LINUX VOICE TUTORIAL

BASH: BEYOND THE COMMAND PROMPT

JOHN LANE

Speed up repetitive tasks, get more power out of the command line or just make life easier – welcome to the world of Bash scripting.

WHY DO THIS?

- Chain commands together to create flexible scripts.
- Get more from the command line.
- Learn a new way of working.

Most Linux users will know *Bash* as the command line prompt. But it is also a powerful programming language – a lot of the code that glues the various parts of your system together is written in *Bash*. You may have looked at some of it and seen seas of parentheses, braces and brackets. This less-than obvious syntax helps make other languages, such as Python, more attractive to beginners. But *Bash* is ubiquitous in the Linux world, and it's worth taking the time to learn how to go beyond the prompt.

A good introduction to *Bash* programming is to put frequently issued command sequences into a file so that you can replay them at the command prompt instead of typing each one. Such a file is called a script, and we often hear "scripting" instead of "programming". *Bash* is nonetheless a language with its own syntax, capabilities and limitations.

The basics

Bash programs, like Python and Ruby, are not compiled into binary executables, but need to be parsed by an interpreter. For *Bash*, this is an executable called **bash** that interprets commands read interactively from its command prompt or from a script. If you're at a *Bash* prompt, it'll be provided by a running **bash** process, and you can feed a script straight to it:

```
$ source myscript
```

But you may not be at such a prompt (you might use another shell, such as *csh* or *ksh*, or you may be at the Run dialog of your desktop). If you set the execute bit on your script:

```
$ chmod +x myscript
```

then you can execute it:

```
$ myscript
```

which causes your shell to ask the operating system's program loader to start it. This creates, or forks, a child process of your shell.

But the script isn't a binary executable, so the program loader needs to be told how to execute it. You do this by including a special directive as the first line of your script, which is why most **bash** scripts begin with a line this:

```
#!/bin/bash
```

The first two characters, **#!**, known as a shebang, are detected by the program loader as a magic number that tells it that the file is a script and that it should interpret the remainder of the line as the executable to load – plus, optionally, any arguments to pass to it along with the script itself. The program loader starts **\bin\bash** in a new process, and this runs the script. It needs the absolute path to the executable because the kernel has no concept of a search path (that is itself a feature of the shell).

Scripts that perform specific tasks are usually executed so they run in a predictable environment. Every process has an environment that it inherits from its parent, and contains so-called environment variables that offer its parent a way to pass information into it. A process can alter its own environment and prepare that of its children, but it cannot affect its parent.

Scripts specifically written to alter the current environment (like **rc** files) are sourced and usually don't have their execute bit set.

One line at a time

Bash reads input one line at a time, whether from a command prompt or a script. Comments are discarded; they start with a hash **#** character and continue to the end of the line (**bash** sees the shebang as a comment). It applies quoting rules and parameter expansion to what remains and ends up with words – commands, operators and keywords that make up the language. Commands are executed and return an exit status, which is stored in a special variable for use by subsequent commands.

Words are separated by metacharacters: a space or one of **|**, **&**, **;**, **(**, **)**, **<** or **>**. Operators are character sequences containing one or more metacharacters.

Metacharacters can have their special meaning removed by quoting. The first form of quoting removes special meaning from all characters enclosed by single quotes. It is not possible to enclose a single quote within single quotes. Double quotes are

POSIX

An IEEE standard for a portable operating system interface, POSIX is frequently mentioned in texts about shell scripting. It means being compatible with something called the Shell Command Language, which is defined by an IEEE standard and implemented as the shell on all Unix-like systems by the **/bin/sh** command.These days **/bin/sh** is usually a symlink to a shell that can run in a POSIX-compliant mode. The **bash** command does this when launched in this way or if given the **--posix** command-line option.

In POSIX mode, Bash only supports the features defined by the POSIX standard. Anything else is commonly called a bashism. See **http://bit.ly/bashposix** for what's different in *Bash*'s POSIX mode.

