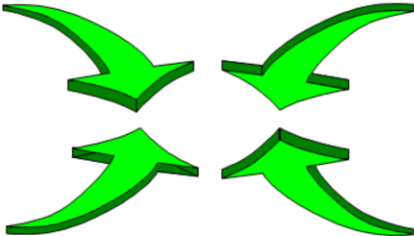


QACADV_v1.0

The Make Utility

- **Dependencies**
- **Macros**
- **Suffix rules**
- **Commands**
- **Forcing a rebuild**
- **Maintaining header files**



make is key utility in building applications, and is available on many operating systems. Its main purpose is to determine automatically which pieces of a large program need to be recompiled, and issue the commands to recompile them.

In this chapter, we will look at the basic rules for defining the source dependencies of a large program. This will be followed by writing some simple macros and looking at rules concerning filename suffixes.

Imbedding shell commands will also be discussed, the most common of which are calls to the C compiler.

We finish up with a solution to the problem of maintaining header file dependencies.

What is make?

- **A Command Generator**
 - Commands are executed by the shell
 - Usually associated with calling a compiler and linker
- **Has its own scripting language**
 - Similar, but different, from the shell
- **Works on dependencies between files**
 - Independent of compilers

make is a powerful command generator, and need not be confined to generating compile and link statements.

It is often associated with the C compiler, but you can build programs in most languages using it, provided it supports at least a *collection* (linker) phase separately.

Even complex programs can usually be recompiled by a single command line call to the compiler, which will also invoke the linker. This can be a lengthy process and, if only one source file has changed, unnecessary. A selective recompile could be difficult to figure out, particularly if an included (header) file was changed. It is critical that dependencies to these file are known and documented. A makefile stores these dependencies and executes selective compiles, and/or other commands.

Dependency Lines

- **Specified in the description file**
 - Text file
 - Usually named `./makefile` or `./Makefile`
 - override with `make -f filename`
- **Consist of target: dependency list ...**
 - Followed by one or more Command lines:
 - `<tab>Command`

```
myprog: myprog.o initial.o menu.o utils.o
        cc -o myprog myprog.o menu.o utils.o -lcurses

myprog.o: myprog.c common.h utils.h menu.h gameunit.h
        cc -c -d myprog.c

menu.o: menu.c common.h utils.h menu.h
        cc -c -d menu.c

utils.o: utils.c common.h utils.h
        cc -c -d utils.c
```

The **makefile** can be edited like any other text file. It can consist of macros, commands, and sets of dependency lists.

The dependency lists are where the relationships between files are stored. It is imperative that the lists are kept up to date. Adding a `#include` statement to a C source file will not automatically update the description file! There are several possible solutions to this problem, and are discussed later ...

In the example, `myprog` is dependent on `myprog.o`, `initial.o`, `menu.o`, and `utils.o`. Each of these object files are in turn dependent on a list of `.c` and `.h` files.

Command lines which follow the list describe how the target may be "made" - see later. They **must** be indented by a `<TAB>`, **not** spaces.

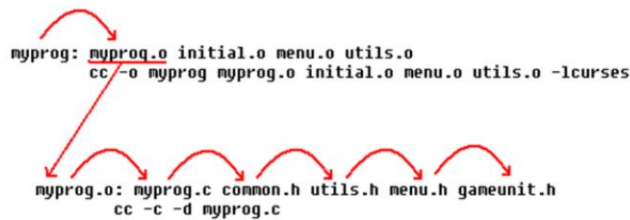
Certain special target names are predefined, some basic ones are:

- .DEFAULT: default commands if none are found for the requested target
- .SUFFIXES: define suffixes (file extensions). See later ...

the version of **make** you are using may support others.

Dependencies in Action

- **Does a file called "myprog" exist?**
 - If yes ...
- **Check myprog.o dependencies.**
 - Is myprog.c newer than myprog.o?
 - If yes, execute the command line (`cc -c -d myprog.c`)
 - If no ...
 - Is common.h newer than myprog.o?
 - If yes, execute the command line
 - If no ...
- etc ...



The date/time stamp is inspected on dependent files to check if they changed since the target was last built. If it has then it must be rebuilt, so further dependencies may be checked. The commands will be executed from the lowest level of dependency upwards. In the example above each of the object (.o) files will be checked in turn.

If a dependency cannot be found and there is no command to create it, then **make** will halt with :

```
make: Don't know how to make target. Stop.
```

The exact text varies between versions of make.

If none of the source files are later than the objects or executables, then **make** will exit having found nothing to do (usually with a suitable message).

Caution: when copying files remember that this often updates the date/time stamp, depending on the method used. Since files are often copied *en masse* in alphabetical order, it can lead to inconsistencies in the make. It is prudent to force a rebuild in such circumstances, see later ...

Macros

- **Used for textual substitution**
 - Macro-name = text # to create
 - \${Macro-name} #or
 - \$(Macro-name) # to use
- **Different from shell variable assignments**
 - Imbedded spaces are preserved
 - but watch end-of-line
 - no need for quotes - they are also preserved
 - Names can start with numbers, but are UPPER CASE by convention
- **Useful for maintaining lists and repeated strings**

Macros are similar to shell environment variables, but have their differences as well. They are useful when maintaining lists, or any other text, which is repeated through the makefile. Only one string need be altered, avoiding inconsistencies.

