Shell Scripting

2014/10/09

Victor Eijkhout

What is a shell?

- Command interpreter: listens for your commands, executes, shows output
- Toolbox: chain together commands with pipes
- Programming language:
  - Variables
  - Control structures
  - (not so much data structures)
  - Interactive use or through programs

The “bash” shell

- sh: one of the original shells (the other is “csh”), written by Stephen Bourne in 1977
- bash: Bourne Again Shell, reimplementation and extension
- We prefer bash: better for shell programming

Variables

- Setting a variable:

```
title="My System Information"
```

- Use:

```
echo $title
```

Exercise 1

```
$ a = "the title"
$ a= "the title"
$ a="the title"
$ echo a
$ echo $a
```

Explore the various ways to go wrong when you declare and use variables

Variables

- Naming rules:
  - Must start with a letter
  - Must not contain embedded spaces
    - Use underscores
  - Must not be a punctuation mark
  - Must not be an existing bash *reserved word*
    - To see a list of reserved words, use the `help` command

Variables

- Some variables are predefined
  `$HOME,$PATH`
- Typical use:
  `my_project_dir=$HOME/projects/main`
- (It would be nice if you didn't have to do this every time you logged in)

Exercise 2

```
$ thevar="the value”
$ bash
bash-3.2$ echo $thevar
# what is the output?
bash-3.2$ exit
$ export thevar="the value"
$ bash
bash-3.2$ echo $thevar
# what is the output
```

Two ways of setting a variable

Exercise 3

```
$ a="foo.bar"
$ echo ${a%%.bar}
$ echo ${a##foo.}
```

Funky stuff to do with variables

```
$ a=$( whoami )
$ echo $a
```

```
$ a=1
$ b=$(( a+2 ))
$ echo $b
```

Exercise 4: Startup files

- Create a file ".bashrc" in your home directory, and add some lines

```
export my_project_dir=$HOME/projects/main
```

- Also create ".profile" with this content:

```
if [ -f ~/.bashrc ]; then
    source ~/.bashrc
fi
```

- Log out, log back in (or just open a new terminal window)

Shell scripts

- In exercise 20 you made a shell script
- Call a script:
  explicit path: `~/bin/work.cmd`
  in your current directory: `./work.cmd`
- Better: `echo $PATH` and put a line in your .profile:

```
export PATH=$PATH:$HOME/bin
```

Shell script programming

- In a shell script, some variables are predefined:
  $# : the number of arguments
  $1, $2, … : the arguments

Exercise 5

```
$ cat > countum
#!/bin/bash

echo "There are … arguments"
```

Finish the script and execute it

Flow Control

- Bash provides several commands to control the flow of execution
  - if
  - exit
  - for
  - while
  - until
  - case
  - break
  - continue

if

```
# First form
if condition ; then
    commands
fi
```

```
# Second form
if condition ; then
    commands
else
    commands
fi
```

```
# Third form
if condition ; then
    commands
elif condition ; then
    commands
fi
```

if: usual form

- Square brackets are shorthand for “test” (important: spaces inside the brackets!)

```
if [ sometest]; then
```

- Numerical tests:

```
if [ $a –gt 2 ]
```

- File tests:

```
if [ -f .profile ]
```

- Man page: **“man test”**

Exercise 6

```
$ cat > namum
#!/bin/bash

echo "There are … arguments"

if [ ... ] ; then
 echo "The first argument is …"
fi
```

Write a script that echos the first argument, if there is at least one argument

Exercise 7

```
$ cat > testum
#!/bin/bash

if [ … ] ; then
 echo "You need to provide an arg"
fi

if [ ... ] ; then
 echo "Your file exists!"
else
 echo "Your file does not exist!"
fi
```

Write a script that tests if a file exists

Loops - for

```
for variable in words; do
    commands
done
```

Exercise 8

Let's do it all on one line:
substitute the name of an existing file in this command

```
for f in * ; do if [ $f = ″name" ] ; then echo "found name" ; fi ; done
```

What happens if you forget the "do" or the "then" or "fi"?

Reading Input

```
#!/bin/bash

echo -n "Enter some text > "
read text
echo "You entered: $text"
```

More control structures: case

```
#!/bin/bash

echo -n "Enter a number between 1 and 3 inclusive >
"
read character
case $character in
    1 ) echo "You entered one."
        ;;
    2 ) echo "You entered two."
        ;;
    3 ) echo "You entered three."
        ;;
    * ) echo "You did not enter a number between 1
and 3."
esac
```

Loops - while

```
#!/bin/bash

number=0
while [ "$number" -lt 10 ]; do
    echo "Number = $number"
    number=$((number + 1))
done
```

Nested commands

- Use backticks “`” to execute a command inside another

```
for i in `ls`; do
    echo $i
done
for i in `ls | grep foo | tr a-z A-Z`; do
    echo $i
done
```

Here Scripts

- The greatest programmers are also the laziest
  - Really they write programs to save them work
- When clever programmers write programs, they try to save themselves typing

Here Scripts

- Here scripts are a form of I/O redirection
- A here script is constructed like this:

```
command << token
content to be used as command's standard input
token
```

- *token* can be any string
  Often: EOF is short for End Of File

Here Scripts

- Changing “<<“ to “<<-” causes bash to ignore the leading tabs (but not spaces).
- Using “ ‘token’ “ prevents expansion

Exercise 9

```
$ cat > makum
#!/bin/bash

cat > report.sh <<EOF
This program says $1
EOF
chmod +x report.sh
```

Write a script that generates a (fairly simple) new script

Exercise 10

```
$ touch prog.c prog.o none.o
$ for ofile in *.o
# report if you find a .o file
# without corresponding .c file
```

This is difficult!

See exercise 3: you can manipulate the $ofile variable…

Shell Functions

- Aliases are good for simple commands
- Use *shell functions* if you want something more complex
- Add the following function to your .bash_profile

```
function today {
    echo “Today’s date is:”
    date +”%A, %B %-d, %Y”
}
```

Shell Functions

- Function is a shell builtin too!
  - As with alias, you can enter this directly on the command prompt

```
$ function day {
> echo “Today’s date is:”
> date +”%A, %B %-d, %Y”
> }
```

type

- There are many types of commands
  - alias, shell function, executable file
- To determine what a command is, use the `type` command

```
$ type command
```

type

```
$ type l
l is aliased to `ls -l'

$ type cd
cd is a shell builtin

$ type function
function is a shell keyword
```