Intro to Bash Scripting

Credit

Mike G mikkey at dynamo.com.ar

What is Bash Scripting?

Basically it is a way to put together a bunch of Linux terminal commands that can be run one right after the other. It has the same attributes as most programming languages but is very picky about syntax and sometimes even spacing!

Getting Started

Bash scripts should have a .sh extension, they will run with no extension though.

You run a bash script like: source [script name] [any command line arguments]

Remember: Always put #! /bin/bash at the top of every script you write

Simple Scripts

Hello World

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

This script has only two lines. The first indicates the system which program to use to run the file. The second line is the only action performed by this script, which prints 'Hello World' on the terminal. If you get something like *./hello.sh: Command not found.* Probably the first line '#!/bin/bash' is wrong, issue whereis bash or see 'finding bash' to see how should you write this line.

Simple Backup

```
#!/bin/bash
tar -cZf /var/my-backup.tgz /home/me/
```

In this script, instead of printing a message on the terminal, we create a tarball of a user's home directory. This is NOT intended to be used, a more useful backup script is presented later in this document.

Redirection

Theory and Quick Reference

There are 3 file descriptors, stdin, stdout and stderr (std=standard).
Basically you can:

1. redirect stdout to a file
2. redirect stderr to a file
3. redirect stdout to a stderr
4. redirect stderr to a stdout

5. redirect stderr and stdout to a file
6. redirect stderr and stdout to stdout
7. redirect stderr and stdout to stderr

   1 'represents' stdout and 2 stderr.

A little note for seeing this things: with the less command you can view both stdout (which will remain on the buffer) and the stderr that will be printed on the screen, but erased as you try to 'browse' the buffer.

Stdout to File

This will cause the output of a program to be written to a file.

```
ls -l > ls-l.txt
```

Here, a file called 'ls-l.txt' will be created and it will contain what you would see on the screen if you type the command 'ls -l' and execute it.

Stderr to File

This will cause the stderr output of a program to be written to a file.

```
grep da * 2> grep-errors.txt
```

Here, a file called 'grep-errors.txt' will be created and it will contain what you would see the stderr portion of the output of the 'grep da *' command.

Stdout to Stderr

This will cause the stderr output of a program to be written to the same filedescriptor than stdout.

```
grep da * 1>&2
```

Here, the stdout portion of the command is sent to stderr, you may notice that in different ways.

Stderr to Stdout

This will cause the stderr output of a program to be written to the same filedescriptor than stdout.

```
grep * 2>&1
```

Here, the stderr portion of the command is sent to stdout, if you pipe to less, you'll see that lines that normally 'disappear' (as they are written to stderr) are being kept now (because they're on stdout).

Stderr and Stdout to File

This will place every output of a program to a file. This is suitable sometimes for cron entries, if you want a command to pass in absolute silence.

```
rm -f $(find / -name core) &> /dev/null
```

This (thinking on the cron entry) will delete every file called 'core' in any directory. Notice that you should be pretty sure of what a command is doing if you are going to wipe it's output.

Pipes

This section explains in a very simple and practical way how to use pipes, and why you may want it.

What They are and Why You'll Want to Use Them

Pipes let you use the output of a program as the input of another one

Simple Pipe with sed

This is very simple way to use pipes.

```
ls -l | sed -e "s/[aeio]/u/g"
```

Here, the following happens: first the command ls -l is executed, and it's output, instead of being printed, is sent (piped) to the sed program, which in turn, prints what it has to.

An Alternative to ls -l *.txt

Probably, this is a more difficult way to do ls -l *.txt, but it is here for illustrating pipes, not for solving such listing dilemma.

```
ls -l | grep "\.txt$"
```

Here, the output of the program ls -l is sent to the grep program, which, in turn, will print lines which match the regex "\.txt$".

Variables

You can use variables as in any programming languages. There are no data types. A variable in bash can contain a number, a character, a string of characters.

You have no need to declare a variable, just assigning a value to its reference will create it.

