

C Programming under Linux

P2T Course, Semester 1, 2004–5
Linux Lecture 6

Dr Graeme A Stewart

Room 615, Department of Physics and Astronomy
University of Glasgow
`graeme@physics.gla.ac.uk`

Summary

- Larger C Projects
- The Make Utility
- How does Make Work?
- Make in More Detail
- Help with Make
- Make inside XEmacs
- Extra Make

`http://www.physics.gla.ac.uk/p2t/`

`$Id: linux-lecture-06.tex,v 1.6 2004/09/24 13:44:54 graeme Exp $`

Building C Projects I

It's easy enough to build a C program from a single source file:

```
$ gcc -o foo foo.c  
[...]  
$ ./foo  
[...]
```

When the file is edited, changing the source code, rerun the compiler and rebuild the program.

Once you start larger projects you'll need to split your source code up into several source files:

```
$ gcc -o fbz foo.c bar.c baz.c  
[...]  
$ ./fbz  
[...]
```

Building C Projects II

As projects get bigger they will accumulate more and more source files (the Linux kernel has almost 5000), so recompiling on the command line is not an option which scales.

It's also inefficient, `gcc -o fbz foo.c bar.c baz.c` will recompile all of these source files – even if they haven't changed.

So, all projects in unix use a utility called `make`, which helps automate building binaries from a whole set of source code files.

In the example above, if only `bar.c` had changed, then `make` would *only* rebuild `bar.o`, then assemble `fbz` from the object files:

```
$ make fbz
gcc -c bar.c
gcc -o fbz foo.o bar.o baz.o
$ ./fbz
[...]
```

A Makefile Example

The `make` utility uses a file, by default called `Makefile` or `makefile`, to determine:

- what 'targets' to build
- how to build them

In the last example the `Makefile` might have looked like;

```
# Makefile for fbz
#
fbz: foo.o bar.o baz.o
    gcc -o fbz foo.o bar.o baz.o

foo.o: foo.c
    gcc -c foo.c

bar.o: bar.c
    gcc -c bar.c

baz.o: baz.c
    gcc -c baz.c
```

Inside a Makefile

The essence of a `Makefile` is that it contains a number of

- *targets* – things to be built
- *dependencies* – things the target is made from
- *rules* – how to make the target

```
fbz : foo.o bar.o baz.o
gcc -o fbz foo.o bar.o baz.o
```

Important: This is a TAB character, not 8 spaces!

In this case the target `fbz` depends upon the object files `foo.o`, `bar.o` and `baz.o` and is rebuilt using the command `gcc -o fbz foo.o bar.o baz.o`.

But hold on – as you never change the object files themselves, how does `make` know when to rebuild `fbz`?

Rule Chains

In our example `make` also had rules on how to build the object files from their corresponding C source files:

```
foo.o: foo.c
    gcc -c foo.c
```

And when we ask `make` to rebuild `fbz` it first checks if the dependencies of `fbz` themselves need to be remade. e.g., if `bar.c` has been changed then `make fbz` works like this:

- `fbz` depends on `foo.o`, `bar.o` and `baz.o`...
- `foo.o` depends on `foo.c`, but it is newer than `foo.c` so doesn't need remade.
- `bar.o` depends on `bar.c`, and `bar.c` has changed since `bar.o` was last made, so I need to remake it:

```
gcc -c bar.c
```

Rule Chains (cont.)

- `baz.o` depends on `baz.c`, but it is newer than `baz.c` so doesn't need remake.
- Now `foo.o`, `bar.o` and `baz.o` are up to date. Do I need to remake `fbz`? Yes – because now `bar.o` is newer than `fbz`:

```
gcc -o fbz foo.o bar.o baz.o
```

In this way `make` can deal with arbitrarily long chains of rules and decide which targets need to be remake.

(There's a subtlety to be pointed out here: in the last example `make` also considered the files `foo.c`, `bar.c` and `baz.c`; but as there were no rules for how to remake them, `make` assumes that they have no dependencies and are always 'up to date'.)

