Shell Scripting

CYBR371: System and Network Security, (2024/T1)

Arman Khouzani, Mohammad Nekooei

Slides modified from "Masood Mansoori"

11 March, 2024

Victoria University of Wellington – School of Engineering and Computer Science

VICTORIA UNIVERSITY OF WELLINGTON TE HERENGA WAKA

Shell

DAD, WHAT ARE CLOUDS MADE OF?

LINUX SERVERS, MOSTLY.

ADVENTURES OF PINEHEAD

ART BY JAMES KENNISON

What is a Shell?

> *In computing, a shell is a computer program that exposes an operating system's services to a human user or other programs.*
>
> *Operating system shells use either a command-line interface (CLI) or graphical user interface (GUI).*
>
> *It is named a shell because it is the outermost layer around the operating system.*
>
> –*Wikipedia*

The first Unix shell was written by Ken Thompson at Bell Labs and distributed with Versions 1 to 6 of Unix, from 1971 to 1975.

UNIX Shells

- **csh**: C-Shell (C-like Syntax, Bill Joy of UC Berkeley, 1978)
- **sh**: Bourne Shell (Steven Bourne of AT&T, 1979)
- **ksh**: Korn-Shell (Bourne+some C-shell, David Korn of AT&T, 1983)
- **tcsh**: TENEX C-Shell (C-Shell with filename and command completion, 1983).
- **bash**: Bourne Again Shell (GNU Improved Bourne Shell, 1989)
  - the default interactive shell for users on most Linux systems.
- **Zsh**: Z shell (1990) an extended Bourne shell including some features of **bash**, **ksh**, and **tcsh**.
  - it is now the default shell in **Kali Linux** and **macOS**.

UNIX Shells

To check your current shell:

- **`$ echo $SHELL`** (SHELL is a pre-defined variable)

To switch shell:

- **`$ exec shellname`** (e.g., **`$ exec bash`**)
- or simply enter the **`shellname`**, (e.g. **`$ bash`**).
- **`$ exit`** returns you back to previous shell.

Shell Scripts

All you need is

#!/bin/bash

Shell Scripts

Shell Script: a text file containing a sequence of commands and constructs for a shell in a Unix-based OS to execute.

- may contain any command that can be entered on command line.

Hashpling: the first line in a shell script.

- specifies which shell to be used to interpret the shell script commands.

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

Shell Scripts: How to execute

Executing a shell script with `read` permission:

- start another shell, specify the script as an argument.

Executing a shell script with `read+execute` permission:

- execute like any executable programme.

Shell Scripting

- Start **`nano scriptfilename.sh`** with the line:

```
#!/bin/sh
```

  - All other lines starting with # are comments.
- Tell Unix that the script file is executable:

```
$ chmod u+x scriptfilename.sh
```

- Execute the shell-script:

```
$ ./scriptfilename.sh
```

Example

```
[root@server1 ~] cat myscript.sh
#!/bin/bash
# this is a comment
date
who
ls -F /
[root@server1 ~] bash myscript.sh
Fri Aug 20 11:36:18 EDT 2010
user1 tty1 2023-02-20 07:47 (:0)
root pts/0 2023-02-20 11:36 (10.0.1.2)
bin/ dev/ home/ media/ proc/ sbin/ sys/ var/
boot/ etc/ lib/ mnt/ public/ selinux/ tmp/
data/ extras/ lost+found/ opt/ root/ srv/ usr/
```

Escape Sequences

Character sequences having special meaning in the **echo** command:

- prefixed by the **** character.
- must use the **-e** option in the **echo** command.

Escape Sequences

Sequence	Character printed
\a	Alert (bell, the ASCII beep)
\b	Backspace
****	Single backslash
\c	prevents a newline following the command
\f	Formfeed
\n	Newline (not at the end of command)
\r	Return (Enter)
\t	Tab
\v	Vertical Tab
\???	The eight-bit character whose value is the octal (base-8) value ???

Common **echo** escape sequences.

Quote Characters

There are three different quote characters with different behaviour. These are:

- " (double quote, weak quote): If a string is enclosed in " " the references to variables (i.e $variable) are replaced by their values. Also back-quote and escape \characters are treated specially.
- ' (single quote, strong quote): Everything inside single quotes are taken literally, nothing is treated as special.
- ` (back quote): A string enclosed as such is treated as a command and the shell attempts to execute it. If the execution is successful the primary output from the command replaces the string.

