LINUX SHELL PROGRAMMING

What is a shell?

- Command Interpreters
- Shells are used for direct user interaction and for smaller programming tasks or shell scripts

Linux Shells

- Bourne Shell - Sh (Steve Bourne) - original Unix shell
- Korn Shell - ksh (David Korn) - like sh + functions, history editing
- Z Shell - zsh (Paul Falstad) - like ksh with many enhancements
- Bourne Again Shell - bash (Ramey/Fox) - enhanced version of Bourne Shell GNU/Linux shell
- C Shell - csh (Bill Joy) - original C-syntax shell, + job control, history
- TC Shell- tcsh (Ken Greer,) - Enhanced version of csh

Linux Shells (Contd…)

- Shells differ in control structure syntax, scripting languages used by them
- Multi-user system can have a number of different shells in use at any time
- **/etc/shells** contains a list of valid shells
- **/etc/passwd** file includes the name of the shell to execute
- Exit a shell with **logout**, **exit** or **CTRL-D**
- **chsh** command - change your login shell

What shells do?

- **All Linux shells have the same basic mode of operation:**

loop
  if (interactive) print a prompt
  read a line of user input/file input
  apply transformations to line
  use first word in line as command name
  execute that command- using other words as arguments
  command not found - generate an error message
  found - execute the command
Wait until the command is finished
go back to loop

What shells do?

(Contd…)

- The "transformations" include:
  - history substitution, file name expansion, variable evaluation, ...
- To "execute that command" the shell needs to:
  - find the file containing the named program
  - start a new process

Command separation and grouping

- Command Separators
  - NEWLINE ('\n')
  - Semicolon(;)
- Use parenthesis to group commands
  - (a;b)&&c

Bourne Again Shell (bash)

- Command Interpreters as well as high level programming language
- As a programming language, it processes groups of commands stored in files called shell scripts

Programming or Scripting

- bash is not only an excellent command line shell, but a scripting language in itself. Shell scripting allows us to use the shell's abilities and to automate a lot of tasks that would otherwise require a lot of commands.

- Difference between programming and scripting languages:
  - Programming languages are generally a lot more powerful and a lot faster than scripting languages. Programming languages generally start from source code and are compiled into an executable. This executable is not easily ported into different operating systems.
  - A scripting language also starts from source code, but is not compiled into an executable. Rather, an interpreter reads the instructions in the source file and executes each instruction. Interpreted programs are generally slower than compiled programs. The main advantage is that you can easily port the source file to any operating system. bash is a scripting language. Other examples of scripting languages are perl, lisp, and tcl.

Simple shell script

- Shell script - file containing commands to be executed by shell
- Using shell scripts - initiate complex series of tasks or a repetitive procedure
- Shell interprets the shell scripts, executes the commands in the script sequentially

Startup files

- Bash uses collection of startup files to help create an environment to run in. Each file has a specific use and may affect login and interactive environments differently.

chmod

- chmod – change file mode bits
- u=user, g=group, o=other, a=all
- read=r=4 , write=w=2, execute=x=1
- + grant , - revoke

- Eg
- chmod u+x file1.c
- chmod u+x, g+x file1.c
- chmod a+x file1.c
- chmod u+x,g+x file1.c
- chmod --reference=file1 file2
- chmod -R u+x dir1
- chmod 755 file1
- chmod 111 file1
- chmod 777 file1

chmod

- STICKY bit

Sticky Bit is mainly used on folders in order to avoid deletion of a folder and its content by other users though they having write permissions on the folder contents. If Sticky bit is enabled on a folder, the folder contents are deleted by only owner who created them and the root user.

- sticky_bit=t=1 (t with x permission, T without x permission)
- chmod o+t /home/guest/dir1

or

chmod +t /home/guest/dir1

chmod 1755 /home/guest/dir1

chmod

- Set User ID/SUID

A program is executed with the file owner's permissions (rather than with the permissions of the user who executes it)

- SetUid=s=4 (s with x permission, S without x permission)
- chmod u+s /home/guest/dir1 or file1

chmod 4755 /home/guest/dir1 or file1

chmod

- Set Group ID/SGID

When setgid is set on a directory, Any file that is created inside a directory that has setgid will have the same group ownership as the parent directory with setgid.

- SetGid=s=2 (s with x permission, S without x permission)
- chmod g+s /home/guest/dir1 or file1

chmod 2755 /home/guest/dir1 or file1

- chmod u+s,g+s /home/guest/dir1 or file1

or

chmod 6777 /home/guest/dir1 or file1

umask

- The user file-creation mode mask (umask) is use to determine the file permission for newly created files. It can be used to control the default file permission for new files.

