LIKE GEEKS

LINUX

Linux Bash Scripting The Awesome Guide Part3

February 12, 2017 admin 2 Comments

So far you've seen how to write Linux **bash scripts** that interact with data, variables, and files and how to control the flow of the bash script Today we will continue our series about **Linux bash scripting**.

I recommend you to review the previous posts if you want to know what we are talking about.

bash scripting part1

Our main points are

Today we will know how to retrieve input from the user and deal with that input so our script becomes more interactive.

parameters.

Command line parameters allow you to add data values to the command line when you execute the script

```
$ ./myscript 10 20
```

This example passes two command line parameters (10 and 20) to the script. So how to read those parameters in our bash script?

Reading parameters

The bash shell assigns special variables, called positional parameters, to all of the command line parameters entered

- $0 being the script's name
- $1 being the first parameter
- *$2* being the second parameter, and so on, up to *$9* for the ninth parameter

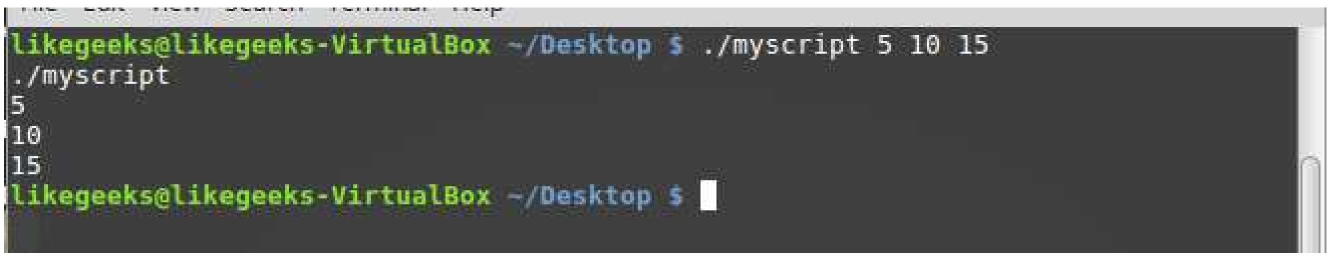
Here's a simple example how to use command line parameter in a shell script

```
#!/bin/bash
echo $0
echo $1
echo $2
echo $3
```

if we run the script with the following parameters and look at the output

```
./myscript 5 10 15
```

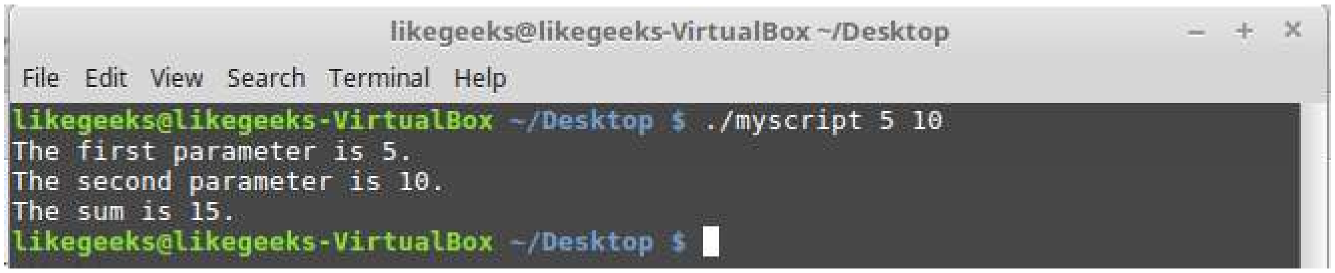
If you need to enter more command line parameters, each parameter must be separated by a space



```
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript 5 10 15
./myscript
5
10
15
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

Here is another example of how we can use two parameters and calculate the sum of them

```
#!/bin/bash

total=$[ $1 + $2 ]

echo The first parameter is $1.

echo The second parameter is $2.