`make` is in fact a *logical* programming language (like `prolog`), instead of an *imperative* one (like `C` or `Java`). The `Makefile` establishes a set of logical relations that `make` uses to draw conclusions (like 'I need to rebuild `fbz`').

Make's Scope

Although most of the examples which we'll look at for `make` concern the building of executable files from C code, `make` is not limited to that task – it can build *any* target from *any* dependency. Here's a simplified extract from the `Makefile` for these lectures:

```
linux-lecture-01.pdf: linux-lecture-01.ps
    ps2pdf linux-lecture-01.ps
```

```
linux-lecture-01.ps: linux-lecture-01.dvi
    dvips linux-lecture-01.dvi -o
```

```
linux-lecture-01.dvi: linux-lecture-01.tex
    latex linux-lecture-01.tex
```

```
linux-lecture-01.4a4.ps: linux-lecture-01.ps
    mpage -4 linux-lecture-01.ps > linux-lecture-01.4a4.ps
```

Here `make` has been taught how to build the `pdf` and printable versions of the lecture notes from the source file

`linux-lecture-01.tex`.

What is a Target

Up to now all the targets and dependencies which we've seen are `files`. The way that `make` works is to compare the timestamps on these files: if a target is newer than all its dependencies then it is assumed to be up to date. If it's older than any of its dependencies then it needs to be rebuilt.

`make` can also have targets which are not files. These are called *phony targets*. As something which does not exist (as a file) cannot have a timestamp, `make` always considers that a phony target needs 'rebuilt', i.e., it always executes the 'build commands'.

Such phony targets are very common:

```
clean:
```

```
    rm -f *.o *~ fbz
```

```
install:
```

```
    cp fbz /usr/local/bin
```

```
    cp fbz.1 /usr/local/man/man1
```

Default Targets, Phony Targets and Chains

When `make` is called with a target name it will remake that target:

```
$ make fbz                # make the fbz target
[ ... ]
$ make clean              # make the clean target
[ ... ]
```

If `make` is called without a target name, it will remake the first target in the makefile:

```
# Top of my makefile
all: binaries documentation

binaries: sort list dataproc

documentation: sort.pdf list.pdf dataproc.pdf
```

In this case `make` is equivalent to `make all`.

Note that all of `all`, `binaries` and `documentation` are phony targets. That's quite acceptable and normal: in this case `make all` is a shorthand target for `make binaries` and `make documentation`.

What's in a Build Rule?

Note that the build commands which `make` executes are just shell commands. This is good, because we can write arbitrarily complex bits of shell code in the build instructions:

`clean:`

```
find . \( -name '*~' -o -name '*.o' \) | xargs rm -f
rm -f fbz
```

Note that each line of shell code is executed in a separate shell environment, so commands which work in concert need to be on the same line or have backslash separators:

`doctar:`

```
cd doc && tar -cvzf - *.tex > ../docs.tar.gz
```

That's not the same as:

`doctar:`

```
cd doc
tar -cvzf - *.tex > ../docs.tar.gz
```

(which is probably wrong as the first line is useless.)

Variables in Make

Makefiles can have variables in them, just like shell scripts:

```
OBJS=foo.o bar.o baz.o  
LIBS=-lfoo -lbar -lm
```

```
fbz: $(OBJS)  
    gcc -o fbz $(OBJS) $(LIBS)
```

Note the important difference with the shell: variables in makefiles need to be enclosed in parentheses, and spaces do not need to be quoted!

(And while we're on the subject of syntax, remember again that build rules start with a TAB, not spaces!)

Just like the shell `make` has lots of special handy variables. Particularly common ones (which you'll see in other people's makefiles) are:

- `$@` The name of the target
- `$<` This target's first dependency

Implicit Rules

`make`, as we pointed out, can be used to build just about anything. There are, however, many scenarios where `make` is used in a completely standard way: e.g., building `foo.o` from `foo.c`. To save effort `make` has a set of predefined *implicit rules*. One of these rules states that:

Compiling C programs

`'N.o'` is made automatically from `'N.c'` with a command of the form
`'$(CC) -c $(CPPFLAGS) $(CFLAGS)'`.

