Warwick Research Software Engineering

# Makefiles

*H. Ratcliffe and C.S. Brady*
Senior Research Software Engineers

“The Angry Penguin”, used under creative commons licence
from Swantje Hess and Jannis Pohlmann.

June 15, 2018

## Contents

## 1 About these Notes

These notes were written by H Ratcliffe and C S Brady, both Senior Research Software Engineers in the Scientific Computing Research Technology Platform at the University of Warwick. This document is an exerpt from notes on Introductory Software Development, a series of Workshops first run in December 2017.

The notes were typeset in LaTeXby H Ratcliffe.
Errors can be reported to rse@warwick.ac.uk

## 2 Other Useful Information

Sometimes we show snippets of commands for shell and make. These often contain parts which you should substitute with the relevant text you want to use. These are marked with {}, such as

```
git branch {name}
```

where you should replace "name" with the relevant text.

### 2.1 Glossaries and Links

We also include a glossary of terms. Words throughout the text that look like this: dependency are links to these. Numbers[1] are links to footnotes placed throughout the text.

[1] Like this

## 3 Make

### 3.1 Why Use a System

For small programs, containing only a few files, it is straightforward to simply type the compile command when you need it. For example, you may have a command like

```
gcc test1.c test2.c -o test
gfortran test1.f90 -o test
gcc -pedantic test1.c -o test -lm
```

For a few files, a few libraries and a few compiler flags this is fine. But as the list grows it becomes easy to forget or misspell files, or forget to link a crucial library. It is also irritating to have to recompile everything every time, even though only a few files have changed, and it is impossible to run the build in parallel.

### 3.2 Build Scripts

The simplest option for automating your code compilation is a script, typically a shell script. This allows you to build with a simple command instead of typing out a long line. You can of course do all sorts of clever things like parameterising the script, so that you can compile and link separately etc, but you can't solve the problems of recompiling everything, nor can you parallelise building.[2] There's also no easy way to tell if you *need* to recompile, since even files which have not changed themselves need re-doing when files they depend on change.

### 3.3 Using Make

In 1976 somebody got fed up of wasting time debugging a program where it turned out the bug had been fixed hours ago but the recompilation was failing. His solution was to create a system to do things automagically, called Make, which is what we're going to discuss here.

Make is the most widely used build tool for compiled languages. The core Make standard is supported by all of the variants, but more advanced features and extensions are specific to a given implementation. Here we are going to restrict to Gnu Make although much of what we introduce is universal.

#### 3.3.1 What Make Does and Doesn't Do

If you have ever built a large code from source, you may be familiar with the classic sequence

```
./configure
make
make install
```

[2]Or rather, this would entail recreating Make yourself

This first configures the build for your particular machine and operating system, checking, for example, the size of an integer, and the availabliity of certain libraries etc, and creates a suitable "Makefile". The next line compiles the code, and the last line installs it, usually by copying the executable into the right place and telling the system where it is.

Make is responsible for the compile step here. A different tool, Gnu Autotools provides the configure step, checking for the name of the compiler and such things. Finally the last line uses Make, but probably only to call some shell or other commands that do the actual installation tasks.

### 3.3.2 Makefiles

To build codes using Make, you first create a Makefile.[3] In the previous subsection, Autotools was used to do this, but you will probably want to create your files by hand.

The basic idea of Make is to build *targets* using *recipes*, based on the modification state of their *prerequisites*(see also dependency). A Make rule contains several sections:

- target - something to be made. For example a file, such as a .o file produced from a .f90 or .c file, an action, such as the install action above.
- prerequisites - things that need to be made before this target can be. Can be other targets, or can be files.
- recipe - the command to build the target.

To tell Make to build a particular target, you type "make [target name]". Make checks all of the target's prerequisites. If any of these need to be rebuilt, it builds them. When this is done, it runs the recipes for the requested target.

Note the obvious property of this: if you were to make a target its own prerequisite you have an infinite loop. This pattern, circular dependency, must be avoided. While Make will warn you so nothing bad happens, you definitely wont get the intended result.

Make decides whether a target needs rebuilding based on modification times. For targets and prerequisites which are files, it checks when they last changed on disk, and if this is newer than the target's last modification, the target is rebuilt. For non-file targets there are some subtleties, discussed in Sec 3.3.10.

### 3.3.3 Rules

Rules are the blocks in the file which look like

```
Target : prerequisites
    recipe
```

The first rule is special and is called the default, and is invoked if you type simply `make`. Otherwise you can invoke a specific rule using `make {target name}`

[3]The name can be Makefile or makefile, or any other name you wish, although you then have to specify the filename to make.

### 3.3.4 Recipes

Make recipe lines must be indented using a TAB character. Spaces will not do. If you forget a tab, or use an editor which swaps them to spaces, you will see something like

```
Makefile:6: *** missing separator.  Stop.
```

Note that the "6" is the line number where the error occured.

