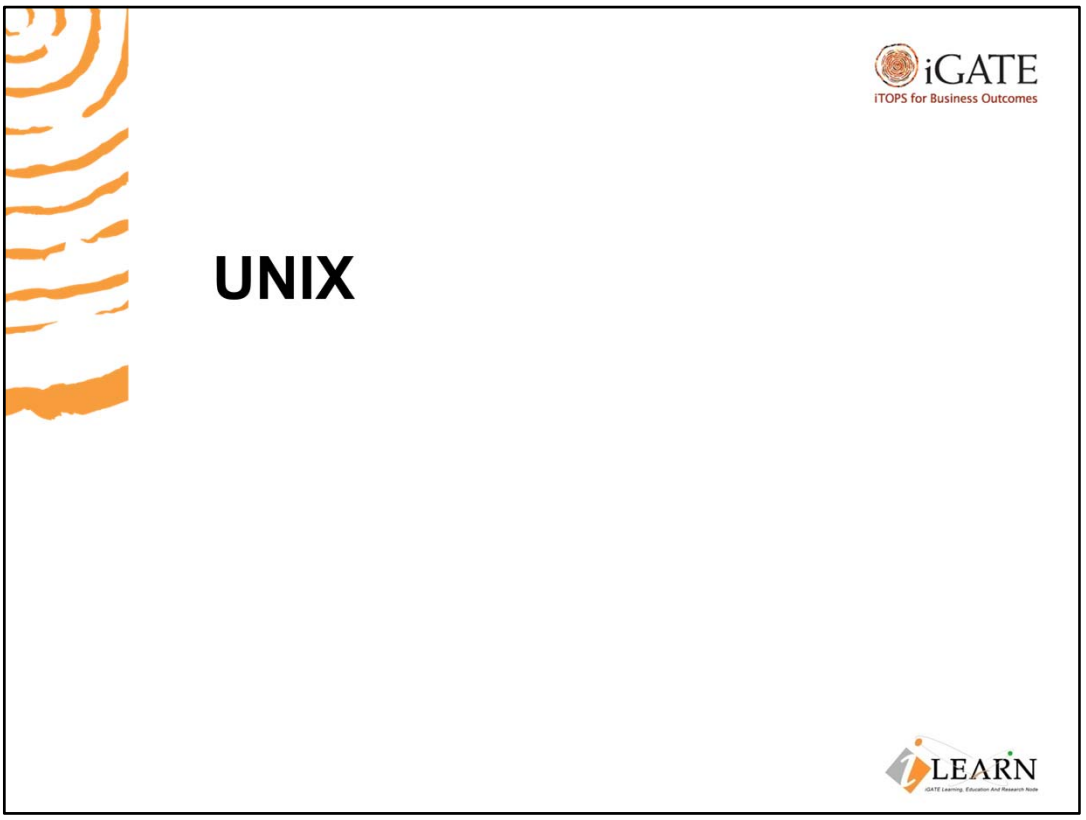


Document History

Date	Course Version No.	Software Version No.	Developer / SME	Change Record Remarks
	1.0	UNIX	Veena Deshpande	
30-Sep-2009	1.1	UNIX	Kishori Khadilkar	Revamped with new methodology
14-Oct-2009	1.1	UNIX	CLS team	Review
08-Jun-2011	1.2	UNIX	Rathnajothi Perumalsamy	Revamp/Refinement

Course Goals and Non Goals

iGATE
iTOPS for Business Outcomes

- **Course Goals**
  - To understand UNIX operating system, Shell commands, Introduction to Processes, Writing simple shell scripts

- **Course Non Goals**
  - NA

iLEARN
iGATE Learning, Education And Research Node

- 3 -

Pre-requisites

- None

Intended Audience

iGATE
iTOPS for Business Outcomes

- Novice Users

iLEARN
iGATE Learning, Education And Research Node

- 5 -

Day Wise Schedule

iGATE
iTOPS for Business Outcomes

- **Day 1**
  - Lesson 1: Introduction to UNIX and Basic UNIX commands
  - Lesson 2: UNIX file system
- **Day 2:**
  - Lesson 3: Filters
  - Lesson 4: Introduction to Bourne Shell
  - Lesson 5: Vi Editor
- **Day 3:**
  - Lesson 6: Processes and related commands
  - Lesson 7: Shell Programming
  - Lesson 8 (Self Study): AWK
- **Day 4:**
  - Lesson 9: Programming Development

iLEARN

Table of Contents

Table of Contents

Table of Contents

iGATE
iTOPS for Business Outcomes

iLEARN
iGATE Learning, Education And Research Node

Table of Contents

Table of Contents

Table of Contents

iGATE
iTOPS for Business Outcomes

iLEARN
iGATE Learning, Education And Research Node

References

iGATE
iTOPS for Business Outcomes

- UNIX Concepts and Application; by Sumitabha Das
- The Unix Programming Environment; by Kernighan and Pike.
- UNIX Primer Plus, Third Edition; by Don Martin, Stephen Prata, Mitchell Waite, Michael Wessler, and Dan Wilson
- Advanced Unix : a programmers guide; by Stephen Prata

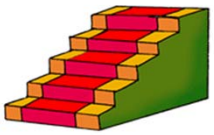
iLEARN
iGATE Learning, Education And Research Node

- 13 -

Next Step Courses (if applicable)

iGATE
iTOPS for Business Outcomes

- Linux Internal



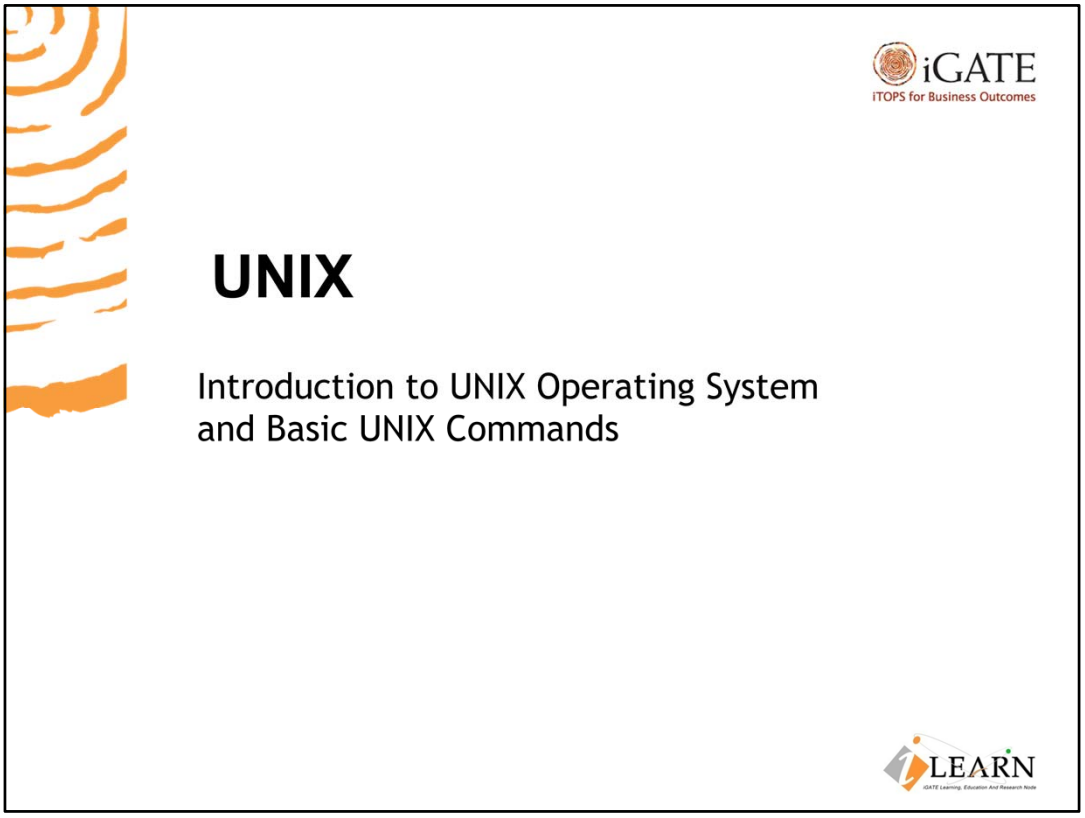
iLEARN
iGATE Learning, Education And Research Node

Other Parallel Technology Areas

iGATE
iTOPS for Business Outcomes

- Windows OS

iLEARN
iGATE Learning, Education And Research Node


iGATE
iTOPS for Business Outcomes
UNIX
Introduction to UNIX Operating System
and Basic UNIX Commands
iLEARN

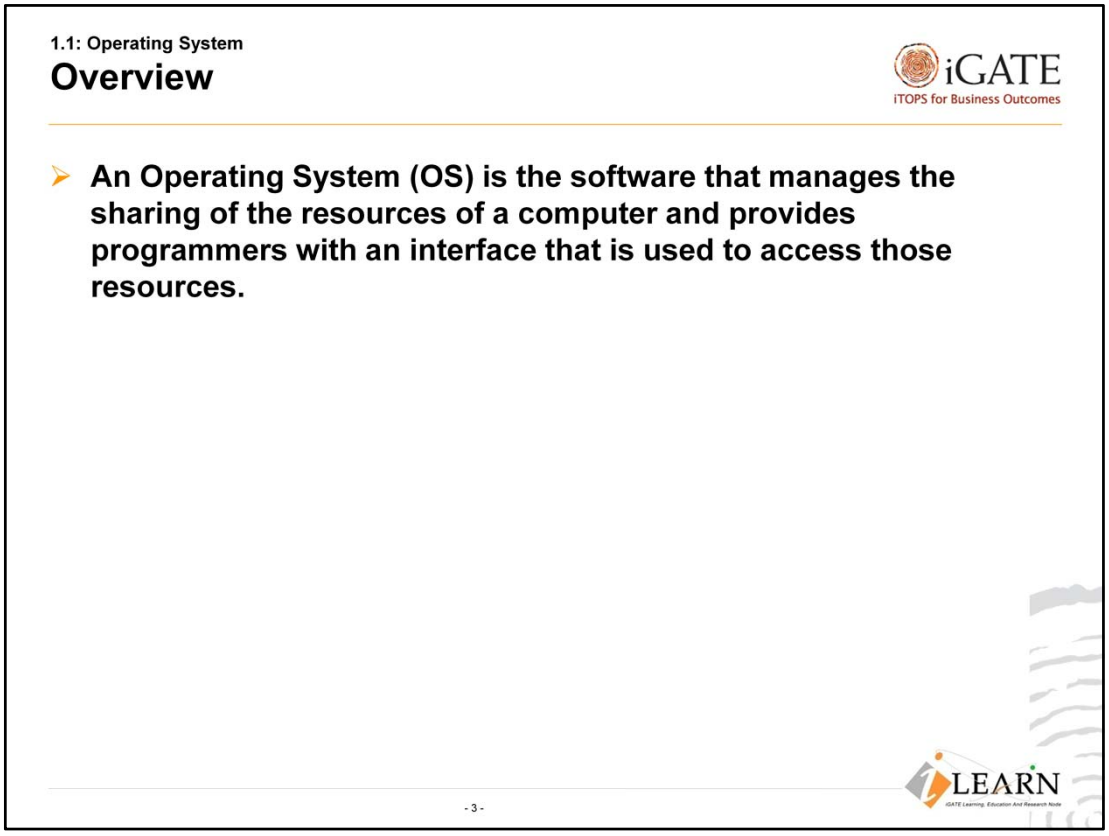
Lesson Objectives

iGATE
iTOPS for Business Outcomes

- **In this lesson, you will learn:**
  - Operating System
    - Functions of Operating System
    - History of UNIX
    - Features of UNIX
    - UNIX System Architecture
  - Basic UNIX Commands

iLEARN
iGATE Learning, Education And Research Node

- 2 -

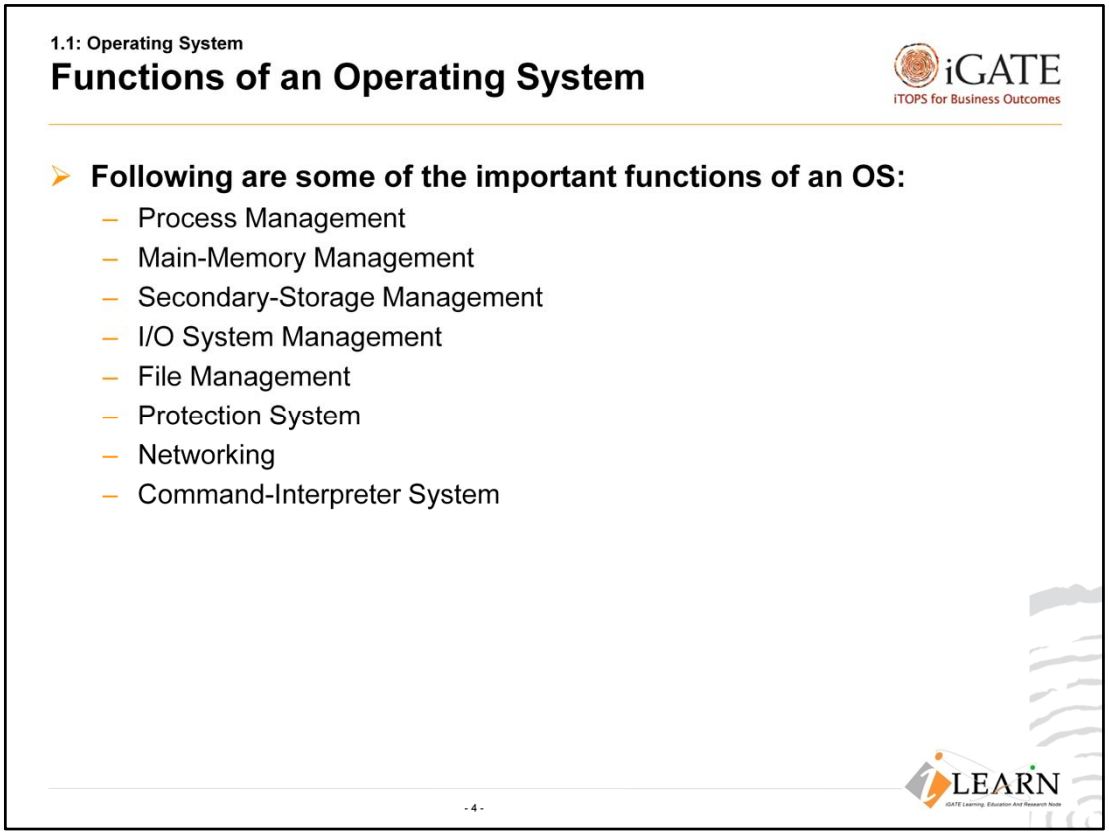


Operating System:
Overview:

An **Operating System (OS)** is the software that manages the sharing of the resources of a computer and provides programmers with an interface used to access those resources.

An Operating System processes **system data** and **user input**, and responds by allocating and managing tasks and internal system resources as a service to users and programs of the system. At the foundation of all system software, an Operating System performs basic tasks such as controlling and allocating memory, prioritizing system requests, controlling input and output devices, facilitating networking and managing file systems. Most Operating Systems come with an application that provides a user interface for managing the operating system, such as a command line interpreter or graphical user interface. The operating system forms a platform for other system software and for application software.

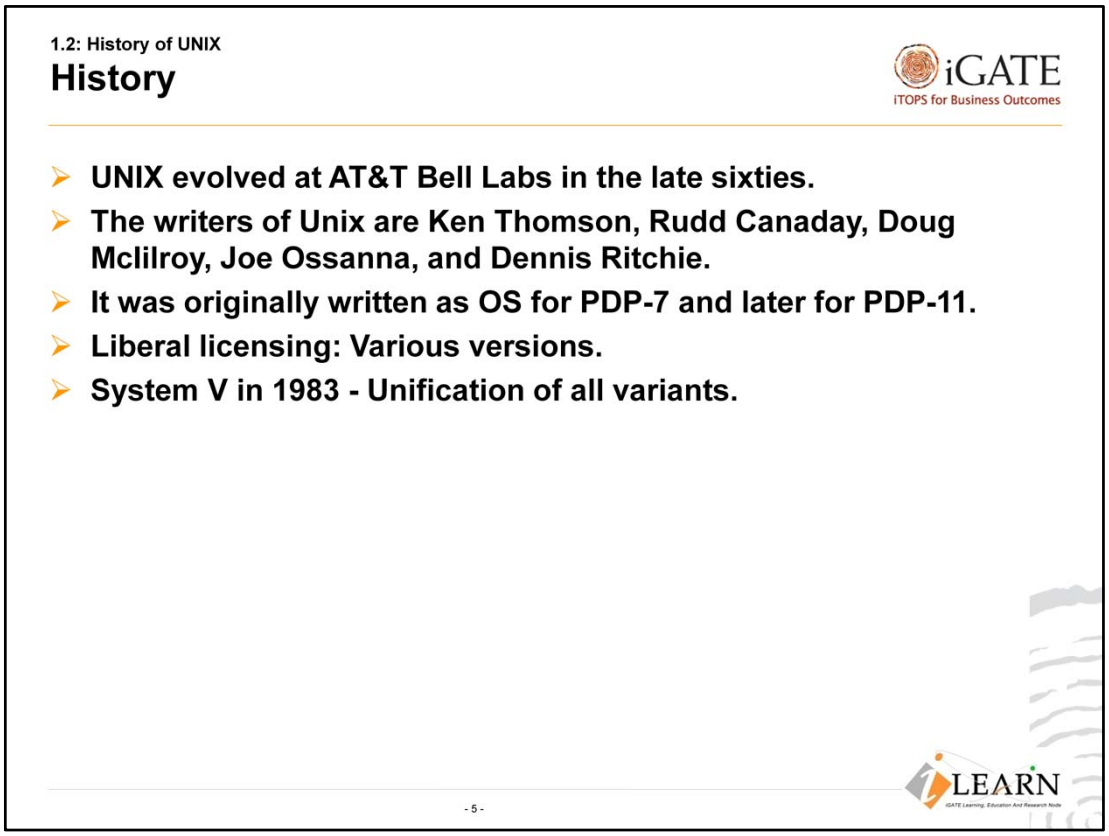
The most commonly used contemporary desktop and laptop (notebook) OS is **Microsoft Windows**. More powerful servers often employ Linux, FreeBSD, and other Unix-like systems. However, these Unix-like operating systems, especially Mac OS X, are also used on personal computers.



Functions of an OS:

Following are the various services provided by the Operating system:

1. **Process Management:** It involves creation and deletion of user and system processes, deadlock handling, and so on.
2. **Main-Memory Management:** It involves keeping track of the parts of memory that are being used, allocating/deallocating memory space as required, etc.
3. **Secondary-Storage Management:** It involves free-space management, disk scheduling, storage allocation.
4. **I/O System Management:** It deals with hardware specific drivers for devices, keeps it all hidden from the rest of the system.
5. **File Management:** It involves creating/deleting files and directories, backup, and so on.
6. **Protection System:** It involves controlling access to programs, processes, or users
7. **Networking:** It generalizes the network access.
8. **Command-Interpreter System:** It provides an interface between the user and the OS.



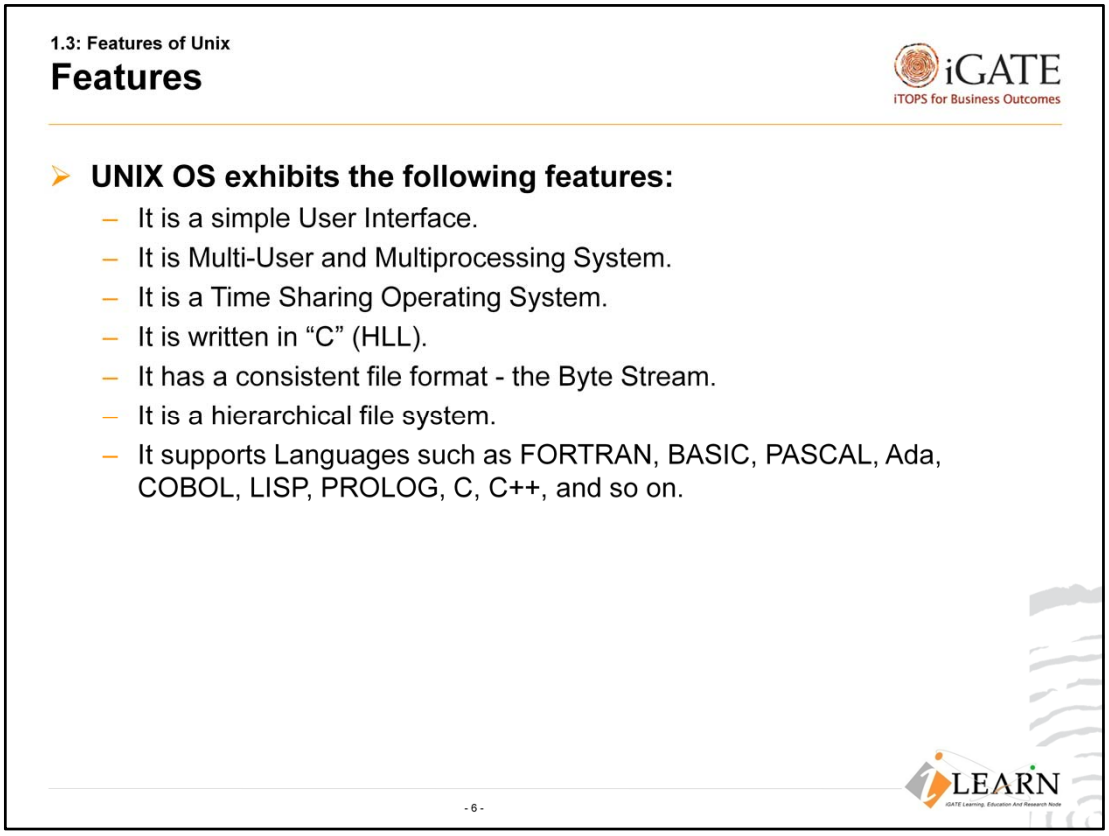
History of UNIX:

Unix had a modest beginning at AT&T Bell Laboratories in late sixties, when AT&T withdrew its team from the MULTICS project. The immediate motivation towards development of Unix was a Space Travel game that Thompson had developed for PDP-7. Finding the game slow on the PDP machine, he requested for a DEC-10 machine. The request was rejected. This led to the idea of designing a new operating system for PDP-7.

As a result, a multitasking system supporting two users was developed. It had an elegant file system, a command interpreter, and a set of utilities. Initially called UNICS (as a pun on MULTICS), by 1970, it came to be known as Unix. Ken Thompson, along with Rudd Canaday, Doug McIlroy, joe Ossanna, and Dennis Ritchie, were the writers of this operating system. Ritchie helped the system to be moved to PDP-11 system in 1970. Ritchie and Kerningham also wrote a compiler for “C”.

Unix was originally written in Assembly language. In 1973, Ritchie and Thompson rewrote the Unix kernel in “C”. This was a revolutionary step, as earlier the operating systems were written in assembly languages. The idea of writing it in “C” was so that the Operating system could now be ported (the speed, in effect, was traded off). It also made the system easier to maintain and adapt to particular requirements.

Around 1974, the system was licensed to universities for educational purposes, and was later available for commercial use. The licensing by AT&T was very liberal – the source code was made available to universities, industries, and government organizations. This led to development of various versions of Unix as everybody added their own utilities. The important ones amongst them include BSD (Berkeley Software Distribution from University of California, Berkeley), SunOS (from Sun Microsystems).



