```
================================================================================
```
C-Scene Issue #2
```
Multi-file projects and the GNU Make utility
```
Author: George Foot
Email: george.foot@merton.ox.ac.uk
Occupation: Student at Merton College, Oxford University, England
IRC nick: gfoot
```
================================================================================
```


0) Introduction

This article will explain firstly why, when and how to split your C source code between several files sensibly, and it will then go on to show you how the GNU Make utility can handle all your compilation and linking automatically. Users of other make utilities may still find the information useful, but it may require some adaptation to work on other utilities. If in doubt, try it out, but check the manual first.

1) Multi-file projects

1.1 Why use them?

Firstly, then, why are multi-file projects a good thing? They appear to complicate things no end, requiring header files, extern declarations, and meaning you need to search through more files to find the function you're looking for.

In fact, though, there are strong reasons to split up projects. When you modify a line of your code, the compiler has to recompile everything to create a new executable. However, if your project is in several files and you modify one of them, the object files for the other source files are already on disk, so there's no point in recompiling them. All you need to do is recompile the file that was changed, and relink the object files. In a large project this can mean the difference between a lengthy (several minutes to several hours) rebuild and a ten or twenty second adjustment.

With a little organisation, splitting a project between files can make it much easier to find the piece of code you are looking for. It's simple - you split the code between the files based upon what the code does. Then if you're looking for a routine you know exactly where to find it.

It is much better to create a library from many object files than from a single object file. Whether or not this is a real advantage depends what system you're using, but when gcc/ld links a library into a program at link time it tries not to link in unused code. It can only exclude entire object files from the library at a time, though, so if you reference any symbols from a particular object file of a library the whole object file must be linked in. If the library is very segmented, the resulting executables can be much smaller than they would be if the library consisted of a single object file.

Also, since your program is very modular with the minimum amount of sharing between files there are many other benefits -- bugs are easier to track down, modules can often be reused in another project, and last but not least, other people will find it much easier to understand what your code is doing.

1.2 When to split up your projects

It is obvisouly not sensible to split up *everything*; small programs like `Hello World' can't really be split anyway since there's nothing to split. Splitting up small throwaway test programs is pretty pointless too. In general, though, I split things whenever doing so seems to improve the layout, development and readability of the program. This is in fact true most of the time.

The decision about what to split and how is of course yours; I can only make general suggestions here, which you may or may not choose to follow.

If you are developing a fairly large project, you should think before you start how you are going to implement it, and create several (appropriately named) files initially to hold your code. Of course, don't hesitate to create new files later in development, but if you do then you are changing your mind and should perhaps think about whether some other structural changes would be appropriate.

For medium-sized projects, you can use the above technique of course, or you might be able to just start typing, and split the file up later when it is getting hard to manage. In my experience, though, it is a great deal simpler to start off with a scheme in mind and stick to it or adapt it as the program's needs change during development.

1.3 How to split up projects

Again, this is strictly my opinion; you may (probably will?) prefer to lay things out differently. This is touching on the controversial topic of coding style; what I present here is simply my personal preference (along with reasons for each of these guidelines):

i) Don't make header files which span several source files (exception: library header files). It's much easier to track and usually more efficient if each header file only declares symbols from one source file. Otherwise, changing the structure of one source file (and its header file) may cause more files to be rebuilt that is really necessary.

ii) Where appropriate, do use more than one header file for a source file. It is often useful to seperate function prototypes, type definitions, etc, from the C source file into a header file even when they are not publicly available. Making one header file for public symbols and one for private symbols means that if you change the internals of the file you can recompile it without having to recompile other files that use the public header file.

iii) Don't duplicate information in several header files. If you need to, #include one in the other, but don't write out the same header information twice. The reason for this is that if you change the information in the future you will only need to change it once, rather than hunting for duplicates which would also need modifying.

iv) Make each source file #include all the header files which declare information in the source file. Doing this means that the compiler is more likely to pick out mistakes, where you have declared something differently in the header file to what it is in the source file.

1.4 Notes on common errors

a) Identifier clashes between source files: In C, variables and functions are by default public, so that any C source file may refer to global variables and functions from another C source file. This is true even if the file in question does not have a declaration or prototype for the variable or function. You must, therefore, ensure that the same symbol name is not used in two different files. If you don't do this you will get linker errors and possibly warnings during compilation.