### 3.3.5 Intermediate Files and Linking

For more complicated programs that aren't built in a single line, there are distinct "compilation" and "linking" steps. First, each source code file is built into an "object" file and then all the object files, and any libraries you need, are "linked" into the final executable program. Object files usually end in .o, sometimes .obj (especially on windows). Fortran module files are similar (although distinct from Fortran object files). If you use C maths libraries, for example, you may have to use "-lm" in your compile step to link them.

### 3.3.6 A First Useful Makefile

For a simple C program, a basic makefile is:

```
code : test.o
    cc -o code test.o
test.o : test.c
    cc -c test.c
```

The first, default rule, has one prerequisite, "test.o" and builds the final program. The second says how to build "test.o", in this case from a single file, "test.c".

Even this simple file unpacks to quite a complicated process. When we type `make`, the first rule is invoked by default. This finds that it must first build "test.o". Make jumps to the rule for "test.o". This depends on "test.c", which is not a target, just a file. So Make checks whether "test.o" needs rebuilding, and does so if necessary. The it goes back up to the code rule, and rebuilds this if necessary.

If we run "make" once, and then run it again, we see things like

```
make: 'target' is up to date.
make: Nothing to be done for 'target'.
```

This is great. We know our code has recompiled, and for large codes we don't waste any time rebuilding unnecessarily.

### 3.3.7 Variables

The simple compilation rules above are useful, but for more interesting tasks you'll want to use variables in your makefile. For example, suppose all your source files are in a directory "src": you could type this in every rule but you risk spelling mistakes, and it becomes a lot of work to rename the directory.

Variables in Make are set and used like

```
variable = value
$(variable)  #Use variable.
```

The dollar indicates this is a variable, while the brackets allow the name to be more than a single character. If you forget the brackets you wont get a helpful error. Instead the line will be interpreted as a single-letter variable followed by the rest of the variable name. Remember the brackets around Makefile variables.

Because of the nature of makefile commands, the file is read multiple times, and then any rules are executed, so some things can appear to happen out of order. This allows the system to be very powerful. For the current purposes all we need to worry about is variable assignments. *Makefile variable assignments have one major difference to those in other programming languages.* The normal "=" operator sets two variables to be the same[4], like this

```
var1 = test
var2 = $(var1)
var1 = test2
$(info $(var2)) #Print the variable. -> test2
```

The ":=" operator does the assignment using the value right now[5]

```
var1 = test
var2 := $(var1)
var1 = test2
$(info $(var2)) #Print the variable. -> test
```

Note also that we don't use quotes on the strings here. Make treats quote marks like any other characters, so they would just becomes part of the string.

Make also provides a selection of *automatic variables* for use in rules. These give you access to things like the target being built and the prerequisites. In particular we have

```
    $@ # target of the current rule
    $< #first prerequisite of currrent rule
    $^ # space separated list of all prerequisites
```

For example this snippet shows the use of these by setting the recipe to write to the shell. The "@" character here tells make not to print the command it is about to run.

```
other : final
    @echo $@
final :
    echo $@
test: other final
    @echo $@ '|' $< '|' $^
```

When we run this with "make test" we get

[4]i.e. var2 becomes a reference to var1
[5]i.e. var2 becomes a copy of var1

```
echo final
final
other
test | other | other final
```

Notice a few things here. Make looks at the "test" target and sees it must first build other and final. It jumps to the "other" rule. This depends on "final", so it jumps to that rule. Final has no prereqs, so it can start building. It builds final (notice we see both the echo command and its result here). Now it returns to "other": this can now be built. After that, it returns to test. It knows "final" is already built so does not do that a second time. Now it builds test, and we see the automatic variable contents separated by pipes (|).

### 3.3.8 Implicit Rules

Make has one final internal variable that is very useful, which is the rule wildcard. The "%" stands for any character sequence, but anywhere it appears in the rule it is the same sequence. This is used in what are called implicit rules, for "implicit" targets, i.e. those that aren't specifically mentioned in the makefile. These are also called pattern rules, because they apply to targets matching a pattern.

The simplest sort of pattern rule looks like

```
%.o : %.c
    cc -c $<
%.o: %.f90
    gfortran -c $<
```

These rules match any target that looks like "[name].o" and build them from the corresponding "[name].c" or "[name].f90".

A full discussion of pattern rules is at https://www.gnu.org/software/make/manual/html_node/Pattern-Rules.html. Note that Make has a lot of built-in rules for the normal procedure for a given language, summarised at https://www.gnu.org/software/make/manual/html_node/Catalogue-of-Rules.html#Catalogue-of-Rules.

### 3.3.9 Multiple Rules

You may notice that the pattern rules above are perfect for generating a .o from a single .c or .f90, but don't allow you to specify more dependencies for your code. For example, in c %.o would usually depend on %.c and %.h. Or you may have some helper functions in a file which all your other files depend on. Make allows you to define multiple rules for the same target. Only the last recipe is run, but prerequisites from all rules are considered. This allows us to use the pattern rules like above, and also, elsewhere in the file, specify our file dependencies. See the example in Sec 3.3.11.[6]

[6] You can also write a rule with multiple targets to add some prereqs to them all. This can be useful for things like helper functions which are prereqs of all your code.

