Make
A brief tutorial
August 28, 2019

1 Introduction

The *make* utility is used to manage files. That is a broad and perhaps confusing statement but accurate. It is more than just a handy way to compile programs but that is what most people equate it to. Well, we will go with that because its really all you want to hear about.

The whole point of *make* is to do something if the "target" (usually some file) is out-of-date. That means that *make* has to be able to decide what out-of-date means. We supply a list of "prerequisites" (again files, sometimes called dependencies) for each target. This list can be empty but usually is not. If the target is not up-to-date, then *make* executes the "recipe," a set of commands to be passed to the shell, that has been given for this target. This `target + prerequisites + recipe` is called a "rule."

When *make* runs, it first checks to see if the target exists. If not, it knows it must execute the recipe it has been given. But before it executes anything, it also checks the prerequisites. If some prerequisite exists but has no rule, it is considered up-to-date. If there is a rule, that rule is checked to make sure the prerequisite is up-to-date, and so on.

Any prerequisites that is out-of-date is "built" (its recipe is executed). Once all the prerequisites are determined to be up-to-date, then the recipe for the original target is executed and that target is built. When *make* is executed as just

```
$> make
```

the first rule is one that is used. If the user has a complex file of rules, she can specify which one used by giving the name as an argument to make:

```
$> make secondary
```

The file containing the rules can be named anything but *make* has a couple of names it looks for by default. First is looks in the current directory for "makefile." If that file is not found, it looks for "`Makefile` ." Depending on the version of *make* being used there may be other default file names searched for. If all those are exhausted, *make* uses built-in rules. You should never depend on that being what you want to happen. If you do not like any of these names or want a special version that is only used occasionally, you can use the **-f** switch an supply any file name.

```
$> make -f tmake
```

would use the file named "tmake" as the input. As an aside, I use "`Makefile` " as the name because in my environment file names are sorted in ASCII order and that puts capital letters are the beginning of a file listing.

2 Targets and Prerequisites

The first thing one must decide on when writing a `Makefile` is a target or set of targets. You can find a huge number of online tutorials and guides to give examples, and most of these focus on compiling C programs (even if they are using a C++ compiler). We will begin with a straight-forward example.

A target is a filename. Usually that filename is, or will be, an executable. For those unfamiliar with UXes, that means a program. For the purpose of this document, a program is file created by some compiler that can be executed on the current operating system. On UXes, there is NO NEED for any file extension on an executable because file extensions were intended to provide information to the user, not to software.

Then if we want to create a program called **first_test**, we begin a line, at the left margin, with

```
first_test:
```

The colon is required, and starting at the left margin keeps any of the various versions of *make* happy. Do not assume that fancy options and things that work on one O/S that allow flexibility or sloppiness will work on another.

Now that we have the target, we will add prerequisites. On the same line, after the colon, add a space separated list of files that must exist and be current in order to create **first_test**. Perhaps something like

```
first_test:  first_test.cpp locals.hpp
```

Remember that when *make* runs, it will try to determine if **first_test** is up-to-date using an algorithm similar to this.

1. If the file exists, record the file's last modification time.
2. Check the modification time of the first prerequisite.
3. Does the prerequisite exist? Then record that modification time. Otherwise note that it does not exist.
4. Is the prerequisite a target?
5. If so is it up-to-date? How? Start at #2.

6. If it is up-to-date (or is NOT a target), go on to the next prerequisite.
7. Continue until all prerequisites have been checked.
8. If the modification time of the target is older than any of the times of the prerequisites or the target does not exist, execute the recipe.
9. If there is no recipe, give an error and quit.

   ```
   make:  *** No rule to make target ‘t1.cpp’, needed by ‘prog’.  Stop.
   ```

You should have noted that this is a recursive operation. Fortunately, we never get so complex that *make* cannot process the `Makefile` in a very short time.