Example

```
#!/bin/bash
echo "cal 03 2023"
echo 'cal 03 2023'
echo `cal 03 2023`
```

```
echo "Today is:" `date`
```

My first shell script

```
$ vi myscript.sh
```

```bash
#!/bin/bash
# The first example of a shell script
directory=`pwd`
echo Hello World!
echo The date today is `date`
echo The current directory is $directory
```

```
$ chmod u+x myscript.sh
```

```
$ ./myscript.sh
```

```
Hello World!
The date today is Wed Mar 20 10:42:24 EST 2024
The current directory is /cybr371/arman
```

Variables

Variables are symbolic names that represent values stored in memory. Three different types of variables in shell:

- Global Variables: Environment and configuration variables, capitalised, such as **`HOME`**, **`PATH`**, **`SHELL`**, **`USERNAME`**, **`PWD`**. When you login, such global variables are already defined, and can be referenced in your shell scripts.
- Local Variables: Within a shell script, you can create as many new variables as needed. Any variable created in this manner remains in existence only within that shell.
- Special Variables: Reversed for OS, shell programming, etc. such as **positional parameters `$0`, `$1`** ...

A few global (environment) variables

SHELL	Current shell
DISPLAY	Used by X-Windows system to identify the display
HOME	Fully qualified name of your login directory
PATH	Search path for commands
MANPATH	Search path for <man> pages
PS1, **PS2**	Primary and Secondary prompt strings
USER	Your login name
TERM	terminal type
PWD	Current working directory

Defining Local Variables

As in any other programming language, variables can be defined and used in shell scripts.

Variables in Shell Scripts are not typed.

Examples:

```
a=1234 # a is NOT an integer, a string instead
b=$a+1 # will not perform arithmetic but will be the
    string '1234+1'
b=`expr $a + 1` # will perform arithmetic so b is 1235
    now. Note the spaces before and after +
# Available operations are: + - / * ** %
b=abcde # b is string
b='abcde' # same as above but much safer.
```

*Note: There must be no spaces between the variable name, the **=** operator, and the value you want to assign to the variable.*

Referencing variables: curly bracket

Having defined a variable, its contents can be referenced by the **$** symbol.

E.g. **`${variable}`** or simply **`$variable`**.

When ambiguity exists **`$variable`** will not work. Use **`${ }`** the rigorous form to be on the safe side.

Example:

```
a='abc'
b=${a}def # this would not have worked without the { }
    as it would try to access a variable named adef
```

```
tdate=`date`
echo ”today's date is: ”+$tdate
```

Using Variables in Scripts

```bash
#!/bin/bash

lines=`cat $1 | wc --lines`
characters=`cat $1 | wc --chars`

echo "the number of lines in $1 is: $lines"
echo "the number of characters in $1 is: $characters"
```

Reading Standard Input

Shell scripts may need input from users.

- Input may be stored in a variable for later use.

read command takes user input from stdin and places it in a variable. Variable name specified by an argument to the **read** command.

```
#!/bin/bash

echo -e "please enter a filename:\c"
read filename

echo -e "please enter a destination directory:\c"
read directoryname

sudo cp $filename $directoryname
echo "file copied successfully."
```

Example

```bash
#!/bin/bash
echo -e "What is your name? -->\c"
read USERNAME
echo "Hello $USERNAME"
```

```
[root@server1 ~] chmod a+x newscript.sh
[root@server1 ~] ./newscript.sh
What is your name? --> Fred
Hello Fred
```

Positional Parameters

When a shell script is invoked with a set of command line parameters each of these parameters are copied into special variables that can be accessed:

- **`$0`**: This variable that contains the name of the script
- **`$1`**, **`$2`**, ..., **`$n1`**: 1st , 2nd, 3rd command line parameter.
- **`$#`**: Number of command line parameters
- **`$$`**: process ID of the shell

Example: **`./myscript one two buckle my shoe`**

During the execution of **`myscript`** variables **$1**, **$2**, **$3**, **$4**, and **$5** will contain **`one`**, **`two`**, **`buckle`**, **`my`**, **`shoe`**, respectively.

Variables

```
$ vi myinputs.sh
```

```
#!/bin/sh
echo Total number of inputs: $#
echo First input: $1
echo Second input: $2
```

```
$ chmod u+x myinputs.sh
$ ./myinputs.sh CYBR 371 Arman
Total number of inputs: 3
First input: CYBR
Second input: 371
```

Defining and Evaluating

- A shell variable take on the generalised form `variable=value` (except in the C shell).

```
$ set x=37; echo $x
37
$ unset x; echo $x
x: Undefined variable.
```

- You can set a pathname or a command to a variable or substitute to set the variable.

```
$ set mydir=`pwd`; echo $mydir
```

Arithmetic Operators

`expr` supports the following operators:

- arithmetic operators: **`+`**, **`-`**, **`*`**, **`/`**, **`%`**
- comparison operators: **`<`**, **`<=`**, **`==`**, **`!=`**, **`>=`**, **`>`**
- boolean/logical operators: **`&`**, **`|`**
- parentheses: **`(`**, **`)`**
- precedence is the same as C, Java