echo The sum is $total.
```



likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help

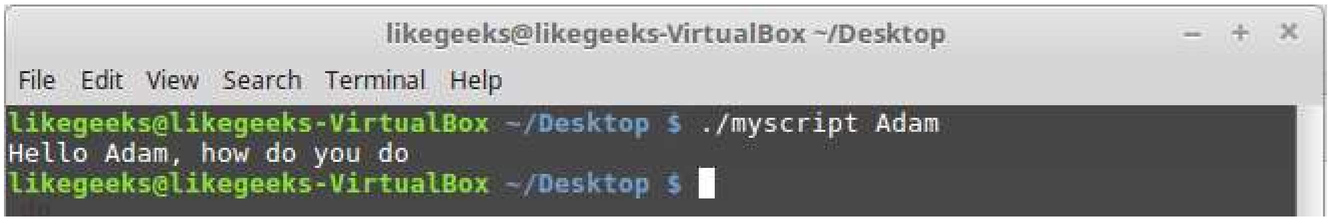
```
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript 5 10
The first parameter is 5.
The second parameter is 10.
The sum is 15.
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

The parameters is not restricted to numbers it could be strings like this

```
#!/bin/bash

echo Hello $1, how do you do
```

```
./myscript Adam
```

And the result is as expected.



likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help

```
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript Adam
Hello Adam, how do you do
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

know the answer from the previous posts. The answer is to use quotations.

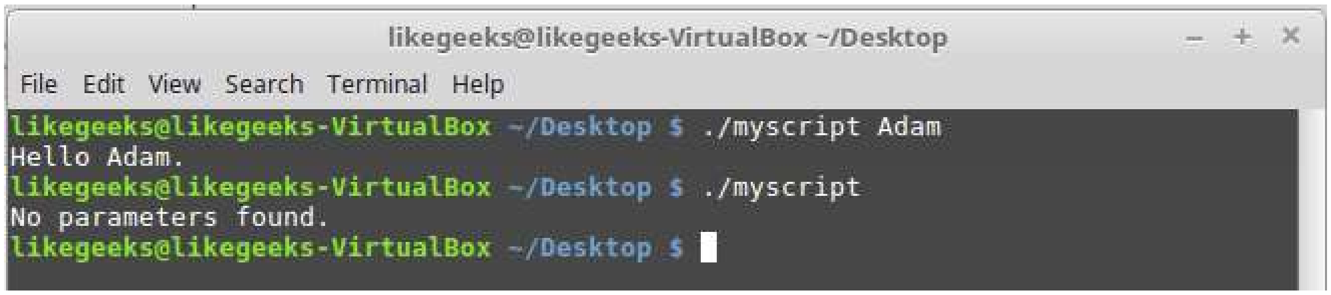
If your script needs more than nine parameters, you must use braces around the variable number, such as ${10}.

Testing parameters

If the script is run without the parameters and your code expecting it, you'll get an error message from your script.

So Always check your parameters to make sure that they exist

```
#!/bin/bash
if [ -n "$1" ]
then
echo Hello $1.
else
echo "No parameters found. "
fi
```


likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help

```
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript Adam
Hello Adam.
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript
No parameters found.
likegeeks@likegeeks-VirtualBox ~/Desktop $ 
```

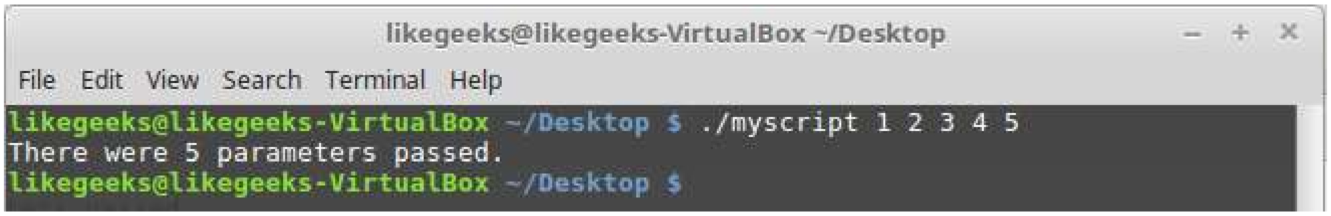
Counting parameters

You can count how many parameters were entered on the command line. The bash shell provides a special variable for this purpose.

The special *$#* variable contains the number of the command line.

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

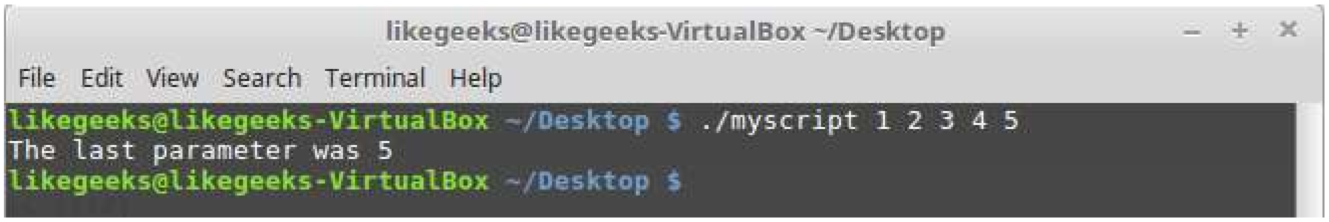
```
./myscript 1 2 3 4 5
```