For most targets that are used in programming courses, the number of prerequisites is small. But sometimes we have more than will fit on a single line. Most editors used for programming on UXes understand the format of `Makefiles` but word wrap becomes a problem. If the editor actually puts in a newline, then an error has been inserted. *make* assumes that if the line after the target is not blank then it is part of the recipe and must start with **tab** character. If it does not begin with a tab *make* will print an error message and exit.

```
Makefile :2:  *** missing separator.  Stop.
```

To prevent this, and allow the user have a large number of prerequisites make allows lines to be “continued” by *escaping* the newline. That means using \ before typing the newline.

```
first_test:  first_test.cpp functions.cpp\
        locals.hpp
```

You can use any amount of whitespace in the prerequisites, including tab characters. And there can be any number of escaped newlines in the list.

3 Recipes

Now that we have a have a target and its prerequisites, we need to put in the commands to create that program. *make* at the lines immediately following the prerequisites. Everyone that begins with a **tab** character is passed to user’s shell to be executed. What does that mean? It means that *make* will cause the recipe lines to given to shell just like what would happen if you were typing in the terminal. For instance if you were compiling the trusty “Hello, World!” program you might use

```
$> g++ -std=c++11 hw.cpp -o hello
```

to generate the executable.

Here is what that could look like in a `Makefile` . The lines that start with a # symbol are comments. Most versions of make allow comments to begin anywhere but they end at the end of the line, like C++ // comments.

```
# Makefile
# Kim Buckner
# Aug. 35, 2091
#
# Example makefile for compiling hello world

hello: hw.cpp
	g++ -std=c++11 hw.cpp -o hello
```

Now you are saying, "why should I create a `Makefile` when I could just type the command at the prompt?" Good question. If you are going to do this once, that would be fine. But imagine that you have been been working on this really obnoxious programming assignment for two weeks. You keep adding files to the project, keep rewriting the files you have, and need to continually test the changes. Trust me, typing something like

```
$> g++ -std=c++11 -ggdb -Wall prog1.cpp prog1.hpp funcs.cpp funcs.hpp
locals.hpp -lm -lpthreads
```

more than once correctly is difficult. Typing it correctly 15 times in two hours at 3am is impossible. On the other hand

```
$> make
```

you can handle, even at 3am.

If there are multiple commands needed for a recipe, simply put each on a separate line. Make sure that each line begins with a tab character. One of the problems that occurs when using a mouse to copy `Makefile` recipes is how the mouse handles tabs. They almost NEVER copy the actual tab character. Normal operation is to copy spaces equivalent to a tab. *make* REQUIRES tab characters. Eight spaces is not the same. You will look at your `Makefile` , see that the format appears to be correct, test it and get that format error message.

```
Makefile :2:  *** missing separator.  Stop.
```

Simply remove the spaces and put in tabs, save the changes, and try again.

When processing a recipe, *make* causes each command line to be "echoed" to the terminal. If you like to see what is happening, like I do, then good. If, however, there is some reason to suppress this action, precede the command line with an at (@) symbol. Why would one like to do this? Suppose that the programmer wants to print some text out. What does not really matter. But the normal way to do this is use the “echo” utility. Try the following `Makefile` .

```
#
# Kim Buckner
# Aug. 35, 2091
#
# Example of suppressing some output

prog : start stop
        echo done
        echo works

start:
        touch start

stop:
        touch stop
```

In case you are wonder, “touch” is a utility that makes the modification time of a file **now**. And if the file does not exist, it is made to exist with a size of 0 bytes. Executing *make* with this `Makefile` (at least the first time) should result in this

being printed to the terminal:

```
$> make
touch stop
touch start
echo done
done
echo works
works
$>
```

Notice that the echoing of the echo commands is pretty annoying. Change that file to be

```
#
# Kim Buckner
# Aug. 35, 2091
#
# Example of suppressing some output

prog : start stop
        @echo done
        @echo works

start:
        touch start

stop:
        touch stop
```

and now executing it gives

```
$> make
done
works
$>
```

Much better.

Now you should be able to write a simple `Makefile` . Of course, `Makefiles` can rapidly become very messy. And the more you use them the more complex they get. Next we will discuss some of the things that can clean up (certainly not simplify) `Makefiles` .

4 Variables

In `Makefiles` it is often the case that we use strings in commands or prerequisites repetitively. Or we want to customize `Makefiles` so that they are easy to change from one installation to another. Or we would have to have a generic format that is easy to adapt to multiple situations. For these cases *make* supports the use of variables. This discussion will focus on simple versions of these. GNU *make* , MS *nmake*, *cmake*, *imake*, Solaris *make* , BSD *make* all do fancy stuff, but there is little in the way of consistency.