Here's a `Makefile` for `fbz` which takes advantage of that:

```
CC=gcc
CFLAGS=-Wall
OBJS=foo.o bar.o baz.o

fbz: $(OBJS)
    $(CC) -o $@ $(OBJS)
```

Using the implicit rule we didn't have to state how to make `foo.o` from `foo.c`. (Note that if we had, that rule would have overridden the implicit rule.)

Implicit Rules and Variables

The last example showed how implicit rules could be 'customised', by changing the standard variables which form part of the rule.

If we had a different C compiler on the system to `gcc` and we wanted to change the compiler that `fbz` is build with, we just redefine to the new compiler, `CC=cc`; and run `make` anew.

Likewise we can add optimisation flags to the C compiler by modifying `CFLAGS`:

```
$ cat Makefile
CC=cc
CFLAGS=-Wall -O2
OBS=foo.o bar.o baz.o
LIBS=-lm

fbz: foo.o bar.o baz.o
    $(CC) -o $@ $(OBS) $(LIBS)

$ make
cc -c foo.c -Wall -O2
cc -c bar.c -Wall -O2
cc -c baz.c -Wall -O2
cc -o fbz foo.o bar.o baz.o -lm
```

Make Help

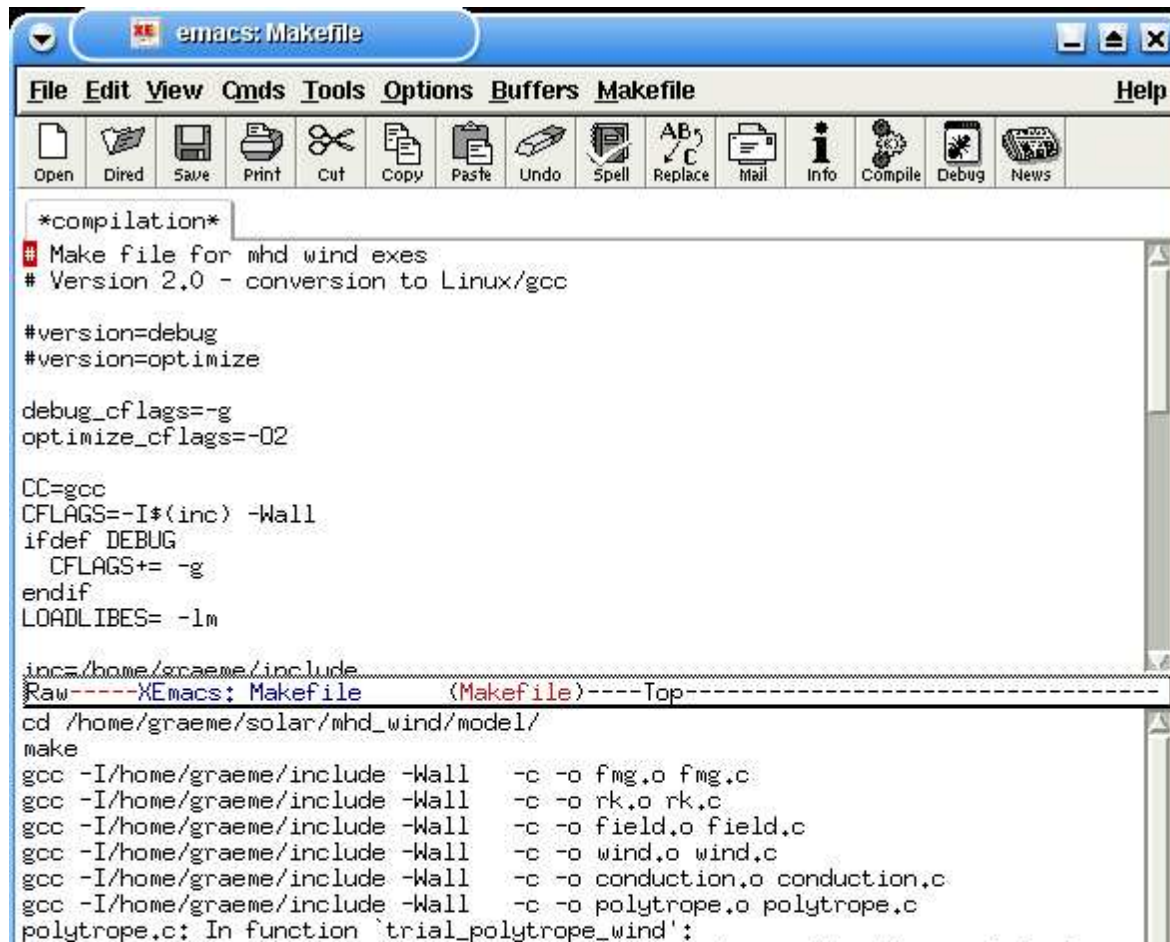
Your principal source of help with `make` is the info node. Start this with `info make` at the shell prompt or `C-h i` in emacs.

