
UNIT I

Shell Scripting

Introduction

Shell

- Interface to the user
 - Command interpreter
 - Programming features
-

Shells in Linux

- Many shells available
- Examples -sh, ksh, csh, bash
- Bash is most popular

Shell's Relationship to the User and the Hardware

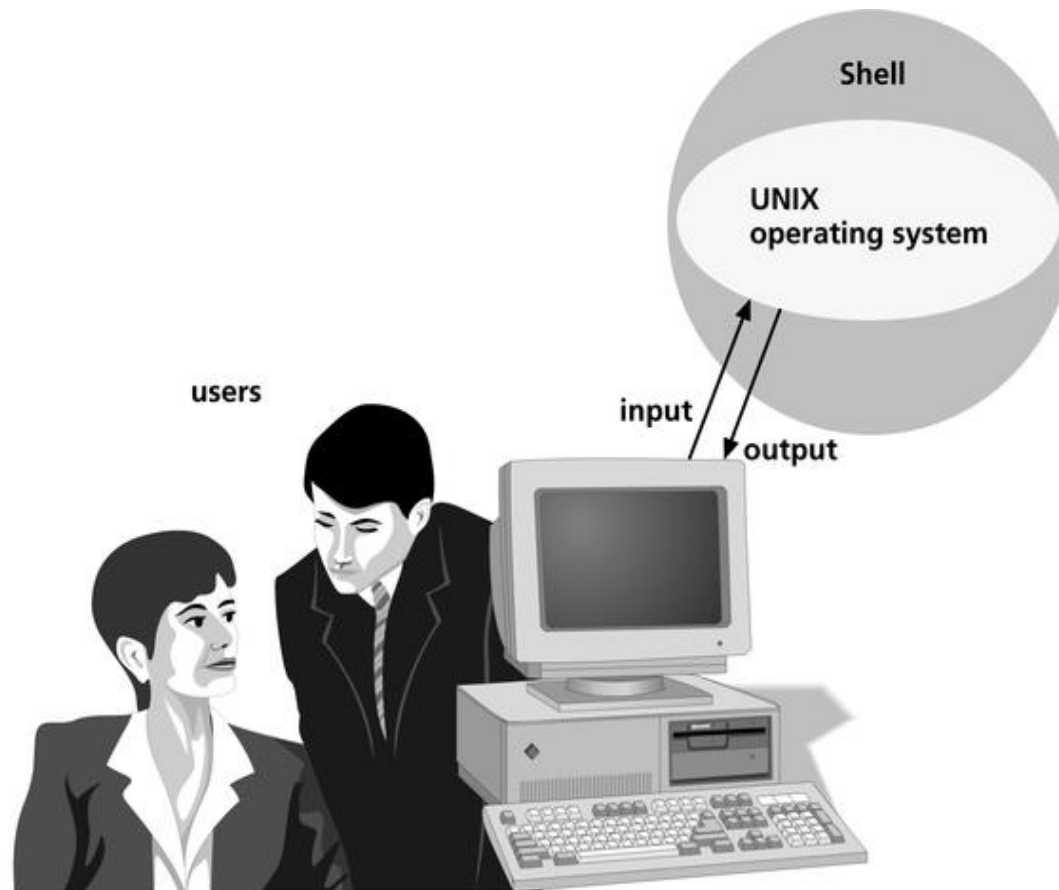


Figure 1-6: Shell's relationship to the user and the hardware

Programming or Scripting ?

- Bash is not only an excellent command line shell, but a scripting language in itself.
 - Difference between programming and scripting languages:
 - ❑ Programming languages are generally a lot more powerful and a lot faster than scripting languages.
 - Start from source code and are compiled into an executable.
 - Run independent of an exterior (or parent) program
 - C, C++, C#, Java,
 - ❑ A scripting language also starts from source code, but is not compiled into an executable.
 - Interpreter reads the instructions in the source file and executes each instruction.
 - Interpreted programs are generally slower than compiled programs.
 - Can port the source file to any operating system. bash is a scripting language.
 - Run inside another program
-
- Other examples of scripting languages are Perl, Lisp, and Python, PHP, JavaScript.