Arithmetic Operators

```
$ vi math.sh
```

```sh
#!/bin/sh

count=5
count=`expr $count + 1`
echo $count
```

```
$ chmod u+x math.sh
$ ./math.sh
6
```

Decision Constructs

Most common type of construct used in shell scripts

Alter flow of a program

- Based on whether a command completed successfully
- Based on user input

The `if` construct: syntax

```
if [this_is_true]
then
    do_these_commands
elif [this_is_true]
then
    do_these_commands
else
    do_these_commands
fi
```

- **`elif`** (else if) and **`else`** statements are optional.
- there can be as many **`elif`** statements as you like!
- **`do_these_commands`** may consist of multiple commands.
  - one per line.
  - indented for readability.
- end of statement must be **`fi`**.
- the condition part may be command or test statement.

The `if` construct: test statement

test statement: used to test a condition.

- generates a True/False value.
- inside square brackets `[ … ]`, or prefixed by the keyword `test`.
  - must have spaces after `[` and before `]`.

special comparison operators: used to combine test statements.

- `-o` (OR)
- `-a` (AND)
- `!` (NOT)

The `if` construct: Common test statements

`[ A = B ]`	**String** A is equal to string B, equivalent to `[ A == B ]`
`[ A != B ]`	**String** A is not equal to string B
`[ -n A ]`	String A is not null, equivalent to `[ A ]`
`[ -z A ]`	String A is null
`[ A -eq B ]`	A is numerically **equal** to B
`[ A -ne B ]`	A is numerically **not equal** to B
`[ A -lt B ]`	A is numerically **less than** B
`[ A -gt B ]`	A is numerically **greater than** B
`[ A -le B ]`	A is numerically **less than or equal to** B
`[ A -ge B ]`	A is numerically **greater than or equal to** B
`[ -r A ]`	A is a file/directory that exists and has **read** permission
`[ -w A ]`	A is a file/directory that exists and has **write** permission
`[ -x A ]`	A is a file/directory that exists and has **execute** permission
`[ -f A ]`	A is a **file** that exists.
`[ -d A ]`	A is a **directory** that exists.
`[ -e A ]`	A is a file/directory that **exists**.
`[ -s A ]`	A is a file/directory that exists and has non-zero **size**.

The `if` construct: Example

myscript.sh

```
#!/bin/bash

echo -e "Today's date is: \c"
date
echo -e "\nThe people logged into the system include:"
who
echo -e "\nWould you like to see the contents of /?(y/n)
    -->\c"
read ANSWER
if [ $ANSWER = "y" -o $ANSWER = "Y" ]
then
    echo -e "\nThe contents of the / directory are:"
    ls -F /
fi
```

The if construct: Example

```
if date | grep "Fri"
then
    echo "It's Friday!"
fi
```

```
if  [  "$1"  ==  "Monday"  ]
then
    echo "The typed argument is Monday."
elif[ "$1" == "Tuesday" ]
then
    echo "Typed argument is Tuesday."
else
    echo "Typed argument is neither Monday nor Tuesday."
fi
```

Note: **=** or **==** both work in test but **==** is better for readability.

The if construct: Example

```
#!/bin/sh
if [ "$#" -ne 2 ] then
    echo "$0 needs two parameters!"
    echo "You are inputting $# parameters."
else
    par1=$1
    par2=$2
fi
echo "$par1"
echo "$par2"
```

```
#!/bin/bash
inputt=$1
[ -d "$inputt" ] && echo "directory"
[ -f "$inputt" ] && echo "file"
```

The case construct

Compares the value of a variable with several different patterns of text or numbers.

```
case  $variable-name  in
    pattern1)
        command1 ...
        commandN
        ;;
    pattern2)
        command1 ...
        commandN
        ;;
    patternN)
        command1 ...
        commandN
        ;;
    *)
    Default condition to be executed
    ;;
esac
```

The case construct: Example

```
#!/bin/bash
echo -e "What would you like to see? Today's date (d), Currently
     logged in users (u), The contents of the / directory (r).
    Enter your choice(d/u/r)-->\c"
read ANSWER
if [ $ANSWER = "d" -o $ANSWER = "D" ]
then
    echo -e "Today's date is: \c"
    date
elif [ $ANSWER = "u" -o $ANSWER = "U" ]
then
    echo -e "\nThe people logged into the system are:"
    who
elif [ $ANSWER = "r" -o $ANSWER = "R" ]
then
    echo -e "\nThe contents of the / directory are:"
    ls -F /
else
    echo -e "Invalid choice! \a"
fi
```

The case construct: Example

```
#!/bin/bash
echo -e "What would you like to see? Today's date (d), Currently
     logged in users (u), The contents of the / directory (r).
    Enter your choice(d/u/r)-->\c"
read ANSWER
case $ANSWER in
    d | D )
        echo -e "\nToday's date is: \c"
        date;;
    u | U )
        echo -e "\nThe people logged in system are:"
        who;;
    r | R )
        echo -e "\nThe contents of / directory are:"
        ls -F /;;
    *)
        echo -e "Invalid choice! \a";;
esac
```

The && and || constructs

Time-saving shortcut constructs.

