Introduction to Make

Last modified: Fri Nov 27 09:56:46 2020

You can buy me a coffee, please

I would appreciate it if you occasionally buy me a coffee *A note to readers - clicking on a header will navigate to and from the index.*

Table Of Contents

What is make? What does the command do?

The original Make documentation by Feldman had a wonderful description of the make utility. When editing code - there are four steps:

1. Think
2. Edit
3. Make
4. Test

In other words, wouldn't it be nice to have a program that takes care of all of the nit-picky details in step 3 for you? *Make* is an excellent program that can help you compile programs.

But suppose you are just writing shell scripts? *Make* isn't really needed, right?

Well, please allow me to convince you otherwise. I use *make* all the time with shell scripts. And the more complicated the process, the more likely I will create a *Makefile* for the process.

Why use make for shell scripts?

Some of you might say to yourself “I just write shell scripts. Why should I use *make*?” There are many reasons I use *make* for shell scripts. Here are some of my reasons:

- My work flow has “steps”, or an order of execution.
- The shell scripts have multiple commands, options, and/or arguments.
- I want to document what I did.
- I am operating on a remote machine and don't have access to a GUI environment.
- I work with a team, and I want to share project-related shell commands without having to have other team members install my shell scripts and aliases.
- I work with several projects, but want to have a uniform management technique.
- I don't want to, or can't, install additionals tools.

Whenever I create a directory for a new project, even if it's a handful of shell scripts, I create a *Makefile*. The best documentation is the source code, and *make* is a great place to start. And if I visit an old project, looking at the *Makefile* refreshes my memory of **what** I did, **how** I used the scripts, and **why** I did certain things. After all, once you find the optimum command and arguments, the steps in a *Makefile* can “immortalize” the exact arguments and sequence of operations.

Sometimes I type in long lines that execute a complex shell script. I want to document it, but I don't want to create a full-blown shell script, especially if it's just a single line. I could make it an alias, but repeating this a hundred times makes an alias file unwieldy. I could put it in a text file to document/remember it, but I also want to execute it. I could write a shell script, but I want to make it easy to share with team members. I could create a directory called *scripts* tha contains all of my recipes, but then I have to get team members to install this in their environment.

Or, I could use *make*. (rimshot)

Debugging shell scripts without make

Let's examine a simple programming work flow without *make*.

I could just type in a shell script in one line, but if it's complicated, and I might want to do it again, I should keep it handy by putting it into a shell script.

Normally I have two windows open - one is editing the file, and the other is “using” the file. For example - I could be editing a awk, bash, sed or python script using **My Favorite Editor™**. And the other window is a terminal window running a shell.

I edit the file in one window, then save the edit. I then go to the other window and “run” the program, typically by typin an up arrow or “!!” history command. Or I use command line editing to make changes. Standard operating procedure.

Sometimes you have to switch between two or more commands in the shell window. This is easy to do with a shell that understands your history of commands. However, your work flow might have steps. And you might forget a step, or perhaps do them in the wrong sequence. Get it wrong and your result won't be right.

Perhaps the exact shell command you type is very long and complicated, and you have to look at a manual page to remember what is needed. Or suppose you want to use various combination of commands as you experiment with the results, such as changing an option for one of the arguments. This can be complicated in a multi-step process.

And then there's the classic problem of executing your program or script, but you forgot to tell the editor to save the changes. We've all done this.

These are the problems *make* is made for.

> System Adminitrators - You want to look at log files and look for patterns and odd activity? Keep reading.

Let's get started.

The default input file to make - Makefile

If you simply type the command

```
make
```

You will get the error “make: Nothing to be done for 'all'.”. This is because you haven't provided “make” with enough information, and since there is no direct brain input (yet), it normally gets this information from a file with the fiendish name of “makefile”.

To be more precise, the “make” command first looks for the file with the name “makefile” and if this file does not exist, then looks for the file “Makefile”. Note the second choice starts with a capital letter.