Shell Scripting

- ◆ A shell program is nothing but a series of commands
- ◆ We give the to-do list – a prg- that carries out an entire procedure. Such programs are known as “shell scripts”
- ◆ Automating command execution
- ◆ Batch processing of commands
- ◆ Repetitive tasks

When to use shell scripts

Shell scripts can be used for a variety of tasks

- Customizing your work environment
 - Every time login to see current date, welcome message etc
- Automating your daily tasks
 - To take backup all your programs at the end of the day
- Automating repetitive tasks
 - Producing sales report of every month etc
- Executing important system procedures
 - Shutdown, formatting a disk, creating a file system on it, mounting and unmounting the disk etc
- Performing same operation on many files
 - Replacing printf with myprintf in all C programs present in a dir etc

When not to use shell scripts

When the task :

- is too complex such as writing entire billing system
- Require a high degree of efficiency
- Requires a variety of software tools.

Disadvantages

- Compatibility problems between different platforms.
- Slow execution speed.
- A new process launched for almost every shell command executed.

The Bourne Again Shell

- Abbreviated bash
- Default in most Linux distributions
- Most widely used on all UNIX platforms
- Derives features from ksh, csh, sh, etc.
- Cat /etc/shells:to see all shell with their path

BASH Programming features

- Support for many programming features
 - variables, arrays, loops, decision operators, functions, positional parameters
 - Pipes, I/O re-direction
 - Misc. features - job control
 - Built in Commands - read, echo, source, alias
-

Shell keywords

- These are words whose meaning has already been explained to shell
- We cannot use as variable names
- Also called as reserved words

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

Variable

- In Linux (Shell), there are two types of variable:
 - (1) **System variables** - Created and maintained by Linux itself. This type of variable defined in CAPITAL LETTERS.
 - (2) **User defined variables (UDV)** - Created and maintained by user. This type of variable defined in lower letters.

- BASH=/bin/bash Our shell name
- HOME=/home/vivek Our home directory
- LOGNAME=students Our logging name
- PATH=/usr/bin:/sbin:/bin:/usr/sbin Our path settings
- PS1=[\u@\h \W]\\$ Our prompt settings
- PWD=/home/students/Common Our current working directory
- SHELL=/bin/bash Our shell name
- USERNAME=vivek User name who is currently login to this PC

TERM =xterm name of the terminal on which you are working

We can see all the system variables and their values using

\$ set or env

How to define User defined variables (UDV)

To define UDV use following syntax:

■ variable name=value

- '**value**' is assigned to given '**variable name**' and Value must be on right side = sign.
- *Example:*
- \$ no=10 # this is ok
- \$ 10=no # Error, NOT Ok, Value must be on right side of = sign.
- To define variable called 'vech' having value Bus
\$ vech=Bus
- To define variable called 'n' having value 10
\$ n=10

Rules for Naming variable name (Both UDV and System Variable)

- Variable name must begin with Alphanumeric character or underscore character (_), followed by one or more Alphanumeric character.

e.g. **HOME,SYSTEM_VERSION,vech,no**

- Don't put spaces on either side of the equal sign when assigning value to variable.

e.g. \$ no=10

- Variables are case-sensitive.

e.g.

\$ no=10

\$ No=11

- Define NULL variable as follows.

e.g.

\$ vech=

\$ vech=""

- Do not use ?,* etc, to name variable names.

- To print or access UDV use following syntax

Syntax:

`$variablename`

- Unset is used to wipe of shell variables

`$unset a`

We cannot wipeout system variables

`$unset PS1` `// will not work`

- To declare constants we use readonly key word

`$a=20`

`$readonly a`

Shell does not allow us to change the value of a

`$readonly` `// display all readonly vars`

echo Command

- Use echo command to display text or value of variable.

- `echo [options] [string, variables...]`

Options

-n Do not output the trailing new line.

-e Enable interpretation of the following

\a alert (bell)

\b backspace

\c suppress trailing new line

\n new line

\r carriage return

\t horizontal tab

\\ backslash

Quoting example

```
$ echo Today is date  
Today is date
```

"	- double quotes
'	- single quote
`	- back quote

```
$ echo Today is `date`  
Today is Thu Sep 19 12:28:55 EST 2016
```

```
$ echo "Today is `date`"  
Today is Thu Sep 19 12:28:55 EST 2016
```

```
$ echo 'Today is `date`'  
Today is `date`
```

Shell Arithmetic

- Use to perform arithmetic operations.

