A Make Primer

[adapted from http://www.cs.niu.edu/~abyrnes/csci241/make.htm]

In any programming project, even a small one, there is a considerable amount of typing involved in building a final executable file from a collection of source code files: Each source code file must be compiled into an object file. The object files must be linked with system libraries into an executable file. The compilation commands may have a considerable number of options. The linking command may contain several system libraries. And if changes are made to any source code file, which files need to recompiled? Or should all of them be recompiled and linked? The opportunities for mistakes are great, especially during peak programming hours (10:00 p.m. - 5:00 a.m.)

The `make` program was designed to build and maintain programming projects.

Options

Like most UNIX commands, `make` has numerous options. The most useful of these are

`-f` filename
: Uses the filename specified as the makefile containing the rules to be used. By default, `make` searches for a file of rules named `makefile` or `Makefile`.

`-k`
: Keep going. By default, `make` stops building a project as soon as an error occurs. The use of this option forces `make` to attempt building other pieces of the project as well. This enables the person building the project to collect a large list of errors for debugging purposes, rather than being limited to only one error at a time.

target
: By giving on the command line the name of a target in the makefile, only the rules necessary to make that target and its dependencies are executed. This is useful for project maintenance. Pseudo-targets (see below) that represent common sequences of actions can be created. For example, by creating a pseudo-target named `clean` which removes all object files and executables, the project can be quickly cleaned up by typing `make clean` at the command prompt.

Makefile Rules

A makefile consists largely of rules, which specify how various intermediate files in the projects are constructed. Rules are separated by blank lines. Makefile rules consist of three parts --- a target list, a dependency list and a list of UNIX commands:

```
        target list:  dependency list
              command 1
              command 2
              command 3
                 .
```

```
                    .
                    .
```

Targets

The target in a makefile rule is usually the name of a file that is to be made as part of the project. This is most commonly an executable file or an object code file. But it doesn't have to be a file (see Pseudo-Targets, below). The target must be separated from the dependency list with a colon. The target name should start in the first column position on the line.

Dependency Lists

Dependency lists are lists of files which must all exist and be **up-to-date** in order to create the target. By up-to-date we mean that the modify date on the file must be later than the modify date on all the files needed to build that file, if there are any.

The files in the dependency list must be separated by spaces and placed on one line. If the line becomes too long for the editor (a very real possibility for the dependency lists in a large project), then the line may be logically extended by typing a `\` **immediately** followed by a new line. Anything on the following line is considered to be logically part of the line extended with the `\`. If the target is an executable file the dependency list is probably a list of object code files. For example

```
raytrace:  trace.o light.o input.o sphere.o polygon.o ray.o \
               bound.o triangle.o quad.o
       (a command goes here to build the executable)
```

Other common dependency lists are made of source code and header files if the target is an object code file.

```
trace.o:  trace.cc trace.h rt_attrib.h light.h jitter.h
       (a command goes here to make the object file)
```

Commands

The commands in a makefile rule is simply a set of UNIX commands that would normally be typed in on the command line to create the target file from the files in the dependency list:

```
pxform: pxform.o similar.o affine.o
     g++ -o pxform pxform.o similar.o affine.o -lm
```

An **extremely important point** about commands in makefile rules is that they **must be indented with a tab character**. They must **not** be indented with any spaces. To use spaces can produce odd errors that are hard to track down.

Astoundingly this has been a severe design flaw in the make program for decades! By visual inspection it is impossible to tell if spaces or tabs are being used. The only way to tell is by using an editor and moving the cursor around or by using a global substitution command to reveal the location of stray blanks.

A command in a makefile rule is run only if the target is out of date with respect to the dependencies. This is determined by examining the timestamps on the target and the dependencies. If any of the dependencies are newer than the target then the target is regenerated (hopefully) by executing the commands. As part of checking the dependencies of any given rule, the make program will verify that the dependency is not the target of some other rule. If it is, then that new rule is evaluated first. Only when all of the dependencies are known to be up-to-date is the comparison made with the target of the current rule.

Rules about Rules

Some other important points about makefile rules:

1. The rules are not steps in a program. They are not run from top to bottom. They are a collection and may be run in any order according to the dependencies that need to be satisfied. The entire collection of rules is analyzed along with the timestamps of the files involved to determine the specific order to run the commands in.
2. If no target is specified when `make` is invoked, then the target of the first rule listed in the file is assumed. This usually means that the rule that makes the project executable(s) from the object code files is listed first. If the project produces only a single executable then the rule that creates the executable is used. If the project creates multiple executables, then a pseudo-target that depends on all of the executables can be used. (See below)
3. Just because it's a common source of errors for beginning makefile users, it is repeated here: **Always use tabs to indent makefile rule commands instead of spaces.**

Pseudo-Targets

The targets of makefile rules need not correspond to actual files on the system. These are called *pseudo targets* and can be very useful for maintaining the project. One use of rules with pseudo targets is to create complex commands that can be easily invoked. These rules have empty dependency lists and are invoked by giving the target name to `make` when it is called.

