ITE315 Module 2 Part A - Bash As A Programming Language

Athens State University

Contents

1 Scripts?

Task Automation By Shell Programming

> The shell is actually a programming language: it has variables, loops, decision-making, and so on. - Kernighan & Pike, *The UNIX Programming Environment*

- **Shell script**: a computer program designed to be run in a UNIX shell, a command-line interpreter.
  - This definition has been extended to mean an interpreted program that is run in an automated mode by an operating system.
- One can view a shell script as being a text file that contains one or more commands that you might type on the command-line

Getting Ready To Write Scripts

- Let's review: home folder, shell variable, environment variable, working directory
- Need to create a place to put your shell scripts
  - Dr. Lewis's habits: put scripts in a `bin` folder in your home folder
- Naming of scripts:
  - Don't name your scripts the same as a Bash built-in or external command (unless you want people to use your version)
  - A debatable convention to use the `.sh` extension for your shell scripts

We'll need somewhere to store our scripts. It's common to put files that will be "executed" on a machine into a folder named `bin`. That's an abbreviation for "binary" and makes the use of those files pretty obvious.

Be careful about what names you use for your scripts. Try the following in your shell:

```
type test
type -a test
```

One of the uses of `test` builtin is to check the type of a file. Note that since `test` is a built-in command in the shell, we can can't use it as a name for one of our scripts. That's why it's pretty common for people to use `.sh` as the extension for shell script files.

1.1 Startup Configuration Files

Configuring A New Shell

- At launch, Bash will run one or more shell scripts to initialize itself
- We can add or modify these files to customize the behavior of the shell
- Exactly which shell script depends on how the shell is executed
  - **Login shell**: The shell launched when you log-on in a non-GUI environment or launched when you start the Terminal app
  - **Interactive shell**: A shell launched from within another shell
  - **Non-interactive shell**: A shell used to execute a shell script

Shell Configuration Files

`/etc/profile`	This is a system level configuration file for all users. It's executed fires

`.bash_profile`
`.bash_login`
`.profile`
	These files are executed in your home folder and are executed in this order if they exist

`.bash_logout`	Executed at logout, useful for clean-up

This is a simplified version of what gets called when you start an interactive shell. You can adjust environment variables in these files to control where Bash looks to find your scripts.

2 Bash As Programming Language

Our First Shell Script

```
#!/bin/bash
# Cleanup
# Run as root, of course.
# Insert code here to print error message and exit if not root.

LOG_DIR=/var/log
# Variables are better than hard-coded values.
cd $LOG_DIR
```

```
10 cat /dev/null > messages
   cat /dev/null > wtmp
12
   echo "Logs cleaned up."
14
   exit
```

This script is designed for a system administrator to use clear the contents of two system log files in the `/var/log` directory. Of course, this would need to be executed by a user with system administrative permissions.

Note that Bash interprets everything following the `#` symbol as being a comment.

Sha-Bang!

- Bash does a little trickery with the first bytes of a shell script.
  - If the first line starts with the `sha-bang` character pair (`#!`), then Bash will execute this file using the program whose immediately follows the sha-bang
- This feature will be very useful when we start working with PERL and Python

Note also that we need to be clear about the difference between **/bin/sh** and `/bin/bash`. The Bourne shell in most UNIX-like operate systems is named `/bin/sh`. On systems that use the Bourne-again shell, that is an alias to **/bin/bash** but Bash will, for compatibility reasons, only use the features that were included in the older shell.

A Side Note: `/dev/null`

- There a number of "special" files in the Linux operating system
- The `/dev/null` file is a special file that is always going to be empty
- So, using `cat` against that file will concatenate the empty file to standard output
  - So, it's a way in the shell script to empty the files

There are other useful special files. For example, `/dev/random` is a special file that is linked to the CPU's hardware random number generator. One way to get a true random number is to read an integer or floating-point value from that file.

3 Parameters, Positional and Otherwise

Parameters

- Shell variables are special case of **parameters**
  - **Positional Parameters**: Values passed into a script from the command-line, think `argc` and `argv`
  - **Special Parameters**: Values that the shell uses to store information about its internal state
  - **Variables**: Values identified by a name

Positional Parameters

- Consider what happens when we run a command like grep
- Each part of the command line is identified by a positional parameter

```
grep -e '[A-Za-z0-9][A-Za-z0-9]*' *.cpp
echo $0 $1 $2
```

The function shift N moves the positional parameters by N positions, if you ran shift (the default value of N is 1), then $0 would be discarded, $1 would become $0, $2 would become $1, and so on: they would all be shifted by 1 position. There are some very clever and simple uses of shift to iterate through a list of paramters of unknown length.

