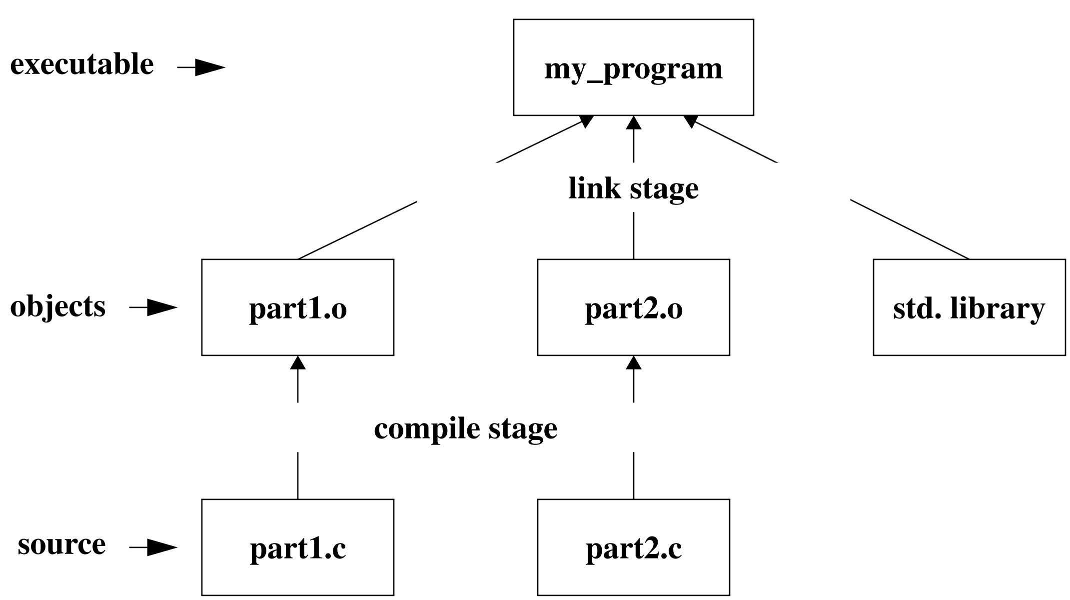
Using *Make* to Maintain Software

by John Locke
(jxxl@cs.nps.navy.mil)
Computer Science Department, Naval Postgraduate School

1. What *Make* Does 1
2. The *Makefile* 2
   2.1 Dependencies 2
   2.2 Variables 3
3. Debugging a *Makefile* 4
4. Using Header Files 5
5. Using Libraries 5
   5.1 A General Discussion 5
   5.2 Using Libraries in *Makefiles* 6
6. Using the *Makefile* for Other Operations 6
7. Useful Features 7
8. Efficient *Makefiles* 8

1. What *Make* Does

With any reasonably complicated program, we attempt to organize source code as an aid to development and future maintenance. This follows the principles of Software Engineering and also common sense. Organization entails subdividing the source into compact procedures with descriptive names; it often entails spreading the source through separate files. If an application consists of multiple programs then the source *must* be in at least as many files as there are programs. At any rate, there tends to be more source files than programs. Building an executable binary (or binaries) from multiple source files consists of compiling each source into an object file of machine code. The objects are then linked into the binary, with a standard system library of pre-compiled code. The following diagram illustrates the basic process:



Note that the general relationship between files in the diagram remains the same throughout program development. *Make* is a utility that allows us to easily express this relationship in order to streamline program development. It solves these basic problems:

1. An application may consist of multiple sources and programs making it difficult to remember how they fit together.
2. It takes too much time to recompile every source file to rebuild a binary if only some have changed.
3. After working on a number of files it is difficult to remember which sources need recompilation, particularly since a source may need recompilation because of another file (e.g. a header) changing.
4. A compilation command can be long, involving a number of options, making it difficult to remember how to compile even a single source.

We encode the relationship between files in a *makefile* which *make* follows to rebuild the program, compiling only those sources that have changed since the last build.

2. The *Makefile*

M*ake*'s guide is a text file the programmer writes named *Makefile* or *makefile*[1]. *Make* automatically searches the current directory for a file with that name. The *makefile* has two sections, variables and dependencies. The variables generally come first in the file, but since they are optional, they will be discussed second.