Features of UNIX:

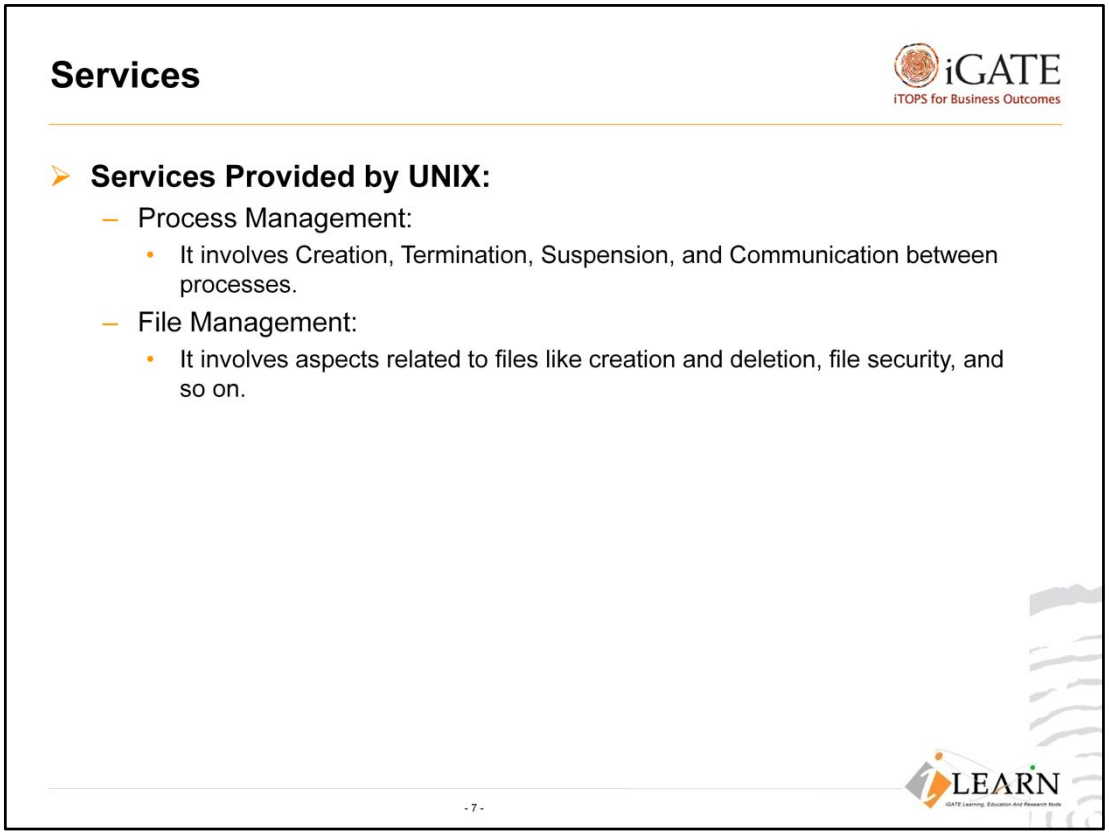
UNIX is a **timesharing operating system** written in "C". It is a **multi-user** as well as a **multiprocessing system**. Many users can work with multiple tasks at the same time. **Round Robin Scheduling** with fixed time slices is the algorithm that is used by the CPU for scheduling tasks.

UNIX, as it is written in "C", is portable. It is available on a variety of platforms ranging from a 8088 based PC to a parallel processing system like Cray 2.

Everything in UNIX is a **file** – even if it is a directory or a device. The file format is consistent. A file is nothing but a **stream of bytes** – it is not considered as containing fixed length records. UNIX follows a hierarchical file system. All files are related with root at the top of the hierarchy – it follows an inverted tree structure.

The kernel is a set of "C" program that controls the resources of a computer and allocates them amongst its users. It lets users run their programs, controls peripheral devices, and provides a file system to manage programs and data. It is loaded into memory when system is booted. It is often called as "the" operating system.

The approach in UNIX is that there is no need to have separate utilities to solve each and every complex problem. Instead, by combining several commands (each of which is capable of doing a simple job), it is possible to solve bigger problems. Pipes and filters allow building solutions from simple commands – they also make UNIX powerful and flexible.



Services provided by UNIX:

Amongst the various services that UNIX operating system provides are Process Management (Creation, Termination, Suspension and communication between processes) and File Management (creation and deletion of files, allowing for dynamic growth of files, security of files etc). Files and processes are two major entities in Unix. A file resides in memory – it is static in nature – as against a process, which is alive and exists in time.

UNIX has a very simple user interface: programmers have written it and it is meant for programmers, hence there are absolutely no “frills”. However, graphical interfaces are becoming available in latter releases.

UNIX includes editors and compilers (for various languages like C, C++, Pascal, Fortran, Prolog, Lisp, Java, and so on). It has several utilities like calculators, graphics and statistical packages, and tools for document preparation.

UNIX includes an online help facility, which is very useful to look at the syntax involving several different options for the many commands of UNIX.

File Management:

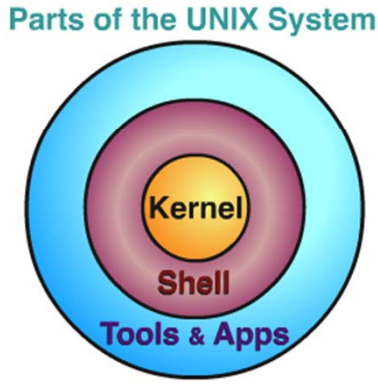
It helps to create or delete files. It assigns read, write, and execute permissions for users, groups, and others to restrict access to file.

1.4: UNIX System Architecture

UNIX System Architecture

iGATE iTOPS for Business Outcomes

- Following is a pictorial representation of the UNIX system:



iLEARN

- 8 -

UNIX System Architecture:

The UNIX system is functionally organized at three levels:

- The **kernel**, which schedules tasks and manages storage
- The **shell**, which connects and interprets users' commands, calls programs from memory, and executes them
- The **tools** and **applications** that offer additional functionality to the operating system

There is a policy of division of labour that is followed between the kernel and the shell on the Unix systems. The **kernel** interacts with the machine hardware, and the **shell** interacts with the User.

The kernel:

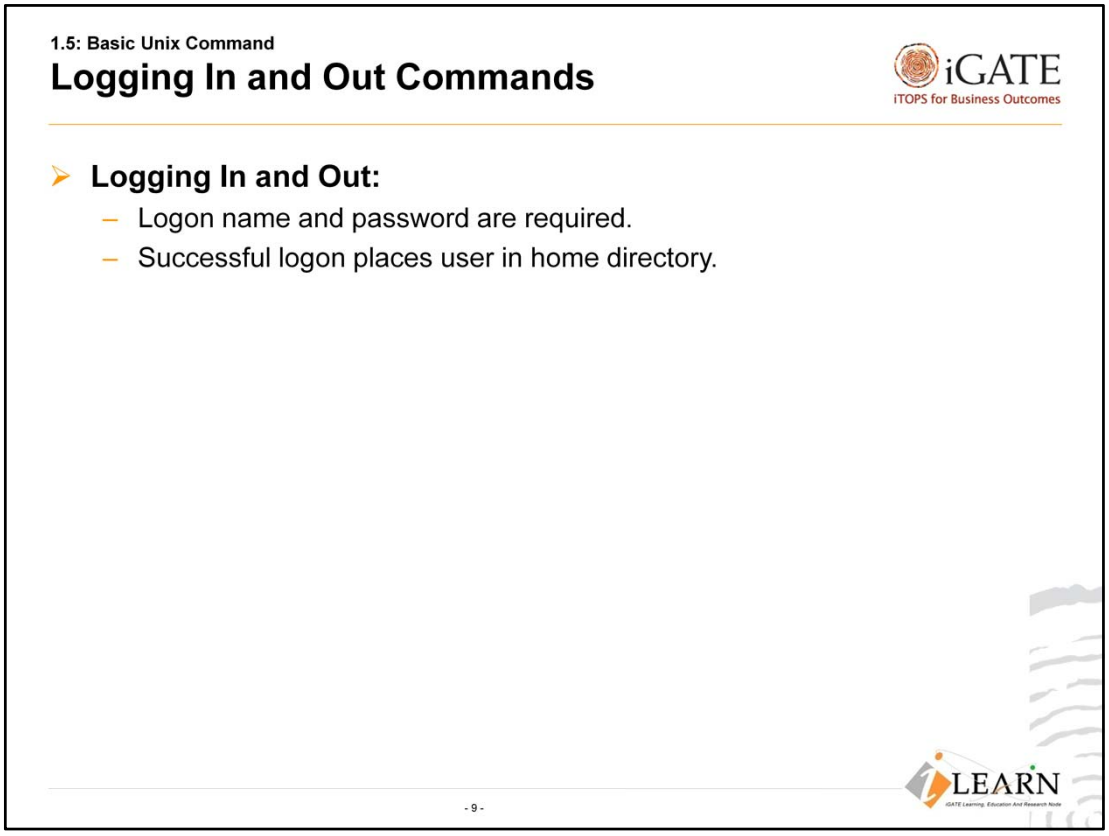
The heart of the operating system, the kernel controls the hardware and turns part of the system on and off at the programmer's command. Suppose you ask the computer to list all the files in a directory. Then the kernel tells the computer to read all the files in that directory from the disk and display them on your screen.

The shell:

The **shell** is technically a **UNIX** command that interprets the user's requests to execute commands and programs. The **shell** acts as the main interface for the system, communicating with the **kernel**. The shell is also programmable in UNIX. There are many different types of shells like **Bourne Shell**, **Korn Shell**, and **C Shell,** most notably the command driven **Bourne Shell** and the **C Shell**, and menu-driven shells that make it easier for beginners to use. Whatever shell is used, its purpose remains the same -- to act as an interpreter between the user and the computer.
We will see more about shell later.

Tools and applications:

There are hundreds of tools available to UNIX users, although some have been written by third party vendors for specific applications. Typically, tools are grouped into categories for certain functions, such as word processing, business applications, or programming.



Basic Unix Commands:

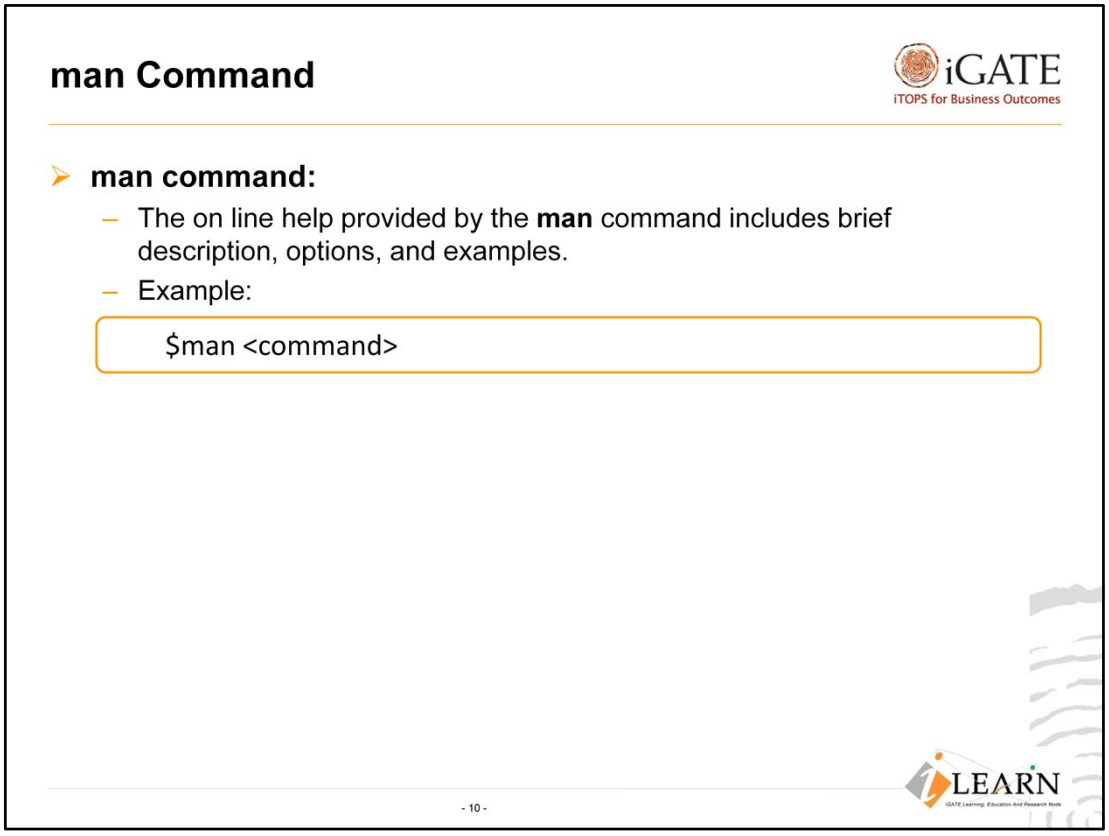
In order to work with Unix, one needs a login name and password, which the system administrator needs to assign. With these in hand, one can "log on" to a Unix System and start a "session" with Unix by typing out the login name and password at the prompt. Once the session is through, one needs to "log off" from the system.

Logging In and Out:

Once the terminal is connected to a Unix system, it displays the "login:" message on screen. The login name, given by the system administrator needs to be typed here. If the password has been set, the system will prompt for the password (printing will be turned off when password is typed).

In case of valid **user name** and **password**, the user will get logged on to the system. The system displays a prompt, typically a **$ sign** (it can be different as it is possible to change prompts) – which indicates that the system is ready to accept commands. There can be some messages and other notifications before the prompt.

On successful logon, user is automatically placed in home directory, the name of home directory being usually the same as the login name.
If there is an error in validating the user name and password, then the system requests the user to re-enter the same. Usually due to security considerations, system allows only a fixed number of attempts to re-enter information.



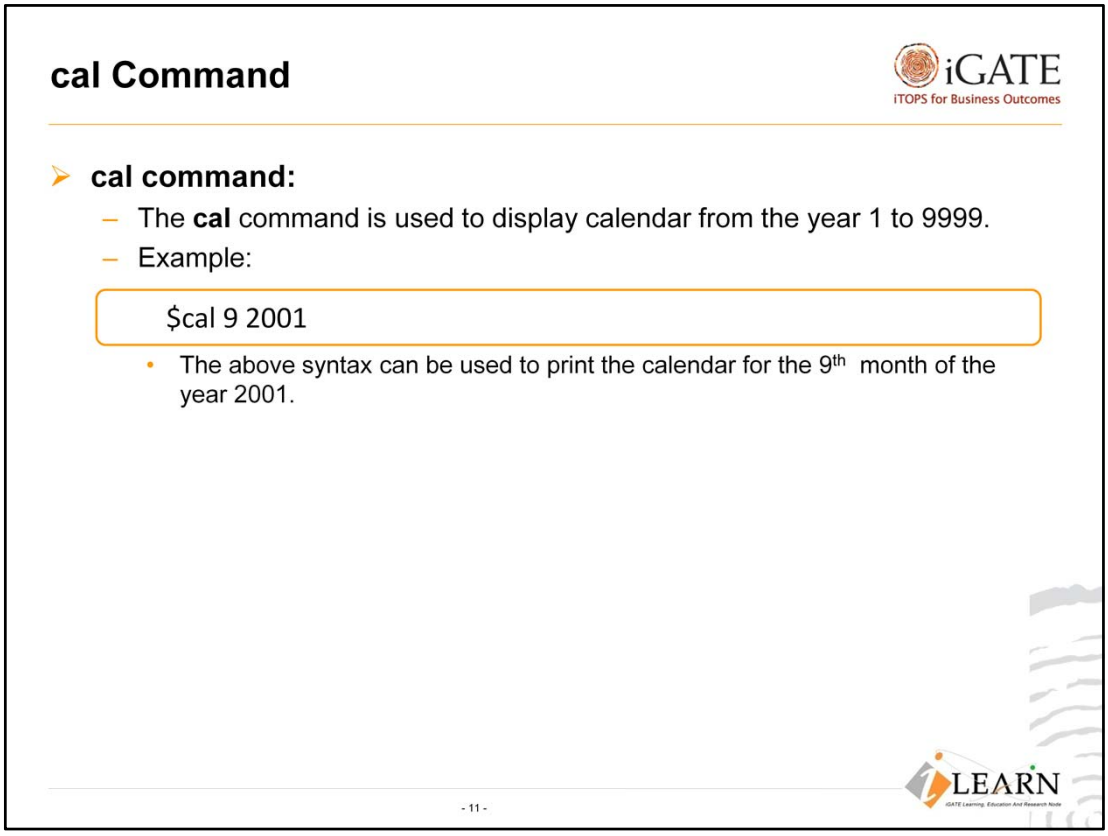
man Command:

Online help using the man command:

Using the **man** command, it is possible to get a brief description about a command including all its options as well as some suitable examples.

Example: To get help on **passwd** command, use the following syntax:

```
$ man passwd
```



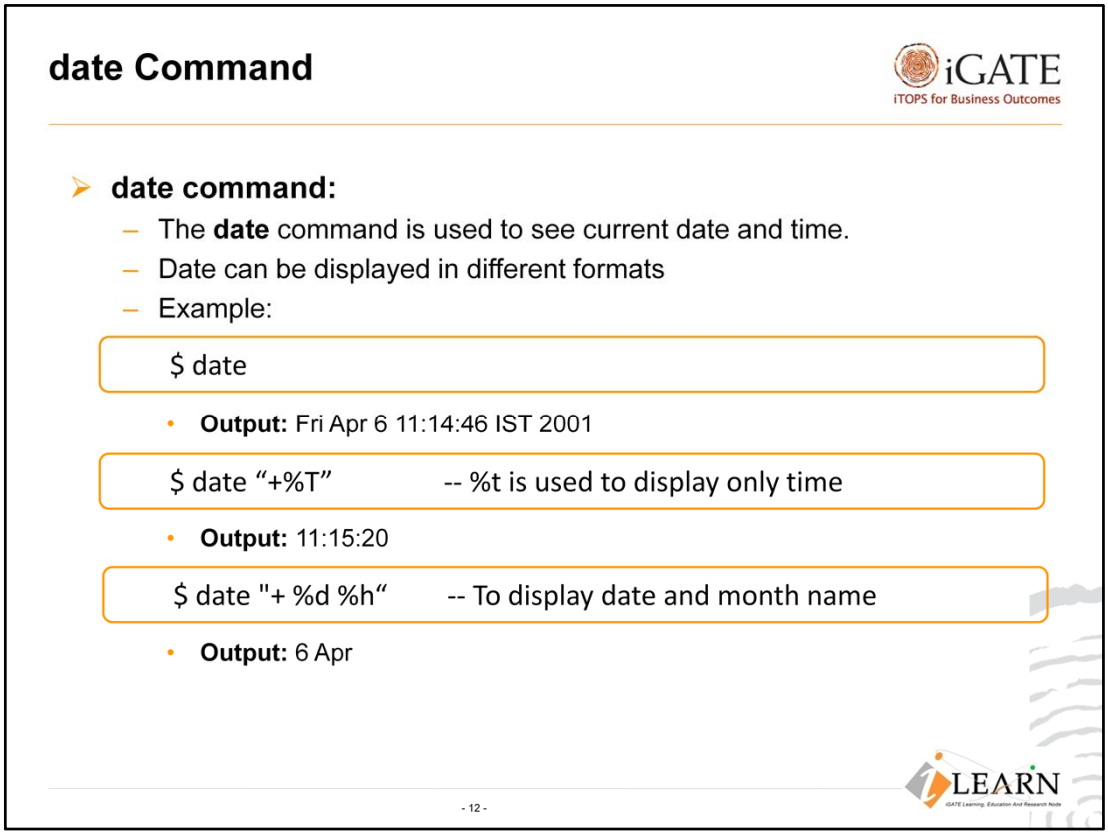
The calendar – cal command:

Any calendar from the year 1 to 9999 (either for a month or complete year), can be displayed using this command.

An example is shown in the above slide.

```
September 2001
      Su Mo Tu We Th Fr Sa
                        1
       2  3  4  5  6  7  8
       9 10 11 12 13 14 15
      16 17 18 19 20 21 22
      23 24 25 26 27 28 29
      30
```

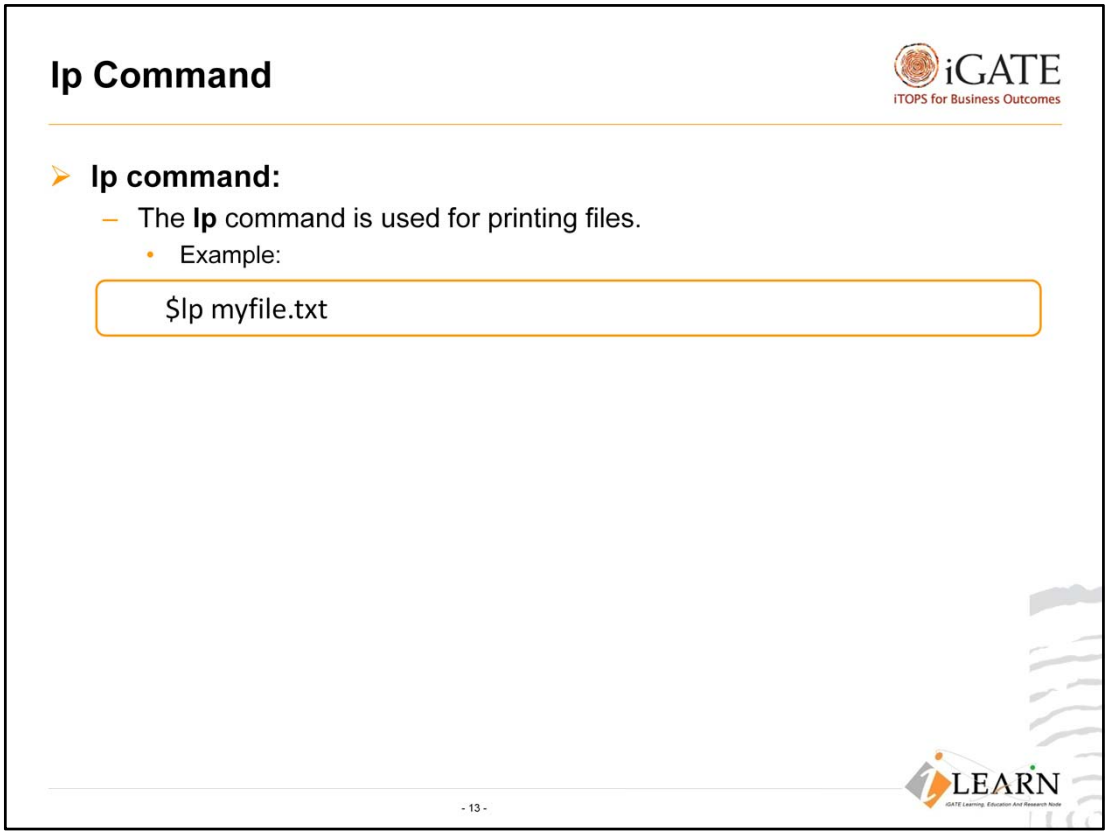
Example: Using the cal command, observe the calendar for September 2001!