Special Parameters

$*	All positional parameters
$@	All positional parameters
$#	The number of positional parameters
$$	The process id of foreground process
$?	Exit value returned by the last command

This is just a representative sample of the special parameters. Others will be introduced throughout our discussion of shell scripting.

4 Quoting and Expansion

The Curious Case Of The Quotation Mark

- Modern operating systems allow you to include spaces, tabs, and even more strange characters in file names
- For those cases, we must wrap command-line arguments in quotes to prevent the shell from treating the spaces as separators
  - One can use either a single quote or double quote
  - But be careful as the shell processes those differently
- One other option is to quote a single character using the backslash character

In all cases, the quoting characters are not passed into the program (that's called *quote removal*). So, commands have no way of knowing exactly how they are invoked.

The Curious Case Of The Quotation Mark

- OK... why the distinction between single quotes and double quotes?
  - The shell processes strings differently depending upon what you select
- The shell will *expand* variables in double quoted strings
  - So, `echo "The folder is $MYDIR"` will be expanded to include the current value of `$MYDIR`
  - Single quoted strings will undergo filename expansion and word splitting

Consider the following script:

```
for='a b*'     # stores "a b*" in the variable "foo"
echo $foo
```

The shell will process the string and expand it based on the contents of the folder. That is not really what you want in this case.

A "Dr. Lewis Good Coding Practice"

Get into the habit of using double quotes by default when you write shell scripts.

- It can help you to avoid some really confusing bugs in scripts
- This is really important when you want to use variable expansion in strings

5 Control Flow In Scripts

Conditional Expressions

```
#!/bin/bash
if test -e source.txt; then
   cp source.txt destination.tx
fi
```

Note the basic structure of an `if` statement in Bash: it is book-ended with the `if` and `fi` constructs:

```
if command ; then
   statements
fi
```

Note the condition is a Bash command! Thus, we have to include the semi-colon to indicate the end of the command.

This Is Going To Irritate You Immensely

Most of you are C++ programmers: so to you, `false` is `0` and `true` is any other value. But note that the `if` in Bash is using commands, so a successful exit status of 0 is viewed as being `true` and a failed non-zero exit status is viewed as being `false`

Arrrrrrrrrrrrrrrgh! It gets worse because if you do use aritmetic expressions in conditions, then Bash falls back to the truth values we know and love.

This can be quite confusing and requires one to stop and think!

5.1 The `test` Command

Thus, the `test` command

> The test utility evaluates the expression and, if it evaluates to true, returns a zero (true) exit status; otherwise it returns 1 (false). If there is no expression, test also returns 1 (false). - `Linux General Commands Manual`

```
test <expression>
[ <expression> ]
```

The `test` command gives the shell a way to test Boolean expressions. Expressions are built using combinations of command-line options. The '`[ ]`' operator is a shorthand way of writing `test`. But *NOTE* that you ***MUST*** remember the semicolon after the command even if you use the short cut.

Useful `test` Operators: Files

-e <FILE>	Does the file exist?
-f <FILE>	Is this a regular file?
-d <FILE>	Is this a directory (folder)?
-r <FILE>	Can I read from this file?
-w <FILE>	Can I write to this file?
-x <FILE>	Is this file executable?

Useful `test` Operators: Files

<FILE1> -nt <FILE2>	Is <FILE1> newer than <FILE2>?
<FILE1> -ot <FILE2>	Is this a regular file?
-L <FILE>	Is this an alias for another file?
-s <FILE>	Is this file empty?

Useful `test` Operators: Strings

-z <STRING>	Is this string the empty string?
<STR1> = <STR2>	Are these strings equal?
<STR1> != <STR2)	Are these strings not equal?

Useful `test` Operators: Arithmetic

<INT1> -eq <INT2>	Are these equal?
<INT1> -ne <INT2>	Are these not equal
<INT1> -lt <INT2>	Is <INT1> less than INT2?
<INT1> -gt <INT2>	Is <INT1> greater than INT2?
<INT1> -le <INT2>	Is <INT2> less than or equal to <INT2>
<INT1> -ge <INT2>	Is <INT1> greater than or equal to <INT2>

Note that the shell has different operators for testing strings versus integers. The common operators ("`=<>`)" are use for testing strings! Note as well that math in the shell is integer math!

