Command Line Programming

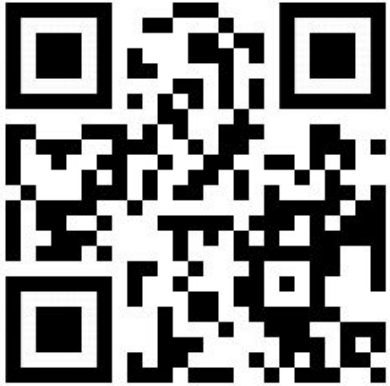
Variables, Scripts and More

Overview

- Variables, Logic
- Control flow
- Functions
- Scripting
- Handling special characters and more fancy tricks

Work along?

http://bit.ly/2GKDnxj



https://bellard.org/jslinux/vm.html?cpu=riscv64&url=https://bellard.org/jslinux/buildroot-riscv64.cfg&mem=256

```
Loading...

Welcome to JS/Linux (riscv64)

Use 'vflogin username' to connect to your account.
You can create a new account at https://vfsync.org/signup .
Use 'export_file filename' to export a file to your computer.
Imported files are written to the home directory.

[root@localhost ~]# ls -l
total 24
-rw-r--r--    1 root     root           113 Sep  9 13:26 bench.py
-rw-r--r--    1 root     root           185 Sep  9 13:26 hello.c
-rw-r--r--    1 root     root           206 Sep  9 13:26 readme.txt
-rw-r--r--    1 root     root          8256 Sep  9 13:26 rv128test.bin
[root@localhost ~]#
```

Paste Here

Echoing string to stdout: echo

https://ss64.com/bash/echo.html

- Writes each given STRING to standard output

```
echo [options]... [String]...
e.g.
    >> echo myString
    myString
```

```
mike@ME-Ubuntu:~$ echo myString
myString
```

Echoing string to stdout: `echo`

- How to write multiple lines with echo?
- Can’t just hit the Enter key, as this will execute the command!

```
mike@ME-Ubuntu:~$ echo myString
myString
mike@ME-Ubuntu:~$ on two lines
on: command not found
mike@ME-Ubuntu:~$
```

Echoing string to stdout: `echo`

- How to write multiple lines with echo?
- Can't just hit the Enter key, as this will execute the command!
- You can try **quotes**:

e.g.

```
>> echo “myString
> on two lines”
myString
on two lines
```

```
mike@ME-Ubuntu:~$ echo "myString
> on two lines"
myString
on two lines
```

Echoing string to stdout: `echo`

- What if you still can't type multiple lines?

- You can try using **escape characters** with the -e option flag
  - Special characters which are interpreted as various behaviours within the terminal.

- **\n** escape character allows us to print a **newline character**
- **\t** prints a **horizontal tab**

echo

Display message on screen, writes each given STRING to standard output, with a space between each and a newline after the last one.

```
Syntax
     echo [options]... [String]...

Options

   -n
     Do not output a trailing newline.

   -E
     Disable the interpretation of the following backslash-escaped characters.

   -e
     Enable interpretation of the following backslash-escaped
     characters in each String:

      \a    Alert (bell)
      \b    Backspace
      \c    Suppress trailing newline
      \e    Escape
      \f    Form feed
      \n    New line
      \r    Carriage return
      \t    Horizontal tab
      \v    Vertical tab
      \\    Backslash
      \NNN  The character whose ASCII code is NNN (octal);
            if NNN is not a valid octal number, it is printed literally.

      \xnnn The character whose ASCII code is the hex value nnn (1 to 3 digits)
```

Echoing string to stdout: echo

- What if you still can't type multiple lines?
- You can try using "escape characters" with the -e option flag

e.g.

```
>> echo "myString\non two lines"

myString

on two lines
```

```
mike@ME-Ubuntu:~$ echo -e "myString\non two lines"
myString
on two lines
```

Commenting

- # symbol denotes rest of line as comment

e.g.:

```
>> echo hello
hello
>> echo #hello

>>
```

```
mike@ME-Ubuntu:~$ echo hello
hello
mike@ME-Ubuntu:~$ echo #hello
```

Variables

- Declared when needed and **typeless by default**
- In the form: `myVar=value`
- Assigned from **right to left**
- Note the lack of spaces! Why?

