Introduction to Bash Scripting

João V. F. Lima

Universidade Federal de Santa Maria
jvlima@inf.ufsm.br
http://www.inf.ufsm.br/~jvlima

August 28, 2016

Outline

Outline

Shell Programming

A shell script is a quick-and-dirty method of prototyping a complex application. *Advanced Bash Scripting Guide.*

Shell Programming

A shell script is a quick-and-dirty method of prototyping a complex application. — *Advanced Bash Scripting Guide.*

When not to use shell scripts

1. Resource-intensive tasks
2. Heavy-duty math operations
3. Cross-platform portability
4. Complex applications
5. Mission-critical applications
6. When *security* is important
7. Project with subcomponents
8. Extensive file operations

1. Need native support for multi-dimensional arrays
2. Need data structures
3. Need to graphics or GUIs
4. Need direct access to hardware
5. Need port or socket I/O
6. Need to use libraries with legacy code
7. Proprietary applications

Hello Bash

Script hello

```
#!/bin/bash

# this is a comment
echo "Hello bash from user: $USER"
```

Hello Bash

Script hello

```
#!/bin/bash

# this is a comment
echo "Hello bash from user: $USER"
```

Turn into executable

```
chmod u+rx hello
```

Hello Bash

Script hello

```
#!/bin/bash

# this is a comment
echo "Hello bash from user: $USER"
```

Turn into executable

```
chmod u+rx hello
```

There is nothing unusual here, only a set of commands that could just have been invoked one by one from the command-line.

sha-bang

The *sha-bang* (`#!`) at the head of a script tells your system that this file is a set of commands to be fed to the command interpreter indicated.
The `#!` is a two-byte *magic number* (type `man magic`).

```
#!/bin/sh
#!/bin/bash
#!/bin/python
#!/usr/bin/perl
#!/usr/bin/tcl
#!/bin/sed -f
#!/bin/awk -f
```

Invoking the script

Script hello

```
#!/bin/bash

echo "I have $# arguments, they are: $*"
```

Invoking the script

Script hello

```
#!/bin/bash

echo "I have $# arguments, they are: $*"
```

Turn into executable

```
chmod u+rx hello
```

Invoking the script

Script hello

```
#!/bin/bash

echo "I have $# arguments, they are: $*"
```

Turn into executable

```
chmod u+rx hello
```

Executing

```
./hello here is my argument
```

Outline

Variable substitution

The name of a variable is a placeholder for its *value*. Referencing its value is called *variable substitution*.

```
a=375
hello=$a
```

No space permitted on either side of = sign.

Variable substitution

The name of a variable is a placeholder for its *value*. Referencing its value is called *variable substitution*.

```
a=375
hello=$a
```

No space permitted on either side of = sign.

```
echo hello
```

Not a variable reference, just the string "hello"

Variable substitution

The name of a variable is a placeholder for its *value*. Referencing its value is called *variable substitution*.

```
a=375
hello=$a
```

No space permitted on either side of = sign.

```
echo hello
```

Not a variable reference, just the string "hello"

```
echo $hello
echo ${hello}
```

No difference, result is 375.

Variable substitution

```
hello="A B    C"
echo $hello
echo "$hello"
```

Quoting a variable preserves whitespaces.

Variable substitution

```
hello="A B    C"
echo $hello
echo "$hello"
```

Quoting a variable preserves whitespaces.

```
echo '$hello'
```

Variable referencing disabled by single quotes.

Variable substitution

```
a=‘echo Hello!‘
echo $a
```

Assigns result of *echo* command to *a* using backquotes.

Variable substitution

```
a=‘echo Hello!‘
echo $a
```

Assigns result of *echo* command to *a* using backquotes.

```
a=‘ls -l‘
echo $a
echo
echo "$a"
```

Assings result of `ls -l` command to *a*. First *echo* removes tabs and newlines, and the second prevers whitespaces.