- Default umask value is 022
- Final permission for files

666-022 = 644

- Final permission for directory

777-022 = 755

touch

- touch file1.txt

- touch -t [[CC]YY]MMDDhhmm[.ss]
- touch -t 200101011200.09 file1.txt
- touch -t 01011200   file1.txt

tput and stty

- tput cols // no of colums
- tput lines // no of rows in given terminal
- tput cup 2 2 // move curser to 1, 1 position

- tput bold
- tput smul // set under line
- tupt rmul // remove underline

```
echo -e "Enter password: "
stty -echo
read password
stty echo
echo “your password is $password”
```

date and time

- date +%a // eg Fri
- date +%A // eg Friday
- date -d "Jan 01 2001" +%A //o/p Monday

- date +%B // eg September
- date +%b // eg Sep

- date +%Y // eg 2013
- date +%y // eg 13

- date “+%d %B %Y”
- date -s "27 September 2013 14:53:22"

date and time contd..

- date +%s // Epoch unix time in second (time 00:00:00 on 1 Jan 1970)
- date +%S // time in second from 0 to 60 sec

- date +%d // date in number eg 31
- date +%D // date in format eg 31/09/13

- date +%I // hours from 0 to 12
- date +%H // hours from 0 to 24
- date +%M // minutes eg 59

date and time contd..

```
start=$(date +%s)
sleep 3
end=$(date +%s)
difference=$(( end - start))
echo Time taken to execute commands is $difference seconds.
```

cal

```
cal
cal 10 2014
cal 2014
mm=10; yy=2014
cal $mm $yy

cal -3
cal -m3      // for given month
```

grep command

Global Regular Expression Print

cat /etc/passwd > test.txt

grep “/bin/bash” test.txt

grep -i “/Bin/Bash” test.txt //ignore the letter case

grep -v “/bin/bash” test.txt // inverse of the match

grep -n “/bin/bash” test.txt // pattern with numbering

grep -c “/bin/bash” test.txt // count of pattern lines

grep command

grep -e “root” -e “wimc” -e “ftp” test.txt
        // to match multiple patter
grep -l “root” test.txt   // display file name containing pat

grep “^r” test.txt   // display lines whose first char is r
grep “h$” test.txt   // display lines whose last char is h
grep “^[^r]” test.txt // display lines whose first char is not r
ls -l | grep “^-”   // display only regular files entry
ls -l | grep “^d”   // display only directory files entry
ls -l | grep “^l”   // disply only symbolic link files entry

grep command

grep -o “/bin/bash” test.txt

// show only the matched strings

grep -o -b “/bin/bash” test.txt

// show byte offset of matching line

grep -r “root” dir

- // seraching in all file recursively

grep -w “bash” file

// check and display full words, not a sub strings

grep command

grep -A|-B|-C <N> “/bin/bash” text.txt

grep -A 3 “/bin/bash”   test.txt
          // display 3 lines after match
grep -B 3 “/bin/bash”   test.txt
          // display 3 lines before match
grep -C 3 “/bin/bash”   test.txt
        // display 3 lines before and 3 lines after match

Simple shell script (Contd…)

- Example:

#!/bin/bash

(Shebang is a line for which #! is prefixed to the interpreter path. /bin/bash is the interpreter command path for Bash.)

Exercise:

- sh shellfile
- . shellfile
- ./shellfile
- shellfile
- bash shellfile
- source shellfile
- ksh/zsh/tcsh shellfile

Exercise:

- ignoreeof (disable CTRL-D logout)
- noclobber (dont overwirte files through redirection)
- noglob (disable special characters used for filename expansion: *,?,~,and [ ].)

- set -o feature (trunthe feature on)
- set +o feature (trun the feature off)

Shell Keywords

- **Keywords:**

echo read set unset
readonly shift export if
else fi while do
done for until case
esac break continue exit
return exec ulimit umask

Standard I/O

- The shell sets up the runtime environment
- One part of this is supplying command-line arguments and environment variables
- Another part is connecting the input/output streams for the new process
- By default it connects:

Name	**File Descriptor**	**Default Destination**
standard input (stdin)	0	keyboard
standard output (stdout)	1	screen
standard error (stderr)	2	screen

- pwd 1> file
- pwdd 2> errorfile
- read x 0< file

I/O Redirection

- I/O redirection - change where processes read from and write to.

- I/O Redirection used to -
  - read data from a file

    cat < /etc/passwd
  - write data to a file

    ls -lR > all.my.files
  - join UNIX commands together

grep "/bin/bash" /etc/passwd | more

Exercies:

- 2> --
- 2>> --
- 2>&1 -- error on stdout
- >& -- to a file or device
- ls -l *.c >programlist

Positional Parameters

- Positional parameters – command name and arguments, refer to them by their position on the command line Eg: $1-$9
  - can’t assign values to these parameters
  - Surround positional parameters consisting of more than a single digit with curly braces. Eg ${10}

Special Parameters

- Special parameters –$ with a special character
  - $# number of arguments
  - “$*” list of arguments “$1 $2 $3”
  - “$@” list of arguments “$1” “$2” “$3”
  - $$ PID of current shell
  - $? exit status of the last executed command
  - $! pid of the last command executed in the background
  - $0 name of script
- Possible to access useful values pertaining to command line arguments and execution of shell commands