One way of doing this is to prefix public symbols with some string which depends on the source file they appear in. For example, all the routines in gfx.c might begin with the prefix `gfx_'. If you are careful with the way you split up your program, use sensible function names,

and don't go overboard with global variables, this shouldn't be a problem anyway.

To prevent a symbol from being visible from outside the source file it is defined in, prefix its definition with the keyword `static'. This is useful for small functions which are used internally by a file, and won't be needed by any other file.

b) Multiply defined symbols (again): A header file is literally substituted into your C code in place of the #include statement. Consequently, if the header file is #included in more than one source file all the definitions in the header file will occur in both source files. This causes them to be defined more than once, which gives a linker error (see above).

Solution: don't define variables in header files. You only want to declare them in the header file, and define them (once only) in the appropriate C source file, which should #include the header file of course for type checking. The distinction between a declaration and a definition is easy to miss for beginners; a declaration tells the compiler that the named symbol should exist and should have the specified type, but it does not cause the compiler to allocate storage space for it, while a definition does allocate the space. To make a declaration rather than a definition, put the keyword `extern' before the definition.

So, if we have an integer called `counter' which we want to be publicly available, we would define it in a source file (one only) as `int counter;' at top level, and declare it in a header file as `extern int counter;'.

Function prototypes are implicitly extern, so they do not create this problem.

c) Redefinitions, redeclarations, conflicting types: Consider what happens if a C source file #includes both a.h and b.h, and also a.h #includes b.h (which is perfectly sensible; b.h might define some types that a.h needs). Now, the C source file #includes b.h twice. So every #define in b.h occurs twice, every declaration occurs twice (not actually a problem), every typedef occurs twice, etc. In theory, since they are exact duplicates it shouldn't matter, but in practice it is not valid C and you will probably get compiler errors or at least warnings.

The solution to this problem is to ensure that the body of each header file is included only once per source file. This is generally achieved using preprocessor directives. We will #define a macro for each header file, as we enter the header file, and only use the body of the file if the macro is not already defined. In practice it is as simple as putting this at the start of each header file:

```
#ifndef FILENAME_H
#define FILENAME_H

and then putting this at the end of it:

#endif

replacing FILENAME_H with the (capitalised) filename of
the header file, using an underline instead of a dot.
Some people like to put a comment after the #endif to
remind them what it is referring to, e.g.

#endif /* #ifndef FILENAME_H */

Personally I don't do that since it's usually pretty
obvious, but it is a matter of style.

You only need to do this trick to header files that
generate the compiler errors, but it doesn't hurt to do
it to all header files.
```

1.5 Rebuilding a multi-file project

It is important to recognise the distinction here between compiling and linking. A compiler takes C source code and generates some form of object code from that source code, without resolving external references. A linker is then invoked, which takes object file(s) and links them together into an executable file, along with standard libraries and other libraries you may specify. At this stage references in one object file to symbols in another are resolved, and depending on the linker unresolved references may be reported, usually as errors.

The basic procedure, then, is to compile your C source files one by one to object format, and finally link all the object files together, along with any libraries you need. How exactly you do this will depend on your compiler; here I shall describe the commands for gcc, which may also work on your compiler even if it is not gcc.

Note that gcc is a multi-purpose tool which calls other components (preprocessor, compiler, assembler, linker) as required; which of these are called depends upon what the input files are and what switches you give it.

Normally if you pass C source files alone it will preprocess, compile and assemble them one by one, then link to an executable file (usually called a.out) from the resulting objectfiles. This would work in our case, but would destroy many of the benefits of splitting the project up in the first place.

If you pass the -c switch, gcc will compile the listed files to object format only, naming the object files after the C source files, replacing the `.c' or `.cc' suffix

with `.o'. If you pass a list of object files, gcc will simply link them to form an executable, again called a.out by default. You can change the name of the output file from either of these by passing the -o switch followed by a filename.

So, after altering a source file you need to recompile it by calling `gcc -c filename.c' and then relink the project by calling `gcc -o exec_filename *.o'. If you alter a header file, you need to recompile all those source files which #include it; you could type `gcc -c file1.c file2.c file3.c' and then relink, for example.

This is, of course, fairly tedious; luckily there are tools available to simplify this process. The second half of this article describes such a tool: the GNU Make utility.