Variable substitution

```
a=‘echo Hello!‘
echo $a
```

Assigns result of *echo* command to *a* using backquotes.

```
a=‘ls -l‘
echo $a
echo
echo "$a"
```

Assings result of `ls -l` command to *a*. First *echo* removes tabs and newlines, and the second prevers whitespaces.

```
host=$(cat /etc/hostname)
system=$(uname -a)
```

Using the `$(...)` mechanism.

Quoting variables

Use double quotes to prevent word splitting.

```
#!/bin/bash

numbers="one two three"

for a in $numbers
do
    echo "$a"
done
```

Quoting variables

Use double quotes to prevent word splitting.

```
#!/bin/bash

numbers="one two three"

for a in $numbers
do
    echo "$a"
done
```

```
one
two
three
```

Quoting variables

Use double quotes to prevent word splitting.

```
#!/bin/bash

numbers="one two three"

for a in "$numbers"
do
    echo "$a"
done
```

Quoting variables

Use double quotes to prevent word splitting.

```
#!/bin/bash

numbers="one two three"

for a in "$numbers"
do
    echo "$a"
done
```

```
one two three
```

Escaping

```
#!/bin/bash

echo "This will print
as two lines."

echo

echo "This will print \
as one line."
```

Escaping

```
#!/bin/bash

echo "This will print
as two lines."

echo

echo "This will print \
as one line."
```

```
This will print
as two lines.

This will print as one line.
```

Escaping

```
#!/bin/bash

echo "\n\n"      # prints \n\n
echo -e "\n\n"   # prints two new lines
```

Escaping

```
#!/bin/bash

echo "\n\n"      # prints \n\n
echo -e "\n\n"   # prints two new lines
```

```
\n\n


```

Outline

Parameters

```
#!/bin/bash

echo "Number of parameters: $#"
echo "Parameter #1 is $1"
echo "Parameter #2 is $2"
echo "Parameter #10 is ${10}"
```

Parameters

```
#!/bin/bash

echo "Number of parameters: $#"
echo "Parameter #1 is $1"
echo "Parameter #2 is $2"
echo "Parameter #10 is ${10}"
```

Executing

```
./helloparams 1 2 3 4 5 6 7 8 9 10 11 12
```

Parameters

```
#!/bin/bash

echo "Number of parameters: $#"
echo "Parameter #1 is $1"
echo "Parameter #2 is $2"
echo "Parameter #10 is ${10}"
```

Executing

```
./helloparams 1 2 3 4 5 6 7 8 9 10 11 12
```

Output

```
Number of parameters: 12
Parameter #1 is 1
Parameter #2 is 2
Parameter #10 is 10
```

Exit and exit status

The *exit* command terminates a script, just as in a C program. It can also return a value.

```
#!/bin/bash

echo hello
echo $?

lalalal
echo $?

exit 113
```

The script executes an *echo*, and then a unrecognized command.

Exit and exit status

The *exit* command terminates a script, just as in a C program. It can also return a value.

```
#!/bin/bash

echo hello
echo $?

lalalal
echo $?

exit 113
```

The script executes an *echo*, and then a unrecognized command.

```
0
127
```

Outline

Test constructs

If/else

```
if [ condition-true ]
then
    command 1
    command 2
else
    command 3
    command 4
fi
```

Test constructs

If/else

```
if [ condition-true ]
then
    command 1
    command 2
else
    command 3
    command 4
fi
```

Warning

In Bash, true is 0 (since 0 means *success* by UNIX convention).

Test constructs

If/elif/else

```
if [ condition1 ]
then
    command 1
    command 2
elif [ condition2 ]
    command 3
    command 4
else
    default-command
fi
```

Test constructs

- `[` is a *command*, a synonym for `test`
- `[[` is a keyword for an *extended test command*.

```
bash$ type test
test is a shell builtin
bash$ type '['
[ is a shell builtin
bash$ type '[['
[[ is a shell keyword
bash$ type ']]'
]] is a shell keyword
bash$ type ']'
bash: type: ]: not found
```

Test constructs

```
if [ 0 ]; then
    echo "0 is true."
else
    echo "0 is false."
fi

if [ 1 ]; then
    echo "1 is true."
else
    echo "1 is false."
fi
```

