

Linux Programming

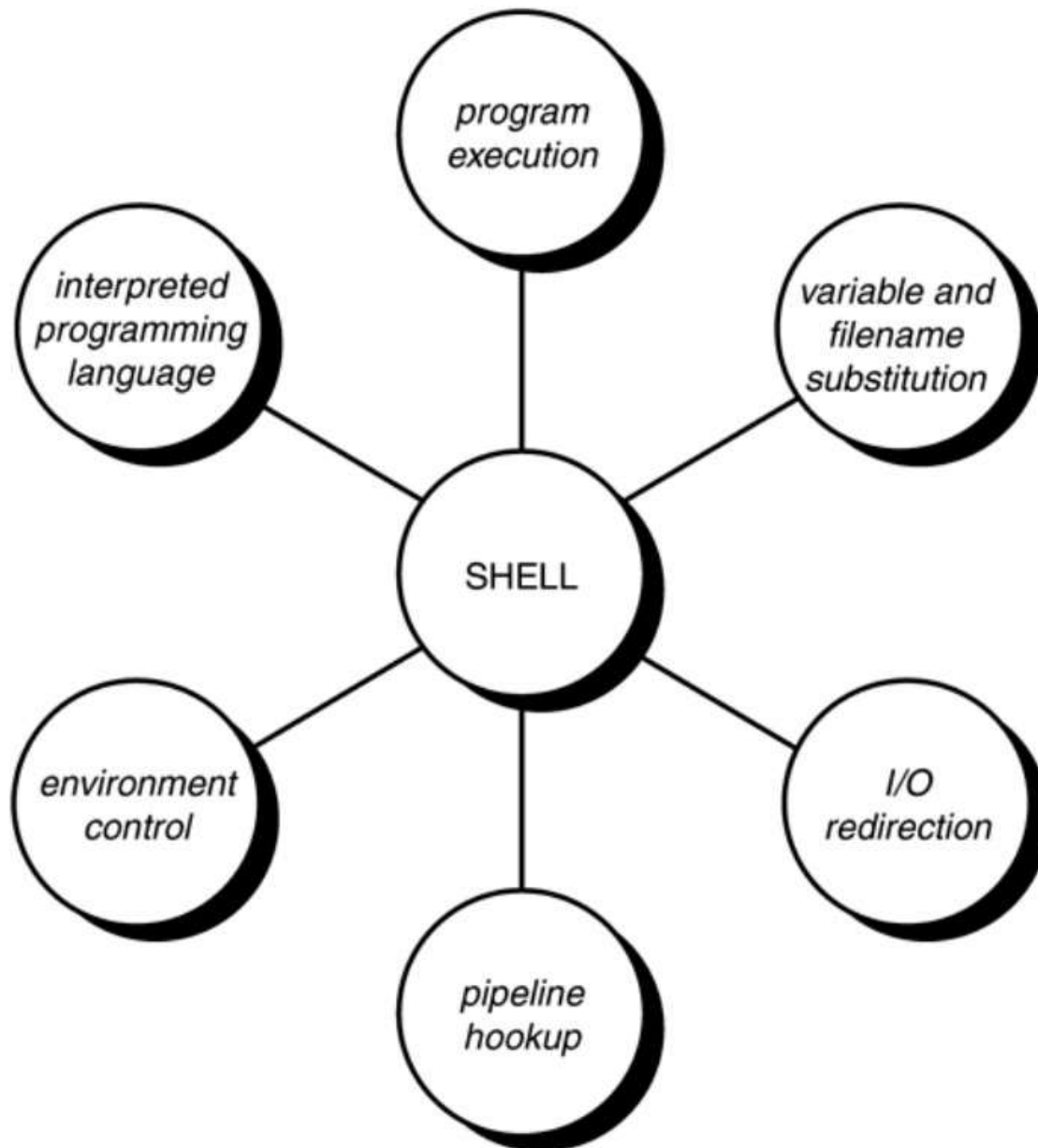
II-UNIT

What is shell ?

The shell is an **interface** between the user and the operating system.

The shell is also known as **command line interpreter**. Since user can't interact with the **kernel** directly, shell will pass the command line to the kernel for execution.

Shell Responsibilities



Program Execution

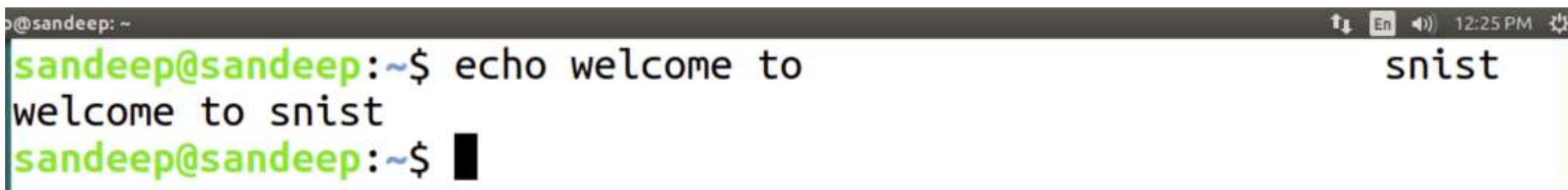
- ⌞ The shell is responsible for the execution of all programs that you request from your terminal.
- ⌞ Each time you type in a line to the shell, the shell analyzes the line and then determines what to do.
- ⌞ Each line follows the same basic format:

Syntax : **program-name arguments**

Ex: `ls -l`

- ⌞ The shell scans this command line and determines the name of the program to be executed and what arguments to pass to the program.
- ⌞ multiple occurrences of whitespace characters are ignored by the shell.

Ex : `echo welcome to snist`

A screenshot of a terminal window. The prompt is 'sandeep@sandeep: ~\$'. The user has entered the command 'echo welcome to snist'. The output of the command is 'welcome to snist', which is displayed on the line immediately following the command. The prompt 'sandeep@sandeep: ~\$' is shown again on the next line, followed by a black cursor block. The terminal window has a dark gray title bar with the text '@sandeep: ~' on the left and system icons (up/down arrows, 'En', a speaker icon, and the time '12:25 PM') on the right.

```
sandeep@sandeep: ~$ echo welcome to snist
welcome to snist
sandeep@sandeep: ~$
```

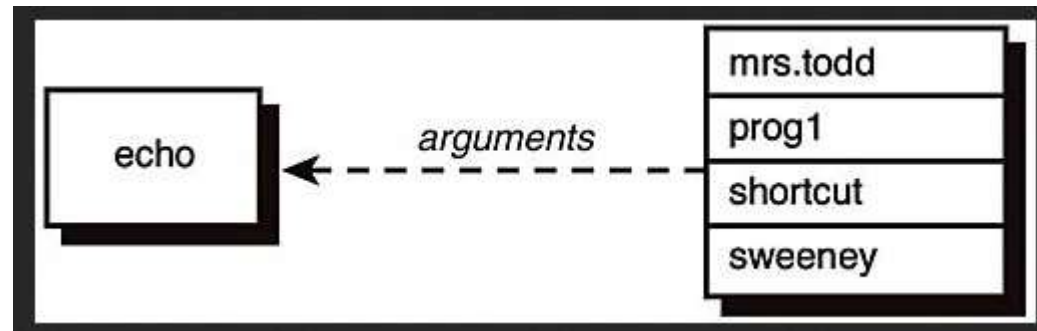
Variable and Filename Substitution

- λ Shell lets you assign values to variables.
- λ Shell substitutes a variables whenever it is preceeded by \$ sign
- λ Shell performs file name substitution on command line
- λ shell scans the command line looking for filename substitution characters *, ?, or [...] before determining the name of the program to execute and its arguments.