Hello World! Using Variables

```
#!/bin/bash
STR="Hello World!"
echo $STR
```

Line 2 creates a variable called STR and assigns the string "Hello World!" to it. Then the VALUE of this variable is retrieved by putting the '$' in at the beginning. Please notice (try it!) that if you don't use the '$' sign, the output of the program will be different, and probably not what you want it to be.

Simple Backup Made Better

```
#!/bin/bash
OF=/var/my-backup-$(date +%Y%m%d).tgz
tar -cZf $OF /home/me/
```

This script introduces another thing. First of all, you should be familiarized with the variable creation and assignation on line 2. Notice the expression '$(date +%Y%m%d)'. If you run the script you'll notice that it runs the command inside the parenthesis, capturing its output.

Notice that in this script, the output filename will be different every day, due to the format switch to the date command(+%Y%m%d). You can change this by specifying a different format.

Some more examples:

```
echo ls
echo $(ls)
```

Local Variables

Local variables can be created by using the keyword *local*.

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

```
HELLO=Hello
function hello {
      local HELLO=World
      echo $HELLO
}
echo $HELLO
hello
echo $HELLO
```

This example should be enough to show how to use a local variable.

Conditionals

Conditionals let you decide whether to perform an action or not, this decision is taken by evaluating an expression.

Dry Theory

Conditionals have many forms. The most basic form is: **if** *expression* **then** *statement* where 'statement' is only executed if 'expression' evaluates to true. '2<1' is an expression that evaluates to false, while '2>1' evaluates to true.xs

Conditionals have other forms such as: **if** *expression* **then** *statement1* **else** *statement2*. Here 'statement1' is executed if 'expression' is true,otherwise 'statement2' is executed.

Yet another form of conditionals is: **if** *expression1* **then** *statement1* **else if** *expression2* **then** *statement2* **else** *statement3*. In this form there's added only the "ELSE IF 'expression2' THEN 'statement2'" which makes statement2 being executed if expression2 evaluates to true. The rest is as you may imagine (see previous forms).

A word about syntax:
The base for the 'if' constructions in bash is this:

```
if [expression];
then
code if 'expression' is true.
fi
```

if .. then

```
#!/bin/bash
if [ "foo" = "foo" ]; then
      echo expression evaluated as true
fi
```

The code to be executed if the expression within braces is true can be found after the 'then' word and before 'fi' which indicates the end of the conditionally executed code.

if .. then … else

```
#!/bin/bash
if [ "foo" = "foo" ]; then
```

```
        echo expression evaluated as true
else
        echo expression evaluated as false
fi
```

Conditionals with Variables

```
#!/bin/bash
T1="foo"
T2="bar"
if [ "$T1" = "$T2" ]; then
        echo expression evaluated as true
else
        echo expression evaluated as false
fi
```

Loops

In this section you'll find for, while and until loops.

The **for** loop is a little bit different from other programming languages. Basically, it let's you iterate over a series of 'words' within a string.

The **while** executes a piece of code if the control expression is true, and only stops when it is false (or a explicit break is found within the executed code.

The **until** loop is almost equal to the while loop, except that the code is executed while the control expression evaluates to false.

for

```
#!/bin/bash
for i in $( ls ); do
        echo item: $i
done
```

On the second line, we declare i to be the variable that will take the different values contained in $(ls).

The third line could be longer if needed, or there could be more lines before the done (4).

'done' (4) indicates that the code that used the value of $i has finished and $i can take a new value.

This script has very little sense, but a more useful way to use the for loop would be to use it to match only certain files on the previous example

C-like for

You can also use this form of looping. It's a for loop more similar to C/perl... for.

```
#!/bin/bash
for i in `seq 1 10`;
```

```
do
      echo $i
done
```

while

```
#!/bin/bash
COUNTER=0
while [  $COUNTER -lt 10 ]; do
      echo The counter is $COUNTER
      let COUNTER=COUNTER+1
done
```

This script 'emulates' the well known (C, Pascal, perl, etc) 'for' structure

until

```
#!/bin/bash
COUNTER=20
until [  $COUNTER -lt 10 ]; do
      echo COUNTER $COUNTER
      let COUNTER-=1
done
```

Functions

As in almost any programming language, you can use functions to group pieces of code in a more logical way or practice the divine art of recursion.