> Most of the time, when programmers create a project, they provide the file “Makefile” instead of “makefile” for a couple of reasons. Normally, the *ls* command will sort filenames so that files that start with a capital letters are shown before those beginning with lower-case letters. Therefore this file is often listed before the others. This is also why people provide a file called “README” - to make it more noticeable.
> The second reason people use “Makefile” over “makefile” is to make it easier for another person to override the default behavior. If you get a source package with a “Makefile” you want to change, (e.g. to change the destination directory) just copy “Makefile” to “makefile” and edit that one. This keeps the original one around as a backup. It also allows you to test changes easily.

Sounds like a good convention. Let's begin.

My First Makefile

Makefile and targets

Let's create a very simple *Makefile*. Edit a file and give it the name “Makefile”. The file should contain the following:

```
all:
        ./script1
```

The syntax is important, and a little fussy. Let me provide a more precise description of the syntax, which in general terms is:

```
<target>: <prerequisites>
<tab>   <recipe>
<tab>   ...
```

Let's break it down.

The word “all” is the name of a target. The recipe in our makefile is “./script1” The “prerequisites” is optional. I'll cover that later. For now, the recipe “makes” the target, or in our case, you execute “./script1” to make “all”. Or to put it another way, when you type

```
make all
```

the *make* program executes “./script1” for you.

Congratulations on your first *makefile*, by the way. Yeah, I know. Underwhelming. Let's build up on this.

As the specifications suggest, you can have more than one recipe for a target. Each recipe is executed by passing the line to the shell for execution. However, and here is the fussy part, there **must** be a tab character before each recipe. *Note that you must use a tab character, and not space characters, before recipes.*

If you forget this, *make* will report an error such as:

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

In this example, *make* reports the filename, and the line number of the error - in this case “2”.

If you are using an editor that changes tabs to spaces, this will cause your *Makefile* to break. And if you cut and paste my examples, you may have to change leading spaces into a tab.

With the basics out of the way, we are ready to use *make* in a more realistic sense. Suppose you have three independent scripts, named script1, script2, and script3.

```
all: step1 step2 step3
step1:
        ./script1
step2:
        ./script2
step3:
        ./script3
```

I've added three new targets and recipes. You can type “make step1”, etc. to execute just one of the programs. Notic however, that we added step1, step2 and step3 as a prerequisite to the “all” target. If you type “make all” - *make* build all three targets. This is because the target “all” has three sub-targets - step1, step2 and step3.

> If you don't give *make* a target, it builds the first one in the file. It's very common to name the first target “all” so *make* builds everything by default.

If for example, you are a system administrator that has to check several log files, and you perform the same actions each time, you can put them all into a Makefile.

You can use *make* to help you remember complex recipes that are project related, and be consistent in operations. For example, suppose to had to generate a report for multiple projects in multiple directories. The shell commands may be different, but you can use “make report” - that is - the same command - for all projects as long as you adjust the *Makefile* in each directory.

Or suppose you had to do a log or data analysis for several projects/programs. You could use the command “make logs” for each project, without having to install or share scripts between teammates.

And there are times when you have to experiment with a long and complex shell command just to get something working, and once it's done, it's done. Yes, you can forget about it, but sometimes it's a good idea to capture these long and complicated commands, to make it easier for you to re-use it the next time. For example, I've used a “wget” command with 8 to 10 command-line options, but I may not remember these exact commands six months later.

So let's learn to master the *Makefile* and add some dependencies to the recipes.

Dependencies and execution order in makefiles

Let's assume that these programs/scripts have to be executed in a certain order, in this case, step1, step2 and step3 in that sequence. We can do this with a minor change to the earlier *makefile*:

```
all: step3
step1:
        ./script1
        touch step1
step2: step1
        ./script2
        touch step2
step3: step2
        ./script3
        touch step3
clean:
        rm  step1 step2 step3
```

Note that my “Makefile” creates three files named step1, step2, and step3 using the *touch* command. This updates th time stamp on a file, and if the file doesn't exist, it creates an empty file.

Also note that the *all* target just has *step3* as a prerequisite, or dependency. And if you look at *step3*, it has *step2* as a dependency. That is, before *script3* is executed, *step2* has to exist. And *step2* is created when *script2 is executed*. But that won't happen until *step1 is created*.

