Special Characters in Bash

Bash Magic

The secrets we're going to learn cannot be summed up in words. In fact, all of the magic we'll conjure up within these pages can be represented with the characters *!*, *$*, *?*, and -. BY STEVEN GOODWIN

Why type more than you need to type? The shell has a few short symbols that will save you many keystrokes if you know how to use them.This article examines historical substitution and other tricks.

It Was Supposed To Be So Easy

Was Supposed To Be So Easy. Typing commands into the shell is all well and good but can be cumbersome when we're presented with long filenames. Tab completion can help in some cases, but if there are a lot of files with very similar names, we usually end up having to type the whole thing anyway. But in many circumstances, if tab completion is the answer, we're probably asking the wrong question. We need to ask instead about history substitution – and *!$*.

The Last Argument

This curious symbol is a place-holder for the last argument of the previous command. *!$* is useful for situations like this,

```
$ mkdir new_folder
$ cd !$
cd new_folder
$
```

Here you can see that the argument is substituted directly before execution, enabling you to see the full command in the shell before it is run – although you don't have time to cancel it if it's wrong! Note that the command is stored in the history with its components fully substituted, not the *!$* version you type, as can be seen with,

```
$  history | tail -3
604 mkdir new_folder
605 cd new_folder
606 history | tail -3
```

!$ is a special case of a much larger technique called *history substitution* The symbol consists of two components: *!*, which indicates which command to use (in this case, the most recent), and *$*, which means the last argument. Both sections can be replaced with a multitude of different variations. Let us first look at replacing the *$*.

Notice that, in general, a colon is required. This is used to separate the *!*, indicating the last command, and the details of which argument within that command to use. The command in question doesn't have to be last one, either. You can retrieve the next-to-the-last command with !-2, for example.

```
$ mount /cdrom
$ cp /cdrom/*.pdf /media/library
$ umount !-2:$
umount /cdrom
```

Without specifying which arguments are to be used (by appending *:$*, for example), the entire command will be repeated. There is a special case to repeat the entire previous command (*!-1*), and that is *!!*. You can also request the most recent command that starts with a particular group of letters. Bash will match as many, or as few, characters as you give it, looking for the most recent command that began with all the specified letters.

```
$ !m
mount /cdrom
```

For those that would prefer an interactive history, press Control+R, start typing, and watch bash automatically find the most recent matching command.

Advanced users may care to supplement the *!$* argument with extra qualifiers. These extra qualifiers will retrieve a specific portion of the argument and no more. They are detailed in Table 2. These qualifiers work with all the above options, allowing for some very powerful, and obtuse, shell control.

These qualifiers can be combined if separated by colons. They are evaluated in a left-to-right fashion, for example,

Table 1: Symbols for History Substitution

Symbol	Description	Typed Example	Resultant command	Notes
!:0 (zero)	Use command	!:0 another	mkdir another	
!:1 (one)	Use 1st argument	cd !:1		cd new1
Other arguments can be referenced in the same, numeric, manner				
!:$	Use last argument	cd !:$	cd new2	You may use *!$* instead
!:1-2	Use 1st and 2nd arguments		echo !:1-2	echo new1 new2
Omitting the first number uses 0 as default. Omitting the last uses the penultimate argument. !:n* uses the *n*'th argument to the end (and is the same as *!:n-$*)				
!		All arguments	touch !	touch new1 new2
Acts like !:1-$. Useful for handling arguments from scripts, or in creating *alias*es. You may use !* instead				

```
$ mplayer ➘
/media/mp3/rock/file.mp3
$ echo !:1:h:t
rock
```

One popular trick is to use *!** as a safety net before deleting files. First you start by checking to ensure that the files you have targeted targeted backup files are correct.

```
$ ls *.bak
one.bak  three.bak  two.bak
```

You can delete them with either *rm !**, *rm -i !** (which asks for confirmation before deleting each file) or *rm !*:p* (which prints the potential command to the window, but doesn't do anything, acting as secondary check.)

```
$ rm !*:p
rm *.bak
```

You can then press the up arrow, or type *!!*, to retrieve the last command you just typed. After you press return, you'll then be able to delete the files for real, as follows:

```
$ !!
rm *.bak
```

Brackets

Bash uses several different sets of brackets. The (rounded) brackets are for subshells, the [square] brackets expand ranges of letters (remember *tr [A-Z] [a-z]* from [1]) and {curly} brackets, or braces, expand each argument within them and combine it with the rest of the argument. Braces provide the benefit of letting the user perform the same operation to several files. See the following, for instance:

```
# Create a small hierarchy, ➘
  tom/user, tom/group, tom/all,➘
  dick/user, dick/group, etc.
$ mkdir -p {tom,dick,harry}➘
 /{user,group,all}
```

You cannot include spaces before or after the commas (although they can be escaped with \ if necessary), but it is fine to leave entries blank. Blank entries enable us to perform tricks like this:

```
$ cp filename{,.bak}
```

Table 2: Extra Qualifiers

Qualifier	Description	Typed Example	Result	
h	Head	cd !$:h	cd /media/mp3/rock	
t	Tail	echo !$:t	echo new	
p	Print - don't execute	rm -rf !$:p	rm -rf /media/mp3/rock/new	
q	Quote words	cd !$:q	cd '/media/mp3/rock/new'	
x also does this, but breaks at newlines, spaces and tabs				
r	Remove extension	echo !:1:r	echo extra/ripped	
Any preceding directory is left intact				
e	Keep only the extension	echo !:1:e	echo .mp3	It keeps the dot!

