Basic Linux Shell Scripting Part 3

Posted By Corey Hynes On October 27, 2002 @ 11:59 am In Linux, Programming | Comments Disabled

In this 3rd and final article in my Shell Scripting series we will study, through example, methods of creating simple useful shell scripts. Some of the topics I will cover will be review of the previous two articles, others will cover some new things. The idea is to take what you have already learned and begin applying it to real world situations.

What do I create my scripts in?

Writing your own shell scripts requires you to know the very basic everyday [1] Linux commands. For example, you should know how to copy, move, create new files, etc. The one thing you must know how to do is to use a text editor. There are three major text editors in Linux, vi, emacs and pico. If you are not familiar with vi or emacs, go for pico or some other easy to use text editor.

CAUTION: It is critical that you take care to not perform any scripting functions as the root user. To avoid this simply type the following commands:

```
Adduser scriptuser
Passwd scriptuser
su scriptuser
```

This will create a user that you can dedicate to running scripts.

Your First BASH Program

Our first program will be the classical “Hello World” program. Yes, if you have programmed before, you must be sick of this by now. However, this is traditional, and who am I to change tradition? The “Hello World” program simply prints the words “Hello World” to the screen. So fire up your text editor, and type the following inside it:

```
#!/bin/bash
echo "Hello World"
```

The first line tells Linux to use the bash interpreter to run this script. In this case, bash is in the /bin directory. If bash is in a different directory on your system, make the appropriate changes to the line. Explicitly specifying the interpreter is very important, so be sure you do it as it tells Linux which interpreter to use to run the instructions in the script. The next thing to do is to save the script. We will call it hello.sh. With that done, you need to make the script executable:

```
shell$ chmod 700 ./hello.sh
```

```
shell$ ./hello.sh
Hello World
```

There it is! Your first program! Boring and useless as it is, this is how everyone starts out. Just remember the process here. Write the code, save the file, and make it executable with chmod.

A More Useful Program

We will write a program that will move all files into a directory, and then delete the directory along with its contents, and then recreate the directory. This can be done with the following commands:

```
shell$ mkdir trash
shell$ mv * trash
shell$ rm -rf trash
shell$ mkdir trash
```

Instead of having to type all that interactively on the shell, write a shell program instead:

```
#!/bin/bash
mkdir trash
mv * trash
rm -rf trash
mkdir trash
echo "Deleted all files!"
```

Save it as clean.sh and now all you have to do is to run clean.sh and it moves all files to a directory, deletes them, recreates the directory, and even prints a message telling you that it successfully deleted all files. So remember, if you find that you are doing something that takes a while to type over and over again, consider automating it with a shell program. You may notice that this is very similar to batch file programming under DOS. In fact, UNIX shells scripts are the grandfather of Batch files.

Comments

Comments help to make your code more readable. They do not affect the output of your program. They are specially made for you to read. All comments in bash begin with the hash symbol: "#", except for the first line (#!/bin/bash). The first line is not a comment. Any lines after the first line that begin with a "#" is a comment. Take the following piece of code:

```
#!/bin/bash
# this program counts from 1 to 10:
for i in 1 2 3 4 5 6 7 8 9 10; do
echo $i
done
```

Even if you do not know bash scripting, you immediately know what the above program does, because of the comment. It is good practice to make use of comments. You will find that if you need to maintain your programs in the future, having comments will make things easier.

Variables

Variables are basically "boxes" that hold values. You will want to create variables for many reasons. You will need it to hold user input, arguments, or numerical values. Take for instance the following piece of code:

```
#!/bin/bash
x=12
echo "The value of variable x is $x"
```

What you have done here, is to give x the value of 12. The line echo "The value of variable x is $x" prints the current value of x. When you define a variable, it must not have any whitespace in between the assignment operator: "=". Here is the syntax:

```
variable_name=this_value
```

The values of variables can be accessed by prefixing the variable name with a dollar symbol:

"$". As in the above, we access the value of x by using echo $x.

