"Linux Gazette...*making Linux just a little more fun!*"

---

bash String Manipulations

By Jim Dennis, jimd@starshine.org

The *bash* shell has many features that are sufficiently obscure you almost never see them used. One of the problems is that the man page offers no examples.

Here I'm going to show how to use some of these features to do the sorts of simple string manipulations that are commonly needed on file and path names.

Background

In traditional Bourne shell programming you might see references to the *basename* and *dirname* commands. These perform simple string manipulations on their arguments. You'll also see many uses of *sed* and *awk* or *perl -e* to perform simple string manipulations.

Often these machinations are necessary perform on lists of filenames and paths. There are many specialized programs that are conventionally included with Unix to perform these sorts of utility functions: *tr*, *cut*, *paste*, and *join*. Given a filename like */home/myplace/a.data.directory/a.filename.txt* which we'll call **$f** you could use commands like:

```
dirname $f
basename $f
basename $f .txt
```

... to see output like:

```
/home/myplace/a.data.directory
a.filename.txt
a.filename
```

Notice that the GNU version of *basename* takes an optional parameter. This handy for specifying a filename "extension" like **.tar.gz** which will be stripped off of the output. Note that *basename* and *dirname* don't verify that these parameters are valid filenames or paths. They simple perform simple string operations on a single argument. You shouldn't use wild cards with them -- since *dirname* takes exactly one argument (and complains if given more) and *basename* takes one argument and an optional one which is not a filename.

Despite their simplicity these two commands are used frequently in shell programming because most shells don't have any built-in string handling functions -- and we frequently need to refer to just the directory or just the file name parts of a given full file specification.

Usually these commands are used within the "back tick" shell operators like *TARGETDIR=`dirname $1`*. The "back tick" operators are equivalent to the *$(...)* construct. This latter construct is valid in Korn shell and *bash* -- and I find it easier to read (since I don't have to squint at me screen wondering which direction the "tick" is slanted).

A Better Way

Although the *basename* and *dirname* commands embody the "small is beautiful" spirit of Unix -- they may push the envelope towards the "too simple to be worth a separate program" end of simplicity.

Naturally you can call on *sed*, *awk*, TCL or *perl* for more flexible and complete string handling. However this can be overkill -- and a little ungainly.

So, *bash* (which long ago abandoned the "small is beautiful" principal and went the way of *emacs*) has some built in syntactical candy for doing these operations. Since *bash* is the default shell on Linux systems then there is no reason not to use these features when writing scripts for Linux.

> If your concerned about portability to other shells and systems -- you may want to stick with *dirname*, *basename*, and *sed*

The *bash* Man Page

The *bash* man page is huge. In contains a complete reference to the "readline" libraries and how to write a **.inputrc** file (which I think should all go in a separate man page) -- and a run down of all the *csh* "history" or **bang!** operators (which I think should be replaced with a simple statement like: "Most of the **bang!** tricks that work in *csh* work the same way in *bash*").

However, buried in there is a section on **Parameter Substitution** which tells us that $foo is really a shorthand for ${foo} which is really the simplest case of several ${foo*:operators*} and similar constructs.

Are you confused, yet?

Here's where a few examples would have helped. To understand the man page I simply experimented with the echo command and several shell variables. This is what it all means:

> Given:
>
> foo=/tmp/my.dir/filename.tar.gz
>
> We can use these expressions:
>
> path = ${foo%/*}
>
> To get: /tmp/my.dir (like *dirname*)
>
> file = ${foo##*/}
>
> To get: filename.tar.gz (like *basename*)
>
> base = ${file%%.*}
>
> To get: filename
>
> ext = ${file#*.}

To get: tar.gz

Note that the last two depend on the assignment made in the second one

Here we notice two different "operators" being used inside the parameters (curly braces). Those are the **#** and the **%** operators. We also see them used as single characters and in pairs. This gives us four combinations for trimming patterns off the beginning or end of a string:

${variable%pattern}
Trim the shortest match from the end
${variable##pattern}
Trim the longest match from the beginning
${variable%%pattern}
Trim the shortest match from the end
${variable#pattern}
Trim the shortest match from the beginning

It's important to understand that these use shell "globbing" rather than "regular expressions" to **match** these patterns. Naturally a simple string like "txt" will match sequences of exactly those three characters in that sequence -- so the difference between "shortest" and "longest" only applies if you are using a shell wild card in your pattern.

A simple example of using these operators comes in the common question of copying or renaming all the *.txt to change the .txt to .bak (in MS-DOS' COMMAND.COM that would be REN *.TXT *.BAK).

This is complicated in Unix/Linux because of a fundamental difference in the programming API's. In most Unix shells the expansion of a wild card pattern into a list of filenames (called "globbing") is done by the shell -- before the command is executed. Thus the command normally sees a list of filenames (like "foo.txt bar.txt etc.txt") where DOS (COMMAND.COM) hands external programs a pattern like *.TXT.

Under Unix shells, if a pattern doesn't match any filenames the parameter is usually left on the command like literally. Under *bash* this is a user-settable option. In fact, under *bash* you can disable shell "globbing" if you like -- there's a simple option to do this. It's almost never used -- because commands like *mv*, and *cp* won't work properly if their arguments are passed to them in this manner.

