Shell Scripting 2

Written by Dave Pawlowski, September 30, 2012

Escape Characters

Sometimes it is necessary to be able to write out or use a character that is reserved for use by the unix system. For example, at the command prompt, try echoing a backslash:

```
% echo \
```

Linux won't know what you are trying to do so it will just print out a question mark. This is because the backslash is used to escape characters in Linux. In other words, putting the slash in front of a character that usually does something special in Linux, you are letting the OS know that you actually want to print that character. This is often the most useful when you want to print quotes.

```
% echo "
```

will give you an error, but

```
% echo \"
```

will print out those quotes.

Loops

Most languages have the concept of loops: If we want to repeat a task twenty times, we don't want to have to type in the code twenty times, with maybe a slight change each time. As a result, we have for and while loops in the shell. There are somewhat fewer features to loops than in other languages, but nobody claimed that shell programming has the power of C.

For Loops

for loops iterate through a set of values until the list is exhausted (note that we are using bash):

for1.sh

```
#!/bin/bash
for i in 1 2 3 4 5
do
  echo "Looping ... number $i"
done
```

Next, try this one:

for2.sh

```
#!/bin/bash
for i in hello 1 * 2 goodbye
do
   echo "Looping ... i is set to $i"
done
```

Try to understand what is happening here. If you don't see it right away, try it with out the *. Being able to use wildcards in your scripts makes life much easier. Also, what happens if you collect the arguments of the loop in paranthesis (i.e. (hello 1 * 2 goodbye))?

While Loops

while loops are also very useful, but can often be a source of bugs in your code. If you don't use them correctly, you can often be stuck in what's called an infinite loop.

For example, try this in the command line:

```
% while 1
% echo true
% end
```

You'll have to hit C-c.

This code will spit out the word true over and over again, forever, because the argument of the while statement (1) always evaluates to being *true*.

So let's use a while loop quickly:

while1.sh

```
#!/bin/bash
input_string="hello"
while [ $input_string != 'bye' ]
do
  echo "Please type something (bye to quit): "
  read input_string
  echo "You typed: $input_string"
done
```

While loops are often most useful for reading a file or document and doing something with that information. The ability to read in data is somewhat limited when shell scripting. That's ok though, as shell scripts are typically used to get ready to use data, not necessarily use the data itself.

Logic

One of the most important types of control statements in any programming language is the if statement.

if1.sh

```
#!/bin/bash
echo $#
if [ $# -eq 0 ]
then
  echo "There are no arguments"
fi

if [ $# -ne 0 ]
then
  echo "There are arguments ($#): $*"
fi
```

Try running this script a few times, starting with:

```
% if1.sh
```

Then, add an argument or two...:

```
% if1.sh hello!
```

```
% if1.sh hello! how are you?
```

```
% if1.sh hello! "how are you?"
```

Make sure you understand what is happening here. Linux automatically sets certain variables when you run a script.

In order to use if statements properly, you need to know how to relate 2 or more variables to one another. You do this by using operators. You've seen a couple of those already above, but there are more (note strings and integers use different operators):

```
Integer Comparison

-eq
  is equal to: if [ "$a" -eq "$b" ]
-ne
  is not equal to: if [ "$a" -ne "$b" ]
-gt
  is greater than: if [ "$a" -gt "$b" ]
-ge
  is greater than or equal to: if [ "$a" -ge "$b" ]
-lt
  is less than: if [ "$a" -lt "$b" ]
-le
  is less than or equal to: if [ "$a" -le "$b" ]
<
  is less than (within double parentheses): (("$a" < "$b"))
<=
  is less than or equal to (within double parentheses): (("$a" <= "$b"))
>
  is greater than (within double parentheses): (("$a" > "$b"))
>=
  is greater than or equal to (within double parentheses): (("$a" >= "$b"))

string comparison
==
  is equal to: if [ "$a" = "$b" ] (note the whitespace between the =)
!=
  is not equal to: if [ "$a" != "$b" ]
<
  is less than, in ASCII alphabetical order: if [[ "$a" < "$b" ]]
  if [ "$a" \< "$b" ]
  Note that the "<" needs to be escaped within a [ ] construct.
>
  is greater than, in ASCII alphabetical order: if [[ "$a" > "$b" ]]
  if [ "$a" \> "$b" ]
  Note that the ">" needs to be escaped within a [ ] construct.
```

In addition, it is possible to do file queries:

```
    -e file          file merely exists (may be protected from user)
    -r file          file exists and is readable by user
    -w file          file is writable by user
    -x file          file is executable by user
    -o file          file is owned by user
    -z file          file has size 0
    -f file          file is an ordinary file
    -d file          file is a directory
```

So, for example, you can test to see if a file exists, and if it is a directory:

filetest.sh

```
#!/bin/bash
if [ $# == 0 ]
then
    echo "You have not provided any arguments.  You
      must provide at least 1!"
    echo "filetest.sh"
    echo "Usage:    filetest.sh filename1 [filename2,filename3,...]"
    echo " "
else
    for file in $*; do
       echo "Testing to see if $file exists..."
       if [ -e $file ]; then
          comment="$file exists!"
       else
         comment="Sorry, $file does not exist"
       fi

      if [ -d $file ]; then
        comment2="and it is a directory"
      else
        comment2=""
      fi

  echo "$comment $comment2"
       echo " "
    done
fi
```

Try running this a few times and make sure you get what you expect. Note, you can enter any filename at all, as long as you include the entire path.