



The Linux Command Line

Ung Văn Giàu
Email: giau.ung@eiu.edu.vn



Starting with Linux Shells

Introduction



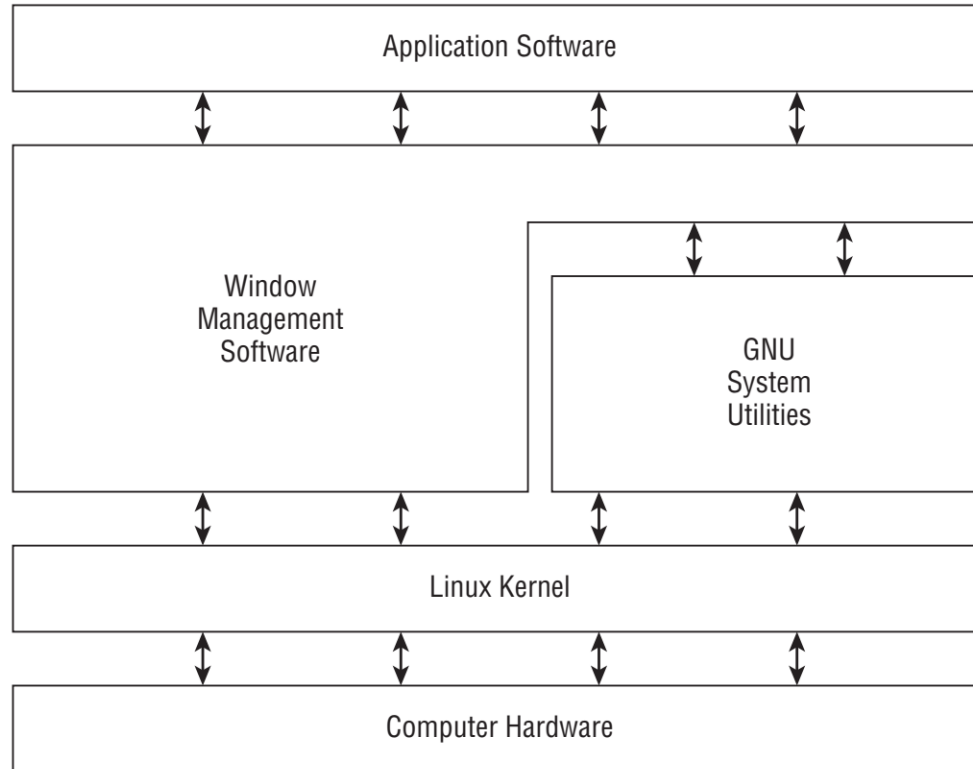
- The Linux OS is **everywhere**

Smartphones, cars, supercomputers, home appliances, home desktops, enterprise servers,...

- Linux is one of the most **reliable, secure and worry-free** OS available

What is Linux?

The Linux System



What is Linux?

4 main parts make up a Linux system:

- The Linux kernel
- The GNU utilities
- A graphical desktop environment
- Application software



What is Linux?

The Linux Kernel

- The **core** of the Linux system
- Controls all the hardware and software
 - allocating hardware
 - executing software
- Primarily responsible for 4 main functions:
 - System memory management
 - Software program management
 - Hardware management
 - Filesystem management

What is Linux?

The GNU Utilities

- Perform standard functions: controlling files and programs
- A complete set of Unix utilities under a software philosophy called open source software (OSS)
- **The Core GNU Utilities**
 - Utilities for handling files
 - Utilities for manipulating text
 - Utilities for managing processes

What is Linux?

The GNU Utilities

▪ The Shell

- A special interactive utility
- Provides a to start programs, manage files on the filesystem, and manage processes running
- The core is the command prompt
- The shell contains a set of internal commands: copying files, moving files, renaming files,...
- **Shell scripts:** group shell commands into files to execute as a program
- The default shell used in all Linux distributions is the **bash shell**

What is Linux?

The Linux Desktop Environment

- In the early days of Linux (the early 1990s) all that was available was a simple text interface
- The more popular graphical desktops:
 - **The X Window System**
 - ✓ The Fedora Linux distribution
 - ✓ The Ubuntu Linux distribution
 - ✓ No desktop environment allows users to manipulate files or launch programs → need a desktop environment on top
 - **The KDE Desktop (1996):** similar to the Microsoft Windows environment
 - **The GNOME Desktop (1999)**
 - ✓ The most popular Linux distribution is Red Hat Linux
 - ✓ Incorporates many features that most Windows users are comfortable

What is Linux?

The Linux Desktop Environment

- The more popular graphical desktops:
 - **The Unity Desktop**
 - ✓ The company responsible for developing Ubuntu, has decided to embark on its own Linux desktop environment, called Unity
 - ✓ Provide a single desktop experience for workstations, tablet devices, and mobile devices
 - **Other Desktops**
Fluxbox, Xfce, JWM, Fvwm,...

What is a “distribution”?

- **Distribution** (distros)
a complete Linux system package
- Linux has many different versions to suit any type of user
new users to hard-core users
- Distribution can be downloaded for free, burned onto disk (or USB thumb drive), and installed

What is a “distribution”?

- There are 3 categories:
 - **Full core Linux distributions**
 - ✓ Contains a kernel, one or more graphical desktop environments, and just about every Linux application that is available, precompiled for the kernel
 - ✓ Red Hat, Fedora, Debian,...
 - **Specialized distributions**
 - ✓ Contain only a subset of applications that would make sense for a specific area of use
 - ✓ CentOS, Ubuntu, Mint,...

What is a “distribution”?

- There are 3 categories:

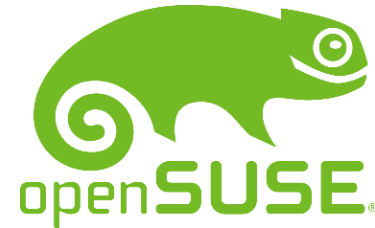
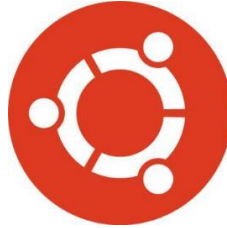
- **LiveCD test distributions**

- ✓ Some Linux distributions create a bootable CD that contains a sample Linux system
- ✓ You can boot your PC from the CD and run a Linux distribution without having to install anything on your hard drive
- ✓ An excellent way to test various Linux distributions
- ✓ Ubuntu, Knoppix, PCLinuxOS,...
- ✓ **Drawbacks:**
 - Applications run more slowly
 - Any changes you make to the Linux system will be gone the next time you reboot

What is a “distribution”?

Popular Linux distributions include:

- UBUNTU
- FEDORA
- LINUX MINT
- DEBIAN
- OPENSUSE
- MANJARO
- ANTERGOS
- SOLUS
- ELEMENTARY OS



What is a “distribution”?