Declaring a function is just a matter of writing function my_func { my_code }.

Calling a function is just like calling another program, you just write its name.

No Params

```
#!/bin/bash
function quit {
      exit
}
function hello {
      echo Hello!
}
hello
quit
echo foo
```

Lines 2-4 contain the 'quit' function. Lines 5-7 contain the 'hello' function If you are not absolutely sure about what this script does, please try it!.

Notice that a functions don't need to be declared in any specific order.

When running the script you'll notice that first: the function 'hello' is called, second the 'quit' function, and the program never reaches line 10.

With Params

```
#!/bin/bash
function quit {
      exit
}
function e {
      echo $1
}
e Hello
e World
quit
echo foo
```

This script is almost identically to the previous one. The main difference is the function 'e'. This function, prints the first argument it receives. Arguments, within functions, are treated in the same manner as arguments given to the script.

User Interfaces

Using select to Make Simple Menus

```
#!/bin/bash
OPTIONS="Hello Quit"
select opt in $OPTIONS; do
      if [ "$opt" = "Quit" ]; then
            echo done
            exit
      elif [ "$opt" = "Hello" ]; then
            echo Hello World
      else
            clear
            echo bad option
      fi
done
```

If you run this script you'll see that it is a programmer's dream for text based menus. You'll probably notice that it's very similar to the 'for' construction, only rather than looping for each 'word' in $OPTIONS, it prompts the user.

Using the Command Line

```
#!/bin/bash
if [ -z "$1" ]; then
      echo usage: $0 directory
      exit
fi
SRCD=$1
```

```
TGTD="/var/backups/"
OF=home-$(date +%Y%m%d).tgz
tar -cZf $TGTD$OF $SRCD
```

What this script does should be clear to you. The expression in the first conditional tests if the program has received an argument ($1) and quits if it didn't, showing the user a little usage message. The rest of the script should be clear at this point.

Miscellaneous

Reading user input with read

In many locations you may want to prompt the user for some input, and there are several ways to achieve this. This is one of those ways:

```
#!/bin/bash
echo Please, enter your name
read NAME
echo "Hi $NAME!"
```

As a variant, you can get multiple values with read, this example may clarify this.

```
#!/bin/bash
echo Please, enter your firstname and lastname
read FN LN
echo "Hi! $LN, $FN !"
```

Arithmetic Evaluation

On the command line (or a shell) try this:

```
echo 1 + 1
```

If you expected to see '2' you'll be disappointed. What if you want BASH to evaluate some numbers you have? The solution is this:

```
echo $((1+1))
```

This will produce a more 'logical' output. This is to evaluate an arithmetic expression. You can achieve this also like this:

```
echo $[1+1]
```

If you need to use fractions, or more math or you just want it, you can use bc to evaluate arithmetic expressions.

if i ran "`echo $[3/4]`" at the command prompt, it would return 0 because bash only uses integers when answering. If you ran "`echo 3/4|bc -l`", it would properly return 0.75.

Getting the Return Value of a Program

In bash, the return value of a program is stored in a special variable called $?.

