Bash Scripting
Lesson 2

Working with Variables and Parameters

About Terminology

- An *argument* is anything that can be used after a command
  - **ls –l /etc** has two arguments
- An *option* is an argument that was specifically developed to change the command behavior
- A *parameter* is a name that is defined in a script to which a specific value is granted
- A *variable* is a label that is stored in memory and contains a specific value

Using and Defining Variables

- Using variables makes a script flexible
- They allow for dynamic scripts that can easily be modified to act on different values
- Let's have a look at an example

For eg:

```
#!/bin/bash
# This script copies /var/log contents and clears current
# contents of the file
# Usage: ./clearlogs

LOGFILE=/var/log/messages

cp $LOGFILE $LOGFILE.old
cat /dev/null > $LOGFILE
echo log file copied and cleaned up

exit 0
```

Defining Variables ways:

- There are three ways to define variables
  - Static: VARNAME=value
  - As an argument to a script, handled using $1, $2 etc. within the script
  - Interactively, using **read**
- Best practice: use uppercase only for variable names
- Examples of these are in the next sections

```
[root@server1 ~]# echo $COLOR

[root@server1 ~]# COLOR=blue
[root@server1 ~]# echo $COLOR
blue
[root@server1 ~]# bash
[root@server1 ~]# echo $COLOR

[root@server1 ~]# exit
exit
[root@server1 ~]# echo $COLOR
blue
[root@server1 ~]# export COLOR=red
[root@server1 ~]# echo $COLOR
red
[root@server1 ~]# bash
[root@server1 ~]# echo $COLOR
red
[root@server1 ~]# exit
exit
[root@server1 ~]#
```

Defining Variables with the read commands

```
#!/bin/bash
#
# Ask what to stop using the kill command and then kill it

echo Which process do you want to kill?
read TOKILL

kill $(ps aux | grep $TOKILL | grep –v grep | awk '{ print $2 }')
```

```
[root@server1 bin]# ps aux | grep http
apache    7068  0.0  0.8 241468  4464 ?        S    Jan10   0:00 /usr/sbin/httpd -DFOREGROUND
apache    7069  0.0  0.8 241468  4464 ?        S    Jan10   0:00 /usr/sbin/httpd -DFOREGROUND
apache    7070  0.0  0.8 241468  4464 ?        S    Jan10   0:00 /usr/sbin/httpd -DFOREGROUND
apache    7071  0.0  0.8 241468  4464 ?        S    Jan10   0:00 /usr/sbin/httpd -DFOREGROUND
apache    7072  0.0  0.9 241468  4956 ?        S    Jan10   0:01 /usr/sbin/httpd -DFOREGROUND
root     15532  0.0  0.1 112640   932 pts/0    R+   14:49   0:00 grep --color=auto http
root     20888  0.0  1.4 237152  7388 ?        Ss    2015   4:06 /usr/sbin/httpd -DFOREGROUND
[root@server1 bin]# ps aux | grep http | awk '{ print $2 }'
7068
7069
7070
7071
7072
15544
20888
[root@server1 bin]# ps aux | grep http | awk '{ print $2}'
```

```
postfix  15407  0.0  0.7  95360  3864 ?        S    14:42   0:00 pickup -l -t unix -u
root     15510  0.0  0.0      0     0 ?        R    14:47   0:00 [kworker/0:0]
root     15521  0.0  0.2 373936  1412 ?        S    14:48   0:00 /usr/sbin/smbd
root     15529  0.0  0.1 107892   616 ?        S    14:48   0:00 sleep 60
root     15530  0.0  0.2 123372  1328 pts/0    R+   14:49   0:00 ps aux
root     20888  0.0  1.4 237152  7388 ?        Ss    2015   4:06 /usr/sbin/httpd -DFOREGROUND
unbound  21348  0.0  1.6 141172  8176 ?        Ssl   2015   0:00 /usr/sbin/unbound -d
[root@server1 bin]# ps aux | grep http
apache    7068  0.0  0.8 241468  4464 ?        S    Jan10   0:00 /usr/sbin/httpd -DFOREGROUND
apache    7069  0.0  0.8 241468  4464 ?        S    Jan10   0:00 /usr/sbin/httpd -DFOREGROUND
apache    7070  0.0  0.8 241468  4464 ?        S    Jan10   0:00 /usr/sbin/httpd -DFOREGROUND
apache    7071  0.0  0.8 241468  4464 ?        S    Jan10   0:00 /usr/sbin/httpd -DFOREGROUND
apache    7072  0.0  0.9 241468  4956 ?        S    Jan10   0:01 /usr/sbin/httpd -DFOREGROUND
root     15532  0.0  0.1 112640   932 pts/0    R+   14:49   0:00 grep --color=auto http
root     20888  0.0  1.4 237152  7388 ?        Ss    2015   4:06 /usr/sbin/httpd -DFOREGROUND
[root@server1 bin]# ps aux | grep http | awk '{ print $2 }'
7068
7069
7070
7071
7072
15544
20888
[root@server1 bin]# ps aux | grep http | grep -v grep | awk '{ print $2 }'
7068
7069
7070
7071
7072
20888
[root@server1 bin]# kill $(ps aux | grep http | grep -v grep | awk '{ print $2 }')
[root@server1 bin]# ps aux | grep http
root     15578  0.0  0.1 112640   932 pts/0    R+   14:51   0:00 grep --color=auto http
[root@server1 bin]#
```

```
#!/bin/bash
#
# Ask what to stop using the kill command and then kill it

echo which process do you want to kill
read TOKILL

kill $(ps aux | grep $TOKILL | grep -v grep | awk '{ print $2 }')
~
```

```
[root@server1 bin]# vim tokill
[root@server1 bin]# tokill
which process do you want to kill
http
kill: usage: kill [-s sigspec | -n signum | -sigspec] pid | jobspec ... or kill -l [sigspec]
[root@server1 bin]# bash -x tokill
+ echo which process do you want to kill
which process do you want to kill
+ read TOKILL
http
++ ps aux
++ grep http
++ grep -v grep
++ awk '{ print $2 }'
+ kill
kill: usage: kill [-s sigspec | -n signum | -sigspec] pid | jobspec ... or kill -l [sigspec]
[root@server1 bin]# ps aux | grep http | grep -v grep | awk '{ print $2 }'
[root@server1 bin]# systemctl start httpd
[root@server1 bin]#
```

