C: Part 12

Language of the 'C'

Following on from last month's article, Steven Goodwin, looks at how the *make* utility can be used to improve the development process. BY STEVEN GOODWIN

In preparing this section, I asked twelve different programmers for the best way to write a make file. I got twelve different answers! Writing *makefiles*, like code, novels or music is a uniquely individual experience. There is no right or wrong way – whatever works (and is readable!) can considered a 'good makefile'! The method we're using here is fairly 'traditional' and shall be developed from first principles, so you can see each step in the process.

Make

So first off; what is a *makefile*? And what is *make*? Well, *make* is a utility that helps reduce development time by allowing us to only rebuild parts of the project (using gcc) that need it; if you have not changed 'converter.c' or included header files, why would you want to spend time compiling it when the result will be the same as it was last week?! Conversely, if you've changed a header file that is used in four places, you would want to rebuild those source files to reflect the changes. A makefile (by default called *Makefile* – the capital M is important!) describes each component of the project, how they should be built, and what constitutes them being 'out of date'. The watchword here is *dependency*.

If we have a two file project where *converter.c* includes *converter.h*, then we can say *converter.c* is dependent on *converter.h*. If *converter.h* changes, it stands to reasons that *converter.c* must also have changed in some way, and so needs to be re-built. We can build a makefile to describe this. We can then build this program by typing:

```
make
```

If you had not named your file 'Makefile' but listing1make, for example, then you will need to use the -f flag.

```
make -f listing1make
```

Line 1 describes a *target*. The text to the left of the colon dictates what we want to produce (an executable file called *convunit* in this case), whilst the right hand side lists the dependant files we have to use in order to build it. Our makefile is effectively saying that should converter.c *or* converter.h change, then 'convunit' will be out of date and needs to be rebuilt.

Each subsequent line after a target that begins with a tab (and only a tab!) holds the command, or commands, that we must execute in order to produce the target file. It stands to reason, therefore, that those commands must produce the target file in some manner. You can include as many commands as you need;

Listing 1: Makefile

```
1 convunit: converter.c converter.h
2       gcc converter.c -o convunit
```

Listing 2: Makefile

```
1 convunit: converter.o
2       gcc converter.o -o ➥
convunit
3
4 converter.o: converter.c ➥
converter.h
5       gcc -c converter.c -o ➥
converter.o
```

semi-colons let you put two or more commands a line, while the backslash is available for line continuation if required. This is sometimes necessary, since each line executes in its own shell, and you might *need* to include several commands. The following would fail if each instruction was placed on a different line.

```
main:  source/converter.c ➥
  cd source;  gcc converter.c
```

make will execute each command in sequence until none is found (i.e. the line does not begin with a tab) or an error occurs. At this point it will stop trying to build that target and exit. To suppress these errors, start each command with a minus sign and it will continue with the next instruction (we'll see where that is used later). Also, as each command is echoed to the screen you may wish to stop this by using the @ prefix.

```
convunit:   converter.c ➥
converter.h
  @echo "Now compiling ➥
converter.c ..."
  gcc converter.c -o convunit
```

Temporary Like Achilles

This makefile can be improved however by building *object* files, and not *executables*. Object files are compiled versions of source code (it can consist of one or more 'C' files), which lack the essential ingredients that make them executable (like access to glibc, and a place to start, for instance!). This not only makes them smaller, but also does not tie them in to any particular executable.

They can be built individually, and then *linked* together with other *object* modules to make one executable. For projects with several source files, this also means that updates can be built with just one compile and one link, which is much more efficient than several compiles, and one link. Object files (by convention) use the .o extension, which is usually pronounced "dot oh!".

Here, we are nesting targets. In this example, *convunit* is dependant on *converter.o* (an object file), which in turn is dependant on the two files *converter.c* and *converter.h*.

Should any of these files change, *convunit* will be re-built. We can place the targets in any order we choose, however, the first target given (*convunit*) is the one built by default and so should be the main executable.

Looking to the bigger picture, we have already split our project into modules (see last month's Linux Magazine issue 24) and have five ready-made targets (core, config, process, output and debug) that map nicely onto five object files. From this we can build a complete makefile for the project.

These last two examples make uses of the '-c' option of GCC, which indicates we want to only build an object file, and not a complete executable.

Our first invocation of *make* will build five object files (the .o files from lines 4,7,10,13 and 16) and one executable (line 1); our second will build none! It will spot that the file *convunit* is newer than all its dependencies (converter.o and config.o process.o output.o debug.o) and report that it is "up to date". Whenever a file changes, only the necessary dependencies will be rebuilt. This is determined by looking at the date stamp of the files in question. You can test this by typing:

```
touch output.h
make
```

This will then build *core.o* and *output.o* (since they are the only targets that depend on *output.h*) and re-link a new executable with the 3 old, and 2 new, object files. It is very rare to include header files like *stdio.h* and *stdlib.h* in the dependencies list.