Special Parameters

- More Special parameters
  - ${#@} number of positional parameter
  - ${#*} number of positional parameter
  - $- flags passed to script(using set)
  - $_ last argument of pervious command

Parameters and Variables

- Parameter is associated with a value that is accessible to the user
- Shell Variables - parameters whose names starts with a letter or underscore, can be alphanumeric
  - User-created variables
  - Environmental variables
- When we want to access a variable, we simply write its name; no need to declare them.
- When we want its value, we precede the name by a $.

```
$ read x
$ y=John
$ echo $x
$ z="$y $y"
```

Exercise:

- mkdir {fall,winter,spring}reprot
- echo doc{unment,final,draft}
- cat < myletter > newletter
- cat nodata 2> myerrors

Shell Variables

- Variables are local to the current execution of shell
- Every variable has initial value - null string.
- You can assign strings that look like numbers to shell variables. e.g. x=1 is equivalent to x="1"
- In some contexts, such values can be treated as numbers.
- In such contexts, non-numeric strings will be treated as zero.

  Example: $ x=5 y=6 z=abc

  $ echo $(($x + $y)) =>11

  $ echo $(($x + $z)) => 5

- ***Variable exists as long as the shell in which it was created exists.*** *To remove this variable use*

unset variable_name

Shell Variables contd..

```
var=cdac
echo $var

var=”cdac acts”
echo $var
echo ${var}

echo ${#var}                // length of variable
```

echo

```
echo "Hello world"
echo Hello world
echo 'Hello world'

year=2013
echo "Hello world $year"
echo Hello world $year
echo 'Hello world $year'
```

echo cont..

```
echo -e "1\t2\t3"

echo -e “enter your name:\c”
echo -n “enter your name:”
```

echo cont..

Printing a colored output:

Color codes reset 0 ; black 30; red 31; green 32

Yellow 33; blue 34; magenta 35; cyan 36; white 37

```
echo -e "\e[1;31m This is red text \e[0m"

echo -e "\e[4;31m This is red text \e[0m"
echo -e "\e[9;31m This is red text \e[0m"
```

echo cont..

Printing a colored background output:

Color codes reset 0 ; black 40; red 41; green 42

Yellow 43; blue 44; magenta 45; cyan 46; white 47

```
echo -e "\e[1;42m This is red text \e[0m"

echo -e "\e[4;42m This is red text \e[0m"
echo -e "\e[5;42m This is red text \e[0m"
```

printf

printf “hello world”

printf hello world
printf 'hello world'

printf “%5s” cdac
printf “%-5s” cdac

printf “%4.2f” 100.123
printf “%-5s %-10s %-4.2f\n” 1 cdac 100.123

Exercies:

- ldir=/home/dir/dirfile
- cp myfile $ldir

Environment Variables

- Available to all processes invoked by the shell
- Becomes available externally by being **exported**

Environment Variables (Contd…)

Variable Name	Purpose
HOME	The user's login directory
SHELL	The current shell being used
UID	Your user id
PATH	The executable path
MANPATH	where man looks for man pages
LD_LIBRARY_PATH	where libraries for executables are found at run time
TERM	the kind of terminal the user is using
DISPLAY	where X program windows are shown
HOSTNAME	the name of the host logged on to

Exercise:

- env
- printenv
- set

History

- Maintains a list of recently issued commands
- Provides shorthand for reexecuting any of the commands in the list.
  - history
  - !n
  - !!
- HISTFILE variable – holds name of the file that holds the history list
- HISTSIZE variable – determines no. of events preserved in the history list during a session

Processes

- Process is the execution of a command by Linux
- Shell is also a process
- When a shell script is executed, a shell process is created, which in turn creates additional processes for every command
- Process can create a new process by using fork system call
- When the shell built-in commands (echo, alias, bg, exec, history, jobs, dirs, pwd etc) are executed, a process is not created
- Unique process identification number (PID) is assigned at the inception of a process
- PID will be s ame as long as the process is in existence
- Child process will have different PID

- When the system is booted, **init** process with “PID 1” is started.
  - Ancestor of all processes that each user works with
  - Child process is orphaned, init process becomes the parent.
  - If any terminal is attached to the system, it forks getty process for each terminal.
    - getty process waits until a user starts to log in.
  - Action of logging in transforms getty process into login process and then into the user’s shell process.

