Makefiles - Collected Slides

Site Search

Slide 1

1 *Title* Makefiles - Title

Title:

Introduction to ``*make*'' for Handling Software Projects

Abstract:

As software projects grow, the task of re-compiling and linking the code together to produce an executable program becomes increasingly more difficult or lengthy. Investing a little time in understanding ``*make*'' and creating your own ```Makefile`'' pays enormous dividends reducing the debugging cycle and self-documenting the tasks necessary to produce an executable. ``*make*'' is a simple and useful tool that compares the dependencies in the ```Makefile`'', with respect to the source file time stamps, executing the given commands to create an up-to-date dependent. This discussion starts with very simple ```Makefile`s'' to more involved and fiendishly clever ones, examining each aspect of ``*make*''.

Slide 2

2 *Introduction* Makefiles - Introduction

make is a useful tool for managing software projects. A `Makefile` can be the collective memory of commands necessary to build or manage a software project. There are several flavors and variants of *make*. In this tutorial we'll concentrate on the GNU *make*, but only those features that can be reasonably expected in one of the other flavors (SysV or BSD).

A `Makefile` contains targets, prerequisites, commands, and macros. All of these terms will become clear in the presentation. The first question to ask is why and when

should a `Makefile` be used. (In this tutorial % is the command prompt and entered commands are in **bold**.)

``Why should I use a `Makefile`?''

When developing software the usual cycle is to edit, compile, execute, debug and repeat until all the bugs are found or when satisfied. This involves lots of repetitive steps. Assuming that the file `sub2.c` was edited, instead of typing:

```
% cc -c sub2.c
% cc -o mainexe main.o sub1.o sub2.o sub3.o
```

just type

```
% make
```

if you have invested a minute or two to create a `Makefile`.

You can save lots of time during the compilation step by breaking up your software project into separate compilation units. Only those units that are changed get recompiled. If the source is non-trivial (*i.e.* large), this can result in significant time savings.

``When should I use a `Makefile`?''

- When there is more than one file to handle.
- If the code is expected to be built on different machines.
- There are special handling steps.
- If you consider your time to be valuable.
- If you expect to rebuild your executable at some later point - the `Makefile` retains the *memory* of the needed steps.

Slide 3

3 *Done Very Simply* Makefiles - Done Very Simply

The very simple `Makefile` contains dependencies, with commands to create the *target* from the *prerequisites*. The entries look like this:

```
target : prerequisites_1 prerequisites_2 ...
<tab>   commands to build target from the prerequisites
<tab>   other commands to associate with target ...

...repeated for each target or created file...
```

Suppose you have a simple project:

```
main.c  proj.h  sub1.c  sub2.c  sub3.c
```

where `main.c` depends on the `sub[1-3].c` files and all the `*.c` source files depend on `proj.h`. Example `Makefile` :

```
# this is a comment
mainx : main.o sub1.o sub2.o sub3.o
        cc -o mainx main.o sub1.o sub2.o sub3.o

main.o : main.c proj.h
        cc -c main.c

sub1.o : sub1.c proj.h
        cc -c sub1.c

sub2.o : sub2.c proj.h
        cc -c sub2.c

sub3.o : sub3.c proj.h
        cc -c sub3.c
```

Now if you type *make* the following commands get executed:

```
% make
cc -c main.c
cc -c sub1.c
cc -c sub2.c
cc -c sub3.c
cc -o mainexe main.o sub1.o sub2.o sub3.o
```

If you edit `proj.h` and run *make*, the same commands are executed. However, if you edit `sub2.c` only the following commands are performed:

```
% make
cc -c sub2.c
cc -o mainexe main.o sub1.o sub2.o sub3.o
```

Even simpler `Makefile`s

The following `Makefile`, which shows nothing but the dependencies and how to build the executable, can be used to perform the same set of operations:

```
mainx : main.o sub1.o sub2.o sub3.o
        cc -o mainx main.o sub1.o sub2.o sub3.o

main.o sub1.o sub2.o sub3.o : proj.h
```

This works because *make* knows how to create `.o` files from `.c` files. These are termed *suffix rules* or *implicit rules*. You can create your own or override the ones provided by *make*. This will be looked at in detail later.

How does *make* know which one to build?