The definition has a more free format than, for example, the Korn shell, in that there may be optional spaces around the equals sign.

Macros may be used within others, provided they are already defined, for example:

```
EXT    =  cpp
FILE   =  myprog.${EXT}
```

Macros names of more than one character require braces { } when substituted, single character names do not (although it is still a good idea). Parentheses () may also be used, but in some versions of **make** they have special a meaning when handling libraries.

Note the comment marker, the hash character # (often referred to as a pound sign in U.S. books).

The line continuation character "escapes" the new line, and may be used to break-up long lists, for example:

```
OBJ = myprog.o initial.o menu.o utils.o other.o screen.o keys.o \
      fileio.o threads.o dbs.o primitives.o trace.o locks.o      \
      lang.o externals.o
```

Other Macro Definitions

- **From the shell**
 - Exported environment variables are available as macros
- **From the command line**
 - Define macros on the make command line
- **Built-in Macros**
 - Describe common utilities, like \$CC, \$LD, \$MAKE
 - Include "helpers", like \$@
- **include files**
 - Syntax is: include filename
 - Not available on all versions

Shell environment variables are available in a makefile as macros, for example:

```
FILE = ${HOME}\myprog.c
```

They can also usefully be set at the command line, which enables control of the make at runtime without having to edit the makefile, more of this later. Here is an example:

```
$make myprog BIN=/user1/proj/bin
```

This could then be used in the makefile to specify the directory for the final executable:

```
cc -o ${BIN}\myprog ${OBJS}
```

There are many built-in macros, here are some of the more common ones:

CC	Name of the C compiler
CFLAGSC	compiler flags, usually -O
LD	Name of the linker
MAKE	Name of the make utility
USER	Current user name
\$@	Name of the current target (or library name)
\$?	Prerequisites younger than the current target

To get a list for your version of **make**, run a full trace on a null file:

```
$make -p -f/dev/null | more
```

Macro Examples...

```
OBJS=myprog.o initial.o menu.o utils.o
LIBS=-lcurses
MYFLAGS=-c -d
myprog: ${OBJS}
    ${CC} ${CFLAGS} $@ ${OBJS} ${LIBS}
myprog.o: myprog.c common.h utils.h menu.h gameunit.h
    ${CC} ${MYFLAGS} myprog.c
initial.o: initial.c common.h
    ${CC} ${MYFLAGS} initial.c
menu.o: menu.c common.h utils.h menu.h
    ${CC} ${MYFLAGS} menu.c
utils.o: utils.c common.h utils.h
    ${CC} ${MYFLAGS} utils.c
```

You may wish to compare this slide with the one entitled "Dependency Lines", four slides back. We have reduced the amount of repeated text, making maintenance easier.

By convention we define our macros at the beginning of the file, this makes them easy to find and maintain.

The macro OBJS is traditionally used to give a list of the object files required to make to main module. We are also using LIBS for any additional libraries, and MYFLAGS for some C compiler flags.

In the command line for myprog, we have used the default CFLAGS and used the \$@ macro to denote the current target.

Macro String Substitution

- Search for a source string and replace

Syntax:

`${Macro-name:string1=string2}`

Replace each occurrence of *string1* with *string2* provided it is at the end of a word.

What do we get?

```
OBJ = fred.o jim.o myprog
ONE = ${OBJ:.o=.c}
TWO = ${OBJ:prog=text}
THR = ${OBJ:fred=jim}
```

Macro string substitution is a powerful feature which enables us to create a new text string from an old one, substituting sub-strings.

In the example above we get:

```
ONE: fred.c jim.c myprog
TWO: fred.o jim.o mytext
THR: fred.o jim.o myprog
```

There is no change in `${THR}` because the string `fred` does not occur preceding a `<TAB>`, `<SPACE>` or `<NL>`.

NOTE: Older BSD based versions may not include this feature.

Suffix Rules

- **Give meaning to a filename suffix (.extension)**
 - Significant suffixes are listed in suffix rules
 - Assumes that target and dependent files have the same prefix
- **Common default suffix rule:**

How to make a .o file from a .c

```
.SUFFIXES: .o .c
```

```
.c.o:
```

```
    ${CC} ${CFLAGS} -c $<
```

.c file is the prerequisite, .o file is the target

\$< is just like \$?, but only used in suffix rules

Certain filename suffixes (also known as extensions) and their relationships are known by the **make** utility. For example, suffix rules mean that **make** "knows" that compiling `myprog.c` will produce `myprog.o`, and `myprog.java` produces `myprog.class`.

Significant suffixes are defined using the `.SUFFIXES` macro, which is normally already set-up by default (`make -p` to check). Defining a value, as in the example above, will append new suffixes to the list (`.SUFFIXES:` with no values will clear out the defaults - not usually desirable).

