Chapter 7: Directory Organization and Makefile Style

So far we've only discussed the C program itself. This chapter explores the programming environment, which includes organizing your program files, and the *make* utility, which turns source programs into a finished work.

Organizing Your Directories

Small programs consisting of only a few files are easy to organize: just stick everything in one directory. But suppose you're an adventurous programmer and decide to write two programs. Do you stick them both in the same directory? No.

Put each program's files in a separate directory. That way you won't have to figure out which file goes with which program. It also keeps the number of files per directory down to a manageable level

Rule 7-1:

Whenever possible, put all the files for one program or library in one directory.

Someday you will probably work on a series of programs, like a set of programs to manage a mailing list. There are programs to enter data, check for duplicates, print labels, and generate reports. All of these programs use a common set of low level list functions. You can't put each of these functions in each program directory. Duplicate files are very difficult to maintain. You need some way to share files.

The solution is to turn the list functions into a library. The library goes in one subdirectory while other subdirectories hold the various programs.

Suppose you have all four programs all going to the Library directory for their subroutines. But the Library directory contains both the source and the library file (*MAIL.LIB)* and headers (*MAIL.H*) used by these programs. Having access to all that data can easily confuse things. You need to limit what they can see.

The solution is to have a special directory for libraries and header files as illustrated by Figure 4-2. When a library is built it is “released” by placing it in this directory. The header files are put here as well. This directory contains the public part of the library, while the private part stays behind in the source directory.

The top level directory (*Mailing List*) should be kept free of sources, except for a *Makefile, READ.ME,* or other compilation. This makes the files in the top level simple. Adding programs to this level adds unneeded complexity.

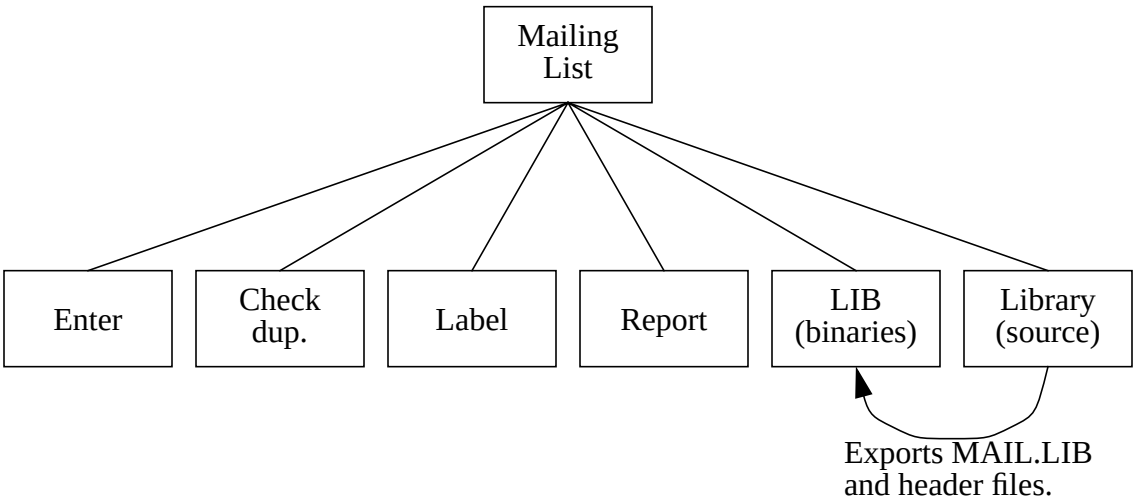

Figure 4-2: Mailing list directory tree

The make Program

Almost all C compilers come with a program building utility called *make.* It is designed to perform the compilation and other commands necessary to turn source files into a program.

To use *make,* you provide it with a description of your program in a file named *Makefile.* This file also contains the transformation rules that tell it how to turn source into objects.

The *Makefile is* divided into six major sections:

1. Heading comments
2. Macro definitions
3. Major targets
4. Other targets
5. Special compilation rules
6. Dependencies

Heading Comments

The first things a programmer needs to know when confronting a strange *Makefile* are “What does it do?” and “How do I use it?” The heading comments of your *Makefile* should answer those questions.

