OPS 102
OPERATING SYSTEMS for PROGRAMMERS

Bash Scripting

Chris Tyler

What is a Script?

A shell script is a simple computer **program** which is **interpreted** by an operating system **shell**.

Scripts are used to automate procedures that could be manually performed from the command line.

How will scripts save hours of my life?

If you're developing a computer program which requires 25 steps to build and test, and you're going to iterate through the build and test process 100 times, you can:

- Perform 2500 steps manually – a lot of work, and error-prone; or
- Write a script containing those 25 steps and then execute it each time you want to build and test your program (100 times)
    - You can even set things up so the script is executed automatically when certain conditions are met – such as saving a change to your source code!

Basic Requirements for Shell Scripts

1. Create a text file containing shell commands.
2. Tell the operating system *which* shell to use to execute the commands.
3. Ensure that the script file has the appropriate permissions.

1. Create a file containing shell commands

- Use any text editor
- Use the same commands that you would type at the command-line
- Save the file

2. Tell the operating system which shell to use

- Add a "shebang" line to the start of the file
  - First character is a ***sh**arp*: #
  - Second character is a ***bang***: !
  - The rest of the line is the absolute pathname of the shell / interpreter

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

- The name "shebang" comes from "sharp" and "bang"
- The #! characters form a "magic number" which the **kernel** notices
- The # character causes the **shell** to interpret the line as a comment, and ignore it

3. Set the correct permissions on the file

- The **kernel** needs execute permission to analyze the shebang line and execute the correct shell/interpreter.

- The **shell** needs read permission to read the contents of the file.

- Set these permissions with the **chmod** command for any user(s) that should be able to execute the file:

```
chmod u+rx scriptname
```

Demo: Basic scripts

- Let's write some simple scripts using commands that we know

- Scripts act like any other executable file, so we can type the script name as a command. However, unless the script is in a directory that is normally searched by the operating system, it won't be able to find it. We'll talk about how to adjust this later, but for now, we can use a pathname that includes a slash to execute a script we've written:

```
$ ./scriptname
```

Setting Variables

- To set a variable:

```
VARIABLE=VALUE
```

- Variable names start with a letter and can contain letters, numbers, and underscores.
- Case matters! **A** and **a** are different variables.
- Don't put spaces on either side of the equal sign
- Note: unlike other languages such as C, you don't need to declare the variable or specify the type of data (e.g., integer, string) which it will hold.

Accessing Variables

- To use a variable value in a command, precede it with a dollar sign:

  ```
  $VARIABLE
  ```

- You can use a variable anywhere in a command:
  - Arguments
  - Command name

- The value of the variable is substituted into the command.

Variables: A Simple Example

```
$ WHAT=World
$ echo Hello $WHAT
Hello World
```

Word Splitting

- The shell uses certain separator characters to split commands into words. For example, spaces are used to break this line into a three parts – a command and two arguments:

```
$ ls -l filename
```

- But this means that arguments, such as filenames, that contain spaces will be interpreted as multiple arguments:

```
$ ls -l file one
```

Preventing Word Splitting with Quoting

- Quoting prevents words from being split.
- It's needed whenever we're dealing with text that contains separators (such as space or tabs).

Types of Quotes

- Single or double quote characters may be used to quote strings:

  ```
  A="Hello World"
  B='Hello World'
  ```

- When double quotes are used, variables will be expanded inside the string.
- When single quotes are used, variables will not be expanded inside the string.

Quoting: Examples

```
$ WHAT="World"

$ echo "Hello $WHAT"
Hello World

$ echo 'Hello $WHAT'
Hello $WHAT
```

Quoting: Examples

```
$ WHAT="World"
$ MSG="Hello $WHAT"
$ echo $MSG
Hello World
$ echo "$MSG"
Hello World
$ echo '$MSG'
Hello $MSG
```

Quoting: One argument vs. Multiple

- When string contains a separator such as a space, and it is unquoted, the shell will interpret it as multiple words. When used as an argument, this will be interpreted as multiple arguments:

```
$ touch "new file"
$ ls -l new file
ls: cannot access 'new': No such file or directory
ls: cannot access 'file': No such file or directory
$ ls -l "new file"
-rw-r--r--. 1 chris chris 0 Jun 18 22:47 'new file'
```

Quoting: One argument vs. Multiple

- This also applies when a variable is used as an argument:

```
$ touch "new file"
$ F="new file"
$ ls -l $F
ls: cannot access 'new': No such file or directory
ls: cannot access 'file': No such file or directory
$ ls -l "$F"
-rw-r--r--. 1 chris chris 0 Jun 18 22:49 'new file'
```

Quoting: One argument vs. Multiple

- **For this reason, you should always double quote variables that may contain a space in their value when using them as command arguments.**

- This is especially true for filenames – you never know when a user is going to put a space in a filename! Many scripts work fine with opaque filenames (those containing no whitespace) but fail with non-opaque filenames.