- *Syntax:*

expr op1 math-operator op2

- *Examples:*

```
$c=`expr 1 + 3` ;      $ c=`expr 10 / 2`  
$ c=`expr 20 % 3` ;    $ c=`expr 10 \* 3`  
$ echo `expr 6 + 3`
```

- expr is capable of carrying out only integer arithmetic
- For float operations we have to use bc command
- Ex: a=10.5;b=3.5

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

Arithmetic Evaluation

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

```
$ let X=10+2*7
```

```
$ echo $X
```

```
24
```

```
$ let Y=X+2*4
```

```
$ echo $Y
```

```
32
```

- An **arithmetic expression** can be evaluated by **\$(expression)** or **`\${expression}`**

```
$ echo “${(123+20)}”
```

```
143
```

```
$ VAL=${123+20}
```

```
$ echo “${10*$VAL}”
```

```
1430
```

Starting Shell Scripting

A Simple Shell Script -

```
#!/bin/bash
```

```
# my first script
```

```
echo "hello world"
```

- ◆ Save as “my_first_script.sh”
 - ◆ First line: special directive
 - ◆ Second line: comment
-

Running A Shell Script -

```
./my_first_script.sh
```

```
chmod 755 my_first_script.sh
```

```
chmod +x my_first_script.sh
```

```
$PATH
```

```
bash my_first_script.sh
```

Execute permission not required

Starting Shell Scripting

A Simple Shell Script -

1.#!/bin/bash

2.# testfile.sh ←———— Comment statement

3.# Uses the date,cal,shell to display

4.# date,calander & shell of system

5. echo "Displaying date: `date`"

6. echo "Displaying calander: `cal`"

7. echo "My shell :\$SHELL"

Starting Shell Scripting

Running A Shell Script -

- Method 1

```
$ chmod +x testfile.sh
```

```
$ ./testfile.sh
```

- Method 2

```
$ bash testfile.sh
```

- Options of sh Command

-n : Read commands, but does not execute them.

-v : Prints the input to the shell as the shell reads it.

-x : Prints command lines and their arguments as they are executed.

This option is used mostly for debugging.

The read Statement

- Use to get input (data from user) from keyboard and store (data) to variable.

Syntax:

read variable1, variable2,...variableN

- n helps to keep the cursor at the same line
- Read -p :input and prompt on same line
- only read -> stores in REPLY
- read -t 3 response (timeout)
- read -s password
- read -a names: as an array
- read fname mname lname

```
#Script to read your name from key-board  
echo "Enter any two numbers:"  
read n1 n2  
n3=`expr $n1 + $n2` # n3=$(( $n1 + $n2 ))  
  
echo " n3=$n3"
```

Wild cards

- * Matches any string or group of characters

Ex: `$ ls *` will show all files

- ? Matches any single character

Ex : `$ ls f?` will show all files whose names are 2 character long and start with 'f'.

- [...] Matches any one of the enclosed characters

Ex : `$ ls [abc]*` will show all files beginning with letters a/b/c

More command on one command line

- *Syntax:*

command1;command2

To run two command with one command line.

Examples:

\$ date; who

Will print today's date followed by users who are currently login.

- Note that You can't use

\$ date who

for same purpose

Executing commands within script

- `echo date`
- `echo `date``
- `echo $(date)`
- `right_now=$(date)`

Why Command Line arguments required

- Positional parameters are assigned from the shell's argument when it is invoked.
- Positional parameter “N” may be referenced as “\${N}”, or as “\$N” when “N” consists of a single digit.
- Telling the command/utility which option to use.
- Informing the utility/command which file or group of files to process (reading/writing of files).