Chain of command

Bash expects one command per line, but this can be a chain: a sequence of commands connected together with one of four special operators. Commands chained with **&&** only execute if the exit status of the preceding one was 0, indicating success. Conversely, commands chained with || execute only if the preceding one failed. Commands chained with a semicolon (;) execute irrespective of how the prior command exited. Lastly, the single-ampersand **&** operator chains commands, placing the preceding command into the background:

```
command1 && command2 # perform command2 only if command1 succeeded
command1 || command2 # perform command2 only if command1 failed
command1 ;  command2 # perform command1 and then command2
command1 &  command2 # perform command2 after starting command1 in the background
```

Chains can be combined, giving a succinct if-then-else construct:

```
command1 && command2 || command3
```

The exit status of a chain is the exit status of the last command to execute.

similar, except some metacharacters still work, most notably the Dollar sign, which performs parameter expansion, and the escape ****, which is the third form of quoting and removes special meaning from the following character only.

Parameters pass information into the script. Positional parameters contain the script's argument list, and special variables provide ways to access them en-masse and also provide other information like the script's filesystem path, its process ID and the last command's exit status.

Variables are parameters that you can define by assigning a value to a name. Names can be any string of alphanumeric characters, plus the underscore (_) but cannot begin with a numeric character, and all values are character strings, even numbers. Variables don't need to be declared before use, but doing so enables additional attributes to be set such as making them read-only (effectively a constant) or defining them as an integer or array (they're still string values though!). Assignment is performed with the **=** operator and must be written without spaces between the name and value. Here are some examples that you might see:

```
var1=hello
var2=1234
declare -i int=100 # integer
declare -r CON=123 # constant
declare -a arr=(foo bar baz) # array
```

Variables default to being shell variables; they aren't part of the environment passed to child processes. For that to happen, the variable must be exported as an environment variable:

```
export $MYVAR
```

Names can use upper- and lower-case characters and are case-sensitive. It's good practice to use lower case names for your own variables and use upper case names for constants and environment variables.

Parameter expansion happens when a parameter's name is preceded by the dollar sign, and it replaces the parameter with its value:

```
echo $1
```

which outputs the script's first argument. These so-called positional parameters are numbered upwards from 1 and 0 contains the filesystem path to the script. Parameter names can be enclosed by { and } if their names would otherwise be unclear. Consider this:

```
$ var=1234
$ echo $var5678
$ echo ${var}5678
12345678
```

The first **echo** receives the value of a non-existant variable **var5678** whereas the second gets the value of **var**, followed by **5678**. The other thing to understand about parameters is that **bash** expands them before any command receives them as arguments. If this expansion includes argument separators, then the expanded value will become multiple arguments. You'll encounter this when values contain spaces, and the solution to this problem is quoting:

```
$ file='big data'
$ touch "$file"
$ ls $file
ls: cannot access big: No such file or directory
ls: cannot access data: No such file or directory
```

Here, **touch** creates a file called **big data** because the **file** variable is quoted, but **ls** fails to list it because it is unquoted and therefore receives two arguments instead of one.

For these two reasons, it is common to quote and delimit parameters when expanding them; many scripts refer to variables like this:

```
"${myvar}"
```

Braces are also required to expand array variables. These are defined using parentheses and expanded with braces:

```
$ myarr=(foo bar baz)
$ echo "${myarr[@]}"  # values
foo bar baz
$ echo "${!myarr[@]}" # indices
```

Special Variables

0	The name of the shell (if interactive) or script.
1 .. n	The positional parameters numbered from **1** to the number of arguments **n**. Braces must be used when expanding arguments greater than 9 (eg ${10}).
*	All the positional parameters. Expanding within quotes gives a single word containing all parameters separated by spaces (eg "$*" is equivalent to "$1 $2 ... $n").
@	All the positional parameters. Expanding within quotes gives all parameters, each as a separate word (eg "$@" is equivalent to "$1 $2 ... $n").
?	The exit status of the most recent command.
$	The process ID of the shell.
!	The PID of the last backgrounded command.