- The server distributions include:

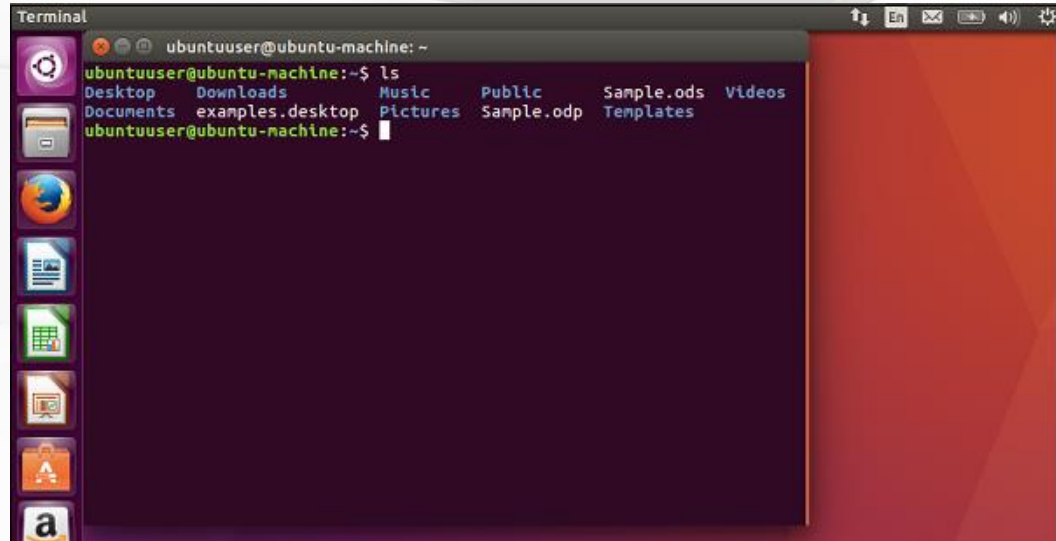
- Ubuntu Server
- Centos
- Red Hat Enterprise Linux
- SUSE Enterprise Linux



- Ubuntu Server and CentOS distributions are free
- Red Hat Enterprise Linux and SUSE Enterprise Linux have an associated price
Those with an associated price also include support

Command Line

- Text **command line interface** (CLI) is provided by the shell
- CLI allows **text input only** and could **display only text** and rudimentary graphics output
- 3 commonly used graphical terminal emulator:
 - GNOME Terminal
 - Konsole Terminal
 - xterm

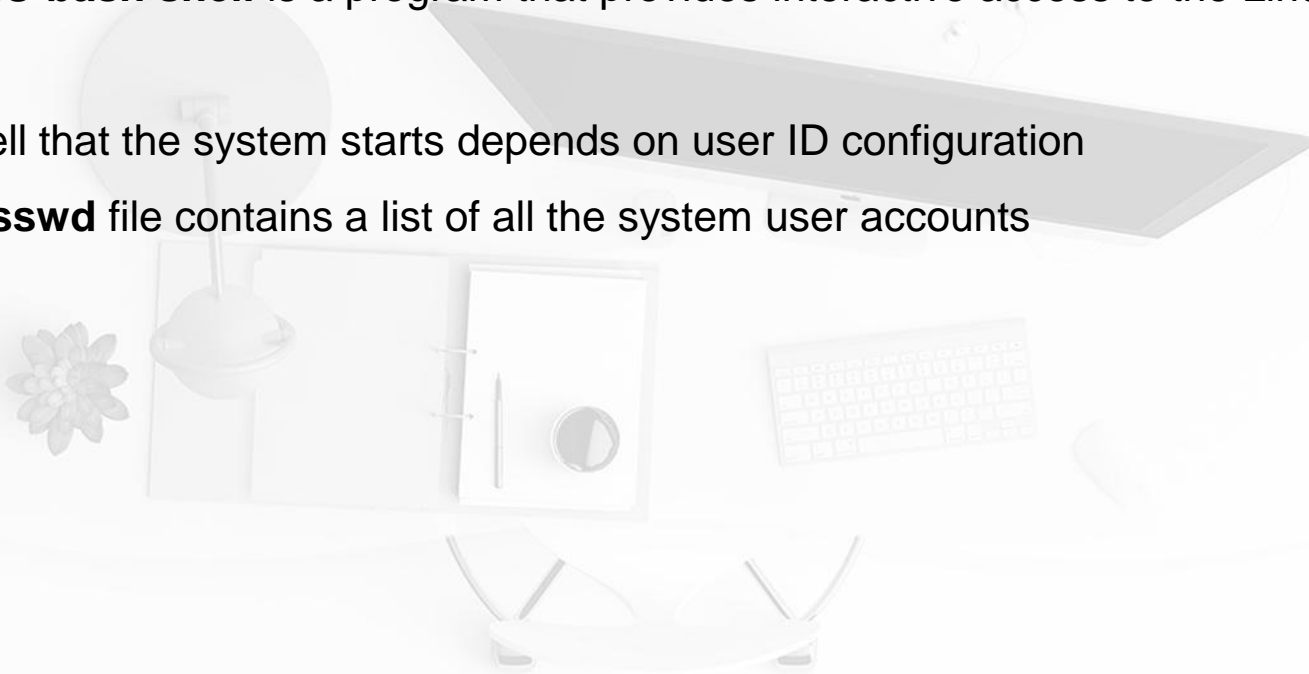




Basic bash Shell Commands

Starting the Shell

- The GNU **bash shell** is a program that provides interactive access to the Linux system
- The shell that the system starts depends on user ID configuration
/etc/passwd file contains a list of all the system user accounts



Using the Shell Prompt

- After you start a terminal, get access to the **shell CLI prompt**

The place where you enter shell commands

- The default prompt symbol is **\$**

This symbol indicates that the shell is waiting for you to enter text

Interacting with the bash Manual

- The **man** command provides access to the manual pages stored on the Linux system
- Entering the man command is **followed by a specific command name**
 - to view through the man pages by pressing the **spacebar**, or go line by line using the **Enter key**
 - press the **q** key to **quit**
 - a command has man pages in multiple section content areas
 - ✓ **man** section# topic
 - ✓ **man** 7 hostname
- Most commands accept the **--help** option

Navigating the Filesystem

Looking at the Linux filesystem

How Linux System references files and directories:

- Does not use drive letters in pathnames
- Linux stores files within a single directory structure, called a **virtual directory**
- The Linux virtual directory structure contains a single base directory, called the **root**
- The first hard drive installed in a Linux system is called the **root drive**
- On the root drive, Linux can use special directories as **mount points**. Mount points are directories in the virtual directory where you can **assign additional storage devices**
- In a Linux filesystem, **common directory names** are used for **common functions**