The info node contains a very useful introduction to `make`, as well as a complete reference to the command.

Make and XEmacs

XEmacs interacts with `make` very well. In XEmacs the command `M-x compile` will start a compile of a project. By default the compile command is `make -k` (but you can edit that).

Once a compile has started XEmacs will open a `*compilation*` window, where the results of running `make` are shown:



```
emacs: Makefile
File Edit View Cmds Tools Options Buffers Makefile Help
Open Dired Save Print Cut Copy Paste Undo Spell Replace Mail Info Compile Debug News
*compilation*
# Make file for mhd wind exes
# Version 2.0 - conversion to Linux/gcc

#version=debug
#version=optimize

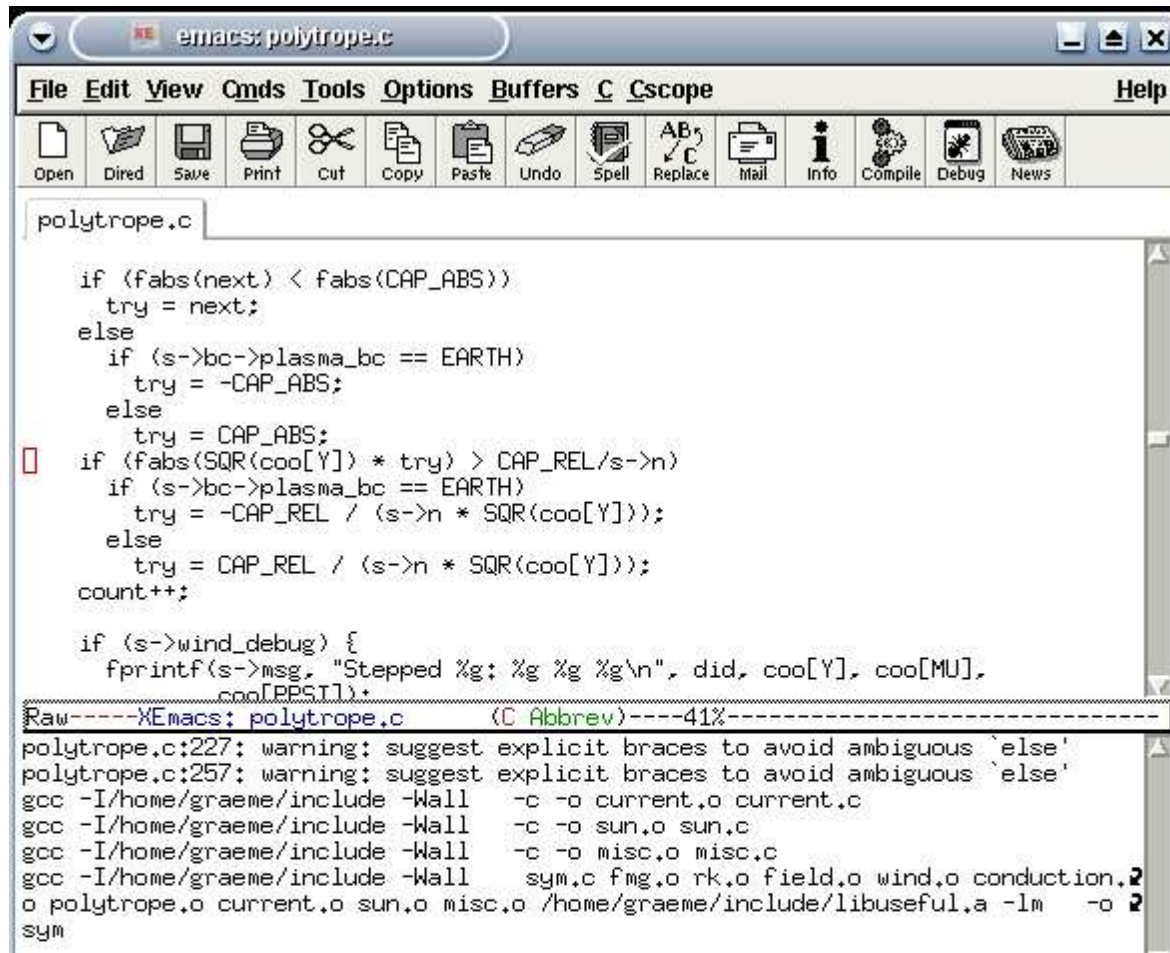
debug_cflags=-g
optimize_cflags=-O2

CC=gcc
CFLAGS=-I$(inc) -Wall
ifdef DEBUG
    CFLAGS+= -g
endif
LOADLIBES= -lm

inc=/home/graeme/include
Raw-----XEmacs: Makefile (Makefile)-----Top-----
cd /home/graeme/solar/mhd_wind/model/
make
gcc -I/home/graeme/include -Wall -c -o fmg.o fmg.c
gcc -I/home/graeme/include -Wall -c -o rk.o rk.c
gcc -I/home/graeme/include -Wall -c -o field.o field.c
gcc -I/home/graeme/include -Wall -c -o wind.o wind.c
gcc -I/home/graeme/include -Wall -c -o conduction.o conduction.c
gcc -I/home/graeme/include -Wall -c -o polytrope.o polytrope.c
polytrope.c: In function `trial_polytrope_wind':
```