Displaying System Date – date:
Unix has an internal clock, which actually stores the number of seconds elapsed from 1 Jan. 1970; and it is used for time stamping. A number of options are separately available to retrieve various components of the date.

Table: Commands used with date command

Option	Description
m	Month
h	Month Name
d	Day of month
y	Year (Last 2 digits)
H	Hour
M	Minutes
S	Second
T	Time in hh:mm:ss
A	Day of week (Sun to Sat)
R	Time in AM/PM

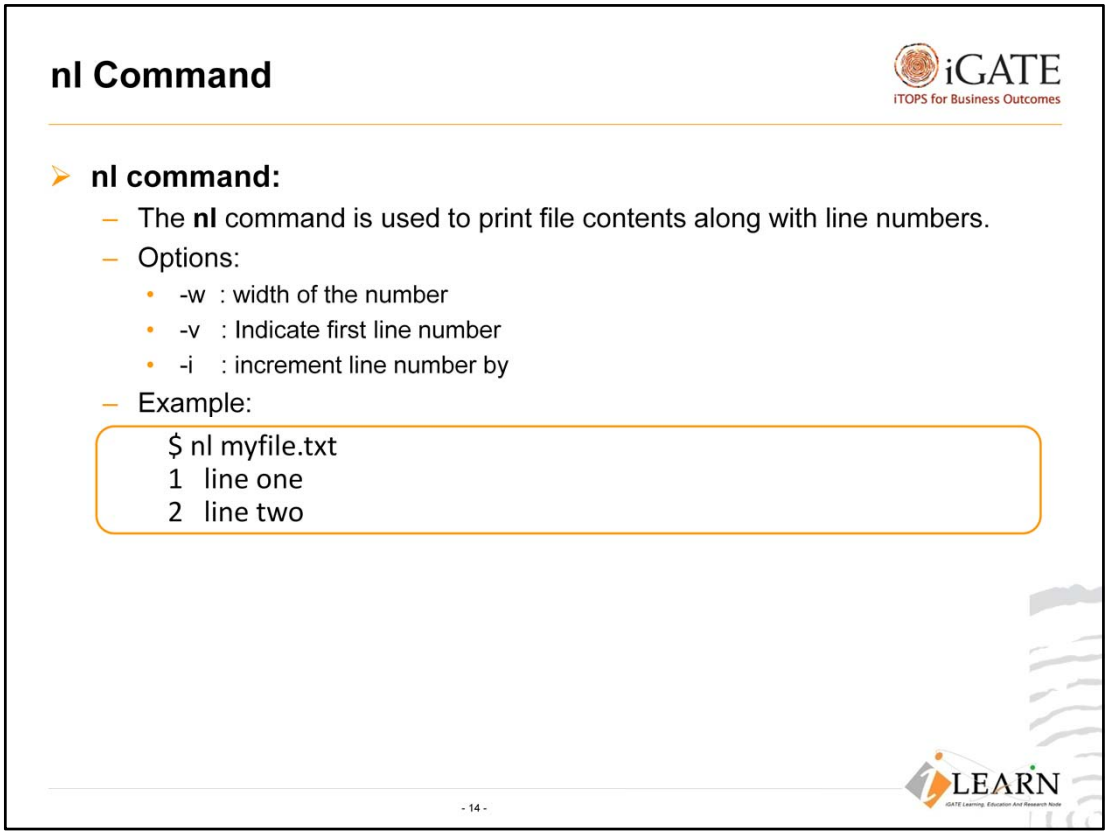


lp command – for printing files:

Jobs can be queued up for printing by using the spooling facility of Unix. Several users can print their files without any conflict. This command can be used to print the file, it returns the job number that can later be used to check the status of job.

The command is used as follows:

```
$ lp file1.txt
```

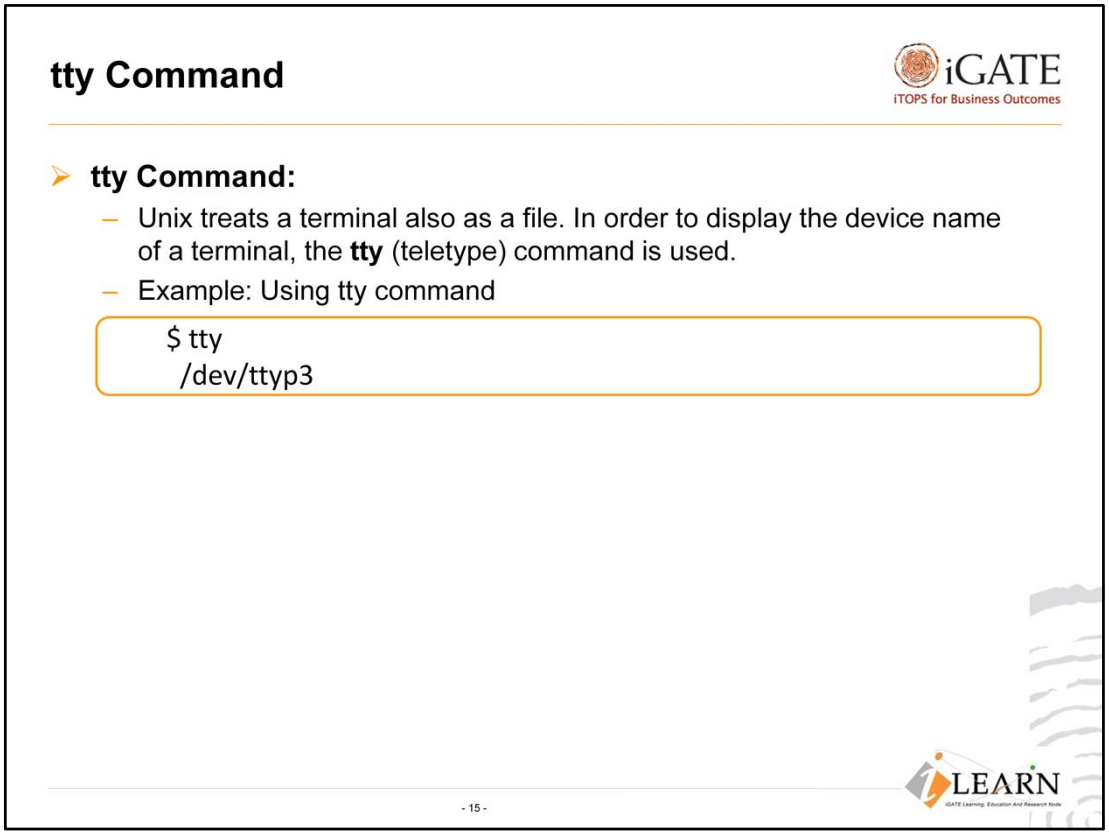


Line Numbering – nl :
This command elaborates schemes for numbering lines.
Options:

- -w : width of the number
- -v : Indicate first line number
- -i : increment line number by

Example: Using the nl command:

```
$nl -w2 -v40 -i7 file1.txt
40      one
47      two
54      three
61      four
```



tty Commands:

Unix treats terminal also as a file. In order to display the device name of a terminal, the command used is tty (**t**ele**ty**pe).

The above slide shows an example on using the **tty** command.

In order to display the current terminal related settings, the **stty** command can be used.

The output of the stty command depends on the Unix implementation.

This command can also be used to change some settings. For example, in order to use **<Ctrl-a>** instead of **<Ctrl-d>** as end of file indicator.

Login details – who command:

Unix maintains an account of all current users of system, the list of which can be printed using this command. The command can also be used to get one's own login details.

Example 1: Using the who command

```
$ who
```

```
ssdesh      ttyp0     Mar 29 09:00
root        tty01     Mar 29 10:32
root        tty03     Mar 29 10:37
root        tty11     Mar 29 09:52
pmaint      tty12     Mar 29 10:00
deshpavn    ttyp1     Mar 29 10:38
munageka    ttyp2     Mar 28 16:55
deshpavn    ttyp3     Mar 29 10:49
```

Example 2:

```
deshpavn   ttyp3      Mar 29 10:49
```

```
$ who am i
```

Summary

iGATE
iTOPS for Business Outcomes

- **In this lesson, you have learnt:**
  - UNIX is multi-user, multiprocessor, time sharing operating system.
  - It uses hierarchical file system.
  - The UNIX system is functionally organized at three levels: Kernel, shell, tools and applications.

iLEARN

- 17 -

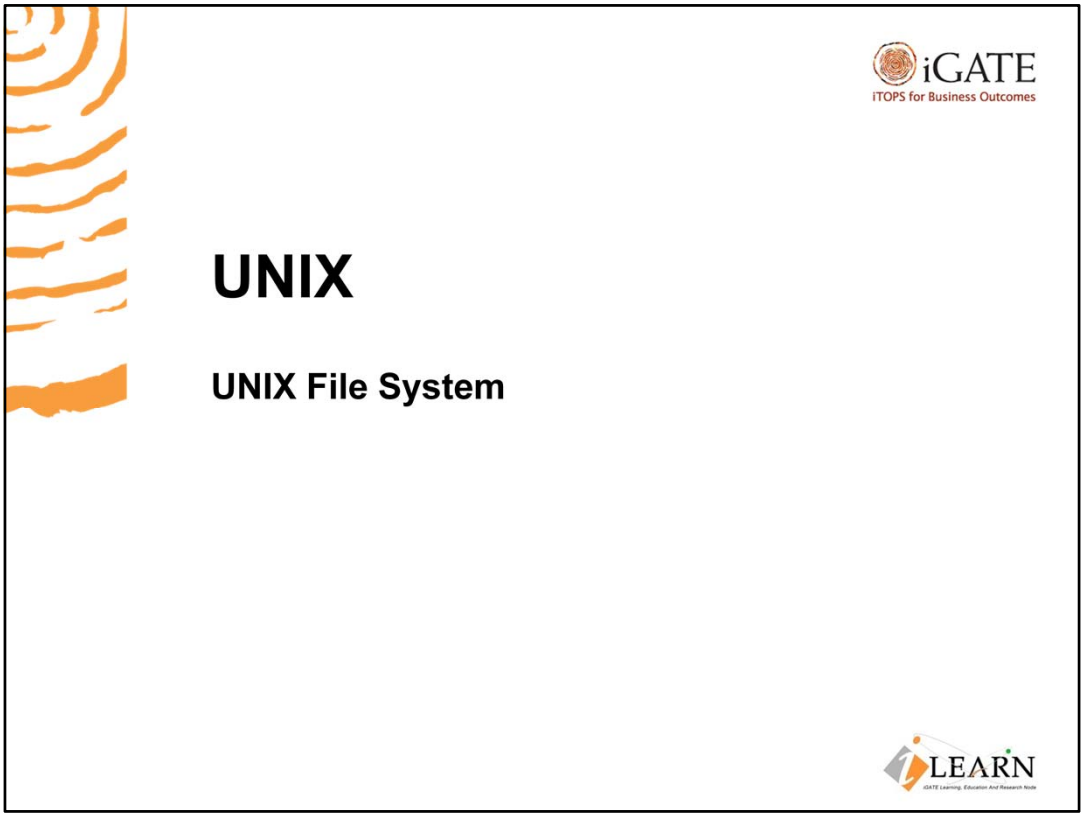
Review Questions

iGATE
iTOPS for Business Outcomes

- Question 1: ___ **controls system hardware.**
- Question 2: **The kernel interacts with the machine hardware, and the shell interacts with the User.**
  - True / False
- Question 3: ___ **command displays details of all users currently logged in.**

iLEARN
iGATE Learning, Education And Research Node

- 18 -


iGATE
iTOPS for Business Outcomes
UNIX
UNIX File System
iLEARN

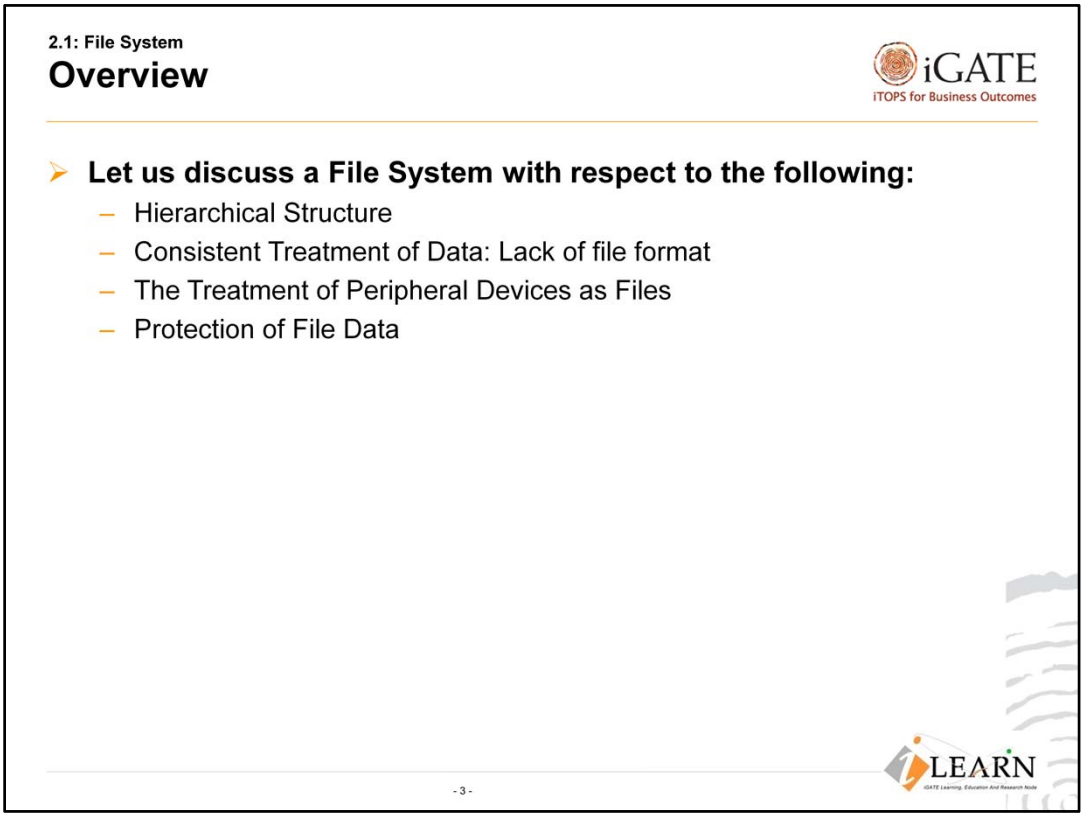
Lesson Objectives

iGATE
iTOPS for Business Outcomes

- **In this lesson, you will learn:**
  - UNIX File system
  - File types
  - File permissions
  - Commands related to file permission
    - mkdir, cd, cat etc…

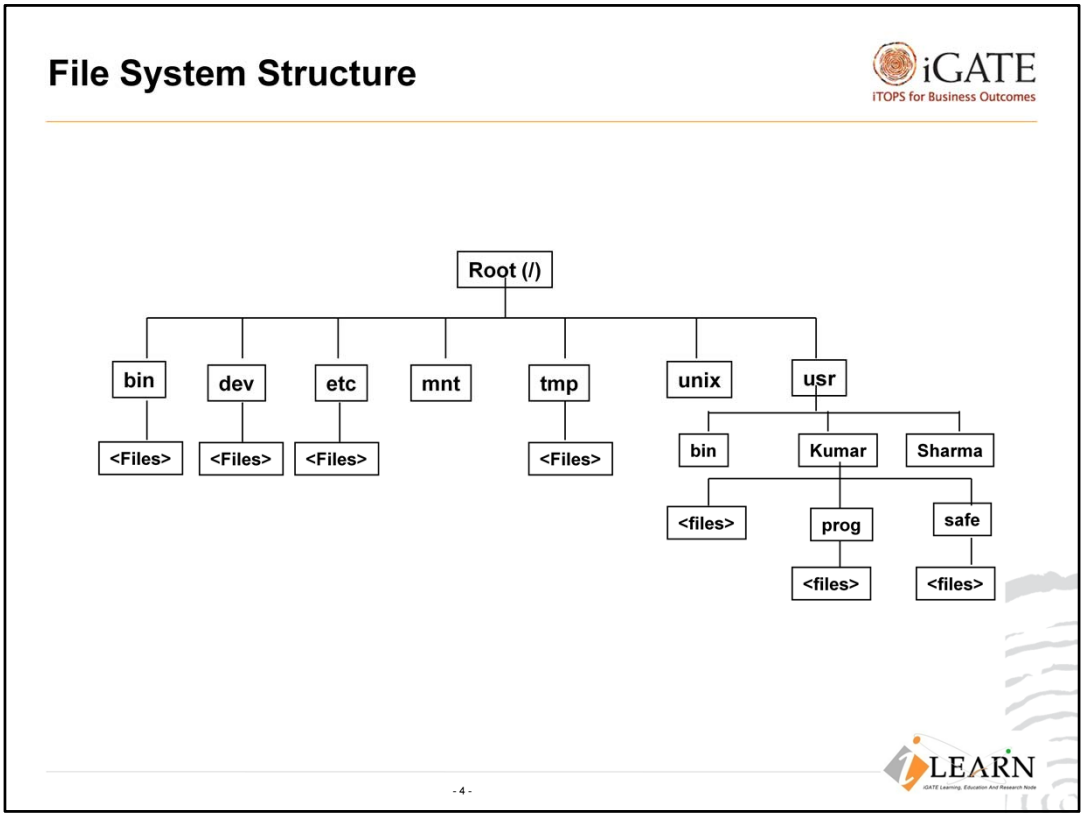
iLEARN

- 2 -



The UNIX File System:

UNIX works with a large number of files, which can belong to several different users. It becomes imperative for UNIX to organize files in a systematic fashion. The simple file system of UNIX is designed as an elaborate **Storage System** with separate compartment becoming available to store files. The system is widely adopted by different systems including DOS.



File System Structure:

The file system in UNIX is hierarchical. The **root**, a directory file represented by /, is at the top of the hierarchy and has several subdirectories (branches) under it. An example of a typical UNIX file structure is given in the above slide.

There can be more than one file system, each with its own root, in a single machine. The number of file systems cannot be less than the number of physical disks but can certainly exceed the latter. However, a file system cannot span multiple disks. If there are more than one file systems, there will always be one main file system.

File System Structure

iGATE iTOPS for Business Outcomes

- / bin **: commonly used UNIX Commands like who, ls**
- /usr/bin **: cat, wc etc. are stored here**
- /dev **: contains device files of all hardware devices**
- /etc **: contains those utilities mostly used by system administrator**
  - Example: passwd, chmod, chown

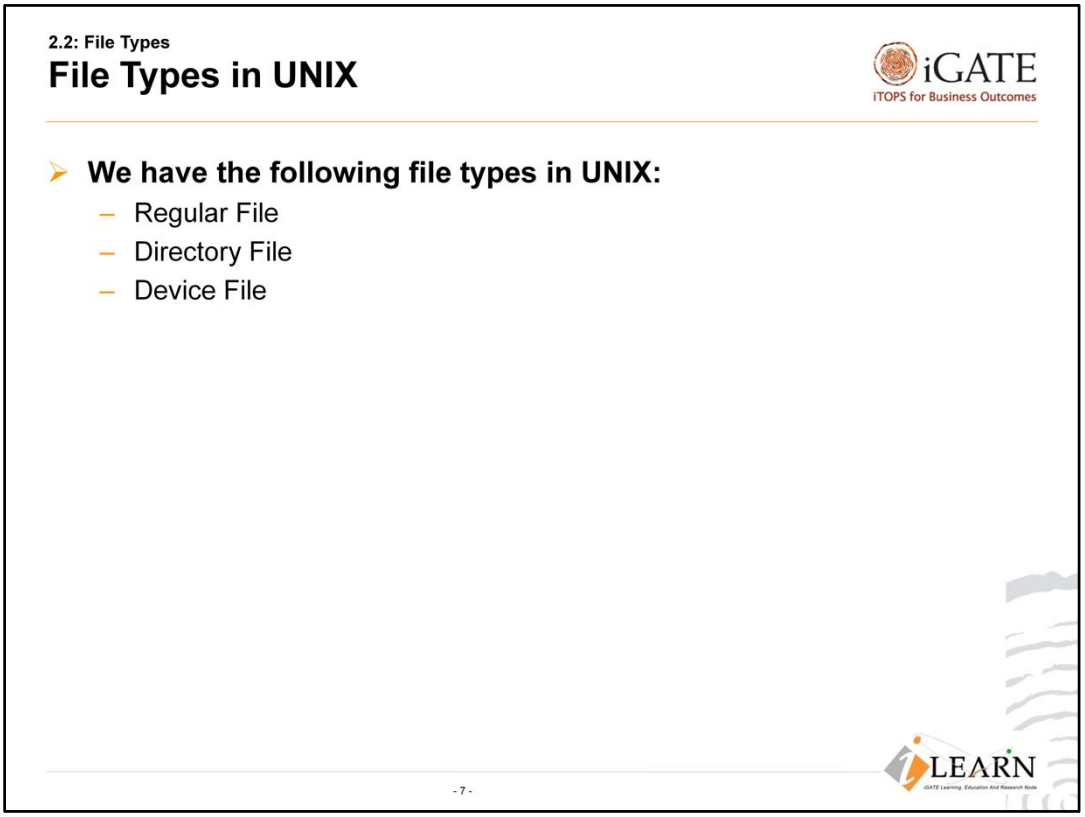
iLEARN iGATE Learning, Education And Research Node

File System

iGATE
iTOPS for Business Outcomes

- /tmp **: used by some UNIX utilities especially vi and by user to store temporary files**
- /usr **: contains all the files created by user, including login directory**
- /unix **: kernel**
- **Release V:**
  - It does not contain / bin.
  - It contains / home instead of /usr.

iLEARN
iGATE Learning, Education And Research Node



UNIX Files:

A UNIX file consists of a sequence of characters, and there are no restrictions on the file structure. The file consists only the actual bytes – and no extra information like its own size, attributes, or even an end of file marker. Extra information is stored in a structure called as **inode (index node)**.