Example : variable name substitution

```
sandeep@snist: ~  
Search your computer  
sandeep@snist:~$ read -p "Assign value for variable X=" X  
Assign value for variable X=10  
sandeep@snist:~$ echo $X  
10  
sandeep@snist:~$
```

Variable and Filename Substitution



Example : File name substitution

```
sandeep@snist: ~/D1
sandeep@snist:~$ mkdir D1
sandeep@snist:~$ cd D1
sandeep@snist:~/D1$ touch f1 f2 f3
sandeep@snist:~/D1$ echo *
f1 f2 f3
sandeep@snist:~/D1$
```

shell recognizes the special character * and substitutes on the command line the names of all files in the current directory

I/O Redirection

- λ It scans the command line for the occurrence of the special redirection characters <, >, or >>

```
sandeep@snist: ~
```

```
sandeep@snist:~$ cat >users
```

```
alice
```

```
bob
```

```
steve
```

```
rob
```

```
snist
```

```
sandeep@snist:~$ wc -l users
```

```
5 users
```

```
sandeep@snist:~$ wc -l < users
```

```
5
```

```
sandeep@snist:~$ cat >>users
```

```
sreenidhi
```

```
student
```

```
sandeep@snist:~$ cat users
```

```
alice
```

```
bob
```

```
steve
```

```
rob
```

```
snist
```

```
sreenidhi
```

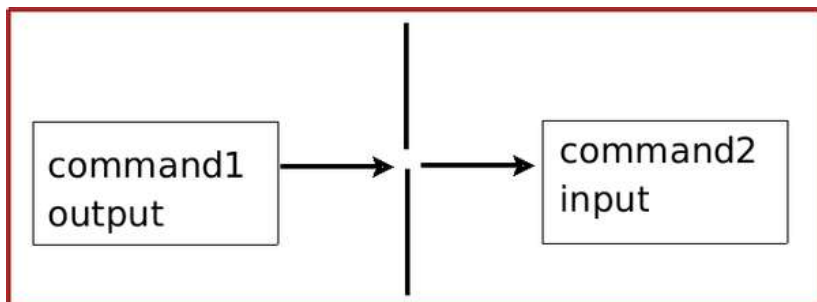
```
student
```

```
-
```

Pipeline Hookup

Shell connects the standard output from the command preceding the | to the standard input of the one following the |. It then initiates execution of both programs.

Syntax: `command_1 | command_2 [| command_3 . . .]`



```
sandeep@snist: ~  
sandeep@snist:~$ ls | wc -l  
78  
sandeep@snist:~$ █
```

```
17311A0501@cse-lab: ~  
[17311A0501@cse-lab ~]$ who | wc -l  
62  
[17311A0501@cse-lab ~]$ █
```


Pipeline Hookup

How to find the number of lines in a file?

```
sandeep@snist: ~  
sandeep@snist:~$ cat users  
alice  
bob  
steve  
rob  
snist  
sreenidhi  
student  
sandeep@snist:~$ cat users | wc -l  
7  
sandeep@snist:~$ █
```

Environment Control

An environment variable is a dynamic-named **value** that can affect the way running **processes will behave** on a computer. They are part of the environment in which a process runs

- Give information about the system behavior

```
sandeep@snist: ~  
sandeep@snist:~$ printenv  
sandeep@snist:~$ env
```

```
sandeep@snist: ~  
sandeep@snist:~$ echo $PATH  
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/  
games:/usr/local/games:/snap/bin  
sandeep@snist:~$ echo $HOME  
/home/sandeep  
sandeep@snist:~$ echo $USERNAME  
sandeep
```

Interpreted Programming Language

- The shell has its own built-in programming language.
- This language is interpreted, meaning that the shell analyzes each statement in the language one line at a time and then executes it.
- This differs from programming languages such as C and FORTRAN, in which the programming statements are typically compiled into a machine executable form before they are executed.

Interpreted Programming Language

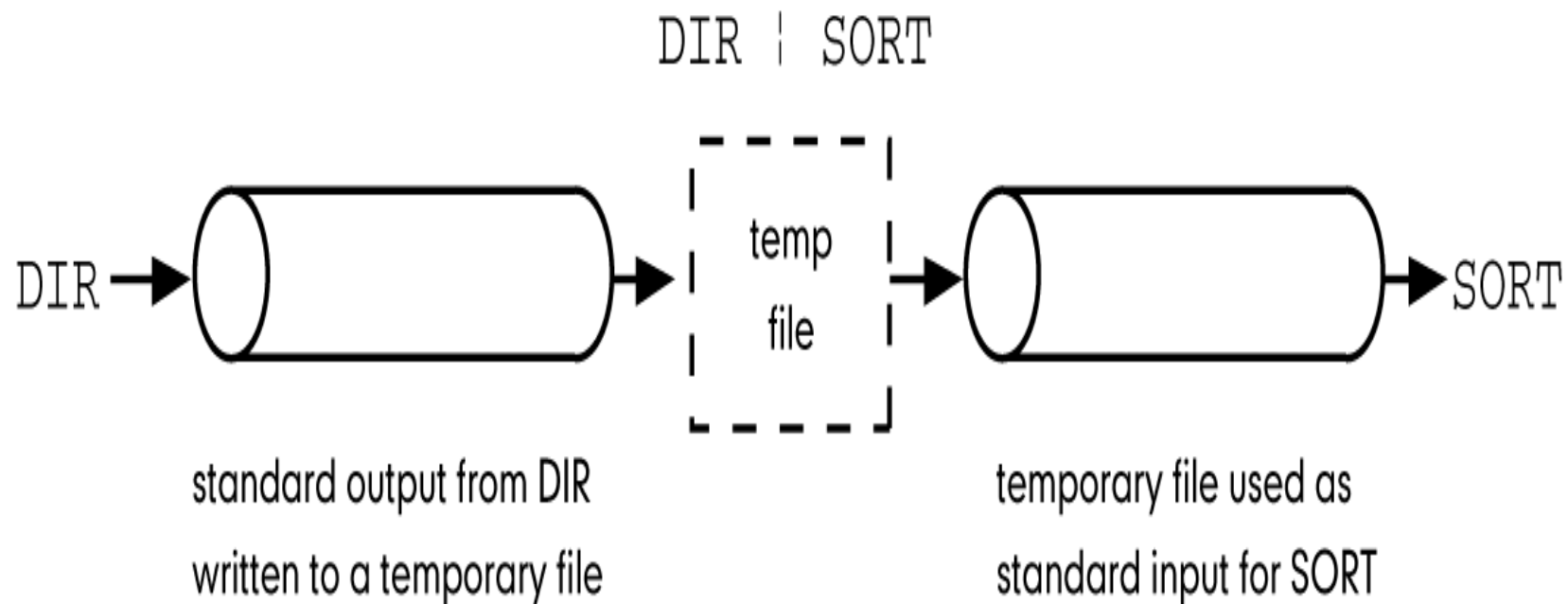
- Easier to debug and modify than compiled ones.
- Take much longer to execute than their compiled equivalents.
- Provides features like other programming languages. looping constructs, decision-making statements, variables, and functions, and is procedure-oriented.
- Modern shells based on the IEEE POSIX standard have many other features including arrays, data typing, and built-in arithmetic operations.

