Unit 3
Shell Programming

Ref: Chapter 8 – Introduction to Unix and Shell Programming by Venkatesh Murthy

Shell & Shell Variables

- Acts as a command interpreter and programming language
- Programs written in shell is called shell programs or shell scripts
- Variables can be used in shell as how in other programming languages
- Rules for constructing variables are similar to naming a file.
- Variable names are constructed using only alphanumeric characters and the underscore (_) character, with the first one being a letter. The names are case sensitive.
- Types of shell variables - System variables, Local variables or User defined variables, Read-only variables, Special variables or positional parameters

System Variables

- System variables are set either during the boot sequence or immediately after logging in
- Working environment of User, depends entirely upon the values of these variables
- These variables are also known as *environment* variables. These are similar to *global variables*.
- These variables are written using uppercase letters only
- Eg: PATH, HOME, IFS, MAIL, SHELL, TERM

Local Variables

- Local variables in the shell are variables that are local to a particular user’s shell
- As these variables are defined and used by specific users, they are also called user-defined variables.
- These variables exist only for a short time during the execution of a shell script.
- they are local to the user’s shell environment and are not available for the other scripts or processes.
- A shell variable is *defined* using an equal to (=) operator *without any spaces on either sides of it*

- variable=value
- All shell variables are of string type
- Default all shell variables are initialized as null strings
- It is not necessary to type declare or initialize shell variables
- Evaluating a Shell variable
  - Shell variables are *evaluated* by prefixing the variable name with a $.
  - When the shell reads a command line, all words that are preceded by a $ are identified and evaluated as variables unless otherwise the $ is despecialized

```
$x=37
$echo $x
37
$echo $xyz             # Since xyz was not initialized,
                       # null string will be the output.
$x=Venkatesh ; y=murthy
$z=$x$y
$echo $z
Venkateshmurthy
$
```

```
$count=`wc file1 file2`
$echo $count
256   4186   23456   file1
176   3426   22135   file2
432   7612   45591   total
$
```

Read only variables

- Variables, the values of which can only be read but not be manipulated, are called read-only variables.
- x=10
- readonly x

Export command

- Export is to export variables across programs and shells
- a=10
- export a
- TERM=vt100
- export TERM
- export a=10
- Export will not work on a new spanned terminal
- When used in shell scripts, it has to be executed using source or .
- Because export will not be done on a new spanned terminal or process.

.profile file

- Every user has a .profile of his or her own
- This file is a shell script that will be present in the home directory of the user
- It is an AUTOEXEC.BAT file of Unix.
- it gets executed as soon as the user logs in
- The system administrator provides each user with a profile that will be just sufficient to have a minimum working environment
- However, it can be edited according to user convenience
- To View: $cat .profile

First Shell Script

- All the necessary commands that are required are put in a separate file in the required sequence, and the file is executed.
- Such a set of commands that are taken together as a single unit within a file and executed at a stretch is called a shell program or a shell script.
- A shell script also includes commands for selective execution (control commands), commands for I/O operation like read and echo commands, commands for repeated execution (loop–control structures)

- A shell script is named just like all other files. A shell script name uses .sh extension. (It is optional)
- A shell program runs in the interpretive mode, that is, one statement is executed at a time.
- shell programs run slower
- Intermediate stage of an application package development
- After testing, they are converted to some higher-level language code (like C)

- clear
- echo "This is my first shell script."
- echo "Today’s date is `date  +%d%m%y` "
- echo “Now the time is `date +%T`"
- echo "GOOD LUCK"
- Any script file will have only read and write permissions upon their creation
- Assign execute permission to the file

- Execution – 3 ways
  - Straightforward method of executing a shell script is by using the shell command sh.
    - Eg: sh <filename>
  - First assign execute permission to the file and then execute directly with the file name
    - $chmod u+x <filename>
    - $./<filename>
  - Run with source or . command
    - . <filename> or source <filename> [dot has to be followed by space and then filename].
    - This doesnot span a new process.