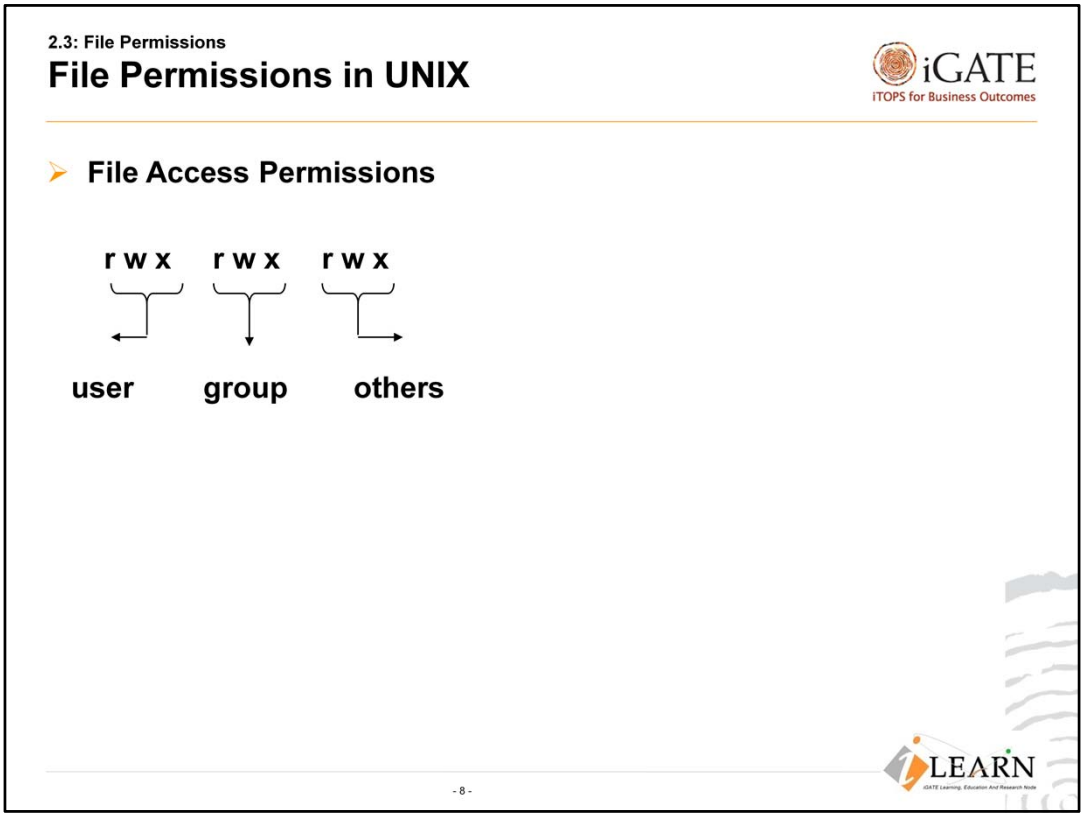
In UNIX, for every file, an inode is stored irrespective of its type.
Everything in UNIX is treated as a file, may be it is a user created or system file, a directory or a peripheral device. UNIX treats all the files in a consistent format. The lack of file formats and the consequent uniformity of files is of advantage to the programmer, as programmers do not need to worry about the file types. Further most of the standard programs work with any file.

Though everything is treated as a file, the significance of the file attributes is dependent on the categorization of files as Ordinary files, Directory Files and Device files.

Ordinary Files or Regular File :

These are also called as regular files. This is the “traditional” file, which can contain programs, data, object, and executable code, as well as all the UNIX programs and other user created files.

All text files belong to this category.



File Permissions:

Every file in UNIX has a set of permissions using which it is determined "who can do what" with the file. This is stored as part of **inode** information, and is useful for maintaining file security.

There are three categories of users: **Owner** (the user), **Group** (user is a member of which group), and **Others** (everybody else on the system). Access permissions of read (examining contents), write (changing contents) and execute (running program) are assigned to a file. These permissions are assigned to file owner, group and others.

It is the file permissions based on which reading, writing or executing a file is decided for a particular user. It helps in giving restricted access to a file. The commands to assign permissions and also to change the same are discussed later.

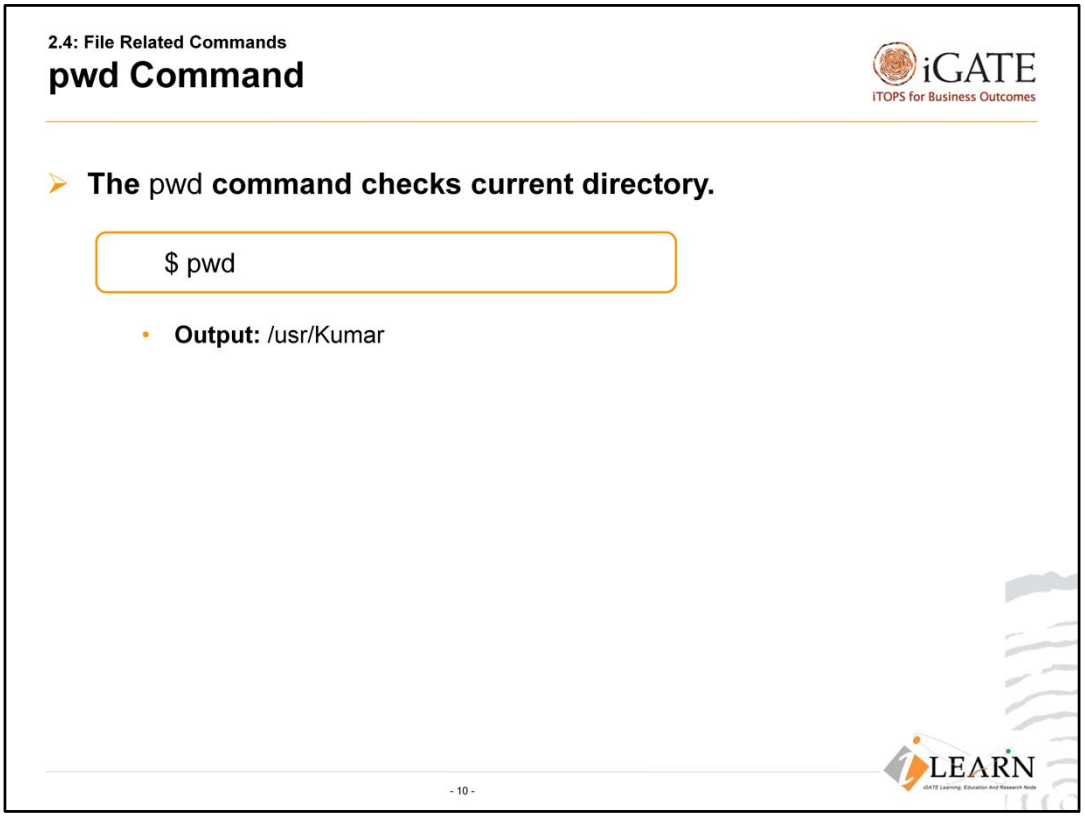
File Permissions in UNIX

iGATE
iTOPS for Business Outcomes

- **Permissions are associated with every file, and are useful for security.**
- **There are three categories of users:**
  - Owner (u)
  - Group (g)
  - Others (o)
- **There are three types of "access permissions":**
  - Read (r)
  - Write (w)
  - Execute (e)

iLEARN
iGATE Learning, Education And Research Node

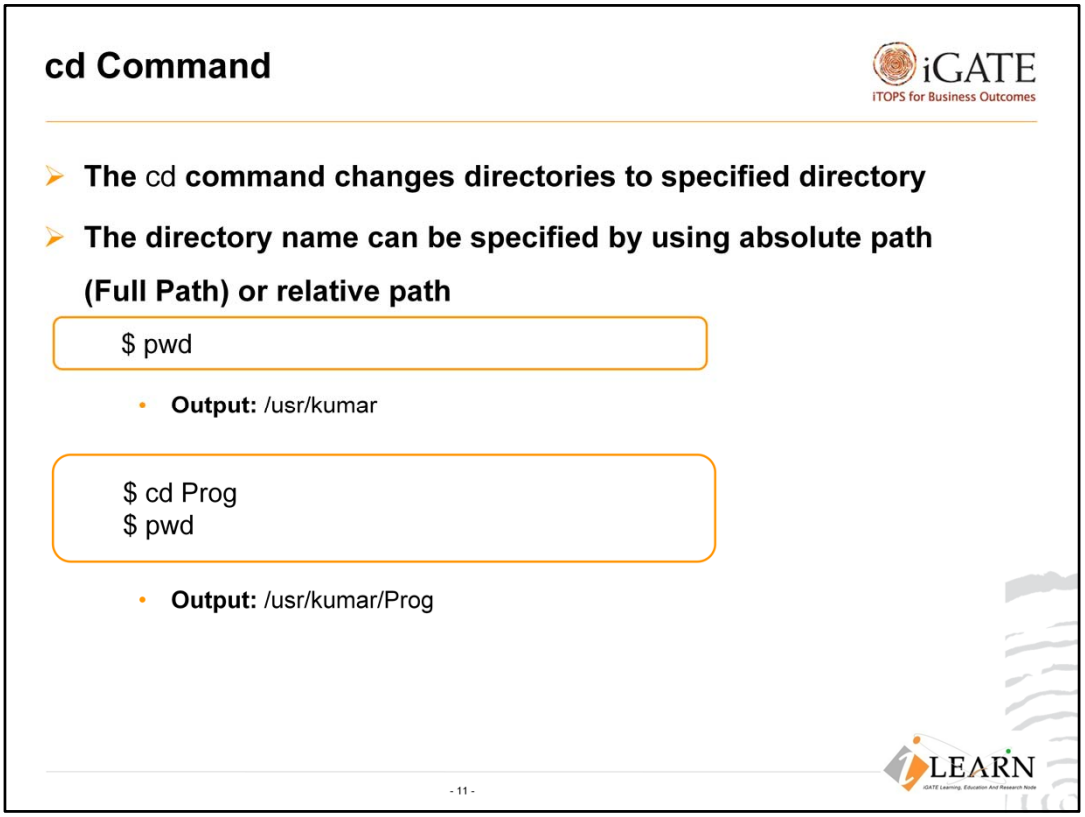
- 9 -



pwd Command:
The **pwd** command is used to print the name of the current working directory.

```
$ pwd
```

Output: /usr1/deshpavn



cd (change directory) Command:
The cd command is used to change the current directory to the directory specified as the argument to the command. The argument can contain absolute as well as relative paths.

```
$ cd deshpavn
$ pwd
```

Output: /usr1/deshpavn

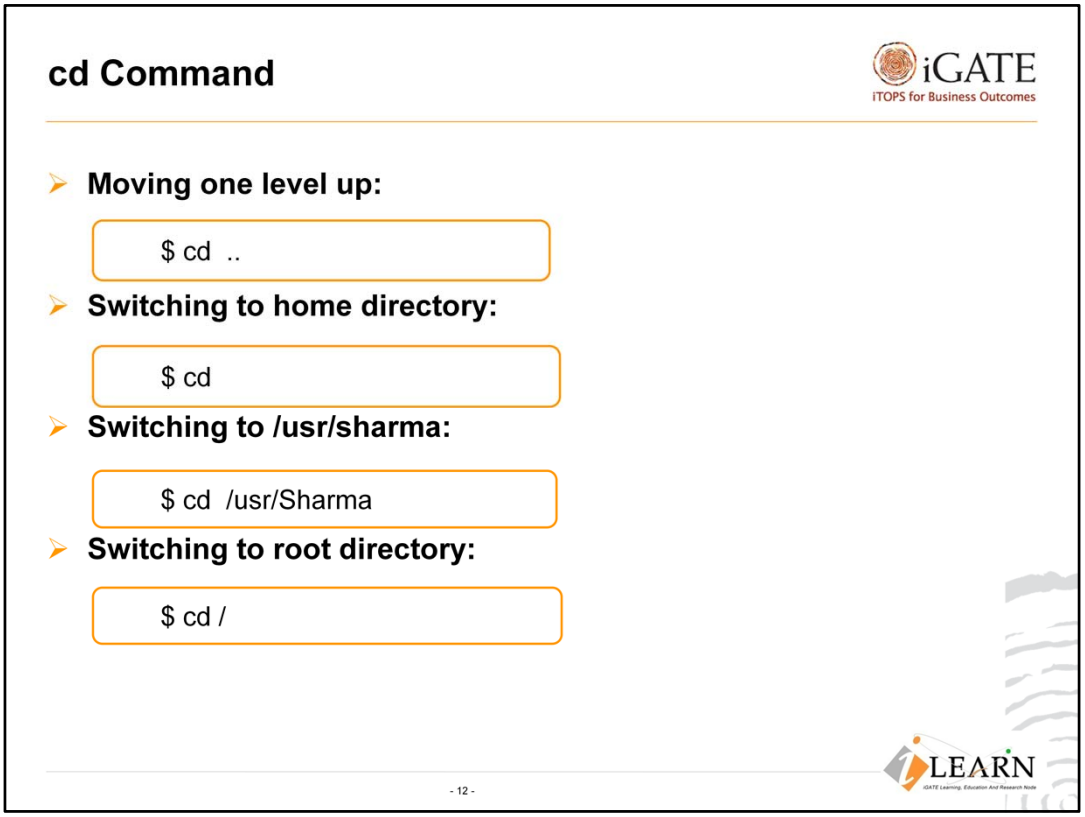
Absolute Path and Relative Path
To locate the file or directory which is not in current working directory full path can be given.
If the specified path starts from root directory i.e from '/' the it is called as **absolute path or full path**. Otherwise it is called as **relative path** because the given path is relative to the current working directory

e.g If current working directory is /usr, then to change the directory to prog
Use following command
$pwd
/usr
$cd **kumar/prog**

In the above example, specified path to prog directory is not starting with '/'. It is relative path from /usr directory(current working directory).

The same command can be given as
$cd /usr/kumar/prog

In above example, since the given path is starting from root directory i.e '/' hence this is called as absolute path or full path.



cd (change directory) Command:
To move one level up in the file system, the command is used as follows:

```
$ pwd
```

Output: /usr1/deshpavn

```
$ cd ..
$ pwd
```

Output: /usr1

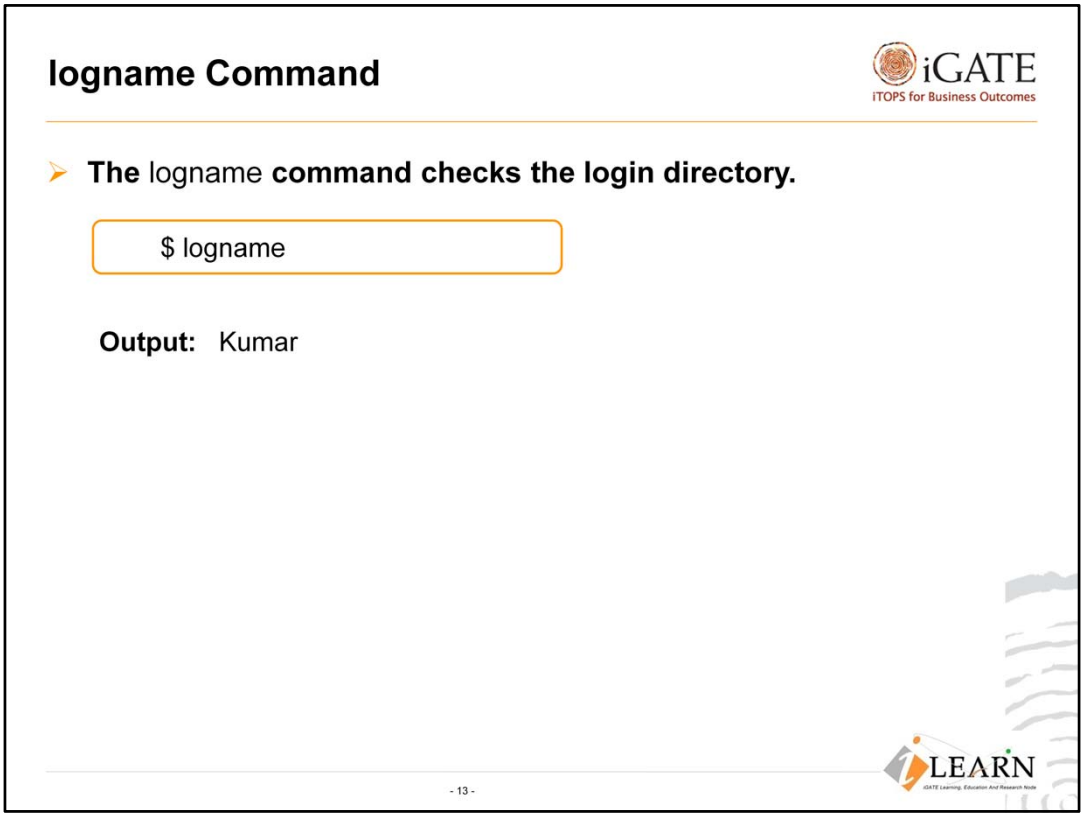
When the command is used without any parameters, it switches to home directory as the current directory.

Output: /usr1

```
$ pwd
```

Output: /usr1/deshpavn

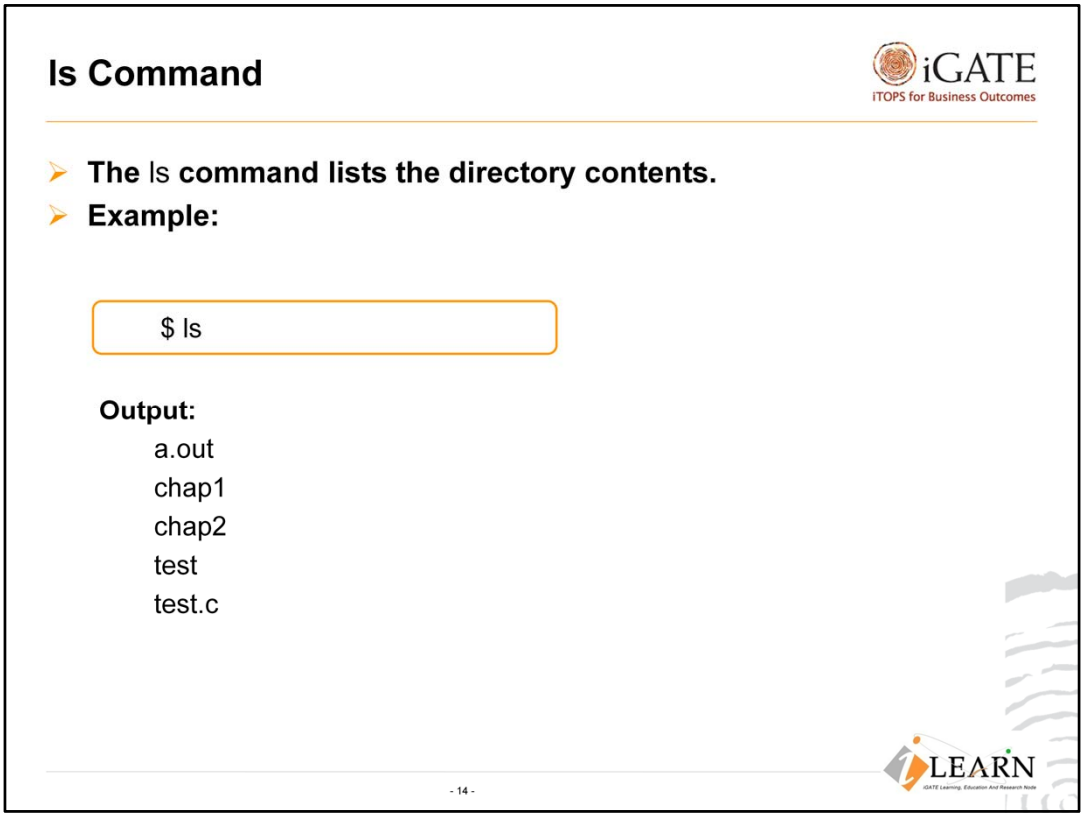
```
$ cd
$ pwd
```



logname Command - Checking Login Directory :
The **logname** command can be used to display the name of login directory, irrespective of what is the current working directory.

```
$ logname
```

Output: deshpavn



ls Command – Listing directory contents:

The **ls** command is used to list all the contents of the specified directory (in case no argument is specified, current directory is assumed).

```
$ ls
```

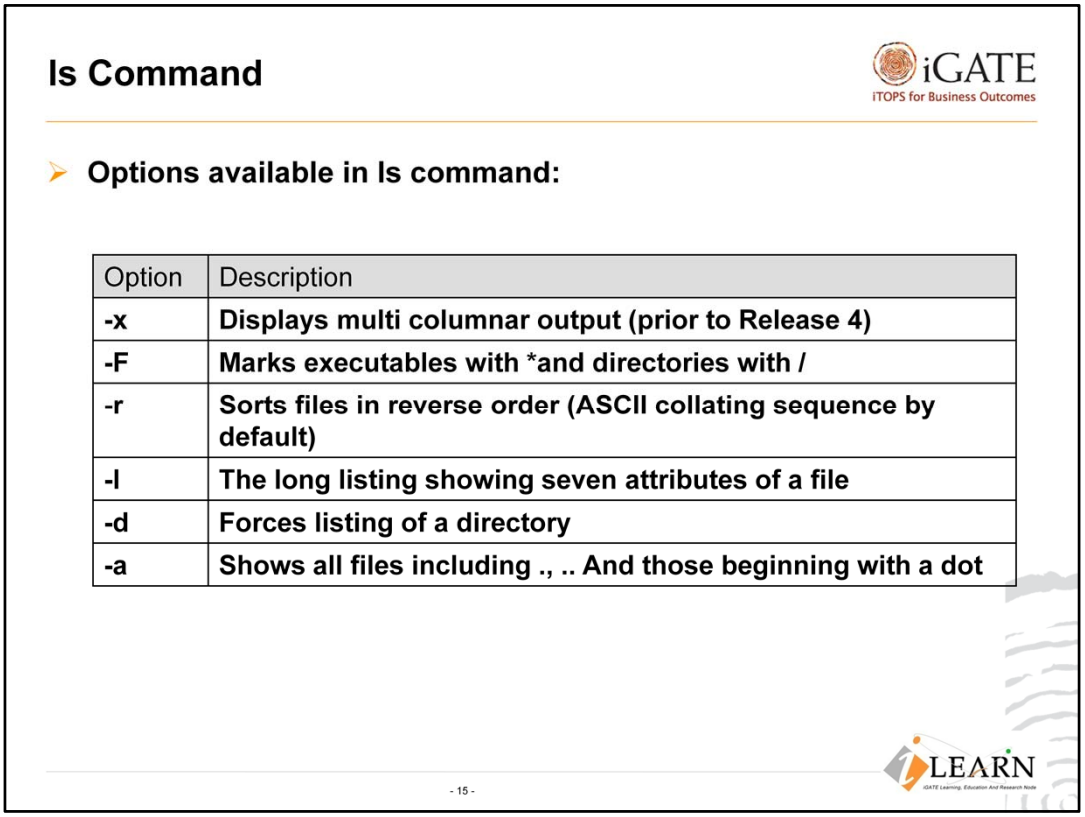
Output:

file1.txt
file2.txt
file3.txt
Mail

```
$ ls
```

Output:

newfile.txt



ls Command – Listing directory contents (contd.):

The ls command comes with several options, which are listed in the table. Many of these options can be combined to get relevant output. The command can also be used with more than one file name that is specified.