This illustrates how to capture the return value of a program, I assume that the directory *dada*does not exist. (This was also suggested by mike)

```
#!/bin/bash
cd /dada &> /dev/null
echo rv: $?
cd $(pwd) &> /dev/null
echo rv: $?
```

Capturing a Command's Output

This little scripts show all tables from all databases (assuming you got MySQL installed). Also, consider changing the 'mysql' command to use a valid username and password.

```
#!/bin/bash
DBS=`mysql -uroot  -e"show databases"`
for b in $DBS ;
do
      mysql -uroot -e"show tables from $b"
done
```

Tables

String Comparison Operators

s1 = s2
s1 != s2
s1 < s2
s1 > s2
-n s1
-z s1
s1 matches s2
s1 does not match s2
s1 is not null (contains one or more characters)
s1 is null

String Comparison Examples

Comparing two strings.

```
#!/bin/bash
S1='string'
S2='String'
if [ $S1=$S2 ];
then
      echo "S1('$S1') is not equal to S2('$S2')"
fi
if [ $S1=$S1 ];
then
      echo "S1('$S1') is equal to S1('$S1')"
fi
```

Arithmetic Operators

+

-
*
/
% (remainder)

Arithmetic Relational Operators

-lt (<)
-gt (>)
-le (<=)
-ge (>=)
-eq (==)
-ne (!=)

Useful Commands

Some of these command's almost contain complete programming languages. From those commands only the basics will be explained. For a more detailed description, have a closer look at the man pages of each command.

sed *(stream editor)*

Sed is a non-interactive editor. Instead of altering a file by moving the cursor on the screen, you use a script of editing instructions to sed, plus the name of the file to edit. You can also describe sed as a filter. Let's have a look at some examples:

```
$sed 's/to_be_replaced/replaced/g' /tmp/dummy
```

Sed replaces the string 'to_be_replaced' with the string 'replaced' and reads from the /tmp/dummy file. The result will be sent to stdout (normally the console) but you can also add '> capture' to the end of the line above so that sed sends the output to the file 'capture'.

```
$sed 12, 18d /tmp/dummy
```

Sed shows all lines except lines 12 to 18. The original file is not altered by this command.

awk *(manipulation of datafiles, text retrieval and processing)*

Many implementations of the AWK programming language exist (most known interpreters are GNU's gawk and 'new awk' mawk.) The principle is simple: AWK scans for a pattern, and for every matching pattern a action will be performed.

Again, I've created a dummy file containing the following lines:

"test123
test
tteesstt"

```
$awk '/test/ {print}' /tmp/dummy
```

test123
test

The pattern AWK looks for is 'test' and the action it performs when it found a line in the file /tmp/dummy with the string 'test' is 'print'.

```
$awk '/test/ {i=i+1} END {print i}' /tmp/dummy
```

When you're searching for many patterns, you should replace the text between the quotes with '-f file.awk' so you can put all patterns and actions in 'file.awk'.

grep *(print lines matching a search pattern)*
We've already seen quite a few grep commands in the previous chapters, that display the lines matching a pattern. But grep can do more.

```
$grep "look for this" /var/log/messages -c
```

The string "look for this" has been found 12 times in the file /var/log/messages.

[ok, this example was a fake, the /var/log/messages was tweaked :-)]

wc *(counts lines, words and bytes)*
In the following example, we see that the output is not what we expected. The dummy file, as used in this example, contains the following text: *"bash introduction howto test file"*

```
$wc --words --lines --bytes /tmp/dummy
```

2 5 34 /tmp/dummy

Wc doesn't care about the parameter order. Wc always prints them in a standard order, which is, as you can see: .

sort *(sort lines of text files)*
This time the dummy file contains the following text:
"b
c
a"

```
$sort /tmp/dummy
```

This is what the output looks like:
a
b
c

bc *(a calculator programming language)*
Bc is accepting calculations from command line (input from file. not from redirector or pipe), but also from a user interface. The following demonstration shows some of the commands. Note that

I start bc using the -q parameter to avoid a welcome message.

```
$bc -q
```

1 == 5
0
0.05 == 0.05

1
5 != 5
0
2 ^ 8
256
sqrt(9)
while (i != 9) {
i = i + 1;
print i
}
123456789
quit

tput *(initialize a terminal or query terminfo database)*
A little demonstration of tput's capabilities:

```
$tput cup 10 4
```

The prompt appears at (y10, x4).

```
$tput reset
```

Clears screen and prompt appears at (y1,x1). Note that (y0,x0) is the upper left corner.

```
$tput cols
```

Shows the number of characters possible in x direction.