■ `$ myshell foo bar`
 `$0` `$1` `$2`

0 - Shell Script name i.e. myshell

1 - First command line argument passed to myshell i.e. foo

2- Second command line argument passed to myshell i.e. bar

Command line arguments

- `$1,$2...$9` –positional parameter representing command line args
- `$#` -total no. of args
- `$0` –name of executed command
- `$*` - gives a single word containing all the parameters passed to the script
- `$@` gives a **array of words containing all the parameters** passed to the script
- `$?` –exit status of last command
- `$$` - pid of current shell
- `$!` -pid of last background process

Setting values of Positional parameters

- We can't assign values to positional parameters directly

1) `$set hello cs students`

2) `$set `cat f1` //f1 is a file`

- If quoting meta characters are used the command given with in the `` (reverse quotes) is replaced by the output

- Set command can set positional parameters upto 9

If we use more we can't access after 1 directly

eg: `$set a b c d e f g h I j k l ; $echo $10 $11`
a0 a1

- To access parameters after 9 , use shift

Eg: `$shift 2`

- First 2 words gone and lost forever

Indirection

```
name="vit"
```

```
reference=name
```

```
echo "${!reference}"
```

```
echo $reference
```

- output: vit

```
name
```

```
name="vit"
```

```
reference=$name
```

```
echo "${!reference}"
```

- echo \$reference

- Output: vit

- vit

Conditional execution i.e. && and ||

- The logical operators are && (read as AND) and || (read as OR).
- The syntax for AND list is as follows :
command1 && command2
 - command2 is executed if, and only if, command1 returns an exit status of zero.(success)e.g grep “date” mydate && echo “pattern found”
- The syntax for OR list as follows :
command1 || command2
 - command2 is executed if and only if command1 returns a non-zero exit status.(failure)e.g. grep “date” mydate || echo “pattern not found”

Conditional execution i.e. && and ||

- You can use both as follows
command1 && command2 (if exist status is zero) || command3 (if exit status is non-zero)
if command1 is executed successfully then shell will run command2 and if command1 is not successful then command3 is executed.
- *Example:*
\$ rm myf && echo "File is removed successfully" || echo "File is not removed"
- If file (myf) is removed successfully, then "*echo File is removed successfully*" statement is executed, otherwise "*echo File is not removed*" statement is executed

BASH Features

Decision operators

- if

```
if <condition>
```

```
then
```

```
    <do something>
```

```
else
```

```
    <do something else>
```

```
fi
```

- Can use the 'test' command for condition

```
if test $a = $b
```

```
then
```

```
    echo $a
```

```
fi
```

Test & []

■ The test Command

- The test command is a built-in shell command that evaluates the expression given to it as an argument and return true if the expression is true, if otherwise, false

■ Invoking test with Brackets

- You can use square brackets ([]) instead of the word test.
- Compares two strings or a single one for a null value
- Compares two numbers
- Checks files attributes

Numeric comparison

test number1 numeric test operator number2

-eq Is number 1 equal to number2 ?

-ne Is number1 not equal to number2 ?

-gt Is number 1 great than number2 ?

-ge Is number1 great than or equal to
number2 ?

-lt Is number 1 less than number2 ?

-le Is number1 less than or equal to number2 ?

String comparison

= string1 = string2 True if both strings are same

!= string1 != string2 True if both strings are different

-n -n string True if string is not empty

-z -z string True if string is empty

File related tests

- `-e` if file exist
- `-f` file exists & is regular
- `-r` file exists & is readable
- `-w` file exists & is writable
- `-x` file exists & is executable
- `-d` file exists & is directory
- `-s` file exists & has size greater than zero
- `f1 -nt f2` f1 is newer than f2
- `f1 -ot f2` f1 is older than f2
- `f1 -ef f2` f1 is linked to f2

-
- Every command in Linux has an exit status
 - 0 -> success, any thing else -> failure
 - Try echo \$? after executing a command
 - Two special commands : true and false
 - ; - command separator when on same line
 - Return status is the exit status of the last command executed
-

Numeric data

```
#!/bin/sh
echo -e "enter two numbers:\n"
read n1 n2
if [ $n1 -gt $n2 ]
then
    echo "$n1 is greater"
else
    echo "$n2 is greater"
fi
```

Run it as:
\$ sh test.sh
enter two numbers:
3 5
5 is greater

Example

```
#!/bin/bash
if [ -f /etc/passwd ];
then
    cp /etc/passwd .
    echo "Done."
else
    echo "This file does not exist."
    exit 1
fi
```

String data

```
echo "enter the string to be searched:"  
read pat  
if [ -z $pat ]; then  
echo " you have not entered string"; exit 1  
fi  
echo "enter the file name:"  
read fname  
if [ ! -n $fname ]; then  
echo " you have not entered filename"; exit 2  
fi  
grep $pat $fname
```

