Experiments11
Shell Programming

Open Source SW Development
CSE22300

Bash Shell Scripting

Bash Shell

- **Linux has a variety of different shells:**
  - **Bourne shell (sh), C shell (csh), Korn shell (ksh), TC shell (tcsh), Bourne Again shell (bash).**

- **Certainly the most popular shell is “bash”**
  Bash is an sh-compatible shell that incorporates useful features from the Korn shell (ksh) and C shell (csh).

- **It is intended to conform to the IEEE POSIX P1003.2/ISO 9945.2 Shell and Tools standard.**

- **It offers functional improvements over sh for both programming and interactive use.**

Programming or Scripting

- **Bash**
  - **Excellent command line shell**
  - **Scripting language**
  - **Allows us to use the shell's abilities and to automate a lot of tasks**

- **Programming Language**
  - **Powerful and a lot faster**
  - **Compiled into an executable.**
  - **Poor portability**

- **Scripting**
  - **Not compiled into an executable**
  - **Interpreter reads the instructions**
  - **Slower than compiled programs.**
  - **Good portability**

Simple Example

- Run vi program like below
  - vi hello.sh

- Type the following inside it

```
#!/bin/bash
echo “Hello World”
```

- Make it executable and run

```
$ chmod 700 hello.sh
$ ./hello.sh
```

Single Quote and Double Quote

- Double Quote
  - String of characters will allow any variables in the quotes to be resolved
- Example

```
$ var=“test string”
$ newvar=“Value of var is $var”
$ echo $newvar
Value of var is test string
```

- Single Quote
  - String of characters will not allow variable resolution

- Example

```
$ var=’test string’
$ newvar=’Value of var is $var’
$ echo $newvar
Value of var is $var
```

Export

- **Export Command**
  - **Puts a variable into the environment so it will be accessible to child processes**
- **Example**

```
$ x=hello
$ bash                 # Run a child shell.
$ echo $x              # Nothing in x.
$ exit                 # Return to parent.
$ export x
$ bash
$ echo $x
hello                  # It's there
```

- **Starting vi**
  - **Type vi <filename> at the shell prompt**

Environmental Variables

- **Environmental Variables**
  - **Set by the system**
  - **Hold special values**
  - **Example**

```
$ echo $SHELL
/bin/bash
$ echo $PATH
/usr/X11R6/bin:/usr/local/bin:/bin:/usr/bin
```

- **Defined**
  - **/etc/profile, /etc/profile.d/ and ~/.bash_profile.**
- **Start-up**
  - **/etc/bashrc and ~/.bashrc**
- **Log-out**
  - **~/.bash_logout**

Read

- The read command allows you to prompt for input and store it in a variable.

- Example

```
#!/bin/bash
echo -n “Enter name of file to delete: ”
read file
echo “Type 'y' to remove it, 'n' to change your mind ... ”
rm -i $file
echo "That was YOUR decision!”
```

Command Substitution

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

- **Example**

```
$ LIST=`ls`
$ echo $LIST
```

- **$(command)**
  - **We can perform the command substitution**
- **Example**

```
$ LIST=$(ls)
$ echo $LIST
```

Arithmetic Evaluation

- **The let Statement**
  - **Used to do mathematical functions:**

- **Example**

```
$ let X=10+2*7
$ echo $X
24
$ let Y=X+2*4
$ echo $Y
32
```

- **$[expression] or $((expression))**

- **Example**

```
$ echo “$((123+20))”
143
$ VALORE=$[123+20]
$ echo “$[123*$VALORE]”
17589
```

Arithmetic Evaluation

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

- **Example**
  - **vi arithmetic.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”
```

Conditional Statements

- Conditionals
  - Let us decide whether to perform an action or not
  - this decision is taken by evaluating an expression.

- Grammar