A variable is simply a string. It must begin with a letter. The convention is that *make* variables are all upper case. This improves readability and understanding of the `Makefile`

All versions of *make* that I have encountered have predefined variables with preset values. Do NOT trust that these values are the ones that you want. But, the use of the "standard" variable names helps with portability. Some examples from the GNU *make* manual.

Name	**Use**	**Default**
AR	Archive-maintaining program	’ar’
AS	Program for compiling assembly files	’as’
CC	Program for compiling C programs	’cc’
CXX	Program for compiling C++ programs	’g++’
CPP	Program for running the C preprocessor, with results to standard output	’$(CC) -E’
FC	Program for compiling or preprocessing FORTRAN and Ratfor programs	’f77’
LEX	Program to use to turn Lex grammars into source code	’lex’
YACC	Program to use to turn Yacc grammars into source code	’yacc’
LINT	Program to use to run lint on source code	’lint’
TEX	Program to make TeX DVI files from TeX source	’tex’
RM	Command to remove a file	’rm -f’
Here are variables whose values are additional arguments for the programs above. The values for all of these is the empty string, unless otherwise noted.		
ARFLAGS	Flags to give the archive-maintaining program	’rv’
ASFLAGS	Extra flags to give to the assembler (when explicitly invoked on a ’.s’ or ’.S’ file).	
CFLAGS	Extra flags to give to the C compiler.	
CXXFLAGS	Extra flags to give to the C++ compiler.	
CPPFLAGS	Extra flags to give to the C preprocessor and programs that use it (the C and FORTRAN compilers).	
FFLAGS	Extra flags to give to the FORTRAN compiler.	
LDFLAGS	Extra flags to give to compilers when they are supposed to invoke the linker, ’ld’, such as -L. Libraries (-lfoo) should be added to the LDLIBS variable instead.	
LDLIBS	Library flags or names given to compilers when they are supposed to invoke the linker, ’ld’. LOADLIBES is a deprecated (but still supported) alternative to LDLIBS. Non-library linker flags, such as -L, should go in the LDFLAGS variable.	
LFLAGS	Extra flags to give to Lex.	
YFLAGS	Extra flags to give to Yacc.	
LINTFLAGS	Extra flags to give to lint.	

As you can see there are a number of these preset. But, for instance, the C compiler on modern Linux is usually *gcc* not *cc*. Most systems have *flex* installed instead of *lex* and those also use *bison* instead of *yacc*. It is important to be sure of the value of the used variables. I normally assume that the defaults are incorrect and redefine the ones I use in the `Makefile` .

Variables can represent lists of file names, options to pass to compilers, programs to run, directories to look in for source files, directories to write output in, or anything else you can imagine. A variable name may be any sequence of characters not containing ':', '#', '=', or whitespace. However, variable names containing characters other than letters, numbers, and underscores should be considered carefully, as in some shells they cannot be passed through the environment to a sub-make and in versions other than GNU those special characters may not be supported at all. As variables are expanded at the point of use, they are often referred to as "macros."

Some variable examples:

```
# Makefile
# Kim Buckner
# Aug. 35, 2091
#
# Example makefile for compiling hello world
# Has variables
CXX=g++
CXXFLAGS=-std=c++11 -Wall -ggdb

hello: hw.cpp
        g++ -std=c++11 hw.cpp -o hello
```

Note here that are no spaces between the variable name, the assignment operator (=) and the start of the right-hand string. Some older versions of *make* did not allow spaces before and after the =, newer versions may or may not care. Like with the prerequisite list, you can escape a newline to continue the right-hand side on multiple lines. Make sure that the variable names start at the left margin.