- **read** will stop the script
- If an argument is provided to **read**, this argument will be treated as a variable and the entered value is stored in the variable
  - Multiple arguments can be provided to enter multiple variables
  - Use **read –a somename** to write all words to an array with the name somename. (Arrays are treated in more detail later)
- If no input is provided, the script will just pause until the user presses the Enter key

Understanding variables and subshells

- A variable is effective only in the shell where it was defined
- Use **export** to make it also available in subshells
- There is no way to make variables available in parent shells
- The following script provides an example

```
#!/bin/bash
#
# Showing variable use between shells

echo which directory do you want to activate
read DIR

cd $DIR
pwd
ls

exit 0
```

```
[root@server1 bin]# tokill
which process do you want to kill
http
[root@server1 bin]# vim todir
[root@server1 bin]# vim todir
[root@server1 bin]# chmod +x todir
[root@server1 bin]# todir
which directory do you want to activate
/home
/home
boris  linda  lisa  mike  paul  user
[root@server1 bin]# vim todir
[root@server1 bin]#
```

Define a variables at the start of the shell

- /etc/profile is processed when opening a login shell.
  - All variables defined here are available in all subshells for that specific user
  - User specific version can be used in ~/.bash_profile
- /etc/bashrc is processed when opening a subshell
  - Variables defined here are included in subshells from that point on
  - User specific versions can be used in ~/.bashrc

While running the script we are in the subshell opening a new environment for running a script it is using exit 0 for get back to the parent shell.

Sourcing

- By using sourcing the contents of one script can be included in another script
- This is a very common method to separate static script code from dynamic content
  - This dynamic content often consists of variables and functions
- Use the **source** command or the . command to source scripts
- Do NOT use **exit** at the end of a script that needs to be sourced

- Master script

```
#!/bin/bash
# Example script to show how sourcing works

. /root/sourceme

echo the value of the variable '$COLOR' is $COLOR

exit 0
```