If you type *make*, the program looks for a file named `makefile` in the current working directory and if it doesn't exist then looks for one named `Makefile` (this is the prefered name to use). It reads this file creating a dependency tree. The first *target* listed in the file is the default one to build when no *target* is given on the command-line.

make checks the time/date stamp of the target compared to the prerequisites. If the prerequisite is later than the target then *make* the associated actions.
Suppose you want to compile `sub3.c` and nothing else, then just type the target on the command-line:

```
% make sub3.o
```

`Makefile` tricks

Targets don't require prerequisites, which means that if they are targeted (on the command-line) they will be executed regardless. This is far more useful than realized. This gives you a way to execute often used groups of commands. The following `Makefile` adds the targets - *help*, *clean*, and the target *clobber* which depends on *clean*:

```
mainx : main.o sub1.o sub2.o sub3.o
        cc -o mainx main.o sub1.o sub2.o sub3.o
```

```
main.o sub1.o sub2.o sub3.o : proj.h

clobber : clean
        -rm -f mainx
clean :
        -rm -f a.out core *.o
help :
        @echo ""
        @echo "make             - builds mainx"
        @echo "make clean       - remove *.o files"
        @echo "make clobber     - removes all generated files"
        @echo "make help        - this info"
        @echo ""
```

Give the following command, which gives the following result

```
% make help

make            - builds mainexe
make clean      - remove *.o files
make clobber    - removes all generated files
make help       - this info
```

Notice that the `Makefile` contains a couple of special characters (`@` and `-`) prior to the commands. First, we need to understand how *make* executes the commands in the `Makefile`. Each line that has an un-escaped new-line is echoed to the screen and is passed off to a shell child process shell which then executes it. If the shell returns a non-zero exit value *make* will then generally abort any further processing with a warning.

The ``-" tells *make* to ignore any return value and to continue on to the next command. Disregarding return values can be set globally with either the ``-*i*" option or the ``fake" target `.IGNORE` in the `Makefile`.

The other special character ``@" tells *make* not to echo the command to *stdout*. Echoing can be disabled globally with either the ``-*s*" option or the fake target `.SILENT`.

Slide 4

4 *Macros and More* Makefiles - Macros and More

The previous examples generally have a lot of repetitive text. *make* provides a simple macro mechanism. Macros are defined in a number of ways (listed in increasing order of precedence):

1. Internally defined by *make*
2. Shell environment variables

```
setenv CC "gcc"
```

3. Macros defined in the `Makefile`:

```
OBJS    = main.o sub1.o sub2.o sub3.o
```

   There can be no <tab>s before the macro name and no colons before the equal (=) sign.
4. Macros can be defined on the command-line:

```
make CC=gcc
```

There is a special circumstance (using the `-e` option) when the second and third entries in the list are swapped in precedence.

Macros are used in the `Makefile` by bracketing the macro name with either parentheses ``()" or braces ``{}" and prepending a dollar sign ``$". If the macro name is a single character the parentheses and braces can be neglected. This is a likely source of errors in `Makefile`s.

By convention macro names should be uppercase, but lowercase letters, numbers and underscores are safe to use also.

If a macro name is not defined *make* will just use a null string instead and not complain. This is not an error.

Can Not Dynamically Reset Macros

One of the problems with macros is that, unlike a script, macros can not be reassigned a different value halfway through the processing. A macro can not have one value when operating on one target and a different value for another. ``Why?" This highlights the difference between scripts and *make*. Scripts are executed in a linear fashion where there is a clear sequence of events to execute. *make*, on the other hand, defines an inverted tree structure of dependencies with associated commands to create targets. The tree can be traversed either depth-first or level descent; therefore, the order of operations is not necessarily linear. *make* must read through the entire `Makefile` before starting any dependency analysis. The last macro declaration defines the overall macro definition to be used everywhere.

There are ways to get around this limitation by using a *recursive make* invocation. However, for most applications it won't be necessary if you design your `Makefile` script accordingly.

Predefined and Internal Macros

make has several predefined macros that makes rule writing easier and more generalized. However, be careful. The accidental substitution of the wrong macro may

cause the overwriting of the source file. In particular, if $< or $? is used when $@ should have been used in the action. The macros of most interest are:

$@ the name of the file to be ``made''

$? the set of dependent names that are younger than the target

$< the name of the related file that caused the action (the precursor to the target) - this is only for suffix rules

$* the shared prefix of the target and dependent - only for suffix rules

**$$** escapes macro substitution, returns a single ``$''.

Suffix rules will be discussed later.

The following example `Makefile` demonstrates some of these special macros:

```
# tests of the various built-in macros
SRCS    = aaa.c bbb.c ccc.c
OBJS    = ${SRCS:.c=.o}

.SILENT :

all : xxx yyy

xxx : $(SRCS)
        echo "target ======== $@"
        echo "all   sources = $(SRCS)"
        echo "newer sources = $?"
        echo "all   objects = $(OBJS)"

yyy : *.c
        echo "target ======== $@"
        echo "all   sources = $(SRCS)"
        echo "newer sources = $?"
```

Which gives the following output:

```
% make
target ======== xxx
all   sources = aaa.c bbb.c ccc.c
newer sources = aaa.c bbb.c ccc.c
all   objects = aaa.o bbb.o ccc.o
target ======== yyy
all   sources = aaa.c bbb.c ccc.c
newer sources = aaa.c bbb.c ccc.c xxx.c
```

The above example shows a couple of new features.