2) The GNU Make utility

2.1 Basic makefile structure

GNU Make's main action is to read through a text file (a makefile) containing (principly) information about which files (`targets') are created from which other files (`dependencies') and what commands should be executed to do this. Armed with this information, make will then look at the files on disk and, if the timestamp on a target is older than that on at least one of its dependencies, make will issue the commands specified in the hope of bringing the target file up to date.

The makefile is normally called (funnily enough) `makefile' or `Makefile', but you can specify other filenames on make's command line. If you don't, it will look for `makefile' or `Makefile' so it's simplest just to use those names.

A makefile consists (mainly) of a sequence of rules of this form:

```
<target> : <dependency> <dependency> ...
(tab)<command>
(tab)<command>
.
.
.
```

For example, consider the following makefile:

```
=== start of makefile ===
myprog : foo.o bar.o
        gcc foo.o bar.o -o myprog

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

```
bar.o : bar.c bar.h
        gcc -c bar.c -o bar.o
=== end of makefile ===
```

This is a very basic makefile - make starts at the top, and uses the first target, `myprog', as its primary goal (the thing it is ultimately trying to keep up-to-date). The rule tells it that whenever the file `myprog' is older than either `foo.o' or `bar.o', the command on the next line should be executed.

However, before checking the timestamps of foo.o and bar.o it first looks through the makefile for rules with foo.o or bar.o as targets. It finds the rule for foo.o, seeing that it depends on foo.c, foo.h and bar.h. It cannot find additional rules saying how to create any of these files, so it then checks the timestamps on disk. If any of these files are newer than foo.o, the command `gcc -o foo.o foo.c' will be executed, bringing foo.o up to date.

The same check is then made for bar.o, depending upon bar.c and bar.h.

Now make returns to the rule for `myprog'. If either of the other two rules were executed, myprog will need rebuilding (one of the .o files will be newer than `myprog') and so the linking command will be executed.

Hopefully at this stage you can see the benefit of using the make utility to build your programs - all the tedious checking mentioned at the end of the previous chapter is done for you by make, checking the timestamps. A simple change to one of your source files will cause that file to be recompiled (since the .o file depends on the .c file) and then the executable will be relinked (since the .o file has now been modified). The real gain, though, shows if you modify a header file - you no longer need to remember which of your source files depended on it, since the information is all there in the makefile. The make utility will happily recompile any files which are listed as depending on any modified header files, and relink if required.

Of course, this depends on you making sure the rules in the makefile are correct, listing only those header files which are #included in the source file...

2.2 Writing make rules

The obvious (and simplest) way to write your rules is by looking at each source file in turn, adding its object file as a target, and the C source file as a dependency along with all the headers it #includes. However, you should also list as dependencies any other headers which are #included by those headers, and any headers they #include, and so on... it gets difficult to track. So is there an easier way?

Of course there is - ask the compiler! It ought to know what headers it would include when compiling each source file. With gcc you can specify the -M switch, and then gcc will send to stdout a rule for each C file you pass, with the object file as a target and the C file and all headers #included therein as dependencies. Note that this rule will include both headers named between angle brackets (`<', `>') and headers named in inverted commas (`"'); it is often a pretty safe bet that the system header files (like stdio.h, stdlib.h, etc) aren't going to change though. If you pass -MM instead of -M to gcc, it will omit any header files whose names were enclosed with angle brackets.

The rule output by gcc won't have a command part; you can either write in your own command, or just leave it and let make use its implicit rule (see section 2.4).

2.3 Makefile variables

I wrote earlier that makefiles contain mainly rules. Another thing they can contain are variable definitions.

A variable in a makefile is somewhat like an environment variable; indeed, environment variables are translated into make variables during the make process. They are case sensitive, and normally specified in upper case. They can be referenced almost anywhere, and so they can be used for many purposes, for example:

i) Holding lists of files. In the makefile above, the rule to make the executable contains the object filenames as dependencies, and the same filenames are passed to gcc in the command for that rule. If a variable were used in both cases, adding new object files would be simpler and less prone to error.

ii) Holding executable filenames. If your project is taken to a non-gcc system, or if you just want to use a different compiler, you would have to change all calls to the compiler to use the new name. Using a variable instead means that you need only change the name in one place, and all the commands would be updated.

iii) Holding compiler flags. Presumably you want all your compilation commands to pass the same set of options (e.g. -Wall -O -g); if you put the option list in a variable then you can put the variable in all your compiler calls and just change the options in one place whenever you need to.

To set a variable, you simply write its name at the start of a line, followed by an = sign, and then its new value. To reference a variable later on you write a dollar sign, then the variable name in brackets. For example, here is the previous makefile rewritten using variables:

```
=== start of makefile ===
OBJS = foo.o bar.o
CC = gcc
CFLAGS = -Wall -O -g

myprog : $(OBJS)
        $(CC) $(OBJS) -o myprog

foo.o : foo.c foo.h bar.h
        $(CC) $(CFLAGS) -c foo.c -o foo.o

bar.o : bar.c bar.h
        $(CC) $(CFLAGS) -c bar.c -o bar.o
=== end of makefile ===
```

There are also various automatic variables, which are defined for each rule. Three useful ones are $@, $< and $^ (no brackets are needed for these). $@ expands to the filename of the target of the rule, $< expands to the first dependency in the dependency list, and $^ expands to the entire dependency list (with duplicate filenames removed). Using these, then, we could write the above makefile as:

```
=== start of makefile ===
OBJS = foo.o bar.o
CC = gcc
CFLAGS = -Wall -O -g

myprog : $(OBJS)
        $(CC) $^ -o $@

foo.o : foo.c foo.h bar.h
        $(CC) $(CFLAGS) -c $< -o $@

bar.o : bar.c bar.h
        $(CC) $(CFLAGS) -c $< -o $@
=== end of makefile ===
```

There are many other things you can do with variables, especially when you mix them with functions. For further information, see the GNU Make manual.

2.4 Implicit rules

Note that in that last makefile example the commands to create the .o files are identical. This is hardly surprising since they both achieve similar goals - creating a .o file from a .c file and some others is a standard procedure. In fact, make already knows how to do it - it has built in rules called implicit rules which tell it what to do if you don't put any commands in a rule.

If we remove the commands from the rules for creating foo.o and bar.o, make will fall back on its implicit rule database and should find a suitable command. Its command uses several variables, so you can easily customise it to

your tastes; it uses the variable CC to run a compiler (just like we did earlier), passing it the CFLAGS variable for C programs (CXXFLAGS for C++ programs), CPPFLAGS (C preprocessor flags), TARGET_ARCH (don't worry about this), then it puts the flag `-c' followed by the variable $< (first dependency), then the flag `-o' followed by the variable $@ (the target file). The effective command for C compilation is:

```
$(CC) $(CFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -c $< -o $@
```

You can define these variables however you like, of course. This should explain why the output from gcc with the -M or -MM switch is suitable for immediate inclusion in a makefile.

2.5 Phony targets

Suppose you had a project in which two executable files needed to be created. You would want the primary goal to create both files, but independently of each other - if one needs rebuilding, the other may not. To achieve this you can use what is called a phony target. A phony target is just like a normal target, but it is not an actual file on disk. Because of this, make assumes that it needs creating, and always executes any commands in its rule, after bringing its dependencies up to date.

So, if we write at the top of our makefile:

```
all : exec1 exec2
```

where exec1 and exec2 are the filenames of our two target executables, make will set this as its primary goal and try to bring `all' up to date on every invocation. Since there are no commands here which will affect a file called `all' on disk, this rule won't actually change the status of `all' at all. However, since the file does not exist make will check that exec1 and exec2 don't need rebuilding, and rebuild them if they are out of date, which is exactly what we wanted to do.

Phony targets can also be used to describe a set of non-default actions. For example, you might want to remove all the files generated by make. To do this, you could make a rule in the makefile like this:

```
veryclean :
  rm *.o
  rm myprog
```

Provided no rules are listed as depending upon the target `veryclean', this will never be executed. However, if the user types `make veryclean' explicitly, make will use this as its primary goal, and run the rm commands.

What if there is a file on disk called veryclean though? In this case, since this rule has no dependencies, the

target `veryclean' must be up to date, and even if the user explicitly asks make to recreate it nothing will happen. The solution here is to declare all your phony targets as .PHONY, telling make not to bother looking for them on disk, not to bother checking implicit rules, and to always assume that the specified target is not up to date. Adding this line to the makefile containing the above rule:

```
.PHONY : veryclean
```

would do the trick. Note that this is a special make rule, that make knows .PHONY is a special target, and of course you can put more dependencies in if you like and make will know that they are all phony targets.

2.6 Functions

Functions in makefiles are very similar to variables - to use them, you write a dollar sign, an open bracket, and then the name followed by a space and a comma-separated list of arguments, and lastly a closing bracket. For example, there is a function called `wildcard' in GNU Make which takes one argument and expands into a space-separated list of all files matching the specification given. To use it you could write something like