- Input script /root/sourceme

```
COLOR=red
```

Eg:

```
[root@server1 bin]# vim master
[root@server1 bin]# cat master
#!/bin/bash
# Example script to show how sourcing works

. /root/sourceme

echo the value of the variable '$COLOR' is $COLOR

exit 0

[root@server1 bin]# vim /root/sourceme
[root@server1 bin]# chmod +x master
[root@server1 bin]# master
the value of the variable $COLOR is yellow
[root@server1 bin]#
```

```
[root@server1 bin]# cd /etc/init.d
[root@server1 init.d]# ls
functions  netconsole  network  README
[root@server1 init.d]# vim netconsole
```

Source the content of other file into our file:

```
#!/bin/bash
#
# netconsole    This loads the netconsole module with the conf
#
# chkconfig: - 50 50
# description: Initializes network console logging
# config: /etc/sysconfig/netconsole
#
# Copyright 2002 Red Hat, Inc.
#
# Based in part on a shell script by
# Andreas Dilger <adilger@turbolinux.com>  Sep 26, 2001

PATH=/sbin:/usr/sbin:$PATH
RETVAL=0
SERVER_ADDRESS_RESOLUTION=

# Check that networking is up.
. /etc/sysconfig/network

# Source function library.
. /etc/rc.d/init.d/functions

# Default values
LOCALPORT=6666
DEV=
```

```
[root@server1 bin]# cd /etc/init.d
[root@server1 init.d]# ls
functions  netconsole  network  README
[root@server1 init.d]# vim netconsole
[root@server1 init.d]# vim /etc/sysconfig/network
[root@server1 init.d]# vim netconsole
[root@server1 init.d]# vim /etc/rc.d/init.d/functions
[root@server1 init.d]# cd
[root@server1 ~]# cd bin
[root@server1 bin]# cd ..
[root@server1 ~]# cd bin/
[root@server1 bin]# ls
clearlogs  hello  master  test  todir  tokill
[root@server1 bin]#
```

```
[root@server1 ~]# cd bin
[root@server1 bin]# vim todir
[root@server1 bin]# . todir
which directory do you want to activate
/home
/home
boris  linda  lisa  mike  paul  user
Connection to 192.168.122.210 closed.
[root@lab ~]# ssh 192.168.122.210
root@192.168.122.210's password:
Last login: Thu Jan 14 16:12:33 2016 from 192.168.122.1
[root@server1 ~]#
```

```
#!/bin/bash
#
# Showing variable use between shells

echo which directory do you want to activate
read DIR

cd $DIR
pwd
ls

exit 0
```

Due to the exit statement that's why it closed the current shell when I run the source . todir

Quoting

- The bash shell uses different special characters

Character	Meaning
~	Home directory
`	Command substitution
#	Comment
$	Variable expression
&	Background job
*	String wildcard
(	Start subshell
)	End subshell
\	Quote next character

Character	Meaning
|	Pipe
[	Start character set wildcard
]	End character set wildcard
{	Start command block
}	End command block
;	Shell command separator
'	Strong quote
"	weak quote
<	Input redirect
>	Output redirect
/	Pathname directory separator
?	Single character wildcard

```
[root@server1 bin]# echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/root/bin
[root@server1 bin]# echo The value of the $PATH variable is $PATH
The value of the /usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/root/bi
in:/usr/bin:/root/bin
[root@server1 bin]#
```

The path print twice times that's why we need to do a coding we need to use the special quotes.

Understanding about the command line parsing

- When a command is interpreted by the shell, the shell interprets all special characters
  - This process is known as *command line parsing*
- Commands themselves may interpret parts of the command line as well
- To ensure that a special character is interpreted by the command and not by the shell, use quoting

Quoting

- Quoting is used to treat special characters literally
- If a string of characters is surrounded with single quotation marks, all characters are stripped of the special meaning they may have
  - Imagine **echo 2 * 3 > 5**, which would be interpreted
  - Or imagine **find . –name '*.doc'** which ensures that **find** interprets *.doc and not the shell
- Double quotes are *weak quotes* and treat only some special characters as special
- A backslash can be used to escape the one following character

```
[root@server1 bin]# echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/root/bin
[root@server1 bin]# echo The value of the $PATH variable is $PATH
The value of the /usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/root/bin vari
in:/usr/bin:/root/bin
[root@server1 bin]# echo 'The value of the $PATH variable is $PATH'
The value of the $PATH variable is $PATH
[root@server1 bin]# echo 2 * 3 > 5
[root@server1 bin]# ls
5  clearlogs  hello  master  test  todir  tokill
[root@server1 bin]# cat 5
2 clearlogs hello master test todir tokill 3
[root@server1 bin]# echo '2 * 3 > 5'
2 * 3 > 5
[root@server1 bin]#
```

