Southern Connecticut State University

SCSU

CSC424 System Administration

Instructor: Dr. Hao Wu

Week 7 Bash Scripting - 2

Outline

- Filename Matching
- Regular Expression
- Linux Text Processing Tools
- **Variables**
- **Arithmetic**
- Control Flow
- Loops
- Function

Your first bash script

- Create you first script named "hello"
- Type the following inside it:

```
#!/bin/bash
echo "Hello World"
```

- The first line tells Linux to use the bash interpreter to run this script.
- In order to run the script, we need to make to script executable:

```
chmod +x hello
```

- To run the script:

```
./hello
```

Variables

- A variable is temporary store for a piece of information. There are two actions we may perform for variables:
  - Assign a value to a variable.
    - var=value (no space before or after!)
  - Read the value from a variable.
    - $var or ${var}
- Uninitialized variables have no value
- Variables are untyped, interpreted based on context

Single and Double Quote

- When assigning character data containing spaces or special characters, the data must be enclosed in either single or double quotes.

- Using double quotes to show a string of characters will allow any variables in the quotes to be resolved

```
[hao@node1 ~]$ var="test string"
[hao@node1 ~]$ newvar="Value of var is $var"
[hao@node1 ~]$ echo $newvar
Value of var is test string
```

Single and Double Quote

- Using single quotes to show a string of characters will not allow variable resolution

```
[hao@node1 ~]$ var='test string'
[hao@node1 ~]$ newvar='Value of var is $var'
[hao@node1 ~]$ echo $newvar
Value of var is $var
```

Environmental Variables

- There are two types of variables:
  - Local variables
  - Environmental variables

- Environmental variables are set by the system and can usually be found by using the env command.
- Shell variables are generally not visible to programs
- All environment variables are also shell variables, but not vice versa
- Environmental variables hold special values. For instance:

```
[hao@node1 ~]$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/root/bin
[hao@node1 ~]$ echo $SHELL
/bin/bash
```

The export command

- The export command puts a variable into the environment so it will be accessible to child processes. For instance:

```
[hao@node1 ~]$ bash                    #Run a child shell
[hao@node1 ~]$ echo $var
                                       #No value in var
[hao@node1 ~]$ exit                    #Return to parent
exit
[hao@node1 ~]$ echo $var
test string
[hao@node1 ~]$ export var
[hao@node1 ~]$ bash
[hao@node1 ~]$ echo $var
test string                            #It's there
```

The export command

- If the child modifies var, it will not modify the parent's original value.

```
[hao@node1 ~]$ export var
[hao@node1 ~]$ bash
[hao@node1 ~]$ echo $var
test string                          #It's there
[hao@node1 ~]$ var="change me"
[hao@node1 ~]$ echo $var
change me
[hao@node1 ~]$ exit
exit
[hao@node1 ~]$ echo $var
test string
```

Environment Variables

- LOGNAME: contains the user name
- HOSTNAME: contains the computer name.
- HOME: home directory
- PATH: list of directories to search
- TERM: type of terminal (vt100, ...)
- TZ: timezone (e.g., US/Eastern)
- RANDOM: random number generator
- PS1: sequence of characters shown before the prompt

Environment Variables

- PS1: sequence of characters shown before the prompt:
  - \t hour
  - \d date
  - \w current directory
  - \W last part of the current directory
  - \u user name
  - \$ prompt character
- Example:

```
[hao@~]$PS1="[This is \u]\$"
[This is hao]$
```

Shell variables

Parameter	Meaning
$0	Name of the current shell script
$1-$9	Positional parameters 1 through 9
$#	The number of positional parameters
$*	All positional parameters, “$*” is one string
$@	All positional parameters, “$@” is a set of strings
$?	Return status of most recently executed command
$$	Process id of current process

Shell variables: Command Line Arguments

- The 'set' command can be used to assign values to positional parameters

```
[hao@node1 ~]$ set arg1 arg2 arg3 arg4
[hao@node1 ~]$ echo $0
bash
[hao@node1 ~]$ echo $*
arg1 arg2 arg3 arg4
[hao@node1 ~]$ echo $#
4
[hao@node1 ~]$ echo $1
arg1
[hao@node1 ~]$ echo $3 $2
arg3 arg2
[hao@node1 ~]$ echo $$
2725
```