It it highly recommended to be familiarized with these programs (at least). There are tons of little programs that will let you do real magic on the command line.

More Scripts

Simple Backup Made Even Better

```
#!/bin/bash
SRCD="/home/"
TGTD="/var/backups/"
OF=home-$(date +%Y%m%d).tgz
tar -cZf $TGTD$OF $SRCD
```

File Renamer

```
#!/bin/sh
# renna: rename multiple files according to several rules
# written by felix hudson Jan - 2000

#first check for the various 'modes' that this program has
#if the first ($1) condition matches then we execute that portion of
the
#program and then exit
```

```
# check for the prefix condition
if [ $1 = p ]; then

#we now get rid of the mode ($1) variable and prefix ($2)
prefix=$2 ; shift ; shift

# a quick check to see if any files were given
# if none then its better not to do anything than rename some
non-existent
# files!!

if [$1 = ]; then
      echo "no files given"
      exit 0
fi
# this for loop iterates through all of the files that we gave the
program
# it does one rename per file given
for file in $*
do
      mv ${file} $prefix$file
done

#we now exit the program
      exit 0
fi

# check for a suffix rename
# the rest of this part is virtually identical to the previous
section
# please see those notes
if [ $1 = s ]; then
      suffix=$2 ; shift ; shift

      if [$1 = ]; then
            echo "no files given"
            exit 0
      fi

      for file in $*
      do
            mv ${file} $file$suffix
      done

      exit 0
```

```
fi
# check for the replacement rename
if [ $1 = r ]; then
shift
# i included this bit as to not damage any files if the user does not
specify
# anything to be done
# just a safety measure

if [ $# -lt 3 ] ; then
      echo "usage: renna r [expression] [replacement] files... "
      exit 0
fi
# remove other information
OLD=$1 ; NEW=$2 ; shift ; shift
# this for loop iterates through all of the files that we give the
program
# it does one rename per file given using the program 'sed'
# this is a simple command line program that parses standard input
and
# replaces a set expression with a give string
# here we pass it the filename ( as standard input) and replace the
necessary
# text
for file in $*
do
      new=`echo ${file} | sed s/${OLD}/${NEW}/g`
      mv ${file} $new
done
exit 0
fi
# if we have reached here then nothing proper was passed to the
program
# so we tell the user how to use it
echo "usage;"
echo " renna p [prefix] files.."
echo " renna s [suffix] files.."
echo " renna r [expression] [replacement] files.."
exit 0
# done!
```

File Renamer Simple

```
#!/bin/bash
# renames.sh
# basic file renamer
criteria=$1
```

```
re_match=$2
replace=$3

for i in $( ls *$criteria* );
do
      src=$i
      tgt=$(echo $i | sed -e "s/$re_match/$replace/")
      mv $src $tgt
done
```

Debugging

Ways Calling BASH

A nice thing to do is to add on the first line

```
#!/bin/bash -x
```

This will produce some interesting output information

Colors!

The Linux terminal supports special formatting of words that it outputs. You can use this like:

```
echo "\033[(code)m Example"
```

You can use multiple types of customization by using many codes separated by semicolons. This way you could get something like bold red text.

Formatting		Text Color		Background Color	
Normal	0	Default	39	Default	49
Bold	1	Black	30	Black	40
Dim	2	Red	31	Red	41
Underlined	4	Green	32	Green	42
Inverted	7	Yellow	33	Yellow	43
Hidden	8	Blue	34	Blue	44
		Magenta	35	Magenta	45
		Cyan	36	Cyan	46
		Light Grey	37	Light Grey	47
		Dark Grey	90	Dark Grey	100
		Light Red	91	Light Red	101
		Light Green	92	Light Green	102
		Light Yellow	93	Light Yellow	103
		Light Blue	94	Light Blue	104
		Light Magenta	95	Light Magenta	105
		Light Cyan	96	Light Cyan	106
		White	97	White	107