XEmacs as an IDE

Even better, any errors which crop up during the compilation can be understood by XEmacs. Middle mouse clicking on the line with the error will open a buffer with the source file and move to the line which caused the error, ready for you to rectify it:



The screenshot shows the XEmacs IDE interface. The top menu bar includes File, Edit, View, Cmds, Tools, Options, Buffers, C, Cscope, and Help. Below the menu is a toolbar with icons for Open, Dired, Save, Print, Cut, Copy, Paste, Undo, Spell, Replace, Mail, Info, Compile, Debug, and News. The main window displays the source file 'polytrope.c' with the following code:

```
polytrope.c

if (fabs(next) < fabs(CAP_ABS))
    try = next;
else
    if (s->bc->plasma_bc == EARTH)
        try = -CAP_ABS;
    else
        try = CAP_ABS;
if (fabs(SQR(coo[Y]) * try) > CAP_REL/s->n)
    if (s->bc->plasma_bc == EARTH)
        try = -CAP_REL / (s->n * SQR(coo[Y]));
    else
        try = CAP_REL / (s->n * SQR(coo[Y]));
count++;

if (s->wind_debug) {
    fprintf(s->msg, "Stepped %g: %g %g %g\n", did, coo[Y], coo[MU],
            coo[PPST1]);
}
```

At the bottom, a buffer window shows the compilation output:

```
Raw-----XEmacs: polytrope.c (C Abbrev)-----41%-----
polytrope.c:227: warning: suggest explicit braces to avoid ambiguous 'else'
polytrope.c:257: warning: suggest explicit braces to avoid ambiguous 'else'
gcc -I/home/graeme/include -Wall -c -o current.o current.c
gcc -I/home/graeme/include -Wall -c -o sun.o sun.c
gcc -I/home/graeme/include -Wall -c -o misc.o misc.c
gcc -I/home/graeme/include -Wall sym.o fmg.o rk.o field.o wind.o conduction.o
polytrope.o current.o sun.o misc.o /home/graeme/include/libuseful.a -lm -o 2
sym
```

Extra Make

We've now covered the essentials of `make`, which should allow you to write your own makefiles for your C projects and to understand other people's makefiles that you come across.