There are two types of variables. Local variables, and environmental variables. Environmental variables are set by the system and can usually be found by using the env command. Environmental variables hold special values. For instance, if you type:

```
shell$ echo $SHELL
/bin/bash
```

You get the name of the shell you are currently running. Environmental variables are defined in /etc/profile and ~/.bash_profile. The echo command is good for checking the current value of a variable, environmental, or local. If you are still having problems understanding why we need variables, here is a good example:

```
#!/bin/bash
echo "The value of x is 12."
echo "I have 12 pencils."
echo "He told me that the value of x is 12."
echo "I am 12 years old."
echo "How come the value of x is 12?"
```

Okay, now suppose you decide that you want the value of x to be 8 instead of 12. What do you do? You have to change all the lines of code where it says that x is 12. But wait… there are other lines of code with the number 12. Should you change those too? No, because they are not associated with x. Confusing right? Now, here is the same example, only it is using variables:

```
#!/bin/bash
x=12 # assign the value 12 to variable x
echo "The value of x is $x."
echo "I have 12 pencils."
echo "He told me that the value of x is $x."
echo "I am 12 years old."
echo "How come the value of x is $x?"
```

Here, we see that $x will print the current value of variable x, which is 12. So now, if you wanted to change the value of x to 8, all you have to do, is to change the line x=12 to x=8, and the program will automatically change all the lines with $x to show 8, instead of 12. The other lines will be unaffected. Variables have other important uses as well, as you will see later on.

Control Structures

Control structures allow your program to make decisions and to make them more compact. More importantly as well, it allows us to check for errors. So far, all we have done is write programs that start from the top, and go all the way to the bottom until there are no more commands left in the program to run. For instance:

```
#!/bin/bash
cp /etc/foo .
echo "Done."
```

This little shell program, call it bar.sh, copies a file called /etc/foo into the current directory and prints "Done" to the screen. This program will work, under one condition. You must have a file called /etc/foo. Otherwise here is what happens:

```
shell$ ./bar.sh
cp: /etc/foo: No such file or directory
Done.
```

So you can see, there is a problem. Not everyone who runs your program will have /etc/foo in their system. It would perhaps be better if your program checked if /etc/foo existed, and

then if it did, it would proceed with the copying, otherwise, it would quit. In pseudo code, this is what it would look like:

```
if /etc/code exists, then
copy /etc/code to the current directory
print "Done." to the screen.
otherwise,
print "This file does not exist." to the screen
exit
```

Can this be done in bash? Of course! The collection of bash control structures are, if, while, until, for and case. Each structure is paired, meaning it starts with a starting “tag” and ends with an ending “tag”. For instance, the if structure starts with if, and ends with fi. Control structures are not programs found in your system. They are a built in feature of bash. Meaning that from here on, you will be writing your own code, and not just embedding programs into your shell program.

if ... else ... elif ... fi

One of the most common structures is the if structure. This allows your program to make decisions, like, “do this if this conditions exists, else, do something else”. To use the if structure effectively, we must make use of the test command. test checks for conditions, that is, existing files, permissions, or similarities and differences. Here is a rewrite on bar.sh:

```
#!/bin/bash
if test -f /etc/foo
then
# file exists, so copy and print a message.
cp /etc/foo .
echo "Done."
else
# file does NOT exist, so we print a message and exit.
echo "This file does not exist."
exit
fi
```

Notice how we indent lines after then and else. Indenting is optional, but it makes reading the code much easier in a sense that we know which lines are executed under which condition. Now run the program. If you have /etc/foo, then it will copy the file, otherwise, it will print an error message. test checks to see if the file /etc/foo exists. The -f checks to see if the argument is a regular file. Here is a list of test’s options:

-d check if the file is a directory
-e check if the file exists
-f check if the file is a regular file
-g check if the file has SGID permissions
-r check if the file is readable
-s check if the file’s size is not 0
-u check if the file has SUID permissions
-w check if the file is write able
-x check if the file is executable

else is used when you want your program to do something else if the first condition is not met. There is also the elif which can be used in place of another if within the if. Basically elif stands for “else if”. You use it when the first condition is not met, and you want to test another condition.