Editing Macros

The first is the limited macro substitution

```
OBJS    = ${SRCS:.c=.o}
```

Which says to substitute ``.o" for the last two characters, if they happen to be ``.c", in the list given by `$(SRCS)`. This provides an easy way to create lists of object files, *man* page files, executables, *etc.* from a single list of sources. However, not every *make* allows this type of substitution. (The Cray, IBM SP, and GNU ones do.)

Dependency Globbing

The target or prerequisites could use file ``globbing" symbols. The above example shows the prerequisites for `yyy` as `*.c`, where `*` matches any number of characters. The other globbing character is `?`, which matches any single character. Note that this does **not** work reliably if the target uses globbing characters. It works for GNU *make*, but not for the Cray or IBM SP. It's generally not a good idea to use globbing at all and I have rarely seen `Makefile`s with it.

Hosts of Other Macros

There are a number of built-in macros, and they can be listed out with all the various suffix rules and environment variables with:

```
make -p -f /dev/null
```

The ones I consider to be the most import are:

`SHELL`
: The shell to use for executing commands. Some *make*s don't honor it and use `/bin/sh` anyways. Others will take the users login shell. You should always set this macro and define it to `/bin/sh` and write your actions appropriately.

`MAKE`
: the name of the *make* command. Only GNU uses the full pathname if specified on the command-line, else just uses command name. Therefore, need to make sure the desired *make* is found first in the command `PATH` if trying to use another version of *make*.

`MAKEFLAGS` or `MFLAGS`
: the options passed to the `Makefile`. However, it's not done consistently. GNU and Cray pass only the option letters collected together (*e.g.* `ni`). IBM SP passes dashed options each separate (*e.g.* `-n -i`). GNU puts the dashed options into `MFLAGS`, a macro not used by the other two!

`VPATH`
: a colon (`:`) separated path of directories to search for prerequisites.

There are numerous macros that refer to various compilers and tools with their associated macro for passing options.

Macro	Flags	Tool
FC/CF/F77	**FFLAGS**	**Fortran compiler**

CC	CFLAGS	C compiler
AS	ASFLAGS	assembler
LD	LDFLAGS	object loader
AR	ARFLAGS	archiver
LEX	LFLAGS	lexical parser
YACC	YFLAGS	grammar parser

This is just a small portion. Except for the Fortran compiler, the rest listed are fairly standard to each implementation. These macros are used by the suffix rules described in the next section.

Slide 5

5 *Implicit or Suffix Rules* Makefiles - Implicit or Suffix Rules

In earlier examples we either explicitly spelled out how each target is ``made'' from the prerequisites or relied on *make* magic to do the right thing.

This section looks into how to write and provide suffix rules, which are also called ``implicit rules''.

Writing your own rules frees the `Makefile` from being intimately dependent on any particular *make* or platform and can shorten `Makefile`s for large projects.

Suffix rules are, in a sense, generic rules for converting, say `.c` files to `.o` files.

Suffix rules are of the form:

```
s1s2 :
        commands to get s2 from s1
```

Suffixes can be any string of characters, and by convention usually include a ``dot'' (`.`), but does not require it. The allowed suffixes are given by the `.SUFFIXES` *fake target*.

The following is a reworking of an earlier example with helpful comments:

```
CC      = gcc
CFLAGS  = -g
LD      = $(CC)
LDFLAGS =
RM      = rm

EXE     = mainx
SRCS    = main.c sub1.c sub2.c sub3.c
OBJS    = ${SRCS:.c=.o}
```

```
# clear out all suffixes
.SUFFIXES:
# list only those we use
.SUFFIXES: .o .c

# define a suffix rule for .c -> .o
.c.o :
        $(CC) $(CFLAGS) -c $<

# default target by convention is ``all''
all : $(EXE)

$(EXE) : $(OBJS)
        $(LD) -o $@ $(OBJS)

$(OBJS) : proj.h

clean :
        -$(RM) -f $(EXE) $(OBJS)
```

Strongly recommend that you write your own suffix rules for all those used within the software project. The need for this is demonstrated by the lack of consistency in macro names for the Fortran compiler shown in the last section.

However, if you are only using C and the associated tools (*lex*, *yacc*, *ar*, and *ld*) then the default macros and suffix rules are probably sufficient since they are fairly standard.