Quoting: Backslashes

- A backslash character outside of quotes or inside double quotes instructs the shell to ignore any special meaning that the following character may have. Examples:

```
$ touch "new file"
$ ls -l new\ file
-rw-r--r--. 1 chris chris 0 Jun 18 22:49 'new file'

$ echo "This string contains a \"quoted\" string"
This string contains a "quoted" string

$ A=Testing
$ echo "   \$A"
   $A
```

Variables vs Environment Variables

- By default, a variable is local to the shell in which it is running.
- You can *export* variables to make them *environment variables.* That means that they are passed to programs that are executed by the shell.
- Environment variables are commonly used to pass configuration information to programs and to configure how programs operate.
- Environment variables are used by all processes, not just the shell!

Common Environment Variables

Environment Variable	Purpose
PS1	Normal (first-level) shell prompt
EDITOR	Name of the default text editor (typically /usr/bin/nano)
PATH	A colon-separated list of directories that will be searched when looking for a command
LANG	The default language – used to select message translations as well as number, currency, date, and calendar formats.
HOME	The user's home directory – used for relative-to-home pathnames.
RANDOM	A random integer (0-32767)

Viewing and Creating Environment Variables

- See all current environment variables and their values:

```
$ set | less
```

- Create an environment variable:

```
$ X=500
$ export X
        ...or...
$ export Y=123
```

Example: PS1 environment variable

- Change the shell prompt:

```
$ PS1="Enter a command: "
Enter a command: date
Sun 18 Jun 2023 11:17:14 PM EDT
Enter a command: PS1="[\u@\h \W]\$ "
[chris@toronto ~]$ PS1="$ "
$
```

Example: PATH environment variable

```
$ cat showdate
#!/usr/bin/bash
date
$ ./showdate
Sun 18 Jun 2023 11:20:18 PM EDT
$ showdate
bash: showdate: command not found...

$ echo $PATH
/home/chris/.local/bin:/home/chris/bin:/usr/lib64/ccache:/usr/loc
al/bin:/usr/local/sbin:/usr/bin:/usr/sbin:/usr/libexec/sdcc:/usr/
libexec/sdcc:/usr/libexec/sdcc
$ PATH="$PATH:."
$ showdate
Sun 18 Jun 2023 11:20:45 PM EDT
```

Example: LANG environment variable

```
$ echo $LANG
en_CA.UTF-8
$ date
Sun 18 Jun 2034 11:32:27 PM EDT
$ foobarbaz
bash: foobarbaz: command not found...

$ LANG=fr_CA.UTF-8
$ date
dim 18 jun 2034 23:32:47 EDT
$ foobarbaz
bash: foobarbaz: commande inconnue...
```

Example: HOME environment variable

```
$ echo $HOME
/home/chris
$ echo ~
/home/chris

$ HOME=/
$ echo ~
/
$ ls ~
afs  bin  boot  dev  etc  home  lib  lib64  lost+found
media  mnt  opt  proc  root  run  sbin  srv  sys  tmp
usr  var
```

Reading a Variable Value from Stdin: read

- You can read values from standard input (stdin) and assign them to a variable with the read command:

  ```
  read variable
  ```

- Example:

  ```
  $ read COURSE
  Seneca OPS102
  $ echo $COURSE
  Seneca OPS102
  ```

Using read with a Prompt String

- You can display a message to the user when reading from stdin by using the –p (prompt) option to read:

```
$ read -p "Please enter a course code: " C
Please enter a course code: OPS102
$ echo "The selected course is $C"
The selected course is OPS102
```

- Of course, you can also use a separate echo command instead!

Demo: Variables in a Script

```
#!/usr/bin/bash
read -p "Please enter your name: " NAME
echo "Pleased to meet you, $NAME"
read -p "Please enter a filename: " FILE
echo "Saving your name into the file..."
echo "NAME=$NAME" >>$FILE
echo "Done."
```

Command Capture

- You can capture the standard output (stdout) of a command as a string using the notation $( )

```
$ echo "The current date and time is: $(date)"
The current date and time is: Mon 19 Jun 2034
12:02:11 AM EDT

$ FILES="$(ls|wc -l)"
$ echo "There are $FILES files in the current
directory $(pwd)"
There are 2938 files in the current directory /bin
```

Public Service Announcement:
Command Capture: Avoid Backticks

- You may see old scripts that use backticks (reverse single quotes) for command capture:

  ```
  $ A=`ls`
  ```

  Don't do this! This is an archaic syntax which is deprecated. Some fonts make it hard to distinguish between backticks and single quotes, and nesting backticks is difficult.