```
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript 1 2 3 4 5
There were 5 parameters passed.
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

How awesome is Linux bash scripting? this variable also provides a geeky way of getting the last parameter on the command line without having to know how many parameters were used. Look at this trick

```
#!/bin/bash

echo The last parameter was ${!#}
```



```
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript 1 2 3 4 5
The last parameter was 5
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

Grabbing all parameters

In some situations, you want to grab all the parameters provided.

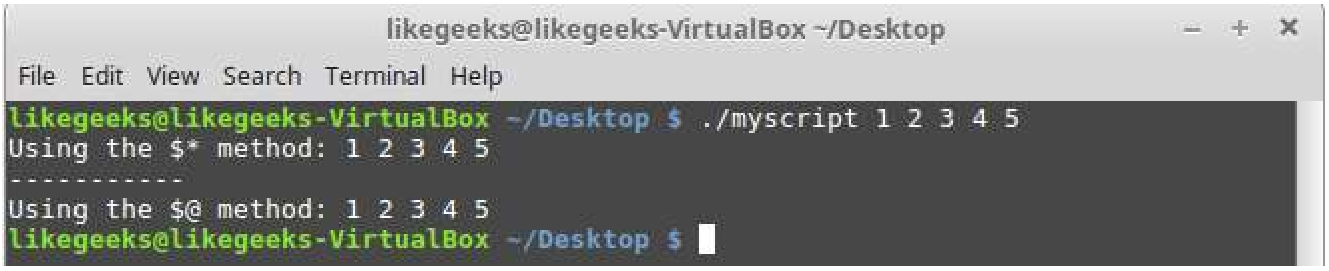
The *$** and *$@* variables in Linux bash scripting provides you all of your parameters. Both of these variables include all the command line parameters within a single variable. So you don't have to grab them by $# variable and iterate over them, just one step

The *$** variable takes all the parameters supplied on the command line as a single word.

The *$@* variable takes all the parameters as separate words in the same string, It allows you to iterate through them

This code shows the difference between them

```
echo "Using the \$* method: $*"

echo "-----------"

echo "Using the \$@ method: $@"
```

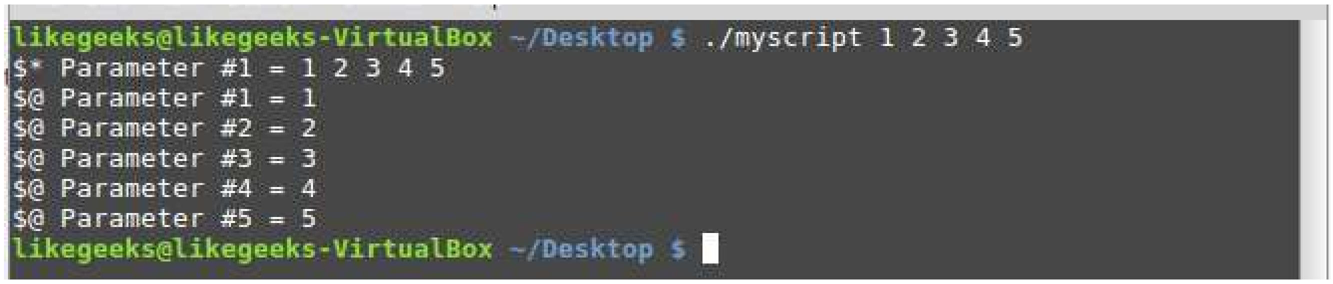


```
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript 1 2 3 4 5
Using the $* method: 1 2 3 4 5
-----------
Using the $@ method: 1 2 3 4 5
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