Processes

(Contd…)

- **ps** command gives the list of processes
- **ps -l** gives a long listing of information about each process

bash $ ps –l

```
      UID   PID  PPID C STIME TTY   TIME CMD
root     1625  1624  0 12:41 ?       00:00:00 login -- globus
globus 1629  1625  0 12:41 pts/0    00:00:00 -bash
root    11886   659  0 15:07 ?       00:00:00 in.rlogind
root    11887 11886  0 15:07 ?       00:00:00 login -- rupa
rupa    12175 11892  0 15:21 pts/1    00:00:00 ps -ef
rupa    12176 11892  0 15:21 pts/1    00:00:00 more
```

(-) infront of bash indicates this shell was started by the login process

Exercise:

- sleep 50 &
- sleep 60 &
- fg %1
- jobs
- kill %1
- sleep 50 (press ctrl + z)
- bg (above proces will become a bg job)

Exercise:

- ps -lax
- pstree
- ps -e -o cmd,pid,ppid,pri,ni,class
- man ps
- Kill pid-no
- Kill -9 pid

Metacharacters

- Special characters to which special significance has been given
- These characters are accorded a **VIP** treatment by Linux
- Sometimes **metacharacters** are also called **‘regular expressions’**

Metacharacters

(Contd…)

Type	Metacharacters
Filename substitution	? * […] [!…]
I/O redirection	> < >> <<
Process	; () & && ||
Quoting metacharacter	\ “ ” ‘ ’ ``
Positional parameters	$1….$9
Special characters	$0 $* $@ $# $! $$

Filename Substitution Metacharacters

- Used for matching filenames in a directory
  - * A wild card character, it can represent any combination of any number of characters
  - ? Represents only one character
  - [..] Gives the shell a choice of any one character from the enclosed list
  - [!..] Gives the shell a choice of any one character except those enclosed in the list

Filename Substitution Metacharacters

(Contd…)

- Examples:
  - ls * Lists all files all
  - ls *.* List all files having ext
  - ls b* Lists all files beginning with character ‘b’
  - ls ?? Lists all files whose names are 2 characters long
  - ls e?f? Lists all 4 character filenames whose first character is ‘e’ and third character is ‘f’
  - ls [abd]* Lists all files whose first character is ‘a’, ‘b’ or ‘d’
  - ls [^abd]* Lists all files whose first character is not ‘a’, ‘b’ or ‘d’
  - ls [c-fmrv-z]* Lists all files whose first character is ‘m’ or ‘r’ or is in the range c to f or v to z
  - ls [!c-j]* Lists all files whose first character is anything other than an alphabet in the range c to j

I/O Redirection Metacharacters

- **NOTE:**

Refer I/O redirection slides presented earlier

Process Execution Metacharacters

- When we want to execute more than one command at the prompt in one stroke, we separate them with a semicolon.
  - Example:
    - ls ; date ; who ; banner Hey
- If we wish certain commands to be executed in a sub-shell, we enclose them in parentheses.
  - Example:
    - (cd newdir ; pwd)
- The metacharacter which delegates a given process execution to the background is **&**
  - Example:
    - sort long_file > result_file &

Conditional Execution Using && and ||

- These metacharacters can be used to optionally execute a command depending on the success or failure of the previous command

- If we want the second command to be executed only if the first succeeds, we write:

  Command_1 && Command_2

- Conversely, if we want the second command to be carried out only if the first fails, we write:

  Command_1 || Command_2

Examples:

```
grep college myfile && cat myfile
grep college myfile || cat myfile
```

Quoting Metacharacters

- The following characters come under this category:

  \ “ ” ‘ ’ ` `

- The \ character takes away the special significance attached to any metacharacter
  - Example:

```
echo  This is a *
echo  This is a \*
```

Quoting Metacharacters

(Contd…)

- The single quotes ‘ ‘ tell the shell to take every enclosed character literally – absolutely no favorites
  - Example:

echo ‘$, \, ?,*, this is india!’
$, \, ?,*, this is india!

Quoting Metacharacters (Contd…)

- The back quotes ` ` replace the command they enclose with its output.
  - Example:

echo Today is `date`

Today is Sun Nov 26 16:16:14 IST 2006

Quoting Metacharacters

(Contd…)

- The double quotes “ ” do pamper some metacharacters, namely, $, \ and ` `. When enclosed within “ ” these metacharacters are allowed to hold their special status.

  - Example:

```
bash $   name=ACTS
bash $   echo “Your college name is $name”
Your college name is ACTS
```

Command Substitution

- Use standard output of a command
  - $(command) or `command`

echo “today date is `date ` “
echo “today date is $(date) “

Exercise:

- winner=dylan
- result='The name is the $winner variable'
- echo $result

- winner=dylan
- result=”The name is the $winner variable”
- echo $result

Exercise:

- winner=dylan
- result=”$winner won \$100.00”
- echo $result

Command name/filename completion

- **TAB** performs command completion
- If only one command which matches the search is available
  - Typing name of the command, pressing TAB results in command completion
- If multiple commands, then bash beeps
  - Pressing TAB for the second time – displays a list of commands that matches the search
  - User intervention is required for identifying the match and then pressing TAB results in command completion.