If you find that you are uncomfortable with the format of the if and test structure, that is:

```
if test -f /etc/foo
then
```

then, you can do it like this:

if [-f /etc/foo]; then

The square brackets form test. If you have experience in C programming, this syntax might be more comfortable. Notice that there has to be white space surrounding both square brackets. The semicolon: ";" tells the shell that this is the end of the command. Anything after the semicolon will be run as though it is on a separate line. This makes it easier to read basically, and is of course, optional. If you prefer, just put then on the next line.

When using variables with test, it is a good idea to have them surrounded with quotation marks. Example:

if ["$name" -eq 5]; then

while ... do ... done

The while structure is a looping structure. Basically what it does is, "while this condition is true, do this until the condition is no longer true". Let us look at an example:

```
#!/bin/bash
while true; do
echo "Press CTRL-C to quit."
done
```

true is actually a program. What this program does is continuously loop over and over without stopping. Using true is considered to be slow because your shell program has to call it up first and then run it. You can use an alternative, the ":" command:

```
#!/bin/bash
while :; do
echo "Press CTRL-C to quit."
done
```

This achieves the exact same thing, but is faster because it is a built in feature in bash. The only difference is you sacrifice readability for speed. Use whichever one you feel more comfortable with. Here is perhaps, a much more useful example, using variables:

```
#!/bin/bash
x=0; # initialize x to 0
while [ "$x" -le 10 ]; do
echo "Current value of x: $x"
# increment the value of x:
x=$(expr $x + 1)
sleep 1
done
```

As you can see, we are making use of the test (in its square bracket form) here to check the condition of the variable x. The option -le checks to see if x is less than, or equal to the value 10. In English, the code above says, "While x is less than 10 or equal to 10, print the current value of x, and then add 1 to the current value of x.". sleep 1 is just to get the program to pause for one second. You can remove it if you want. As you can see, what we were doing here was testing for equality. Check if a variable equals a certain value, and if it does, act accordingly. Here is a list of equality tests:

Checks equality between numbers:
x -eq y Check is x is equals to y
x -ne y Check if x is not equals to y
x -gt y Check ifx is greater than y
x -lt y Check if x is less than y

Checks equality between strings:
x = y Check if x is the same as y
x != y Check if x is not the same as y
-n x Evaluates to true if x is not null
-z x Evaluates to true if x is null.

The above looping script we wrote should not be hard to understand, except maybe for this line:

```
x=$(expr $x + 1)
```

The comment above it tells us that it increments x by 1. But what does $(…) mean? Is it a variable? No. In fact, it is a way of telling the shell that you want to run the command expr $x + 1, and assign its result to x. Any command enclosed in $(…) will be run:

```
#!/bin/bash
me=$(whoami)
echo "I am $me."
```

Try it and you will understand what I mean. The above code could have been written as follows with equivalent results:

```
#!/bin/bash
echo "I am $(whoami)."
```

You decide which one is easier for you to read. There is another way to run commands or to give variables the result of a command. This will be explained later on. For now, use $(…).

until ... do ... done

The until structure is very similar to the while structure. The only difference is that the condition is reversed. The while structure loops while the condition is true. The until structure loops until the condition is true. So basically it is “until this condition is true, do this”. Here is an example:

```
#!/bin/bash
x=0
until [ "$x" -ge 10 ]; do
echo "Current value of x: $x"
x=$(expr $x + 1)
sleep 1
done
```

This piece of code may look familiar. Try it out and see what it does. Basically, until will continue looping until x is either greater than, or equal to 10. When it reaches the value 10, the loop will stop. Therefore, the last value printed for x will be 9.

for ... in ... do ... done

The for structure is used when you are looping through a range of variables. For instance, you can write up a small program that prints 10 dots each second:

```
#!/bin/bash
echo -n "Checking system for errors"
for dots in 1 2 3 4 5 6 7 8 9 10; do
echo -n "."
done
echo "System clean."
```

In case you do not know, the -n option to echo prevents a new line from automatically being

added. Try it once with the -n option, and then once without to see what I mean. The variable dots loops through values 1 to 10, and prints a dot at each value. Try this example to see what I mean by the variable looping through the values:

```
#!/bin/bash
for x in paper pencil pen; do
echo "The value of variable x is: $x"
sleep 1
done
```

When you run the program, you see that x will first hold the value paper, and then it will go to the next value, pencil, and then to the next value, pen. When it finds no more values, the loop ends.