- When only one decision needs to be made during execution.

Syntax:

- **`command && command`**
- **`command || command`**

&&: Second command executed only if the first completes successfully.

||: Second command executed only if the first fails.

The && and || constructs: Examples

```
#!/bin/bash
if mkdir /etc/sample
then
    cp /etc/hosts /etc/sample
    echo "The hosts file was successfully copied to /etc/sample"
else
    echo "The /etc/sample directory could not be created."
fi
```

```
#!/bin/bash
mkdir /etc/sample && cp /etc/hosts /etc/sample
```

```
#!/bin/bash
mkdir /etc/sample || echo "Could not create /etc/sample"
cp /etc/hosts /etc/sample || echo "Could not copy /etc/hosts"
```

Loop Constructs: The `for` Constructs

Can be used to process a list of objects.

```
for var_name in string1 string2 ...
do
these_commands
done
```

During execution sets **`var_name`** to a string name, and executes the commands between do and done for that string. Repeats for all the strings.

The for Constructs: Examples

```
#!/bin/bash
for NAME in bob sue mary jane frank lisa jason
do
    mail -s "Your new project schedule" $NAME < newschedule
    echo "$NAME was emailed successfully"
done
```

```
[root@server1 ~] chmod a+x emailusers.sh
[root@server1 ~] ./emailusers.sh
bob was emailed successfully
sue was emailed successfully
mary was emailed successfully
jane was emailed successfully
frank was emailed successfully
lisa was emailed successfully
jason was emailed successfully
```

The for Constructs: Examples

```
echo -e "What directory has the files that you would like to
    rename?-->\c"
read DIR
for NAME in $DIR/*
do
    mv $NAME $NAME.txt
done
```

```
for i in $(seq 1 10);
do
    echo " $i times 5 is  $((  i * 5 )) "
done
```

```
sum=0
for i in $(seq 1 $1);
do
    sum=`expr $sum + $i`
    # or equivalently ((sum=sum+i)) or sum=$(expr $sum + $i)
done
echo "The sum of numbers from 1 through $1 is ${sum}!"
```

The `while` construct: syntax

```
while this_returns_true
do
    these_commands
done
```

Example:

```
#!/bin/sh
i=1
sum=0
while [ $i -le $1 ]
do
    sum=`expr $sum + $i`
    i=`expr $i + 1`
done
echo "The sum of numbers from 1 through $1 is ${sum}!"
```

The `while` construct: Examples

```
#!/usr/bin/bash
file=temp.txt
while read -r line;
do
    echo $line
done < ”$file”
```

```
#! /bin/bash
file=temp.txt
echo ”Enter the content into the file $file”
while read line
do
    echo $line >> $file
done
```

Assigning outputs to variables

```
#!/bin/bash
echo -e "Enter a folder's name:\c"
read foldername

commandoutput=$(ls -lh $foldername)

echo $commandoutput
echo -e "\n\n\n"
echo "$commandoutput"
```

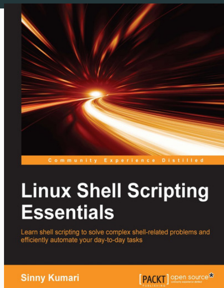
- You can use pipe `|` to redirect stdout from one command to the stdin of another.
- You can create your own variables and `export` them so that they are available to programmes started by the shell.

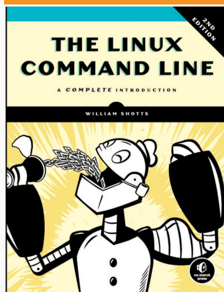
Extra Reading

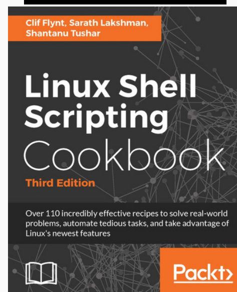
Great references (e-book available through VUW library):

- CompTIA Linux+ Certification All-in-One Exam Guide, Jordan, Ted.; Strohmayer, Sandor.; 2023
- https://tldp.org/LDP/Bash-Beginners-Guide/html/
- https://www.freecodecamp.org/news/bash-scripting-tutorial-linux-shell-script-and-command-line-for-beginners/
- https://www.geeksforgeeks.org/bash-scripting-introduction-to-bash-and-bash-scripting/
- https://linuxconfig.org/bash-scripting-tutorial

Extra References (all available through VUW library)







Next: Overview of TCP/IP