LV PRO TIP

You can use a "." instead of "source" to run a script in the current environment.

```
0 1 2
$ echo "${#myarr[@]}" # count
3
```

Arrays are indexed by default and do not need to be declared. You can also create associative arrays if you have **bash** version 4, but you need to declare them:

```
$ declare -A hash=([key1]=value1 [key2]=value2)
$ hash[key3]=value3
$ echo ${hash[@]}
value3 value2 value1
$ echo ${!hash[@]}
key3 key2 key1
$ echo ${hash[key1]}
value1
```

Braces are also used for inline expansion, where **///:a,b\///1** becomes **a1 b1** and **///:1..5\///** becomes **1 2 3 4 5**. Braces also define a command group: a sequence of commands that are treated as one so that their input and output can be redirected:

```
{date; ls;} > output.log
```

A similar construct is the subshell. Commands written in parentheses are launched in a child process. Expanding them enables us to capture their output:

```
now=$(date +%T)
```

Although our example used a child process, the parent blocked; it waited for the child to finish before continuing. Child processes can also be used to run tasks in parallel by backgrounding them:

```
(command)&
```

This enables your script to continue while the 'command' runs in a separate process. You can **wait**, perhaps later on in your script, for it to finish.

Unlike the subshell, the command group does not fork a child process and, therefore, affects the current environment. They cannot be used in a pipeline and they cannot be expanded to capture their output. Subshells can do these things and are also useful for running parallel processes in separate environments.

Do the maths

You'll also encounter double parentheses; these are one way to do integer arithmetic (**bash** doesn't have floating-point numbers); **let** and **expr** are others:

```
profit=$(($income - $expenses))
profit=$((income - expenses))
let profit=$income-$expenses
profit=$(expr $income - $expenses)
```

The double parentheses form allows spaces to be inserted and the dollar signs to be omitted from the expression to aid readability. Also note that the use of **expr** is less efficient, because it's an external command. Arithmetic expansion also allows operators similar to those found in the C programming language, as in this common idiom to increment a variable:

```
$ x=4
$ let x++
$ echo $x
5
```

Finally, we have square brackets, which evaluate expressions and expand to their exit status. They're used to test and compare parameters, variables and file types. There are single- and double-bracket variants; the single bracket expression is an alias for the **test** command – these are equivalent:

```
"$myvar" == hello
test "$myvar" == hello
```

The double bracket expression is a more versatile extended test command (see **help [[**), which is a keyword and part of the language syntax. **test** is just a command that has the opening bracket as an alias and, when used that way, expects its last argument to be a closing bracket. This is an important difference to understand, because it affects how the expression is expanded. **test** is expanded like arguments to any other command, whereas an extended **test** expression is not expanded but parsed in its entirety as an expression with its own syntax, in a way that's more in line with other programming languages.

It supports the same constructs as **test** (see **help test** or **man test**), performs command substitution and expands parameters. Values don't need to be quoted, and comparison operators (**=**, **&&**, **||**, **>** and **<**) work as expected, plus the **=~** operator compares with a regular expression:

```
$ [[ hello =~ ^he ]] && echo match
match
```

Like any command, both single- and double-bracket expressions expand to their exit status and can be used in conditionals that use it to choose the path of execution:

```
if c; then c; fi
if c; then c; else c; fi
if c; then c; elif c; then c; else c; fi
```

where **c** is a command. The semicolons can be omitted if the following word appears on a new line. Each command can be multiple commands but it is the exit status of the final conditional command that determines the execution path. Conditionals can be nested too:

LV PRO TIP

The *Advanced Bash Scripting Guide* contains an unofficial style guide **http://bit.ly/bashstyle**.