Declare

- declare -i var_name=val // declare int var
- declare -r var_name=val // constant var
- declare -x var_name=val // exported
- declare -a arr_name
- declare -f fun_name
- Declare -l var_name=val //

- Declare -p var_name // display attributes and val

check man page

man builtins

Arithmetic Operations

- c=$(($a+$b));
- declare -i x=10, declare -i y=20, declare -i z

z=($x+$y); or z=$x+$y;

- The command let

a=10 ; b=20 ; let c=$a+$b; or let c=a+b;

- The command ***expr*** is used

```
echo  `expr  $x  +  $y`
echo  `expr  $x  +  5`
result=`expr  $x  +  $y`   or
result=$(expr $x + $y)
```

Arithmetic operations:

```
/*  %  +  -
```

Arithmetic Operations contd..

- let a=10 ; let b=20 ; let c=a+b
- var1=20
- let var1++
- echo $var1 // o/p is 21

- let var1+=5
- let var1-=1

- let var1=10
- let var2=20
- let result=$[var1 + var2]
- let result=$[var1 + 5]

Array

- arr=(10 20 30 40 50)
- or

  arr[0]=10 arr[1]=20 arr[2]=30

  arr[3]=40 arr[4]=50

- echo ${arr[1]}
- index=2
- echo ${arr[index]}

- echo ${arr[*]} or echo ${arr[@]}
- echo ${#arr[*]} or echo ${#arr[@]}

- Echo ${!arr[*]} or echo ${!arr[@]}

Associative Array

- In associative array, we can use any text data as an array index.
- declare -A associate_arr
- associate_arr=([ele1]=data1 [ele2]=data2)
- Or

associate_array[ele1]=data1

associate_array[ele2]=data2

-

- declare -A money_exc
- money_exc=([USdollar]=62 [AUSdollar]=50)

- echo ${money_exc[*]} or echo ${money_exc[@]}
- echo ${#money_exc[*]} or

echo ${#money_exc[@]}

- Echo ${!money_exc[*]} or echo ${!money_exc[@]}

funtion

```
function fun1( ) {
    echo “this is fun1 just to try”
}
Or
fun2( )  {
    echo “this is fun2 just to try”
}
fun2          // function call

fun2() {
    res = $(( $1 + $2 ))
    echo “ result is $res”
}
fun2   100 200    // function call with parameter
```

Funtion contd..

```
fun3( ) {
    printf “in function call”
    sleep 1
    fun3
}

Export -f fun3
```

Try command
type fun_name

Funtion contd..

```
fun5()
{
        res=$(($1 + $2))
        echo "res is $res"
        return $res
}

#result="$(fun5 100 200)"
fun5  10 20
#echo "return value from fun $result"
echo "return value from fun $?"
```

dot (.) command

- . ./shellprog
- when a script executes an external command or script, a new environment (a subshell) is cre ated, the command is executed in the new environment, and the environment is then discarded
- Same thing can be achived by

source shellprog

Variables

- We can use variables as in any programming languages. Their values are always stored as strings, but there are mathematical operators in the shell language that will convert variables to numbers for calculations.

- We have no need to declare a variable, just assigning a value to its reference will create it.

- Example
  - #!/bin/bash

```
STR=“Hello World!”
echo $STR
```

- Line 2 creates a variable called STR and assigns the string "Hello World!" to it. Then the value of this variable is retrieved by putting the '$' in at the beginning.

The export command

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

- If the child modifies x, it will not modify the parent's original value. Verify this by changing x in the following way:
  - $ x=ciao

$ exit
$ echo $x
hello

Exercise:

- PS1=---->
- export PS1
- bash

PS2 is variable which comes as secondary prompt

PS2=--

- PATH=$PATH:/home/guest/bin
- export PATH

- PS1= '\t \W '
- PS1 ='\t \h '
- PS1 ='\t \W \$'

The read command

- 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!“
```

Entering and Displaying Values

- The ***read*** and ***echo*** commands are used

Conditional Statements

- Conditionals let us decide whether to perform an action or not. This decision is taken by evaluating an expression. The most basic form is:
  - if [*expression*]

then

statements

elif [*expression*]

then

statements

else

statements

fi

  - the elif (else if) and else sections are optional

The *test* command

- This command is provided to specify the control statement or condition
- It can perform several types of tests like numeric test, string test and file test

Expressions (Contd.)

- Number (Arithmetic) 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:
  - [$var1 -eq $var2] (true if var1 same as var2, else false)
  - [$var1 -ge $var2] (true if var1 greater then or equal to var2, else false)
  - [$var1 -le $var2] (true if var1 less then or equal to var2, else false)
  - [$var1 -ne $var2] (true if var1 is not same as var2, else false)
  - [$var1 -gt $var2] (true if var1 greater then var2, else false)
  - [$var1 -lt $var2] (true if var1 less then var2, else false)

Expressions (Contd.)

- More Number (Arithmetic) Comparisons operator : use within double parentheses ((...))
  - > greter than
  - >= greter than or equal
  - < less than
  - <= less than or equal

Example

- **vi test1.sh**