File data

```
echo "enter file name:"
read fname
if [ ! -e fname ]; then
echo " file does not exist"
elif [ ! -r fname ]; then
echo "file is not readable"
elif [ ! -w fname ]; then
echo "file is not writable "
else
    echo "file is both  readable & writable"
fi
```

Example of if statement

```
$ cat > showfile
#!/bin/sh
#Script to print file
if cat $1
then
echo -e "\n\n File $1, found"
fi
```

- Run above script as:
\$ chmod 755 showfile
OR
\$./showfile foo

FOR 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.**
- Example

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

Example of for loop

```
#!/bin/sh
for i in 1 2 3 4 5
do
    echo "Welcome $i times"
done
```

Run it above script as follows:

```
$ chmod +x testfor
$ ./testfor
```

- The for loop first creates i variable and assigns a number to i from the list of number from 1 to 5, The shell execute echo statement for each assignment of i.
- This process will continue until all the items in the list were not finished, because of this it will repeat 5 echo statements.

Example of for loop

```
#!/bin/bash
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,...)

```
$ cat for1.sh
#!/bin/bash
for x
do
    echo "The value of variable x is: $x"
    sleep 1
done
```

```
$ for1.sh arg1 arg2
```

The value of variable x is: arg1

The value of variable x is: arg2

A C-like for loop

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

```
for (( EXPR1 ; EXPR2 ; EXPR3 ))
```

```
do
```

```
    statements
```

```
done
```

- First, the arithmetic expression EXPR1 is evaluated. EXPR2 is then evaluated repeatedly until it evaluates to 0. Each time EXPR2 is evaluated to a non-zero value, statements are executed and EXPR3 is evaluated.

```
$ cat for2.sh
```

```
#!/bin/bash
```

```
echo -n "Enter a number: "; read x
```

```
for (( i=1 ; $i<$x ; i=$((i+1)) )); do
```

```
sum = `expr $sum + $i`
```

```
done
```

```
echo "the sum of the first $x numbers is: $sum"
```

```
#!/bin/sh
#Script to test for loop
if [ $# -eq 0 ]
then
echo "Error - Number missing form command line
argument"
echo "Syntax : $0 number"
exit 1
fi
n=$1
for i in 1 2 3 4 5 6 7 8 9 10
do
echo "$n * $i = `expr $i \* $n`"
done
```

■ run it as:

```
$ chmod 755 mtable
$ ./mtable 7
```

■ **7 * 1 = 7**
7 * 2 = 14

```
#!/bin/sh
for (( i = 1; i <= 5; i++ ))    ### Outer for loop ###
do

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

    echo "" ##### print the new line ###

done
```

- Run the above script as follows:

```
$ chmod u+x nestedfor.sh
```

```
$ ./nestedfor.sh
```

```
1 1 1 1 1
```

```
2 2 2 2 2
```

```
3 3 3 3 3
```

```
4 4 4 4 4
```

```
5 5 5 5 5
```

While Statements

- 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

\$ cat while.sh

#!/bin/bash

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 "the sum of the first \$x numbers is: \$sum"

While loop

```
■ #!/bin/sh
#Script to test while statement
if [ $# -eq 0 ]
then
    echo "Error - Number missing from
command line argument"
    echo "Syntax : $0 number"
    exit 1
fi
n=$1
i=1
while [ $i -le 10 ]
do
    echo "$n * $i = `expr $i \* $n`"
    i=`expr $i + 1`
done
```

Run the above script as follows:

The case Statement

- The case statement is good alternative to Multilevel if-then-else-fi statement.
- It enable you to match several values against one variable.
- Its easier to read and write.

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

Case example