Both variables produce the same output but if you want to know the difference look at the following example

```
#!/bin/bash

count=1

for param in "$*"

do

echo "\$* Parameter #$count = $param"

count=$(( $count + 1 ))

done

count=1

for param in "$@"

do

echo "\$@ Parameter #$count = $param"

count=$(( $count + 1 ))

done
```

Now you see the difference.



```
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript 1 2 3 4 5
$* Parameter #1 = 1 2 3 4 5
$@ Parameter #1 = 1
$@ Parameter #2 = 2
$@ Parameter #3 = 3
$@ Parameter #4 = 4
$@ Parameter #5 = 5
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

$** variable treated all the parameters as a single parameter, while the *$@ variable treated each parameter separately. So you can use any one of those variables according to your needs

Shift command

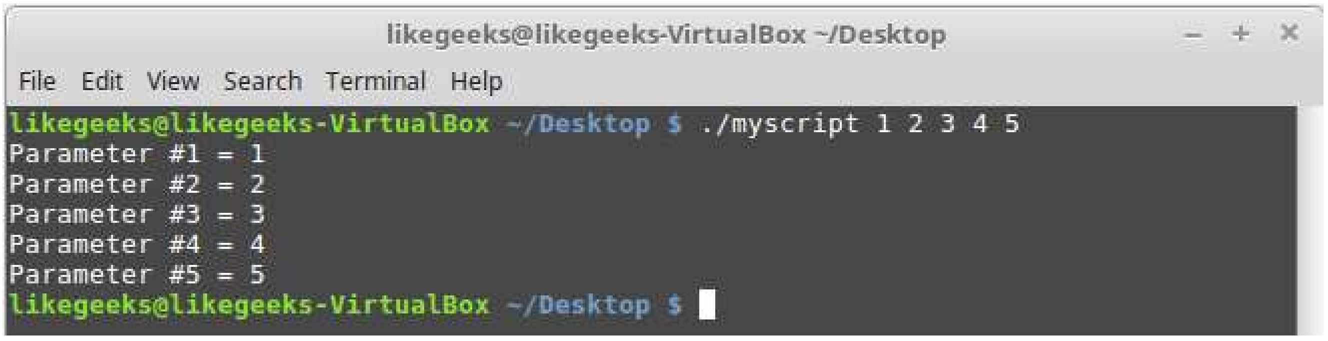
The shift command has some little risk in Linux bash scripting it literally shifts the command line parameters in their relative positions.

When you use the shift command, it moves each parameter variable one position to the left by default. So, the value for variable *$3* is moved to *$2*, the value for variable *$2* is moved to *$1*, and the value for variable *$1* is discarded (note that the value for variable *$0*, the script name, remains unchanged).

This is another great way to iterate through parameters

```
#!/bin/bash
count=1
while [ -n "$1" ]
do
echo "Parameter #$count = $1"
count=$(( $count + 1 ))
shift
done
```

Here the script performs a while loop, checking the length of the first parameter's value. When
The first parameter's length is zero, the loop ends. After testing the first parameter, the s


likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help

```
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript 1 2 3 4 5
Parameter #1 = 1
Parameter #2 = 2
Parameter #3 = 3
Parameter #4 = 4
Parameter #5 = 5
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

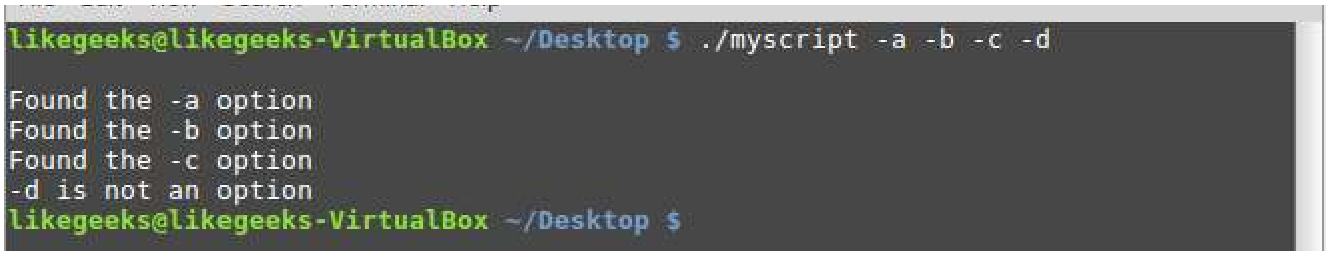
Careful when using shift command because when a parameter is shifted its value is removed and cannot be recovered.