Navigating the Filesystem

Looking at the Linux filesystem

TABLE 3-3 Common Linux Directory Names

Directory	Usage
/	root of the virtual directory, where normally, no files are placed
/bin	binary directory, where many GNU user-level utilities are stored
/boot	boot directory, where boot files are stored
/dev	device directory, where Linux creates device nodes
/etc	system configuration files directory
/home	home directory, where Linux creates user directories
/lib	library directory, where system and application library files are stored
/media	media directory, a common place for mount points used for removable media
/mnt	mount directory, another common place for mount points used for removable media
/opt	optional directory, often used to store third-party software packages and data files

Navigating the Filesystem

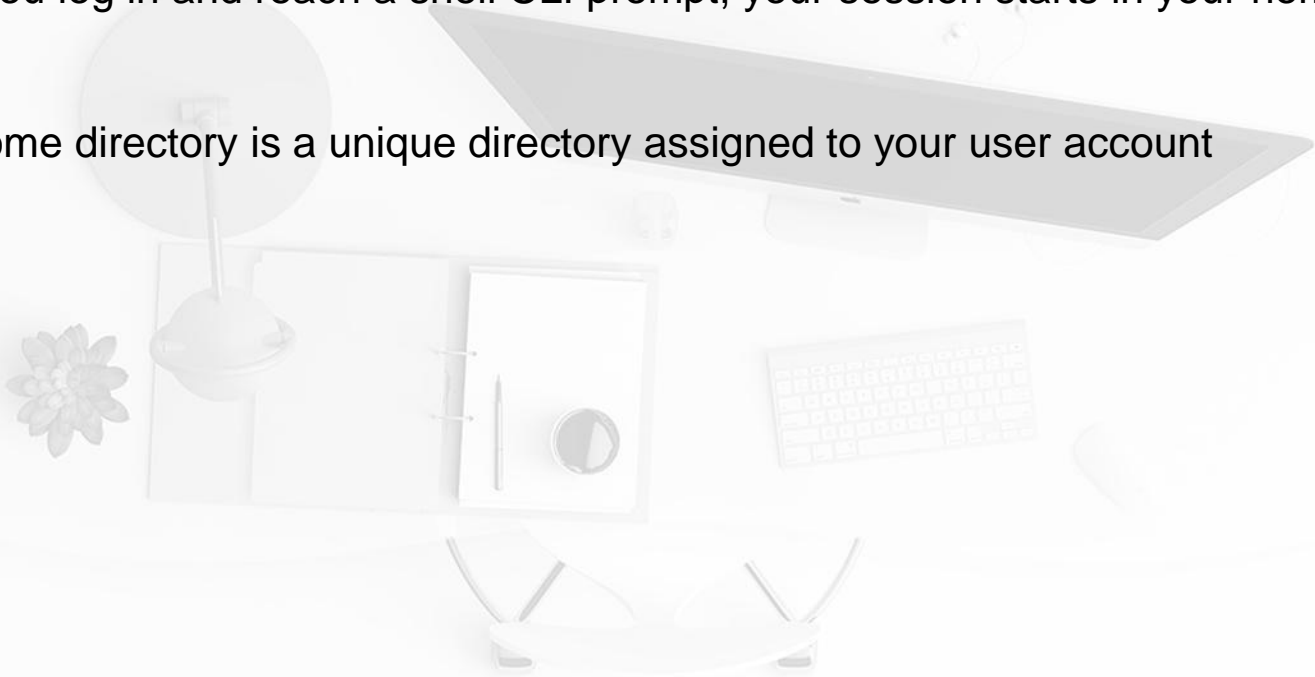
Looking at the Linux filesystem

/proc	process directory, where current hardware and process information is stored
/root	root home directory
/sbin	system binary directory, where many GNU admin-level utilities are stored
/run	run directory, where runtime data is held during system operation
/srv	service directory, where local services store their files
/sys	system directory, where system hardware information files are stored
/tmp	temporary directory, where temporary work files can be created and destroyed
/usr	user binary directory, where the bulk of GNU user-level utilities and data files are stored
/var	variable directory, for files that change frequently, such as log files

Navigating the Filesystem

Looking at the Linux filesystem

- When you log in and reach a shell CLI prompt, your session starts in your home directory
- Your home directory is a unique directory assigned to your user account





Directory and File Management

Directory and File Management

▪ Using absolute directory references

- defines exactly where the directory is, starting at the root
- always begins with a forward slash (/) (= root)
- E.g. `cd /var`

▪ Using relative directory references

- specify a destination directory reference relative to your current location
- starts with:
 - ✓ either a directory name (if traversing to a directory under current directory)
 - ✓ or a special character
- can also use a special character to indicate a relative directory location
 - ✓ The single dot (.) to represent the current directory
 - ✓ The double dot (..) to represent the parent directory (one level)

Directory and File Management

▪ Print working directory

- Displays the shell session's current directory location
- Syntax: **pwd**

▪ List contents

- List the files and folders within a directory
- Syntax: **ls [Options] [directory]**
 - ✓ **-a**: all (include hidden files that are files with filenames starting with a period (.))
 - ✓ **-l**: long list format
- **Note**: `ls -l -a = ls -la`

Directory and File Management

- **Filtering listing output**

- determine which files or directories it should display in the output
- The filter works as a simple **text-matching string**
`ls -l my_script`
- ls also recognizes standard wildcard characters and uses them to match patterns:
 - ✓ A question mark (?) to represent one character
 - ✓ An asterisk (*) to represent any number of characters
- Using the asterisk and question mark in the filter is called **file globbing**

Directory and File Management

- **Change directory**

- Move the shell session to another directory
- Syntax: **cd [directory]**
- E.g.,
 - ✓ **cd ~** = **cd**: navigate to your home directory
 - ✓ **cd /home**
 - ✓ **cd ..**

Directory and File Management

- **Create directories**

- Create new empty directories
- Syntax: **mkdir [Options] folder_name**
 - p**: make parent directories as needed
- E.g.:
 - ✓ `mkdir dir1`
 - ✓ `mkdir dir1 dir2 dir3`
 - ✓ `mkdir /tmp/tutorial`
 - ✓ `mkdir -p /home/giau/test_2/cse454`

Directory and File Management

▪ Create a File

- Syntax 1: **touch file_name**

E.g.: touch file1

- Syntax 2: **echo “Contents” > file_name**

✓ E.g.: echo “Lower case” > a.txt

✓ E.g.: echo “Upper case” >> a.txt

▪ Copy files / directories

- Syntax: **cp [OPTIONS] Source[s] destination**

- Option:

✓ **-i**: prompt before overwrite

✓ **-R or -r**: copy directories recursively

Directory and File Management

▪ Using tab auto-complete

- To start typing a filename or directory name and then **press the tab key** to have the shell complete
- The trick: give the shell enough filename characters so it can distinguish the desired file from other files