Array variables

- An array is a variable containing multiple values.
- Any variable may be used as an array.
- There is no maximum limit to the size of an array, nor any requirement that member variables be indexed or assigned contiguously.
- Arrays are zero-based: the first element is indexed with the number 0.
- Array variable assignment:
  - array=(var1 var2 var3 …)

Array variables

- Accessing the value in array variables
  - In order to refer to the content of an item in an array, use curly braces

```
[hao@node1 ~]$ a=(one two three four) #create an array
[hao@node1 ~]$ echo ${a[*]}           #get all values
one two three four
[hao@node1 ~]$ echo ${a[2]}           #get the third value
three
[hao@node1 ~]$ a[2]=2                 #change the third value
[hao@node1 ~]$ echo ${a[*]}
one two 2 four
[hao@node1 ~]$ a[4]=five              #add a new value to array
[hao@node1 ~]$ echo ${a[*]}
one two 2 four five
[hao@node1 ~]$ echo ${a[@]}
one two 2 four five
```

Array variables

- Deleting the value in array variables

```
[hao@node1 ~]$ unset a[1]
[hao@node1 ~]$ echo ${a[*]}
one 2 four five
[hao@node1 ~]$ unset a
[hao@node1 ~]$ echo ${a[*]}
```

Manipulating Strings

- Bash supports a number of string manipulation operations.

  ${#string} gives the string length

  ${string:position} extracts sub-string from $string at $position

  ${string:position:length} extracts $length characters of sub-string from $string at $position

```
[hao@node1 ~]$ st=0123456789
[hao@node1 ~]$ echo ${#st}
10
[hao@node1 ~]$ echo ${st:6}
6789
[hao@node1 ~]$ echo ${st:6:2}
67
```

User Input

- shell allows to prompt for user input
- Syntax:
  - **`read varname [more vars]`**
- or
  - **`read -p "prompt" varname [more vars]`**
- words entered by user are assigned to
- **`varname`** and "**`more vars`**"
- last variable gets rest of input line

User Input Example

```
#! /bin/sh
read -p "enter your name: " first last

echo "First name: $first"
echo "Last name: $last"
```

Command Substitution

- The backquote “`” is different from the single quote “´”. It is used for command substitution: `command`

```
[hao@node1 ~]$ LIST=`ls -a`
[hao@node1 ~]$ echo $LIST
. .. .bash_history .bash_logout .bash_profile .bashrc .ca
che .config hello .viminfo
```

- We can perform the command substitution by $(command)

```
[hao@node1 ~]$ LIST=$(ls -a)
[hao@node1 ~]$ echo $LIST
. .. .bash_history .bash_logout .bash_profile .bashrc .ca
che .config hello .viminfo
```

Arithmetic Evaluation

- The let statement can be used to do mathematical functions:

```
[hao@node1 ~]$ let X=5+4*8
[hao@node1 ~]$ echo $X
37
[hao@node1 ~]$ let Y=X*2-21
[hao@node1 ~]$ echo $Y
53
```

- An arithmetic expression can be evaluated by $[expression] or $((expression))

```
[hao@node1 ~]$ echo "$((121+111))"
232
[hao@node1 ~]$ x=$[12*12]
[hao@node1 ~]$ echo $[$x*2]
288
```

Arithmetic Evaluation

- Available operators: +, -, /, *, %
- Example

```
[hao@node1 ~]$ cat arthmetic.sh
#!/bin/bash
echo -n “Enter the first number: ”; read x
echo -n “Enter the second number: ”; read y
add=$(($x + $y))
sub=$(($x - $y))
mul=$(($x * $y))
div=$(($x / $y))
mod=$(($x % $y))
# print out the answers:
echo “Sum: $add”
echo “Difference: $sub”
echo “Product: $mul”
echo “Quotient: $div”
echo “Remainder: $mod”
```