Introduction to Shell Scripting - Part 2

By Ben Okopnik

Originally published in Issue 54 of Linux Gazette, May 2000

Last month, we took a look at some basics of creating a shell script, as well as a few of the underlying mechanisms that make it all work. This time around, we'll see how loops and conditional execution let us direct program flow in scripts, as well as looking at a few good shell-writing practices.

Conventions

The only thing to note in this article are ellipses (...) - I use them to indicate that the code shown is only a fragment, and not an entire script all by itself. If it helps, think of each ellipse as one or more lines of code that is not actually written out.

Loops and Conditional Execution

for; do; done

Often, scripts are written to automate some repetitive task; as a random example, if you have to repeatedly edit a series of files in a specific directory, you might have a script that looks like this:

```
#!/bin/bash

for n in ~/weekly/*.txt
do
    ae $n
done
echo "Done."
```

or like this:

```
#!/bin/bash
for n in ~/weekly/*.txt; do ae $n; done; echo "Done."
```

The code in both does exactly the same thing - but the first version is much more readable, especially if you're building large scripts with several levels. As good general practice in writing code, you should indent each level (the commands inside the loops); it makes troubleshooting and following your code much easier.

The above control structure is called a 'for' loop - it looks for items remaining in a list (e.g., 'are there any more files, beyond the ones we have already read, that fit the "~/weekly/*.txt" template?'). If the test returns true, it assigns the name of the current item in the list to the loop variable ("n" in this case) and executes the loop body (the part between "do" and "done"), then checks again. Whenever the list runs out, 'for' stops looping and passes control to the line following the 'done' keyword - in our example, the "echo" statement.

A little trick I'd like to mention here. If you want to make the "for" loop 'spin' a certain number of times, the shell syntax can be somewhat tiresome:

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

```
for i in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
do
    echo $i
done
```

What a pain! If you wanted it to iterate, say, 250 times, you'd have to type all of that out! Fortunately, there's a 'shortcut' - the '`seq`' command, which prints a sequence of numbers from 1 to the given maximum, e.g.,

```
#!/bin/bash

for i in $(seq 15)
do
    echo $i
done
```

This is functionally the same as the previous script. '`seq`' is part of the GNU "shellutils" package and is probably already installed on your system. There's also the option of doing this sort of iteration by using a "while" loop, but it's a bit more tricky.

while; do; done

Often, we need a control mechanism that acts based on a specified condition rather than iterating through a list. The 'while' loop fills this requirement:

```
#!/bin/bash

pppd call provider &

while [ -n "$(ping -c 1 192.168.0.1|grep 100%)" ]
do
    echo "Connecting..."
done

echo "Connection established."
```

The general flow of this script is: we invoke '`pppd`', the PPP paenguin... I mean, daemon :), then keep looping until an actual connection is established (if you want to use this script, replace 192.168.0.1 with your ISPs IP address). Here are the details:

1) The "`ping -c 1 xxx.xxx.xxx.xxx`" command sends a single ping to the supplied IP address; note that it has to be an IP address and not a URL - "`ping`" will fail immediately due to lack of DNS otherwise. If there's no response within 10 seconds, it will print something like

```
PING xxx.xxx.xxx.xxx (xxx.xxx.xxx.xxx):
56 data bytes
ping: sendto: Network is unreachable
ping: wrote xxx.xxx.xxx.xxx 64 chars,
ret=-1

  --- xxx.xxx.xxx.xxx ping statistics
---
1 packets transmitted, 0 packets received,
100% packet loss
```

2) The only line we're interested in is the one that gives us the packet loss percentage; with a single packet, it can only be 0% (i.e., a successful ping) or 100%. By piping the output of "ping" through the "grep 100%" command, we narrow it down to that line, if the loss is indeed 100%; a 0% loss will not

produce any output. Note that the "100%" string isn't anything special: we could have used "ret=-1", "unreachable", or anything else that's unique to a failure response.