2.1 Dependencies

The dependencies section is where the relationship between files is spelled out. Note that the previous illustration shows an inverted tree structure. This *dependency tree* is what is encoded in the *makefile* (though not visually). This is how the tree of the illustration might look in a *makefile* (line numbers added for reference):

```
my_program: part1.o part2.o
        cc -o my_program part1.o part2.o

part1.o: part1.c
        cc -c part1.c

part2.o: part2.c
        cc -c part2.c
```

Down the left column are a series of *labels*. They represent nodes on the dependency tree and end in a colon. Labels are usually file names. Following the label is a series of dependencies, that is, files that the file in the label is built from. Dependent files have their own labels. The indented lines below the labels are the *action(s)* to be taken when all depen-

1. Actually, it can be named anything if *the* "-f" flag is used, e.g. *make -f myfilename.*

dencies have been followed down the tree. *(The indentation MUST be a tab, not the equivalent number of spaces![2])*

What governs movement down the tree? What satisfies the dependent relationship? The answer is *file creation time*. If the dependent files are newer than the file that depends on them, then they have changed since the last build and the dependency is followed.

Let's step through the above example using the line numbers. Line 1: the executable *my_program* depends on objects *part1.o* and *part2.o.* Is *part1.o* newer than *my_program*? If so, the executable is out of date, jump to the label for *part1.o*. Line 4: the object *part1.o* depends on the source file *part1.c.* Is *part1.c* newer than *part1.o*? If so, check for a dependency for *part1.c.* There is none, of course, the leaf node has been reached, so perform the action to bring *part1.o* up-to-date (line 5). The source is compiled to an object, as specified by the "-c" flag.

Jump back to line 1 for the next dependency. Is *part2.o* newer than *my_program*? If so, jump to the label for *part2.o* (line 7). Is *part2.c* newer than *part2.o*? If so, perform the action to bring *part2.o* up-to-date (line 8).

Jump back to line 1. All dependencies have been followed, so perform the action to bring *my_program* up-to-date (line 2). The objects are linked and the resulting executable is named *my_program*, as specified by the "-o" (output) flag. (The illustration shows the standard library being linked with the objects. This does not need to be specified in the *makefile* since the linker does it automatically.)

2.2 Variables

Here is the same basic *makefile* with some variables added:

```
# sample makefile

CC = cc
OFILES = part1.o part2.o
CFLAGS = -g
#CFLAGS = -O
HDRS = constants.h globals.h

my_program: $(OFILES)
	$(CC) $(CFLAGS) -o my_program $(OFILES)

part1.o: part1.c $(HDRS)
	$(CC) $(CFLAGS) -c part1.c

part2.o: part2.c $(HDRS)
	$(CC) $(CFLAGS) -c part2.c
```

2. Warning: copying text from one window to another with the mouse converts tabs to spaces!

Line 1 starts with a '#' and is therefore a comment. Comments can be added anywhere in the *makefile*. The variables are on lines 3-7. Note that the variables all come *before* the dependency tree (lines 9-16).

Variables can have any name. A variable VARNAME is referenced with the expression $(VARNAME). Where the variable is used, a simple substitution is made between $(VARNAME) and the text the variable is set to.

The main reason for variables is to allow a single substitution to be made throughout a *makefile*. For instance, in line 3 we set the variable CC to "cc". On the action lines where we specify a compiler (10,13,16), $(CC) gets replaced with "cc", the standard C compiler. If at some time we wanted to switch to GNU's C compiler, for instance, only one change to the *makefile* would be required, and that would be to set "CC = gcc". To switch to the C++ compiler, we would use "CC = CC".

On line 4, OFILES is set to the list of object files that make up *my_program*. This is useful because this list is used twice, on lines 9 and 10. Using the OFILES variable means we will only have to maintain one list as new objects, e.g. *part3.o*, are added to the build.

On line 5 is CFLAGS, a common variable name for the miscellaneous options passed to the compiler. In this example it is set to "-g", the option that causes symbol tables to be included in the objects for the benefit of the *dbx* debugger. Note that line 6 also has a CFLAGS, but commented out. It is set to "-O", a request for the optimizing features of the compiler to be invoked. Why have separate CFLAGS? The "-g" and "-O" flags are incompatible--we cannot compile for both optimization and symbolic debugging. When debugging is finished and a faster executable is desired, we simply switch the comment symbol from line 6 to 5 and recompile the entire program.