Directory and File Management

- **Create links**

- A link is a placeholder in a directory that points to the real location of the file
- Two types:

- ✓ **A symbolic link: In -s Target [Directory]**

- simply a physical file that points to another file somewhere

- ✓ **A hard link: In Target [Directory]**

- creates a separate virtual file that contains information about the original file and where to locate it

Directory and File Management

- **Rename or Move a file / directory**

- Renaming file is called **moving file**
- The **mv** command is available to move both files and directories to another location or a new name
- Syntax: **mv [Options] Source[s] Dest**
- E.g.:
 - ✓ Move: `mv combined.txt dir1`
 - ✓ Rename: `mv backup_combined.txt combined_backup.txt`

Directory and File Management

- **Delete files / directories**

- Syntax: **rm [options] Dest**
- The shell has no recycle bin or trashcan
- Options:
 - ✓ If removing lots of files and don't want to be bothered with the prompt, use the **-f** parameter to force the removal
 - ✓ **-i**: prompt before every removal
 - ✓ **-r, -R**: remove directories and their contents recursively
 - ✓ **-d**: remove empty directories

Directory and File Management

- **View the whole file**

- Syntax: **cat [options] file_name**

- ✓ Option:

- n**: numbers all the lines

- ✓ For large files, the text in the file just quickly scrolls off the display without stopping

- Syntax: **more [option] file_name**

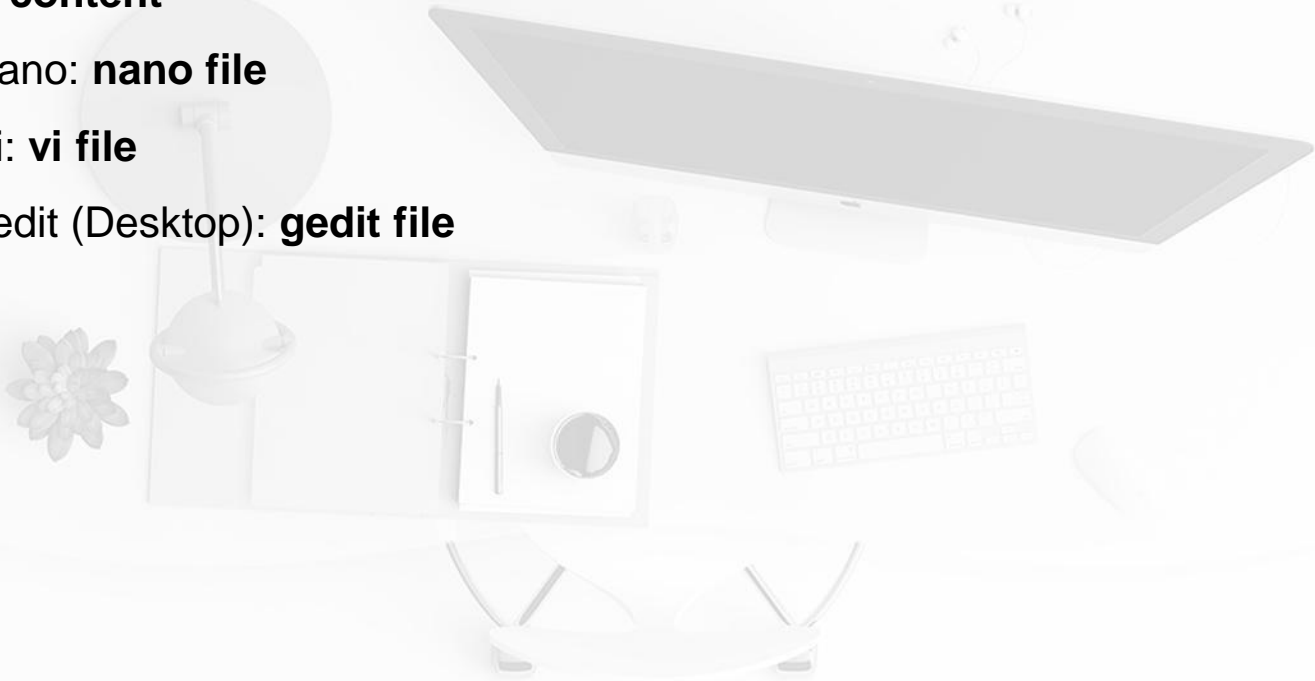
- ✓ Options:

- <number>**: the number of lines per screenful

- +<number>**: display file beginning from line number

Directory and File Management

- **Edit file content**
 - Use Nano: **nano file**
 - Use Vi: **vi file**
 - Use gedit (Desktop): **gedit file**



Nano Editor

- To open an existing file or to create a new file, type:
nano file_name
- At the bottom of the window, there is a list of the most basic command shortcuts
- All commands are prefixed with either **^** or **M** character
 - The caret symbol (^) represents the **Ctrl** key
 - The letter **M** represents the **Alt** key
- For example:
 - Ctrl + o: to save changes
 - Ctrl + x: to exit nano
 - Ctrl + g: to get a list of all commands

