Great Command-line Combinations

By Jerry Peek

Tools with graphical user interfaces (GUIs) can be easy to learn. All of their commands and options are typically laid out on menus and dialog boxes, making it easy to discover what the tool can do. If "easy" and "intuitive" are your main criteria for programs, then a GUI tool may always be the right choice.

But the simplicity of GUI interfaces can also make them weak and inflexible. For instance, imagine that you're a programmer and you're cleaning out your Linux filesystem. You'd like to find object files that haven't been accessed in six months or more, all through your filesystem, and run `make clean` in those directories. Can a graphical filesystem browser do that in one step? Probably not. However, the power of the many *non*-GUI Linux utility programs, joining forces through the shell, would let you make quick work of that job.

Let's look at some powerful things you can do on a command-line. Even if you don't want to do exactly these things, you're likely to get some ideas for related uses. This "relatedness" ability of the shell and its command-line — letting you combine tools to do just what you need to do — is the very spirit of "Power Tools."

Faster than *Emacs* and *vi*

If you make a mistake on the command line, how can you fix it? Most modern shells let you recall and change previous command lines by pressing the up-arrow key (or CONTROL-p or ESCAPE-k) and editing the command-line using *Emacs* or *vi* commands. Before shells did this, though, the C shell had another way to edit: *history substitution*, a set of shell operators starting with the `!` (often called "bang") character. Some of these operators are still faster and easier to use than command-line editing. Here are three:

1. **`!$`** ("bang dollar") expands to the *last* argument on the previous command line. This is usually a filename.

Let's see an example. You just listed the file */a/b/foo.c*. Now you'd like to edit it with *vi*. There's no need to move to the previous line and then use editing commands to change `ls -l` to `vi`. You can get the previous filename with just two keystrokes: `!$`. The shell shows the expanded command line before running it:

```
$ ls -l /a/b/foo.c
-rw-r--r--  ... Apr 20 11:31 /a/b/foo.c
$ vi !$
vi /a/b/foo.c
... vi starts ...
```

2. **`!*`** expands to *all* arguments on the previous command line. This lets you repeat a list of (say) filenames with a new command or fix a command name you mistyped.

For example, there's no such command *emsca* [sic], but the typo is easily remedied with `!*`:

```
% emsca /b/c/foo.d /e/f/bar.g
emsca: Command not found.
% emacs !*
emacs /b/c/foo.d /e/f/bar.g
...emacs starts...
```

3. The sequence *`old^new`* changes the string *`old`* on the previous command line to *`new`*.

So, in the previous example, we could have typed `^sca^acs`. A shorter version of this operator — a single `^` — is great for removing extra characters (with no replacement). For example, if you type two `g`s instead of one, repeat the command and remove one of them with `^g`:

```
$ rm /some/longg/pathname
rm: /some/longg/pathname: Not found
$ ^g
rm /some/long/pathname
```

Merging Output with a Subshell

How can you collect the output of a lot of commands and use it somewhere else? In a GUI, you can copy from each command's window and paste to another — assuming that all of those windows are "copy-able."

In a shell, you can collect command outputs by using a *subshell*. The operator `( )` runs one or more commands in another shell process. You can also redirect the output of that subshell as you would any other process. Subshells let you run multiple commands in a subshell and collect all of their outputs at once.

For instance, you'd like to email your project leader a copy of the *make* output from three different project directories. Here's one way:

```
% (cd adir; make; cd ../bdir; make; \
?  cd ../cdir; make) | mail joe@foo.com
```

The subshell operators run a series of commands separated

by semicolons (`;`), the shell's command separator. Typing a backslash (`\`) at the end of a line tells the shell to keep reading another line, so the shell prints its secondary prompt (here, a `?`). After the closing parenthesis, we pipe the output of the subshell, which is the output of all its commands. The *mail* program reads the text from its standard input, via the pipe, and sends it to `joe@foo.com`.