Bash scripting options

Options are single letters preceded by a dash that alters the behavior of a command.

```
#!/bin/bash
echo
while [ -n "$1" ]
do
case "$1" in
-a) echo "Found the -a option" ;;
-b) echo "Found the -b option" ;;
-c) echo "Found the -c option" ;;
*) echo "$1 is not an option" ;;
esac
shift
done
```

```
$ ./myscript -a -b -c -d
```



```
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript -a -b -c -d

Found the -a option
Found the -b option
Found the -c option
-d is not an option
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

The case statement checks each parameter for valid options. When one is found, the appropriate commands are run in the case statement.

Separating options from parameters

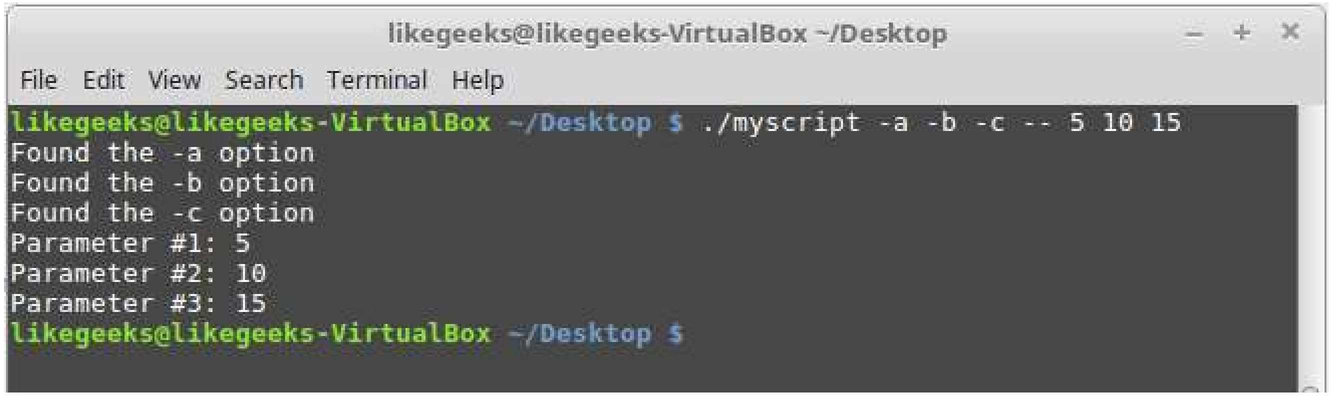
Often in Linux bash scripting, you'll run into situations where you'll want to use both options and parameters for a bash script. The standard way to do this is to separate the two with a special character code that tells the script when the options are finished and when the normal parameters start.

This special character is the double dash (–). The shell uses the double dash to indicate the end of the options list. After seeing the double dash, your script can safely process the remaining command line parameters as parameters and not options

```
#!/bin/bash
while [ -n "$1" ]
do
case "$1" in
-a) echo "Found the -a option" ;;
-b) echo "Found the -b option";;
-c) echo "Found the -c option" ;;
--) shift
break ;;
*) echo "$1 is not an option";;
esac
```

```
done
count=1
for param in $@
do
echo "Parameter #$count: $param"
count=$(( $count + 1 ))
done
```

This bash script uses the break command to break out of the while loop when it encounters the double dash.


likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help

```
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript -a -b -c -- 5 10 15
Found the -a option
Found the -b option
Found the -c option
Parameter #1: 5
Parameter #2: 10
Parameter #3: 15
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

As you can see from the result when the script reaches the double dash, it stops processing options and assumes that any remaining parameters are command line parameters. I love Linux bash scripting.