Pipes And Input Redirection, Output Redirection

Pipes:

- ➔ Standard output of one program used as standard input to next program
- ➔ Used with filter commands to further refine data
- ➔ Not limited to two programs

➔ Pipe symbol is the vertical broken bar | and is used between two commands.



```
snist@snist-HP-280-G2-SFF:~/cse$ who > f1.txt
```

```
snist@snist-HP-280-G2-SFF:~/cse$ cat f1.txt
```

```
snist    tty7          2017-12-27 14:41 (:0
```

❑ Now use `wc` to make a complete count of this file contents, we use `-l` option to count the number of lines only

```
snist@snist-HP-280-G2-SFF:~/cse$ wc -l < f1.txt
```

```
1
```

```
snist@snist-HP-280-G2-SFF:~/cse$
```

❑ Using intermediate file(`f1.txt`), we effectively counted the number of users.

- ❑ This method of running two commands separately has two obvious disadvantages
 - For long running commands, this process can be slow. The second command can't act unless the first has completed its job
 - You need an intermediate file that has to be removed after completion of the job. When handling large files temporary files can build up easily and eat up disk space in no time
- ❑ Here `who`'s standard output was redirected and so was `wc`'s standard input, and both used the same disk file.
- ❑ The shell can connect these streams using a special operator `|` (pipe) and avoid creation of the disk file.

```
snist@snist-HP-280-G2-SFF:~/cse$ who | wc -l
```


- ❑ Here, the output of **who** has been passed directly to the input of **wc**. And **who** is said to be piped to **wc** .
- You can know use one to count the number of files in the current directory
 - **snist@snist-HP-280-G2-SFF:~/ cse \$ ls | wc -l**

3

snist@snist-HP-280-G2-SFF:~/ cse \$

- There is no restriction on the number of commands you can use in a pipeline. But you must know the behavioural properties of these commands to place them there.

Redirection-Input Redirection, Output Redirection

- ❑ In the context of redirection, the terminal is a generic name that represents the screen, display or keyboard
- ❑ We see command output and error messages on the terminal (display), and we sometimes provide command input through the terminal (keyboard)
- ❑ The shell associates three files with the terminal
 - ❑ Two for the display and one for the keyboard
- ❑ Even our terminal is also represented by specific device name (/dev/tty)
- ❑ They perform all terminal related activity with the three files that the shell makes available to every command.

- ❑ These special files are actually streams of characters which many commands see as input and output
- ❑ A stream is simply a sequence of bytes
- ❑ When a user login, the shell makes available three files representing three streams
- ❑ Each stream is associated with a default device
 - ❑ Standard Input – The file(or stream) representing input, which is connected to the keyboard
 - ❑ Standard Output – The file(or Stream) representing output, which is connected to the display
 - ❑ Standard Error – The file(or Stream) representing error messages that emanate from the command or shell, which is connected to the display

- ❑ Every command that uses streams will always find these files open and available
- ❑ The files are closed when the command completes execution

Standard Input:

- ❑ Redirection of input is not as common as redirection of output.
- ❑ Most commands are designed to take their input from files anyway, instead of from *STDIN*.
- ❑ You redirect input by using the < operator. For example:
 more < killout.txt
- ❑ If you call up a command that expects a filename, and you don't provide one, input will come from the keyboard until a **Ctrl-D** is read.

❑ Example of a command using *STDIN as the input file*.

\$: cat

Some text is typed here

- This text will be stored up and sent to *STDOUT* when a **Ctrl-D** is read.
- ^D (user pressed Ctrl-D)
- Some text is typed here.

Ex: `wc < sample.txt`

- On seeing the `<`, the shell opens the disk file, sample .txt for reading
- It unplugs the standard input file from its default source and assign it to sample.txt
- `wc` reads from standard input which has earlier been reassigned by the shell to sample.txt

Standard Output:

- ❑ All commands displaying output on the terminal actually write to the standard output file as a stream of characters .
- ❑ There are three possible destinations of this stream
 - ❑ The terminal, the default destination
 - ❑ A file using the redirection symbols `>` and `>>`
 - ❑ As input to another program using a pipeline
- ❑ You can replace the default destination (the terminal) with any file by using the `>` operator followed by file name
- ❑ ex: `$ wc sample.txt > newfile`
`$ cat newfile`

`3 14 71 sample.txt`

- ❑ If the output file doesnot exist, the shell creates it before executing the command.
- ❑ If it exists,the shell overwrites it.
- ❑ The shell provides the >> symbol to append to the file

Ex: \$ wc sample.txt >> newfile

How it works:

Ex: wc sample.txt > newfile

- On seeing the > , the shell opens the disk file, new file, for writing
- It unplugs the standard output file from its default destination and assigns it to the newfile

- `wc` (not the shell) opens the file `sample.txt` for reading
- `wc` writes to standard output which has earlier been reassigned by the shell to `newfile`.

Standard Error:

- ❑ The standard files is represented by a number called file descriptor
- ❑ A file is opened by referring to its pathname, but subsequent read and write operations identify the file by this file descriptor
- ❑ The kernel maintains a table of file descriptors for every process running in the system.

- ❑ The first three slots are generally allocated to the three standard streams .
 - ❖ 0 – standard input
 - ❖ 1 – standard output
 - ❖ 2 – standard error
- ❑ These descriptors are implicitly prefixed to the redirection symbols
- ❑ For instance, `>` and `1>` mean the same thing to the shell, while `<` and `0<` also identical
- ❑ When you enter an incorrect command or try to open an nonexistent file, certain diagnostic messages show up on the screen, this is called standard error stream

Ex: `$ cat foo`

cat: cannot open foo

- ❖ cat fails to open the file and writes to the standard error
- ❖ Error stream cant be captured with `>`

Ex: `cat foo > errorfile`

- The diagnostic output has not been sent to errorfile. It's obvious that standard error cant be redirected in the same way standard output can (with `>` or `>>`)
- Even though standard output and standard error use the terminal as the default destination, the shell possesses a mechanism for capturing them individually.
- Redirecting standard error requires the use of the `2>` symbols

Ex:

```
$ cat foo 2> errorfile
```

```
$ cat errorfile
```

```
cat: cannot open foo
```

- This works, you can also append diagnostic output in a manner similar to the one in which you append standard output.

Ex:

```
$ cat foo 2>> errorfile
```

Running a Shell Script

- You can type in a sequence of commands and allow the shell to execute them interactively, or you can store these commands in a file which you can invoke as a program.
- The collection of Unix commands is called a script
- Script is weekly or loosely typed program.
- To check available shells

\$ cat /etc/shells

Making Scripts Executable

- After creating a script, we must make it executable.
- This is done with the ***chmod*** command.
- *Example:*
 - ***\$ chmod 744 script_file***
 - ***\$ chmod u+x script_file***

Executing the Script

- After the script has been made executable, it is a command and can be executed just like any other command
- Two methods of executing a shell script: as an independent command or as an argument to a subshell command.

- ***Independent Command***

- To Execute a bash shell script we do not need to be in the bourne again shell as long as the interpreter designator line is included as the first line of the script.
- To execute the script as an independent command, we simply use its name as in the following example

\$ script_name

- ***Child Shell Execution***

- To ensure the script is properly executed, we can create a child shell and execute it in the new shell.
- This done by specifying the shell before the script name as in the following example:

\$ bash script_name

- In this case, the interpreter designator line is not needed, but the user needs to know which shell the script requires.
- But the above method is very error-prone method, it is recommended that all scripts include the interpreter designation

- A command is a program which interacts with the kernel to provide the environment and perform the functions called for by the user.
- The shell is a command line interpreter. The user interacts with the kernel through the shell. You can write ASCII (text) scripts to be acted upon by a shell.
- The shell sits between you and the operating system, acting as a command interpreter.
- It reads your terminal input and translates the commands into actions taken by the system.