Arithmetic!

- Bash can do *integer* arithmetic
- To evaluate a arithmetic expression and return a value, use $(( ))
- To evaluate a arithmetic expression without returning a value, use (( ))
- Dollar-sign prefixes for variables are not required inside $(( )) or (( ))

```
$ A=100
$ B=12
```

```
$ echo $((A*B))
1200
$ echo $((B++))
12
$ echo $B
13
```

```
$ ((A++))
$ echo $A
101

$ ((C=A*B*2))
$ echo "The answer is $C"
The answer is 2626
```

Exit Status Codes

- When a program runs, it exits with a numeric value. This goes by any of several names:
  - Exit code
  - Status code
  - Exit status code
  - Error code
- Usually, an exit status code of zero (0) means that no errors were encountered, and a non-zero code means that something went wrong.
  - It may be easiest to think of this as the error code, with 0 meaning no errors.
- Alternately, program authors can use this value as they see fit, so the status code may indicate something else, like the number of data items processed.

Exit Status Codes: $?

- The special variable $? can be used to find out the exit status of the last command executed:

```
$ ls /foo/bar/baz
ls: cannot access '/foo/bar/baz': No such file or directory
$ echo $?
2

$ ls /usr/bin/bash
/usr/bin/bash
$ echo $?
0
```

Exit Status Codes: Why do we care?

- Exit status codes are the key to conditional logic (if...) and looping (for/while/until/...) in bash scripting.

Conditional logic: if / then / else / elif / fi

- The if command takes two or more lists of commands, and uses the result of one list to control the execution of the other:

```
if list1                # If the exit status is success...
then
      list2             # … then run this
fi
```

Conditional logic: if / then / else / elif / fi

```
if grep -q "OPS102" testfile
then
        echo "The course is mentioned in the 
file"    fi
```

Conditional logic: if / then / else / elif / fi

- There are else and elif (else-if) keywords too:

```
if    list1        # If the exit status is success...
then
      list2        # … then run this

elif  list3        # Else if this exits with success...
then
      list4        # … then do this

else
      list5        # Otherwise do this
fi
```

Conditional logic: if / then / else / elif / fi

```
if grep -q "OPS102" testfile
then
        echo "The course is mentioned in the file"

else
        echo "The file doesn't mention OPS102"
fi
```

Conditional logic: if / then / else / elif / fi

```
if grep -q "OPS102" testfile
then
        echo "The course is mentioned in the file"

elif grep -q "ULI101" testfile
then
        echo "The old ULI101 course is in the file"


else
        echo "The file doesn't mention OPS102 or ULI101"
fi
```

The test command

- The test command (which is a shell builtin) can perform a variety of comparisons and test, and returns success (0 exit status) if the test succeeds. For example:

```
test "$NAME" == "Chris"
```

- This is used with if/then/fi:

```
if test "$NAME" == "Chris"
then
     SUPERPOWERS="Yes"
fi
```

The test command and [[ ]]

- The double square bracket is an aliases for the test command. When you use this alias, an extra argument with the closing square brackets is expected:

```
if [[ "$NAME" == "Chris" ]]
then
     SUPERPOWERS="Yes"
fi
```

- Single square bracket is supported in all Posix shells; double square bracket provides an improved version of the test command in Bash and other common shells.

/usr/bin/test and [ ]

- Besides the bash builtin test (or [[ ]]), there are two other versions of test available:

`/usr/bin/test` *or* `/usr/bin/[`	external version of test
shell builtin: [	Posix-compliant builtin version of test

- These versions of test have a slightly different syntax. For simplicity, we won't be using them in this course. Refer to the corresponding manpages (test(1) or bash(1)) for additional information if you're interested.