For example, a common and useful rule in any project has the pseudo target of "`clean`".

```
clean:
	-rm *.o my_executable_files_here
```

This rule, when invoked, will remove all of the object files and any executable files that are listed as part of the command. This rule is invoked by typing `make clean` at the command line prompt.

Many helper rules such as `all` for rebuild everything and `tar` for archiving the entire project can be added. By convention, rules of this form are placed towards the end of the makefile, after the other rules that actually build the project.

Another thing to note in the above example is the hyphen in front of the `rm` command. When placed immediately in front of the command to be executed, a hyphen will cause the make program to ignore any errors associated with the command. For example,

```
% make clean
rm *.o

....   A file editing session takes place here


% make clean
rm: cannot remove *.o : No such file or directory
make: [clean] Error 1 (ignored)
%
```

Another use of a pseudo target is to create a rule for projects with multiple executables. This rule has a target and a dependency list, but no executables:

```
project:  program1  prog2  mystuff3


program1: program1.o class2.o otherthings.o
     g++ -o program1 program1.o class2.o otherthings.o -lm

...  and so on
```

In this case, the pseudo target `project` depends on multiple executable files. Typing `make project` or placing this rule at the top of the rule list for default application will cause `make` to create all of the executables.

Variables

Consider the following situation. In a large project, a bug has come up. It is not known what the source of the bug is. You have been assigned to debug the problem. This requires going to the makefile and adding the debugging option (`-g`) to all of the compiling commands. And after the problem is solved, you must go back and restore those options to their original form.

To help this situation, makefiles allow the use of variables. A makefile variable is assigned a string value in a similar to an assignment in a programming language:

```
CC = g++
```

The symbol CC is the variable and its value is `g++`. Makefile variable names are case sensitive. By convention, they are all capital letters. Variables are typically set once, at the beginning of the makefile.

There are some standard variable names that are used for common purposes. The name `CC` is used to hold the name of the compiler. The variable `CCFLAGS` holds C++ compiler options and `CFLAGS` holds C compiler options.

Unlike most programming languages, using the value of makefile variable does not consist of simply giving the variable name. To use a makefile variable it is necessary to put the name in parantheses and place a dollar sign in front. For example, the variable `CC` is given a value as described above. To use the variable, it is necessary to write the expression `$(CC)`. To use the variable `CCFLAGS`, the expression `$(CCFLAGS)` must be used:

```
        CC = g++
        CCFLAGS = -O -Wall

        ...


        twister:  twister.o  rotate.o
              $(CC) $(CCFLAGS) -o twister  twister.o rotate.o -lm
```

Built-in Rules

[under construction]

Example

Here is a simple example of a complete makefile for a relatively small project consisting of several pieces.

```
PROGNAME=fm
CC=g++

CCFLAGS=-O
CFLAGS=$(CFLAGS)
LDFLAGS=-lm
LIB=walshlib.a

SRCS=\
bitutils.cpp\
main.cpp\
mathlib.cpp\
linkage.cpp

HDRS=\
constants.h\
bitutils.h\
main.h\
mathlib.h\
linkage.h

ALLFILES=$(HDRS) $(SRCS) makefile

LIBOBJS=\
bitutils.o\
main.o\
mathlib.o\
linkage.o

main:   main.o $(LIB)
        $(CC) -o $(PROGNAME) $(CCFLAGS) main.o $(LIB) $(LDFLAGS)

all:
        touch  $(SRCS)
        make

$(LIB): $(LIBOBJS)
```

```
        echo Build libs
        ar vr $@ $?

size:   $(HDRS)  $(SRCS)
        wc $?

srcs:   $(HDRS)  $(SRCS)
        echo $(HDRS)  $(SRCS)

allfiles: $(ALLFILES)
        echo $(ALLFILES)

clean:
        rm -f *.o core

clobber:
        rm -f *.o *.a core

shar:
        shar -Z $(ALLFILES) > main.shar

backup: $(FILES)
        mv Backup/backup Backup/backup.old
        shar $(ALLFILES) | gzip > Backup/backup

tar:
        tar -cvf ~/fm.tar $(ALLFILES)
```