Looking at Predefined Rules and Macros

The following command will list the predefined rules and macros. The option ``-f'' tells *make* which `Makefile` to use, in this case `/dev/null` the empty file. The output is somewhat lengthy so it's a good idea to pipe it into a pager or into a file.

```
% make -p -f /dev/null
```

Looking at the predefined suffix rules are useful for composing your own and to observe which macros are used for each compiling tool.

Slide 6

6 *VPATH and RCS* Makefiles - VPATH and RCS

The macro `VPATH` tells *make* where, in addition to the local directory, to search for prerequisites to satisfy the *make* rules. `VPATH` has the same format as the shell `PATH`, a colon (:) delimited selection of directory paths. Don't include ``.'' since the current directory is always searched.

```
VPATH   = ./RCS:../:../RCS:$(HOME)/sources
```

The following example keeps the sources under RCS source control. If a source, say `sub1.c`, is put under RCS control and the `RCS` directory exists then it is kept in `RCS/sub1.c,v` and the compilable source only exists when checked out with ``co''.

```
CC      = gcc
CFLAGS  = -g
LD      = $(CC)
LDFLAGS =

RM      = rm
ECHO    = echo

EXE     = mainx
SRCS    = main.c sub1.c sub2.c sub3.c
OBJS    = ${SRCS:.c=.o}

VPATH   = ./RCS

.SUFFIXES:
.SUFFIXES: .o .c .c,v

.c.o :
        @$(ECHO) "===== .c -> .o rule"
        $(CC) $(CFLAGS) -c $<

.c,v.o :
        @$(ECHO) "===== using RCS for .c to .o"
        co -u $*.c 2>/dev/null
        $(CC) $(CFLAGS) -c $*.c
        $(RM) -f $*.c

all : $(EXE)

$(EXE) : $(OBJS)
        $(LD) -o $@ $(OBJS)

$(OBJS) : proj.h

clean :
        -$(RM) -f $(EXE) $(OBJS)
```

Suppose the directory looks like this:

```
% ls -l
-rw-r-----   1 rk    owen     465 Jun 17 08:29 Makefile
drwxr-x---   2 rk    owen    1024 Jun 17 08:31 RCS/
-rw-r-----   1 rk    owen      36 Jun 17 08:31 proj.h
-rw-r-----   1 rk    owen      44 Jun 17 08:27 sub2.c
```

Which shows that `sub2.c` is checked out, presumably, for modifying. The ``-r'' option below says to not use any of the pre-defined implicit rules. It was necessary in this example to force the use of the implicit rules defined in the `Makefile` since the pre-defined ones interfere with ours. The `Makefile` gives the following results:

```
% make -r
===== using RCS for .c to .o
co -u main.c 2>/dev/null
```

```
gcc -g -c main.c
rm -f main.c
===== using RCS for .c to .o
co -u sub1.c 2>/dev/null
gcc -g -c sub1.c
rm -f sub1.c
===== .c -> .o rule
gcc -g -c sub2.c
===== using RCS for .c to .o
co -u sub3.c 2>/dev/null
gcc -g -c sub3.c
rm -f sub3.c
gcc -o mainx main.o sub1.o sub2.o sub3.o
```

Notice that for sources in RCS that they are checked out, compiled, and the source is removed, since they're not necessary to keep around. Suppose we've been modifying `sub2.c` and rerun *make*.

```
% make
===== .c -> .o rule
gcc -g -c sub2.c
gcc -o mainx main.o sub1.o sub2.o sub3.o
```

Slide 7

7 *Redefining Macros* Makefiles - Redefining Macros

Resetting Macros For Different Conditions

In an earlier section it was stated that macros could not be redefined dynamically within a `Makefile`. The last setting for a macro is the value used through out the `Makefile`.

There is a way around this by using a *recursive make*. In other words, a `Makefile` that calls itself with different targets or macro definitions. In the following example, if the *make* is not called with a proper argument or no argument it uses the ``*fake*" target `.DEFAULT` to execute the command given there. For the sake of economy it just recursively calls the *make* itself with the target `help`.

