SHELL SECRETS COMMAND LINE SERIES

Shell Exit codes and flow control

PART 4 Marco Fioretti helps you to come out of your shell and structure your code for group projects.

LAST TIME

In *LXF67* we looked at arrays, dialogs and even went as far as image processing. If you missed the issue, call 0870 8374722 or +44 1858 438794 for overseas orders.

Parts one to three of this series looked at several basic commands that you can use straight away in simple scripts. In this final part, we'll be structuring and managing code for more complex tasks – perhaps work that's co-developed with other programmers, or that depends on the result of other programs or scripts.

In such collaborative projects, how does a script know what's happening, or if something is true? The answer is that it looks at numerical clues left by the previous scripts or commands. Every command-line program leaves an exit or return status behind after it runs. Exit status 0 means success, and errors are signalled with an integer code between 1 and 255. 126, for example, means that there was a problem with file permissions.

When the last instruction in a script is an exit command with no arguments, the exit status given to the line before it will relate to the whole script. The special variable **$?** always contains the exit status of the last command executed in a script or at the prompt. To verify this, just type the following two commands:

```
exit 50
echo $?
```

Checks on external objects or events can be performed with the test built in, or by using square bracket operators. The latter is a more efficient option; here are some examples of it:

```
[ -f  mailrc ]
[ -d  MailDir ]
[ $COUNTER ]
[ "$NAME" -eq "Carl" ]
```

As you can see, these tests can be performed on files, numbers or strings. In this example, the first two return true if **mailrc** is an actual file and **MailDir** is a directory. The third will be false unless **$COUNTER** is equal to 1, and the last will fail if **$NAME** returns a value other than Carl. (Find details on syntax in the *bash* man page). Remember that the **((...))** and **let** constructs also return a 0 exit status if the calculations they perform yield a non-zero value. As confusing as this is, at least it's consistent.

Using for loops

Once tests and exist codes have let you know what's going on with your script you can start to do something about it. A very simple script follows one straight flow: do this, then that, then this other thing... More complex scripts must repeat some steps several times, or make choices on their own, while running. Let's look at which *bash* constructs can give a script this kind of autonomy. A **for** loop simply does something to or with each element of a fixed list:

```
for XYZ in list_of_arguments
 do
  something using the current value of XYZ
 done
```

You can also use **for** loops interactively, as long as there is a semicolon after the list. Try typing this at the prompt:

```
for XYZ in "Hello" "Johnny Dear"; do echo $XYZ; done
```

You can work through a two-dimensional array with an iterative loop – going over each entry in the array once for the first dimension, and once for the second – but a more readable alternative might be the *set* command, which assigns each sub-string of a list to a positional variable:

```
for $member in "John London" "Ann Liverpool" "Bill Glasgow"
do
 set --$member
 echo "$1 lives in $2"
done
```

This is really useful when parsing plain text databases where each line is a separate record. In general, however, the power of this and other loops is only unleashed when the argument list is generated on the fly, out of some other command.

```
for file in $( find / -type l -name '*html' )
do
#whatever
done
```

The first instruction above is all you need to find and process in real time all the files in your system that have a .html extension but are just links (**-type l**) to other pages.

While and until commands

Sometimes you'll want to do something to each element of a more or less fixed set. Other times you need to carry out an action an unknown number of times, until something else happens. In these cases you need the **while** and **until** commands. **while** tests for a condition at the top of a loop, and keeps looping as long as that condition is false. **until** has the same syntax but does the opposite – that is, it keeps looping as long as its condition is true. These two loops do the same thing:

```
while [condition is false ]
 do
 command...
 done
```

and

```
until [condition is true]
do
command...
done
```

You can terminate these loops earlier if you need, with **break** or **continue** statements. **break** does just what you would expect of it: it breaks the loop that it's in. **continue** only stops the current iteration: it skips all the remaining commands.

Even with these constructs it is possible to nest loops. Of course, a break will have different effects depending on where it is in the loop hierarchy. To understand this, try to uncomment the break instructions in this script, one at a time:

```
#! /bin/bash
A=0
while [[ "$A" -lt "5" ]]
do
echo A: $A
B=0
if [ "$A" == "2"  ]
then
echo "      Hello from outer loop"
#break
else
while [[ "$B" -lt "4" ]]
if [ "$A" == "1"  ]
then
echo "      Hello from inner loop"
#break
else
echo "      B : $B"
fi
let B=$B+1
done
fi
let A=$A+1
done
```

What if...