Comments

- # is identified as comment
- It can be written as part of documentation
- It will not be interpreted

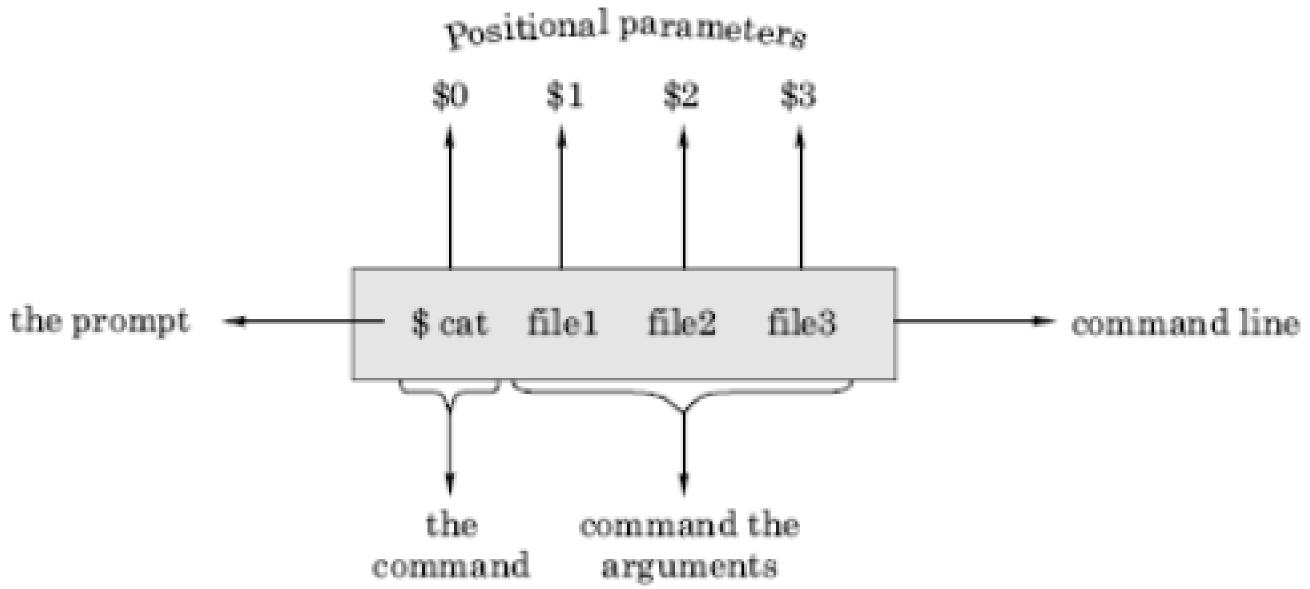
Read

- The read command is used to give input to a shell program (script) interactively
- This command reads just one line and assigns this line to one or more shell variables
  - echo what is your name \?
  - read name
  - echo The name is $name
- The read command can take multiple arguments
- Values for more than one variable can be input or assigned, using a single read command.
  - read a b c

- Here, number of values input are either less or more than the number of read's arguments
  - If number of values input are less than the number of arguments, then the arguments or variables to which values are not input will be initiated to null.
  - If number of values input are more than the number of arguments, then first (*n* – 1) values are assigned to the first (*n* – 1) arguments and all the remaining input values are assigned to the nth argument. This is an important feature. For example, let *w*, *x*, *y* and *z* are the four input values. The execution of the $read a b c command, assigns *w* to a, *x* to b, and *y* to c.

Positional Parameters

- Arguments submitted with a shell script are called positional parameters
- This method of passing on the values to a shell script is a non-interactive method
- First argument is passed on as parameter no 1, second argument is passed on as parameter no 2
- These are $1, $2, …, $9
- The arguments are assigned as values to the special variables $1, $2, $3


Positional parameters
$0
$1
$2
$3
the prompt
$ cat file1 file2 file3
command line
the command
command the arguments

- $0 is a special shell variable that holds the parameter number 0 (zero), the program name.
- $#, $* and $@ Variables
- $# variable holds a count of the total number of parameters
- $* variable holds the *list of all the arguments.*
- $@ variable also holds the list of all the arguments present in the command line
- $* and $@ when used within quote marks, the contents of $* is considered as a single string whereas in the case of $@ each of the arguments is independently quoted and considered as independent string arguments

set

- Assigning Values to Positional Parameters
- positional parameters cannot be assigned using the equal to (=) operator
  - $set friends in need are friends indeed
  - friends to the parameter $1, in to the parameter $2, need to the parameter $3 and so on.
    - $echo $1 $4 $6
    - friends are indeed
- **Displaying date in a required format**
  - **set `date`**
  - echo $1 $3 $2 $6