ls Command

iGATE
iTOPS for Business Outcomes

- **Options available in ls command:**

Option	Description
-t	**Sorts files by modification time**
-R	**Recursive listing of all files in sub-directories**
-u	**Sorts files by access time (when used with the –t option)**
-i	**Shows i-node number of a file**
-s	**Displays number of blocks used by a file**

iLEARN

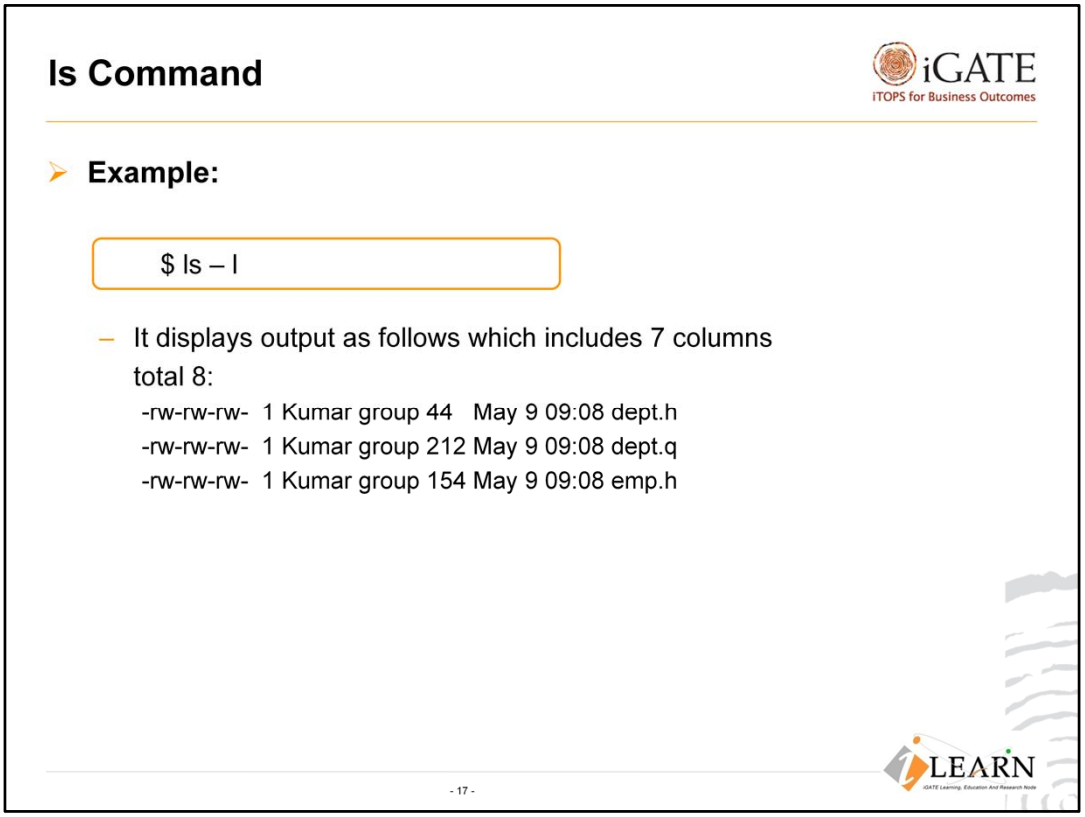
- 16 -

ls Command – Listing directory contents (contd.):
Some of the examples are considered here:
Example 1: To produce multiple column output (-x):

```
$ ls -x
```

Output: file1.txt file2.txt file3.txt mail

Example 2: To identify directories and executable files (-F):
Here two tags, that is ***** and **/**, are used to indicate executable file and a directory respectively.



ls Command – Listing directory contents (contd.):
Example 3: To get an informative listing i.e. long listing (-l)

```
$ ls -l
```

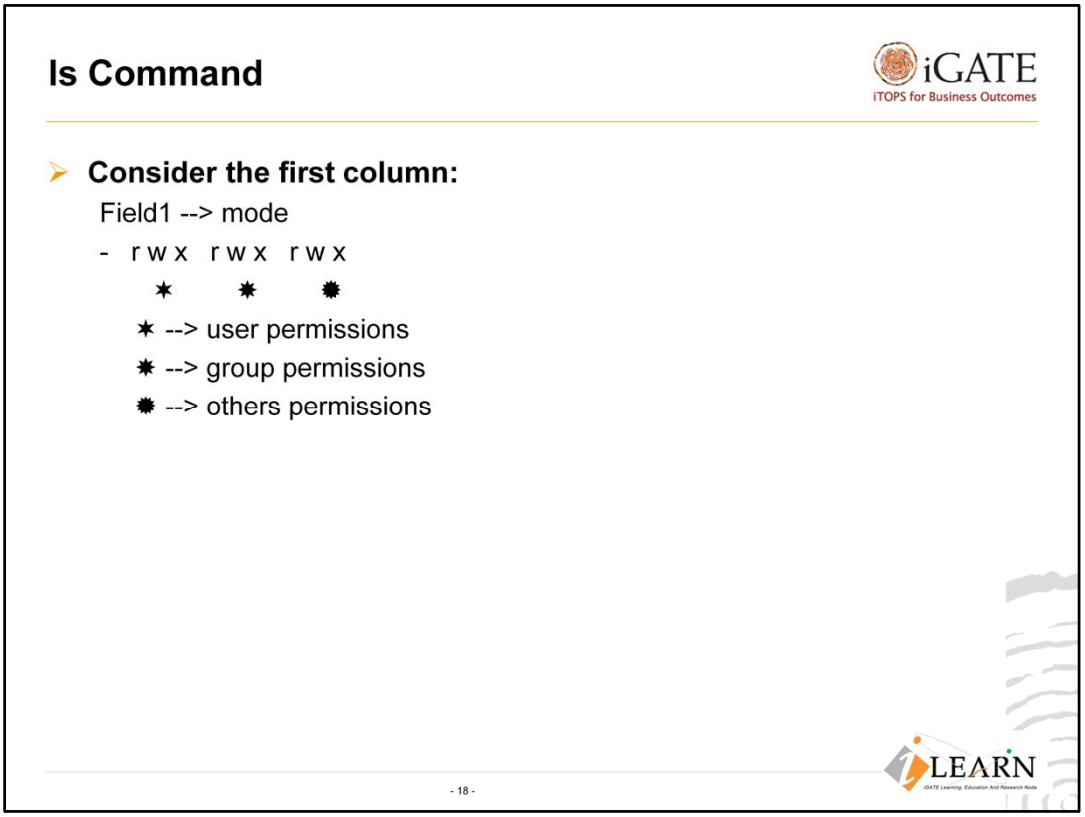
```
total 12
-rw-r--r--   1 deshpavn group        60 Mar 29 10:43 file1.txt
-rw-r--r--   1 deshpavn group        61 Mar 29 10:44 file2.txt
-rw-r--r--   1 deshpavn group        60 Mar 29 10:44 file3.txt
drwx------   2 deshpavn group       512 Mar 14 10:00 mail
drwxr-xr-x   2 deshpavn group       512 Mar 29 10:46 testdir1
drwxr-xr-x   2 deshpavn group       512 Mar 29 10:47 testdir2
```

Seven attributes are listed which are explained in further slides in order as they appear in listing.

Type and Permissions associated with file:

The first bit "**d**" indicates that file is a **directory file**, while "**–**" indicates that file is a **regular file**. In case of device files, one would normally find bit "**c**" for **character based device** and bit "**b**" for **block based devices**.

File permissions are the read, write, and execute permissions set for owner, group, and others. Significance of these permissions with respect to directories are discussed in another example.



ls Command – Listing directory contents (contd.):

Permissions are interpreted as follows:

For a directory: The **"r" permission** implies that one can find out the contents of the directory using commands like ls. The **"w" permission** implies that it is possible to create and delete files in this directory. It is possible to remove even the files that are write-protected. The **"x" permission** means search rather than execute. That is, it implies that the directory can be searched for a file. Also, when 'x' permission is set for a directory, one can do change dir to that directory and not otherwise.

Example: Using "r- -" will ensure that users can see the directory contents using ls, but cannot really use the contents.

To list all hidden files (-a):

The ls command does not list all the hidden files unless –a attribute is used.

To display inode information and number of blocks used for file:

```
$ ls -i
```

```
658 file1.txt
3952 file2.txt
3956 file3.txt
434 mail
3957 testdir1
3960 testdir2
```

```
$ ls -i testdir1
```

```
658 fileln1.txt
```

ls Command

iGATE
iTOPS for Business Outcomes

- **File type**
  - 1 st character represents file type:

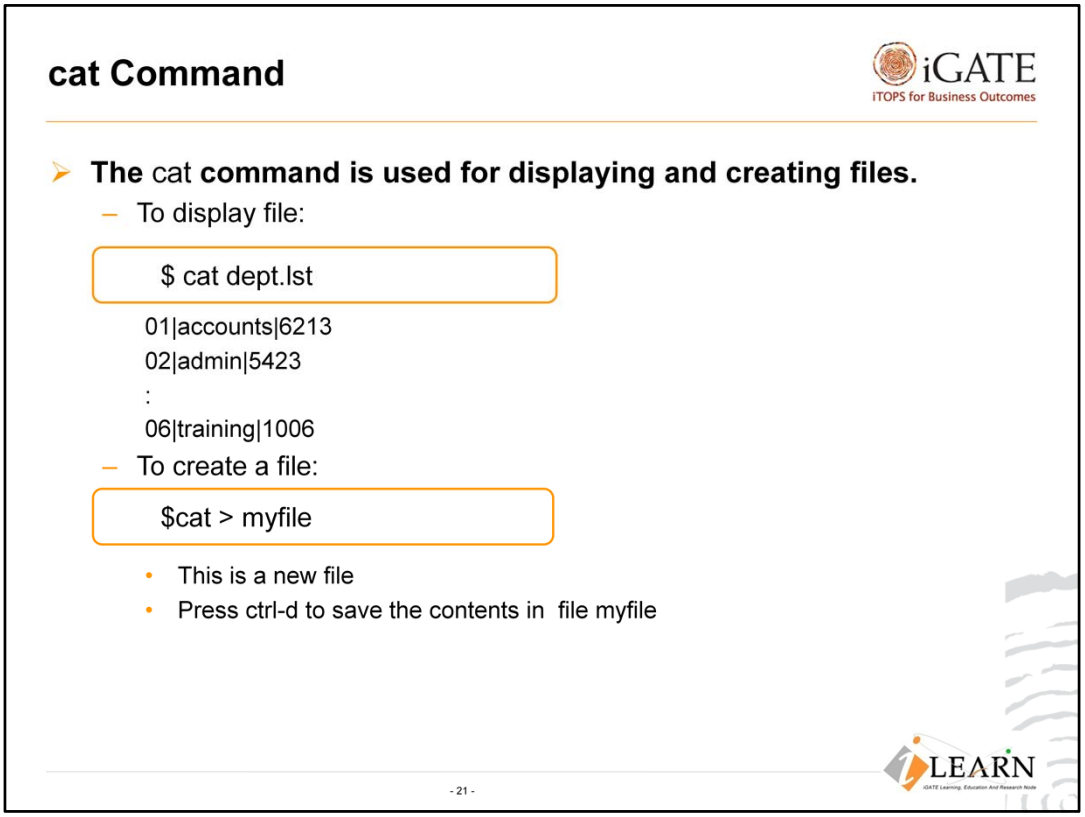
    - r w x r w x r w x

    - - --> regular file
    - d --> directory file
    - c --> character - read
    - b --> block read

iLEARN
iGATE Learning, Education And Research Node

ls Command

- Field2 : indicates number of links
- Field3 : File owner id
- Field4 : Group id
- Field5 : File size in bytes
- Field6 : Date/time last altered
- Field7 : Filename



cat Command (Displaying and Creating files):

The **cat** command can be used to display one or more files (if there is more than one file, then contents of the remaining immediately follow without any header information). To control the scrolling of text on screen, you can use **<Control-s>** and **<Ctrl-q>**.

```
$ cat file1.txt
```

Output: This is a sample text file.

This is the first file created.

```
$ cat file1.txt file2.txt
```

Output: This is a sample text file.
This is the first file created.
This is a sample text file.
This is the second file created.

The command can also be used to create a file as described below. The meaning and significance of the ">" symbol will be discussed later.

```
$ cat > newfile.txt
```

Output: This is a new file being created.
Use the redirections to save contents into a file.
To indicate end of input, press <Ctrl-d>

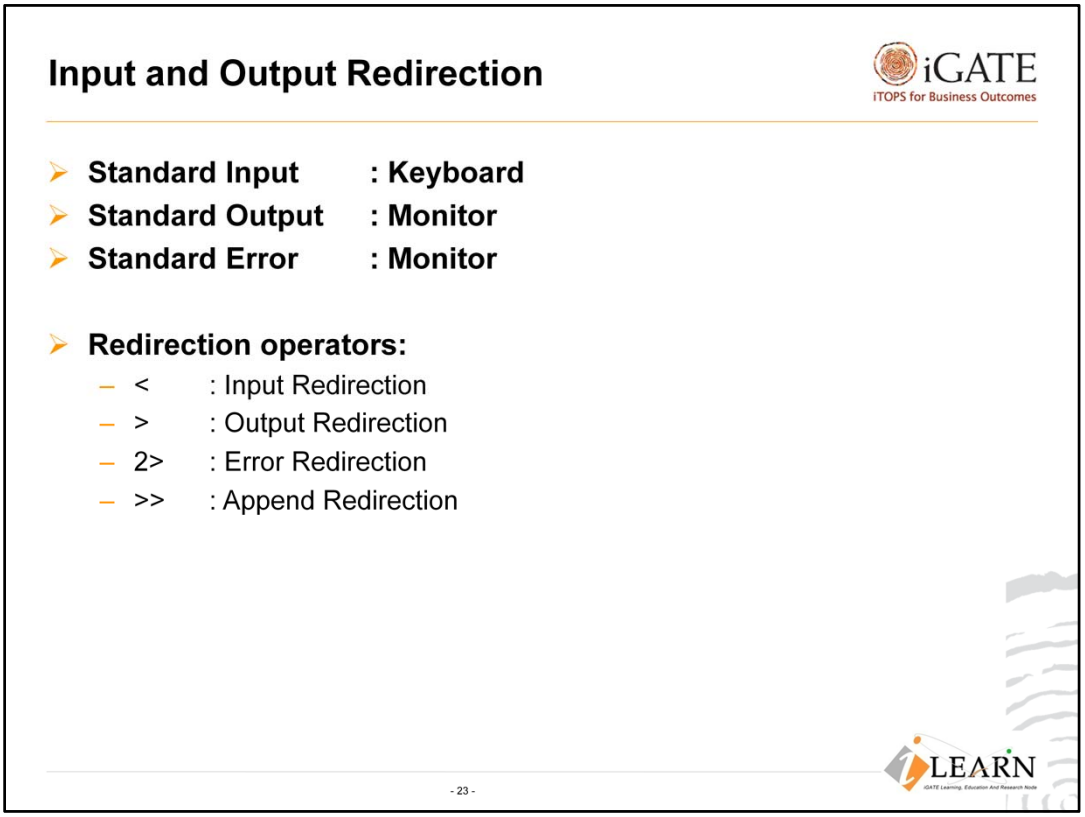
cat Command

iGATE
iTOPS for Business Outcomes

- **The** cat **command can be used to display contents of more than one file.**
  - It displays contents of chap2 immediately after displaying chap1.

```
$ cat chap1 chap2
```

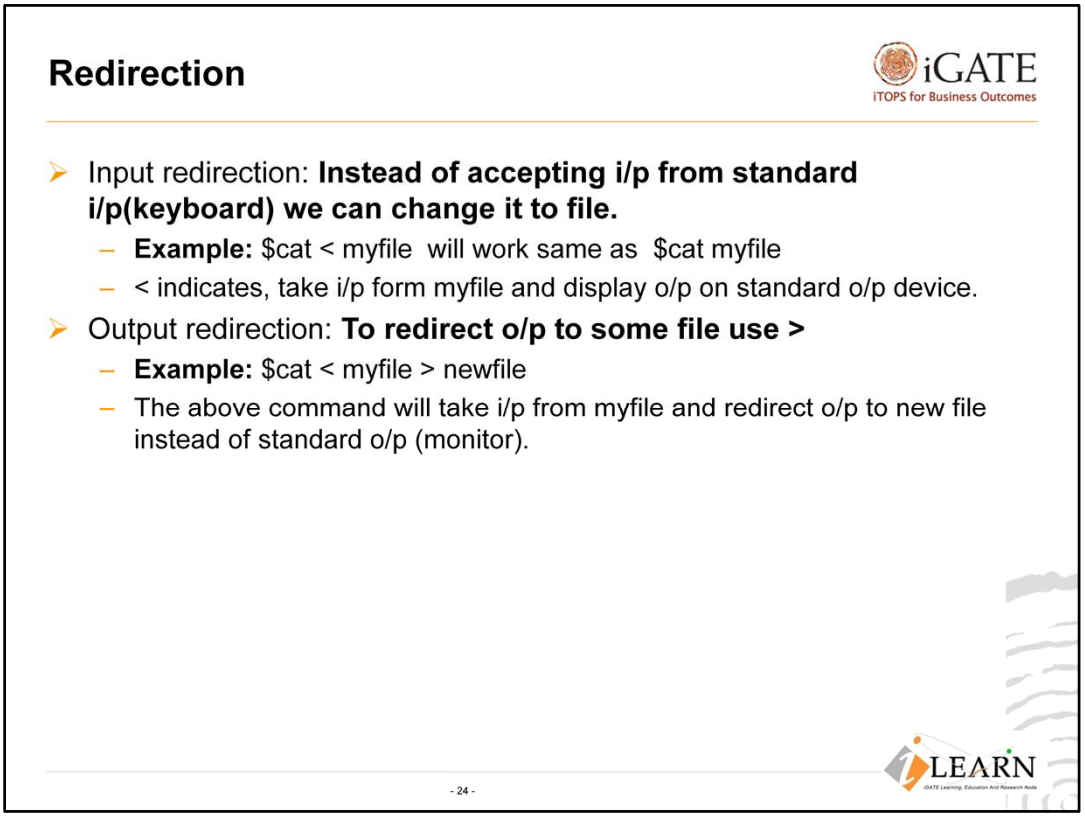

iLEARN



Input and Output Redirection:

Many commands work with **character streams**. The default is the keyboard for input (standard input, file number 0), and terminal for the output (standard output, file number 1). In case of any errors, the system messages get written to standard error (file number 2), which defaults to a terminal.

UNIX treats each of these streams as files, and these files are available to every command executed by the shell. It is the shell's responsibility to assign sources and destinations for a command. The shell can also replace any of the standard files by a physical file, which it does with the help of **metacharacters** for redirection.



Redirection:

In order to take the input from a file (instead of standard i/p), the character **<** is used.

Example: $ cat < file1.txt

This is example of i/p redirection.

To redirect the output to a file, **>** is used. If outfile does not exist, it is first created; otherwise, the contents are overwritten. In order to append output to the existing contents, **>>** is used.

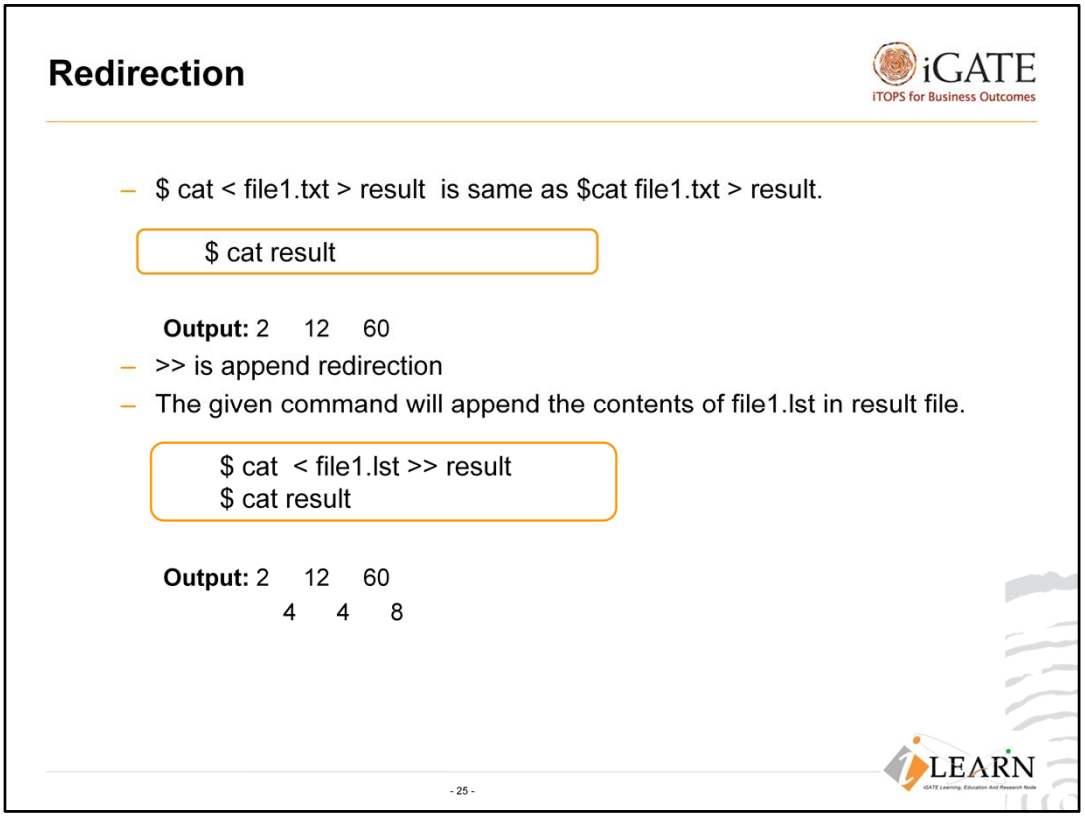
In the following command, the **cat** command will take i/p from file **file1.txt** and send the o/p to **result file**. If result file exists, then contents will be overwritten, otherwise new result file will be created.

Example:

$ cat < file1.txt > result is same as $cat file1.txt > result

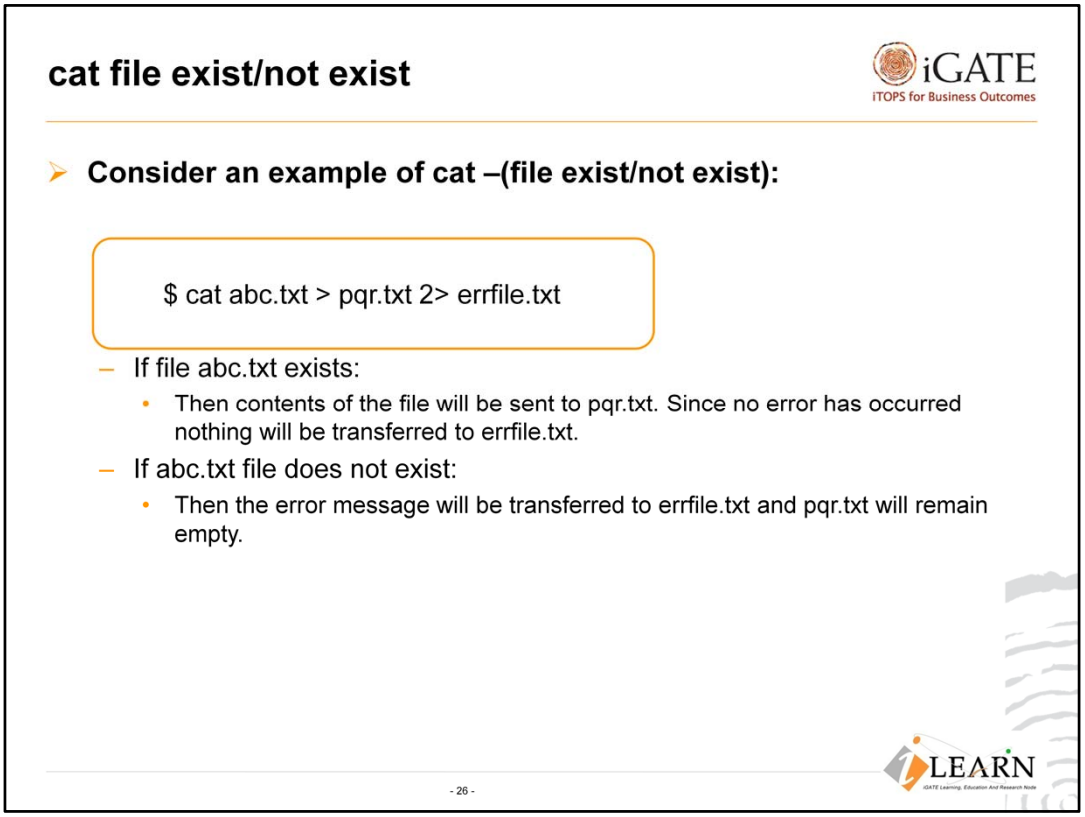
$ cat result

2 12 60



Note:
>> - is append redirection
If you want to retain previous contents of a file and append new contents to a file then use append redirection operator.

```
$ cat  < file1.lst >> result
$ cat result
    2    12    60
    4     4     8
```



cat file ext/not exit:
As shown in the above slide, it is possible to combine redirection operators on a single command line. The order of the redirection symbols does not affect the working of the command.