3) The square brackets that contain the statement are a synonym for the 'test' command, which returns '0' or '1' (true or false) based on the evaluation of whatever's inside the brackets. The '-n' operator returns 'true' if the length of a given string is greater than 0. Since the string is assumed to be contiguous (no spaces), and the line we're checking for is not, we need to surround the output in double quotes - this is a technique that you will use again and again in script writing. Do note that the square brackets require spaces around them - i.e., `[-n $STRING]` won't work; `[ -n $STRING ]` is correct. For more info on the operators used with 'test', type "help test"; a number of very useful ones are available.

4) As long as the above test returns "true" (i.e., as long as the "ping" fails), the 'while' loop will continue to execute - by printing the "Connecting..." string every ten seconds. As soon as a single ping is successful (i.e., the test returns "false"), the 'while' loop will break and pass control to the statement after "done".

"until; do; done"

The 'until' loop is the reverse of the 'while' - it continues to loop as long as the test is false, and fails when it becomes true. I've rarely had the occasion to use it; the 'while' loop and the flexibility of the available tests usually suffice, and this construct is just "syntactic sugar" for those who prefer to avoid logical inversions.

"if; then; [else]; fi"

There are many times when we need to check for the existence of a condition and branch the execution based on the result. For those times, we have the 'if' statement:

```
...

if [ $BOSS="jerk" ]
then
    echo 'Take this job and shove it!'
else
    echo 'Stick around; the money is good.'
fi

 ...
```

<grin> I guess it's not quite that easy... but the logic makes sense. Anyway, if a variable called BOSS has been defined as "jerk" (C programmers take note: '=' and '==' are equivalent in a test statement - no assignment occurs), then the first 'echo' statement will be executed. In all other cases, the second 'echo' statement will run (if $BOSS="idiot", you'll still be working there. Sorry about that. :). Note that the 'else' statement is optional, as in this script fragment:

```
...

if [ -n $ERROR ]
then
    echo 'Detected an error; exiting.'
    exit
fi

 ...
```

This routine will obviously exit if the ERROR variable is anything other than empty - but it will not affect the program flow otherwise.

"case; in; esac"

The remaining tool that we can use for conditional branching is basically a multiple 'if' statement, based on the evaluation of a test. If, for example, we know that the only possible outputs from an imaginary program called 'intel_cpu_test' are 4, 8, 16, 32, or 64, then we can write the following:

```
#!/bin/bash

case $(intel_cpu_test) in
    4) echo "You're running Linux on a calculator??";;
    8) echo "That 8088 is past retirement age...";;
    16) echo "A 286 kinda guy, are you?";;
    32) echo "One of them new-fangled gadgets!";;
    64) echo "Oooh... serious CPU envy!";;
     *) echo "What the heck are you running, anyway?";;
esac
```

(Before all you folks flood me with mail about running Linux on an 8088... you can't run it on a calculator either. :)

Obviously, the "*" at the end is a catch-all: if someone at the Intel Secret Lab runs this on their new CPU (code name "UltraSuperHyperWhizBang"), we want the script to come back with a controlled response rather than a failure. Note the double semicolons - they 'close' each of the "pattern/command" sets and are (for some reason) a common error in "case/esac" constructs. Pay extra attention to yours!

break and **continue**

These statements interrupt the program flow in specific ways. The "break", once executed, immediately exits the enclosing loop; the "continue" statement skips the current loop iteration. This is useful in a number of situations, particularly in long loops where the existence of a given condition makes all further tests unnecessary. Here's a long (but hopefully understandable) pseudo-example:

```
...

while [ "$hosting_party" = "true" ]
do
    case $FOOD_STATUS
    in
         potato_chips_gone) replace_potato_chips;;
         peanuts_finished) refill_peanut_bowl;;
         pretzels_gone) open_new_pretzel_bag;;
         ...
         ...
    esac


if [ police_on_scene ]
then
         talk_to_nice_officers
         continue
fi
```

```
case $LIQUOR_STATUS
in
         vodka_gone) open_new_vodka_bottle;;
         rum_gone) open_new_rum_bottle;;
         ...
         ...
esac

case $ANALYZE_GUEST_BEHAVIOR
in
         lampshade_on_head)     echo "He's been drinking";;
         talking_to_plants)     echo "She's been smoking";;
         talking_to_martians)   echo "They're doing LSD";;
         levitating_objects)    echo "Who spiked my lemonade??";;
        ...
        ...
        ...
esac

done

echo "Dude... what day is it?"
```

A couple of key points: note that in checking the status of various party supplies, you might be better off writing multiple "if" statements - both potato chips **and** pretzels may run out at the same time (i.e., they are not mutually exclusive). The way it is now, the chips have top priority; if two items do run out simultaneously, it will take two loops to replace them.