Here is a much more useful example. The following program adds a .html extension to all files in the current directory:

```
#!/bin/bash
for file in *; do
echo "Adding .html extension to $file..."
mv $file $file.html
sleep 1
done
```

If you do not know, * is a wild card character. It means, “everything in the current directory”, which is in this case, all the files in the current directory. All files in the current directory are then given a .html extension. Recall that variable file will loop through all the values, in this case, the files in the current directory. mv is then used to rename the value of variable file with a .html extension.

case ... in ... esac

The case structure is very similar to the if structure. Basically it is great for times where there are a lot of conditions to be checked, and you do not want to have to use if over and over again. Take the following piece of code:

```
#!/bin/bash
x=5 # initialize x to 5
# now check the value of x:
case $x in
0) echo "Value of x is 0."
;;
5) echo "Value of x is 5."
;;
9) echo "Value of x is 9."
;;
*) echo "Unrecognized value."
esac
```

The case structure will check the value of x against 3 possibilities. In this case, it will first check if x has the value of 0, and then check if the value is 5, and then check if the value is 9. Finally, if all the checks fail, it will produce a message, “Unrecognized value.”. Remember that “*” means “everything”, and in this case, “any other value other than what was specified”. If x holds any other value other than 0, 5, or 9, then this value falls into the *’s category. When using case, each condition must be ended with two semicolons. Why bother using case when you can use if? Here is the equivalent program, written with if. See which one is faster to write, and easier to read:

```
#!/bin/bash
x=5 # initialize x to 5
if [ "$x" -eq 0 ]; then
echo "Value of x is 0."
elif [ "$x" -eq 5 ]; then
```

```
echo "Value of x is 5."
elif [ "$x" -eq 9 ]; then
echo "Value of x is 9."
else
echo "Unrecognized value."
fi
```

Quotations

Quotation marks play a big part in shell scripting. There are three types of quotation marks. They are the double quote: “, the forward quote: ‘, and the back quote: `. Does each of them mean something? Yes.

The double quote is used mainly to hold a string of words and preserve whitespace. For instance, “This string contains whitespace.”. A string enclosed in double quotes is treated as one argument. For instance, take the following examples:

```
shell$ mkdir hello world
shell$ ls -F
hello/ world/
```

Here we created two directories. mkdir took the strings hello and world as two arguments, and thus created two directories. Now, what happens when you do this:

```
shell$ mkdir "hello world"
shell$ ls -F
hello/ hello world/ world/
```

It created a directory with two words. The quotation marks made two words, into one argument. Without the quotation marks, mkdir would think that hello was the first argument, and world, the second.

Forward quotes are used primarily to deal with variables. If a variable is enclosed in double quotes, its value will be evaluated. If it is enclosed in forward quotes, its value will not be evaluated. To make this clearer, try the following example:

```
#!/bin/bash
x=5 # initialize x to 5
# use double quotes
echo "Using double quotes, the value of x is: $x"
# use forward quotes
echo 'Using forward quotes, the value of x is: $x'
```

See the difference? You can use double quotes if you do not plan on using variables for the string you wish to enclose. In case you are wondering, yes, forward quotes can be used to preserve whitespace just like double quotes:

```
shell$ mkdir 'hello world'
shell$ ls -F
hello world/
```

Back quotes are completely different from double and forward quotes. They are not used to preserve whitespace. If you recall, earlier on, we used this line:

```
x=$(expr $x + 1)
```