```
SOURCES = $(wildcard *.c)
```

which would create a list of all files ending in `.c' and put it in the SOURCES variable. Of course, you don't have to store the results in variables.

Another useful function is the patsubst function. It takes three parameters - the first is a pattern to match, the second shows what to replace it with, and the third is a space-separated list of words to process. For example, after the variable definition above,

```
OBJS = $(patsubst %.c,%.o,$(SOURCES))
```

would take all the words (filenames) in the SOURCES list and for each, if it ends in `.c', it will replace the `.c' with a `.o'. Note that the % symbol matches one or more characters, and the string it matches each time is called the stem. In the second parameter, the % is read as whatever stem it matched in the first parameter.

2.7 A pretty effective makefile

With the information so far we can write quite an effective makefile, which will be able to do most of our dependency checking for us, and will fit most projects without much modification.

Firstly we need a basic makefile which will build the

program. We can make it search the current directory for source files, and assume that they are all part of the project, by using a variable SOURCES as above. It is probably wise to also include *.cc, in case the compilation is for C++.

```
SOURCES = $(wildcard *.c *.cc)
```

Using patsubst we can then create a list of object files which will be created; if our sources list contains .cc files as well as .c files we'll need to nest calls to patsubst like so:

```
OBJS = $(patsubst %.c,%.o,$(patsubst %.cc,%.o,$(SOURCES)))
```

The innermost patsubst call will replace the .cc files' suffixes only, forming a list which the outermost patsubst processes, replacing the .c files' suffixes.

Now we can form a rule to build the executable:

```
myprog : $(OBJS)
  gcc -o myprog $(OBJS)
```

Further rules may not be necessary; gcc knows already how to create the object files. Next, we can make a rule to create the dependency information:

```
depends : $(SOURCES)
  gcc -M $(SOURCES) > depends
```

This creates a file called `depends' whenever it does not exist or a source file is newer than the existing depends' file, which contains the rules gcc created for the source files. Now we need to get make to consider these rules as part of the makefile. The technique here is rather like the #include system in C - we simply ask make to include this file in the makefile, like so:

```
include depends
```

GNU Make will see this, and check that `depends' is up to date; if it is not, it will recreate it, following the rule we gave. This done, it will include the (new) set of rules and proceed to process the primary goal, `myprog'. On seeing the rule for myprog, it will check all the object files are up to date - using the rules from the `depends' file, which we know is up to date itself.

This system is fairly inefficient, however, since whenever a source file is changed all the source files must be preprocessed again to create the `depends' file, and it isn't 100% safe either since changing a header file will not cause the dependency information to be updated. However, it is quite useful as it stands.

2.8 A more effective makefile

This is a makefile I use for most things I do. It should
build most projects without modification. I have used it
mainly with djgpp, a DOS port of gcc, so the executable
name, `alleg' library, and the RM-F variable reflect this.

```
=== start of makefile ===

#####################################
#                                   #
#          Generic makefile         #
#                                   #
#           by George Foot          #
# email: george.foot@merton.ox.ac.uk #
#                                   #
#   Copyright (c) 1997 George Foot  #
#        All rights reserved.       #
#                                   #
#     No warranty, no liability;    #
#   you use this at your own risk.  #
#                                   #
#     You are free to modify and    #
#   distribute this without giving  #
#   credit to the original author.  #
#                                   #
#####################################

### Customising
#
# Adjust the following if necessary; EXECUTABLE is the target
# executable's filename, and LIBS is a list of libraries to link in
# (e.g. alleg, stdcx, iostr, etc). You can override these on make's
# command line of course, if you prefer to do it that way.