```
echo -e "1.List of files\n
```

```
2.No. of processes\n
```

```
3.Today's date\n
```

```
4.Logged users\n
```

```
5.exit\n"
```

```
echo "Enter your choice"
```

```
read ch
```

```
case $ch in
```

```
1)ls ;;
```

```
2)ps ;;
```

```
3)date ;;
```

```
4)who ;;
```

```
5)exit ;;
```

```
*) echo "Wrong choice, enter again"
```

```
esac
```


Case Statement - an example

```
#!/bin/sh
```

```
echo -n "Enter key: "
```

```
read key
```

```
case "$key" in
```

```
a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z)
```

```
echo "$key is a lower case."
```

```
::
```

```
A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z)
```

```
echo "$key is an upper case."
```

```
::
```

```
0|1|2|3|4|5|6|7|8|9) echo "$key is a number."
```

```
::
```

```
*) echo "$key is a punctuation."
```

```
::
```

```
esac
```

Case Statement - a better example

```
#!/bin/sh
read -n1 -p "Hit a key: " key
case "$key" in
    [a-z])    echo "$key is a lower case."
    ;;
    [A-Z])    echo "$key is an upper case."
    ;;
    [0-9])    echo "$key is a number."
    ;;
    *)        echo "$key is a punctuation."
    ;;
esac
```

Example of case statement

```
#!/bin/sh
if [ -z $1 ]
then
    rental="*** Unknown vehicle ***"
elif [ -n $1 ]
then
    rental=$1
fi
case $rental in
    "car") echo "For $rental Rs.20 per km";;
    "van") echo "For $rental Rs.10 per km";;
    "jeep") echo "For $rental Rs.5 per km";;
    "bicycle") echo "For $rental 20 paisa per km";;
    *) echo "Sorry, I can not get a $rental for you";;
esac
```

■ run it as follows:

```
$ chmod +x car
```

```
$ car van
```

```
$ car car
```

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.

```
$ cat continue.sh
```

```
#!/bin/bash
```

```
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
```

Break Statements

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

```
$ cat break.sh
```

```
#!/bin/bash
```

```
LIMIT=19
```

```
echo
```

```
echo "Printing Numbers 1 through 20, but something happens after 2 ... "
```

```
a=0
```

```
while [ $a -le "$LIMIT" ]
```

```
do
```

```
  a=`expr $a + 1`
```

```
  if [ "$a" -gt 2 ]
```

```
  then
```

```
    break
```

```
  fi
```

```
  echo -n "$a "
```

```
done
```

```
echo;
```

```
exit 0
```

Until Statements

The until structure 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
```

```
$ cat countdown.sh
```

```
#!/bin/bash
```

```
echo "Enter a number: "
```

```
read x
```

```
echo ; echo Count Down
```

```
until [ "$x" -le 0 ]
```

```
do
```

```
    echo $x
```

```
    x=`expr $x - 1`
```

```
    sleep 1
```

```
done
```

```
echo ; echo GO !
```

How to de-bug the shell script?

- Need to find the errors (bugs) in shell script and correct the errors (remove errors - debug).
- Use -v and -x option with sh or bash command to debug the shell script.
- General syntax is as follows:
sh option { shell-script-name }
OR
bash option { shell-script-name }
- Option can be
 - v Print shell input lines as they are read.
 - x After expanding each simple-command, bash displays the expanded value of PS4 system variable, followed by the command and its expanded arguments

```
#!/bin/sh
```

```
tot=`expr $1 + $2`  
echo $tot
```

- run it as
\$ **chmod 755 test.sh**
\$ **./dsh1.sh 4 5**
9
\$ **sh -x test.sh 4 5**

- *# Script to show debug of shell*
tot=`expr \$1 + \$2`
expr \$1 + \$2
++ expr 4 + 5
+ tot=9
echo \$tot
+ echo 9
9

- See the above output, -x shows the exact values of variables

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
```

Functions

- Function is series of instruction/commands.
- Function performs particular activity in shell i.e. it had specific work to do or simply say task.
- To define function use following syntax
- `function-name ()`
`{ command1 command2 commandN return }`
- **\$ SayHello()**
{
echo "Hello \$LOGNAME, Have a nice day"
return
}

- To execute this function just type its name as follows:
\$ SayHello
Hello Aparna, Have a nice day.
- After restarting your computer you will lose this SayHello() function, since it's created for current session only.
- To overcome this problem, add your function to */etc/bashrc* file.
- To add function to this file you must logon as root.
- First logon as root or if you already logon with your name, and want to move to root account, then use following command
- **\$ su**
- Open file **/etc/bashrc** using vi and goto the end of file (by pressing shift+G) and type the SayHello() function:

Functions

```
#!/bin/bash
hello()
{
echo "You are in function hello()"
}
```

```
echo "Calling function hello()..."
hello
echo "You are now out of function hello()"
```

- In the above, we called the `hello()` function by name by using the line: `hello` . When this line is executed, bash searches the script for the line `hello()`. It finds it right at the top, and executes its contents.

- Functions are normally defined on the command line or within a script

- # vi /etc/bashrc