In other words, we've created not just a set of steps, but an explicit sequence or oder of operation.

Make understands these dependencies and keeps track of them for you. If you execute “make” it will perform the following commands

```
./script1
touch step1
./script2
touch step2
```

```
./script3
touch step3
```

However, if you execute *make* a second time, it will do nothing, because everything is already made. In otherwords, the prerequisites have been met.

The files step1, step2 and step3 have to be removed if you want to execute the three scripts again. That's why I added a target called “clean” that when executed, it removes these files. You can type “make clean;make” and the three scripts will be executed again.

> The “make clean” target is a common convention in *makefiles*.

Using input and output files as targets and dependencies

I used the files *step1* etc., as targets. This works, and is simple, but it can cause problems. These files can get out of sync with your scripts and programs. It's better to have targets and dependencies that are part of the flow of data.

We have three scripts above. Let's assume that they are normally piped together, like this:

```
./script1 <data.txt | ./script2 | ./script3 >script3.out
```

I'm going to modify the *makefile* to perform the above script, in steps:

```
all: script3.out
script3.out:  script2.out
        ./script3 <script2.out >script3.out
script2.out:  script1.out
        ./script2 <script1.out >script2.out
script1.out: data.txt
        ./script1 <data.txt >script1.out
clean:
        rm *.out
```

This eliminates the need for those “do-nothing” step files. Instead, we are using files than contain real data. If we type “make” the scriptram will execute

```
./script1 <data.txt >script1.out
./script2 <script1.out >script2.out
./script3 <script2.out >script3.out
```

This is closer to a real *makefile*, but there is an important feature missing. *Make* can deal with data files, but its real power is dealing with source code. In this case, if *script1* etc. is a script, we want *make* to realize that if the script changes, the output will as well. So a more useful *makefile* would look like:

```
all: script3.out
script3.out: ./script3 script2.out
        ./script3 <script2.out >script3.out
script2.out: ./script2 script1.out
        ./script2 <script1.out >script2.out
script1.out: ./script1 data.txt
        ./script1 <data.txt >script1.out
clean:
        rm *.out
```

If you look at the target “script3.out”, you will see two prerequisites now. As before, we have “script2.out”, but we added “script3” as well. That is, if you edit the script or change the input data - you have to rerun the script.

Now we're ready. This version will detect when the program (or script) changes, and when it does, it re-runs that step

Don't forget that your scripts might have additional dependencies. For example, if “script2” executed a *awk* script called “script2.awk”, you may want to add this as a dependency for the “script2.out” target. *Make* doesn't parse your files, it just looks at timestamps, so you have to add these dependencies yourself.

> While you could just pipe the three scripts together, this methodology is very handy when debugging - especially if you don't know which script has the bug, and if the scripts take a long time to execute. Not only does it just rerun the minimum steps after a script change, but it captures all the data that would normally be discarded if the scripts were piped together.

How does make know when to execute a recipe?

The *make* utility uses the time stamps and the recipes to determine when it has to regenerate a target. In the exampl above, it looks at the time stamps of the input file and the executable program/script to determine if any action is

needed. *Make* also has some build-in rules it uses, but we will cover that in a future tutorial.

If you are a system administrator, you can see that this can be useful, because “make” will detect if a log file changes and only generate new data if it does.

Integrating make with your editor

Integrating your editor with *make* can improve your efficiency a lot. How many times have you tried a code fix to discover that you forgot to save the changes? Instead, your editor will automatically save files for you, and even bette auto-guide you to the line in the file that has the error? *You want to do this, gang.* Let's go.

Using vi/vim and make

First of all, I have to apologize that this is just an intro into *vim/make* integration. To be honest, I used *vi* for about a decade before *vim 2.0* was released, and by then I switched to *Emacs*. Consequently, I never mastered the advance features of *vim*. But I'll do the best I can to get you started.