Using a double quote

- Double quotes ignore pipe characters, aliases, tilde substitution, wildcard expression, and splitting into words using delimiters
- Double quotes *do* allow parameter substitution, command substitution, and artithmetic expression evaluation
- Best practice: use single quotes, unless you specifically need parameter, command, or artithmetic substitution

Handing script arguments

- Any argument that was used when starting a script, can be dealt with from within the script
- Use $1, $2 and so on to refer to the first, the second, etc. argument
- $0 refers to the name of the script itself
- Use ${nn} or **shift** to refer to arguments beyond 9
- Arguments that are stored in $1 etc. are read only and cannot be changed from within the script

For eg:

```
#!/bin/bash
#
# Simple demo script with arguments
# Run this script with the names of one or more people

echo "Hello $1 how are you"
echo "Hello $2 how are you"
exit 0
```

```
[root@server1 bin]# vim simplearg
[root@server1 bin]# chmod +x simplearg
[root@server1 bin]# ./simplearg bob pete
Hello bob how are you
Hello pete how are you
[root@server1 bin]# ./simplearg bob
Hello bob how are you
Hello  how are you
[root@server1 bin]# ./simplearg bob pete lisa linda lori
Hello bob how are you
Hello pete how are you
[root@server1 bin]# ./simplearg bob pete
```

Handling the argument in the smart way

- The previous example works only if the amount of arguments is known on beforehand
- If this is not the case, use **for** to evaluate all possible arguments
- Use $@ to refer to all arguments provided, where all arguments can be treated one by one
- Use $# to count the amount of arguments provided
- Use $* if you need a single string that contains all arguments (use in specific cases only)

For egs:

```
#!/bin/bash
# Script that shows how arguments are handled

echo "\$* gives $*"
echo "\$# gives $#"
echo "\$@ gives $@"
echo "\$0 is $0"

#Showing the interpretation of \$*
for i in "$*"
do
        echo $i
done

#Showing the interpretation of \$@
for j in "$@"
do
        echo $j
done
exit 0
```

Handling user input through arguments or inputs

```
#!/bin/bash
# Script that shows how to make sure that user input is provided

if [ -z $1 ]
then
        echo provide filenames
        read FILENAMES
else
        FILENAMES="$@"
fi

echo the following filenames have been provided: $FILENAMES
```

```
[root@server1 bin]# vim argorread
[root@server1 bin]# chmod +x argorread
[root@server1 bin]# argorread
provide filenames
a b c
the following filenames have been provided:
[root@server1 bin]# argorread a b c
the following filenames have been provided: a b c
[root@server1 bin]#
```

Understanding the need to use shift.

- Shift removes the first argument from a list so that the second argument gets stored in $1
- Shift is useful in older shell versions that do not understand ${10} etc.
- Consider the two following examples

```
#!/bin/bash
#
# Simple demo script with arguments
# Run this script with the names of one or more people

echo "Hello $1 how are you"
echo "Hello $2 how are you"
echo "Hello ${10} how are you"
exit 0
```

Use of the shift operator

```
#!/bin/bash
# Showing the use of shift

echo "The arguments provided are $*"
echo "The value of \$1 is $1"
shift
echo "The new value of \$1 is $1"

exit
```

```
[root@server1 bin]# shiftarg a b c
The arguments provided are a b c
The value of $1 is a
The new value of $1 is b
[root@server1 bin]# vim shift
```

Now we need to do something new as well

```
#!/bin/bash
# Showing the use of shift

echo "The arguments provided are $*"
echo "The value of \$1 is $1"
echo "The entire string is $*"
shift
echo "The new value of \$1 is $1"
echo "The entire strin now is $*"

exit
```

```
[root@server1 bin]# shiftarg a b c
The arguments provided are a b c
The value of $1 is a
The new value of $1 is b
[root@server1 bin]# vim shiftarg
[root@server1 bin]# shiftarg a b c
The arguments provided are a b c
The value of $1 is a
The entire string is a b c
The new value of $1 is b
The entire strin now is b c
[root@server1 bin]#
```

Eg 2 for the variable and argument is :

```
#!/bin/bash
# Provide 10 arguments or more when running this script

echo "Hello $1"
echo "Hello $2"
echo "Hello $10"
echo "Hello ${10}"

exit
```