- When you log into the system you are given a default shell.
- When the shell starts up it reads its startup files and may set environment variables, command search paths, and command aliases, and executes any commands specified in these files.
- The original shell was the Bourne shell, sh.
- Every Unix platform will either have the Bourne shell, or a Bourne compatible shell available.
- Another popular shell is C Shell. The default prompt for the C shell is %.

- You can write shell programs by creating scripts containing a series of shell commands.
- The first line of the script should start with `#!` which indicates to the kernel that the script is directly executable.
- You immediately follow this with the name of the shell, or program (spaces are allowed), to execute, using the full path name. So to set up a Bourne shell script the first line would be: `#! /bin/sh`
- The first line is followed by commands
- Within the scripts `#` indicates a comment from that point until the end of the line, with `#!` being a special case if found as the first characters of the file.

`#!/bin/bash`

▣ You also need to specify that the script is executable by setting the proper bits on the file with `chmod`, e.g.:

```
$ chmod +x shell_script
```

Command Structure:

Command <Options> <Arguments>

- Multiple commands separated by `;` can be executed one after the other

The Shell as a Programming Language

Creating a Script

- To create a shell script first use a text editor to create a file containing the commands.
- For example, type the following commands and save them as first.sh
- `#!/bin/sh`
 - is special and tells the system to use the /bin/sh program to execute this program.

```
#!/bin/sh

# first.sh
# This file looks through all the files in the current
# directory for the string POSIX, and then prints those
# files to the standard output.

for file in *
do
    if grep -q POSIX $file
    then
        more $file
    fi
done

exit 0
```

The command **exit 0**

➤ Causes the script program to exit and return a value of 0, which means there were not errors.

Making a Script Executable

- ✓ There are two ways to execute the script.
- ✓ 1) invoke the shell with the name of the script file as a parameter, thus:

`/bin/sh first.sh` (Or)

- 2) change the mode of the script to executable and then after execute it by just typing its name.

- ✓ `chmod +x first.sh first.sh`
- ✓ Actually, you may need to type:
- ✓ `./first.sh`
- ✓ to make the file execute unless the path variable has your directory in it.

Shell Metacharacters

1. File name substitution.
2. I/O Redirection.
3. Process execution.
4. Quoting Metacharacters.
5. Positional parameters.
6. Special characters.

File name substitution (?, *, [..],[!..])

❑ These metacharacters are used to match the filenames in a directory.

? ◇ stands for any one character

***** ◇ any combination of any number of characters.

[..] ◇ 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.

Examples

\$ ls a* ◇ list of all files beginning with character ‘a’.

\$ ls ?? ◇ list of all those files whose names are two characters long.

\$ ls a?b? ◇ list of all files whose first character is ‘a’ and third character is ‘b’

\$ ls [c-z]* ◇ the range of ‘c’ to ‘z’

\$ ls [!d-m]* ◇ list of all files whose first character is anything other than an in the range ‘d’ to ‘m’.
alphabet

I/O Redirection(>, <, >>)

$M > \&N$ → Merges the standard output and standard error if $m=1$ and $n=2$.

Here m and n denote file descriptors. They can take the values 0,1,2, representing standard input and standard output and standard error respectively.

Example

```
$ ls > myfile 2 > &1
```

o/p : consists o/p of `ls` in file `myfile`.

The o/p of `time` is always sent on the standard error.

Process execution (;,(),&,&&,||)

;
→ To run more than one command in onestroke.

()
→ To run commands in sub shell.

&
→ Runs the process in the background.

&&
→ To execute the second command if the first succeeds.

||
→ To carry out the second command if first fails.

Examples

```
$ ls;who;time
```

```
$ (cd mydir;pwd)
```

```
$sort abcd > abcd2 &
```

```
$grep “supplement” paper > newspaper    && cat paper
```

```
$ grep “sachin” addresses || grep “sachin” adrfile &&  
cat afile
```

What if the first grep is successful in || ?

The second command is skipped and the next command is executed.

Quoting Metacharacters (\, ” “, ‘ ‘, ` `)

\ → to remove shell dilemma, we \ before any Metacharacters.

` ` → Back quotes replace command with its o/p.

‘ ‘ → To take every enclosed character literally.

“ “ → Allows to hold special status of Metacharacters.

EXAMPLES

\$ echo Hello; Word

Hello

\$ echo Hello\; Word

Hello; Word

- The \$ sign is one of the meta characters, so it must be quoted to avoid special handling by the shell

\$ echo "I have \\$1200"

I have \$1200

- Backslash tells shell to ignore next character.

\$ echo \$HOSTNAME

Mamatha-Mamatha

\$ echo \\$HOSTNAME

\$HOSTNAME

1. Anything which is enclosed between single and double quote can be treated as single string. They are useful ,if you need to refer to a file that has spaces in its name.

```
$ touch "first script"
```

```
$ ls -l first script
```

No such file or directory

```
$ ls -l "first script"
```

```
-rw-r--r 1 mamatha mamatha .....
```

2. Both suppress the meaning of **wild-cards**

```
$ echo *.c
```

```
ab.c  anc.c  ws.c f1.c f2.c
```

```
echo "*.c"
```

```
*.c
```

```
$ echo '*.c'
```

- 3.** Command Substitution can be performed using double quotes i.e Back quote can be used within double quote

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

```
Today date is  Fri Jan 7 18:22:22 IST 2018
```

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

```
Today date is `date`
```



4. The basic difference between single quote and double quote is that single quote does not perform any kind of substitution i.e that is what ever written inside single quote is simply a string whereas double quote allow substitution.

```
$ a=5
```

```
$ echo "value of a is $a"
```

```
5
```

```
$ echo 'value of a is $a'
```

```
value of a is $a
```

# EXAMPLES

- # | Expression | Result | Comment
- 1 | "\$a" | apple | variables are expanded inside ""
- 2 | '\$a' | \$a | variables are not expanded inside "
- 3 | ""\$a"" | 'apple' | " has no special meaning inside ""
- 4 | ""\$a"" | "\$a" | "" is treated literally inside "
- 5 | '\" | \*\*invalid\*\* | can not escape a ' within "; use "" or \$'\'' (ANSI-C quoting)
- 6 | "red\$arocks" | red | \$arocks does not expand \$a; use \${a}rocks to preserve \$a
- 7 | "redapple\$" | redapple\$ | \$ followed by no variable name evaluates to \$
- 8 | '\" | \" | \ has no special meaning inside "
- 9 | "\" | ' | \' is interpreted inside ""
- 10 | "\" | " | \" is interpreted inside ""

# EXAMPLES

```
$ echo this is a *
```

```
$ echo this is a *
```

```
$ echo '$,\,?, all just the way they look'
```

```
$ echo "your name is $name"
```

```
$ echo today is `date`
```

# Shell Variables

- ❖ Variables are generally created when you first use them.
- ❖ By default, all variables are considered and stored as strings.
- ❖ Variable names are case sensitive
- ❖ No type declaration is necessary before u can use a shell variable.
- ❖ Variables provide the ability to store and manipulate the information with in the shell program. The variables are completely under the control of user.

- ❖ You can define and use variables both in the command line and shell scripts. These variables are called shell variables.