For you *vim* users, a first step you may want to do is to map a function key to the *make* command. Once example is t add this to your *~/.vimrc* file - in this case the F9 key:

```
" You want to save files when you execute make. You have two choices. Either:
" (1) set autowrite whenever you execute :make
set aw
" or (2) Press F8 to save all open files
:map <f9> :wa
" and to execute make, press the F9 key:
" Press F9 to execute make
:map <f9> :make
```

It true that binding a function key to make just saves a few steps, as you can always type “:make” in *vi*.

When you want to execute make, you do have to make sure your changes are saved to the file system. If you like option 2, you have to press the F8 function key. If you have option 1, by setting autowrite on, then this will happen automatically. Now, when you press the F9 function key, *vim* will execute the *make* command, allow you to specify th target and arguments, and wait for you to press the ENTER key. When this is done, *vim* will show you the results.

You can review the results of the compile using the “:copen” (abbreviation “:co”) command which opens up a quickfix window. If there are errors that *vim* can parse, you can press ENTER and jump to the different errors, and edit the file For example, if you have an error in your *Makefile*, you can use the quickfix window to jump to the line that caused th error. The command “:cclose” will close this window, and you can toggle this using the “:cw” command.

Vim allows you to defined the *makeprg* variable to execute make with command line options, or to use some progran other than make. I saw one example where someone changed *makeprg*to execute the java compiler. Personally, I think this is a bad idea. First of all, if a second person wishes to duplicate your steps, they have to modify their own *vim* startup file the same way. Second, they also have to use *vim* instead of their preferred editor. These options belong in a Makefile, which becomes documented as part of the source code, rather than hidden in a personal preference.

Using GNU Emacs with make

GNU Emacs has support for *make* built in. However, you may want to map a keystroke to the *make* command. I use the following to map “Control-C m” and “Control-C M” to *make*.

```
;;; map Control-C m and Control-C M to make
(global-set-key \C-cm" 'compile)
(global-set-key "\C-cM" 'compile)
```

Sometimes I have shift on by accident, so I've mapped both upper and lower case “M” to *make*.

When I press “Control-C m”, Emacs will ask me if I want to save any files that have not been saved. It then prompts me with the default *make* command with arguments. I can edit this and it will remember this for next time. Then Emac will launch the make command and pipe the results into a buffer called “*compilation*”. If this process takes a while, you can still edit your scripts and data while watching the results. The status is updated on the status line, which says “Compilation:exit” when *make* is finished. I find this handy for long-running jobs.

If you repeat the command, and it is still running, Emacs will ask you if you want to terminate the current compile. The command “Control-C Control-K” will kill the current compilation. I do this often when I realized I goofed and want to re-run the compile job.

Once the compilation is done, you can press “Control-X `”, which is *command-next-error*. This will read the errors in the compilation buffer, locate the first error, and go to the line in the file that caused the error. If you repeat this command, it will go to the next error, even if it's in a different file. Using this, you can quickly navigate through your errors and then recompile. This won't work very well in programs that don't generate errors that can be parsed.

If you are dealing with more than one “Makefile”, such as nested directories, or multiple projects, the *compile* command will run in the current directory of the file you are actively editing.

If you are using Emacs in graphics mode, and you edit a “Makefile”, the menubar will show current *Makefile* commands. I'll try to describe these in a later tutorial.

I wanted to make sure you understood the normal workflow. It's time to learn how to adjust your *Makefile* to be more a more productive user.

Variables in Makefiles

Make was an early program in the history of Unix systems, and the support for variables is unusual. Variables are indicated with a dollar sign, but the name of the variable is either a single letter, or a longer variable name surrounded by parentheses. Current versions of *make* also allow you to use curly braces as well. Some examples of variables in recipe are:

```
all:
        printf "variable B is $B\n"
        printf "variable Bee is $(Bee)\n"
        printf "variable Bee is also ${Bee}\n"
```

You have to be careful, because if you used “$Bee” in a make file, you are referring to the variable “$B” with the letters “ee” appended to it. So if variable $B had the value “Whoop”, then “$Bee” has the value “Whoopee”.