However here's a way to accomplish a similar result:

```
for i in *.txt; do cp $i ${i%.txt}.bak; done
```

... obviously this is more typing. If you tried to create a shell function or alias for it -- you have to figure out how to pass this parameters. Certainly the following seems simple enough:

```
function cp-pattern { for i in $1; do cp $i ${i%$1}$2; done
```

... but that doesn't work like most Unix users would expect. You'd have to pass this command a pair of specially *chosen*, and *quoted* arguments like:

```
cp-pattern '*.txt' .bak
```

... note how the second pattern has no wild cards and how the first is quoted to prevent any shell globbing. That's fine for something you might just use yourself -- if you remember to quote it right. It's easy enough to add check for the number of arguments and to ensure that there is at least one file that exists in the $1 pattern. However it becomes much harder to make this command reasonably safe and robust. Inevitably it becomes less "unix-like" and thus more difficult to use with other Unix tools.

I generally just take a whole different approach. Rather than trying to use *cp* to make a backup of each file under a slightly changed name I might just make a directory (usually using the date and my login ID as a template) and use a simple *cp* command to copy all my target files into the new directory.

Another interesting thing we can do with these "parameter expansion" features is to iterate over a list of components in a single variable.

For example, you might want to do something to traverse over every directory listed in your path -- perhaps to verify that everything listed therein is really a directory and is accessible to you.

Here's a command that will echo each directory named on your path on it's own line:

```
p=$PATH until [ $p = $d ]; do d=${p%%:*}; p=${p#*:}; echo $d; done
```

... obviously you can replace the *echo $d* part of this command with anything you like.

Another case might be where you'd want to traverse a list of directories that were all part of a path. Here's a command pair that echos each directory from the root down to the "current working directory":

```
p=$(pwd) until [ $p = $d ]; do p=${p#*/}; d=${p%%/*}; echo $d; done
```

... here we've reversed the assignments to *p* and *d* so that we skip the root directory itself -- which must be "special cased" since it appears to be a "null" entry if we do it the other way. The same problem would have occurred in the previous example -- if the value assigned to *$PATH* had started with a ":" character.

Of course, its important to realize that this is not the only, or necessarily the best method to parse a line or value into separate fields. Here's an example that uses the old *IFS* variable (the "inter-field separator in the Bourne, and Korn shells as well as *bash*) to parse each line of */etc/passwd* and extract just two fields:

```
cat /etc/passwd | ( \
        IFS=: ; while read lognam pw id gp fname home sh; \
                do echo $home \"$fname\"; done \
                )
```

Here we see the parentheses used to isolate the contents in a subshell -- such that the assignment to IFS doesn't affect our current shell. Setting the IFS to a "colon" tells the shell to treat that character as the separater between "words" -- instead of the usual "whitespace" that's assigned to it. For this particular function it's very important that IFS consist solely of that character -- usually it is set to "space," "tab," and "newline.

After that we see a typical *while read* loop -- where we read values from each line of input (from */etc/passwd* into seven variables per line. This allows us to use any of these fields that we need from within the loop. Here

we are just using the *echo* command -- as we have in the other examples.

My point here has been to show how we can do quite a bit of string parsing and manipulation directly within *bash* -- which will allow our shell scripts to run faster with less overhead and may be easier than some of the more complex sorts of pipes and command substitutions one might have to employ to pass data to the various external commands and return the results.

Many people might ask: *Why not simply do it all in **perl**?* I won't dignify that with a response. Part of the beauty of Unix is that each user has many options about how they choose to program something. Well written scripts and programs interoperate regardless of what particular scripting or programming facility was used to create them. Issue the command *file /usr/bin/** on your system and and you may be surprised at how many Bourne and C shell scripts there are in there

In conclusion I'll just provide a sampler of some other *bash* parameter expansions:

${parameter:-word}
Provide a default if *parameter* is unset or null.
Example:

```
echo ${1:-"default"}
```

Note: this would have to be used from within a functions or shell script -- the point is to show that some of the parameter substitutions can be use with shell numbered arguments. In this case the string "default" would be returned if the function or script was called with no $1 (or if all of the arguments had been *shift*ed out of existence. *${parameter:=word}*
Assign a value to *parameter* if it was previously unset or null.
Example:

```
echo ${HOME:="/home/.nohome"}
```

${parameter:?word}
Generate an error if *parameter* is unset or null by printing *word* to *stdout*.
Example:

```
: ${HOME:="/home/.nohome"}
```

${TMP:?"Error: Must have a valid Temp Variable Set"}

This one just uses the shell "null command" (the : command) to evaluate the expression. If the variable doesn't exist or has a null value -- this will print the string to the standard error file handle and exit the script with a return code of one.

Oddly enough -- while it is easy to redirect the standard error of processes under *bash* -- there doesn't seem to be an easy portable way to explicitly generate message or redirect output **to** stderr. The best method I've come up with is to use the /proc/ filesystem (process table) like so:

```
function error { echo "$*" > /proc/self/fd/2 }
```

... *self* is always a set of entries that refers to the current process -- and *self/fd/* is a directory full of the

currently open file descriptors. Under Unix and DOS every process is given the following pre-opened file descriptors: stdin, stdout, and stderr.

${parameter:+word}
Alternative value. ${TMP:+"/mnt/tmp"}
use /mnt/tmp instead of $TMP but do nothing if TMP was unset. This is a weird one that I can't ever see myself using. But it is a logical complement to the ${var:-value} we saw above.
${#variable}
Return the length of the variable in characters.
Example:

```
echo The length of your PATH is ${#PATH}
```

Published in Issue 18 of the Linux Gazette, June 1997