The first paragraph of the heading comments explains what the *Makefile* creates. For example:

```
################################################
# Makefile for crating the program "hello"     #
################################################

################################################
# This Makefile creates the math libraries:    #
#       fft.a, curve.a and graph.a             #
################################################
```

Customization Information

Programmers use the preprocessor **#ifdef** to allow for compile time configuration of their program. For example, there might be a debug version and a production version. Or there might be several different flavors, one for each of the various operating systems the program is designed to run on.

The effect of these conditional compilation directives filter up to the *Makefile*. For example, defining the macro `CFLAGS` as `-DDEBUG` may produce a test program, while the definition `-DPRODUCTION` may produce the production version.

Any configuration information should be listed in the heading comments. This way a programmer can immediately see what customization must be performed on the *Makefile* before he starts to build the program.

For example:

```
#
# Set the variable SYSTEM to the appropriate value for your
# operating system.
#

#    SYSTEM=-DBSD4_3     For Berkeley UNIX Ver. 4.3
#    SYSTEM=-DSYSV       For AT&T System V UNIX
#    SYSTEM=-DSCO        For SCO UNIX
#    SYSTEM=-DDOS        For DOS (Borland Turbo C)
#
```

Standard targets

The standard form of the *make* command is:

```
make target
```

Here, *target* selects what you want to make. (Microsoft's *make is* a notable exception to this standard.) The programmer needs to know which targets are valid and what they do. Putting a list of targets in the heading provides this information.

For example:

```
################################################
# Target are:                                  #
#    all - create the program "hello"          #
#    clean - remove all object files           #
#    clobber - same as clean                   #
#    install - put the program in $(DESTDIR)   #
################################################
```

Over the years a standard set of four targets have developed:

all This target compiles all the programs. This is the standard default target.

install This target copies the program and associated files into the installation directory. For local commands this can be */usr/local/bin.* For production software, this will be the official release directory.

clean This target removes all program binaries and object files and generally cleans up the directory.

clobber Like clean, this target removes all derived files—that is, all files that can be produced from another source. In cases where a software control system such as *SCCS* or *RCS* is used, it means removing all source files from the software control system.

lint (UNIX systems)

This target runs the source files through the program checker

This list represents the minimum “standard” set of targets. Other optional target names have also come into use over the years.

depend or *maketd*

Creates a list of dependencies automatically and edits them into the *Makefile.* There are several utilities to do this, including a public domain program called *maketd.*

srcs Checks out the sources from a software control system such as *SCCS* or *RCS.*

print Prints the sources on the line printer.

xrf Creates a cross reference printout.

debug Compiles the program with the debug flag enabled.

shar Makes a char format archive. This format is widely used to distribute sources over the Internet and USENET.

Macro Definitions

The *make* utility allows the user to define simple text macros, such as:

```
SAMPLE=sample.c
```

The macros are used to define a variety of items, such as the source files to be compiled, the compiler name, compilation flags, and other items.

The *make* program predefines a number of macros. (The actual list of Redefined macros varies, so check your manual to see which macros are defined for your version of *make.)*

Each macro definition should be preceded by a short, one-line comment that explains the macro. Also use white space to separate each comment/macro combination.

For example:

```
# The standard C compiler
CC = cc

# Compile with debug enabled
CFLAGS = -g

# The source to our program
SOURCE = hello.c

# The object file
OBJECT = hello.o
```

Common macro definitions

There are no standard macro definitions; however, the following is a list of the most common:

CC The C compiler

CFLAGS Flags supplied to the C compiler for compiling a single module.

LDFLAGS Flags supplied to the C compiler for loading all the objects into a single program.

SRCS or *SOURCES*

The list of source files.