Consequently, I always use parentheses or curly braces (it doesn't make any difference which one you use) around variables, to make sure no one confuses “$(B)ee” with “$(Bee)”

There are special variables whose names are special characters. I'm not going to cover those yet.

Including a dollar sign $ in a Makefile

Since the dollar sign indicates a variable, what do you do if you want a dollar sign left alone? In this case, simply use “$$” to indicate a single dollar sign. For instance, if you wanted to use a regular expression containing a dollar sign in a *makefile*, you would need to double the dollar sign. Suppose you wanted to search for all lines that had a number at the last character, using the regular expression“"[0-9]$"”, a sample recipe would be:

```
grepthelines:
        grep '[0-9]$$' data.txt
```

Note that *make* interprets the lines and evaluated variables before it passes the results to your shell. The shell only sees a single “$”. Normally, meta-characters within Strong Quotes are left alone. But *make* processes the lines before it sends them to the shell.

Setting variables in Makefiles

There are two ways to set Makefile variables. I'm not talking about shell or environment variables.

Setting a variable inside a Makefile

Setting variables is simple; simply put them on a line (not starting with a TAB character), with an equals sign. Variable can refer to other variables. An example might be

```
# Example of a Makefile with variables
OUTFILES = script1.out script2.out script3.out
TMPFILES = script1.tmp script2.tmp script3.tmp
TEMPORARYFILES = $(OUTFILES) ${TMPFILES)
# Which programs am I going to install?
PROGS = script1 script2 script3
# Note I am using a shell environment variable here
INSTALL = /usr/local/bin
clean:
        rm $(TEMPORARYFILES)
```

```
install:
        cp -i ${PROGS} ${INSTALL}
```

The variable TEMPORARYFILES is the value of two other variables concatenated - as strings. It does make sense that *make* variables are string-based.

Setting makefile variables on the command line

In the last example, if you executed the command “make install”, *make* would copy files to the */usr/local/bin* directory. You can over-ride this on the command line. The syntax is

```
make target [variable=value ...]
```

For example, if you typed

```
make install INSTALL=~/bin
```

Then *make* would install the programs into ~/bin.

Evaluating Shell Variables with Makefile variables

When you define variables, the shell is used to evaluate the line before it passes the results to *make*. For example, you can use the following definitions:

```
OUTFILES = *.out
INSTALL = $${HOME}/bin
```

OUTFILES will be equal to all of the files that match the pattern “*.out” while the INSTALL variable is based on the environment variable HOME.

Don't confuse shell variables with Makefile variables

Note that I used “$${HOME}” in the previous example. The double dollar sign tells *make* to pass a single dollar sign t the shell, which uses it to get an environment variable.

Don't make the mistake of trying to set a variable in a recipe and assume it works the same as a definition. Here's is an example with both:

```
#Define a variable in a Makefile
A = 123
all:
        A=456; printf "A = ${A}\n";
        A=456; printf "A = $${A}\n";
        printf "A = $${A}\n";
```

This will print

```
% make
A=456; printf "A = 123\n";
A = 123
A=456; printf "A = ${A}\n";
A = 456
printf "A = ${A}\n";
A =
```

The first time printf is executed, A single $ is used, so the definition in the *Makefile* is used. The second time a double dollar sign is used, so the shell variable is used, and “456” is printed.

The third printf prints the shell variable “A” as an empty string, because it is undefined. Each line of the recipe is executed by a new shell. In this example, three shells are executed - one for each of the lines containing *printf*. And each shell has it's own view set of variables.

> Please note that when a recipe has multiple lines, each line is executed with its own shell. Variables set in one line are not passed to a second line. Also note that you can execute recipes which launch a background job by ending the recipe with an ampersand.

Errors when executing make

There are a few ways you can make a mistake, and have *make* complain.

Syntax errors in the Makefile

First of all, Make will complain about errors in a makefile. If you don't have a tab before a recipe, it will complain

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

If you ask it to make a target and it doesn't know how, it will tell you there is no rule. It can detect loops in dependancies as a few other errors.

Errors during execution

The second type of error occurs when *make* is executing other programs.