EXECUTABLE := mushroom.exe
LIBS := alleg

# Now alter any implicit rules' variables if you like, e.g.:

CFLAGS := -g -Wall -O3 -m486
CXXFLAGS := $(CFLAGS)

# The next bit checks to see whether rm is in your djgpp bin
# directory; if not it uses del instead, but this can cause (harmless)
# `File not found' error messages. If you are not using DOS at all,
# set the variable to something which will unquestioningly remove
# files.

ifneq ($(wildcard $(DJDIR)/bin/rm.exe),)
RM-F := rm -f
else
RM-F := del
endif

# You shouldn't need to change anything below this point.

SOURCE := $(wildcard *.c) $(wildcard *.cc)
OBJS := $(patsubst %.c,%.o,$(patsubst %.cc,%.o,$(SOURCE)))
DEPS := $(patsubst %.o,%.d,$(OBJS))
MISSING_DEPS := $(filter-out $(wildcard $(DEPS)),$(DEPS))
MISSING_DEPS_SOURCES := $(wildcard $(patsubst %.d,%.c,$(MISSING_DEPS)) \
```

```
                                        $(patsubst %.d,%.cc,$(MISSING_DEPS)))
CPPFLAGS += -MD

.PHONY : everything deps objs clean veryclean rebuild

everything : $(EXECUTABLE)

deps : $(DEPS)

objs : $(OBJS)

clean :
        @$(RM-F) *.o
        @$(RM-F) *.d

veryclean: clean
        @$(RM-F) $(EXECUTABLE)

rebuild: veryclean everything

ifneq ($(MISSING_DEPS),)
$(MISSING_DEPS) :
        @$(RM-F) $(patsubst %.d,%.o,$@)
endif

-include $(DEPS)

$(EXECUTABLE) : $(OBJS)
        gcc -o $(EXECUTABLE) $(OBJS) $(addprefix -l,$(LIBS))

=== end of makefile ===
```

A few things are worth explaining about this. Firstly, I have defined most of my variables using := instead of =. The effect of this is to immediately expand all function and variable references in the definition. With =, the function and variable references are left alone, meaning that changing the value of a variable can change other variables' values. For example:

```
A = foo
B = $(A)
# Now B is $(A) which is `foo'.
A = bar
# Now B is still $(A), but it is now `bar'.
B := $(A)
# B is now `bar'.
A = foo
# B is still `bar'.
```

After a # symbol make ignores any text until the end of the line.

The ifneq...else...endif system is a way of conditionally disabling/enabling parts of a makefile. ifeq takes two parameters. If they are equal, it includes the portion of the makefile up to the else (or endif, if there is no else); if not, it includes the portion between else and endif if the else is present. ifneq is exactly the opposite.

The filter-out function takes two space-separated lists, and expands to the second list with all members of the first list removed. I have used it here to take the DEPS list and remove all members which exist, leaving behind any which are missing.

The CPPFLAGS as I mentioned earlier contains flags to pass to the preprocessor in implicit rules. The -MD switch is like -M, but the information is sent to a file whose name is formed by removing the .c or .cc from the source file and replacing it with a .d (which explains why I form the DEPS variable that way). The files mentioned in DEPS are included in the makefile later on using `-include', which suppresses any errors if the files are not found on disk.

If any dependency files are missing, the makefile will remove the corresponding .o file from disk as well, causing make to rebuild it. Since CPPFLAGS specifies -MD, the .d file will be recreated too.

Lastly, the addprefix function expands to the list given in its second parameter, with its first parameter prepended to each word of the list.

The targets of this makefile (which can be passed on the command line to select them) are:

- everything (default): Update the main executable, also creating or updating a `.d' file and a `.o' file for each source file.
- deps: Just create/update a `.d' file for each source file.
- objs: Create/update the `.d' files and the object files for each source file.
- clean: Delete all the intermediate/dependency files (*.d and *.o).
- veryclean: Do `clean' and also delete the executable.
- rebuild: Do `veryclean' and `everything'; i.e. rebuild from scratch

Of these, clean, veryclean and rebuild are the only really useful ones apart from the default of everything.

I am not aware of any way in which this makefile can fail, given a directory of source files, unless the dependency files have been mangled. If this does occur, simply typing `make clean' should fix the problem by removing all the dependency and object files. It's best not to mess around with them, of course. If you see a way this makefile could fail to do its job, please do let me know so that I can fix it.

3 In conclusion

~~~~~~~~~~~~~~~

I hope this article has explained clearly enough how multi-file projects work, and has shown how to use them in a way which is logical and safe. You should be able to use the GNU Make utility well enough now to manage small projects, and if you understood what was written in the later sections you should not have any trouble with it.

GNU Make is a powerful tool, and although it was designed primarily for building programs in this way it has many other uses. For more information on the utility, its syntax, functions, and other features, you should (as with any GNU tool) consult the info pages about it.

C Scene Official Web Site : http://cscene.oftheinter.net
C Scene Official Email : cscene@mindless.com

~~~~~~~~~~~~~~~