OBJS or *OBJECTS*

The list of object files. Some of the newer versions of *make* have an extension that allows you to automatically generate this macro from the *SRCS* macro. For example, the following line tells Sun's *make* that *OBJS* is the same as *SRCS,* except change all the *.c* extensions to *.o*.

```
0BJS = $(SRCS:.c=.o)
```

HDRS or *HEADER*

The list of header files.

DESTDIR The destination directory, where the *install* target puts the files.

Configurable variables

As mentioned earlier, macros are frequently used for configuration information. When it comes to actually defining the variable, it is useful to list all the definitions and then comment all but the selected one. For example:

```
# Define one of the following for your system
SYSTEM=-DBSD4 3     # For Berkeley UNIX Version 4.3
#SYSTEM=-DSYSV      # For AT&T System V UNIX
#SYSTEM=-DSCO       # For SCO UNIX
#SYSTEM=-DDOS       # For DOS (Borland Turbo C)
```

Major Targets

So far, we've just been defining things. At this point it's time to tell *make* to actually do something. This section contains the rules for all the major targets listed in the comment header. These targets are grouped just after the macros so they can be easily located.

For example:

```
all: hello

install: hello
         install -c hello /usr/local/bin

clean:
    rm -f hello.o

clobber: clean
```

Other Targets

Often a *Makefile* contains several intermediate or minor targets. These are to help build things for the major targets. For example, the major target *all* calls upon the minor target *hello.*

Minor targets follow the major ones.

Example:

```
hello: $(OBJECTS)
        $(CC) $(CFLAGS) -o hello $(OBJECTS)
```

Special Rules

The *make* program knows about all or most standard compilers, such as the C compiler. Sometimes you need to define a rule for a special compiler, such as the parser generator *yacc*. This program takes grammars (*y* files) and turns them to C code

The *Makefile* rule for this program is:

```
#
# Use yacc to turn xxx.y into xxx.c
#
.y.c:
        yacc $*.y
        mv yacc.xx.cc $*.c
```

Notice that every special rule has a comment explaining what it does.

This target section can also be used to override the default rules. For example, if all your C files need to run through a special pre-processor, you can install your own rule for C compilation:

```
#
# Run the files through "fixup" before compiling them
#
.c.o:
        fixup $*.c
        $(CC) $(CFLAGS) -c $*.c
```

Some *make* programs provide you with a default rule file. **Under no circumstances should you change this file.** Doing so changes causes *make* to behave in a nonstandard way. Also, programmers expect the complete compilation instructions to be kept in the program's *Makefile,* not hidden in some system file.

Dependencies

The dependencies section shows the relationship between each of the binary files and their source. For example:

```
hello.o: hello.c banner.h
```

tells *make* that hello.o is created from *hello.c* and *banner.h.*

Dependency checking is the weakest point in the *make* command. Frequently this section is out of date or missing entirely. Advanced *make* programs have an automatic dependency checking, thus eliminating the need for this section.

Other solutions have also sprung up. The public domain utility *maketd* and other similar programs automatically generate dependency lists. They all depend on this section being at the end of the *Makefile.*

Example

The full *Makefile* for the *hello* program is:

```
################################################
# Makefile for creating the program "hello"    #
# Set the variable SYSTEM to the appropriate   #
# value for your operating system.             #
#                                              #
# SYSTEM=-DBSD4_3 For Berkeley UNIX Version 4.3 #
# SYSTEM=-DSYSV       For AT&T System V UNIX   #
# SYSTEM--DSCO        For SCO UNIX             #
# SYSTEM=-DDOS        For DOS (Borland Turbo C) #
#                                              #
# Targets are:                                 #
#       all - create the program Hello         #
#       clean - remove all object files        #
#       clobber - same as clean                #
#       install - put the program in           #
#                         /usr/local/bin       #
################################################

#
# Macro definitions
#

# The standard C compiler
CC = cc

# Compile with debug enabled
CFLAGS = -g

# The source to our program
SOURCE = hello.c

# The object file
OBJECT = hello.o

# Define one of the following for your system

SYSTEM=-DBSD4_3         # For Berkeley UNIX Version 4.3
#SYSTEM=-DSYSV          # For AT&T System V UNIX
#SYSTEM=-DSCO           # For SCO UNIX
#SYSTEM=-DDOS           # For DOS (Borland Turbo C)
```

```
# Compile with debug enabled
CFLAGS = -g $(SYSTEM)

#
# Major targets
#

all: hello

install: hello
        install -c hello /usr/local/bin

clean:
        rm -f hello.o

clobber: clean

#
# Minor targets
#
hello: $(OBJECTS)
        $(CC) $(CFLAGS) -o hello $(OBJECTS)
#
# No special rules
#

#
# Dependencies
#
hello.o: hello.c banner.h
```