Building More Complex Expressions

- The `test` command defines `-a` and `-o` operators for the Boolean AND and OR operations in expression
  - Avoid using these operators, as the semantics get really weird
- The preferred method is use the shell list control operators

```
if [ -n "$var"] && [ -e "$var$" ] ; then
    echo "\$var is not null and a file named $var exists!"
fi
```

Note that the POSIX standard for UNIX-like operating systems is very wishy-washy in its definition of the behavior of the `test` command. This means different implementations of Bash may behavior differently in testing.

This is why if we are looking to implement the behavior where you need to check two things in a Boolean expression, it's better to use the shell's list control operators and test separately.

Suppose you want to check to see if two conditions are true at the same time (i.e., AND):

1. Is a string empty (in other word it is "null")?
2. If a command generates output

Here's how you would do attempt to solve the problem using `test`'s Boolean operators followed by how you would solve the problem using the shell's list control operators:

```
if [ -z "false" -a -z "$(echo I am executed >&2)" ] ; then ...

if [ -z "false" ] && [-z -z "$(echo I am not executed >&2)" ] ; then..
```

In the first case, the arguments are expanded by Bash **before** the echo command is executed!

Side note: note the use of the parenthesis at the call to the `echo` command. This takes the output from that command and stores into a temporary variable that we use as if it was a shell variable.

Note how the list control operators work:

- The && operator implies that the second command is executed if, and only if, the first command returns an exit status of zero
- The || operator implies that the second command is executed if, and only if, the first command returns a non-zero exit status.

The return status for both operators is the exit status of the last command executed in the list.

5.2 Examples and Pitfalls

Testing That A File Or Folder Exists

```
if [ -d $d ] ; then
    echo "Guess what? $d is a folder!"
else
    echo "Boo... $d isn't a folder!"
fi
```

Note that you can suffer a common newbie mistake with this type of test:

```
if [-d $d ]
then
    echo "Yep, $d is a direcctory."
fi
```

Don't forget that `test` or `[` are Bash commands! Forgetting this fact will result in you seeing odd errors.

Working With Strings

```
#!/bin/bash
t1="foo"
t2="bar"
if [ "$T1" = "$T2"] ; then
  echo "yep, they the same"
else
  echo "nope, they differ"
fi
```

Math Tests

```
if [ $1 -gt 100 ]
then
   echo That\'s might just be a big number.
fi
```

Maths Is Hard (Sometimes In The Shell)

```
(( a = 23 )) # Setting a variable to a numeric value.
(( a++ ))    # Our dear friend, the post-increment
# Just like C++, post- and pre- increment have side-effects
n=1 ; let --n && echo "True" || echo "False" #False
n=1 ; let n-- && echo "True" || echo "False" #True
```

- There's a bunch of shell oddness in this code!

We really need to do some "traditional" math sort of things. This is where the `let` and `(( ... ))` constructs come into play.

Take a good look at the last two lines. Pay close attention to the use of the list control operators, remembering that truth values are the reverse in the shell than what we're used to seeing in other programming languages.

Nesting The Ifs

```
if [ $1 -gt 100 ]
then
   echo That\'s might just be a big number.

   if (( $1 % 2 == 0 )); then
      echo And it\s also an even number
   fi
fi
```

IF-ELSEIF-ELSE Constructs

```
#!/bin/bash
# elif statements
if [ $1 -ge 18 ]; then
   echo You may go to the party.
elif [ $2 == 'yes' ] ; then
   echo You may go to the party but be back before midnight.
else
   echo You may not go to the party.
fi
```

6 The case Statement

The case Statement

```
#!/bin/bash
# case example
case $1 in
 start)
  echo starting
  ;;
 stop)
  echo stoping
  ;;
 restart)
  echo restarting
  ;;
 *)
  echo don\'t know
  ;;
esac
```

There are times where you want to take a different path in your control flow based on how a value has been set rather than upon a Boolean decision. In languages like C++, we have `switch` statement, but in Bash we have `case` statement. This construct begins with `case` and ends with `esac` ("case" in reverse). The `;;` construct is the equivalent of the C++ `break` statement.

Note that shell decides which path to take in a `case` statement by matching the control string against a set of regular expressions. So, the `*)` expression is the equivalent of the `default` case in a `switch` statement.