```
clear
echo
echo –n “Enter a number: “
read num
if test $num  -eq  0
then
 echo “The number entered by you is zero”
elif test $num  -lt 0
then
 echo “The number entered by you is negative”
else
 echo “The number entered by you is positive”
fi
```

Without the *test* command

- Instead of specifying ***test*** command explicitly whenever we want to check for condition, the condition can be enclosed in square brackets

- Example is given on the next slide ➔ ➔ ➔

Example

- **vi test2.sh**

```
clear
echo
echo -n "Enter a number: "
read num
if [ $num -eq 0 ]
then
      echo "The number entered by you is zero"
elif [ $num -lt 0 ]
then
          echo "The number entered by you is negative"
else
      echo "The number entered by you is positive"
fi
```

Expressions

- An expression can be: String comparison, Numeric comparison, File operators and Logical operators and it is represented by [expression]:

- String Comparisons:
  - = compare if two strings are equal
  - != compare if two strings are not equal
  - -n evaluate if string is not null
  - -z evaluate if string is null

- Examples:
  - [$var1 = $var2] (true if var1 same as var2, else false)
  - [$var1 != $var2] (true if var1 not same as var2, else false)
  - [-n $var1] (true if var1 has a length greater then 0, else false)
  - [-z $var2] (true if var2 has a length of 0, otherwise false)

Expressions

- More String Comparisons operator:
  - == compare if two strings are equal
  - \< compare if first string is less than second
  - \> compare if first string is greter than second

Example

- **vi test3.sh**

```
clear
echo
echo -n "Enter two names: "
read name1 name2
if [ $name1 = $name2 ]
then
        echo "The names entered by you are the same"
else
        echo "The names are different"
fi
```

Expressions (Contd.)

- Files operators:
  - -d check if path given is a directory
  - -f check if path given is a file
  - -e check if file exists
  - -r check if read permission is set for file or directory
  - -s check if a file has nonzero size
  - -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 size is nonezero, 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 (Contd.)

- More Files operators:
  - -h check if given file is symbolic file
  - -b check if given file is block special device file
  - -c check if given file is character special device file
  - -p check if given file is named pipe file
  - -S check if given file is socket file
  - -O check if you own this file
  - -G check if group id of file same as current user

  - -u check if given file has set user id permission
  - -g check if given file has set group id permission
  - -k check if given file has sticky bit permission

  - F1 -nt F2 File F1 is newer than F2
  - F1 -ot F2 File F1 is older than F2
  - F1 -ef F2 Files F1 and F2 are hard links to the same file

Example

- **vi test4.sh**

```
clear
echo
echo -n "Enter the name of a directory: "
read dir_name
if [ -d $dir_name ]
then
        echo $dir_name is a directory
else
echo $dir_name is a file
fi
echo
```

case Statement

- Used to execute statements based on specific values. 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;
  - a *) is used to accept any value not matched with list of values

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

Example

- **vi test5.sh**

```
clear
echo
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
    echo
```

Iteration Statements

- The **for** structure is 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.

vi test6.sh

```
clear
echo
sum=0
for num in 1 2 3 4 5
do
sum=`expr $sum + $num`
done
echo The sum is: $sum
echo
```

```
for (( i=1; i<=5; i++ ))
 do
    for (( j=1; j<=i; j++ ))
        do
                echo -n "$i"
    done
done
```

Iteration Statements (Contd.)

- **vi test7.sh**

```
for x in paper pencil pen; do
  echo “The value of variable x is: $x”
  sleep 1
done
```

- if the list part is left off, var is set to each parameter passed to the script ($1, $2, $3,…)
  - **vi test8.sh**

```
for x
do
    echo “The value of variable x is: $x”
    sleep 1
done
./test8.sh eena meena deeka
```

while Statement

- The while structure is a looping structure. Used to execute a set of commands while a specified condition is true. The loop terminates as soon as the condition becomes false. If condition never becomes false, loop will never exit.
- while *expression*
  do
  statements
  done
- **vi test9.sh**

```
clear
echo
echo -n "Enter a number: "; read x
sum=0; i=1
while [ $i -le $x ]; do
sum=`expr $sum + $i`
i=`expr $i + 1`
done
echo
echo "The sum of the first $x numbers is: $sum"
echo
```

Menu example

```
clear ; loop=y
while [ “$loop” = y ] ;     do
  echo “Menu”;  echo “====”
  echo “D: print the date”
  echo “W: print the users who are currently log on.”
  echo “P: print the working directory”
  echo “Q: quit.”
  echo
  read –s choice
  case $choice in
      D | d) date ;;
      W | w) who ;;
      P | p) pwd ;;
      Q | q) loop=n ;;
      *) echo “Illegal choice.” ;;
  esac
  echo
done
```

continue Statement

- The **continue** command causes a jump to the next iteration of the loop, skipping all the remaining commands in that particular loop cycle.
- **vi test10.sh**