Internal and external commands

Some commands are implemented within *Bash* and are known as builtins. They are more efficient than other external commands because they don't have the overhead of forking a child process. Some builtins have equivalent external commands that pre-date them being implemented within **bash**. Keywords are similar to builtins but are reserved words that form part of the language syntax. You can use **type** to see what a word means in **bash**:

```
$ type cat
cat is /usr/bin/cat
$ type echo
echo is a shell builtin
$ type /usr/bin/echo
/usr/bin/echo is /usr/bin/echo
$ type if
if is a shell keyword
```

You can get help on builtin commands and keywords:

```
$ help {
{ ... }: { COMMANDS ; }
    Group commands as a unit.
```

A question of truth

A Boolean expression is either true or false. In *Bash*, true and false are shell builtins (you may also find equivalent external commands in **/usr/bin**) and, like all commands, they return an exit status where zero indicates success and a non-zero value indicates failure. So, 'true' returns 0 and 'false' returns 1.

You may be tempted to write something like this:

```
var=true
```

This assigns a variable called **var** with the value of the four-character string **true**, and has nothing to do with the **true** command. Similarly,

```
if [[ $var == true ]]; then...
```

compares the value of **var** with the four-character string **true**, whereas

```
if true; then...
```

always succeeds. Here **true** is the command and its exit statis is 0, indicating success.

To confuse things further, arithmetic expansion sees 1 as true and 0 as false, and sees the words "true" and "false" as (potentially undefined) variables rather than the builtins described above.

```
$ echo $((true == false))
1
```

That happens because both **true** and **false** are undefined variables that expand to the same value (nothing) and are therefore equal. This makes the expression true which, arithmetically, is 1.

```
if condition
then
 if nested-condition
  command
 else
  command
 fi
fi
```

while and **until** loops are also controlled by exit status:

```
while c; do c; done
until c; do c; done
```

The **for** loop is different – it iterates over a series of words:

```
for i in foo bar baz
do
 something
done
```

but you can use brace expansion to simulate a counting loop:

```
for i in {1..10}
```

Function definition

No programming language would be complete without some way to group and reuse code, and **bash** has functions. A function is easy to define, either:

```
function myfunc {
}
```

or (preferably, and POSIX compliant):

```
myfunc () {
}
```

Functions have the same naming rules as variables but it's conventional to use lower-case words separated by underscores. They can be used wherever commands can, and are given arguments in the same way, although the function definition doesn't define any (the parentheses must be empty). The function sees its arguments as positional parameters.

Variables defined outside a function are visible inside, and variables defined inside are accessible outside, unless declared as local:

```
function f() {
 in1=456
 local in2=789
 echo $out$in1$in2
}
out=123
f              # 123456789
echo $out$in1$in2 # 123456
```

You can be caught out by local variables. Here's an example: if a function **f1** defines a local, then calls another function **f2**, that local is also visible inside **f2**. When a function defines local variables, they are visible to any functions that it calls. Also, you can define one function inside another but you might not get what you expect. All functions are names and have similar scope. Function definitions are executed – that means that a function defined inside another function will be redefined every time that function is called.

Functions return an exit status, which is either the exit status of the last command to be executed within the function, or explicitly set with "return". Exit status is a number between 0 (meaning success) and 255. You can't return anything more complex than that.

There are, however, tricks that you can use to return more complex data from a function. Using global variables is a simple solution, but another common one is to pass the name of a varaible as a parameter and use **eval** to populate it:

```
myfunc() {
 local resultvar=$1
 local result='a value'
 eval $resultvar='"$result"'
}
myfunc foo
echo $foo  # a value
```

eval enables you to build a command in a string and then execute it; so, in the example above, the function passes in **foo** and this gets assigned to the local **resultvar**. So, when **eval** is called, its argument is a string containing **foo='a value'** that it executes to set the variable **foo**. The single quotes ensure that the value of **result** is treated as one word.

These are the main parts of the language, and should be sufficient for any *Bash* script to make sense, but there are many nuances and techniques that you can still learn. Your journey beyond the prompt has just begun...

John Lane is a technology consultant. He doesn't know where our jetpacks are, but he does help businesses use Linux.

LV PRO TIP

test is both a built-in and external command (**/usr/bin/test**).

LV PRO TIP

Try to always use double bracket expressions unless POSIX compliance is important.