Vi Editor

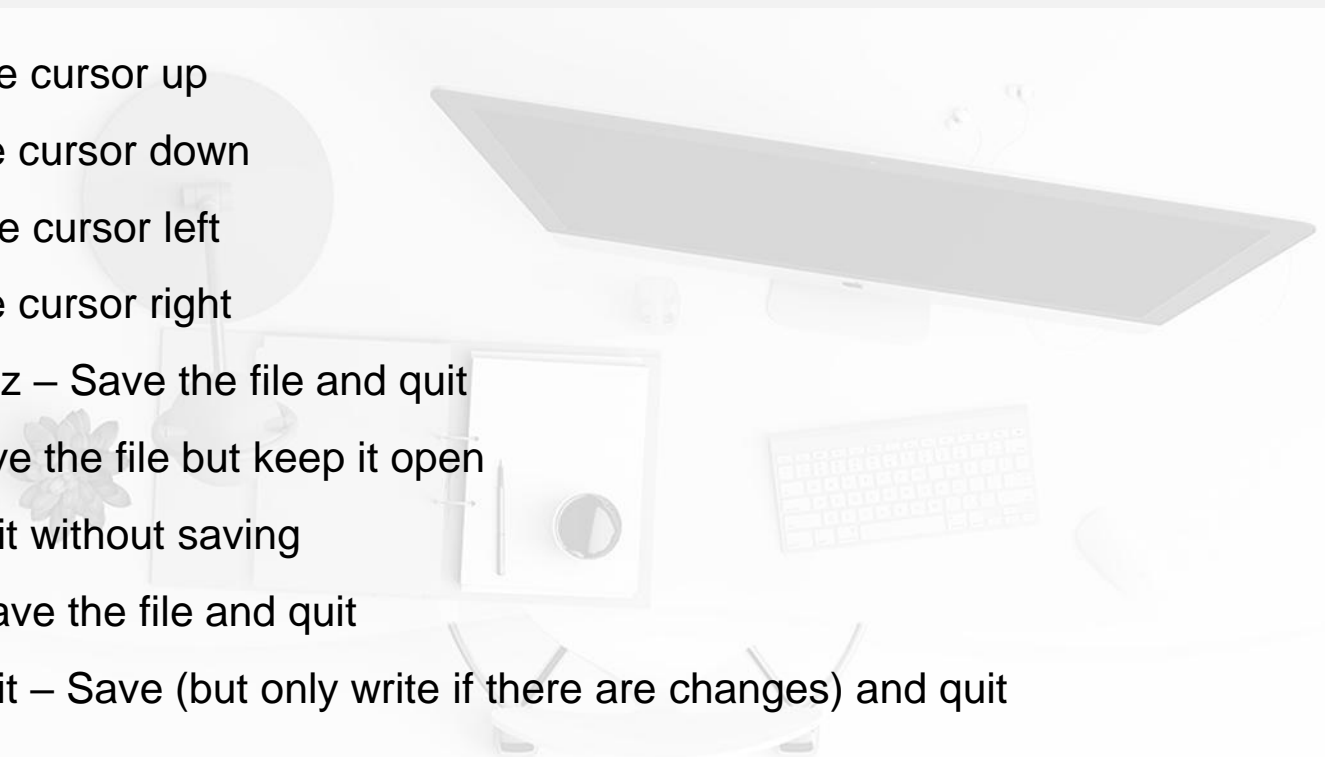
- The **vi** (visual) editor has **two modes of operation**:
 - **Command mode**: enables you to perform administrative commands on the file
 - **Insert mode**: enables you to insert text into the file
- Switch to the Insert mode from the command mode by pressing '**i**'
- To return to the command mode and save the changes, press the **Esc** key

Vi Editor

Vi basic commands

- i – Insert text before the current cursor location (goes into insert mode)
- a – Insert text after the current cursor location (goes into insert mode)
- ESC – Terminate insert mode
- u – Undo last change
- o – Open a new line (goes into insert mode)
- dd – Delete line
- D – Delete from the cursor position to the end of the current line
- dw – Delete word
- x – Delete character at the cursor
- X – Delete the character before the cursor location

Vi Editor

- k – Move cursor up
 - j – Move cursor down
 - h – Move cursor left
 - l – Move cursor right
 - Shift + zz – Save the file and quit
 - :w – Save the file but keep it open
 - :q! – Quit without saving
 - :wq – Save the file and quit
 - :x or :exit – Save (but only write if there are changes) and quit
- 



Basic commands

Basic commands

▪ **sudo command**

- Execute a command as another user (default as superuser)
- Syntax: **sudo command**

▪ **clear command**

- Clear the terminal screen
- Syntax: **clear**

Basic commands

▪ Output redirection

- The most basic type of redirection is sending output from a command to a file

command > outputfile

- Use >> to append output from a command to an existing file

data >> outputfile

▪ Input redirection

Takes the content of a file and redirects it to a command

- **command < inputfile**

- E.g.: `wc < test6`

Basic commands

▪ Pipes

- Send the output of one command to the input of another command
 - ✓ `ls > list.txt`
 - ✓ `sort < list.txt`
- The pipe is put between the commands to redirect the output from one to the other
 - ✓ **`command1 | command2`**
 - ✓ E.g.: `ls | sort`
- There's no limit to the number of pipes you can use in a command
 - E.g.: **`ls | sort | more`**

Basic commands

- **ping command**

- Send ICMP ECHO_REQUEST to a network host
- Syntax: **ping [Options] dest**
- Options:
 - ✓ -c count: stop after sending count ECHO_REQUEST packets
 - ✓ -i <interval>: seconds between sending each packet
 - ✓ -s <size>: use <size> as number of data bytes to be sent
- E.g.: ping google.com

- Use **ifconfig -a** to display all interfaces which are currently available, even if down

Basic commands

▪ **grep command**

- Search for PATTERN in each FILE
- Syntax: **grep [OPTIONS] PATTERN [FILEs]**
- Options:
 - ✓ **-i**: ignore case distinctions
 - ✓ **-w**: force PATTERN to match only whole words
 - ✓ **-x**: force PATTERN to match only whole lines
 - ✓ **-n**: print line number with output lines
- E.g.: `grep -i 'hello world' menu.h main.c`

Basic commands

- **wget command**

- Download a file from the web (HTTP, HTTPS, FTP)
- Syntax: **wget [options] [url]**
- Options:
 - ✓ **-O new_name**: save the downloaded file under a different name
 - ✓ **-P directory**: save the downloaded file to specific directory
- E.g.:
wget <https://cdn.kernel.org/kernel/v4.x/linux-4.17.2.tar.xz>

Compression and Decompression

■ tar command

- **Compress** many **files or directories** together into a .tar/.tar.gz/.tar.bz2 archive, and extract the archive
- Syntax: **tar [options] [files]**
- Options:
 - ✓ **-x**: extract files from an archive
 - ✓ **-v**: verbosely list files processed
 - ✓ **-f**: use archive file
 - ✓ **-z**: filter the archive through gzip
 - ✓ **-j**: filter the archive through bzip2
 - ✓ **-c**: create a new archive
 - ✓ **-p**: preserve file permissions
- E.g.:
 - ✓ `tar xf archive.tar` = `tar -xf archive.tar`: extract all files from archive.tar
 - ✓ `tar cf archive.tar file_1 file_2`: create archive.tar from files



```
# ls -l file
-rw-r--r-- 1 root root 0 Nov 19 23:49 file
```

Diagram illustrating the permissions for the file `file`:

- File type** (indicated by the first character `-`)
- Owner (rw-)** (indicated by the next three characters `rw-`)
- Group (r--)** (indicated by the next three characters `r--`)
- Other (r--)** (indicated by the last three characters `r--`)

Legend:

- `r` = Readable
- `w` = Writeable
- `x` = Executable
- `-` = Denied

Ownership

Ownership

- **chown command**

- Change the owner and/or group of each FILE to OWNER and/or GROUP
- Syntax: **chown [OPTIONS] [OWNER][:[GROUP]] FILES**
- Option:
 - R: operate on files and directories recursively
- E.g.:
 - ✓ `chown www-data:www-data /var/www/index.html`
 - ✓ `chown -R www-data:www-data /var/www/`

File / Directory Permission

- View the permissions of files and directories:

ls -l or ls -n

```
# ls -l file
-rw-r--r-- 1 root root 0 Nov 19 23:49 file
```

Diagram illustrating the permissions `-rw-r--r--`:

- File type:** `-` (indicated by a red arrow pointing to the first character)
- Owner (rw-):** `rw` (indicated by a yellow dashed box and a yellow arrow pointing to the second and third characters)
- Group (r--):** `r--` (indicated by a blue dashed box and a blue arrow pointing to the fourth, fifth, and sixth characters)
- Other (r--):** `r--` (indicated by a purple dashed box and a purple arrow pointing to the seventh, eighth, and ninth characters)

Legend:

- `r` = Readable
- `w` = Writeable
- `x` = Executable
- `-` = Denied

- File type: a file (`_`) or a directory (`d`)
- The next fields represent the permission groups: **owner**, **group**, and **other**

File / Directory Permission

- **Permission Groups**

Permission	Description
Owner	Permissions used by the assigned owner of the file or directory
Group	Permissions used by members of the group that owns the file or directory
Other	Permissions used by all users other than the file owner, and members of the group that owns the file or the directory

File / Directory Permission

▪ Permission Set

- **Each permission group has 3 permissions**, called a permission set
- Each set consists of **read**, **write**, and **execute** permissions
- **Each file or directory has 3 permission sets** for the three types of permission groups
- The first permission set represents the **owner** permissions, the second set represents the **group** permissions, and the last set represents the **other** permissions

File / Directory Permission

▪ Permission Set

- The **read, write, and execute** permissions are represented by the characters **r, w, and x**, respectively
- The presence of any of these characters, such as **r**, indicates that the particular permission is granted.
- A dash (**-**) symbol in place of a character in a permission set indicates that a particular permission is denied
- Linux **assigns initial permissions automatically** when a new file or directory is created

File / Directory Permission

▪ Permission Set

- Permissions can also be represented numerically:

✓ **r = 4**

✓ **w = 2**

✓ **x = 1**

✓ **E.g.:**

➤ **-(rw-)(rw-)(r--)**

➤ **-(42-)(42-)(4--)**

➤ **664**

File / Directory Permission

▪ **chmod command**

- To alter the permissions of a file
- Syntax:
 - ✓ **chmod [Options] Mode[,Mode...] File/Dir**
 - ✓ **chmod [Options] Octal-Mode File/Dir**
- Option:
 - R**: change files and directories recursively
- E.g.,
 - ✓ **chmod u+x file_name**
 - ✓ **chmod g+rw file_name**
 - ✓ **chmod o-w file_name**
 - ✓ **chmod u=rwx,g=rx,o= file_name**
 - ✓ **chmod 774 file_name**



Package Management

Package Management

apt-get command

- APT package handling utility
- Retrieve packages and information about them from authenticated sources for installation, upgrade and removal of packages together with their dependencies
- Syntax: **apt-get command pkg1 [pkg2 ...]**

Package Management

apt-get command

- Most used commands:
 - **update** - Retrieve new lists of packages
 - **upgrade** - Perform an upgrade
 - **install** - Install new packages
 - **remove** - Remove packages
 - **purge** - Remove packages and config files
 - **autoremove** - Remove automatically all unused packages

Package Management

apt-get command

- Most used commands:
 - dist-upgrade - Distribution upgrade
 - clean - Erase downloaded archive files
 - autoclean - Erase old downloaded archive files
 - source - Download source archives
 - download - Download the binary package into the current directory



Basic Script Building

Using Multiple Commands

- The key to shell scripts is the **ability to enter multiple commands** and process the results from each command, even possibly passing the results of one command to another
- The shell allows you to chain commands together into a single step
- To **run two commands together**, enter them on the same prompt line, separated with a semicolon

date ; who → shell script

Creating a Script File

Fast and Easy Web Server Installation

What do you want to do?

- 1) Create or create again the username user
- 2) Create users profile (color in bash)
- 3) Update and Install (Apache, PHP, MySQL, SQLite, Django, Subversion)
- 4) Configuring SSH and IPTABLES
- 5) Configure and securitizing Apache
- 6) Configure and securitizing MySQL
- 7) Create SVN & TRAC repos
- 8) Create a Mail Server
- 9) Create a cron backup (mysql, apache, trac & svn)
- 10) Set DNS and to add Google Apps MX records (Only SliceHost.com)
- 11) Install Trac and its Plugins
- 12) I do not know, exit!

ehost.com)

Industrial Camera
Systems Web Inspection
and Event Capturing

Show all dependencies

Creating a Script File

- Use a text editor to **create a file** and then **enter the commands**
- When creating a shell script file, **must specify the shell** you are using in the first line

#!/bin/bash

- After indicating the shell, commands are entered onto each line of the file
can use the semicolon (;) and put both commands on the same line
- A line starts with the **#** symbol aren't interpreted → used to leave **comments**

Creating a Script File

- **Save this script** in a file
- **Give the file owner permission to execute** the file, using the **chmod** command
`chmod +x file_name`
- **Run the script**, using an absolute or relative file path to reference our shell script file in the prompt

./file_name

Creating a Script File

Write the first script

- Script name: test.sh
- Script code:

```
#!/bin/bash  
# This is the first script and the first comment  
pwd  
ls
```

Displaying Messages

- The **echo** command can display a simple text string
- To delineate text strings:
 - By default, you don't need to use quotes
 - Can use either double or single quotes
- Use the **-n parameter** to echo a text string on **the same line** as a command output
- Example:
 - `echo This is a test`
 - `echo "This is a test to see if you're paying attention"`
 - `echo 'Rich says "scripting is easy".'`
 - `echo -n "The time and date are: "`

User Input

- The **read** command is used to ask the user for input. This command takes the input and will save it into a variable.
- Syntax: **read [options] [variable_name(s)]**
- Options:
 - -p: specify a prompt without line break
 - -s: hide the input from terminal
- E.g.:
 - read var_name
 - read -p 'Username: ' username
 - read -sp 'Password: ' password

Using Variables

Variables allow you to **temporarily store information within the shell script** for use with other commands in the script

- **Environment variables**

- **set** command to display a complete list of active environment variables available

```
$ set
BASH=/bin/bash
[...]
HOME=/home/Samantha
HOSTNAME=localhost.localdomain
HOSTTYPE=i386
IFS=$' \t\n'
IMSETTINGS_INTEGRATE_DESKTOP=yes
IMSETTINGS_MODULE=none
LANG=en_US.utf8
LESSOPEN='|/usr/bin/lesspipe.sh %s'
LINES=24
LOGNAME=Samantha
[...]
```

Using Variables

▪ Environment variables

- Use them within scripts by using the environment variable's name preceded by \$

```
$ cat test2
#!/bin/bash
# display user information from the system.
echo "User info for userid: $USER"
echo UID: $UID
echo HOME: $HOME
$
```

- To display an actual \$, must precede it with a backslash character (\)
echo "The cost of the item is \\$15"