```
clear
echo
LIMIT=19
echo
echo "Printing Numbers 1 through 20 (but not 3 and 11)"
a=0
while [ $a -le $LIMIT ]; do
a=`expr $a + 1`
if [ $a -eq 3 ] || [ $a -eq 11 ]
then
continue
fi
echo -n "$a "
done
echo
echo
```

break Statement

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

Check out **test10.1.sh**

until Statement

- The **until** statement is very similar to the while structure. The until structure loops until the condition is true. So basically it is “until this condition is true, do this”.
- until [*expression*]
  do
  statements
  done
- **vi test11.sh**

```
echo “Enter a number: ”; read x
echo ; echo “Count down begins...”
until [ $x -le 0 ]; do
 echo $x
      x=`expr $x – 1`
      sleep 1
done
echo
```

Aliases

- Used as shorthand for frequently-used commands
- Syntax for bash – alias <shortcut>=<command>
  - alias ll="ls –lF"
  - alias la="ls –la"
- Put aliases in your .bashrc (if bash) file to set them up whenever you log in to the system!

Exercise:

- alias list=”ls -l”
- alias
- unalias list

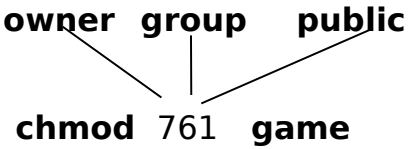
Access lists and groups

- Mode of access: read, write, execute
- Three classes of users

  RWX
  a) owner access 7 ⇒ 1 1 1

  RWX
  b) group access 6 ⇒ 1 1 0

  RWX
  c) public access 1 ⇒ 0 0 1
- Ask manager to create a group (unique name), say G, and add some users to the group.
- For a particular file (say *game*) or subdirectory, define an appropriate access.



Attach a group to a file

chgrp G game

Exercise: File Operations

- file *
- find reports -name monday
- find programs -name '*.c' -ls
- find . -name '*.*' -ls

Exercise: Link

- ln -s original-file added-file
- ln -s /root/file1 /root/dir1/dir2/dir3/file

- ln original-file added-file
- ln file1 file2

Exercise: Tar

- tar cvf mytarfile.tar mydir
- tar xvf mytarfile.tar

- tar rvf mytarfile.tar new_dir
- tar tvf mytarfile.tar
- tar uvf mytarfile.tar my_updated_dir

Exercise: Tar

- tar zcvf myarch.tar.gz mydirt
- tar zxvf myarch.tar.gz

- tar jcvf myarch.tar.bz2 mydir
- tar jxvf myarch.tar.bz2

Exercise: Zip

- zip unzip
- gzip gunzip
- bzip2 bunzip2
- xz unxz

File Extension:

- .rpm (redhat pakage manger file)
- .tz (tar with compress)
- .Z (compress)
- .bin (self extracting software file)
- .deb (debian linux package)

RPM:

- Rpm --install --verbose --hash --test sw.rpm
- Rpm -ivh sw.rpm
- Rpm -q sw

- Rpm -e sw
- Rpm -V bash
- Rpm -Va

Some Tips

- All shell variables are string variables . In the statement **a=20** the ’20’ stored in a is treated not as a number, but as a string of characters **2** and **0**. Naturally , we cannot carry out arithmetic operations on them unless we use a command called **expr** .
- A variable may contain more than one word . In such cases , the assignment must be made using double quotes .
- All the variable inside a shell script die the movement the execution of the script is over .
- We can create a null variable by following way

  $ d=“” $ d=‘’ $d=

- If a null variable is used any where in a command the shell manage to ignore it , For Example

  $ var=“” $var=“”

  $wc –l $var1 $var2 file1

- we can write comments using # as beginning character of the line
- The multiplication symbol must always be preceded by a \. Otherwise the shell treats it as a wildcard character for all files in the current directory.
- Term of the expression provided to **expr** must be separated by blanks.

  Thus , the expression **expr** 10+20 is invalid .
- **expr** is capable of carrying out only integer arithmetic. To carry out arithmetic on real number it is necessary to use the bc command.

  exa : a=10.2 b=12.2

  c=`echo $a + $b | bc`

Basic command used for filtering

tr

sed

head

tail

cut

sort

uniq

grep

Tr command

```
cat multi_blanks.txt | tr -s '\n'

echo "JUST TO TRY" | tr 'A-Z' 'a-z'

cat text | tr '\t' ' '

echo "JUST 123 TO 456 TRY" | tr -d '0-9'

echo "Hello 123 world 456" | tr -d [:digit:]

echo "JUST TO TRY" | tr [:upper:]  [:lower:]
```

sed command

```
echo new | sed  s/new/old/

echo “this is new” | sed  s/new/old/

sed  's/try/TRY'  < file1.txt

sed  -e  's/[0-9]//g'

var1=$(echo $input | sed -e 's/[0-9]//g')
```

sed command

```
Seq 1 30 > file1.txt
cat  file1.txt | sed -n 'p'
cat  file1.txt | sed '1d'
cat  file1.txt | sed '2d'
cat  file1.txt | sed '1,5d'
cat  file1.txt | sed '4,10d'
cat  file1.txt | sed '1,5!d'
cat  file1.txt | sed '1~3d'
cat  file1.txt | sed '2~2d'
```

sed command