If it's given a target of `linux`, `unicos`, or `aix` it then recursively calls itself with the macros `CC` and `LD` appropriately set for those platforms.

```
RM      = rm
ECHO    = echo

EXE     = mainx
SRCS    = main.c sub1.c sub2.c sub3.c
OBJS    = ${SRCS:.c=.o}

# do this if given an invalid target
.DEFAULT :
        @$(MAKE) help
```

```
help :
        @$(ECHO) "--------------------------------------"
        @$(ECHO) "do 'make xxx' where xxx=linux|unicos|aix"
        @$(ECHO) "--------------------------------------"

all : $(EXE)

########################################################################
# this only works reliably with GNU ``make'' which correctly handles
# MAKE & MFLAGS
########################################################################
linux :
        $(MAKE) $(MFLAGS) CC=gcc LD=gcc         all

unicos :
        $(MAKE) $(MFLAGS) CC=cc LD=segldr       all

aix :
        $(MAKE) $(MFLAGS) CC=xlc LD=ld          all

$(EXE) : $(OBJS)
        $(LD) -o $@ $(OBJS)

$(OBJS) : proj.h

clean :
        -$(RM) -f $(EXE) $(OBJS)
```

In this example we give an invalid target to exercise the `.DEFAULT` target, and we use the `-s` option which is equivalent to adding the `.SILENT` target to suppress command echoing.

```
% make -s xxx
--------------------------------------
do 'make xxx' where xxx=linux|unicos|aix
--------------------------------------
```

Now we try the following, where the option `-n` says to echo any executed commands, but do not perform them (except for the recursive *make* if using GNU):

```
% make -s -n aix
make -sn CC=xlc LD=ld           all
xlc    -c main.c -o main.o
xlc    -c sub1.c -o sub1.o
xlc    -c sub2.c -o sub2.o
xlc    -c sub3.c -o sub3.o
ld -o mainx main.o sub1.o sub2.o sub3.o
```

Using this technique adds flexibility and allows you to tailor your `Makefile` for each platform you anticipate using.

As indicated in the `Makefile` it will reliably run with the GNU *make* only since there is no real standard regarding the macros `MAKE` and `MAKEFLAGS`. The GNU *make* generally ports well to all common UNIX platforms so obtaining one is not difficult. However, all is not lost if you don't or can't have a GNU *make* if you use environment

variables. The following example shows a passable way. The `/bin/env` temporarily sets environment variables that are passed on to the executable only.

```
% env MAKE=/bin/make MFLAGS=-ni /bin/make aix
        /bin/make -ni CC=xlc LD=ld              all
        xlc  -c -o main.o main.c
        xlc  -c -o sub1.o sub1.c
        xlc  -c -o sub2.o sub2.c
        xlc  -c -o sub3.o sub3.c
        ld -o mainx main.o sub1.o sub2.o sub3.o
```

Slide 8

8 *Recursive Make For Sub-directories*

Makefiles - Recursive Make For Sub-directories

Large software projects generally are broken into several sub-directories, where each directory contains code that contributes to the whole.

The way it can be done is to do a recursive *make* descending into each sub-directory. To keep a common set of macros that are easily maintained we use the `include` statement which is fairly common in most *make*s

The following is the directory structure of the sources:

```
-rw-r-----   1 rk  owen     625 Jun 17 16:42 Makefile
-rw-r-----   1 rk  owen     142 Jun 17 16:43 Makefile.inc
-rw-r-----   1 rk  owen     133 Jun 17 14:32 main.c
-rw-r-----   1 rk  owen     120 Jun 17 14:34 proj.h

drwxr-x---   2 rk  owen    1024 Jun 17 16:54 subdira
-rw-r-----   1 rk  owen      45 Jun 17 14:28 subdira/sub2a.c
-rw-r-----   1 rk  owen      45 Jun 17 14:28 subdira/sub3a.c
-rw-r-----   1 rk  owen     346 Jun 17 16:34 subdira/Makefile
-rw-r-----   1 rk  owen      45 Jun 17 14:25 subdira/sub1a.c

drwxr-x---   3 rk  owen    1024 Jun 17 16:54 subdir
-rw-r-----   1 rk  owen     524 Jun 17 16:07 subdir/Makefile
-rw-r-----   1 rk  owen      52 Jun 17 14:33 subdir/sub1.c
-rw-r-----   1 rk  owen      52 Jun 17 14:33 subdir/sub2.c
-rw-r-----   1 rk  owen      52 Jun 17 14:33 subdir/sub3.c

drwxr-x---   2 rk  owen    1024 Jun 17 16:54 subdir/subsubdir
-rw-r-----   1 rk  owen      47 Jun 17 14:28 subdir/subsubdir/subsub3.c
-rw-r-----   1 rk  owen     390 Jun 17 16:35 subdir/subsubdir/Makefile
-rw-r-----   1 rk  owen      47 Jun 17 16:53 subdir/subsubdir/subsub1.c
-rw-r-----   1 rk  owen      47 Jun 17 14:28 subdir/subsubdir/subsub2.c
```

Here is the `Makefile.inc` and `Makefile` in the root:

`Makefile.inc`

```
# put common definitions in here
CC      = gcc
```

```
PRJCFLAGS       = -g
LD      = gcc
LDFLAGS =
AR      = ar
ARFLAGS =
RANLIB  = ranlib
RM      = rm
ECHO    = echo

SHELL   = /bin/sh

.SILENT :
```

Makefile

```
include Makefile.inc

DIRS    = subdir subdira
EXE     = mainx
OBJS    = main.o
OBJLIBS = libsub.a libsuba.a libsubsub.a
LIBS    = -L. -lsub -lsuba -lsubsub

all : $(EXE)

$(EXE) : main.o $(OBJLIBS)
        $(ECHO) $(LD) -o $(EXE) $(OBJS) $(LIBS)
        $(LD) -o $(EXE) $(OBJS) $(LIBS)

libsub.a libsubsub.a : force_look
        $(ECHO) looking into subdir : $(MAKE) $(MFLAGS)
        cd subdir; $(MAKE) $(MFLAGS)

libsuba.a : force_look
        $(ECHO) looking into subdira : $(MAKE) $(MFLAGS)
        cd subdira; $(MAKE) $(MFLAGS)

clean :
        $(ECHO) cleaning up in .
        -$(RM) -f $(EXE) $(OBJS) $(OBJLIBS)
        -for d in $(DIRS); do (cd $$d; $(MAKE) clean ); done

force_look :
        true
```

Which produces this output:

```
% make
looking into subdir : make -s
ar rv ../libsub.a sub1.o sub2.o sub3.o
a - sub1.o
a - sub2.o
a - sub3.o
ranlib ../libsub.a
looking into subsubdir : make -s
ar rv ../../libsubsub.a subsub1.o subsub2.o subsub3.o
a - subsub1.o
a - subsub2.o
```

```
a - subsub3.o
ranlib ../../libsubsub.a
looking into subdira : make -s
ar rv ../libsuba.a sub1a.o sub2a.o sub3a.o
a - sub1a.o
a - sub2a.o
a - sub3a.o
ranlib ../libsuba.a
looking into subdir : make -s
looking into subsubdir : make -s
gcc -o mainx main.o -L. -lsub -lsuba -lsubsub
```

Suppose we *touch* a file deep in the directory structure:

```
% touch subdir/subsubdir/subsub2.c
% make
looking into subdir : make -s
looking into subsubdir : make -s
ar rv ../../libsubsub.a subsub2.o
r - subsub2.o
ranlib ../../libsubsub.a
looking into subdira : make -s
looking into subdir : make -s
looking into subsubdir : make -s
gcc -o mainx main.o -L. -lsub -lsuba -lsubsub
```

Notice that the `Makefile` has a dummy target named `force_look` that the libraries depend on. This ``*file*" is never created hence *make* will always execute that *target* and all that depend on it. If this was not done then *make* would have no idea that `libsubsub.a` depends on `subdir/subdir/subsub2.c` unless we include these dependencies in the root `Makefile`. This defeats the purpose of breaking up a project into separate directories. This mechanism pushes the dependency checking into lower level `Makefile`s.

Here is a representive sub-directory `Makefile` :

```
include ../Makefile.inc

CFLAGS  = $(PRJCFLAGS) -I..
OBJLIBS = ../libsub.a ../libsubsub.a
OBJS    = sub1.o sub2.o sub3.o

all : $(OBJLIBS)

../libsub.a : $(OBJS)
        $(ECHO) $(AR) $(ARFLAGS) rv ../libsub.a $?
        $(AR) $(ARFLAGS) rv ../libsub.a $?
        $(ECHO) $(RANLIB) ../libsub.a
        $(RANLIB) ../libsub.a

../libsubsub.a : force_look
        $(ECHO) looking into subsubdir : $(MAKE) $(MFLAGS)
        cd subsubdir; $(MAKE) $(MFLAGS)

clean :
        $(ECHO) cleaning up in subdir
        -$(RM) -f $(OBJS)
        cd subsubdir; $(MAKE) $(MFLAGS) clean
```

```
force_look :
        true
```

We use the pre-defined implicit rules, and define `CFLAGS` with project wide options and where to find the include files.

Notice the ganged shell commands to *cd* into a subdirectory and to execute a *make*. It's not for compactness or convenience ... it's required for correct behavior. The following section will detail some of these issues.

Slide 9

9 *Make and the Shell* Makefiles - Make and the Shell

In the `Makefile` each action line is a separate invocation of a shell child process and any shell variables or current working directory changes are independent of each other.

The following table shows how to compress the various Bourne shell statements onto a single logical line. However, it's a good idea to break up the statements with white-space formatting on to separate lines. This can be done by *escaping* the new-line.