Makefile for Bison + Flex

This file is for use in compiling **calc.l** and **calc.y**.

```
BIN  = calc
CC   = gcc
CFLAGS = -g
# CCFLAGS = -DCPLUSPLUS -g  # for use with C++ if file ext is .cc
# CFLAGS = -DCPLUSPLUS -g  # for use with C++ if file ext is .c

SRCS = $(BIN).y $(BIN).l
OBJS = lex.yy.o $(BIN).tab.o
LIBS = -lfl -lm

$(BIN): $(OBJS)
        $(CC) $(CCFLAGS) $(OBJS) $(LIBS) -o $(BIN)

$(BIN).tab.h $(BIN).tab.c: $(BIN).y
        bison -v -t -d $(BIN).y

lex.yy.c: $(BIN).l $(BIN).tab.h
        flex -d $(BIN).l  # -d debug

all:
        touch $(SRCS)
        make

clean:
	rm -f $(OBJS) $(BIN) lex.yy.c $(BIN).tab.h $(BIN).tab.c $(BIN).tar

tar:
	tar -cvf $(BIN).tar $(SRCS) makefile
```

Table of Predefined Macros

This is from the Sun make man page.

Use	**Macro**	**Default Value**
Library	AR	ar
Archives	ARFLAGS	rv
Assembler	AS	as
Commands	ASFLAGS	
	COMPILE.s	$(AS) $(ASFLAGS)
	COMPILE.S	$(CC) $(ASFLAGS) $(CPPFLAGS) -c
C	CC	cc
Compiler	CFLAGS	
Commands	CPPFLAGS	
	COMPILE.c	$(CC) $(CFLAGS) $(CPPFLAGS) -c
	LINK.c	$(CC) $(CFLAGS) $(CPPFLAGS) $(LDFLAGS)
C++	CCC	CC
Compiler	CCFLAGS	CFLAGS
Commands	CPPFLAGS	
	COMPILE.cc	$(CCC) $(CCFLAGS) $(CPPFLAGS) -c
	LINK.cc	$(CCC) $(CCFLAGS) $(CPPFLAGS) $(LDFLAGS)
	COMPILE.C	$(CCC) $(CCFLAGS) $(CPPFLAGS) -c
	LINK.C	$(CCC) $(CCFLAGS) $(CPPFLAGS) $(LDFLAGS)
FORTRAN 77	FC	f77
Compiler	FFLAGS	
Commands	COMPILE.f	$(FC) $(FFLAGS) -c
	LINK.f	$(FC) $(FFLAGS) $(LDFLAGS)
	COMPILE.F	$(FC) $(FFLAGS) $(CPPFLAGS) -c
	LINK.F	$(FC) $(FFLAGS) $(CPPFLAGS) $(LDFLAGS)

FORTRAN 90	FC	f90
Compiler	F90FLAGS	
Commands	COMPILE.f90	$(F90C) $(F90FLAGS) -c
	LINK.f90	$(F90C) $(F90FLAGS) $(LDFLAGS)
	COMPILE.ftn	$(F90C) $(F90FLAGS) $(CPPFLAGS) -c
	LINK.ftn	$(F90C) $(F90FLAGS) $(CPPFLAGS) $(LDFLAGS)
Link Editor	LD	ld
Command	LDFLAGS	
lex	LEX	lex
Command	LFLAGS	
	LEX.l	$(LEX) $(LFLAGS) -t
lint	LINT	lint
Command	LINTFLAGS	
	LINT.c	$(LINT) $(LINTFLAGS) $(CPPFLAGS)
Modula 2	M2C	m2c
Commands	M2FLAGS	
	MODFLAGS	
	DEFFLAGS	
	COMPILE.def	$(M2C) $(M2FLAGS) $(DEFFLAGS)
	COMPILE.mod	$(M2C) $(M2FLAGS) $(MODFLAGS)
Pascal	PC	pc
Compiler	PFLAGS	
Commands	COMPILE.p	$(PC) $(PFLAGS) $(CPPFLAGS) -c
	LINK.p	$(PC) $(PFLAGS) $(CPPFLAGS) $(LDFLAGS)
Ratfor	RFLAGS	
Compilation	COMPILE.r	$(FC) $(FFLAGS) $(RFLAGS) -c
Commands	LINK.r	$(FC) $(FFLAGS) $(RFLAGS) $(LDFLAGS)
rm Command	RM	rm -f
sccs	SCCSFLAGS	
Command	SCCSGETFLAGS	-s
yacc	YACC	yacc
Command	YFLAGS	

	YACC.y	$(YACC) $(YFLAGS)
Suffixes List	SUFFIXES	.o .c .c~ .cc .cc~ .y .y~ .l .l~ .s .s~ .sh .sh~ .S .S~ .ln .h .h~ .f .f~ .F .F~ .mod .mod~ .sym .def .def~ .p .p~ .r .r~ .cps .cps~ .C .C~ .Y .Y~ .L .L .f90 .f90~ .ftn .ftn~

Further Reading

MIT overview of make
GNU on make

Robert Heckendorn Last updated:Jan 27, 2004