Using Variables

▪ User variables

- A shell script allows you to set and use your own variables within the script
- Any text string of up to 20 letters, digits, or an underscore character and case sensitive
- Values are **assigned** to user variables using an **equal sign**. **No spaces** can appear between the variable, the equal sign, and the value

```
var1=10
```

```
var2=-57
```

```
var3=testing
```

```
var4="still more testing"
```

- User variables can be referenced using **\$** or **\${variable}**

Using Variables

- **User variables**

```
$ cat test3
#!/bin/bash
# testing variables
days=10
guest="Katie"
echo "$guest checked in $days days ago"
days=5
guest="Jessica"
echo "$guest checked in $days days ago"
$
```

Using Variables

▪ Command substitution

- The ability to extract information from the output of a command and assign it to a variable
- There are **2 ways to assign the output of a command to a variable**:
 - ✓ **The backtick character (`)**: `test=`date``
 - ✓ **The `$()` format**: `test=$(date)`

Using Variables

- **Special variables**

- \$0: The filename of the current script
- \$n: These variables correspond to the arguments with which a script was invoked
- \$#: The number of arguments supplied to a script
- \$@: The variable includes all the command line parameters within a single variable

Performing Math

There are 2 different ways to perform mathematical operations:

- The **expr** command

```
$ cat test6
#!/bin/bash
# An example of using the expr command
var1=10
var2=20
var3=$(expr $var2 / $var1)
echo The result is $var3
```

Operator	Description
ARG1 ARG2	Returns ARG1 if neither argument is null or zero; otherwise, returns ARG2
ARG1 & ARG2	Returns ARG1 if neither argument is null or zero; otherwise, returns 0
ARG1 < ARG2	Returns 1 if ARG1 is less than ARG2; otherwise, returns 0
ARG1 <= ARG2	Returns 1 if ARG1 is less than or equal to ARG2; otherwise, returns 0
ARG1 = ARG2	Returns 1 if ARG1 is equal to ARG2; otherwise, returns 0
ARG1 != ARG2	Returns 1 if ARG1 is not equal to ARG2; otherwise, returns 0
ARG1 >= ARG2	Returns 1 if ARG1 is greater than or equal to ARG2; otherwise, returns 0
ARG1 > ARG2	Returns 1 if ARG1 is greater than ARG2; otherwise, returns 0
ARG1 + ARG2	Returns the arithmetic sum of ARG1 and ARG2
ARG1 - ARG2	Returns the arithmetic difference of ARG1 and ARG2
ARG1 * ARG2	Returns the arithmetic product of ARG1 and ARG2
ARG1 / ARG2	Returns the arithmetic quotient of ARG1 divided by ARG2
ARG1 % ARG2	Returns the arithmetic remainder of ARG1 divided by ARG2
STRING : REGEXP	Returns the pattern match if REGEXP matches a pattern in STRING

Performing Math

There are 2 different ways to perform mathematical operations:

- Using brackets

- **`$(operation)`**

- Example: `var1=$(1 + 5)`

```
$ cat test7
#!/bin/bash
var1=100
var2=50
var3=45
var4=$((var1 * (var2 - var3))
echo The final result is $var4
$
```

Working with the if-then Statement

The most basic type of structured command is the if-then statement

```
if command
then
    commands
fi
```

```
$ cat test1.sh
#!/bin/bash
# testing the if statement
if pwd
then
    echo "It worked"
fi
$
```

Exploring the if-then-else Statement

The if-then-else statement provides another group of commands in the statement

```
if command
then
    commands
else
    commands
fi
```

```
#!/bin/bash
# testing the else section
#
testuser=NoSuchUser
#
if grep $testuser /etc/passwd
then
    echo "The bash files for user $testuser are:"
    ls -a /home/$testuser/.b*
    echo
else
    echo "The user $testuser does not exist on this system."
    echo
fi
```


Nesting ifs

```
if command1
then
    commands
elif command2
then
    more commands
fi
```

```
$ cat test5.sh
#!/bin/bash
# Testing nested ifs - use elif
#
testuser=NoSuchUser
#
if grep $testuser /etc/passwd
then
    echo "The user $testuser exists on this system."
#
elif ls -d /home/$testuser
then
    echo "The user $testuser does not exist on this system."
    echo "However, $testuser has a directory."
#
```

Testing condition

- To test a condition, use square brackets:

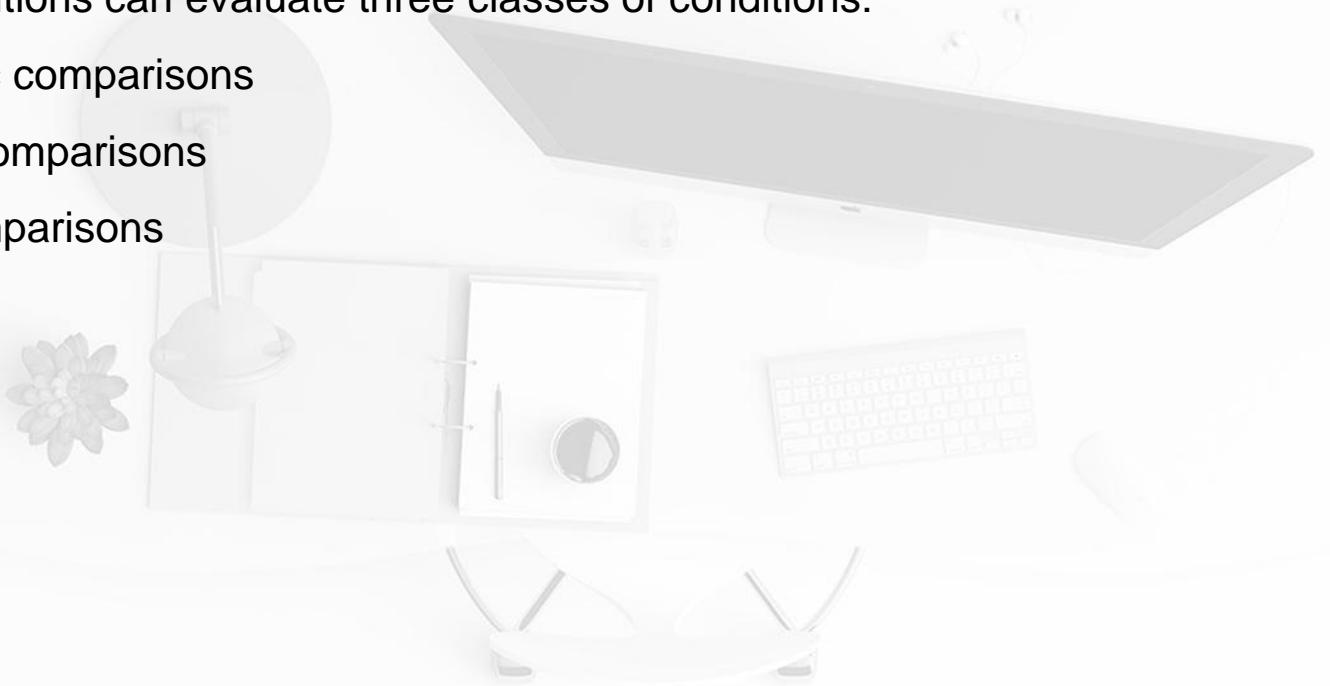
```
if [ condition ]  
then  
    commands  
fi
```