```
SayHello()
```

```
{
```

```
    echo "Hello $LOGNAME, Have a nice day"
```

```
    return
```

```
}
```

- **Function Chaining:** It is the process of calling a function from another function

```
#!/bin/bash
```

```
orange () { echo "Now in orange"
```

```
apple
```

```
}
```

```
apple () { echo "Now in apple" }
```

```
orange
```

Functions

```
$ cat function.sh
```

```
#!/bin/bash
```

```
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
```

Function: example

```
factorial()  
{  
    if [ $1 -le 1 ]  
    then  
        return 1  
    else  
        a=`expr $1 - 1`  
        factorial $a  
        b=`expr $1 \* $?`  
        return $b  
    fi  
}  
factorial 4  
echo "factorial 3 = $?"
```

Recursion

```
#!/bin/bash
#Recursive factorial function
factorial()
{
local=$1
if [ $local -le 1 ]
then
echo $local
else
a=`expr $local - 1`
#Recursive call
factorial $a
f=`expr $? \* $local`
echo $f
fi
}
```

```
#main script
read -p "Enter the number:" n
if [ $n -eq 0 ]
then
echo 1
else
#calling factorial function
factorial $n
fi
```


Local & global variables

global x and y

x=200

y=100

math(){

local variable x and y with passed args

local x=\$1

local y=\$2

echo \$((\$x + \$y))

}

echo "x: \$x and y: \$y"

call function

echo "Calling math() with x: \$x and y: \$y"

math 5 10

x and y are not modified by math()

echo "x: \$x and y: \$y after calling math()"

echo \$((\$x + \$y))

Unsetting Functions

- Once a function has been defined, it can be undefined via the unset command:

Syntax: `unset fun_name`

- For example,
- `unset SayHello`
- After a function has been unset it cannot be executed

Alias

- An *alias* is an abbreviation or an alternative name, usually mnemonic, for a command.
- Aliases are defined using the alias command:

Syntax: alias name="cmd"

Ex: alias ls="ls -l"

- **Unalias** : Once an alias has been defined, it can be unset using the unalias command:

Syntax: unalias name

- Ex: unalias ls

Alias vs function

- Aliases are similar to functions in that they associate a command with a name. Two key differences are
 1. In an alias, cmd cannot be a compound command or a list.
 2. In an alias, there is no way to manipulate the argument list (\$@).
- Due to their limited capabilities, aliases are not commonly used in shell programs.

Using Arrays with Loops

- In the bash shell, we may use **arrays**. 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)
```

- We may have **up to 1024 elements**. To **extract** a value, type **`${arrayname[i]}`**

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

```
dog
```

- 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
```

BASH Features: I/O Redirection

1. Operators- >, <, >>, >&, |

2. File Redirection

☐ **command >file (cat> fname)**

☐ **command <file (tr <fname)**

☐ **command 2> file (cat fname; if fname does not exist)**

BASH Features

Environment

- env variables - \$HOME, \$PWD, \$LOGNAME
- Aliases - alias cp='cp -i'
- Use .bashrc to set env variables and aliases

Job Control

- Foreground and Background jobs (&)
- Ctrl-z, bg, fg, jobs
- kill %job

BASH Features

Quoting

- **Escape character (\)**

echo \$hello

\$hello

- **Partial quoting ("...")**

**Escape special characters using **

Prevents word-splitting

- **Full quoting ('...')**

Cannot use escape sequences

Cannot embed single quotes

Other Shell-scripting tools

Basic commands

- ls, cat ,cp, mv, find, expr ,date, time, etc

Filters

- grep, sort, uniq, diff, cut, paste, etc.
- Regular Expressions

Other programs

- sed, awk, lex, yacc
- tty, stty, tset

Regular Expressions

- **Pattern matching rules**
- **Used in grep, sed, awk**
- **Basic reg. exps**
 - **, ^, \$, *, ?, +, [^...]**

Other Shell-scripting tools

Sed

- Search and Replace
- Delete

Awk

- Search and process patterns
- Process input streams

Usage of Shell-scripts

Where to use shell-scripting

- System Administration

- Automate tasks
- Repeated tasks

- Development

- Allows testing a limited sub-set of functionality

- Testing tools

- Daily usage

- Simple scripts

- reminders, diary, mail, etc.
-

Usage of Shell-scripts

Where Not to use shell-scripting

- Resource-intensive tasks
 - Cross-platform portability (Use C, JAVA, etc.)
 - Complex and mission-critical applications
 - Where security is important
 - Proprietary, closed-source applications
-

Optimising scripts

Speed improvement

- Use/write programs for slow operations

e.g: use grep to speeden searches and awk for mathematical ops

- Optimise loops
- Minimise I/O : BASH is slow with files
- Use awk, Perl, etc. where speed is reqd.

Keeping scripts small

- Modularised scripting
-

Script to reverse given no