We can keep checking the food status while trying to convince the cops that we're actually holding a stamp-collectors' meeting (in fact, maintaining the doughnut supply is a crucial factor at this point), but we'll skip right past the liquor status - as it was, we got Joe down off the chandelier just in time...

The "continue" statement skips the last part of the "while" loop as long as the "police_on_scene" function returns 'true'; essentially, the loop body is truncated at that point. Note that even though it is actually inside the "if" construct, it affects the loop that surrounds it: both "continue" and "break" apply only to loops, i.e., "for", "while", and "until" constructs.

Back to the Future

Here is the script we created last month:

```
#!/bin/bash
# "bkup" - copies specified files to the user's ~/Backup
# directory after checking for name conflicts.

a=$(date +%T-%d_%m_%Y)
cp -i $1 ~/Backup/$1.$a
```

Interestingly enough, shortly after finishing last month's article, I was cranking out a bit of C code on a machine that didn't have 'rcs' (the GNU Revision Control System) installed - and this script came in very handy as a 'micro-RCS'; I used it to take "snapshots" of the project status. Simple, generalized scripts of this sort become very useful at odd times...

Error Checking

The above is a workable script - for you, or anyone who cares to read and understand it. Let's face it, though: what we want from a program or a script is to type the name and have it work, right? That, or tell us exactly why it didn't work. In this case, though, what we get is a somewhat cryptic message:
`cp: missing destination file`

```
Try `cp --help' for more information.
```

For everyone else, and for ourselves down the road when we forget exactly how to use this tremendously complex script with innumerable options :), we need to put in error checking - specifically, syntax/usage information. Let's see how what we've just learned might apply:

```
#!/bin/bash

if [ -z $1 ]
then
    echo "'bkup' - copies the specified file to the user's"
    echo "~/Backup directory after checking for name conflicts."
    echo
    echo "Usage: bkup <filename>"
    echo
    exit
fi


  a=$(date +%T-%d_%m_%Y)

cp -i $1 ~/Backup/$1.$a
```

The '-z' operator of 'test' returns '0' (true) for a zero-length string; what we're testing for is 'bkup' being run without a filename. The very beginning is, in my opinion, the best place to put help/usage information in a script - if you forget what the options are, just run the script without any, and you'll get an instant 'refresher course' in using it. You don't even have to put in the original comments, now - note that we've basically incorporated our earlier comments into the usage info. It's still a good idea to put in comments at any non-obvious or tricky places in the script - that brilliant trick you've managed to pull off may cause you to cuss and scratch your head next year if you don't.

Before we wrap up playing with this script, let's give it a few more capabilities. What if you wanted to be able to send different types of files into different directories? Let's give that a shot, using what we've learned:

```
#!/bin/bash

if [ -z $1 ]
then
    echo "'bkup' - copies the specified file to the user's ~/Backup"
    echo "directory tree after checking for name conflicts."
    echo
    echo "Usage: bkup filename [bkup_dir]"
    echo
    echo "bkup_dir Optional subdirectory in '~/Backup' where the file"
    echo " will be stored."
    echo
    exit
fi

if [ -n $2 ]
then
    if [ -d ~/Backup/$2 ]
    then
         subdir=$2/
    else
         mkdir -p ~/Backup/$2
         subdir=$2/
```

```
    fi
fi

a=$(date +%T-%d_%m_%Y)
cp -i $1 ~/Backup/$subdir$1.$a
```

Here is the summary of changes:

1) The comment section of the help now reads "...directory tree" rather than just "directory", indicating the change we've made.

2) The "Usage:" line has been expanded to show the optional (as shown by the square brackets) argument; we've also added an explanation of how to use that argument, since it might not be obvious to someone else.