What if you redirect the subshell's output to a file? Which directory is the output written into? (In other words, do the *cd* commands in the subshell have any effect?) Changes in the subshell don't affect the parent shell. The parent shell stays in whatever current directory it had.

For instance, in the command below, the `make_outputs` file is created in the current working directory before the subshell even starts running:

```
% (cd adir; make; cd ../bdir; make; \
? cd ../cdir; make) > make_outputs
```

This kind of technique works anywhere, not just with *cd* commands. It's also a nice lead-in to the next handy command-line technique: redirecting the output of a loop.

Let a Loop Do the Work, Part I

Bourne-type shells like *bash* don't only let you redirect the output (and input) of subshells. You can redirect the output of almost any shell construct, including *if* and *case* statements (tests) and loops. (This doesn't work in *csh* or *tcsh*, though.)

You can use this technique to keep a log of what happens in a loop. For instance, start a loop by running *echo* to print a message or *date* to show the current time. Redirect the output of the loop to a file or pipe it to some command, and you gather the standard outputs of all commands within the loop, one after another.

Loops aren't just for shell scripts! You can type them interactively at a prompt. Let's re-run the previous example in *bash* with a *for* loop, adding an *echo* to output a label before each *make* output. (The *bash* secondary prompt is `>`.)

```
$ for d in [a-c]dir
> do
>   cd "$d"
>   echo "======= $d ======="
>   make
>   cd ..
> done | mail joe@foo.com
```

The shell expands the wildcard `[a-c]dir` into `adir bdir cdir` and stores each of those names in the shell variable `d` before running the commands from `do` to `done`. The output of the loop is piped to the *mail* program. Joe gets a message like this:

LISTING ONE: Converting image files with *netpbm*

```
$ for file in *.tif
> do
>   echo "===== doing $file ===="
>   tifftopnm $file |
>   pnmscale -height 100 |
>   pnmtopng > ${file%.*}.png
> done
==== doing img0321.tif ====
tifftopnm: writing PPM file
==== doing img0343.tif ====
tifftopnm: writing PPM file
  ...
$ ls
img0321.png  img0343.png  img0369.png
img0321.tif  img0343.tif  img0369.tif
  ...
```

```
======= adir =======
...make output from adir...
======= bdir =======
...make output from bdir...
======= cdir =======
...make output from cdir...
```

Of course, loops aren't only useful when you redirect their outputs. They're handy for any series of commands you need to repeat.

Let's use the *netpbm* utilities (from http://netpbm.sourceforge.net) to make PNG-format thumbnails, 100 pixels high, from a bunch of TIFF-format files. *Listing One* shows the loop typed at a shell prompt.

To get the same base filename with a *.png* extension (for example, to get *img0321.png* from *img0321.tif*, we're using the parameter expansion operator `${file%.*}`. It expands the filename from `$file`, removing everything from the dot to the end. That reads "expand `$file`, stripping off the smallest possible string from the end of the value that matches a dot followed by any number of other characters." Then we add `.png` to the end of the stripped name. (In *csh* and *tcsh*, use `${file:r}` instead.)

If you had a hundred TIFF files to convert, this could save you a lot of time!

Let a Loop Do the Work, Part II

Let's get back to the example at the start of this article: searching a directory tree and removing old object files. There are several ways to tackle this problem. One is by writing a script in Perl or another language that's good at handling Linux filesystems and file properties. But, if you have a shell waiting in your terminal window, why not type the commands you need right there?

If you're new to shells — or the only "shell" you've used is *COMMAND* from Microsoft Windows — this may seem a bit risky: typing a command to clean your filesystem, not even writing a script. Well, you do need experience to understand what the command is doing. (But then, you need experience to write a script, too!)

A shell is actually a programming language interpreter. Its language is command-lines. Let's use it. We'll pair a loop with the Linux *find* utility (covered in the September 2002 column, available online at http://www.linux-mag.com/2002-09/power_01.html).