```
echo "enter any number"
```

```
read n
```

```
rev=0 ;
```

```
sd=0
```

```
while [ $n -gt 0 ]
```

```
do
```

```
sd=`expr $n % 10`
```

```
rev=`expr $rev \* 10 + $sd`
```

```
#echo -n $sd
```

```
n=`expr $n / 10`
```

```
done
```

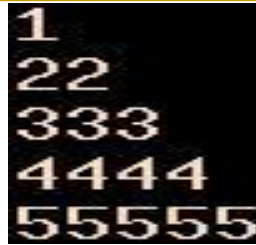
```
echo "Reverse number is $rev"
```

Write script to print given numbers sum of all digit

```
echo "enter any number"
read n
sum=0 ;sd=0
while [ $n -gt 0 ]
do
    sd=`expr $n % 10`
    sum=`expr $sum + $sd`
    n=`expr $n / 10`
done
echo "Sum of digit for number is $sum"
```

System date wise msg

```
hr=`date | cut -c12-13`  
if [ $hr -lt 12 ]; then  
mess="Good Morning $LOGNAME, Have nice  
day!"  
fi  
if [ $hr -gt 12 -a $hr -le 16 ] ;then  
mess="Good Afternoon $LOGNAME"  
fi  
if [ $hr -gt 16 -a $hr -le 18 ]; then  
mess="Good Evening $LOGNAME"  
fi
```



```
1
22
333
4444
55555
```

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



```
1
12
123
1234
12345
```

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

```
echo "Stars"
```

```
for (( i=1; i<=5; i++ ))
```

```
do
```

```
for (( j=1; j<=i; j++ ))
```

```
do
```

```
echo -n " *"
```

```
done
```

```
echo ""
```

```
done for
```

```
(( i=5; i>=1; i-- ))
```

```
do
```

```
for (( j=1; j<=i; j++ ))
```

```
do
```

```
echo -n " *"
```

```
done
```

```
echo "" done
```



```
MAX_NO=0
```

```
echo -n "Enter Number between (5 to 9) : "
```

```
read MAX_NO
```

```
if ! [ $MAX_NO -ge 5 -a $MAX_NO -le 9 ]  
; then
```

```
echo "I ask to enter number between 5 and  
9, Okay"
```

```
exit 1
```

```
fi
```

```
clear
```

```
for (( i=1; i<=MAX_NO; i++ ))
```

```
do
```

```
for (( s=MAX_NO; s>=i; s-- ))
```

```
do
```

```
echo -n " "
```

```
done
```



```
for (( j=1; j<=i; j++ ))
do
echo -n " $i"
done
echo ""
done
for (( i=1; i<=MAX_NO; i++ ))
do
for (( s=MAX_NO; s>=i; s-- ))
do
echo -n " "
done
for (( j=1; j<=i; j++ ))
do
echo -n " ."
done
echo " done "
```

Script to sort n numbers in ascending order

```
echo "enter array size"
read n
echo "enter array elements"
for (( i = 0; i < n ; i++ ))
do
    read nos[$i]
done
# Now do the Sorting of numbers -
for (( i = 0; i < n ; i++ ))
do
    for (( j = $i; j < n; j++ ))
    do
```

Script to sort n numbers in ascending order

```
if [ ${nos[$i]} -gt ${nos[$j]} ]; then
    t=${nos[$i]}
    nos[$i]=${nos[$j]}
    nos[$j]=$t
fi
done
done
echo "sorted elements"
for (( i = 0; i < n ; i++ ))
do
    echo ${nos[$i]}
done
```