The syntax for defining suffix rules is slightly different to defining dependency lines. The suffixes come in pairs, the prerequisite followed by the target, *with no intervening spaces*.

Commands

- **Executed as if from the shell**
 - must be prefixed by a <TAB> character
 - a newline terminates each command
- **Each command line is executed separately:**

```
cd bin
rm *
```

**is not the same
as:**

```
cd bin;rm *
or
cd bin;\
rm *
```

- **An error will stop make**
Unless prefixed with a hyphen -

The commands are usually executed by the Bourne shell, but most versions allow this to be altered with the SHELL macro. Any shell command may be executed, even "if" statements, but remember that a newline terminates the command, so you must "escape" any imbedded in the command, for example:

```
if test ! -x ${EXEC} \
then \
    echo "It failed!" \
fi \
```

Normally an error (non-zero return) in a command will halt **make**, which is usually what you want. This may be overridden for individual commands by prefixing them with a hyphen, for example:

```
test:
    echo "Executing badprog"
    - badprog
    echo "Continued"

$ make -s test

make: badprog: Command not found
make: [test] Error 127 (ignored)
Continued
```

Most versions of **make** allow all errors to be ignored by the use of the -i command line switch. To move onto the next target after an error, use -k (see man pages).

Forcing a Rebuild

- **Not often required**
 - After date/time stamps have been updated (cp, ftp, etc.)
- **Brute force**
 - Remove executables and objects

```
rebuild: tidy
    make myprog
tidy:
    - rm -f myprog *.o
```

- **Trick make with a dummy target**
 - A non-existent prerequisite is always more up to date than its target

```
rebuild:
    make myprog FRC=forced_build
forced_build:
myprog: ${FRC}
```

There are some occasions when a rebuild may be necessary, and programmers have a number of methods of achieving this.

The safest is to fool **make** with a dummy target, in this example it is named `forced_build` (it could be anything which is unique). This target will be ignored unless we specifically invoke it, which is the job of the `rebuild` target. The command for this is a (recursive) call to **make**, setting the `FRC` macro, again it could be any macro, but `FRC` is used by most people.

We now exploit a side effect of the rules governing targets: **a non-existent prerequisite is *always* more up to date than its target**. We have made `force_build` a prerequisite of `myprog`, and (since it does not exist) it is more up to date than `myprog`, so `myprog` is rebuilt !

The command line to invoke this would be:

```
$ make rebuild
```

Maintaining Header Files

- **Not automatic!**
 - If a header file changes, and a program's dependencies do not include it, then it will not get recompiled.
- **Use the -M (or -MM) compiler option to create a list of dependent header files**
 - Not supported by all compilers
- **Append this list to a "new" makefile**

Keeping header file dependencies up to date can be a particular problem. Fortunately, most Unix C (and C++) compilers support the -M command line option, which outputs a list of dependant header files, pre-fixed by the name of the object. This output is specifically designed for use by **make**.

An example follows.

If your compiler does not offer the -M option, and you have to write your own utility, remember that the job is more complex than it first appears. Header files can be nested (included from other files), `#include` statements may be commented out, or may be subject to other pre-processor statements, like `#if` or `#ifdef`. You may also have to ignore system header files, like `<stdio.h>` (that is what the -MM option does in the GNU compiler).

Maintaining Header Files: Example

```
OBJS = myprog.o initial.o menu.o utils.o
SOURCES = ${OBJS:.o=.c}
LIBS = -lcurses
myprog: ${FRC}
    ${MAKE} "CFLAGS=${CFLAGS}" "FRC=${FRC}" makefile
    ${MAKE} "CFLAGS=${CFLAGS}" "FRC=${FRC}" ${OBJS}
    ${CC} -o $@ $(OBJS) $(LIBS)
makefile: ${FRC}
    rm -f $@
    cp build.root $@    # where build.root is the current file name
    echo '# Automatically generated header file dependencies' >> $@
    # Note: -MM is specific to the GNU gcc compiler (others use -M)
    ${CC} -MM ${SOURCES} >> $@
    chmod -w $@
rebuild:
```

The file above, which (for once) we should not call "makefile", will maintain and build our myprog application. Let us say we have named the file **build.root**.

```
$ make -f build.root
```

will start at the first target it finds, myprog,, which calls **make** on the makefile target. That will copy build.root to makefile (cp build.root \$@), then execute the C compiler for all the current source files using the -MM flag (for the GNU gcc compiler). Standard output will be appended to the new makefile, after the rebuild: line. Strictly speaking the output is from the pre-processor (rather than the compiler), and is formatted specifically for this task. It looks like this:

```
myprog.o: myprog.c common.h utils.h menu.h gameunit.h
```

Assuming that worked (remember **make** will stop on error), then **make** is called again, this time to build the objects.

The rebuild target is used to update the makefile, with:

```
$ make -f build.root makefile FRC=rebuild
```

Summary

- **Make has more power than you usually need**
- **It is not confined to building C applications**
- **A makefile consists of:**
 - Macros
 - Dependency lines
 - Suffix rules
 - Commands
- **Some thought is required for maintaining #include files**
- **Versions of make differ**