To redirect the standard error, 2> is used.

```
$ cat xyzfile
```

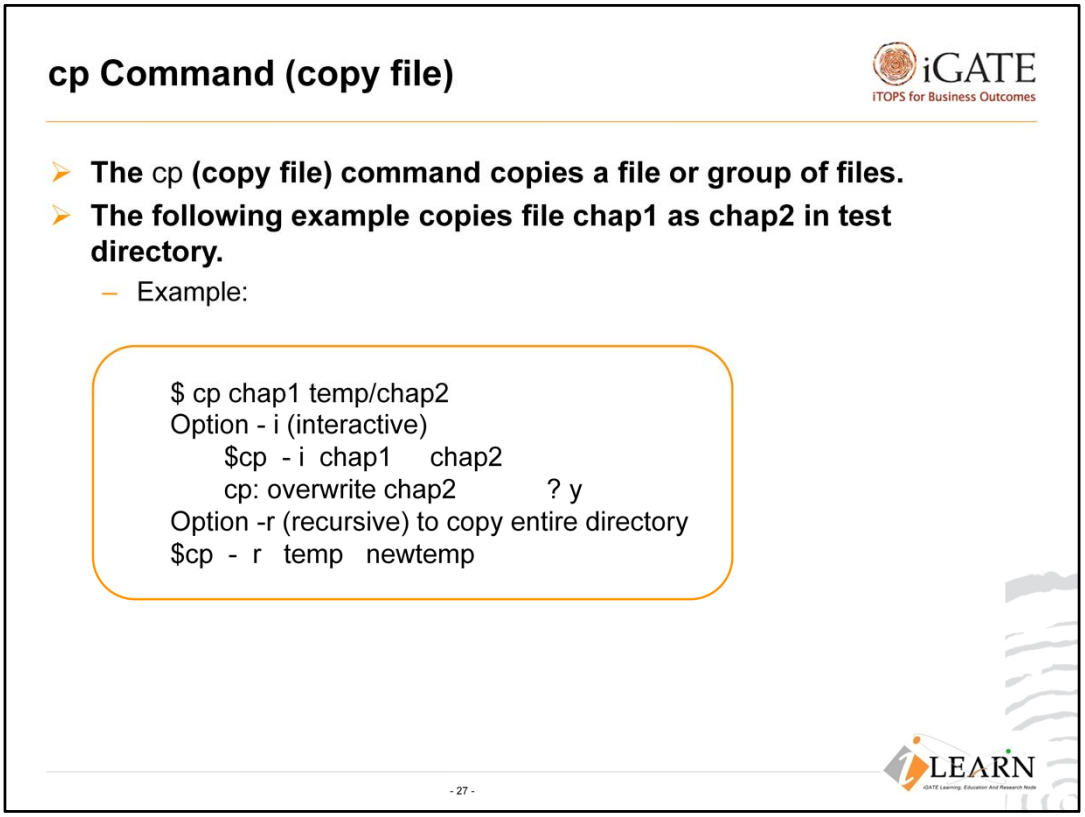
cat: cannot open xyzfile: No such file or directory (error 2)

```
$ cat xyzfile 2> errfile
$ cat errfile
```

cat: cannot open xyzfile: No such file or directory (error 2)

```
$ cat xxfile 2>> errfile
$ cat errfile
```

cat: cannot open xyzfile: No such file or directory (error 2)
cat: cannot open xxfile: No such file or directory (error 2)



cp Command (Copying Files):
The copy command copies a file or groups of files.

```
$ cp newfile.txt anotherfile.txt
$ cp newfile.txt testdir1/nfile1.txt
```

Copy all files with extension txt:

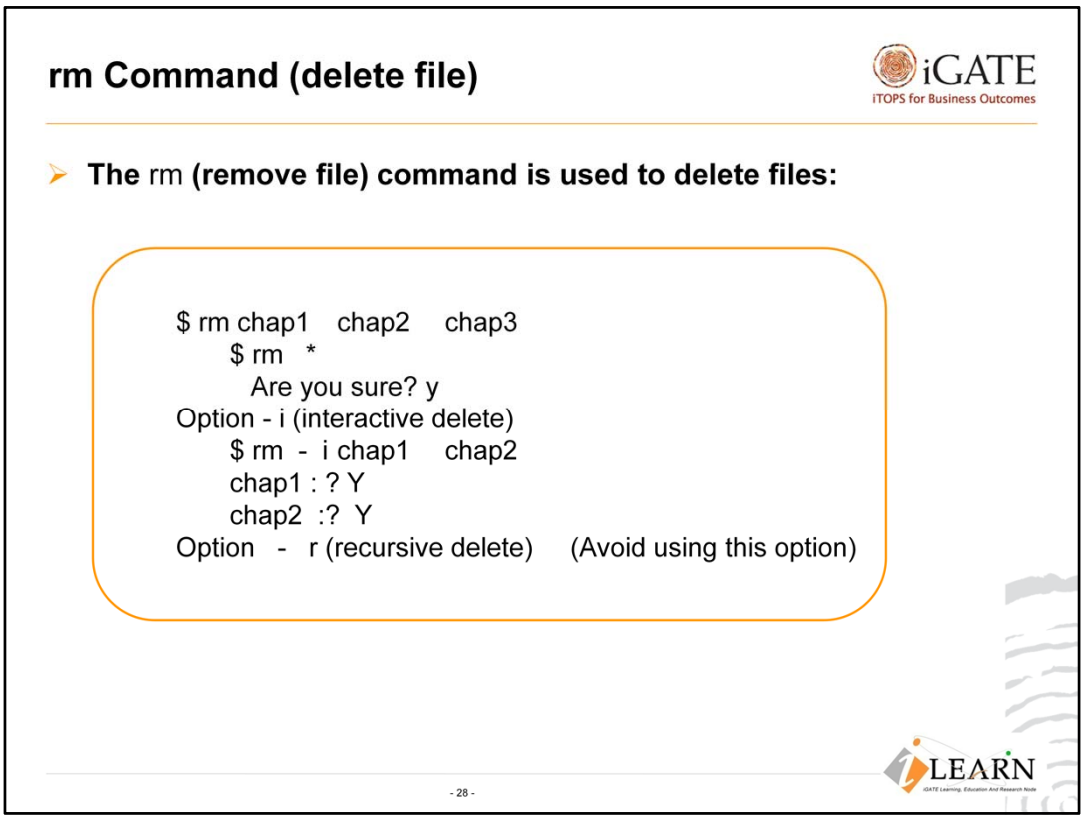
```
$ cp *.txt *.dat
```

All files with extension will be copied with same name but extension will change to dat.
Example:
abc.txt will be copied as abc.dat
pqr.txt will be copied as pqr.dat
Files cannot be copied if they are read protected, or if destination file/directory is write protected.

Copying file using redirection:

```
$cat newfile.txt > anotherfile.txt
```

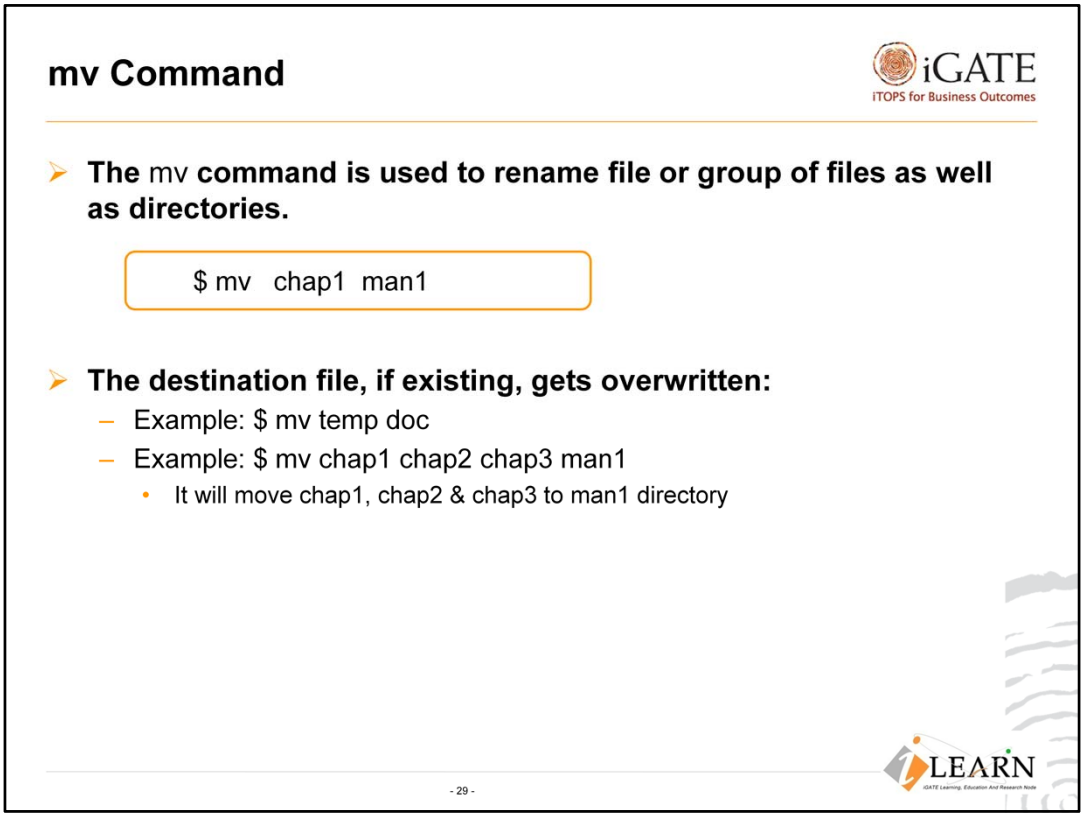
The result of above command is same as $ cp newfile.txt anotherfile.txt



rm Command (Removing Files) :

This command is used to delete files. There are options for **interactive (-i) delete** and **recursive (-r) delete**. The **–r** option will delete files from subfolders also.

```
$ rm newfile.txt
Option - r (recursive delete)
$ rm -r *
(Warning: Pl. do not use this option)
```

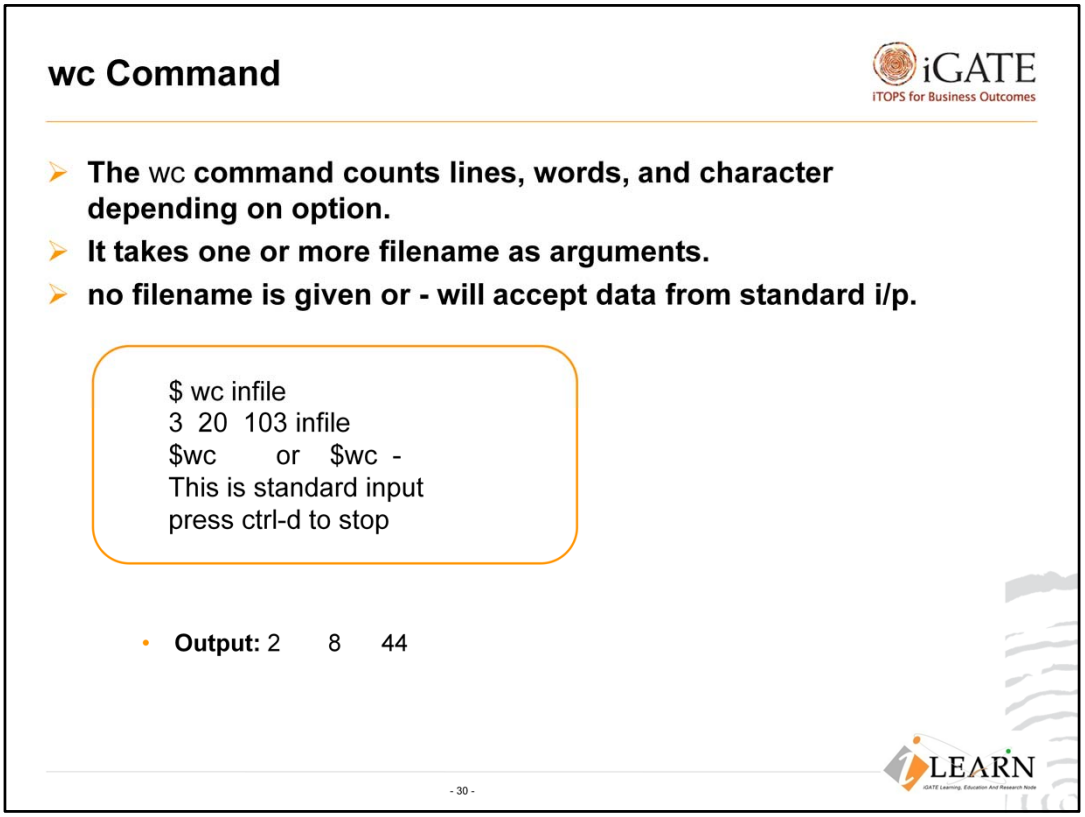


mv Command (Renaming Files):

Files as well as directories (belonging to the same parent) can be renamed using this command.

```
$ mv anotherfile.txt newfile.txt
$ mv testdir1 testdir2
$ ls -l
```

```
total 12
-rw-r--r--   3 deshpavn group        60 Mar 29 10:43 file1.txt
-rw-r--r--   1 deshpavn group        61 Mar 29 10:44 file2.txt
-rw-r--r--   1 deshpavn group        60 Mar 29 10:44 file3.txt
drwx------   2 deshpavn group       512 Mar 14 10:00 mail
-rw-r--r--   1 deshpavn group       126 Mar 29 10:54 newfile.txt
drwxr-xr-x   3 deshpavn group       512 Mar 29 10:58 testdir2
```



Line, word, and character counting – wc

This command can be used to count the number of lines (-l option),words (-w option), or characters (-c option) for one or more files. If we specify multiple files, then the list of files should be separated by space. If no file name is specified, then it will accept data from standard i/p, that is from the keyboard.

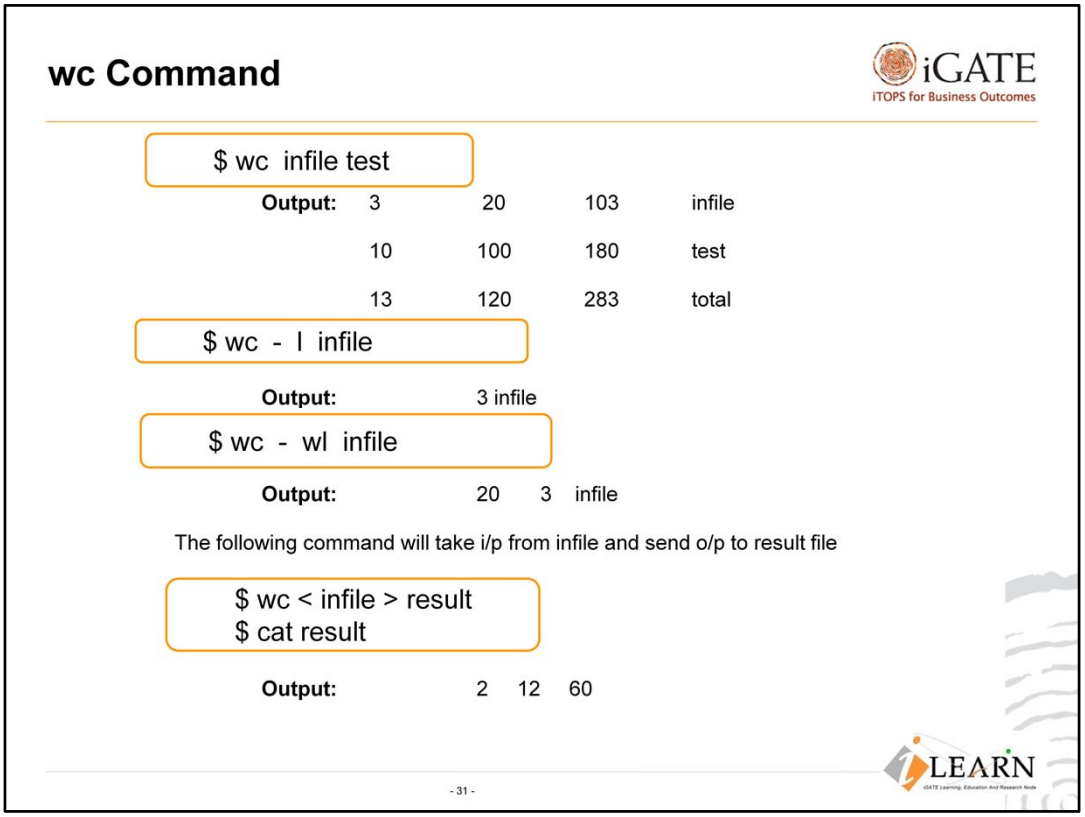
Example:

```
$ wc file1.txt
```

2 12 60 file1.txt

```
$ wc -lw file1.txt
```

212 file1.txt



wc Command with Redirection:

In order to take the input from a file (instead of standard i/p), the character < is used.

```
$ wc < file1.txt
```

Output: 2 12 60

To redirect the output to a file, > is used. If outfile does not exist, it is first created; otherwise, the contents are overwritten. In order to append output to the existing contents, >> is used.

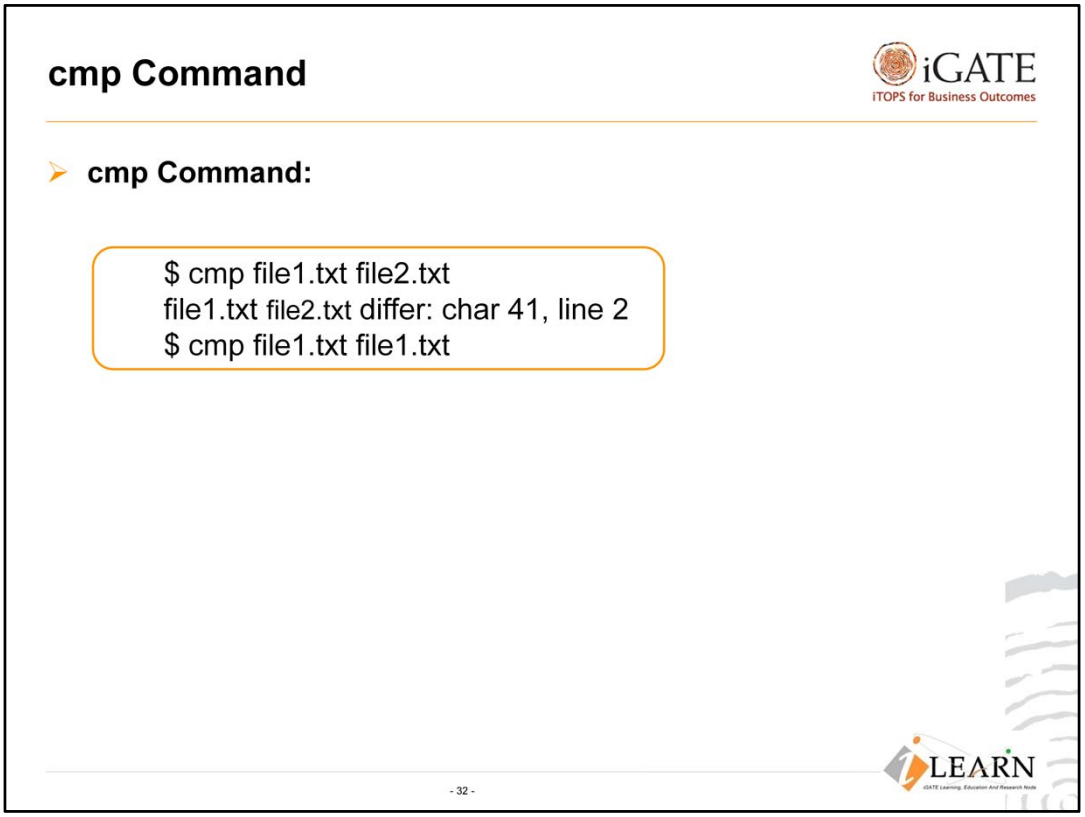
In following command, wc will take i/p from file file1.txt and send the o/p to result file. If result file exists, contents will be overwritten, otherwise new result file will be created.

```
$ wc < file1.txt > result
$ cat result
```

Output: 2 12 60

```
$ wc < cfile1.lst >> result
$ cat result
```

Output: 2 12 60
4 4 8

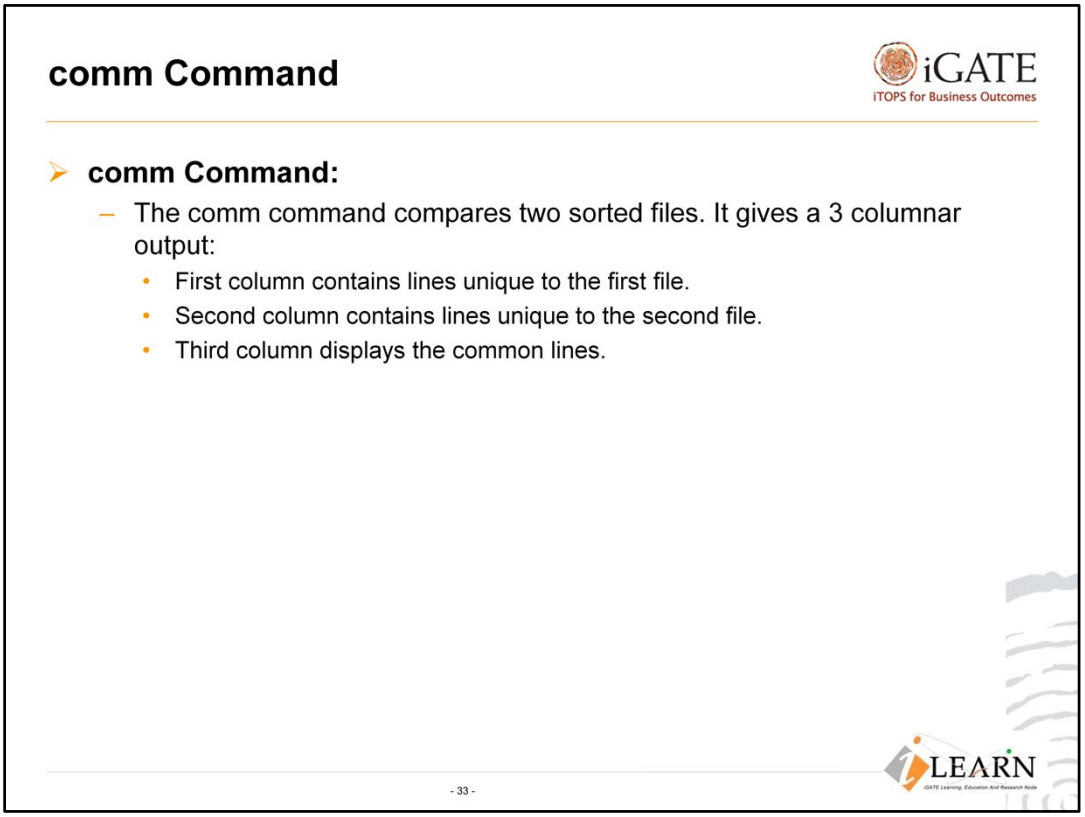


cmp Command (Comparing Files):