- **Positional Parameters and Excess Arguments**
  - $set Everyone has the capacity to learn from mistakes. He learns a lot from experience.
  - $echo $1 $2 $3 $4 $5 $6 $7 $8 $9 $10 $11
  - Everyone has the capacity to learn from mistakes. He Everyone0 Everyone1

- Set has multiple functionalities
- Set with no arguments and options – displays the system variables
- Set command with options
  - -x –v are used to debug scripts
- Set command with double hyphen(- -)
  - Set `ls –l myfile` [you will get error here bcoz – is considered as option to the set command.
  - So, replace with - - which has special meaning
    - Set - - `ls –l myfile`

Shift

- Handling Excess Command Line Arguments
- Only nine positional parameters are taken
- Excess will not be prompted as error
- Instead unambiguous results will be produced
- Such situations can be handled by shift command
- The shift statement shifts out the values assigned to the positional parameters to the *left* by an integer value mentioned with the shift statement as its argument
  - $shift 5 moves the parameter values to the left by five positions
- It should be noted that when certain values are shifted using a shift command, shifted values would be lost.

exit

- This command is used to terminate the execution of the script as and when required.
- It is not necessary to use at the end of every script
- This command can have numerical argument
- If no arguments is used, this command returns 0
- When an argument is given, it returns the non-zero value upon execution
- In general, 0 exit value indicates success and a non-zero indicates a failure

$?

- To know the exit status of command execution
- It can also be used in decision making in shell programs
- Usage: echo $?
- Eg: ls –l f1.txt

```
    f1.txt
    echo $?
     0
ls –l f2.txt
f2.txt:No such file or directory
echo $?
 1
```

expr for Calculations

- Variables as arguments:

```
count=5
count=`expr $count + 1`
echo $count
```

  - Variables are replaced with their values by the shell!
- expr supports the following operators:
  - arithmetic operators: +,-,*,/,%
  - comparison operators: <, <=, ==, !=, >=, >
  - boolean/logical operators: &, |
  - parentheses: (,)
  - precedence is the same as C, Java

Assigning Command Output to a Variable

- Using backquotes, we can assign the output of a command to a variable:

```
#!/bin/sh
files=`ls`
echo $files
```

- Very useful in numerical computation:

```
#!/bin/sh
value=`expr 12345 + 54321`
echo $value
```

Branching Control Structures

- Simple **If** control structure

```
if test_expression
then
          true-block
fi
```

- **If-else structure**

```
if test_expression
then
    true-block
else
    false-block
fi
```

- **If ---- elif ---- else**

```
if test_expression
then
    command(s)
elif test_expression
then
    command(s)
else
    command(s)
fi
```

Test statement

- Built-in shell command that evaluates the expression given to it as an argument
- Returns *true* if the evaluation of the expression returns a 0 (zero) or *false* if the evaluation returns non-zero.
- Eg: if test "$answer" = "Y” or if ["$answer" = "Y"]
- More than one condition can be tested by connecting them together using logical operators such as the logical and operator (–a), the logical or operator (–o) and the logical not operator (!).

- Test works in three ways
  - Compare two numbers
  - Compare two strings or a single one for a null value
  - Checks a file attributes

Numerical comparison with test

Operator	*Meaning*
–eq	equal to
–ne	not equal to
–gt	greater than
–ge	greater than or equal to
–lt	less than
–le	less than or equal to

```
x=5; y=7
if test $x –eq $y
then
   echo Both are equal
else
   echo Both are unequal
```

String Comparison

String Tests	*Meaning*
–z string	True if length of the string is zero, that is, the string is null.
–n string	True if length of the string is nonzero, that is, if string exists.
String1 = string2	True if string1 and string2 are identical.
String1 != string2	True if string1 and string2 are not identical.
String1	True if string1 is not the null string.

File related tests

Test	*Exit status*
–e file	True if file exists.
–f file	True if file exists and is a regular file.
–r file	True if file exists and is readable.
–w file	True if file exists and is writable.
–x file	True if file exists and is executable.
–d file	True if file exists and is a directory.
–c file	True if file exists and is a character special file.
–b file	True if file exists and is a block special file.
–h file	True if file exists and is a link file.
-s file	True if file exists and has a size greater than zero (0).