"There Goes the Neighbourhood"
An Original Full-Colour Print by J.D. "Illiad" Frazer
Available in May of 2004
at the UserFriendly.Org
Online Store
"There Goes the Neighbourhood."
UserFriendly.Org
The Original Geek Humour Comic Strip

which expands to the arguments *filename* and *filename.bak*.

$! and the Process ID

The *$!* symbol holds the process ID of the last command you ran as a background process. This is the same ID that appears on screen:

```
$ finger steev &
[1] 18959
$ echo $!
18959
```

This feature can serve as a poor man's *timeout*, whereby commands are run in the background, and, if they haven't completed after a specific time, they are killed!

```
TEMPFILE=`mktemp`
DURATION=$1
shift

$* >$TEMPFILE &
PID=$!

sleep $DURATION

kill -9 $PID 2>/dev/null

cat $TEMPFILE
rm $TEMPFILE
```

It would be called thus:

```
./timeout.sh 2 finger steev@somehost.com
```

Powers of the Minus Sign

In shell programming, the minus sign doesn't have negative connotations, as it is used in two of its most powerful, and under-used, features. When used on its own, a single minus is often used to denote standard input in place of a filename. This is not a feature of the shell *per se*, but it is a convention amongst many tools, such as *cat*. The same effect can also be achieved by using */dev/stdin* as the input file.

This example uses - to concatenate standard input with an existing file.

```
$ cat - original > new
This text appears as the first line in 'new'. Followed by the rest of the "original" file
^D
$
```

And so from one minus sign, to two! The double minus means "end of parameters." This symbol provides an elegant way to tell the shell that whatever follows is a command argument (like a filename), even if it looks like an option. This becomes an essential feature when you realize that *-a* is a legitimate filename in Linux, as well as a command line option. Create a file called *-a* in your home directory and try to delete it,

```
$ rm -a
rm: invalid option -- a
Try `rm --help' for more info.
$
```

The same error would occur if you typed *rm ** in a directory where such a file occurred. However, being surround by so many other files, the problem would be less obvious. By using -- to indicate the end of the parameter list, our intention can be realized quite simply.

```
$ rm -- -a
```

You could also delete this file from a GUI file manager or with the command *rm ./-a*, but neither are suitable for a batch file that has to handle indeterminable files. In these cases -- should always be used, since a malicious user could name his files '-a', '-b', '-c', and so on, hoping that a file scanner would either misunderstand the files or terminate prematurely, leaving other suspect content undisturbed.

$? and Exit Status

$? is another useful variable. This expletive holds the *exit status*. The exit status describes the result of the last command. By convention this exit status will be zero (0) if the last command was successful and non-zero if the last command wasn't successful. The precise value of the exit status is determined by the severity of the error, so a minor infraction would be one (1), whilst the inability to execute the command in the first place would be 127. As well as chaining commands together, this symbol can be used as part of a self-aware installer,

```
gawk >/dev/null 2>&1
if [ $? -eq 0 ]; then
    USE_AWK="gawk"
else
   nawk >/dev/null 2>&1
    if [ $? -eq 0 ]; then
        USE_AWK="nawk"
    else
        echo "I need an awk - any awk - to run!"
        exit 1;
    fi
fi
```

The preceding script picks an available version of AWK from those installed on the system. If *gawk* is reachable, the exit status return code is `0` and *gawk* is used. If *gawk* is not reachable on the system, the script continues with *nawk*. The order that the versions are referenced in the script determines the priority of the versions, while the redirection keeps our screen tidy. Not every command can work without arguments, so sometimes you will have to invoke the help or version number to get a zero return code.

Experiment with fitting these Bash techniques into your own routine. They'll save you time and keystrokes. ■

Alias

Some people set up *aliases* in their *.bashrc* or *.alias* files to ease the burden of typing. Some standard inclusions are *alias ll='ls -l'* and *alias la='ls -A'*. However, there are numerous others referenced by Google and employed by the web-surfing public. After all these years, my favourite is still *alias cd..='cd ..'* for all those times my fingers miss the space bar!

INFO

[1] Steven Goodwin: "Hidden in plain sight", Linux Magazine, Issue 43 / June 2004, p48–49

THE AUTHOR

When builders go down the pub they talk about football. Presumably therefore, when footballers go down the pub they talk about builders! When Steven Goodwin goes down the pub he doesn't talk about football. Or builders. He talks about computers. Constantly...