Processing options with values

When you dig deep onto Linux bash scripting Sometimes you need options with additional parameter value like this

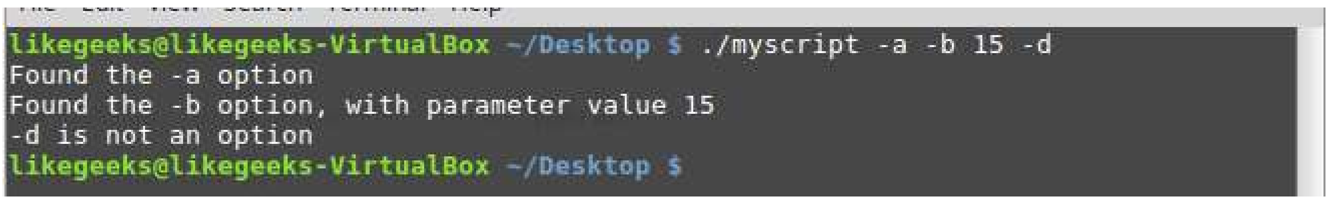
```
./myscript -a test1 -b -c test2
```

Your script must be able to detect when your command line option requires an additional parameter and be able to process it.

```
while [ -n "$1" ]
do
case "$1" in
-a) echo "Found the -a option";;
-b) param="$2"
echo "Found the -b option, with parameter value $param"
shift ;;
-c) echo "Found the -c option";;
--) shift
break ;;
*) echo "$1 is not an option";;
esac
shift
done
count=1
for param in "$@"
do
echo "Parameter #$count: $param"
count=$(( $count + 1 ))
done
```

And if we run it with these options

```
./myscript -a -b test1 -d
```



```
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript -a -b 15 -d
Found the -a option
Found the -b option, with parameter value 15
-d is not an option
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

In this example, the case statement defines three options that it processes.

The -b option also requires an additional parameter value. Because the parameter being processed is *$1*, you know that the additional parameter value is located in *$2* (because all the parameters are shifted after they are processed). Just extract the parameter value from the *$2* variable. Of course, because we used two parameter spots for this option, you also need to set the shift command to shift one additional position.

Well, that works well but there are limitations. For example, this doesn't work if you try to combine multiple options in one parameter like this

```
./myscript –abc
```

Surely it will give you abc is not an option

Fortunately, there's another method for processing options that can help you in this situation

Standardizing Options

When you start your Linux bash scripting, it's completely up to you to choose which letter options you select to use and how you select to use them.

However, a few letter options have achieved a somewhat standard meaning in the Linux world.

And here is the list of the common options

-a	Shows all objects

-d	Specifies a directory
-e	Expands an object
-f	Specifies a file to read data from
-h	Displays a help message for the command
-i	Ignores text case
-l	Produces a long format version of the output
-n	Uses a non-interactive (batch) mode
-o	Specifies an output file to redirect all output to
-q	Runs in quiet mode
-r	Processes directories and files recursively
-s	Runs in silent mode
-v	Produces verbose output
-x	Excludes an object
-y	Answers yes to all questions

If you work with Linux You'll probably recognize most of these option meanings.

Using the same meaning for your options helps users interact with your script without having to worry about manuals.

Providing command line options and parameters is a great way to get data from your bash script users, but sometimes your script needs to be more interactive.

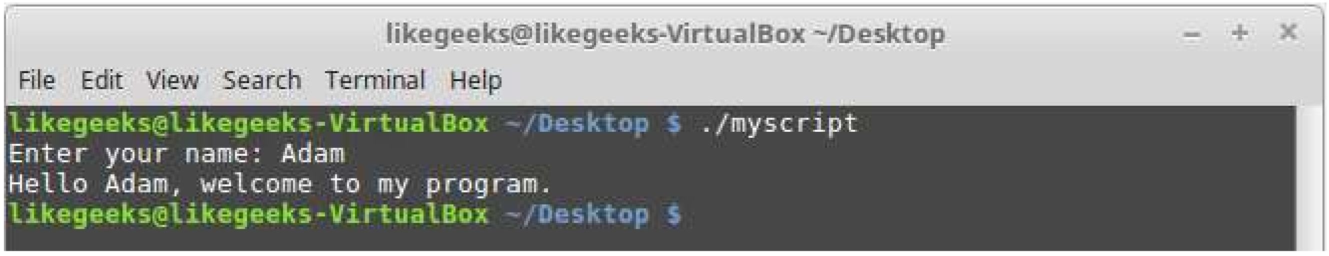
Sometimes you need data from the user while the bash scripting is running.

The bash shell provides the read command just for this purpose.

The read command accepts input either from standard input (the keyboard) or from another file descriptor. After receiving the input, the read command places the data into a variable and here is an example.