On line 7 HDRS is set to a list of header files. Again, this reduces to one the list of headers to be maintained. A secondary point is that an object can be rendered out-of-date without its source file changing. Often, changing a header file demands recompilation of files you may have forgotten uses that header. Building the dependencies into the *makefile* as the program is developed will reduce potential programming problems.

3. Debugging a *Makefile*

From observing the previous sample *makefile*, it should be clear that using variables renders the *makefile* harder to read. Additionally, setting out a dependency tree linearly in a text file obscures the shape of the tree and the sequence actions will be performed in. In fact, there is no required order for the dependencies. On top of that, for any given invocation *make* may only process a subset of the actions since not all source will require recompilation. So what will *make* actually do if we run it on a particular *makefile* at a particular point in time? This question is answered with *make*'s "-n" option. Invoking the option causes *make* to print a list of the actions it *would* take, in the order it would take them, with all variable substitutions made. Using the previous sample *makefile*, here is the result:

```
% make -n
cc -g -c part1.c
cc -g -c part2.c
cc -g -o my_program part1.o part2.o
```

Additionally, the "-d" flag will print out reasons for making the commands as well as the commands, e.g. *make -d -n.*

4. Using Header Files

In a C program there are two ways to include a header file, with angle braces or with quotation marks:

```
#include <stdio.h>
#include "globals.h"
```

Braces indicate that the file can be found in a directory on the *header search path* which, by default, is /usr/include. Header files in quotation marks must be specified by a complete path--the example implies that *globals.h* is in the current directory.

If a number of header files have been created for a program or are used by separate programs, it may be convenient to store them in a separate directory and to add that directory to the header search path. Then instead of including the file with a line like:

```
#include "/work/my_acct/my_prog/h/my_globals.h"
```

We could use:

```
#include <my_globals.h>
```

This is easily facilitated by adding a variable to the *makefile*:

```
INCDIR = -I/work/my_acct/my_prog/h
```

In turn, the variable must be used in the action for relevant compilations:

```
$(CC) $(CFLAGS) -c part1.c $(INCDIR)
```

5. Using Libraries

5.1 A General Discussion

Libraries are special object files of functions grouped according to purpose. A program is automatically linked to the standard library giving access to system calls and other common functions. Other libraries, like the math library or the X Window library, must be specified in the link level action. Calling a library routine--*sqrt*(), for example--without linking in the appropriate library results in this kind of error:

```
% cc my_prog.c
ld: Undefined symbol
  _sqrt
```

Sqrt() is in the math library, so this is the proper command to link it in:

```
% cc my_prog.c -lm
```

The "-l" option indicates a library, with a format of "-l*libname*". Library files have a standard filename format of *lib[libname].a*. The math library is therefore named *libm.a.* By default, it and other system libraries are found in /usr/lib.

In general, compiler options can go in any order as long as matched pairs, like "-o my_prog", are kept together. However, some systems *require* the library option to come last. On those systems, no error message will be produced by not listing it last, but the program will fail for seemingly inexplicable reasons. For this reason, *always put the library option last!*

Also, *libraries only get linked into a program once!* Do not link them to object files. This is another case where no error messages are produced but strange errors can result. Here is an example of what *not* to do:

```
cc -c part1.c -lm
cc -c part2.c -lm
cc -o my_program part1.o part2.o -lm     (correct)
```

Only the last link of the library is correct.

5.2 Using Libraries in *Makefiles*

Two common *makefile* variables for libraries are:

```
LIBS = -lm -lX11 -lmy_lib
MYLIBDIR = -L.
```

Putting the library options in LIBS cleans up the compile line in the *makefile*. Setting MYLIBDIR with a "-L" option puts another directory argument on the *library search path*. (The default path is /usr/lib.) In this case, the directory, ".", represents the current directory. Used in conjunction with "-lmy_lib", the linker will look for *libmy_lib.a* in the current directory. Here is what a complete command might look like:

```
$(CC) $(CFLAGS) -o my_prog $(OFILES) $(MYLIBDIR) $(LIBS)
```

