Tip 33: Replace shell commands with their outputs

Ben Klemens

6 December 2011

level: you want something more than pipes
purpose: use outputs as inputs to the next step

Last time (Entry #082), I gave you a four-item list of things your shell can do. Number three was expansions: replacing certain blobs of text with other text.

Variables are a simple expansion. If you set a variable like

```
onething="another thing"
```

on the command line [C shell users: `set onething="another thing"`], then when you later type

```
echo $onething
```

then `another thing` will print to screeen.

Shell variables are a convenience for you to use while working at the command prompt or throwing together a quick script. They are stupendously easy to confuse with *environment variables*, which are sent to new processes and read via a simple set of C functions. Have a look at Appendix A of *Modeling with Data* for details on turning shell variables into environment variables.

Also, your shell will require that there be no spaces on either side of the =, which will annoy you at some point. [This rule is for the purposes of supporting a feature that is mostly useful for makefiles.] But there you have it: our easiest and most basic substitution of one thing for another.

[Isn't it conveniently nifty that the $ is so heavily used in the shell, and yet is entirely absent from C code, so that it's easy to write shell scripts that act on C code (like in Tip #9 (Entry #059)), and C code to produce shell scripts? It's as if the UNIX shell and C were written by the same people to work together.]

For our next expansion, how about the backtick, which on a typical keyboard shares a key with the ~ and is not the more vertical-looking single tick `'`. [The vertical tick indicates that you don't want expansions done: `echo '$onething'` will actually print `$onething`.] The backtick replaces the command you give with the output from the command, doing so macro-style, where the command text is replaced in place with the output text. Here's an example in which we count lines of C code by how many lines have a `;`, `)`, or `}` on them; given that lines of source code is a lousy metric for most purposes anyway, this is as good a means as any, and has the bonus of being one line of shell code:

```
 #count lines with a ), }, or ;, and let that count be named Lines.
Lines=`grep '[)};]' *.c | wc -l`

 #count how many lines there are in a directory listing; name it Files.
Files=`ls *.c |wc -l`

echo files=$Files and lines=$Lines

 #Arithmetic expansion is a double-paren.
 #In bash, the remainder is truncated; more on this later.
echo lines/file = $(($Lines/$Files))

 #Or, use those variables in a here script.
 #By setting scale=3, answers are printed to 3 decimal places.
bc << ---
scale=3
$Lines/$Files
---
```

OK, so now you've met variable substitution, command substitution, and in the sample code I touched on arithmetic substitution for quick desk calculator math. That's what I deem to be the low-hanging fruit; I leave you to read the manual on history expansion, brace expansion, tilde expansion, parameter expansion, word splitting, pathname expansion, glob expansion, and the difference between `" "` and `' '`.