```
#!/bin/bash
echo -n "Enter your name: "
read name
echo "Hello $name, welcome to my program."
```

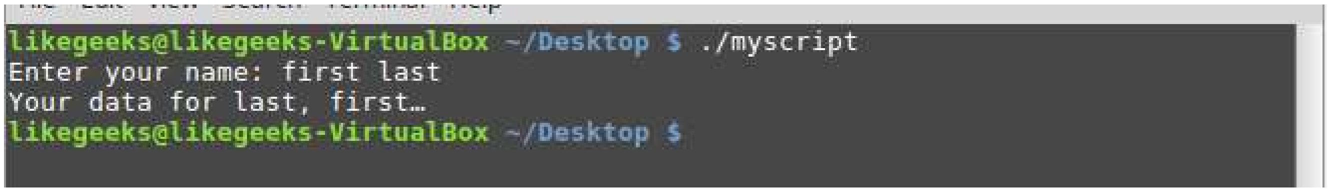
Notice that the echo command that generated the prompt uses the –n option. This prevents the newline character at the end of the string, allowing the script user to enter data immediately after the string, instead of on the next line.


likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help

```
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript
Enter your name: Adam
Hello Adam, welcome to my program.
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

You can specify multiple variables like this

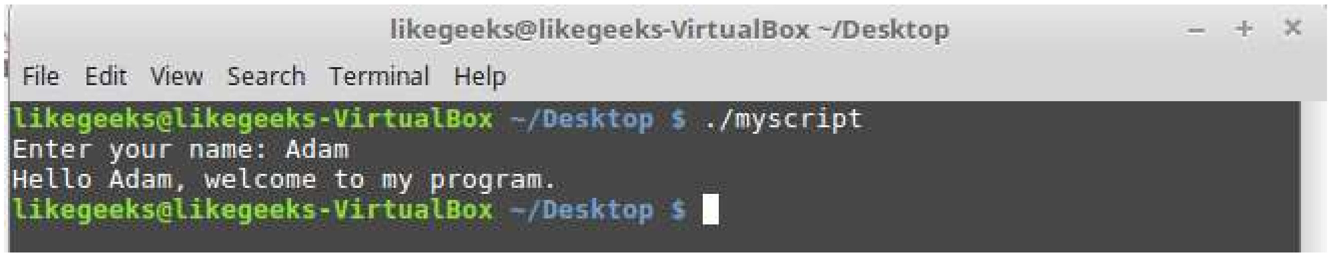
```
#!/bin/bash
read -p "Enter your name: " first last
echo "Your data for $last, $first…"
```



```
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript
Enter your name: first last
Your data for last, first…
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

If you don't specify any variable for read command the read command places any data it receives in the special environment variable REPLY.

```
#!/bin/bash
read -p "Enter your name: "
echo Hello $REPLY, welcome to my program.
```

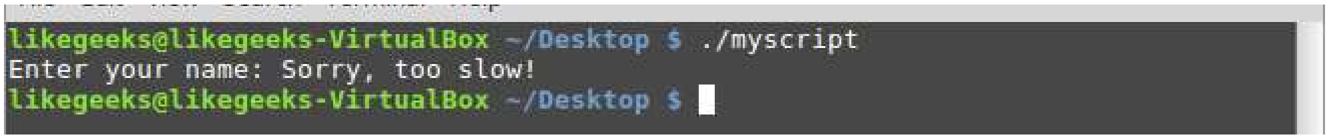


```
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript
Enter your name: Adam
Hello Adam, welcome to my program.
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

If your bash script must go on regardless of whether any data was entered, you can use the -t option to specify a timer. The -t option specifies the number of seconds for the read command to wait for input.

```
#!/bin/bash
if read -t 5 -p "Enter your name: " name
then
echo "Hello $name, welcome to my script"
else
echo "Sorry, too slow! "
fi
```