If you execute a complex series of steps or recipes, and an error occurs, *make* will stop. For example, in my earlier example:

```
all: script3.out
script3.out: ./script3 script2.out
        ./script3 <script2.out >script3.out
script2.out: ./script2 script1.out
        ./script2 <script1.out >script2.out
script1.out: ./script1 data.txt
        ./script1 <data.txt >script1.out
clean:
        rm *.out
```

If the script script1 has an error, then *make* will stop execution right after it tries to create “script1.out” If there are more than one recipe for a target, *make* will stop after the first error. If there were four lines (or recipes) to execute, and the third one aborts because of an error, the fourth line won't get executed. This is intuitive, because it's what you want to happen. But there are some subtle points. Suppose you had the following recipe:

```
test:
        false;true
        exit 1;exit 0
```

The “false” program is a standard Unix command that simple exits with an error. If it was the only command on that line, then *make* will halt execution. However, *make* asks the shell to execute several commands on the line, (“false” and “true”), and the last command didn't fail. So the shell tells *make* that the line executed successfully. This “ignores the previous error on the line. So *make* then executes the second line.

In this case, the shell executes “exit 1” directly, just like the *false* command does, but this time the shell - which is executing the programs, itself exits, which causes *make* to stop. The “exit 0” is never seen.

while loops in a Makefile

Sometimes you need to write a script that processes a file, and it only processes a single file at a time. Yet you need t repeat this for several files.

As an example, I wanted to search a directory for PDF files and calculate the SHA1 hash for each one. I used this to see if any PDF files changed, or new ones were created. I used the following recipe:

```
genpdfsums:
        ./FindPDFFiles | while IFS= read line;do sha1sum "$$line";done >PDFS.hashs
```

Note that I set IFS to an empty string in case any filename has a space as part of the name. This assumes each inpu line is a filename.

Line Continuation in Makefiles

Because *make* executes a shell, you can use the same conventions for line continuation as the shell. For example, here is a recipe I used when I examined the results of a *wget* command to extract URLs:

```
TRACE = cat
#TRACE = tee /tmp/trace.out
all_urls.out:  Makefile
        cat */wget.err | grep '^--' | \
```

```
        grep -v '(try:' | awk '{ print $$3 }'  | ${TRACE} |\
        grep -v '\.\(png\|gif\|jpg\)$$' | sed 's:?.*$$::' | grep -v '/$$' | sort | uniq >all_urls.ou
```

When my *makefiles* get complicated, I often put “Makefile” as a dependancy in the target. In the above example, if I edit the Makefile, then “all_urls.out” is out of date.

That is, my “source code” is my *Makefile*.

You may also notice I included an example of how I integrate *make* with shell script debugging. In this case I created variable called “TRACE” which I can use to debug my shell script. By changing the definition of “TRACE”, I can capture the output in the middle of a complex shell script.

Conditional execution in Makefiles

Sometimes you need to test for a condition and execute part of a shell script. The shell has the commands "||" and "&&" which can be used to execute simple flow control. The characters "&&" execute the rest of the line if the conditic is true. To test for a false condition, use "||".

Let's suppose you are testing to make sure two programs generate the same results, but you only want to test this when the programs change.

Here's one way to do it. If either "script1" or "script2" change, then execute them. If the results differ, execute "report"

```
diff: script1.out script2.out
        diff -b script1.out script2.out >/dev/null || ./Report
script1.out: script1
        ./script1 >script1.out
script2.out: script2
        ./script2 >script2.out
```

I redirected the output of *diff* to /dev/null to hide the info when I run the make command. I'm just interested in the statu of the command, not the data.

Here is another example where I examine a directory for files, looking for new files that I haven't scanned before. I keep track of all of the files I have scanned in the file "scanned.log". If I find a file that is not in that file, I check it using program canned "Scan". That program will append the filename to the file "scanned.log". This *Makefile* recipe does this by using two while loops. The first one is given a list of filenames using *find*, and the second one only sees filenames that are not in the file "scanned.log". I'm using *grep* with the "-q" quick and the "-s" silent option to suppress error messages about unreadable files. To repeat, I just want the job status of *grep*, not the data or errors.