As you already know, the result of the command expr $x + 1 is assigned to variable x. The exact same result can be achieved with back quotes:

```
x=`expr $x + 1`
```

Which one should you use? Whichever one you prefer. You will find the back quote used more often than the $(…). However, I find $(…) easier to read, especially if you have something like this:

```
$!/bin/bash
echo "I am `whoami`"
```

Arithmetic with BASH

BASH allows you to perform arithmetic expressions. As you have already seen, arithmetic is performed using the expr command. However, this, like the true command, is considered to be slow. The reason is that in order to run true and expr, the shell has to start them up. A better way is to use a built in shell feature which is quicker. So an alternative to true, as we have also seen, is the ":" command. An alternative to using expr, is to enclose the arithmetic operation inside $((…)). This is different from $(…). The number of brackets will tell you that. Let us try it:

```
#!/bin/bash
x=8 # initialize x to 8
y=4 # initialize y to 4
# now we assign the sum of x and y to z:
z=$(($x + $y))
echo "The sum of $x + $y is $z"
```

As always, whichever one you choose, is purely up to you. If you feel more comfortable using expr to $((…)), by all means, use it.

bash is able to perform, addition, subtraction, multiplication, division, and modulus. Each action has an operator that corresponds to it:

ACTION OPERATOR
Addition +
Subtraction -
Multiplication *
Division /
Modulus %

Everyone should be familiar with the first four operations. If you do not know what modulus is, it is the value of the remainder when two values are divided. Here is an example of arithmetic in bash:

```
#!/bin/bash
x=5 # initialize x to 5
y=3 # initialize y to 3

add=$(($x + $y)) # add the values of x and y and assign it to variable add
sub=$(($x - $y)) # subtract the values of x and y and assign it to variable
sub
mul=$(($x * $y)) # multiply the values of x and y and assign it to variable
mul
div=$(($x / $y)) # divide the values of x and y and assign it to variable div
mod=$(($x % $y)) # get the remainder of x / y and assign it to variable mod

# print out the answers:
echo "Sum: $add"
echo "Difference: $sub"
echo "Product: $mul"
echo "Quotient: $div"
echo "Remainder: $mod"
```

Again, the above code could have been done with expr. For instance, instead of add=$(($x + $y)), you could have used add=$(expr $x + $y), or, add=`expr $x + $y`.

Reading User Input

Now we come to the fun part. You can make your program so that it will interact with the user, and the user can interact with it. The command to get input from the user, is read. read is a built in bash command that needs to make use of variables, as you will see:

```
#!/bin/bash
# gets the name of the user and prints a greeting
echo -n "Enter your name: "
read user_name
echo "Hello $user_name!"
```

The variable here is user_name. Of course you could have called it anything you like. read will wait for the user to enter something and then press ENTER. If the user does not enter anything, read will continue to wait until the ENTER key is pressed. If ENTER is pressed without entering anything, read will execute the next line of code. Try it. Here is the same example, only this time we check to see if the user enters something:

```
#!/bin/bash
# gets the name of the user and prints a greeting
echo -n "Enter your name: "
read user_name

# the user did not enter anything:
if [ -z “$user_name” ]; then
echo “You did not tell me your name!”
exit
fi

echo “Hello $user_name!”
```

Here, if the user presses the ENTER key without typing anything, our program will complain and exit. Otherwise, it will print the greeting. Getting user input is useful for interactive programs that require the user to enter certain things. For instance, you could create a simple [2] database and have the user enter things in it to be added to your database.

Functions

Functions make scripts easier to maintain. Basically it breaks up the program into smaller pieces. A function performs an action defined by you, and it can return a value if you wish. Before I continue, here is an example of a shell program using a function:

```
#!/bin/bash
# function hello() just prints a message
hello()
{
echo "You are in function hello()"
}

echo “Calling function hello()…”
# call the hello() function:
hello
echo “You are now out of function hello()”
```

Try running the above. The function hello() has only one purpose here, and that is, to print a message. Functions can of course be made to do more complicated tasks. In the above, we called the hello() function by name by using the line:

```
hello
```

When this line is executed, bash searches the script for the line hello(). It finds it right at the

top, and executes its contents.