Generalized form:

**variable=value**

Eg:

```
$ x=10
```

```
$ echo $x
```

```
10
```

- ❖ To remove a variable use unset.

**\$ unset x**

❖ All shell variables are initialized to null strings by default.

❖ To explicitly set null values use

**x= or x='' or x=""**

❖ To assign multiword strings to a variable use

**\$ msg='u have a mail'**

❖ To make variable global use **export**

**Ex: \$ echo \$0**

**bash**

**\$ x=10**

**\$ echo &x**

**10**

**\$ ksh**

**\$ echo \$0**

**ksh**

**\$ echo &x**

**\$ export x**

# Environment Variables

- They are initialized when the shell script starts and normally capitalized to distinguish them from user-defined variables in scripts
- Environmental variables control the user environment.
- Environmental variables are set by the environment, the operating system, during the startup of your shell.
- Environment variables are dynamic values which affect the processes or programs on a computer. They exist in every operating system, but types may vary.

- Reading environment variables same as shell variables

For example :

- An environment variable named HOME, you can access the value of this variable by using the dollar(\$) sign, for example \$HOME
- The environmental variables are in uppercase.



| <b>Environment Variables</b> | <b>Description</b>                                |
|------------------------------|---------------------------------------------------|
| <b>\$HOME</b>                | <b>Home directory</b>                             |
| <b>\$PATH</b>                | <b>List of directories to search for commands</b> |
| <b>\$PS1</b>                 | <b>Command prompt</b>                             |
| <b>\$PS2</b>                 | <b>Secondary prompt</b>                           |
| <b>\$SHELL</b>               | <b>Current login shell</b>                        |
| <b>\$0</b>                   | <b>Name of the shell script</b>                   |
| <b>\$#</b>                   | <b>No . of parameters passed</b>                  |
| <b>\$\$</b>                  | <b>Process ID of the shell script</b>             |

- `$#` - Stores the number of command-line arguments that were passed to the shell program.
- `$?` - Stores the exit value of the last command that was executed.
- `$*` - Stores all the arguments that were entered on the command line (`$1 $2 ...`).
- `"$@"` - Stores all the arguments that were entered on the command line, individually quoted (`"$1" "$2" ...`).

Example:

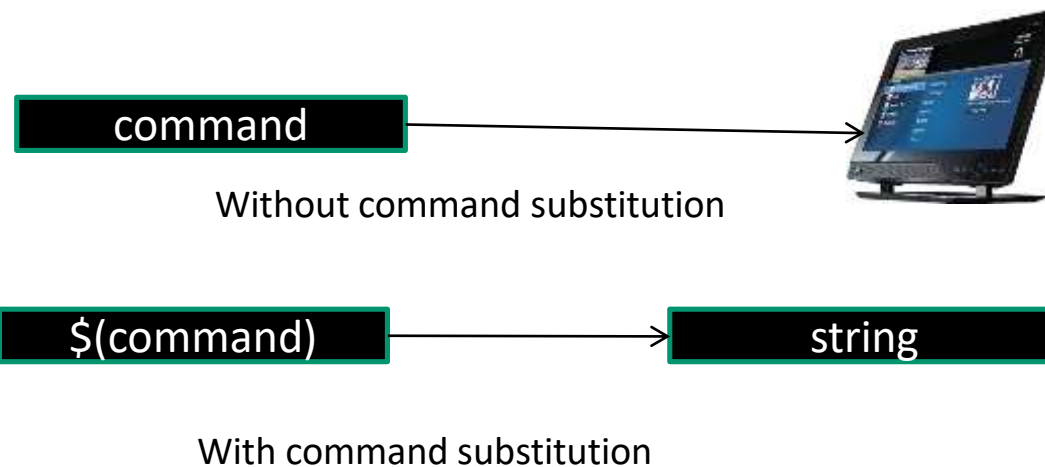
`$ ./command -yes -no /home/username`

- `$# = 3`
- `$* = -yes -no /home/username`
- `$@ = array: {"-yes", "-no", "/home/username"}`

# Command Substitution

- When a Shell executes a command, the output is directed to standard output( most of the time, standard output is associated with monitor)
- Shell allows the standard output of one command to be used as an argument of another command
- Some times we need to change the output to a string that we can store in another string or a variable .
- Command substitutions provides the capability to convert the result of a command to a string

- The command substitution operator that converts the output of a command to a string is a dollar sign and a set of parentheses.
- Backquote ( ` ` ) or backtick is another metacharacter that used for command substitution
- When scanning the command line, the shell executes the enclosed command and replaces the enclosed command line with the out put of the command



- ✓ You can use this feature to generate meaningful messages

Ex:

```
$ echo “ There are `ls | wc -l` file in the current directory”
```

There are 58 files in the current directory

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

The date Today is Sat Jan 7 19:01:18 IST 2002

- ✓ When the single quote is used:

```
$ echo ‘There are `ls | wc -l` file in the current directory’
```

There are `ls |wc -l` files in the current directory

# Shell Commands

- The commands used inside the shell for programming tasks are known as the shell commands, those are:

## 1) **expr command**

- The expr utility is used to evaluate arithmetic operations, as shell doesn't support arithmetic operators.
- Expression will be evaluated by expr utility and the result is sent to the standard output.
- Special symbols such as \*,>,< must be preceded by a \, otherwise the shell treats it as a meta-character and yields wrong results

*Example:* \$ i = 10 ; j = 10

\$ echo `expr \$i + \$j`

20

\$ echo `expr \$i \\* \$j`

100

## 2) `who | sort`

- `who` command displays who logged onto the system and provides an account of all users
- It displays a three column output, the first column displays the user-ids of users currently working on the system.
- The second column displays the system name.
- The third column displays the date and time.
- `Sort` command sorts text files. It sorts all the lines from the standard input.
- `who | sort` before displaying the `who` output on the screen it is piped to `sort`.
- `Sort` receives the output of `who` as standard input, it then sorts the output according to the first alphabet in each line and displays it on screen



```
user@myhost:~$ who
```

```
fred pts/0 2015-05-16 15:59 (:0.0)
```

```
fred pts/1 2015-05-16 16:01 (:0.0)
```

```
mary pts/2 2015-05-17 10:18 (:0.0)
```

### 3) `ls | wc -l`

- `ls` command lists all the files and directories on the standard output.
- `wc -l` command counts the number of files it receives as standard input
- `ls` command is piped to `wc -l` command where the standard output of `ls` is given as standard input to `wc` command.
- Thus, it displays the count of files and directories on the standard output

## break

- Break statement causes the control to come out of the loop instantly.
- It terminates the loop.

```
break.sh
```

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

```
x = 5
```

```
while[$x -gt 3] #outer while loop
```

```
do y = 5
```

```
while[$y -gt 2] #inner while loop
```

```
do
```

```
if[$y -eq 3]
```

```
then
```

```
 break; #immediately terminates inner loop when y =3
```

```
else
```

```
 echo $x $y
```

```
fi
```

```
y = `expr $y - 1`
```

```
done
```

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

```
done
```

## 5) tee command

- The tee command copies standard input to standard output and at the same time copies it to one or more files.
- If the stream is coming from another command, such as who, it can be piped to the tee command.

*The tee command*

*tee*

|                       |
|-----------------------|
| Options<br>-a :append |
|-----------------------|

*file-list*

*Example:*

\$The following command (with the help of tee command) writes the output both to the screen (stdout) and to the file.