To elaborate, the first *while* loop executes a test if the filename is in the scanned.log file. If it is not in the file, the echo command sends the filename to standard output, which is, in this case, the second *while* loop. The second loop scar the file.

This way, a complex makefile will execute commands only if it hasn't been executed before, but using data managed by your own program.

```
all:
        find . -name \*.txt | while read f;do grep -sq "$$f" scanned.log || echo  "$$f" ;done | \
        while IFS= read n; \
        do Scan "$$n"; \
        done
```

Common command line arguments

I'll discuss some of the common command line arguments I use with *make*. Note that some of these options have multiple arguments to specify the same option, (e.g. “-f *filename*” or “--file=*filename*”) . Check the manual page for more details.

Using a different Makefile

As I mentioned before, *make* first looks for the file “makefile”, and if this isn't found, it looks for “Makefile”. You can override this with the “-f” option:

```
# execute make with the default name file
make
```

```
# Execute make using the file Makefile.new
make -f Makefile.new
```

You can Use this to debug your *makefiles*, or manage variations.

Keep going with the -k argument

Normally, *make* stops when an error occurs. You can use the “-k” argument to tell *make* to ignore the error and keep going. You can to be careful with this if errors cause data to be invalid. Buf if you have a long-running build process, you may want to keep going as long as possible.

Do Nothing - Make with the -n argument

When you start using *make*, you may want to see what *make* will do before you execute it. I often have this issue when I look at someone else's *Makefile*. Typing “sudo make install” on someone else's code should make you very nervous. Besides examining the contents of the *Makefile*, there is another choice: execute “make -n”. The “-n” option is a do-nothing option, It just prints the commands that would be executed, without executing them. Nothing is changed and no files are created.

Recipe prefixes

Certain characters can be used in the front of a recipe to control how *make* execute the recipe. These can make *mak* (I couldn't resist), less annoying.

Ignoring errors with the - prefix

You can use the “-k” option to ignore errors, but that's not always what you want. Suppose, for instance, you want to delete all *.out files. If the files exist, no problem. But if you execute a “rm *.out” recipe when no files that match that pattern exist, this will generate an error, and *make* will stop. You can put the prefix “-” at the begnnning of a line to tell *make* to ignore any run-time errors on that line. Here's an example:

```
all:
        ....
clean:
        # The - in the next line is a prefix
        -/bin/rm *.out
        printf "this executes even if no files are deleted"
```

However, you will still see the error. It just won't cause *make* to exit.

Hiding recipes with the @ prefix

I've used “printf” in some of these examples to document my *makefiles*. However, when you execute *make*, we end u seeing *make* show us the printf command before it executes, and then the printf command executes - so we end up seeing this notice twice. You can prevent *make* from echoing lines to the terminal using the “@” prefix:

```
all:
        ....
clean:
        # The - in the next line is a prefix
        -/bin/rm *.out
        @printf "All clean now"
```

You can combine these prefixes.

```
test:
        @-/bin/rm *.out >/dev/null 2>1
        @printf "done"
```

However, this may still generate errors. For instance, I still get the message:

```
Makefile:2: recipe for target 'test' failed
make: [test] Error 1 (ignored)
```

If I add a “touch junk.out” to the recipe:

```
test:
        @touch junk.out
```

```
	@-/bin/rm *.out >/dev/null 2>1
	@printf "done"
```

Then when I execute this recipe, the only thing I see is “done”

Conclusion to Part 1 - Using Make with Shell scripts

I think I've given you enough to start to combine *make* with your shell scripts. *Make*'s greatest power is used when compiling source code, but I find it helpful when writing complex shell scripts. For example, I had to create dozens of long, complex shell commands when I worked on a project where I had to search an external web server for confidential information

I used a dozen different tools to scan the server, extract and download all files, look for duplicates, extract the metadata from these servers, and then scan the metadata for keywords. I had to get this done as quickly and efficiently as possible, and I wanted to keep track of everything I tried, so I could reuse it in the future. My main tool was *make*.

A future tutorial will cover the advanced features used for programmers.