Functions are always called by their function name, as we have seen in the above. When writing a function, you can either start with function_name(), as we did in the above, or if you want to make it more explicit, you can use the function function_name(). Here is an alternative way to write function hello():

```
function hello()
{
echo "You are in function hello()"
}
```

Functions always have an empty start and closing brackets: “()”, followed by a starting brace and an ending brace: “{…}”. These braces mark the start and end of the function. Any code enclosed within the braces will be executed and will belong only to the function. Functions should always be defined before they are called. Let us look at the above program again, only this time we call the function before it is defined:

```
#!/bin/bash
echo "Calling function hello()..."
# call the hello() function:
hello
echo "You are now out of function hello()"


# function hello() just prints a message
hello()
{
echo “You are in function hello()”
}
```

Here is what we get when we try to run it:

```
shell$ ./hello.sh
Calling function hello()...
./hello.sh: hello: command not found
You are now out of function hello()
```

As you can see, we get an error. Therefore, always have your functions at the start of your code, or at least, before you call the function. Here is another example of using functions:

```
#!/bin/bash
# admin.sh - administrative tool


# function new_user() creates a new user account
new_user()
{
echo “Preparing to add a new user…”
sleep 2
adduser # run the adduser program
}


echo “1. Add user”
echo “2. Exit”


echo “Enter your choice: ”
read choice


case $choice in
1) new_user # call the new_user() function
;;
*) exit
;;
```

```
esac
```

In order for this to work properly, you will need to be the root user, since adduser is a program only root can run. Hopefully this example (short as it is) shows the usefulness of functions.

Trapping

You can use the built in command trap to trap signals in your programs. It is a good way to gracefully exit a program. For instance, if you have a program running, hitting CTRL-C will send the program an interrupt signal, which will kill the program. trap will allow you to capture this signal, and will give you a chance to either continue with the program, or to tell the user that the program is quitting. trap uses the following syntax:

```
trap action signal
```

action is what you want to do when the signal is activated, and signal is the signal to look for. A list of signals can be found by using the command trap -l. When using signals in your shell programs, omit the first three letters of the signal, usually SIG. For instance, the interrupt signal is SIGINT. In your shell programs, just use INT. You can also use the signal number that comes beside the signal name. For instance, the numerical signal value of SIGINTis 2. Try out the following program:

```
#!/bin/bash
# using the trap command

# trap CTRL-C and execute the sorry() function:
trap sorry INT

# function sorry() prints a message
sorry()
{
echo “I’m sorry Dave. I can’t do that.”
sleep 3
}

# count down from 10 to 1:
for i in 10 9 8 7 6 5 4 3 2 1; do
echo $i seconds until system failure.”
sleep 1
done
echo “System failure.”
```

Now, while the program is running and counting down, hit CTRL-C. This will send an interrupt signal to the program. However, the signal will be caught by the trap command, which will in turn execute the sorry() function. You can have trap ignore the signal by having “”” in place of the action. You can reset the trap by using a dash: “-”. For instance:

```
# execute the sorry() function if SIGINT is caught:
trap sorry INT

# reset the trap:
trap - INT

# do nothing when SIGINT is caught:
trap ” INT
```

When you reset a trap, it defaults to its original action, which is, to interrupt the program and kill it. When you set it to do nothing, it does just that. Nothing. The program will continue to run, ignoring the signal.

AND & OR

We have seen the use of control structures, and how useful they are. There are two extra things that can be added. The AND: “&&” and the OR “||” statements. The AND statement looks like this:

condition_1 && condition_2

The AND statement first checks the leftmost condition. If it is true, then it checks the second condition. If it is true, then the rest of the code is executed. If condition_1 returns false, then condition_2 will not be executed. In other words:

if condition_1 is true, AND if condition_2 is true, then…

Here is an example making use of the AND statement:

```
#!/bin/bash
x=5
y=10
if [ "$x" -eq 5 ] && [ "$y" -eq 10 ]; then
echo "Both conditions are true."
else
echo "The conditions are not true."
fi
```

Here, we find that x and y both hold the values we are checking for, and so the conditions are true. If you were to change the value of x=5 to x=12, and then re-run the program, you would find that the condition is now false.