```
cat /etc/passwd > file2.txt
cat  file2.txt | sed  's/root/toor'
cat  file2.txt | sed  's/root/toor/g'
cat  file2.txt | sed  's/Root/toor/ig'

cat  file2.txt | sed  's/root/toor/3'   //3rd instance
cat  file2.txt | sed  's/root//g'

cat  file2.txt | sed  '35s/bash/newBash/g'
cat  file2.txt | sed  '1,5s/bash/newBash/g'
cat  file2.txt | sed  's/root/r/g; s/kaushal/k/g'

cat  file2.txt | sed -e 's/root/r/' -e 's/kaushal/k'
cat  file2.txt | sed -n '/wimc15/,/vlsi15/p'       // print
```

sed command

```
cat /etc/passwd > file2.txt

cat  file2.txt | sed  's/^/>  /'
cat  file2.txt | sed  's/$/EOL/'

cat  file2.txt | sed  -n -e '/bash/p'

cat fileWithComment.txt | sed -n '/^#/p'
cat fileWithComment.txt | sed '/^#/d'

cat fileWithBlankLines.txt | sed  -n  '/^$/p'
cat fileWithBlankLines.txt | sed  '/^$/d'
                                  // delete blank lines
```

```
if [ -z $(echo $char | sed -e 's/[0-9]//g') ]

case $char in
[0-9]) echo "$char is Number/digit"    ;;
[[:upper:]]) echo "$char is UPPER character" ;;
[a-z]) echo "$char is lower character" ;;
*) echo "$char is Special symbol"  ;;



echo {1..9}
echo {a..z}
echo {A..Z}
```

seq

```
seq 1  10
seq 1   2  10
seq 10  -1  1
seq 10  -2  1
```

head and tail command

```
seq 1  20 > file.txt

head  file.txt
head -n 5 file.txt
head -n -5 file.txt // print all excluding last 5 lines

tail file.txt
tail -n 5 file.txt
tail -n +5 file.txt // print all excluding first 5 lines
```

bc

```
echo “100.25+200.50” | bc
echo  “sqrt(99)”  | bc
echo  “sqrt(99)”  | bc  -l
echo  “scale=2;sqrt(99)”  | bc
var1=99  ; echo “scale=2; sqrt($var1)”  | bc

var1=100; var2=200; echo “$var1/$var2” | bc

var1=100; var2=200; echo “scale=2;$var1/$var2” |
bc
```

Eg. Revese String

```
#!/bin/bash
var1=$1
length=`echo ${#var1}`
while [ $length -ne 0 ]
do
        temp=$temp`echo $var1 | cut -c $length`
        ((length--))
done
echo $temp
```

Need to explore temp=$temp and cut command

Eg. Revese String

To understand how temp=$temp works

Try following to append to existing value of variable

```
var1=1234
var1=$var1”56”                  // to append 56 to 1234
Or
var1=$var1'56'                  //

var2=56
var1=$var1`echo $varr2`    // with help of command

var1=cdac
var2=acts
var1=$var1$var2
```

Eg. Revese String

Using rev command

```
var1=$1
temp=`echo $var1 | rev`
echo $temp
```

// try tac command to display file content in reverse order

```
str=cdac
echo ${str:0:1}
echo ${str:0:2}

str=acts
i=0
while [ $i -lt ${#str} ]
do
        arr[$i]=${str:$i:1}
        let ++i
done
echo ${arr[2]}
echo ${arr[*]}
```

Awk

```
ls -l | awk '{print $1}'

ps lax | awk '{print $4}' | grep [1] | wc

ps lax | awk '{print $4}' |  grep  [1] | grep ^1$ | wc


cat /etc/passwd | cut -d \: -f 7 | uniq
cat /etc/passwd | cut -d : -f 7 | sort | uniq

df | head -n 2 | tail -n 1 | awk '{print $4}'
df | grep “/dev/sda*”  | awk '{print $4}'
```

```
i=0
for x in $@
do
        arr[i]=$x;
        let i++;
done

echo printing stuff
for((i=0 ; i<$# ; i++))
do
echo ${arr[i]}
done
```

```
function()
{
    var=55555
    echo $var
}

ret=$(function)

echo  $ret
```

```
function()
{
        var=256;
        echo 3;
        return $var;
}
ret=$(function)
echo $ret

echo $?
```

```
BACKUP_FOLDER_NAME="BACKUP_"$(date +%d_%m_
%Y_%H_%M_%S)

mkdir $BACKUP_FOLDER_NAME
```

Exercise: File operation

- chown username filename
- chown -R username filename

- chgrp groupname filenam
- chgrp -R groupname filename

Exercise: File operation

- Check list
- mount and unmount command
- .bashrc file
- dirs, pushd, popd