6. Using the *Makefile* for Other Operations

By default, executing *make* sets the first colon-terminated label (*label:*) in the *makefile* to the root of a dependency tree. But the first label doesn't have to be the root; any label can be the root. In fact, there can be any number of roots; additionally, any tree node can be considered the root of the portion of the tree below it. This can be accomplished by specifying a root label when *make* is executed, e.g. *make label*. It might be desirable to invoke *make* for a subset of the tree:

```
% make part1.o
```

It might also be desirable to build a *makefile* for a set of related programs:

```
all: prog1 prog2

prog1: $(PROG1OBJS)
         $(CC) $(CFLAGS) -o prog1 $(PROG1OBJS)

prog2: $(PROG2OBJS)
         $(CC) $(CFLAGS) -o prog2 $(PROG2OBJS)
```

This example provides the flexibility of building either a single program, e.g. *make prog2*, or every program with either *make* or *make all*.

Makefiles commonly include utilities for program maintenance other than program building:

```
clean:
         cp my_prog my_prog.backup
         rm -f *.o *.a my_prog core
```

In this example, *make clean* makes a backup file of the executable, then deletes all files other than the source, useful as a prelude to a complete rebuild of the software. (Note that since no dependencies are listed for *clean,* the actions are always performed.)

7. Useful Features

Naturally, there are numerous bells & whistles in *make*. Here are a few examples:

1. Line continuation - following the UNIX standard for configuration files, lines can be continued with a backslash:

```
OFILES = part1.o part2.o part3.o part4.o part5.o \
   part6.o
```

2. Nested variables - variables may be used in other variables:

```
HDRPATH = ..
INCLUDES = -I$(HDRPATH)/other_app/h \
           -I$(HDRPATH)/other_app/h/net \
           -I$(HDRPATH)/other_app/h/win
CFLAGS = -O $(INCLUDES)
```

3. Setting defined constants - *#define*'s are used in C programs to allow conditional compilation:

```
#define DEBUG
```

```
#ifdef DEBUG
        printf("Checkpoint 1\n");
#endif
```

The weakness here is that DEBUG is set in the source code. It can also be set in the *makefile*, allowing easy toggling with '#' without having to edit source:

```
DEFINES = -DDEBUG
CFLAGS = -g $(DEFINES)

prog:
        cc $(CFLAGS) prog.c
```

Another technique, which requires no editing of the *makefile*, sets the variable on the command line (and overrides that variable if present in the *makefile*):

```
% make "DEFINES = -DDEBUG"
```

4. *Make* Output - *makefiles* can be quite extensive, particularly ones that come with public domain software packages which typically consist of uncompiled source. *Make* prints its actions as it executes them but that can be more than a screenfull. To preserve *make*'s output, simply redirect to a file:

```
% make > make_result &
```

Also, preceding a *makefile* action with a '@' will suppress it from echoing. In this example, only the second action will print:

```
prog: $(OFILES)
        @ cp prog prog.old
        cc -o prog $(OFILES)
```

8. Efficient *Makefiles*

The example *makefiles* shown thus far completely spell out the operations that *Make* will execute. But *make* actually has enough knowledge to make correct assumptions about the program-building process, allowing minimal *makefiles* and, more important, low maintenance. Here is a stripped-down *makefile* for a simple source configuration and the result of executing it:

```
% ls
makefile   part1.c     part2.c

% cat makefile
OBJS = part1.o part2.o

my_prog: $(OBJS)
        cc -o my_prog $(OBJS)

% make -n
cc -c part1.c
```

```
cc -c part2.c
cc -o my_prog part1.o part2.o
```

The key to the *makefile* is that *make*, by default, expects an object file "*filename*.o" to be made by compiling "*filename*.c". For any .o file listed as a dependency, *make* generates a *cc -c* for its source. Therefore, this *makefile* can be easily maintained for additional source files by adding their object names to OBJS. *Make* also recognizes standard variable names like CC[3]:

```
% cat makefile
OBJS = part1.o part2.o

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

% make -n
cc -c part1.c
cc -c part2.c
cc -o my_prog part1.o part2.o
```

But a simple *makefile* like this is also inflexible in the treatment of the sources--it's a one-size-fits-all solution. If individual sources require special treatment, like different sources being dependent on different headers, the dependencies and commands have to be spelled out.

3. Execute *make -p* to print all internal rules and variables.