**\$ ls | tee file**

➤ By default tee command overwrites the file. You can instruct tee command to append to the file using the option **-a** as shown below.

```
$ ls | tee -a file
```

➤ You can also write the output to multiple files as shown below.

```
$ ls | tee file1 file2 file3
```

## 6) Aliases

- An alias provides a means of creating customized commands by assigning a name to a command.
- An alias is created by using the alias command. Its format is **alias name=command-definition**

Example :

Using alias to rename the list command

```
$ alias dir=ls
```

```
$ dir
```

alias of command with options

```
$ alias dir='ls -l'
```

```
$ dir
```

*Listing aliases*

```
$ alias
```

*Removing aliases*

```
$ alias dir
```

```
$ unalias dir
```

*Removing all aliases*

```
$ unalias -a
```

## 7) eval command

- The eval command is used when the Bourne shell needs to evaluate a command twice before executing it.

### Example 1:

If we need to store the name of a variable in a second variable and then use the print command to display the value of the original variable.

#### Wrong way to use a variable in a variable

```
$x=23
$y=x
$print $y
x
```

#### Another wrong way to use a variable in a variable

```
$x=23
$y=x
$print $$y
 $x
```

#### Correct way to use a variable in a variable

```
$ eval print $$y
```

**Example-2:** To execute that command stored in the string

`/home/mamatha > a="ls | more"`

`/home/mamatha > $a`

bash: command not found: ls | more

`/home/user1 > #` Above command didn't work as ls tried to list file with name pipe (|) and more. But these files are not there

`/home/mamatha > eval $a`

file.txt

mailids

remote\_cmd.sh

sample.txt

Tmp

`/home/mamatha >`



**Example-3:**To print the value of variable which is again variable with value assigned to it

```
$ a=10
```

```
$ b=a
```

```
$ c='$'$b (note: The dollar sign must be escaped with '$')
```

```
$ echo $c
```

*output:*

```
$a
```

```
$ eval c='$'$b
```

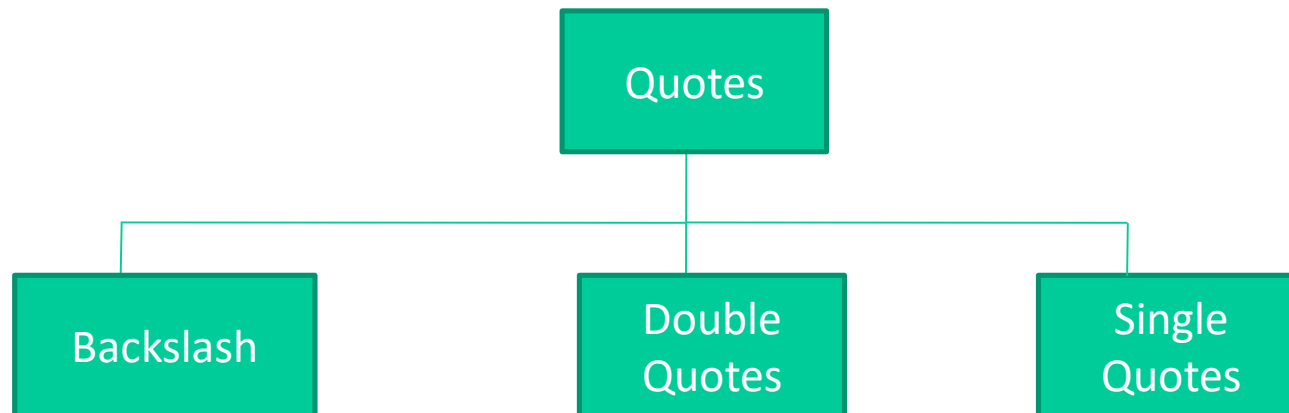
```
$ echo $c
```

*output:*

```
10
```

# Quoting

- Shell uses a selected set of metacharacters in commands.
- Metacharacters are characters that have a special interpretation.  
For example the pipe (|)
- In order to use them as a normal text we must tell the shell interpreter that they should be interpreted differently. This is called quoting the metacharacters.
- To achieve quoting there are three metacharacters collectively as quotes,



## Backslash

- The backslash metacharacter (\) is used to change the interpretation of the character that follows it, i.e., it converts a literal character into a special character and vice versa.

Example:

- The character n is interpreted as a literal character by the shell to change its interpretation as a newline character we use backslash before it (\n).
- Similarly the use of a greater than symbol (>) in a command is interpreted as a special character (output redirection), to change its interpretation as a literal text we use a backslash before it (\>)

```
$echo My name is - \> Ramesh\n
My name is ->Ramesh
```

## Double Quotes

- Double quotes (“”) are used to change the meaning of several characters.
- They change the special interpretation of most metacharacters like <, >, ?, & and so on.
- These metacharacters are treated as literal characters.

Example:

```
$ echo “Metacharacters are : >,<,?,|,&“
Metacharacters are : >,<,?,|,&
```

- Double quotes cannot remove the special interpretation of a dollar sign in front of a variable and single quotes.

```
$ echo a = 10
```

```
$ echo “a is $a and single quote is ‘b’ “
a is 10 and single quote is ‘b’
```

# Single Quotes

- Like double quotes, single quotes change the special interpretation of metacharacters.
- It not only treats metacharacters like `>`, `<`, `?`, `$` and so on as literals but also the metacharacters dollar sign (`$`) and every double quotes.

Example:

```
$ echo a = 10
```

```
$ echo 'characters are < > "b" $a ? &'
```

```
characters are < > "b" $a ? &
```

# Control Structures

- Control structures alter the flow of the program
- Control Structures are:
  1. if-then-else
  2. case
  3. loops
    - a. for
    - b. while
    - c. Until
  4. Handling signals

- **THE SIMPLE IF STATEMENT**  
**SYNTAX:**

**if [ condition ]; then**  
**statements**  
**Fi**

- Executes the statements only if **condition** is true

Ex:

```
#!/bin/bash
Basic if statement
if [$1 -gt 100]
then
echo Hey that\'s a large number.
pwd
fi
Date
```

Output: `./ if_example.sh 15`

Sat 6 Jan 5:06:35 2018

`$ ./if_example.sh 150`

- Hey that's a large number.
- /home/ryan/bin
- Sat 6 Jan 5:06:35 2018



# **• THE IF-THEN-ELSE STATEMENT**

```
if [condition]; then
 statements-1
else
 statements-2
fi
```

- executes statements-1 if condition is true
- executes statements-2 if condition is false

- **THE NESTED IF**

```
if [condition]; then
 statements
elif [condition]; then
 statement
else
 statements
fi
```

- The word **elif** stands for “else if”
- It is part of the if statement and cannot be used by itself

## **primary**

## **Meaning**

|             |                                                      |
|-------------|------------------------------------------------------|
| [ -a FILE ] | True if FILE exists.                                 |
| [ -b FILE ] | True if FILE exists and is a block-special file.     |
| [ -c FILE ] | True if FILE exists and is a character-special file. |
| [ -d FILE ] | True if FILE exists and is a directory.              |
| [ -e FILE ] | True if FILE exists.                                 |
| [ -f FILE ] | True if FILE exists and is a regular file.           |
| [ -g FILE ] | True if FILE exists and its SGID bit is set.         |
| [ -h FILE ] | True if FILE exists and is a symbolic link.          |
| [ -k FILE ] | True if FILE exists and its sticky bit is set.       |

|             |                                                                   |
|-------------|-------------------------------------------------------------------|
| [ -r FILE ] | True if FILE exists and is readable.                              |
| [ -s FILE ] | True if FILE exists and has a size greater than zero.             |
| [ -t FD ]   | True if file descriptor FD is open and refers to a terminal.      |
| [ -u FILE ] | True if FILE exists and its SUID (set user ID) bit is set.        |
| [ -w FILE ] | True if FILE exists and is writable.                              |
| [ -x FILE ] | True if FILE exists and is executable.                            |
| [ -O FILE ] | True if FILE exists and is owned by the effective user ID.        |
| [ -G FILE ] | True if FILE exists and is owned by the effective group ID.       |
| [ -L FILE ] | True if FILE exists and is a symbolic link.                       |
| [ -N FILE ] | True if FILE exists and has been modified since it was last read. |
| [ -S FILE ] | True if FILE exists and is a socket.                              |

[ FILE1 -nt FILE2 ]

True if FILE1 has been changed more recently than FILE2, or if FILE1 exists and FILE2 does not.