This is because they are *standard* headers, and changing the function prototypes or macros here would require a change in the *glibc* libraries also. That last happened many years ago with the switch from version 5 to 6, and required a complete recompile of all system and user software.

Showroom Dummies

To ease the task of maintenance, *make* supports macro substitutions which you can use to save re-typing repetitive command line switches. This is especially useful for changing compiler and linker options as one macro can replace everything in one go.

Macros are, by convention, always upper case and defined as a 'name=

Listing 3: Makefile

```
1 convunit: converter.o config.o process.o output.o debug.o
2       gcc converter.o config.o process.o output.o debug.o -o convunit
3
4 converter.o: converter.c converter.h config.h output.h process.h ➥
debug.h
5       gcc -c converter.c -o converter.o
6
7 config.o: config.c converter.h config.h process.h
8       gcc -c config.c -o config.o
9
10 process.o: process.c converter.h process.h
11      gcc -c process.c -o process.o
12
13 output.o: output.c converter.h output.h process.h
14      gcc -c output.c -o output.o
15
16 debug.o: debug.c converter.h debug.h process.h
17      gcc -c debug.c -o debug.o
```

substitution' pair. They are used with the $(NAME) syntax and are substituted automatically before executing any build command. This way, any errors are explained with real commands and parameters, instead of macro names that may be quite complex and obtuse.

```
CC = gcc
CFLAGS = -Wall

converter.o: converter.c ➘
converter.h
  $(CC) $(CFLAGS) converter.c
```

A number of macros exist by default (type "*make -p*" in a shell to find out which) but these can still be changed if necessary. There are also a number of standard macros that you will see, so you should become at least comfortable with them (see tables 1 & 2).

Macros can also be set from the shell, by giving the 'name = substitution' pair as an argument to make.

```
make CFLAGS=-Wall
```

Table 1: Conventional Macros

Macro	Description	Example
CC	Name of the C compiler	GCC
MAKE	The make utility	make
AS	Assembler	as
LD	Linker	ld
FC	Name of the Fortran compiler (really!)	f77

Table 2: Common Macros

Macro	Description
TARGETS	The names of the targets being compiled
SOURCES	Those files to be compiled
LIBS	Directories for other libraries
INC	Directories for other headers files
CFLAGS	Compiler flags
LFLAGS	Linker flags

Table 3: Special Variables

$@	Name of the current target
$$@	As $@, but only available on dependency line
$?	Files that are newer than the target, and so need building
$%	?Member files of library files?
$<	$? for suffix rules
$*	$@ for suffix rules. The files suffix is omitted, however.

Notice that the equals sign is used without spaces as it helps distinguish between a macro definition and target name. For other examples of CFLAGS, see the BOXOUT: Useful compiler flags.

College Girls Are Easy

Another one of *make*'s many features to improve the quality of life are *implied dependencies*. *Make* knows that a C file generates a .o object file, and that it must use *gcc* to do so; the dependency of the .o on the .c is implied and so *make* can perform the compile operation automatically! This allows you to reduce a typical line to:

```
config.o: config.c converter.h ➘
config.h process.h
```

On the surface, it might appear that we have lost the means to use macros and apply special compile flags to gcc. Not so! By using the CFLAGS macro (which is common) we can add warnings, compiler optimisations, or any number of switches we want, and they will get used within the implied dependency.

Notice, however, that line 3 provides an explicit build instruction because *make* doesn't understand that a collection of .o files need to be built into an executable. This is because it can not make the connection between the executable (*convunit*) and the object files. By changing line 2 and calling our ELF 'converter' instead, we can do without line 3.

```
2 converter: converter.o ➘
config.o process.o output.o ➘
debug.o
```

The implied dependencies of an executable (*converter*, in the case above) is its equivalent .o file, and anything else given on the right hand side of the colon. That is – its usual dependencies.

For advanced work, it is possible to create your own implied dependencies; they are called suffix rules.

Time After Time

In addition to macros, there are a number of special variables with a similar appearance to macros, as both start with a '$' symbol. When building make files they can be used to enhance error messages, or to provide parameters to other programs. They also work inside quoted strings.

```
converter.o: converter.c $$
converter.h
```

Box 1: Targets

.SILENT:	Does not echo any command executed. Equivalent to prefixing each command with an @
.IGNORE:	Ignore any errors from the commands. Equivalent to - on each command.
.PRECIOUS	Does not remove the target file being removed after an error.
.DEFAULT	Tries to build this if the given target doesn't exist.
.PHONY	Indicates that these targets do not really compile into programs. Used for cases like 'clean' and 'install', in case there's a file called (say) 'clean' in the current directory that could confuse the situation.