Using command substitution

- Command substitution allows using the result of a command in a script
- Useful to provide ultimate flexibility
- Two allowed syntaxes:
  - **`command`** (deprecated)
  - **$(command)** (preferred)
- **ls –l $(which passwd)**
- Consider the following example

For egs:

```
#!/bin/bash
# This script copies /var/log contents and clears current
# contents of the file
# Usage: ./clearlogs

cp /var/log/messages /var/log/messages.$(date +%d-%m-%y)
cat /dev/null > /var/log/messages
echo log file copied and cleaned up

exit 0
```

You are creating a log file on the daily basis

```
[root@server1 bin]# date +%d-%m-%y
14-01-16
[root@server1 bin]# touch ~/file-$(date +%d-%m-%y)
[root@server1 bin]# ls -l /root/file-14-01-16
-rw-r--r--. 1 root root 0 Jan 14 17:18 /root/file-14-01-16
[root@server1 bin]#
```

String Verification

- When working with arguments and input, it is useful to be able to verify availability and correct use of a string
- Use **test –z** to check if a string is empty
  - `test –z $1 && exit 1`
- Use **[[...]]** to check for specific patterns
  - `[[ $1 == '[a-z]*' ]] || echo $1 does not start with a letter`
- More about this in Lesson 3

```
[root@server1 bin]# [[ $1 == '[a-z]*' ]] || echo $1 does not start with a letter
does not start with a letter
[root@server1 bin]# ANIMAL=cow
[root@server1 bin]# [[ $ANIMAL == '[a-z]*' ]] || echo $ANIMAL does not start with a letter
```

In the above task it compare with the space that's why it print the echo statement

```
[root@server1 bin]# [[ $ANIMAL == '[a-z]*' ]] || echo $ANIMAL does not start with a letter
cow does not start with a letter
[root@server1 bin]# [[ $ANIMAL=='[a-z]*' ]] || echo $ANIMAL does not start with a letter
[root@server1 bin]#
```

Using Here Documents

- In a here document, I/O redirection is used to feed a command list into an interactive program or command, such as for instance **ftp** or **cat**
- Use it in scripts to replace echo for long texts that need to be displayed
- Use it if in a script a command is called that opens its own prompt, such as an FTP client interface

```
#!/bin/bash
# script that shows how here documents can be used
# as an alternative to the echo command

cat <<End-of-message
-----------------------------------
This is line 1 of the message
This is line 2 of the message
This is the last line of the message
-----------------------------------
End-of-message
```

Output

```
[root@server1 bin]# vim heredoc1
[root@server1 bin]# chmod +x heredoc1
[root@server1 bin]# heredoc1
-----------------------------------
this is line 1 of the message
this is line 2 of the message
this is the last line of the message
-----------------------------------
[root@server1 bin]#
```

Other egs:

```
#!/bin/bash
# script that shows how text to be sent by wall
# is feeded through a here document
wall <<EndOfMessage
The menu for today is:
1.      Healthy soup and salad
2.      Chips and fish
3.      Steak with carrots
EndOfMessage
```

Eg number : 3

```
#!/bin/bash
# example of a scripted FTP session

lftp localhost <<EndOfSession
ls
get hosts
bye
EndOfSession

echo the file is now downloaded
```

```
[root@server1 bin]# lftp localhost
lftp localhost:~> ls
-rw-r--r--    1 0        0         4139432 Jul 04  2014 nmap-6.40-4.el7.x86_64.rpm
drwxr-xr-x    2 0        0            4096 Nov 26 18:26 repodata
lftp localhost:/> get nmap-6.40-4.el7.x86_64.rpm
4139432 bytes transferred
lftp localhost:/> bye
[root@server1 bin]#
```

Exercise 2

- Write a script with the name totmp. This script should copy all files of which the names are provided as arguments on the command line to the users home directory. If no files have been provided, the script should use **read** to ask for file names, and copy all file names provided in the answer to the users home directory.

Exercise 2 Solution

```
#!/bin/bash
# Script that shows how to make sure that user input is provided

if [ -z $1 ]
then
        echo provide filenames
        read $FILENAMES
else
        FILENAMES="$@"
fi

echo the following filenames have been provided: $FILENAMES
for i in $FILENAMES
do
        cp $i $HOME
done
```