Variables

- Issues with spaces:

e.g.:

```
>> myVar = 10
myVar: command not found
>> myVar= 10
10: command not found
```

```
mike@ME-Ubuntu:~$ myVar = 10
myVar: command not found
mike@ME-Ubuntu:~$ myVar= 10
10: command not found
```

Variables

- Issues with spaces **within values**:

e.g.:

```
>> myString=hello world
world: command not found
>> myString=”hello world”
```

```
mike@ME-Ubuntu:~$ myString=hello world

Command 'world' not found, but can be installed with:

sudo snap install world
```

Variables

- Correct without the spaces:

e.g.:

```
>> myVar=10
>> myString=”Hello World”
```

Variables

- Reference a variable with $varname

e.g.:

```
>> myVar=10
>> echo $myVar
10
>> echo myVar
myVar
```

```
mike@ME-Ubuntu:~$ myVar=10
mike@ME-Ubuntu:~$ echo $myVar
10
mike@ME-Ubuntu:~$ echo myVar
myVar
```

Variables: Numbers

- BASH can interpret variables as different types when needed.
- You can store a float (e.g. 3.14) as a string, but then use it as a float when required.
- More on this in the arithmetic section.

Variables: Strings

- BASH stores its variables as Strings
- Index using the notation:
  - ${String:position:length}

e.g.:

```
>> myVar=Hello
>> ${myVar:1:2}
el
```

```
mike@ME-Ubuntu:~$ myVar=Hello
mike@ME-Ubuntu:~$ echo ${myVar:1:2}
el
```

Arrays

- You can also declare an array as follows:
  - array=(value value value)
- Index with ${array[index]} (from 0)

e.g.:

```
>> A=(cat dog fish)
>> echo ${A[1]}
dog
```

Conditionals: if

- Form:

```
if [ condition ]
then
    statements
fi
```

Note the spaces in the condition

Conditionals: if

e.g.:

```
>> if [ 1 ]
> then
> echo “hello there”
> fi
hello there
```

```
mike@ME-Ubuntu:~$ if [ 1 ]
> then
> echo "hello there"
> fi
hello there
```

Conditionals: if elif else

- Form:

```
if [ condition ]
then
    Statements
elif [ condition ]
    Statements
else
    Statements
fi
```

Conditionals: if elif else

e.g.:

```
>> if [ 0 -eq 1 ]
> then
> echo “hello there”
> else
> echo “goodbye”
> fi
goodbye
```

```
mike@ME-Ubuntu:~$ if [ 0 -eq 1 ]
> then
> echo "hello there"
> else
> echo "goodbye"
> fi
goodbye
```

String Comparison expressions

- Equal to: ==
- Not Equal: !=
- Less than / Greater than: < / >
- Empty / Not Empty string: -z / -n

String Comparison expressions

e.g.:

```
“cat” == “cat” # True
-z “” # True
-z “cat” # False
```

```
mike@ME-Ubuntu:~$ if [ "cat" == "cat" ]
then
echo "True"
fi
True
```

String Comparison expressions

e.g.:

```
“cat” == “cat” # True
-z “” # True
-z “cat” # False
```

```
mike@ME-Ubuntu:~$ if [ -z "" ]
> then
> echo "True"
> fi
True
```

String Comparison expressions

e.g.:

```
“cat” == “cat” # True
-z “” # True
-z “cat” # False
```

```
mike@ME-Ubuntu:~$ if [ -z "cat" ]
> then
> echo "True"
> fi
```

Numerical Comparison expressions

- Equal to: -eq
- Not Equal: -ne
- Less than / Greater than: -lt / -gt
- lt or Equal / gt or Equal: -le / -ge

Numerical Comparison expressions

e.g.:

```
1 -eq 1 # True
1 -lt 1 # False
1 -le 1 # True
```

```
mike@ME-Ubuntu:~$ if [ 1 -eq 1 ]
> then
> echo "True"
> fi
True
```

Numerical Comparison expressions

e.g.:

```
1 -eq 1 # True
1 -lt 1 # False
1 -le 1 # True
```

```
mike@ME-Ubuntu:~$ if [ 1 -lt 1 ]
> then
> echo "True"
> fi
```

Loops: for

- Form:
  for iterator
  do
  statements
  done

- Iterator can be different type, depending on task

Loops: for

- Can be a list of values:
  for element **in** value value value

e.g.:

```
>> for word in hello world
> do
>  echo “The word is $word”
> done
The word is hello
The word is world
```

Loops: for

```
mike@ME-Ubuntu:~$ for word in hello world
> do
> echo "The word is $word"
> done
The word is hello
The word is world
```