Case Conditional

Syntax:

```
case expression in
pattern1) commands1;;
pattern2) commands2;;
pattern3) commands3;;
…..
esac
```

Case Conditional

```
echo –e "Menu \n
1.List of files\n 2. Processes of user\n 3.Today's date\n 4. quit\n
    Enter your option:\c"
read choice
case "$choice" in
1)    ls -l;;
2)    ps –f ;;
3)    date;;
4)    exit;;
*)     echo "Invalid option"    # ;; not required for last option
esac
```

Contd

- Matching Multiple patterns

```
Echo "Do you wish to continue? (y/n):"
read ans
case "$ans" in
y/Y) ;;                 [yY] [eE] [sS]);;
n/N)exit;;              [nN] [oO])exit;;
esac
```

The while Loop

- While loops repeat statements as long as the next Unix command is successful.
- Syntax

```
while condition is true
do
        Commands
done
```

- For example:

```
#!/bin/sh
i=1
sum=0
while [ $i -le 100 ]; do
  sum=`expr $sum + $i`
  i=`expr $i + 1`
done
echo The sum is $sum.
```

for Loops

- for loops allow the repetition of a command for a specific set of values
- Syntax:

```
for var in value1 value2 …
do
  command set
done
```

  - command_set is executed with each value of var (value1, value2, …) in sequence

for Loop Example

```
#!/bin/sh
# times table – print out a multiplication table
for i in 1 2 3
do
   for j in 1 2 3
  do
     value=`expr $i \* $j`
      echo -n "$value "
  done
  echo
done
```

List in for

- List from variables
  - for i in $a $b $c
- List from wildcards
  - for file in *.c
- List from positional parameters
  - for pat in $*
- List from command substitution
  - for file in `cat clist`

Points to remember in for statement

- The list of values should not be separated by comma [eg: for day in Mon, Tue, Wed, Thu, Fri]
- The list of values should not be enclosed in a double quote. [eg: for day in "Mon Tue Wed Thu Fri“]
- As a best practice, you should always quote the bash variables when you are referring it. [eg: weekdays="Mon Tue Wed Thu Fri" ; for day in "$weekdays“]

- If you don’t specify the keyword “in” followed by any list of values in the bash for loop, it will use the positional parameters [eg: for day]
- You can use the output of any UNIX / Linux command as list of values to the for loop by enclosing the command in back-ticks ` ` [eg: for i in `date`]
- To loop through files and directories under a specific directory, just cd to that directory, and give * in the for loop [eg: cd ~ ; for item in *]

- When you don’t provide the start, condition, and increment in the bash C-style for loop, it will become infinite loop. [eg: for ((; ;))]
- Using comma in the bash C-style for loop [eg: for ((i=1, j=10; i <= 5 ; i++, j=j+5))]

The until Loop

- Until loops repeat statements until the next Unix command is successful.
- For example:

```
#!/bin/sh
x=1
until [ $x -gt 3 ]
do
   echo x = $x
   x=`expr $x + 1`
done
```

Statements which help in loop

- sleep secs – to delay for a while
- Break – come out of the scope of theloop prematurely
- Continue – to resume the loop with next iteration
- & to make the loop in background

More on expr command

- Expr works well for integer arithmetic
- For real numbers it can be done with the help of echo and bc statements
- Eg: c=`echo $a + $b |bc`
- Because of piping, echo does not display its output, rather it will redirect its output to the bc command.
- Scale function sets the precision for real numbers.
- area=`echo "scale=2 \n ½ *$base*$height" | bc`

Make the shell script as command

- How to execute without ./?
- **One way**
  - Include your current path in the PATH system variable
  - PATH = $PATH:<path>
  - You can check the path setting using echo $PATH
  - Now u can execute your script without ./
- **Second way**
  - gedit .bashrc
  - alias <aliasname>='./filename.sh’
  - save and close .
  - Then type this in your terminal source ~/.bashrc to apply the changes you made . then simply type the name you have given there after alias to access your script .
  - make sure that you have placed in the home directory .

Shell Script Debugging

- Set also serves as a debugging tool apart from assigning values to positional parameters
- When used inside the script or even at command prompt it echoes each statement on the terminal preceded by + symbol when each statement is executed.
- A statement set –x can be included in any script to invoke debugging
- To turn off set use set +x
- This will help to trace each line of the script and know why/where it is not working