Other hints:

- Set the macro `SHELL=/bin/sh` to make sure which shell will be invoked. (If you pick some other shell there's no guarantee that the implementation of *make* will honor it.)
- Use `$$variable` to reference a shell variable (such as in the `for` loop). The ``$$" tells *make* to not do any macro expansion.
- Use *test* instead of `[` and `]`, since errors occur when there is inadequate spacing given for the brackets.
- Either quote or rely on string catenation for comparisons, *e.g.*
  `test "$$x" = "abc"` or
  `test x$$x = xabc`
  test doesn't handle empty arguments too well!
- Become familiar with *expr* for doing simple math operations and parsing.
- Likewise with *basename* and *dirname* for parsing out parts of the file name and path.
- Understand when to use quotes (`"`) or apostrophes (`'`).

Bourne Shell conditional and looping statements

Expanded	Condensed
if *com1* then *com2* fi	if *com1* ; then *com2* ; fi

if *com1*
then
  com2
else
  com3
fi
	if *com1* ; then *com2* ; else *com3* ; fi

if *com1*
then
  com2
elif *com3*
then
  com4
else
  com5
fi
	if *com1* ; then *com2* ; elif *com3* ; \
	  then *com4* ; else *com5* ; fi

case *value* in
pattern1) *com1* ;;
pattern2) *com2* ;;
pattern3) *com3* ;;
esac
	case *value* in *pattern1*) *com1* ;; \
	  pattern2) *com2* ;; \
	  pattern3) *com3* ;; esac

for *variable* in *list* do *command* done	for *variable* in *list*; do *command* ; done
while *com1* do *com2* done	while *com1* ; do *com2* ; done
until *com1* do *com2* done	until *com1* ; do *com2* ; done

Slide 10

10 *Make and the Double colon* Makefiles - Make and the Double colon

make has a little used feature that is signified with the double colon (::). This tells

make to associate more than one *action* for a *target*. Normally, with a the single colon (:), you can have multiple target and prerequisites to map the dependencies, but only one of them can have an associated action. For example:

```
aaa.o : aaa.c in.h
        cc -c aaa.c

aaa.o : another.h
aaa.o : yetanother.h
```

The double colon allows your to do something like this:

```
libxxx.a :: sub1.o
        ar rv libxxx.a sub1.o

libxxx.a :: sub2.o
        ar rv libxxx.a sub2.o
```

Where the library `libxxx.a` depends on `sub[12].o` and will add them into the library collection as needed.

The double colon mechanism is very useful for automatically generated `Makefile`s. The actions can be collected together in a verticle sense, as opposed to the usual approach that lists prerequisites in a horizontal sense. The latter is more efficient from a *make* operational view, but can be difficult to automate `Makefile` generation for a batch of source files.

The following is a very contrived example that demonstrate some of the shell looping tricks and how to automatically self-generate a Makefile.

The ``header" part of the Makefile is not automatically generated and is carried from one version to the next. It has the executable `mainx` depend on the Makefile and below it depends on the `.o` files. The `Makefile` itself depends on the sources in the current directory. If you add a new source file, say `sub4.c`, *make* will then execute the action for the `Makefile` target. The action is a very sophisticated use of shell looping. It passes the `Makefile` through *sed* to cut out the header part, then echo and append the sentinel lines to `Makefile`. The `for` loop keys on every `.c` file in the current directory. It uses *basename* to create variables with the `.o` name. The action uses *gcc -MM* to parse a source file to generate a target with prerequisites. The following *echo*s generate dependencies for

`mainx` <- `.o` file <- `.c` file, *etc*.

Next, it creates a collected action, `clean`, for removing the `.o` file. Finally, the action does a recursive *make* to build the executable with the new `Makefile`!

```
#
LD      = gcc
SHELL   = /bin/sh
```

```
mainx : Makefile
        $(LD) -o mainx *.o

clean ::
        -$(RM) mainx

Makefile : *.c
        @sed -e '/^### Do Not edit this line$$/,$$d' Makefile \
                > MMM.$$$$ && mv MMM.$$$$ Makefile
        @echo "### Do Not edit this line" >> Makefile
        @echo "### Everything below is auto-generated" >> Makefile
        @for f in *.c ; do echo ===$$f=== 1>&2 ; ff=`basename $$f .c`.o ; \
        gcc -MM $$f ; echo ""; echo "mainx : $$ff" ; echo "$$ff : $$f" ; \
        echo '  $$(CC) $$(CFLAGS) -c '"$$f" ; echo "" ; echo "clean ::" ; \
        echo '  -$$(RM) '"$$ff" ; echo "" ; done >> Makefile
        @$(MAKE)

### Do Not edit this line
### Everything below is auto-generated
main.o: main.c proj.h

mainx : main.o
main.o : main.c
        $(CC) $(CFLAGS) -c main.c

clean ::
        -$(RM) main.o

sub1.o: sub1.c proj.h

mainx : sub1.o
sub1.o : sub1.c
        $(CC) $(CFLAGS) -c sub1.c

clean ::
        -$(RM) sub1.o

sub2.o: sub2.c proj.h

mainx : sub2.o
sub2.o : sub2.c
        $(CC) $(CFLAGS) -c sub2.c

clean ::
        -$(RM) sub2.o

sub3.o: sub3.c proj.h

mainx : sub3.o
sub3.o : sub3.c
        $(CC) $(CFLAGS) -c sub3.c

clean ::
        -$(RM) sub3.o
```