Now that we have defined the variables, we need to use them. That is quite simple. In a prerequisite list or a recipe, simply surround the variable name with parentheses, (), or braces, {}, and precede the whole thing with a dollar sign. Unfortunately, leaving out the parentheses or braces is not a syntactic error. *make* just assumes that the variable is a single letter. Here is the `Makefile` with the variables used. preceded with a dollar sign, $, and

```
# Makefile
# Kim Buckner
# Aug. 35, 2091
#
# Example makefile for compiling hello world
# Uses variables
CXX=g++
CXXFLAGS=-std=c++11 -Wall -ggdb

hello: hw.cpp
        ${CXX} ${CXXFLAGS} hw.cpp -o hello
```

Here is the output generated by using this `Makefile` . You can see by the echoed command line that the variables expanded correctly.

```
$> make
g++ -std=c++11 -Wall -ggdb hw.cpp -o hello
$>
```

If the `Makefile` is changed to remove the braces around the variable names

```
# Makefile
# Kim Buckner
# Aug. 35, 2091
#
# Example makefile for compiling hello world
# Uses variables incorrectly!!!
CXX=g++
CXXFLAGS=-std=c++11 -Wall -ggdb

hello: hw.cpp
        $CXX $CXXFLAGS hw.cpp -o hello
```

and then executed, we get

```
$>make
XX XXFLAGS hw.cpp -o hello
make:  XX: Command not found
make:  *** [hello] Error 127
$>
```

5 Special targets, rules, variables

5.1 Special Targets

One of the things that needs to be understood about *make* is that, by default, only the **first** rule is automatically executed when run as

```
$> make
```

That means that unless some prerequisite needs to be checked, no other rules are even examined. But then what if you have a number of rules, or a number of files, or...

First, you can execute any rule, with the same caveat as above, by doing

```
$> make target
```

and you can list as many targets as you like. This is great if you have some specialized rule that you may want only executed occasionally. But it is annoying to have to do this every time you use *make* and need to build, or at least check, a number of targets. For that reason, *make* allows targets that may never exist, targets without prerequisites, and rules without recipes. Here is an example to discuss. (next page please)

```
# Makefile
# Kim Buckner
# Aug. 35, 2091
#
# Example makefile for compiling hello world
# Phony targets and multiple desired results.
CXX=g++
CXXFLAGS=-std=c++11 -Wall -ggdb
RM=/bin/rm -f

all: hello bye howdy

hello: hw.cpp
        ${CXX} ${CXXFLAGS} hw.cpp -o hello

bye: bye.cpp
        ${CXX} ${CXXFLAGS} bye.cpp -o bye

howdy: how.cpp
        ${CXX} ${CXXFLAGS} how.cpp -o howdy

clean:
        \${RM} bye hello howdy
```

The first target and the last target are what GNU *make* considers “phony” targets. That is a target which is never meant to exist. But *make* will treat them like a regular target, look in the filesystem to determine the last modification time. Of course it does not exist so the rule is always executed. In this case fist the rule for “hello” will be checked and if “hello” is out-of-date it will be built, then “bye,” then “howdy.” And all you have to use is *make* with no arguments and the rule “all” is executed. The word “all” has no special meaning, it has just become one of the those habits that programmers use.

The last target, “clean,” is similar but you must use

```
$> make clean
```

to use it. This is another “everyone does it” kind of rules. Another you might that is similar is “tidy”; used when you want to delete or move some results of the compilation (like object files) but leave the executables.

In addition to these types of targets, GNU *make* has a special target **.PHONY**. This one is designed to give make the list of the all the phony targets so that the program will never check the filesystem to see if they exist. Depending on the situation, this could reduce time. The thing most useful about it though, is that it provides documentation in the `Makefile` that other programmers can easily understand.

```
# Makefile
# Kim Buckner
# Aug. 35, 2091
#
# Example makefile for compiling hello world
# Phony targets and multiple desired results.
CXX=g++
CXXFLAGS=-std=c++11 -Wall -ggdb
RM=/bin/rm -f

.PHONY: all clean

all: hello bye howdy

hello: hw.cpp
        ${CXX} ${CXXFLAGS} hw.cpp -o hello

bye: bye.cpp
        ${CXX} ${CXXFLAGS} bye.cpp -o bye

howdy: how.cpp
        ${CXX} ${CXXFLAGS} how.cpp -o howdy

clean:
        \${RM} bye hello howdy
```

5.2 Pattern and Implicit Rules

make may use "implicit" rules when trying to figure out how to build some target. These are more like generic rules to certain types of targets/files. *make* has a number of these built in. That is the primary reason for all those predefined variables we saw earlier. This can result in some rather strange errors, for instance you might expect a "...no rule to make..." error and instead get "'cc': command not found."