- **Must have a space** after the first bracket and a space before the last bracket

Testing condition

Test conditions can evaluate three classes of conditions:

- Numeric comparisons
- String comparisons
- File comparisons



Testing condition

Using numeric comparisons

Comparison	Description
<code>n1 -eq n2</code>	Checks if n1 is equal to n2
<code>n1 -ge n2</code>	Checks if n1 is greater than or equal to n2
<code>n1 -gt n2</code>	Checks if n1 is greater than n2
<code>n1 -le n2</code>	Checks if n1 is less than or equal to n2
<code>n1 -lt n2</code>	Checks if n1 is less than n2
<code>n1 -ne n2</code>	Checks if n1 is not equal to n2

Testing condition

Using numeric comparisons

Example:

```
$ cat numeric_test.sh
#!/bin/bash
# Using numeric test evaluations
#
value1=10
value2=11
#
if [ $value1 -gt 5 ]
then
    echo "The test value $value1 is greater than 5"
fi
#
if [ $value1 -eq $value2 ]
then
    echo "The values are equal"
else
    echo "The values are different"
fi
```

Testing condition

Using string comparisons

Comparison	Description
<code>str1 = str2</code>	Checks if <code>str1</code> is the same as string <code>str2</code>
<code>str1 != str2</code>	Checks if <code>str1</code> is not the same as <code>str2</code>
<code>str1 < str2</code>	Checks if <code>str1</code> is less than <code>str2</code>
<code>str1 > str2</code>	Checks if <code>str1</code> is greater than <code>str2</code>
<code>-n str1</code>	Checks if <code>str1</code> has a length greater than zero
<code>-z str1</code>	Checks if <code>str1</code> has a length of zero

Testing condition

Using string comparisons

Example 1:

```
$ cat test7.sh
#!/bin/bash
# testing string equality
testuser=rich
#
if [ $USER = $testuser ]
then
    echo "Welcome $testuser"
fi
$
$ ./test7.sh
Welcome rich
$
```

Testing condition

Using string comparisons

Example 2:

```
$ cat test9b.sh
#!/bin/bash
# testing string sort order
val1=Testing
val2=testing
#
if [ $val1 \> $val2 ]
then
    echo "$val1 is greater than $val2"
else
    echo "$val1 is less than $val2"
fi
$
$ ./test9b.sh
Testing is less than testing
$
$ sort testfile
testing
Testing
```


Testing condition

Using file comparisons

Comparison	Description
<code>-d file</code>	Checks if <code>file</code> exists and is a directory
<code>-e file</code>	Checks if <code>file</code> exists
<code>-f file</code>	Checks if <code>file</code> exists and is a file

Testing condition

Using file comparisons

Comparison	Description
<code>-r file</code>	Checks if <code>file</code> exists and is readable
<code>-s file</code>	Checks if <code>file</code> exists and is not empty
<code>-w file</code>	Checks if <code>file</code> exists and is writable
<code>-x file</code>	Checks if <code>file</code> exists and is executable
<code>-O file</code>	Checks if <code>file</code> exists and is owned by the current user
<code>-G file</code>	Checks if <code>file</code> exists and the default group is the same as the current user
<code>file1 -nt file2</code>	Checks if <code>file1</code> is newer than <code>file2</code>
<code>file1 -ot file2</code>	Checks if <code>file1</code> is older than <code>file2</code>

Testing condition

Using file comparisons

Example:

```
$ cat test11.sh
#!/bin/bash
# Look before you leap
#
jump_directory=/home/arthur
#
if [ -d $jump_directory ]
then
    echo "The $jump_directory directory exists"
    cd $jump_directory
    ls
else
    echo "The $jump_directory directory does not exist"
fi
#
$
$ ./test11.sh
The /home/arthur directory does not exist
$
```

Considering Compound Testing

The if-then statement allows you to use Boolean logic to combine tests.

You can use these two Boolean operators:

```
[ condition1 ] && [ condition2 ]  
[ condition1 ] || [ condition2 ]
```

The for Command

Allow you to create a loop that iterates through a series of values

```
for var in list
do
    commands
done
```

The for Command

- **Reading a list from a variable**

```
$ cat test4
#!/bin/bash
# using a variable to hold the list

list="Alabama Alaska Arizona Arkansas Colorado"
list=$list" Connecticut"

for state in $list
do
    echo "Have you ever visited $state?"
done
```

The for Command

- **Reading values from a command**

```
$ cat states
Alabama
Alaska
Arizona
Arkansas
Colorado
Connecticut
Delaware
Florida
Georgia
```

```
$ cat test5
#!/bin/bash
# reading values from a file

file="states"

for state in $(cat $file)
do
    echo "Visit beautiful $state"
done
```

The for Command

- **Changing the field separator**
 - The IFS (internal field separator) environment variable defines a list of characters the bash shell uses as field separators
 - A space
 - A tab
 - A newline
 - You can temporarily change the IFS environment variable values
`IFS=$'\n'`

The for Command

- **Changing the field separator**

```
$ cat test5b
#!/bin/bash
# reading values from a file

file="states"

IFS=$'\n'

for state in $(cat $file)
do
    echo "Visit beautiful $state"
done
```

```
$ ./test5b
Visit beautiful Alabama
Visit beautiful Alaska
Visit beautiful Arizona
Visit beautiful Arkansas
Visit beautiful Colorado
Visit beautiful Connecticut
Visit beautiful Delaware
```

The for Command

- **Reading a directory using wildcards**

```
$ cat test6
#!/bin/bash
# iterate through all the files in a directory

for file in /home/rich/test/*
do

    if [ -d "$file" ]
    then
        echo "$file is a directory"
    elif [ -f "$file" ]
    then
        echo "$file is a file"
    fi
done
```

Controlling the Loop

Commands help us control what happens inside of a loop:

- The **break** command
- The **continue** command





Q&A

References

- **Unix Vs Linux: What is Difference Between UNIX and Linux**

<https://www.softwaretestinghelp.com/unix-vs-linux/amp/>

- **The Linux command line for beginner**

<https://ubuntu.com/tutorials/command-line-for-beginners#6-a-bit-of-plumbing>