### 3.3.10 Other Bits

Make has many other features to aid you in processing filenames, putting things in the right directories, and controlling your compilation by passing variables to make itself. We are not going to go into the details here, as there are many good tutorials out there, and you are best served learning the details when you actually need them. However, one last feature of Make is very common and quite important, which is using Make for a target which is not a file.

As we saw above, Make can run any command you wish, and targets need not correspond to a filename. These targets are always rebuilt, as Make has no way of tracking their last modification. A common use of this is to have a "clean" target, which cleans up intermediate and output files. This is fine, unless a file ever exists which is called "clean", when make will think this target is up-to-date and do nothing. To avoid this, and to make things clearer, make has a special target, `.PHONY`. Any prereqs of this special target are assumed to be phony, i.e. not to correspond to any file.

### 3.3.11 Full Example Makefile

A basic, but useful, Makefile is along the following lines:

```
# Set compiler name
CC = cc

#Set phony targets
.PHONY : clean

#Default rule
code : test.o
	$(CC) -ocode test.o
#Pattern for compiling .c files
%.o : %.c
	$(CC) -c $<

#List the dependencies of test.o here
#It will be built with the rule above
test.o : test.c
#Clean rule
clean:
	@rm -rf code
	@rm -rf *.o
```

### 3.3.12 Parallel Building

The simple build scripts tend to use a single line to compile everything and then link in a single step. With Make, you instead give it dependencies, so it knows the order in which it needs to build files. This means it can work out which rules can be built simultaneously, and parallelise your building. Using `make -j {n_procs}` Make will use up to n_procs, but only as many as it can. This can speed things up a lot.

#### 3.3.13 Pitfalls

Make is very powerful but it does have some traps that are easy to fall into.

- Forgotten dependencies

  Target wont rebuild when it needs to

  Can be very hard to diagnose

  Your compiler can often generate dependencies (see gcc -M)

- Circular dependencies

  Make will ignore these

  Might create a forgotten dependency

- Permanent rebuilding

  Non-file targets always rebuild

  Any rule that doesn't actually build the target file can too

- Overcomplication

  You can do all sorts of conditional compilation with Make

  See Sec 3.4 for tools beyond make

## 3.4 Cmake and Other Tools

As well as Make which we focused on, there are many other tools for building code based on dependencies and rules. Several of these, such as Gnu Autotools and qmake generate Makefiles for you; some use Make as a backend but also support other options, such as cmake, and some have their own system, such as meson. These have their own strengths and weaknesses. You may encounter them when using libraries, for example qmake is made by the developers of the QT GUI libraries, and is almost essential for building QT projects.

Once your makefiles need to account for things like multiple platforms (OSX, Linux flavours etc) or installing their own copies of needed libraries, you will want to look into these tools. However, Make can serve most of your needs for quite a long time.

# Glossary

**automagically** (humorous) Automatically, as if by magic. Used mostly for systems with nice properties of doing what is actually needed rather than following a simple recipe, or when the details of how it works are tedious and boring, but the outcome very useful. 2

**backwards compatibility** A guarantee that anything possible or valid in an older version remains possible, so that e.g. input or output files from an older version can still be used. Sometimes this means that you can make the code behave exactly as it used to, sometimes it means only that you can use the files as a base for new files. For example Excel can read any Excel file, from any version correctly. *See also* forwards compatibility & sideways compatibility,

**compiler flag** (Aka directive) Command line arguments passed to the compiler to control compilation. For example in C you can define a value (for use with #ifdef etc) using -D[arg_name][= value]. Optimisation levels (how hard the compiler works to speed up or reduce memory use of your program) are usually set with a directive like -O[level number]. 2

**dependency** In general terms dependencies are the libraries, tools or other code which something uses (depends on). In build tools specifically, they are also known as prerequisites and are the things which must be built or done before a given item can be built or done. For example I may have a file-io module which I have to build before I can build my main code. 1, 3

**forwards compatibility** A guarantee that files etc from a newer code version will still work with the older version, although some features may be missing. E.g. a file format designed to be extended: extended behaviour will be missing, but will simply be ignored by old versions. *See also* backwards compatibility & sideways compatibility,

**library** Code to solve a problem, intended to be used by other programmers in their programs. For example, in C there is the Standard Library which contains things like mathematical functions, string handling and other core function which is not part of the language itself. *See also* module & package, 9

**module** A piece of a program, something like chapters in a book. In python modules are the things you import (possibly from a larger package). Fortran has modules explicitly, that are created using MODULE [name] and made available elsewhere with the USING statement. In C modules are closest to namespaces. *See also* package & library, 9

**package** A piece of software, usually stand-alone. In contrast a library is usually code only used by other programs, but there is a lot of overlap. This may contain multiple smaller modules. In context of OSs, packages are software that can be installed, whether they're available as compiled binaries or source code. *See also* module & library, 9

**sideways compatibility** A guarantee that code remains compatible with other code. For example you may create files for another program to read, and you want to make sure that your output remains compatible with their input requirements, even when these may change. *See also* forwards compatibility & backwards compatibility,