You define an implicit rule by writing a pattern rule. A pattern rule looks like an ordinary rule, except that its target contains the character '%' (exactly one of them). The target is considered a pattern for matching file names; the '%' can match any nonempty substring, while other characters match only themselves. The prerequisites likewise use '%' to show how their names relate to the target name.

Thus, a pattern rule '%.o : %.c' says how to make any file *stem*.o from another file *stem*.c.

Note that expansion using '%' in pattern rules occurs after any variable or function expansions, which take place when the makefile is read.

A rule to make any object (.o) file from a C++ source code file might be

```
%.o : %.cpp
        $(CXX) -c $(CXXFLAGS) $< -o $@
```

And here is a rule that has two targets.

```
%.tab.c %tab.h: %.y
        $(YACC) -d $<
```

Note that you **cannot** add other prerequisites unless

- they all have the same *stem*,
- they exist or can be built,
- and they are needed for all possible targets.

5.3 Automatic Variables

Previous we have discussed variables that the programmer defines in the `Makefile` . But note in the two previous examples that there are variables which do not match our discussion. These special variables are automatically assigned values by *make* as it executes the rules. Here are a listing of the ones used by GNU *make* that are generally portable. There are a number of others in reference manual but they are not generally useful in undergraduate courses.

`$@`	The file name of the target of the rule. If the target is an archive member, then '`$@`' is the name of the archive file. In a pattern rule that has multiple targets (see Introduction to Pattern Rules), '`$@`' is the name of whichever target caused the rule's recipe to be run.
`$%`	The target member name, when the target is an archive member. See Archives. For example, if the target is foo.a(bar.o) then '`$%`' is bar.o and '`$@`' is foo.a. '`$%`' is empty when the target is not an archive member.
`$<`	The name of the first prerequisite. If the target got its recipe from an implicit rule, this will be the first prerequisite added by the implicit rule (see Implicit Rules).
`$?`	The names of all the prerequisites that are newer than the target, with spaces between them. For prerequisites which are archive members, only the named member is used (see Archives).

`$^`	The names of all the prerequisites, with spaces between them. For prerequisites which are archive members, only the named member is used (see Archives). A target has only one prerequisite on each other file it depends on, no matter how many times each file is listed as a prerequisite. So if you list a prerequisite more than once for a target, the value of `$^` contains just one copy of the name. This list does not contain any of the order-only prerequisites; for those see the '`$|`' variable, below.
`$+`	This is like '`$^`', but prerequisites listed more than once are duplicated in the order they were listed in the makefile. This is primarily useful for use in linking commands where it is meaningful to repeat library file names in a particular order.
`$|`	The names of all the order-only prerequisites, with spaces between them.
`$*`	The stem with which an implicit rule matches (see How Patterns Match). If the target is `dir/a.foo.b` and the target pattern is `a.%.b` then the stem is `dir/foo`. The stem is useful for constructing names of related files. In a static pattern rule, the stem is part of the file name that matched the '%' in the target pattern. In an explicit rule, there is no stem; so '`$*`' cannot be determined in that way. Instead, if the target name ends with a recognized suffix (see Old-Fashioned Suffix Rules), '`$*`' is set to the target name minus the suffix. For example, if the target name is '`foo.c`', then '`$*`' is set to '`foo`', since '`.c`' is a suffix. GNU make does this bizarre thing only for compatibility with other implementations of make. You should generally avoid using '`$*`' except in implicit rules or static pattern rules. If the target name in an explicit rule does not end with a recognized suffix, '`$*`' is set to the empty string for that rule.

Automatic variables are very handy. However, you should get so enamored with them that build errors into the make file.

6 Last Comments

make is a very handy utility. It will certainly make your life easier when you get used to using it. However, You must be warned about working on multiple machines at one. For instance, if you are editing a file on your home computer, uploading it to one of the department machines, then compiling it, you may get an error like

```
make:  Warning:  File ‘hw.cpp’ has modification time 200 s in the future
g++ -std=c++11 -Wall -ggdb hw.cpp -o hello
make:  warning:  Clock skew detected.  Your build may be incomplete.
```

simply because the clocks on the separate machines are not in sync.