```
if [ expression ];
then
        statements
elif [ expression ];
then
        statements
else
        statements
fi
```

- Put spaces after [and before], and around the operators and operands.

Expressions

- **Expression**
  - **String comparison**
  - **Numeric comparison**
  - **File operators**
  - **Logical operators**

- **String Comparisons:**
  - **= compare if two strings are equal**
  - **!= compare if two strings are not equal**
  - **-n evaluate if string length is greater than zero**
  - **-z evaluate if string length is equal to zero**

- **Examples:**
  - **[s1 = s2]   (true if s1 same as s2, else false)**
  - **[s1 != s2]   (true if s1 not same as s2, else false)**
  - **[s1]   (true if s1 is not empty, else false)**
  - **[-n s1]   (true if s1 has a length greater then 0, else false)**
  - **[-z s2]   (true if s2 has a length of 0, otherwise false)**

Expressions

- **Number Comparisons:**
  - **-eq compare if two numbers are equal**
  - **-ge compare if one number is greater than or equal to a number**
  - **-le compare if one number is less than or equal to a number**
  - **-ne compare if two numbers are not equal**
  - **-gt compare if one number is greater than another number**
  - **-lt compare if one number is less than another number**

- **Examples:**
  - **[n1 -eq n2] (true if n1 same as n2, else false)**
  - **[n1 -ge n2] (true if n1greater then or equal to n2, else false)**
  - **[n1 -le n2] (true if n1 less then or equal to n2, else false)**
  - **[n1 -ne n2] (true if n1 is not same as n2, else false)**
  - **[n1 -gt n2] (true if n1 greater then n2, else false)**
  - **[n1 -lt n2] (true if n1 less then n2, else false)**

Expressions

- **Files operators:**
  - **-d** check if path given is a directory
  - **-f** check if path given is a file
  - **-e** check if file name exists
  - **-r** check if read permission is set for file or directory
  - **-s** check if a file has a length greater than 0
  - **-w** check if write permission is set for a file or directory
  - **-x** check if execute permission is set for a file or directory

- **Examples:**
  - **[-d fname]** (true if fname is a directory, otherwise false)
  - **[-f fname]** (true if fname is a file, otherwise false)
  - **[-e fname]** (true if fname exists, otherwise false)
  - **[-s fname]** (true if fname length is greater then 0, else false)
  - **[-r fname]** (true if fname has the read permission, else false)
  - **[-w fname]** (true if fname has the write permission, else false)
  - **[-x fname]** (true if fname has the execute permission, else false)

Expressions

- **Logical operators**
  - **! negate (NOT) a logical expression**
  - **-a logically AND two logical expressions**
  - **-o logically OR two logical expressions**

- **Example:**

```
#!/bin/bash
echo -n “Enter a number 1 < x < 10:”
read num
if [ “$num” -gt 1 –a “$num” -lt 10 ];
then
        echo “$num*$num=$(($num*$num))”
else
        echo “Wrong insertion !”
fi
```

Expressions

- **Logical operators:**
  - **&&		logically AND two logical expressions**
  - **||		logically OR two logical expressions**

- **Example:**

```
#!/bin/bash
echo -n "Enter a number 1 < x < 10: „
read num
if [ “$number” -gt 1 ] && [ “$number” -lt 10 ];
then
	echo “$num*$num=$(($num*$num))”
Else
	echo “Wrong insertion !”
fi
```

Shell Parameters

- **Positional parameters**
  - **The shell's argument**
  - **Positional parameter "N" may be referenced as "$N"**

- **Special parameters**
  - **${#} is the number of parameters**
  - **${0} returns the name of the shell script**
  - **${*} gives a single word containing all the parameters passed to the script**
  - **$@ gives an array of words containing all the parameters**

- **Example**

```
#!/bin/bash
echo "$#; $0; $1; $2; $*; $@"
```

Case Statement

- Often used in place of an if statement if there are a large number of conditions.

- Value used can be an expression
- Each set of statements must be ended by a pair of semicolons;
- *) is used to accept any value not matched with list of values

```
case $var in
val1)
        statements;;
val2)
        statements;;
*)
        statements;;
esac
```

Case Statement

- Example

```
#!/bin/bash
echo -n “Enter a number 1 < x < 10: ”
read x
case $x in
        1) echo “Value of x is 1.”;;
        2) echo “Value of x is 2.”;;
        3) echo “Value of x is 3.”;;
        4) echo “Value of x is 4.”;;
        5) echo “Value of x is 5.”;;
        6) echo “Value of x is 6.”;;
        7) echo “Value of x is 7.”;;
        8) echo “Value of x is 8.”;;
        9) echo “Value of x is 9.”;;
        0 | 10) echo “wrong number.”;;
        *) echo “Unrecognized value.”;;
        esac
```

Iteration Statements

- The for structure
  - Used when you are looping through a range of variables.

```
for var in list
do
        statements
done
```

- Statements are executed with var set to each value in the list.

- Example

```
#!/bin/bash
let sum=0
for num in 1 2 3 4 5
do
        let “sum = $sum + $num”
done
echo $sum
```

Using Arrays with Loops

- The simplest way to create one is using one of the two subscripts:

```
pet[0]=dog
pet[1]=cat
pet[2]=fish
pet=(dog cat fish)
```

- To extract a value, type ${arrayname[i]}

```
$ echo ${pet[0]}
```

- To extract all the elements, use an asterisk as:

```
echo ${arrayname[*]}
```

- We can combine arrays with loops using a for loop:

```
for x in ${arrayname[*]}
do
        ...
done
```

C-like for loop

- **An alternative form of the for structure is**