[ FILE1 -ot FILE2 ]

True if FILE1 is older than FILE2, or if FILE2 exists and FILE1 does not.

[ FILE1 -ef FILE2 ]

True if FILE1 and FILE2 refer to the same device and inode numbers.

[ -o OPTIONNAME ]

True if shell option "OPTIONNAME" is enabled.

[ -z STRING ]

True if the length of "STRING" is zero.

[ -n STRING ] or [ STRING ]

True if the length of "STRING" is non-zero.

[ STRING1 == STRING2 ]

True if the strings are equal. "=" may be used instead of "==" for strict POSIX compliance.

[ STRING1 != STRING2 ]    True if the strings are not equal.

[ STRING1 < STRING2 ]    True if "STRING1" sorts before "STRING2" lexicographically in the current locale.

[ STRING1 > STRING2 ]    True if "STRING1" sorts after "STRING2" lexicographically in the current locale.

[ ARG1 OP ARG2 ]    "OP" is one of -eq, -ne, -lt, -le, -gt or -ge. These arithmetic binary operators return true if "ARG1" is equal to, not equal to, less than, less than or equal to, greater than, or greater than or equal to "ARG2", respectively. "ARG1" and "ARG2" are integers.

## Combining expressions

### Operation

### Effect

[ ! **EXPR** ]

True if **EXPR** is false.

[ ( **EXPR** ) ]

Returns the value of **EXPR**. This may be used to override the normal precedence of operators.

[ **EXPR1** -a **EXPR2** ]

True if both **EXPR1** and **EXPR2** are true.

[ **EXPR1** -o **EXPR2** ]

True if either **EXPR1** or **EXPR2** is true.

# THE CASE STATEMENT

- use the case statement for a decision that is based on multiple choices

Syntax:

```
case word in
 pattern1) command-list1
 ::
 pattern2) command-list2
 ::
 patternN) command-listN
 ::
esac
```

*case pattern*

- checked against word for match

may also contain:

**\***

**?**



# THE WHILE LOOP

- Purpose:

To execute commands in “command-list” as long as “expression” evaluates to true

Syntax:

```
while [expression]
do
 command-list
done
```

# THE UNTIL LOOP

- Purpose:

To execute commands in “command-list” as long as “expression” evaluates to false

Syntax:

```
until [expression]
do
 command-list
done
```

## The while loop vs the until loop

- The until loop executes until a nonzero status is returned.
- The while command executes until a zero status is returned.
- The until loop always executes at least once

### Example

Create a shell script called f1.sh:

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

```
i=1
```

```
until [$i -gt 6]
```

```
do
```

```
echo "Welcome $i times."
```

```
i=$((i+1))
```

```
done
```

➤ save and close the file. Run it as follows:

```
$ chmod +x until.sh
```

```
$./until.sh
```

outputs:

Welcome 1 times.

Welcome 2 times.

Welcome 3 times.

Welcome 4 times.

Welcome 5 times.

# THE FOR LOOP

- Purpose:

To execute commands as many times as the number of words in the “argument-list”

Syntax:

```
for variable in argument-list
do
 commands
done
```

Example :

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

```
for var in 0 1 2 3 4 5 6 7 8 9
```

```
Do
```

```
 echo $var
```

```
Done
```

Upon execution, you will receive the following result –

0

1

2

3

4

5

6

7

8

9

## Handling signals

- Unix allows you to send a signal to any process
- -1 = hangup                    **kill -HUP 1234**
- -2 = interrupt with ^C    **kill -2 1235**
- no argument = terminate    **kill 1235**
- -9 = kill                    **kill -9 1236**
  - -9 cannot be blocked
- list your processes with
  - **ps -u userid**

| Number | SIG     | Meaning                     |
|--------|---------|-----------------------------|
| 0      | 0       | On exit from shell          |
| 1      | SIGHUP  | Clean tidyup                |
| 2      | SIGINT  | Interrupt                   |
| 3      | SIGQUIT | Quit                        |
| 6      | SIGABRT | Abort                       |
| 9      | SIGKILL | Die Now (cannot be trapped) |
| 14     | SIGALRM | Alarm Clock                 |
| 15     | SIGTERM | Terminate                   |



# Signals on Linux

**% kill -l**

1) SIGHUP    2) SIGINT    3) SIGQUIT    4) SIGILL  
5) SIGTRAP    6) SIGABRT    7) SIGBUS    8) SIGFPE  
9) SIGKILL    10) SIGUSR1    11) SIGSEGV    12) SIGUSR2  
13) SIGPIPE    14) SIGALRM    15) SIGTERM    16) SIGSTKFLT  
17) SIGCHLD    18) SIGCONT    19) SIGSTOP    20) SIGTSTP  
21) SIGTTIN    22) SIGTTOU    23) SIGURG    24) SIGXCPU  
25) SIGXFSZ    26) SIGVTALRM    27) SIGPROF    28) SIGWINCH  
29) SIGIO    30) SIGPWR    31) SIGSYS    34) SIGRTMIN  
35) SIGRTMIN+1    36) SIGRTMIN+2    37) SIGRTMIN+3    38) SIGRTMIN+4  
39) SIGRTMIN+5    40) SIGRTMIN+6    41) SIGRTMIN+7    42) SIGRTMIN+8  
43) SIGRTMIN+9    44) SIGRTMIN+10    45) SIGRTMIN+11    46) SIGRTMIN+12  
47) SIGRTMIN+13    48) SIGRTMIN+14    49) SIGRTMIN+15    50) SIGRTMAX-14  
51) SIGRTMAX-13    52) SIGRTMAX-12    53) SIGRTMAX-11    54) SIGRTMAX-10  
55) SIGRTMAX-9    56) SIGRTMAX-8    57) SIGRTMAX-7    58) SIGRTMAX-6  
59) SIGRTMAX-5    60) SIGRTMAX-4    61) SIGRTMAX-3    62) SIGRTMAX-2  
63) SIGRTMAX-1    64) SIGRTMAX

^C is 2 - SIGINT

# Handling signals

- Default action for most signals is to end process  
term: signal handler
- Bash allows to install custom signal handler
- When you press the *Ctrl+C* or Break key at your terminal during execution of a shell program, normally that program is immediately terminated, and your command prompt returns. This may not always be desirable. For instance, you may end up leaving a bunch of temporary files that won't get cleaned up.