Suppose the source `sub4.c` is added to the existing project. It need not be explicitly added to the `Makefile`, just type ``*make*". It regenerates the `Makefile` and builds the executable accordingly.

```
% make
===main.c===
```

```
===sub1.c===
===sub2.c===
===sub3.c===
===sub4.c===
make[1]: Entering directory `/u/owen/rk/make/src/ex5
'
cc  -c sub4.c
gcc -o mainx *.o
make[1]: Leaving directory `/u/owen/rk/make/src/ex5'
make: `mainx' is up to date.
```

This is a fiendishly clever `Makefile`. normally I stay away from such overly clever *make* tricks, opting for explicit simplicity and control. However, it does demonstrate the power *make* and what is possible.

Slide 11

11 ``*Make" Summary* Makefiles - ``Make" Summary

A `Makefile` contains:

Dependency and Action Lines

- Comments

  ```
  # comments start with `#' and end with the new-line
  ```

- Macro Definitions

  ```
  macro_name = string
  ```

  The *`macro_name`* can contain any upper- or lower-case letters, digits, and the underscore (_). Upper-case letters are prefered by convention. Macro substitution occurs when given as `$(`*`macro_name`*`)` or `${`*`macro_name`*`}`. Single character `$(`*`macro_name`*`)`s are special and don't require the parentheses ``()" or brackets ``{}" for substitution.

- Dependency Lines

  ```
  targets :[:] [prerequisites]  [; [commands]]
  ```

  Defines the dependency relationships. Actions follow. There can not be any space before the targets.

- Suffix or Implicit Rules

  ```
  suffix[suffix] :[:]
  ```

  Defines implicit or generic rules for creating a file with the second suffix dependent on a file with the same base name and the first given suffix. No

spaces before or between suffices

- Actions

  ```
  <tab> [- @] shell command
  ```

  Actions must start with a tab, else not recognized. The shell command must be a valid single line command. The shell invocation is usually the Bourne shell (`/bin/sh`).

- Include Statement

  ```
  include file
  ```

  reads and evaluates *`file`* as if part of the current `Makefile`. Must not have any white-space at beginning of line.

Internal Macros

$@	the name of the file to be ``made''
$?	the set of dependent names that are younger than the target
$<	the name of the related file that caused the action (the precursor to the target) - this is only for suffix rules
$*	the shared prefix of the target and dependent - only for suffix rules
$$	escapes macro substitution, returns a single ``$''.

Macro String Substitution

${*macro_name*:*s1*=*s2*}	substitutes *s2* for any *s1* string that occurs in the list at the end any word delimited by white-space.

Special Macros

SHELL	tells *make* which command shell to invoke for actions. This is not always honored, and in general set it to `/bin/sh` and write all actions for the Bourne shell.
VPATH	the path including the current working directory that *make* will search for prerequisites to satisfy the rules.

Special Targets

.DEFAULT	Its associated actions are invoked whenever *make* is given a

	target that is not defined.
.IGNORE	Ignores all return codes from actions. Same as command-line option ``-i''. By default *make* will stop processing whenever a non-zero return status is received from an action.
.SILENT	Will not echo the action as its processed. Same as command-line option ``-s''. By default *make* will echo the action to *stdout* prior to invocation.
.SUFFIXES	Appends any given ``prerequisites'' to the list of suffixes with implicit rules. If none are given then wipe the list of suffixes.

Slide 12

12 *Conclusion* Makefiles - Conclusion

make and `Makefile`s are very useful for:

- Managing software projects
- Reducing build times during debugging cycles
- Keeping a record of compilation steps and options

Not all of the features were given here, but these are the major ones. The other features are not universal and are implementation dependent.

A useful mastery of *make* takes very little time to acquire. Time well spent!

Last Modified:

Brought to you by: *R.K. Owen, Ph.D.*
This page is *http://owen.sj.ca.us/rkowen/howto/slides/make/slides/ALL.html*