We saw earlier that you can redirect the output of a Bourne-type shell loop. You can also redirect the *input* to a loop. *Listing Two* shows the technique (the lines have been numbered for reference.) A `while` loop runs a command over and over until that command returns a nonzero exit status. The command to use here is `read`, which reads one line of input from its standard input and stores it in a shell variable.

The output of *find* from line 1 is piped into a `while` loop, which starts on line 2. The standard input comes from *find*, via the pipe. When there's no more output from *find*, `read` returns a nonzero exit status and the loop terminates.

Inside the loop, we start by changing the current directory at line 4. If that fails, *cd* returns a nonzero status and the `||` operator executes `break` to end the loop. (To make things simple, we gave *find* an absolute path, */proj* — so it will output absolute pathnames. If we'd used relative pathnames, we'd need to add another *cd* command at the end of the loop to get back to the starting directory before the next pass through the loop.)

On line 5, `$(echo *.o)` asks the shell to expand that wildcard. If the shell returns a literal `*.o`, there were no matching object files, so the `test` returns a "true" (zero) status and the `&&` operator executes `continue`, which re-starts the loop (gets another directory name from *find*). Line 6 is executed only if there are some object files (if the script didn't branch at line 5). `test -z` tests for empty output from `find *.o`. If none of those object files have been accessed in more than 180 days, *find* produces no output, the `test -z` succeeds, and the `continue` re-starts the loop with the next directory. But if `test -z` succeeds, there are old files, so lines 7 and 8 run to echo the name of the directory to the terminal and do *make clean*.

Line 9 redirects the output of the loop — and the output of all commands inside it — to the *less* pager. Before the pipe to *less*, the `2>&1` operator merges all error messages from the standard error channel (file descriptor 2) onto the standard output (file descriptor 1) so all messages are piped to *less*. This setup makes sure that you can see all of the loop outputs; they won't scroll off.

This long-winded explanation may make the technique seem like a lot of work, but if you understand the shell techniques used, you can type a loop like this with only a minute or two of thought. (That's a lot faster than clicking through lots of subdirectories with a GUI file browser.)

To enhance the script, you might add more tests: for instance, skipping *RCS* directories, checking that a makefile exists, and so on. You could do a dry run, too, using `ls -lut` to list the object files, or run `make -n clean` to show what *make* would do without doing it.

Because this is typed on the command line and on-the-fly, you probably won't add comments or apply clear coding techniques that you'd use in a script file that will stay around for a while. However, command line scripts like this one can easily be captured (say, using cut-and-paste from an *xterm* window) and refined and saved for use again and again.

LISTING TWO: Cleaning a directory tree

```
1: $ find /proj -type d -print |
2: > while read dir
3: > do
4: >   cd "$dir" || break
5: >   test "$(echo *.o)" = '*.o' && continue
6: >   test -z "$(find *.o -atime +180 -type
         f -print)" && continue
7: >   echo "Cleaning $dir"
8: >   make clean
9: > done 2>&1 | less
```

POWER TIP: BEING *NICE*

A process that takes a lot of system resources can "steal" those resources from other processes that need them more. From a shell, you can tell Linux to give certain programs fewer system resources. Simply put *nice* before the command name. (You probably shouldn't start interactive or GUI programs this way. If your system gets busy, a *nice*d program can "freeze.")

Here's a rewritten image-processing loop with the CPU-intensive commands *nice*d:

```
$ for file in *.tif
> do
>   echo "===== doing $file ===="
>   nice tifftopnm $file |
>   nice pnmscale -height 100 |
>   nice pnmtopng > ${file%.*}.png
> done
```

It's a bit more efficient to *nice* the entire loop. Put the loop into a file (like *do_thumbnails*). Then either make the file executable and type `nice ./do_thumbnails` or start a *nice*d shell to read the file, as in `nice bash do_thumbnails`.

Jerry Peek is a freelance writer and instructor who has used Unix and Linux for over 20 years. He's happy to hear from readers; see http://www.jpeek.com/contact.html.