.o libsquareroot.o

%.o: %.c
gcc -fPIC -c $<
```

So our project has a growing list of object files that appears more than once. It would be nice to define this list just once, then reference the list variable where ever we need. So let’s do that with our two new operators. Change the Makefile to:

```
OBJECTS := libcubed.o libpowerfour.o libpowerfive.o libsquareroot.o

test_link: test_link.C libmymath.so
gcc -o test_link test_link.C libmymath.so

libmymath.so: $(OBJECTS)
gcc -shared -Wl,-soname,libmymath.so -o libmymath.so $(OBJECTS)

%.o: %.c
gcc -fPIC -c $<
```

Now the list of object files is just called `OBJECTS` and is used with `$(OBJECTS)`.

A couple more shorthand symbols for make files:

- “\$@” Filename of the “target” (see above)
- “\$^” All files in the “prerequisites” (see above)

Using these new shorthands, in the most recent Makefile above, these two lines:

```
test_link: test_link.C libmymath.so
gcc -o test_link test_link.C libmymath.so
```

become

```
test_link: test_link.C libmymath.so
gcc -o $@ $^
```

Similarly, these following two lines:

```
libmymath.so: $(OBJECTS)
gcc -shared -Wl,-soname,libmymath.so -o libmymath.so $(OBJECTS)
```

become

```
libmymath.so: $(OBJECTS)
gcc -shared -Wl,-soname,$@ -o $@ $(OBJECTS)
```

And so the new Makefile is:

```
OBJECTS := libcubed.o libpowerfour.o libpowerfive.o libsquareroot.o

test_link: test_link.C libmymath.so
gcc -o $@ $^

libmymath.so: $(OBJECTS)
gcc -shared -Wl,-soname,$@ -o $@ $(OBJECTS)

%.o: %.c
gcc -fPIC -c $<
```

We see that our Makefile is become more concise, though also my cryptic-looking.

1.5 Phony Targets

- Target and prerequisites are not files
- A common “phony target” is `clean`
 - Deletes object files, libraries, and executables
 - Makes source ready for complete rebuild
- We will implement a `clean` target here.

Add these lines to the end of our Makefile:

```
.PHONY: clean
clean:
    rm *.o *.so test_link
```

What’s happening?

- Running `make clean` will cause make to build the `clean` target. The `clean` target here has no prerequisites and then executes its command just as you would from the terminal window manually again.
- This will work as long a there does NOT exist a file called `clean`. If there is one, then the Makefile will think that target `clean` is always up-to-date and not actually execute it.
- We can pre-empt this behaviour by telling the Makefile that `clean` is a “phony” target and it should not actually look for a file called `clean`. This is what the “`.PHONY: clean`” line does.
- **IMPORTANT:** Always make sure that this `clean` rule goes at the end of the Makefile. Also, make sure that the rule for the executable is the first one in the Makefile.

1.6 Organizing Large Projects and Makefile

(Many thanks to Daniel Cruz Alonso for contributing this section)

For large projects, we typically do not keep all of our source code, header files, and object files in the home directory of the analysis. Instead, let's keep our source code in a `src/` directory, the object files in an `obj/` directory, and library files in a `lib/` directory.

```
jason-> mkdir src
jason-> mkdir obj
jason-> mkdir lib
jason-> mv *.C src/
jason-> mv *.o obj/
jason-> mv *.a lib/
```

Let's start a new `Makefile` from scratch, though it will be based on adjustments to the previous one. Begin as before with a list of the object files assigned to a single array:

```
OBJECTS := libcubed.o libpowerfour.o libpowerfive.o libsquareroot.o
```

Now let's define some variables for the new directories we just created and moved files to:

```
SRCDIR := src
OBJDIR := obj
LIBDIR := lib
```

Every object file needs to be preceded by the directory name defined and there is indeed a shortcut for this. `_OBJS` is a variable definition created so that the adding and subtracting of needed (or unneeded) object files in the `OBJECTS` definition becomes easier (otherwise, for every object file you want to add you'd have to precede it with `$(OBJDIR)/`). The `patsubst` command takes care of this. It's a find-and-replace command that combs over whitespace-separated words. Its syntax is `$(patsubst <pattern>, <replacement>, <text>)`. It will look at every word in `<text>`, and if it finds a match for `<pattern>`, it will replace it with `<replacement>`.

So add to the `Makefile`:

```
_OBJS := $(patsubst %, $(OBJDIR)/%, $(OBJECTS)),
```

making it so that every `.o` file in `OBJECTS` (hence the wildcard, `%`) is replaced by that same `.o` file preceded by the intended `obj` directory. So from

```
libcubed.o libpowerfour.o libpowerfive.o libsquared.o
```

you end up with

```
obj/libcubed.o obj/libpowerfour.o obj/libpowerfive.o obj/libsquared.o
```

Subsequently, what was originally

```
%.o: %.C
    gcc -fPIC -c $<
```

is now

```
$(OBJDIR)/%.o: $(SRCDIR)/%.C
    gcc -fPIC -c -o $@ $<
```

Now it looks for the prerequisites where the `.C` files now are (in their source directory), and the added `"-o"` flag enforces the object file to have a name designated by the user; in this case, it's `$@`, which is the target, which is now `$(OBJDIR)/%.o`. Thus, the object files now end up in their own `obj/` directory, and don't clutter the main user directory.

Next, when creating the shared library, instead of

```
libmymath.so: $(OBJECTS)
    gcc -shared -Wl,-soname,$@ -o $@ $(OBJECTS)
```

it's now

```
$(LIBDIR)/libmymath.so: $(_OBJS)
    gcc -shared -Wl,-soname,$@ -o $@ $^
```

The main difference (aside from adding the library path `$(LIBDIR)`) is that instead of adding as a prerequisite `$(OBJECTS)`, now I add `$(_OBJS)`.

Finally, don't forget to add directory names to the lines linking the executable:

```
test_link: $(SRCDIR)/test_link.C $(LIBDIR)/libmymath.so
gcc -o $@ $^
```

I also added the shorthand symbols we've seen before in the second line.

The full Makefile now looks like:

```
OBJECTS := libcubed.o libpowerfour.o libpowerfive.o libsquareroot.o
SRCDIR := src
OBJDIR := obj
LIBDIR := lib
_OBJS := $(patsubst %, $(OBJDIR)/%, $(OBJECTS))
```

```
test_link: $(SRCDIR)/test_link.C $(LIBDIR)/libmymath.so
gcc -o $@ $^
```

```
$(LIBDIR)/libmymath.so: $(_OBJS)
gcc -shared -Wl,-soname,$@ -o $@ $^
```

```
$(OBJDIR)/%.o: $(SRCDIR)/%.C
gcc -fPIC -c -o $@ $<
```

```
.PHONY: clean
clean:
rm $(OBJDIR)/*.o $(LIBDIR)/*.so test_link.x
```

Testing it:

```
jason-> make clean
rm obj/*.o lib/*.so test_link.x
jason-> make
gcc -fPIC -c -o obj/libcubed.o src/libcubed.C
gcc -fPIC -c -o obj/libpowerfour.o src/libpowerfour.C
gcc -fPIC -c -o obj/libpowerfive.o src/libpowerfive.C
gcc -fPIC -c -o obj/libsquareroot.o src/libsquareroot.C
gcc -shared -Wl,-soname,lib/libmymath.so -o lib/libmymath.so obj/libcubed.o obj/libpowerfour.o obj/libpowerfi
gcc -o test_link src/test_link.C lib/libmymath.so
jason-> ./test_link
12392.946289
286791.375000
```