Sometimes, a program doesn't have to iterate over the same route again and again, but must choose a route to go down once. An **if/then** instruction just decides which of two paths to follow in a flow diagram, according to the exit status of a test or a command. Syntax-wise, the **if** test and **if [condition is true]** forms are equivalent. **if/then** blocks can be nested if needed. In this simple example, **elif** is simply an abbreviation for 'else if':

```
if [ something_is_true ]
then
 # do this
elif [ something_else_is_true ]
then
 # do_that
else
 # just do whatever is written here
fi
```

An **if/then** block is OK if there are only two choices, but what if there are more? Sure, you could use several such blocks nested or cascaded in some way, but things would get ugly quite fast. Luckily, we have the **case** instruction. This is just the equivalent of the **switch** keyword used in C programming. Don't worry if you've never used C before: this basic example of an interactive menu shows all the syntax you need to know.

```
while [ "$os" == "" ]
do
echo "Choose an operating system"
echo "[L]inux"
echo "[W]indows"
echo
read os
case "$os" in
 "L" | "l" )
echo 'Excellent Choice'
;;
 "W" | "w" )
echo 'Yuck! Are you sure?'
  ;;
 *)
echo 'Come on, make your choice!'
;;
esac
done
```

Normally, **case** works on one test variable – **$os** in this script. The various branches of code are separated by double semicolons. Each one begins with a list of all the possible values of the test variable that will trigger the execution of the following commands. Equivalent values are separated by the pipe character (|). The various alternatives are evaluated from top to bottom, stopping at the first one that matches. In the script above, if you type '**L**' or '**l**', the execution will end by printing 'Excellent Choice' to the screen. Should you enter '**W**' or '**w**' (why?) the first branch will be ignored and you'll get the reaction you deserve. The final option, *****), simply does what is necessary if **$os** has any other value not explicitly mentioned in the previous cases. Remember to always add such a final default branch – to send back an error message if nothing else.

Running shell functions

The first time you try out these techniques, you'll notice a couple of things that might seem a bit odd. First of all, chances are that you'll spot a lot of small (and some not so small) chunks of repetitive code, in which only a few initial parameters change. How can you structure it to avoid repetition and make it easier to see the general flow of the source code?

The first step is to create some shell functions. These are simply blocks of code that implement some specified task, and they must be declared before they can be called. The most portable way to do so is this:

```
Name_Of_Your_Function () {
command_1
command_2
return
}
```

To run a shell function you invoke its name and provide any required arguments in the right order. As with stand-alone scripts, these arguments will be available within the function in the variables **$1**, **$2**, etc. Functions also have their local I/O streams, which can be redirected, or can be fed with HERE documents:

```
A_Shell_Function <$file
Another_Shell_Function <<Function_data
Christmas
Easter
Function_data
```

The first instruction sends to **A_Shell_Function** the content of **$file**; the second runs it with **Christmas** as the first argument and **Easter** as the second. Variables can be declared local to a shell function, using the **local** command. This also makes recursion possible – if you really want to have it in something less computationally efficient than a shell script.

Last but not least, even shell functions return an exit status. This will relate to the last command executed or to the argument given to the final **return** command. Again, the calling script will find the exit status in **$?**. LXF

QUICK TIP

Sourcing old files

It can be helpful to place common code in a separate file, which can be loaded with just one instruction by any script that needs it. Write

```
. some_file_name
[arguments]
```

or

```
source some_file_name
[arguments]
```

to virtually paste the content of some_file_name just before launching the script. If any arguments are supplied, they become the positional parameters when some_file_name is executed. If some_file_name is not found or cannot be read, the return status will be given as false.

NEXT STEPS

Hopefully, by now you know enough to start serious scripting – just remember to always back everything up before you change anything. Oh, and to learn more, bookmark the Advanced Bash-Scripting Guide at www.tldp.org/LDP/abs/html.