However, it's worth pointing out that `make` can do a number of other things, which you might not need to do in your first makefiles, but it would be useful to be aware that such facilities exist.

New Implicit Rules

You can tell `make` about your own implicit rules. This is very useful if you want to define a way of processing a particular type of file.

e.g., from the `Makefile` for these lecture notes:

```
.PRECIOUS: %.dvi
%.dvi: %.tex
    latex $< && latex $<

.PRECIOUS: %.ps
%.ps: %.dvi
    dvips $< -o

%.4a4.ps: %.ps
    psnup -l -4 -pa4 $< $@

%.pdf: %.ps
    ps2pdf $<
```

These are the implicit rules for building `pdf` and printable `ps` files for any lecture. (`.PRECIOUS` means keep these intermediate files, which would normally be deleted.)

Make Functions

make has a number of useful builtin functions – all documented in the info node.

Two of the most useful are `wildcard` and `subst`:

- `$(wildcard PATTERN)` replaces itself with the results of a shell like wildcard in the current directory: e.g.
`$(wildcard *.c)` expands to all the `*.c` files in the current directory.
- `$(subst MATCH,REPLACE,STRING)` expands to the value of `STRING` with all occurrences of `MATCH` replaced by `REPLACE`.
e.g., `$(subst .c,.o,foo.c bar.c)` expands to
`foo.o bar.o`

These two functions are particularly powerful when used in concert:

```
foo: $(subst .c,.o,$(wildcard *.c))
```

This ensures that the target `foo` depends on all `.o` source files in the current directory.

Extra Notes

- If you need to pass a variable to the build command, then you have to use the syntax `$$VAR` – this stops `make` from expanding the variable itself (it's like escaping a variable in the shell):

```
buildclean:
```

```
    cd $$HOME/buildroot && rm -fr *
```

- GNU Make contains lots of goodies in it which are not part of standard unix `make` (this is why non-Linux systems might have both `make` and `gmake`). These include:
 - The ability to append to variables using `+=`

GNU Make Conditionals

- GNU Make can also...
 - Have conditional rules and definitions with constructions like `ifeq $(CONDITION) SOMETHING else OTHERTHING endif`

The one conditional you should know how to use is `ifdef`. This provides a simple switch based on whether a variable is defined:

```
CFLAGS=-O2
ifdef DEBUG
    CFLAGS+= -g
endif
```

Then, if `DEBUG` has a value we set the `-g` flag to add debugging information.

This flag could be set in the environment,

```
$ DEBUG=1 make
```

or just further up the makefile.

Last Notes on Building Packages

`make` is terribly useful for small and medium sized projects. However, it does not scale infinitely well – writing rules and dependencies quickly becomes tedious and it's also difficult to express differences in unix flavours though a makefile.

For these type of projects GNU offers `autoconf` and `automake`. These tools, in particular `autoconf`, automate the generation of makefiles tuned to a particular system. If you download a package to build from source the first step in building it is usually to run `./configure`, which is the `autoconf` script:

```
$ wget http://pkedu.fbt.eitn.wau.nl/~olivier/downloads/bluefish-0.12.tar.bz2
$ tar -xvjf bluefish-0.12.tar.bz2
$ cd bluefish-0.12
$ ./configure                                # <- this builds the Makefile
[ ... ]
$ make && make install
```


Copyright

All these notes are Copyright (c) 2003, Graeme Andrew Stewart.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.

A copy of the license is here:

<http://www.physics.gla.ac.uk/p2t/fdl.txt>