Box 2: Useful compiler flags

-D_DEBUG_FLAGS	Automatically defines the macro '_DEBUG_FLAGS' to the source code.
-g Include	GNU debugging information into the executable. This allows you to use gdb to step through the program one line at a time.
-c	Compile and assemble, but don't link. i.e. create the object file
-o converter	Specify the output file
-Wall	Specifies the warning level. 'All' is best.
-O3	Specify the optimisation level. o is off (debug), 3 is the highest. Using -Os will optimize for space, instead of speed.
-fPIC	Switch specific flag options. Here, PIC tells gcc to produce position independent code (if possible). The option name is case insensitive. Used to produce libraries that would work in more than one place.
-I /usr/local/apache2/include	Also search the named directory for header files. Same the INC common macro.

Note: There is no space between the flags switch and the parameter, except with 'I'.

```
  @echo "Trying to build $@ ➥
(because $? are too new!)"
  $(CC) $(CFLAGS) converter.c
```

For a list of these special variables, please refer to table 3.

Shoot That Poison Arrow

When *make* is run without arguments it will look for the first target in the makefile and try to build it. If the makefile contains more than one project, you should create an extra target named *all*, which is dependent on each of the other targets. This way, every project will get built with a single call to *make*. You can also build a specific target by including it as an argument.

```
make testbed
make config.o
```

Now, most Linux users who build from sources are familiar with the trio of *./configure*, *make*, *make install*. If both the above sentences are true, then 'install' must be the name of a target. Funnily enough, it is! The 'install' target often includes commands to copy configuration and executable files to the appropriate place. These targets, however, are phony – they don't really produce a file – and as such need to be indicated by adding a .PHONY line to the make file (see listing 3 and BOXOUT: Targets)

We can use this knowledge to enhance our *makefile* by adding *clean* and *install*. Notice that in the case of *clean* we ignore all errors, and with *install* we suppress the echo command; and will require superuser privileges. In these cases no dependencies are given, meaning the instructions are executed every time that particular target is called. This produces a complete makefile, ready for use!

There's a guy works down the chip shop?

As time goes on, and projects change, the makefile will become outdated. We'll need to add more targets, change dependencies, or remove old files. Doing this manually can become a bind, so there are a number of tools to help you, such as *mkdepend*, *mkmkf* and *makedepend*. We shall look at this latter.

As the name suggests, *makedepend* will build a list of dependencies for the files specified on its command line. So, assuming all our source files in the same directory (and it contains no rogue files from other projects), we can type:

```
makedepend *.c
```

And a complete list of dependencies (including things like stdio.h, and stdlib.h) will be built, stored in the makefile. And in the correct format!

Makedepend does a couple of clever things here. First off, it makes a back-up of your original makefile and calls it 'Makefile.bak'. Then it appends the dependency information to *Makefile*. What is clever here is that a second call to *makedepend* will not re-append the same data. Even in a small project such as ours, *makedepend* can add 50 or more lines to the makefile. How does it know? Well, it adds a comment marked 'DO NOT DELETE' before the appended text. If this already exists, *makedepend* removes the text below it, and adds the new information.

Naturally, calling *makedepend* without arguments will not find any dependencies and thus produce an empty block at the bottom of the file. This is still useful, as it makes the makefiles small enough to fit in a magazine! And as long as we add the dependencies back to the makefile before trying to compile, all is well!

With the exception of .DEFAULT, each can affect specific targets by including its name as a dependency. If no target is specified, then it will affect all targets within the makefile. ■

Listing 4: Makefile

```
1 # We're now using implied dependencies!
2 convunit: converter.o config.o process.o output.o debug.o
3       gcc converter.o config.o process.o output.o debug.o -o convunit
4 converter.o: converter.c converter.h config.h output.h process.h ➥
debug.h
5 config.o: config.c converter.h config.h process.h
6 process.o: process.c converter.h process.h
7 output.o: output.c converter.h output.h process.h
8 debug.o: debug.c converter.h debug.h process.h
```

Listing 5: Makefile

```
01 CFLAGS -Wall
02
03 converter: converter.o config.o process.o output.o debug.o
04 converter.o: converter.c converter.h config.h output.h process.h ➥
debug.h
05 config.o: config.c converter.h config.h process.h
06 process.o: process.c converter.h process.h
07 output.o: output.c converter.h output.h process.h
08 debug.o: debug.c converter.h debug.h process.h
09
10 clean:
11      -rm *.o converter
12
13 install:
14      @echo "Copying conf file to /etc"
15      cp convert.conf /etc
16
17 .PHONY: clean install
```

THE AUTHOR

The language of 'C' has been brought to you today by Steven Goodwin and the pages 62–65. Steven is a lead programmer, currently finishing off a game for the Nintendo GameCube console. When not working, he can often be found relaxing at London LONIX meetings.