Common Expressions

Whenever possible, use macros for common directories or other text. For example:

```
#
# Poor practice
#
INSTALL_BIN = /usr/local/bin    # Place to put the binaries
INSTALL_MAN = /usr/local/man    # Place to put the man pages
INSTALL_HELP = /usr/local/lib   # Place to put help info.
#
# Better practice
#
DESTDIR=/usr/local
INSTALL_BIN = $(DESTDIR)/bin # Place to put the binaries
INSTALL_MAN = $(DESTDIR)/man    # Place to put the man pages
INSTALL_HELP = $(DESTDIR)/lib   # Place to put help info.
```

and

```
#
# Poor practice
#

# Yacc switches
YACC_FLAGS = -c -t -I/project/include -I/general/include
# C switches
CFLAGS = -c -g -I/project/include -I/general/include

#
# Good practice
#
INCLUDES=-I/project/include -I/general/include

# Yacc switches
YACC_FLAGS = -c -t $(INCLUDES)

# C switches
CFLAGS = -c -g $(INCLUDES)
```

Complexity

Installing a program can be tricky. I've seen a shell script with more than 100 lines created just to install a single program. There is a temptation to put long, complex command sets into the *Makefile.* Because of the difficulties of both shell program and *Makefile* format, this results in a large, complex, and impossible to maintain piece of code.

In general, it is best to put large command scripts in a batch file. This makes it easier to test, debug, and comment them.

Portability Considerations

*Makefile*s have a standard format that is portable across most systems. However, compile time options differ from system to system. For example, a program written to work on both UNIX and DOS will require two entirely different commands sets to create it. Stuffing two sets of compilation instructions in a single *Makefile* can get messy. When this happens, it is best to create a separate *Makefile* for each system. The standard method for naming these various *Makefiles* is *<system>.mak.* Some standard names are:

bsd.mak	BSD4.3 UNIX Makefile
att.mak	AT&T System V
sun.mak	SUNOS UNIX system
turboc.mak	DOS using Borland's Turbo C
msc.mak	DOS using Microsoft's C compiler
sco.mak	SCO UNIX

This list can grow quite long as programs are ported to more and more systems. A *read.me* file must be distributed with the software to describe how to select the proper *Makefile.*

Generic Makefiles

Some of the more advanced *make* commands have an include facility that allows the inclusion of other files in the *Makefile.* Some programmers have tried to create generic *Makefile*s, to be used like this:

```
#
# Define some macro names to be
# used by the generic Makefile
#
SRCS=hello.c
OBJS=hello.o
PROGRAM=hello

include(Makefile.generic)
```

In theory, this should work nicely. There is one generic *Makefile* that does everything, then all you have to do is set things up properly.

In practice, though, it's not so simple. Creating a program is never a standard process and far too many have their little peculiarities. Trying to program around them in a generic *Makefile* is extremely tricky.

One approach is to create a generic *Makefile* to be used as a template for making custom *Makefiles* The problem with this approach is that when you want to add a new target to every

Makefile. you must edit each one.

The solution? There isn't one. This is a classic trade-off of standardization vs. flexibility. Generic *Makefile*s are standard but inflexible. Individual *Makefile*s are flexible but hard to standardize.

Conclusion

*Makefile*s are as important to the programming process as the program itself. A well designed *Makefile* makes it easy to create a program. Comments are necessary to tell programmers the vital information that lets them create future versions of your program.