Loops: for

- Can be a “traditional” style iterator:
  for ((init; condition; inc/dec)) #Note spaces

e.g.:

```
>> for (( i = 0; i < 2; i++ ))
> do
>  echo “The number is $i”
> done
The number is 0
The number is 1
```

Loops: for

```
mike@ME-Ubuntu:~$ for (( i = 0; i < 2; i++ ))
> do
> echo "number is $i"
> done
number is 0
number is 1
```

Loops: while

- Form:

```
while [ conditional expression ]
do
    statements
done
```

Loops: while

e.g.:

```
>> count=0
>> while [ $count -lt 2 ]
> do
>  echo “The number is $((count++))”
> done
The number is 0
The number is 1
```

Loops: while

```
mike@ME-Ubuntu:~$ count=0
mike@ME-Ubuntu:~$ while [ $count -lt 2 ]
> do
> echo "The number is $((count++))"
> done
The number is 0
The number is 1
```

Functions:

- Form:
  function function_handle () {
  statements
  }

Functions: Calling

e.g.:

```
>> function myFunc () {
>  echo “In the function”
> }
>> myFunc
In the function
```

```
mike@ME-Ubuntu:~$ function myFunc () {
> echo "In the function"
> }
mike@ME-Ubuntu:~$ myFunc
In the function
```

Functions: Input Arguments

- Arguments are passed in by a list of variables named as sequential numbers
- **$1, $2, ... , $n** where there are n arguments
- **$0** is reserved and denotes the function handle
- Not sure how many were passed in?
  - Use **$#**

Functions: Input Arguments

e.g.:

```
>> function myFunc () {
>  echo $#
>  echo $2
> }
>> myFunc cat dog
2
dog
```

```
mike@ME-Ubuntu:~$ myFunc cat dog
2
dog
```

Functions: Returning Values

- Values are returned with **echo** putting a value to **stdout**
- From stdout we can then **redirect** and **pipe** or assign into variables etc.
- The **return** keyword is different, this terminates the function and provides an status code

Functions: Returning values

e.g.:

```
>> function gimme_five () {
>  echo 5
> }
>> myVar=$(gimme_five)
>> echo $myVar
5
```

```
mike@ME-Ubuntu:~$ function gimme_five () { echo 5; }
mike@ME-Ubuntu:~$ myVar=$(gimme_five)
mike@ME-Ubuntu:~$ echo $myVar
5
```

Scripts: Writing

- We want to create a script of potentially many command calls and conditional flow
- Create .sh extension file containing BASH syntax
- We can use a variety of methods to write a shell script:
  - Editors like nano, vim, notepad++..
  - Commands like echo redirected to a file ...

Scripts: Writing

- Nano application will allow you to edit a file.

- Can move back and forth through the lines and edit.

- Saving to file is then CTRL+o to write out
- Enter filename and ENTER
- CTRL+x to exit nano

Scripts: Writing

- echo command and redirection can be used to write multiple lines to file (see previous slides)

e.g.:

```
>> echo “echo Hello
> echo World” >> helloworld.sh
>> sh helloworld.sh
Hello
World
```

```
mike@ME-Ubuntu:~$ echo "echo hello
> echo world" >> helloworld.sh
mike@ME-Ubuntu:~$ sh helloworld.sh
hello
world
mike@ME-Ubuntu:~$
```

Scripts: Writing

- Or if we can only write one line

e.g.:

```
>> echo -e “echo Hello\necho World” >> hello.sh
>> sh hello.sh
Hello
World
```

```
mike@ME-Ubuntu:~$ echo -e "echo Hello\necho World" >> hello.sh
mike@ME-Ubuntu:~$ sh hello.sh
Hello
World
```

Scripts: The Shebang/Hashbang

- Depending on the current shell, we may need to point to the correct interpreter
- A Shebang / Hashbang is a path to the interpreter
- Sits at the top of the script
- Can also help an IDE to highlight syntax

Scripts: The Shebang/Hashbang

- #!/bin/bash

This Shebang points us to the BASH interpreter and allows us to use BASH commands where needed.

return_example.sh

```
#!/bin/bash

my_function () {
  local func_result="some result"
  echo "$func_result"
}

func_result="$(my_function)"
echo $func_result
```

Scripts: Executing

- Running the script requires executing it via:
  - sh scriptname.sh
  - bash scriptname.sh

e.g.:

```
>> sh return_example.sh # from previous slide
some result
```