If you do not enter data for five seconds the script will execute the else clause and print sorry message



```
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript
Enter your name: Sorry, too slow!
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

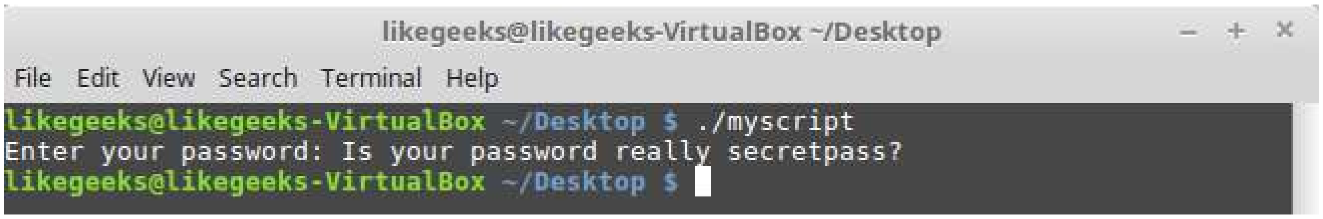
Reading password

In Linux bash scripting Sometimes you don’t want that input to display on the screen like entering a password.

The -s option prevents the data entered in the read command from being displayed on the screen.

The data is displayed, but the read command sets the text color to the same as the background color.

```
#!/bin/bash
read -s -p "Enter your password: " pass
echo "Is your password really $pass? "
```


likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help

```
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript
Enter your password: Is your password really secretpass?
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

Reading from a file

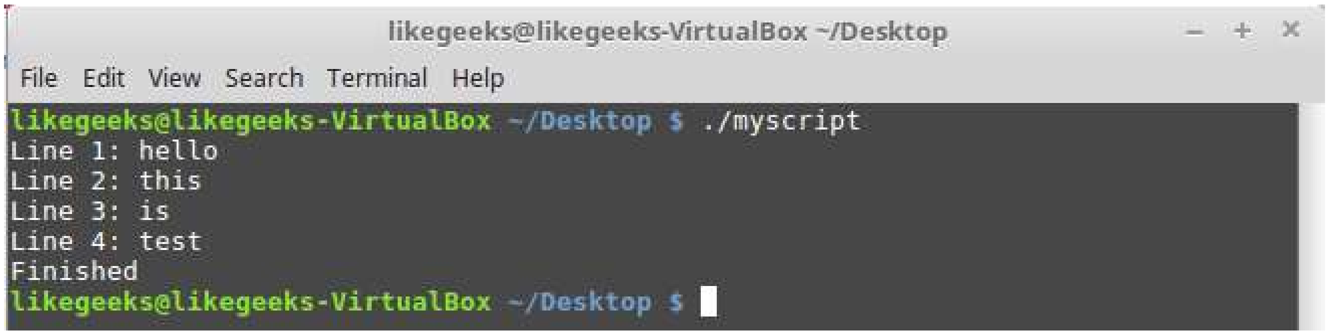
The read command reads a single line of text from the file on each call.

When no more lines are left in the file, the read command just stop.

Now if you want to get all file data you can pipe the result of cat command of the file to while command that contains read command (I know the cat command smells like noob but I want beginners and gurus got my point).

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

```
cat myfile | while read line
do
echo "Line $count: $line"
count=$(( $count + 1 ))
done
echo "Finished"
```



likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help

```
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript
Line 1: hello
Line 2: this
Line 3: is
Line 4: test
Finished
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

We just pass the file content to while loop and iterate over every line and print the line number and the content and each time you increase the count by one simple enough huh?

I hope you find this post interesting and I'm going to make more posts about Linux bash scripting

Thanks.

10

Admin

https://likegeeks.com

RELATED ARTICLES

LINUX

Bash scripting the awesome guide part6 Bash functions

February 17, 2017 admin

Before we talk about bash functions let's discuss this situation. When writing bash scripts, you'll find yourself that you are using the same code in multiple places. If you get tired writing the same blocks of code over and over in your bash script. It would be nice to just write the block of code […]

46

LINUX

Main Linux Commands Easy Guide

January 31, 2017 admin

Main Linux Commands That You Should Know As A Beginner On previous post we discussed how to install Linux now we are going to talk about the most powerful thing in Linux which is Linux commands or shell commands for the whole documentation of Linux Commands, you can check Linux Documentation if you will use Linux […]

0

LINUX

Linux bash scripting the awesome guide part5

February 15, 2017 admin

On the last post, we've talked about input and output and redirection in bash scripting. Now you start building some Linux bash scripts, you may wonder how to run and control them on your Linux system. The only way we've run scripts is directly from the command line interface in real-time mode. This isn't the only […]

60

◂ Bash scripting the awesome guide Part2

Shell scripting the awesome guide part4 ▸