Using the cmp Command:

The cmp command can be used to compare if two files are matching or not. The comparison is done on a byte by byte basis. In case files are identical, no message is displayed. Otherwise, locations of mismatches are echoed.

```
$ cmp file1.txt file2.txt
        file1.txt file2.txt differ: char 41, line 2
  $ cmp file1.txt file1.txt
```

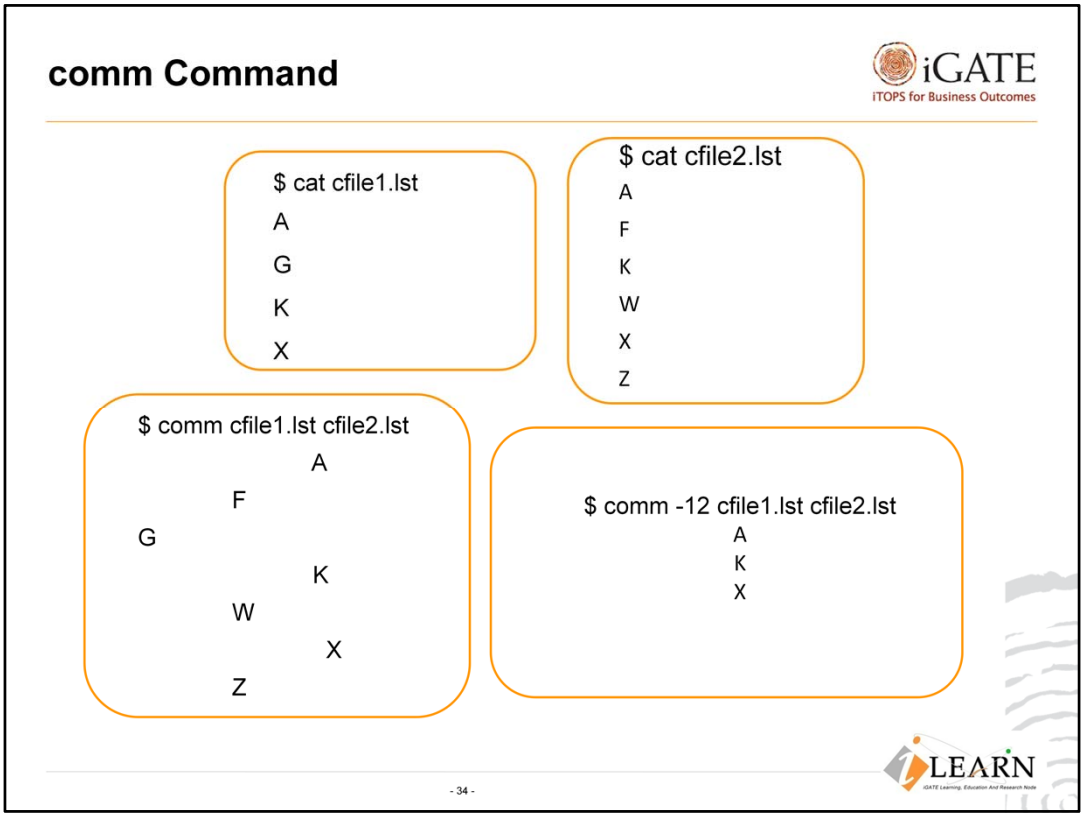


comm Commands (Comparing Files):

The comm command compares two sorted files. It gives a 3 columnar output.
First column contains lines unique to the first file.

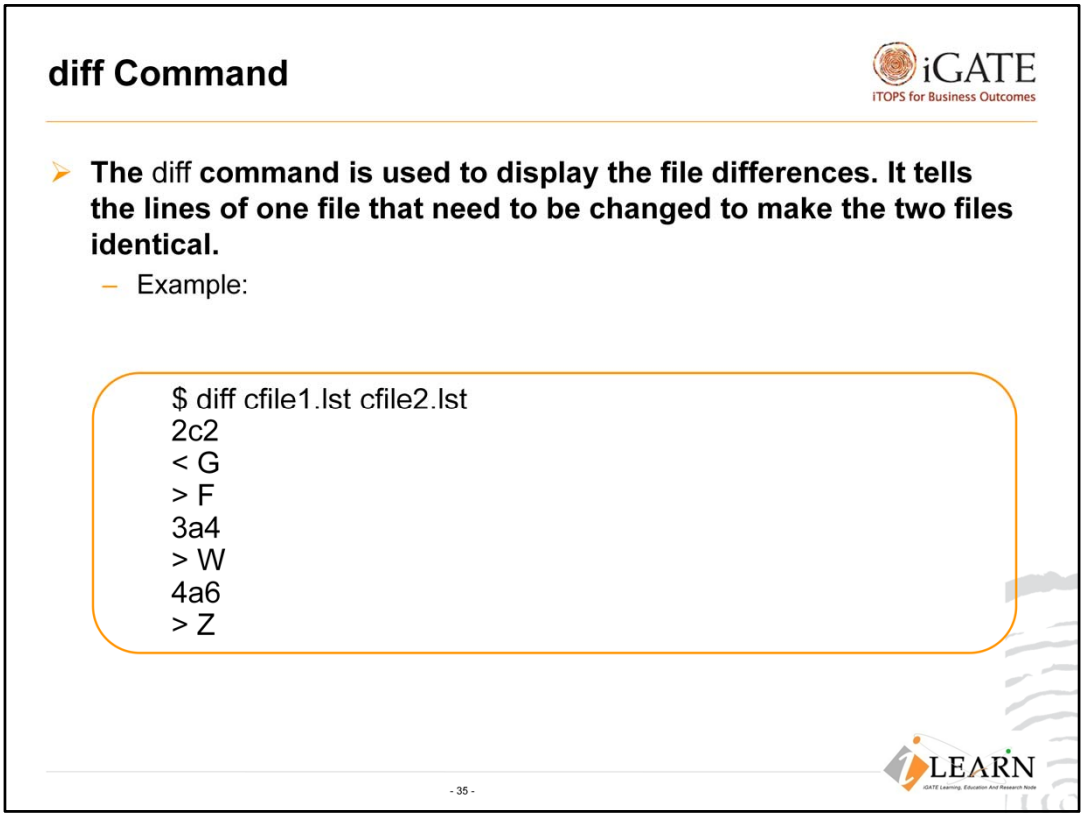
Second column contains lines unique to the second file.
Third column displays the common lines.

Selective column output can be obtained by using options –1, -2 or –3. It would drop the column(s) specified from the output.



In above example
$ comm cfile1.lst cfile2.lst
This command is showing output in 3 columns
First column display lines unique to the cfile1.lst.
Second column display lines unique to the cfile2.lst.
Third column displays the lines common in both files.

$ comm -12 cfile1.lst cfile2.lst
This command will hide first and second column and display only 3 rd column i.e lines common in both files.

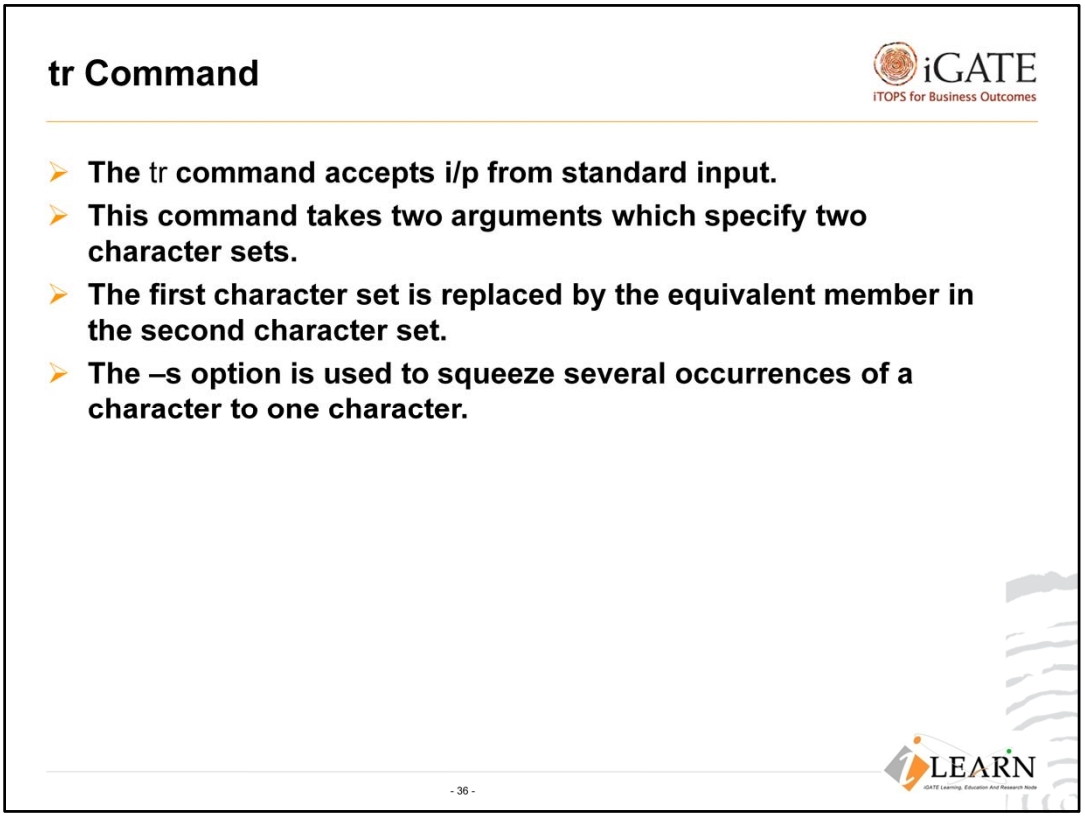


diff Command:

The file differences can be displayed using the diff command. It tells which lines of one file need to be changed to make the two files identical.

Example: Using the diff command

```
$ diff cfile1.lst cfile2.lst
2c2          change line 2 of first file which is line 2 in 2nd file
< G          replacing this line
---          with
> F          this line
3a4          append after line 3 in first file
> W          this line
4a6          append after line 4 in first file
> Z          this line
```

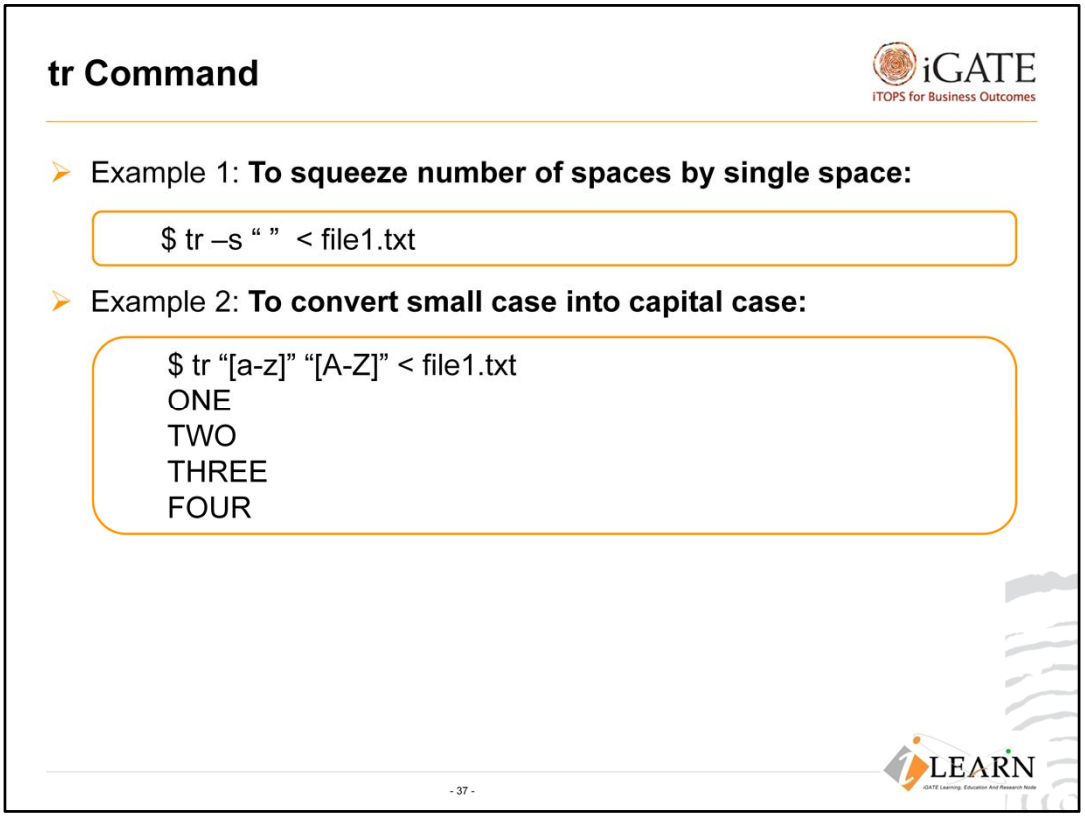


tr Command:
It accepts i/p from standard i/p.
Hence to give input to tr command we have to use i/p redirection operator.
Example:

```
$tr –s” “ file1.txt
```

This is wrong command it will not take i/p from file1.txt.
The correct way of giving i/p is as follows:

```
$tr –s“  “  < file1.txt
```

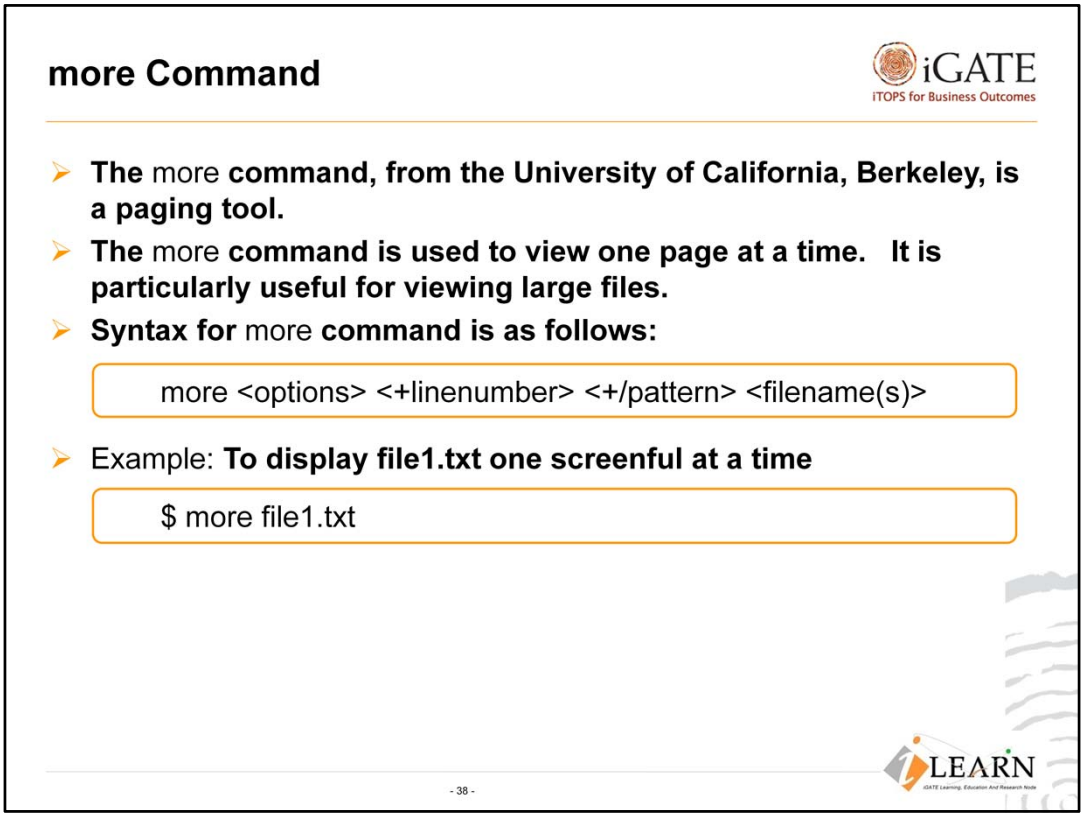


tr Command:

Example: To convert small case into capital case:

```
$ tr “[a-z]” “[A-Z]” < file1.txt
```

The tr command will replace a with A , b with B, and so on.



more Command (Viewing a file one screen at a time):

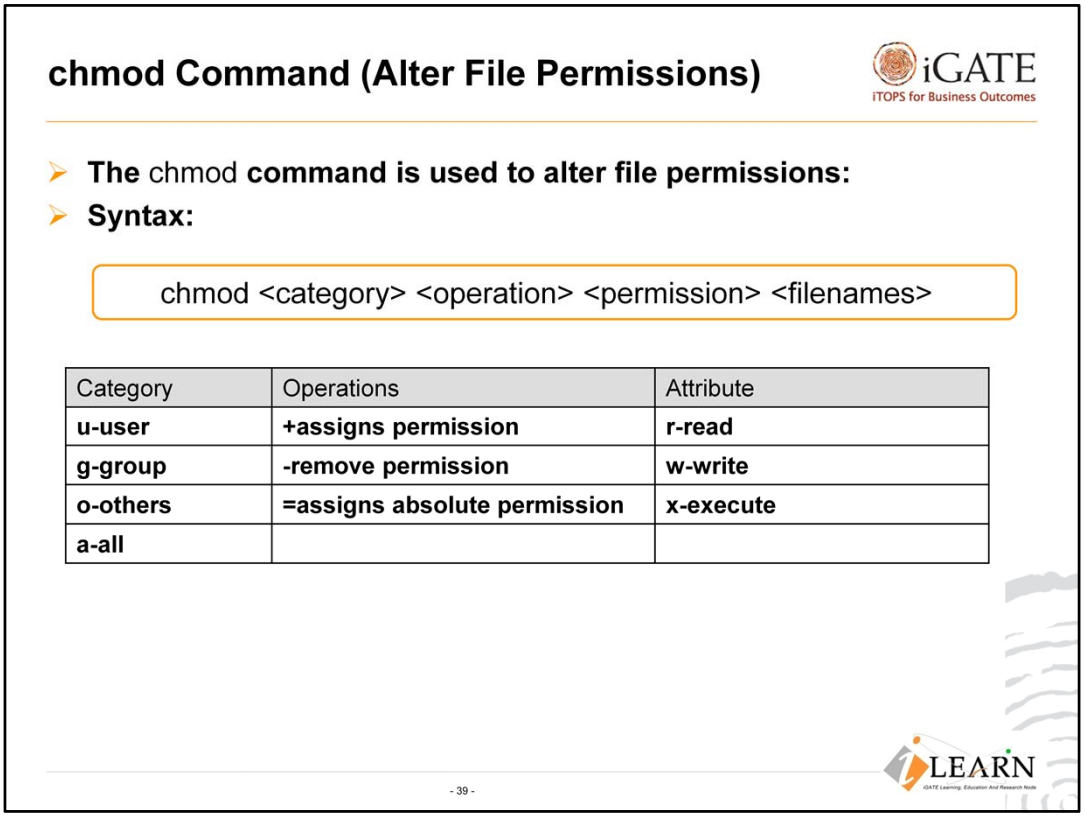
This command from the University of California, Berkeley, is a paging tool – it can be used to view one page at a time. It is particularly useful for viewing large files.

Syntax of this is as follows:

```
more <options> <+linenumber> <+/pattern> <filename(s)>
```

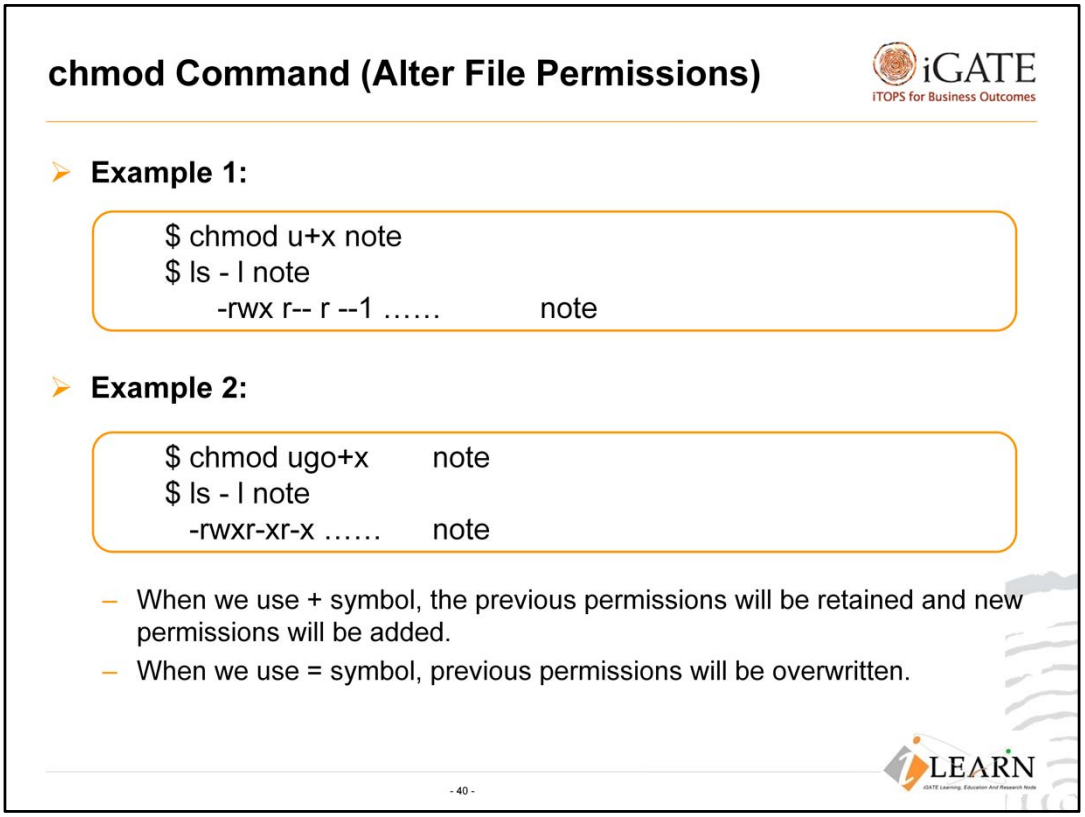
There are a number of options available with the more command. Some of them are listed below:

Command	Description
Spacebar	Displays next screen
Ks	Skips K lines forward
Kf	Skips K screens forward
=	Displays current line number
:f	Displays current file name and line number
K:n	Skips to Kth next file specified on command line
K:p	Skips to Kth previous file specified on command line
/pattern	Searches forward for specified pattern
.	Repeats previous command
Q	Exits command



In UNIX operating system every file is owned by the user who created that file. Only owner or Superuser can change the file permissions using chmod command.