```
for (( EXPR1 ; EXPR2 ; EXPR3 ))
do
        statements
done
```

- **Example**

```
#!/bin/bash
echo –n “Enter a number: ”; read x
let sum=0
for (( i=1 ; $i<$x ; i=$i+1 )) ;
do
    let “sum = $sum + $i”
done
echo “the sum of the first $x numbers is: $sum”
```

Debugging

- **-x**
  - **displays each line of the script with variable substitution and before execution**
- **-v**
  - **displays each line of the script as typed before execution**

- **Usage:**
  - **#!/bin/bash –v or #!/bin/bash –x or #!/bin/bash –xv**

- **Example**

```
#!/bin/bash –x
echo –n “Enter a number: ”; read x
let sum=0
for (( i=1 ; $i<$x ; i=$i+1 )) ;
do
        let “sum = $sum + $i”
Done
echo “the sum of the first $x numbers is: $sum”
```

While Statements

- **The while structure is a looping structure**
  - **Used to execute a set of commands while a specified condition is true.**

```
while expression
do
   statements
done
```

- **Example**

```
#!/bin/bash
echo –n “Enter a number: ”; read x
let sum=0; let i=1
while [ $i –le $x ];
do
        let “sum = $sum + $i”
         i=$i+1
Done
echo “the sum of the first $x numbers is: $sum”
```

Continue Statements

- The continue command
  - Causes a jump to the next iteration of the loop
  - skipping all the remaining commands in that particular loop cycle.

- Example

```
#!/bin/bash
LIMIT=19
echo
echo “Printing Numbers 1 through 20 (but not 3 and 11)”
a=0
while [ $a -le “$LIMIT” ];
do
        a=$(($a+1))
        if [ “$a” -eq 3 ] || [ “$a” -eq 11 ]
        then
                continue
        fi
        echo -n “$a ”
done
```

Break Statements

- The break command terminates the loop (breaks out of it).

- Example

```
#!/bin/bash
LIMIT=19
echo
echo “Printing Numbers 1 through 20, but something happens after 2 … ”
a=0
while [ $a -le “$LIMIT” ]
do
        a=$(($a+1))
        if [ “$a” -gt 2 ]
        then
                        break
        fi
        echo -n “$a ”
done
echo; echo; echo
exit 0
```

Until Statements

- The until structure is very similar to the while structure
- The until structure loops until the condition is true

```
until [expression]
do
        statements
done
```

- Example

```
#!/bin/bash
echo “Enter a number: ”; read x
echo ; echo Count Down
until [ “$x” -le 0 ];
do
        echo $x
        x=$(($x –1)
        sleep 1
done
echo ; echo GO !
```

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**

- **Example**

```
$ st=0123456789
$ echo ${#st}
10
$ echo ${st:6}
6789
$ echo ${st:6:2}
67
```

Parameter Substitution

- **Manipulating and/or expanding variables**
- **${parameter-default}, if parameter not set, use default.**

```
$ echo ${username-`whoami`}
alice
$ username=bob
$ echo ${username-`whoami`}
bob
```

- **${parameter=default}, if parameter not set, set it to default.**

```
$ unset username
$ echo ${username=`whoami`}
$ echo $username
alice
```

- **${parameter+value}, if parameter set, use value, else use null string.**

```
$ echo ${username+bob}
bob
```

Parameter Substitution

- **${parameter?msg}, if parameter set, use it, else print msg**

```
$ value=${total?’total is not set’}
total: total is not set

$ total=10
$ value=${total?’total is not set’}
$ echo $value
10
```

Functions

- Functions
  - make scripts easier to maintain
  - breaks up the program into smaller pieces

- Example

```
#!/bin/bash
hello()
{
echo “You are in function hello()”
}
echo “Calling function hello()…”
hello
echo “You are now out of function hello()”
```

Functions

- **Example**

```
#!/bin/bash
function check() {
    if [ -e "/home/$1" ]
    then
            return 0
    else
            return 1
    fi
}
echo “Enter the name of the file: ” ; read x
if check $x
then
    echo “$x exists !”
else
    echo “$x does not exists !”
fi.
```

Assignment 01

- **Recycle bin**
  - **Usage : rv my_files or my_directories**
  - **Move files and directoies into /HOME_DIRECTORY/trash**
  - **Support multiple files and directories**

- **Get Process Pid**
  - **Usage : psget process_name**
  - **This command returns process id**
  - **Use grep**

Assignment 02

- Find Duplicate
  - Usage : finddup directory
  - Find duplicate file names in directory (recursive)
  - Print duplicate file path

- Find Duplicate
  - Usage : finddupc directory
  - Find duplicate file contents in directory (not recursive)
  - Use md5sum for check contents of file
  - Print duplicate file path