Test constructs

```
if [ 0 ]; then
    echo "0 is true."
else
    echo "0 is false."
fi

if [ 1 ]; then
    echo "1 is true."
else
    echo "1 is false."
fi
```

```
0 is true.
1 is true.
```

Test constructs

```
# the -q option to grep suppresses output
if grep -q root /etc/passwd; then
    echo "Root exists."
fi
```

Test constructs

```
# the -q option to grep suppresses output
if grep -q root /etc/passwd; then
    echo "Root exists."
fi
```

Output

```
Root exists.
```

Test constructs

```
decimal=15
octal=017   # = 15 (decimal)
hex=0x0f    # = 15 (decimal)

if [ "$decimal" -eq "$octal" ]
then
  echo "$decimal equals $octal"
else
  echo "$decimal is not equal to $octal"
  # 15 is not equal to 017
fi
```

Test constructs

```
decimal=15
octal=017   # = 15 (decimal)
hex=0x0f    # = 15 (decimal)

if [ "$decimal" -eq "$octal" ]
then
  echo "$decimal equals $octal"
else
  echo "$decimal is not equal to $octal"
  # 15 is not equal to 017
fi
```

Result

Doesn't evaluate within [single brackets]!

Test constructs

```
decimal=15
octal=017   # = 15 (decimal)
hex=0x0f    # = 15 (decimal)

if [[ "$decimal" -eq "$octal" ]]
then
  echo "$decimal equals $octal"   # 15 equals 017
else
  echo "$decimal is not equal to $octal"
fi
```

Test constructs

```
decimal=15
octal=017   # = 15 (decimal)
hex=0x0f    # = 15 (decimal)

if [[ "$decimal" -eq "$octal" ]]
then
  echo "$decimal equals $octal"   # 15 equals 017
else
  echo "$decimal is not equal to $octal"
fi
```

Result

Evaluates within [[double brackets]]!

Test constructs

```
decimal=15
octal=017   # = 15 (decimal)
hex=0x0f    # = 15 (decimal)

if [[ "$decimal" -eq "$hex" ]]
then
  echo "$decimal equals $hex"   # 15 equals 0x0f
else
  echo "$decimal is not equal to $hex"
fi
```

Test constructs

```
decimal=15
octal=017   # = 15 (decimal)
hex=0x0f    # = 15 (decimal)

if [[ "$decimal" -eq "$hex" ]]
then
  echo "$decimal equals $hex"   # 15 equals 0x0f
else
  echo "$decimal is not equal to $hex"
fi
```

Result

`[[ $hexadecimal ]]` also evaluates!

File test operators

Returns true if . . .

1. `-e` file exists
2. `-f` file is a *regular* file
3. `-s` file is not zero size
4. `-d` file is a directory
5. `-r` file has read permission
6. `-w` file has write permission
7. `-x` file has execute permission

File test operators

Simple tests

```
#!/bin/bash

if [ -e hello ]; then
    echo "File hello exists."
else
    echo "hello is not here!."
fi

if [ -r /etc/shadow ]; then
    echo "WARN: I can read passwords!"
fi
```

Other comparison operators

Integer comparison:

1. `-eq` is equal to
2. `-ne` is not equal to
3. `-gt` is greater than
4. `-ge` is greater than or equal to
5. `-lt` is less than
6. `-le` is less than or equal to

Use of integer comparison

```
if [ "$a" -eq "$b" ]; then
    echo "Equals!"
fi
```

Other comparison operators

String comparison:

1. `==` is equal to
2. `!=` is not equal to
3. `<` is less than
4. `>` is greater than
5. `-z` string is *null*
6. `-n` string is not *null*

Warning

Using an unquoted string is an unsafe practice. *Always* quote a tested string.

Other comparison operators

Compound comparison:

1. `-a` logical and
2. `-o` logical or
3. `&&` logical and inside `[[`
4. `||` logical or inside `[[`

Compound comparison

```
if [ 1 -eq 2 -o 2 -ne 3 ]; then
    echo "True"
fi

if [[ "$USER" == "root" && 2 -eq 2 ]]; then
    echo "Also true"
fi
```