## Syntax:

**trap 'handler commands' signals**

## Example:

**trap 'echo do not hangup' 1 2**

- Two common uses for trap in shell scripts

➤ Clean up temporary files

Ex: \$ trap "rm -f \$WORKDIR/f1 WORKDIR/f2; exit" 2

➤ Ignore signals

Ex: \$ trap " 2

# Arithmetic in Shell

## Shell Arithmetic

- Use to perform arithmetic operations.

*Syntax:*

**expr op1 math-operator op2**

*Examples:*

```
$ expr 1 + 3
```

```
$ expr 2 - 1
```

```
$ expr 10 / 2
```

```
$ expr 20 % 3
```

```
$ expr 10 * 3
```

```
$ echo `expr 6 + 3`
```

**Note:**

expr 20 % 3 - Remainder read as 20 mod 3 and remainder is 2.

expr 10 \\* 3 - Multiplication use \\* and not \* since its wild card.

## Shell Arithmetic

For the last statement note the following points.

- (1) First, before `expr` keyword we used ``` (back quote) sign not the (single quote i.e. `'`) sign. Back quote is generally found on the key under tilde (`~`) on PC keyboard OR to the above of TAB key.
- (2) Second, `expr` is also end with ``` i.e. back quote.
- (3) Here `expr 6 + 3` is evaluated to 9, then `echo` command prints 9 as sum
- (4) Here if you use double quote or single quote, it will NOT work

For e.g.

```
$ echo "expr 6 + 3" # It will print expr 6 + 3
```

```
$ echo 'expr 6 + 3' # It will print expr 6 + 3
```

# TEST COMMAND

- Checks file types and compares values.
- **test** is used as part of the conditional execution of shell commands.
- **test** exits with the status determined by **EXPRESSION**. Placing the **EXPRESSION** between square brackets ([ and ]) is the same as testing the **EXPRESSION** with **test**.
- To see the exit status at the command prompt, echo the value "\$?". A value of 0 means the expression evaluated as true, and a value of 1 means the expression evaluated as false.

## Syntax:

test *EXPRESSION*

(or)

[ *EXPRESSION* ]

## Examples:

```
test 100 -gt 99 && echo "Yes, that's true." || echo "No, that's false."
```

This command will print the text "**Yes, that's true.**" because **100** is greater than **99**.

```
test 100 -lt 99 && echo "Yes." || echo "No."
```

This command will print the text "**No.**" because **100** is not less than **99**.

```
["awesome" = "awesome"];
```

```
echo $?
```

This command will print "0" because the expression is true; the two strings are identical.

```
[5 -eq 6];
```

```
echo $?
```

This command will print "1" because the expression is false; 5 does not equal 6.



## Comparing Numbers:

If you are comparing elements that parse as numbers you can use the following comparison operators:

- eq - does value 1 equal value 2
- ge - is value 1 greater or equal to value 2
- gt - is value 1 greater than value 2
- le - is value 1 less than or equal to value 2
- lt - is value 1 less than value 2
- ne - does value 1 not equal value 2

## Comparing Text:

If you are comparing elements that parse as strings you can use the following comparison operators:

**=** - does string 1 match string 2

**!=** - is string 1 different to string 2

**-n** - is the string length greater than 0

**-z** - is the string length 0

## Examples:

```
test "string1" = "string2" && echo "yes" || echo "no"
```

(displays "no" to the screen because "string1" does not equal "string2")

# Functions

- A shell function is similar to a shell script
  - stores a series of commands for execution later
  - shell stores functions in memory
  - shell executes a shell function in the same shell that called it
- Where to define
  - In .profile
  - In your script
  - Or on the command line
- Remove a function
  - Use unset built-in

- must be defined before they can be referenced
- usually placed at the beginning of the script

Syntax:

```
function-name () {
 statements
}
```

# Function parameters

- Need not be declared
- Arguments provided via function call are accessible inside function as \$1, \$2, \$3, ...
- \$# reflects number of parameters
- \$0 still contains name of script  
(not name of function)

## **Local Variables in Functions**

- Variables defined within functions are global, i.e. their values are known throughout the entire shell program
- Keyword “local” inside a function definition makes referenced variables “local” to that function

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

```
Define your function here
```

```
Hello ()
```

```
{ echo "Hello World $1 $2"
```

```
 return 10 }
```

```
Invoke your function
```

```
Hello Zara Ali
```

```
Capture value returned by last command
```

```
ret=$?
```

```
echo "Return value is $ret"
```

**Output:**

```
$/test.sh
```

```
Hello World Zara Ali
```

```
Return value is 10
```

```
#!/bin/sh
myfunc()
{
echo "I was called as : $@"
x=2 }

Main script starts here
echo "Script was called with $@"
x=1

echo "x is $x"
myfunc 1 2 3
echo "x is $x"
```

Output:

**\$ scope.sh a b c**

Script was called with a b c

x is 1

I was called as : 1 2 3



# Debugging Shell Scripts

- Debugging is troubleshooting errors that may occur during the execution of a program/script
- The following two commands can help you debug a bash shell script:
  - echo  
use explicit output statements to trace execution
  - set

## *Debugging using “set”*

- The “set” command is a shell built-in command
- has options to allow flow of execution
  - v option prints each line as it is read
  - x option displays the command and its arguments
  - n checks for syntax errors
- options can be turned on or off
  - To turn on the option: `set -xv`
  - To turn off the options: `set +xv`
- Options can also be set via she-bang line  
**`#!/bin/bash -xv`**

(or)

## **x option to debug a shell script**

Run a shell script with -x option.

```
$ bash -x script-name
```

```
$ bash -x domains.sh
```

(or)

➤ The **-x** option, short for **xtrace** or **execution trace**, tells the shell to echo each command after performing the substitution steps. Thus, we can see the values of variables and commands. Often, this option alone will help to diagnose a problem.

Ex:

- We can use `set -x` command in shell script itself:

*#!/bin/bash*

**clear**

*# turn on debug mode*

**set -x**

**for f in \***

**do**

**file \$f**

**done**

*# turn OFF debug mode*

**set +x**

**ls** *# more commands*

➤The **-v option** tells the shell to run in **verbose mode**. In practice , this means that shell will echo each command prior to execute the command. This is very useful in that it can often help to find the errors.

Ex:

```
snist:~/mamta> set -v
```

```
snist:~/mamta> ls
```

```
ls
```

```
commented-scripts.sh script1.sh
```

```
snist:~/mamta> set +v
```

```
set +v
```

```
snist:~/mamta> ls *
```

```
commented-scripts.sh script1.sh
```

➤ The **-n** option, shot for **noexec** ( as in no execution), tells the shell to not run the commands. Instead, the shell just checks for syntax errors. This option will not convince the shell to perform any more checks. Instead the shell just performs the normal **syntax check**. With **-n** option, the shell doesn't execute your commands, so you have a safe way to test your scripts if they contain syntax erro

Ex: shell script name **debug\_quotes.sh**

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

```
echo "USER=$USER
```

```
echo "HOME=$HOME"
```

```
echo "OSNAME=$OSNAME"
```

**Now run the script with -n option**

```
$ sh -n debug_quotes
```

```
debug_quotes: 8: debug_quotes: Syntax error: Unterminated
quoted string.
```

➤ We can use the debug function for debugging specific statement in the shell script. An example is shown below:

```
#!/bin/bash

_DEBUG="on"

function DEBUG()
{ ["$_DEBUG" == "on"] && $@
}

DEBUG echo 'Printing Numbers'

for i in `seq 1 3`
do
echo $i done
```

Running the script:

```
$ bash script.sh 1 2 3 $ export _DEBUG=on Printing Numbers 1 2 3
```

Thank you