Tests 1: Filesystem entries (Files/Dirs/Links)

- This first group of tests deals with filesystem entries, such as files and directories. Each test expects one argument, a filename:

-f *filename*	*filename* is a regular file
-d *filename*	*filename* is a directory
-L *filename*	*filename* is a symbolic link

Tests 2: File Permissions

- These tests accept one argument, a filename:

-r *filename*	*filename* is readable
-w *filename*	*filename* is writable
-x *filename*	*filename* is executable

Tests 3: Strings

- These tests accept two string arguments, which are compared:

string1 == *string2*	strings match
string1 != *string2*	strings don't match
string1 > *string2*	*string1* greater than (sorts before) *string2*
string1 < *string2*	*string1* greater than (sorts before) *string2*

Tests 4: Integers

- These tests accept two integer arguments, which are compared:

integer1 -eq *integer2*	integers are equal
integer1 -ne *integer2*	integers are not equal
integer1 -gt *integer2*	integer1 is greater than integer2
integer1 -ge *integer2*	integers1 is greater than or equal to integer 2
integer1 -lt *integer2*	integers is less than integer2
integer1 -le *integer2*	integer1 is less than or equal to integer2

All the Rest of the Tests

- These are just the most commonly-used tests.
- See the bash(1) manpage for other tests that might be useful.

Negating and Combining Tests

- You can negate (invert) a test with the ! operator:

```
[[ ! -f "$F" ]]   # check that $F isn't a regular file
```

- You can combine tests using the && and || operators:

```
    [[ $A –eq $B   &&   $C –eq $D ]]  # && means "and"
    [[ $X –eq $Y   ||   $X –eq $Z ]]  # || means "or"
```

- This should, of course, look familiar! (Why? Consider who wrote Unix and who created C...)

Using Tests

- Remember to quote arguments which include whitespace separators.
- Be careful with the < and > comparison operators – if you have a syntax error, you may accidentally redirect data (which in the case of > may truncate a file!).

Examples of using test: Strings

```
#!/usr/bin/bash
architecture="$(uname -m)" # uname gets system information

if [[ "$architecture" == "x86_64" ]]
then
   echo "Your computer architecture is Intel/AMD x86_64."
elif [[ "$architecture" == "aarch64" ]]
then
   echo "Your computer uses the 64-bit Arm architecture."
else
  echo "Your computer uses an unrecognized architecture."
fi
```

Examples of using test: Integer Numbers

```
#!/usr/bin/bash
read -p "Enter the customer's date of birth: " B

# Calculate the time in seconds that the customer turns/tuned 19
D="$(date -d "$B + 19 years" +%s)"

# See if the current time in seconds is greater than that
NOW="$(date +%s)"

# Tell the user if the customer is old enough to be served alcohol
if [[ "$D" -lt "$NOW" ]]
then
    echo "The customer is of legal drinking age in Ontario."
else
    echo "The customer is too young to legally drink in Ontario."
fi
```

Examples of using test: Integer Numbers

```
#!/usr/bin/bash

COINFLIP=$((RANDOM % 2))
if [[ "$COINFLIP" == 0 ]]
then
    echo "Heads! ☺"
else
    echo "Tails ☹"
fi
```

Examples of using test: File and Permissions

```
#!/usr/bin/bash
read -p "Enter the file to be deleted: " F
if [ ! -f "$F" ]
then
    echo "The filename '$F' does not refer to a regular file - skipping."
elif [ ! -w "$F" ]
then
    echo "The file '$F' is not writeable (by you) - skipping."
else
    read -p "Delete the regular file '$F'? (Y/N): " YESNO
    if [[ "$YESNO" == "Y" || "$YESNO" == "y" || "$YESNO" == "Yes"
             || "$YESNO" == "yes" || "$YESNO" == "YES" ]]
    then
        echo "Deleting the file '$F'..."
        rm "$F"    # We should add some code to check if the rm succeeds or fails
        echo "...done."
    else
        echo "Skipping the file '$F' as requested."
    fi
fi
```

Script Parameters

- It's useful to be able to call a script with positional parameters (arguments).
- These can be accessed within a script as $0, $1, $2, $3, and so forth.
- $0 is the name of the script itself.
- $# is the number of positional parameters.
- The shift command gets rid of the first parameter and shifts every parameter to a lower number.

Script Parameters

```
$ cat params
#!/usr/bin/bash
echo "Number of parameters:  $#"
echo "Parameter 1:           $1"
echo "Parameter 2:           $2"
echo "Parameter 3:           $3"
echo "Parameter 4:           $4"

$ ./params red green blue
Number of parameters:  3
Parameter 1:           red
Parameter 2:           green
Parameter 3:           blue
Parameter 4:
```

Script Parameters

```
$ cat params2
#!/usr/bin/bash
echo "Number of parameters:  $#"
shift
echo "Parameter 1:           $1"
echo "Parameter 2:           $2"
echo "Parameter 3:           $3"
echo "Parameter 4:           $4"

$ ./params2 red green blue
Number of parameters:  3
Parameter 1:           green
Parameter 2:           blue
Parameter 3:
Parameter 4:
```

Script Parameters

- `$*` and `$@` both return ALL of the parameters.