3) An added "if" construct that checks to see if $2 (a second argument to 'bkup') exists; if so, it checks for a directory with the given name under "~/Backup", and creates one if it does not exist (the "-d" tests if the file exists and is a directory).

4) The 'cp' command now has a 'subdir' variable tucked in between "Backup/" and "$1".

Now, you can type things like

```
bkup my_new_program.c c
bkup filter.awk awk
bkup filter.awk filters
bkup Letter_to_Mom.txt docs
```

etc., and sort everything into whatever categories you like. Plus, the old behavior of "bkup" is still available -

bkup file.xyz

will send a backup of "file.xyz" to the "~/Backup" directory itself; useful for files that fall outside of your sorting criteria.

By the way: why are we appending a "/" to $2 in the "if" statement instead of right in the "cp" line? Well, if $2 *doesn't* exist, then we want 'bkup' to act as it did originally, i.e., send the file to the "Backup" directory. If we write something like

cp -i $1 ~/Backup/$subdir/$1.$a

(note the extra "/" between $subdir and $1), and $2 isn't specified, then $subdir becomes blank, and the line above becomes

cp -i $1 ~/Backup//$1.$a

- not that it hurts anything, but we want to stick with standard shell syntactic practice wherever possible (since shell quirks, such as a double '/' being ignored, are not guaranteed to stick around.)

In fact, it's a really good idea to consider all the possibilities whenever you're building variables into a string; a classic mistake of that sort can be seen in the following script -

```


DO NOT USE THIS SCRIPT!
```

```
#!/bin/bash
# Written by Larry, Moe, and Shemp - the Deleshun PoWeR TeaM!!!
# Checked by Curly: "Why, soitainly it woiks! Nyuk-nyuk-nyuk!"

# All you've gotta do is enter the name of this file followed by
# whatever you want to delete - directories, dot files, multiple
# files, anything is OK!

rm -rf $1*

DO NOT USE THIS SCRIPT!
```

Well, at least they commented it. :)

What happens if somebody does run "three_stooges", and *doesn't enter a parameter*? The active line in the script becomes

rm -rf *

Assuming that you're Joe User in your home directory, the result is pretty horrible - it'll wipe out all of your personal files. It becomes a catastrophe if you're the root user in the root directory - **the entire system goes away**!!

Viruses seem like such friendly, harmless things about now...

Be careful with your script writing. As you have just seen, you have the power to destroy your entire system in a blink.

```
Unix was never designed to keep people from doing stupid things,
because that policy would also keep them from doing clever things.
 -- Doug Gwyn

Unix gives you just enough rope to hang yourself - and then a
couple more feet, just to be sure.
 -- Eric Allman
```

The philosophy makes sense: unlimited power in the tools, restriction by permissions - but it imposes a responsibility: you must take appropriate care. As a corollary, whenever you're logged in as root, do not run any shell scripts that are not provably harmless (note the Very Large assumptions hanging off that phrase - "*provably harmless*"...)

Wrapping it up

Loops and conditional execution are a very important part of most scripts. As we analyze other shell scripts in future articles, you'll see some of the myriad ways in which they can be used - a script of even average complexity cannot exist without them.

Next month, we'll take a look at some tools that are commonly used in shell scripts - tools that may be very familiar to you as command-line utilities - and explore how they may be connected together to produce desired results. We'll also dissect a couple of scripts - mine, if no one else is brave enough to send in the results of their keyboard concoctions. (Be Afraid. Be Very Afraid. :)

I welcome all comments and corrections in regard to this series of articles, as well as any interesting scripts that you may send in. All flames will be sent to `/dev/null` (oh no, it's full...)

Until next month -
Happy Linuxing!

"Script Quote" of the month:

```
What's this script do?

'unzip; touch; finger; mount; gasp; yes; umount; sleep'

Hint for the answer: not everything is computer-oriented. Sometimes you're
in a sleeping bag, camping out with your girlfriend.''
 -- Frans van der Zande
```

References

The "man" pages for 'bash', 'seq', 'ping', 'grep'
The "help" command for 'for', 'while', 'until', 'if', 'case', 'test', 'break', 'continue'
"Introduction to Shell Scripting - The Basics" in the previous issue