The OR statement is used in a similar way. The only difference is that it checks if the leftmost statement is false. If it is, then it goes on to the next statement, and the next:

condition_1 || condition_2

In pseudo code, this would translate to the following:

if condition_1 is true, OR if condition_2 is true, then…

Therefore, any subsequent code will be executed, provided at least one of the tested conditions is true:

```
#!/bin/bash
x=3
y=2
if [ "$x" -eq 5 ] || [ "$y" -eq 2 ]; then
echo "One of the conditions is true."
else
echo "None of the conditions are true."
fi
```

Here, you will see that one of the conditions is true. However, change the value of y and re-run the program. You will see that none of the conditions are true.

If you think about it, the if structure can be used in place of AND and OR, however, it would require nesting the if statements. Nesting means having an if structure within another if structure. Nesting is also possible with other control structures of course. Here is an example of a nested if structure, which is an equivalent of our previous AND code:

```
#!/bin/bash
x=5
```

```
y=10
if [ "$x" -eq 5 ]; then
if [ "$y" -eq 10 ]; then
echo "Both conditions are true."
else
echo "The conditions are not true."
fi
fi
```

This achieves the same purpose as using the AND statement. It is much harder to read, and takes much longer to write. Save yourself the trouble and use the AND and OR statements.

Using Arguments

You may have noticed that most programs in Linux are not interactive. You are required to type arguments, otherwise, you get a “usage” message. Take the more command for instance. If you do not type a filename after it, it will respond with a “usage” message. It is possible to have your shell program work on arguments. For this, you need to know the “$#” variable. This variable stands for the total number of arguments passed to the program. For instance, if you run a program as follows:

shell$ foo argument

$# would have a value of one, because there is only one argument passed to the program. If you have two arguments, then $# would have a value of two. In addition to this, each word on the command line, that is, the program’s name (in this case foo), and the argument, can be referred to as variables within the shell program. foo would be $0. argument would be $1. You can have up to 9 variables, from $0 (which is the program name), followed by $1 to $9 for each argument. Let us see this in action:

```
#!/bin/bash
# prints out the first argument
# first check if there is an argument:
if [ "$#" -ne 1 ]; then
echo "usage: $0 "
fi
```

echo “The argument is $1•

This program expects one, and only one, argument in order to run the program. If you type less than one argument, or more than one, the program will print the usage message. Otherwise, if there is an argument passed to the program, the shell program will print out the argument you passed. Recall that $0 is the program’s name. This is why it is used in the “usage” message. The last line makes use of $1. Recall that $1 holds the value of the argument that is passed to the program.

Redirection and Piping

As we covered before, redirection and piping are very common. Here are some examples and a little review in the context of usable scripts.

```
shell$ echo "Hello World"
Hello World
```

Redirection allows you to redirect the output somewhere else, most likely, a file. The “>” operator is used to redirect output.

```
shell$ echo "Hello World" > foo.file
shell$ cat foo.file
Hello World
```

Here, the output of echo “Hello World” is redirected to a file called foo.file. When we read the contents of the file, we see the output there. There is one problem with the “>” operator. It will overwrite the contents of any file. What if you want to append to it? Then you must use the append operator: “>>”. It is used in the exact same was as the redirection operator, except that it will not overwrite the contents of the file, but add to it.

Piping allows you to take the output from a program, and then run the output through another program. Piping is done using the pipe operator: “|”.:

```
shell$ cat /etc/passwd | grep shell
shell:x:1002:100:X_console,,,:/home/shell:/bin/bash
```

Here we read the entire /etc/passwd and then pipe the output to grep, which in turn, searches for the string shell and then prints the entire line containing the string, to the screen. You could have mixed this with redirection to save the final output to a file:

```
shell$ cat /etc/passwd | grep shell > foo.file
shell$ cat foo.file
shell:x:1002:100:X_console,,,:/home/shell:/bin/bash
```

It worked. /etc/passwd is read, and then the entire output is piped into grep to search for the string shell. The final output is then redirected and saved into foo.file. You will find redirection and piping to be useful tools when you write your shell programs.