- When quoting:
  - `"$*"` returns all the parameters as a single string – not usually useful.
  - `"$@"` returns each parameter as a separate string – usually what you want.

Script Parameters

```
$ cat params3
#!/usr/bin/bash
ls -l "$*"
echo ---
ls -l "$@"

$ touch a b c
$ ./params3 a b c
ls: cannot access 'a b c': No such file or directory
---
-rwxr-xr-x. 1 chris chris 463 Jun 21 11:55 a
-rwxr-xr-x. 1 chris chris 121 Jun 21 11:55 b
-rwxr-xr-x. 1 chris chris 532 Jun 21 11:55 c
```

Script Parameters

- Let's look at a bash code to check that the user has provided 2 arguments.

- In this code, we're also including the name of the script in the error message, sending the error message to stderr, and exiting with a unique error code.

Script Parameters

```
$ cat paramcheck
#!/usr/bin/bash
if [[ "$#" -ne 2 ]]
then
    echo "$(basename $0): Error: 2 arguments expected" >&2
    exit 1
fi

$ ./paramcheck foo bar
$ ./paramcheck foo
paramcheck: Error: 2 arguments expected
```

Looping

- There are four types of loops available in bash:
  - `for` *`variable`* `in` *`values`* `; do … ; done`
  - `for ((` *`setup`*`;` *`control`*`;` *`iteration`* `)) ; do … ; done`
  - `while` *`list`* `; do … ; done`
  - `until` *`list`*`; do … ; done`

Looping: for *variable* in *values*

- This type of loop accepts a list of values. The first value is assigned to the variable and the loop is executed, and then the process is repeated with each remaining value.

- The *values* could be:
  - A list of constants: `for CITY in Toronto Vaughan Oshawa`
  - Parameters: `for X in "$@"`
  - A file globbing pattern: `for FILE in *.jpg`
  - … Or anything else that consists of one or more values as separate words

Looping: for *variable* in *values*

```
$ cat tidyup
#!/usr/bin/bash
for FILE in *.backup *.bck
do
    if [[ -r "$FILE"  ]]
    then
        read -p "Delete file '$FILE' (Y/N)? " YESNO
        if [[ "$YESNO" == "y" || "$YESNO" == "Y" ]]
        then
            echo "Deleting file '$FILE'"
            rm "$FILE"
        else
            echo "'$FILE' was not deleted."
        fi
    fi
done

$ touch oldfile.backup source.bck
$ ./tidyup
Delete file 'oldfile.backup' (Y/N)? N
'oldfile.backup' was not deleted.
Delete file 'source.bck' (Y/N)? Y
Deleting file 'source.bck'
```

Looping: for ((*setup*; *control*; *iteration*))

- This type of loop works pretty much the same as a C-style for loop. Example:

```
for (( i=0; i<10; i++ ))
do
      echo "$i"
done
```

- Remember the double-parenthesis!

Looping: while *list*; do *. . .* ; done

- This type of loop executes as long as *list* returns success (exit == 0).
- Example:

```
while [[ "$(who | wc –l)" -gt 1 ]]
do
  echo "There are other users logged in:"
  who
  sleep 10
done
```

Looping: until *list*; do *. . .* ; done

- This type of loop executes as long as *list* doesn't return success (exit != 0).
- Example:

```
until [[ "$(date +%u)" == "6" ]]
do
  echo "Waiting until Saturday..."
  sleep $((24 * 60 * 60))
done
```

Using Scripts

- Scripts may be used to customize your environment on a Linux system.
- There are two scripts in your home directory that are executed automatically by bash. They are both named starting with a period (dot), which causes them to be "hidden" (not normally displayed by the ls command):
  - ~/.bash_profile – this script is executed once per login. This is a good place to put commands that set up your work environment, including envars.
  - ~/.bashrc– this script is executed whenever bash starts up (which may be several times per login session). This is the right place to put things such as command aliases (which are not inherited by child processes).

Warning!

- A broken ~/.bash_profile or ~/.bashrc script may PREVENT you from successfully logging in to your account!
- To protect yourself:
  1. Test ~/.bash_profile and ~/.bashrc scripts while logged in to your account by explicitly specifying their names.
  2. If that is successful, <u>stay logged in to your account</u> while initiating a new login to test the scripts in the login context. For example, if you are logging in remotely, *stay logged in* on one ssh session while initiating a new, separate ssh login session to test the scripts.