Temporary Files

Temporary files are useful when you want to store information, or persist information to disk for a short period of time. A temporary file can be created with the $$ symbol. This symbol uses a random number generator to either prefix or suffix the file and ensure it has a unique name.

```
shell$ touch hello
shell$ ls
hello
shell$ touch hello.$$
shell$ ls
hello hello.689
```

There it is, your temporary file

Return Values

Most programs return a value depending upon how they exit. For instance, if you look at the manual page for grep, it tells us that grep will return a 0 if a match was found, and a 1 if no match was found. Why do we care about the return value of a program? For various reasons. Let us say that you want to check if a particular user exists on the system. One way to do this would be to grep the user’s name in the /etc/passwd file. Let us say the user’s name is foobar:

```
shell$ grep "foobar" /etc/passwd
shell$
```

No output. That means that grep could not find a match. But it would be so much more helpful if a message saying that it could not find a match was printed. This is when you will need to capture the return value of the program. A special variable holds the return value of a program. This variable is $?. Take a look at the following piece of code:

```
#!/bin/bash
# grep for user foobar and pipe all output to /dev/null:
grep "foobar" /etc/passwd > /dev/null 2>&1
# capture the return value and act accordingly:
```

```
if [ "$?" -eq 0 ]; then
echo "Match found."
exit
else
echo "No match found."
fi
```

Now when you run the program, it will capture the return value of grep. If it equals to 0, then a match was found and the appropriate message is printed. Otherwise, it will print that there was no match found. This is a very basic use of getting a return value from a program. As you continue practicing, you will find that there will be times when you need the return value of a program to do what you want.

If you happen to be wondering what 2>&1 means, it is quite simple. Under Linux, these numbers are file descriptors. 0 is standard input (eg: keyboard), 1 is standard output (eg: monitor) and 2 is standard error (eg: monitor). All normal information is sent to file descriptor 1, and any errors are sent to 2. If you do not want to have the error messages pop up, then you simply redirect it to /dev/null. Note that this will not stop information from being sent to standard output. For example, if you do not have permissions to read another user's directory, you will not be able to list its contents:

```
shell$ ls /root
ls: /root: Permission denied
shell$ ls /root 2> /dev/null
shell$
```

As you can see, the error was not printed out this time. The same applies for other programs and for file descriptor 1. If you do not want to see the normal output of a program, that is, you want it to run silently, you can redirect it to /dev/null. Now if you do not want to see either standard input or error, then you do it this way:

```
shell$ ls /root > /dev/null 2>&1
```

You should recall that this is IO Redirection, and the null device represents nothing or nowhere.

Now what if you want your shell script to return a value upon exiting? The exit command takes one argument. A number to return. Normally the number 0 is used to denote a successful exit, no errors occurred. Anything higher or lower than 0 normally means an error has occurred. This is for you, the programmer to decide. Let us look at this program:

```
#!/bin/bash
if [ -f "/etc/passwd" ]; then
echo "Password file exists."
exit 0
else
echo "No such file."
exit 1
fi
```

By specifying return values upon exit, other shell scripts you write making use of this script will be able to capture its return value. This is similar to programming with functions in other languages.

Porting BASH Scripts

The most well written scripts are portable. That means that they work on many different version of Linux and Unix with little modification. The biggest mistake you can make when considering portability is to use applications that only exist on a single version of Linux. The foo program is an example. It servers the same purpose as echo, but does not exist on all flavors of Linux.

Written by Corey Hynes - [3] Visit Website

URLs in this post:
[1] Linux: http://www.2000trainers.com/tutorials/linux/
[2] database: http://www.2000trainers.com/tutorials/database/
[3] Visit Website: http://