Superuser
In UNIX the superuser is responsible for system administration. **root** is the conventional name of the superuser who has all rights or permissions.
It is never good practice for anyone to use root as their normal user account, since simple typographical error in entering commands can cause major damage to the system. It is advisable to create a normal user account instead and then use the su command to switch to root user when necessary.



chmod Command (Working with file permissions):

The **chmod** command is used to change the file permissions. Permissions can be specified in two ways:
by using symbolic notation, or
by using octal numbers

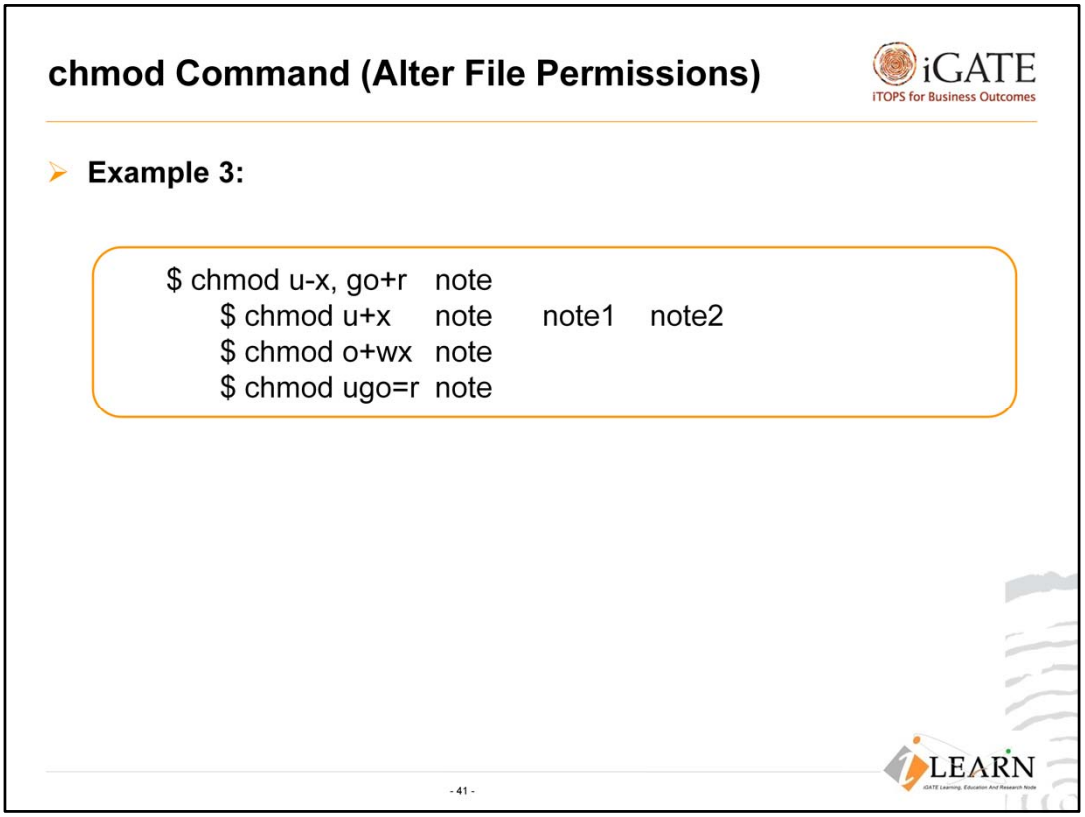
The table given below describes the category, operation, and attributes required by the **chmod** command:

Category	Operation	Attribute & value
u-user	+ assign permission	r-read (4)
g-group	- remove permission	w-write (2)
o-others	= assign absolute permission	x-execute (1)
A-all		

In **octal notation**, r is 4, w is 2, and x is 1. To give read and write permission to only user, we use number 600. 1st number indicates permissions for user(4+2), 2nd number for group, and 3rd number as others.

The following command will give read, write, and execute permissions to user(4+2+1=7), and read, write permissions to group(4+2), and execute (1) permission to others.

Let us consider some examples.



chmod Command (Working with file permissions):

Example 1:

```
$ chmod 761 file1.txt
$ ls -l file1.txt
-rw-rw-r--  3 deshpavn group        60 Mar 29 10:43 file1.txt
```

Example 2:

```
$ chmod ugo-w file1.txt
$ ls -l file1.txt
-r--r--r--  3 deshpavn group        60 Mar 29 10:43 file1.txt
```

Example 3:

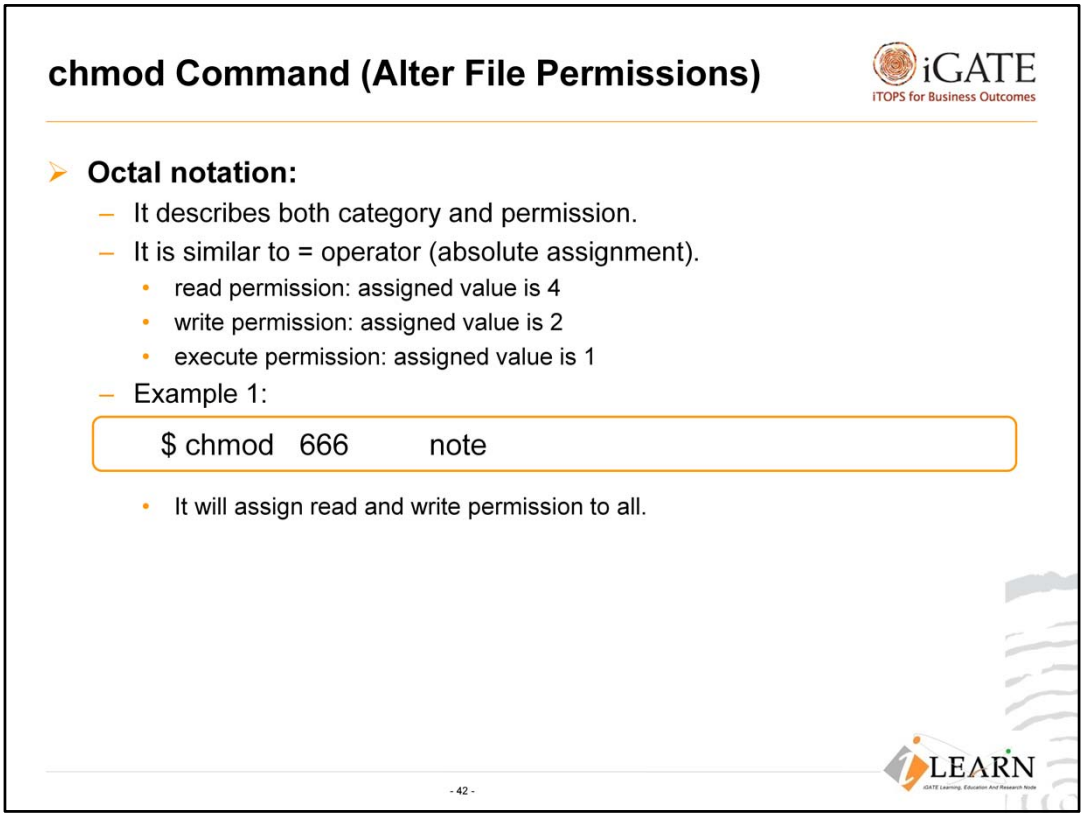
```
$ chmod a+w file1.txt
$ ls -l file1.txt
-rw-rw-rw-  3 deshpavn group        60 Mar 29 10:43 file1.txt
```

Example 4:

```
$ chmod o-w,ug+x file1.txt file2.txt
$ ls -l file1.txt file2.txt
-rwxrwxr--  3 deshpavn group        60 Mar 29 10:43 file1.txt
-rwxr-xr--  1 deshpavn group        61 Mar 29 10:44 file2.txt
```

Example 5:

```
$ chmod 644 file1.txt file2.txt
$ ls -l file1.txt file2.txt
-rw-r--r--  3 deshpavn group        60 Mar 29 10:43 file1.txt
-rw-r--r--  1 deshpavn group        61 Mar 29 10:44 file2.txt
```



chmod Command (Working with file permissions):

Setting Permissions:

The chmod command uses a string, which describes the permissions for a file, as an argument.

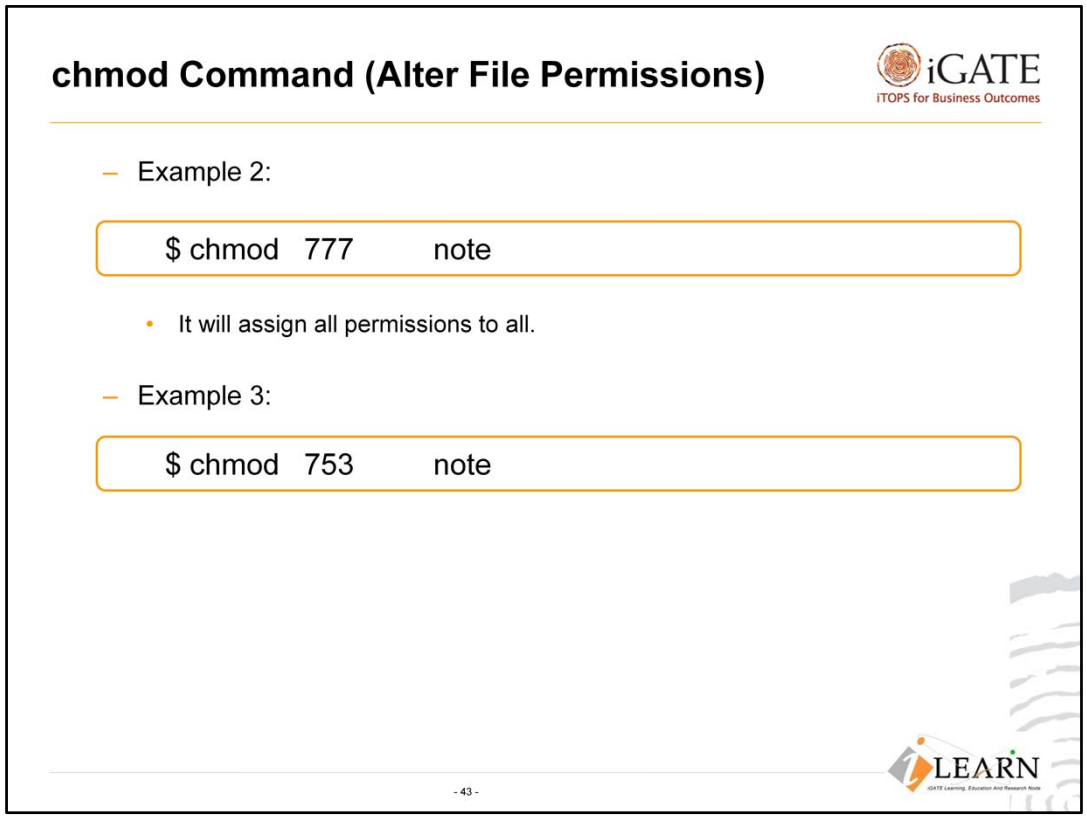
The permission description can be in the form of a number that is exactly three digits. Each digit of this number is a code for the permissions level of three types of people that might access this file:

Owner
Group (a group of users where owner is part of)
Others (anyone else browsing around on the file system)

The value of each digit is set according to the rights that each of the types of people listed above have to manipulate that file.

Permissions are set according to numbers**. Read is 4. Write is 2. Execute is 1**. The sums of these numbers, give combinations of these permissions:

0 = no permissions whatsoever; this person cannot read, write, or execute the file
1 = execute only
2 = write only
3 = write and execute (1+2)
4 = read only
5 = read and execute (4+1)
6 = read and write (4+2)
7 = read and write and execute (4+2+1)



chmod Command (Working with file permissions):

Permissions are given using these digits in a sequence of three: one for owner, one for group, one for world.

Let us look at how I can make it impossible for anyone else to do anything with my apple.txt file but me:

```
$ chmod 700 apple.txt
```

If someone else tries to look into apple.txt, they get an error message:

```
$ cat apple.txt
cat: apple.txt: Permission denied
$
```

If I want other people to be able to read apple.txt, I would set the file permissions as shown below:

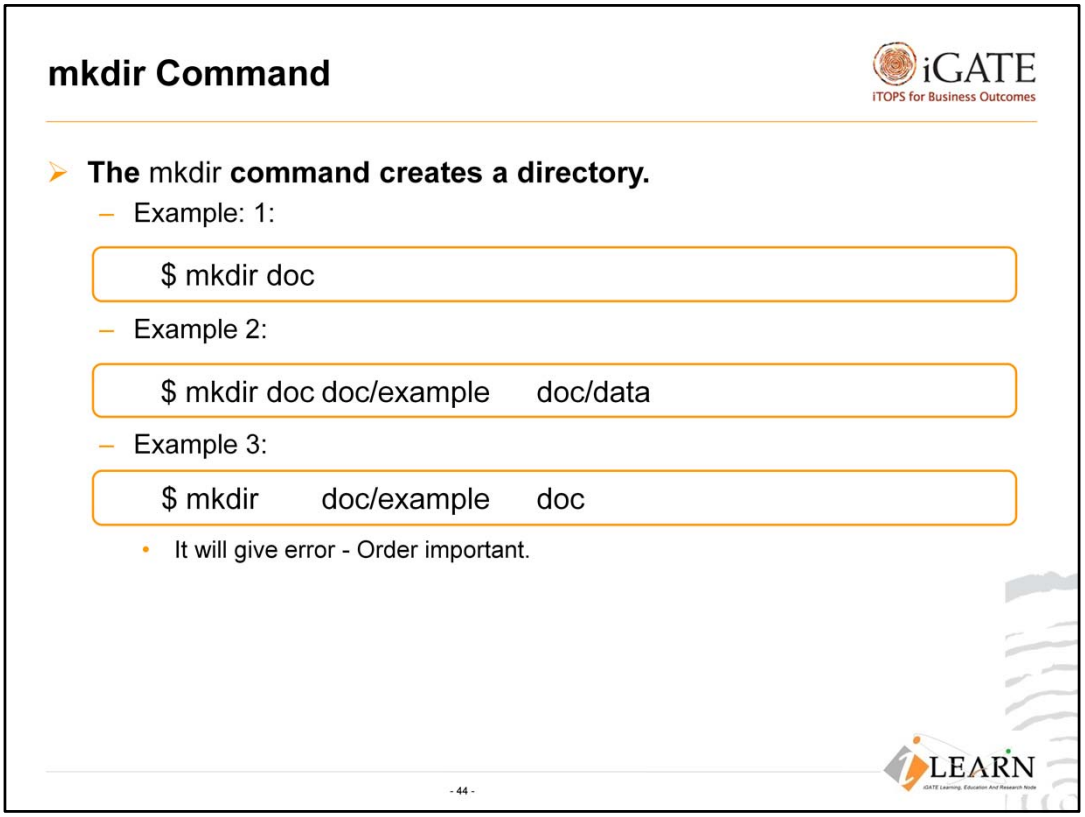
```
$ chmod 744 apple.txt
$
```

Detecting File Permissions:

You can use the ls command with the -l option to show the file permissions set. For example, for apple.txt, I can do a change as follows:

```
$ ls -l apple.txt
-rwxr--r--  1 december december      81 Feb 12 12:45 apple.txt
```

The sequence -rwxr--r-- tells the permissions set for the file apple.txt. The first - tells that apple.txt is a file. The next three letters, rwx, show that the owner has read, write, and execute permissions. Then the next three symbols, r--, show that the group permissions are read only. The final three symbols, r--, show that the others permissions are read only.



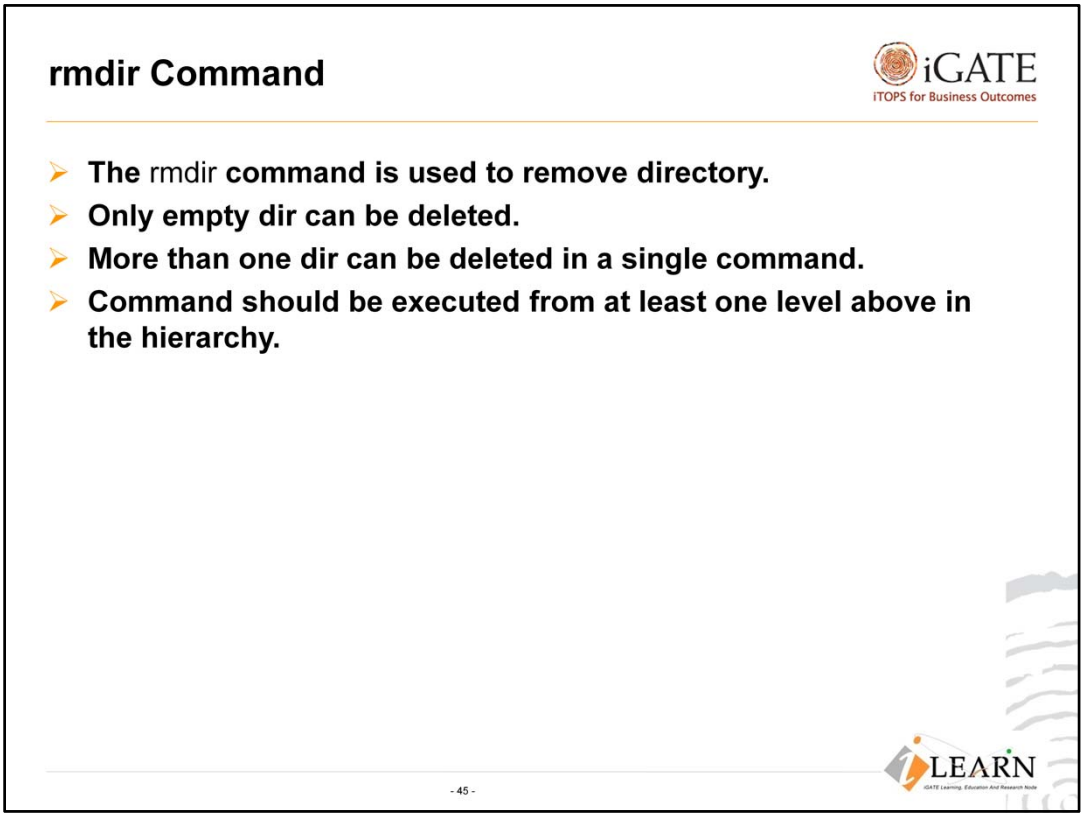
mkdir Command (Creating directory):

A single directory, or a number of subdirectories, can be created using the mkdir command. A directory will not get created if there is already another directory by the same name under the parent directory.

Besides, appropriate permissions will be required.

Example:

```
$ mkdir newdir1
$ ls -F
file1.txt
file2.txt
file3.txt
mail/
newdir1/
newfile.txt
testdir2/
$ mkdir newdir2 newdir2/subdir1 newdir2/subdir2
$ ls -l newdir2
total 4
drwxr-xr-x   2 deshpavn group       512 Mar 29 11:09 subdir1
drwxr-xr-x   2 deshpavn group       512 Mar 29 11:09 subdir2
```



rmdir Command (Removing directory):

The **rmdir** command can be used to delete one or more directories.

Any directory can be deleted only if it is empty. It is important to note that the command needs to be issued from a directory, which is hierarchically above the directory to be deleted.

Example:

```
$ rmdir newdir1
```

rmdir Command

iGATE
iTOPS for Business Outcomes

- **Example 1:**

```
$ rmdir doc
```

- **Example 2:**

```
$ rmdir doc/example doc
```

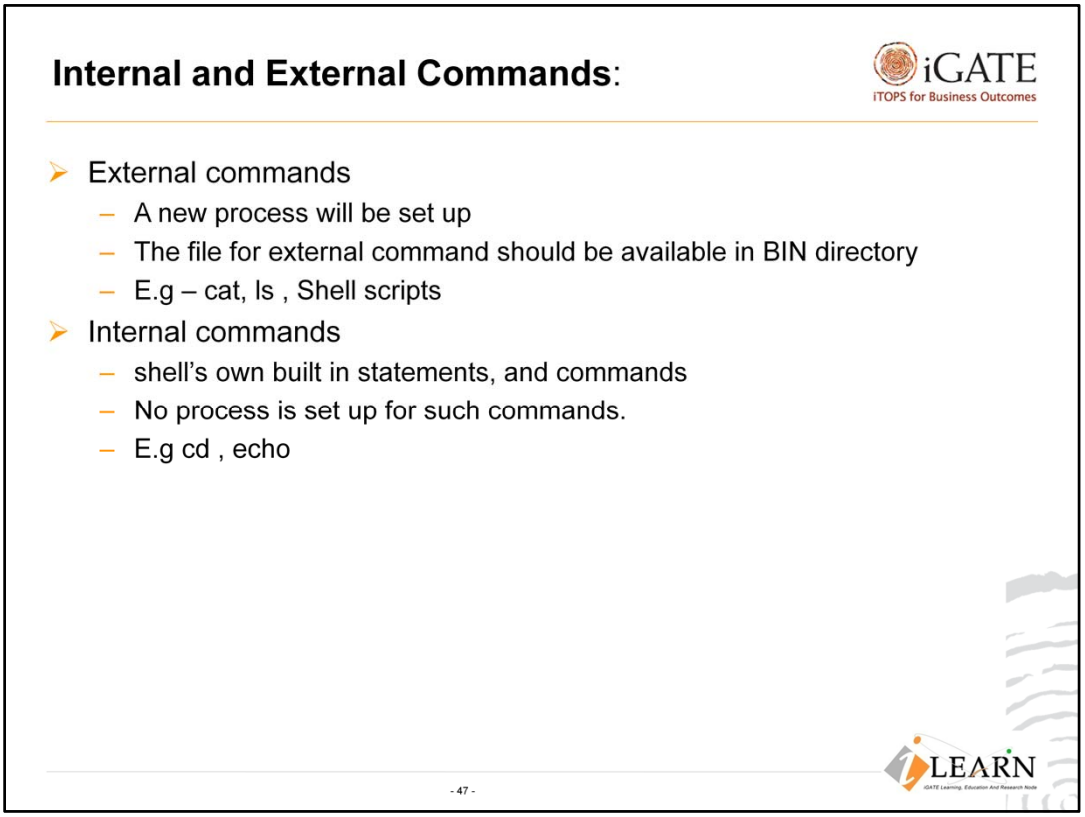
- **Example 3:**

```
$ rmdir doc  doc/example
```

  - It will give error.

iLEARN

- 46 -



Internal and External Commands:

The shell recognizes two types of commands – external and internal.

```
$ rmdir newdir1
```

External commands

•These are commands like cat, ls, and so on or utilities. Shell scripts also come under the category of external commands.
•A new process will be set up for the external commands.
•The file for external command should be available in BIN directory

Internal commands

•These are the shell's own built in statements, and commands such as cd, echo, and so on.
•No process is set up for such commands. For external command it is necessary that some commands are built into the shell itself and no process is set up. That is because it is very difficult or sometimes impossible to implement some commands as external commands.

Summary

iGATE
iTOPS for Business Outcomes

- **In this lesson, you have learnt:**
  - UNIX organizes files in hierarchical manner.
  - File access can be secured using different file permissions.
  - < - Input Redirection
  - > - Output Redirection
  - 2> - Error Redirection
  - chmod command is used to change file permissions.

iLEARN

- 48 -

Review Questions

iGATE
iTOPS for Business Outcomes

- **Question 1: To copy all files with extension txt to mydir directory ___ command is used, if mydir is parent directory of current directory.**
  - Option 1: cp *.txt ..
  - Option 2: cp *.txt ../mydir
  - Option 3: cp mydir *.txt

- **Question 2: 2> symbol is used as error redirection**
  - True / False
- **Question 3: cd . changes the directory to ___.**
- **Question 4: Which of the following command will give only read permission to all for file file1.txt?**
  - Option 1: chmod a=r file1.txt
  - Option 2: chmod a+r file1.txt
  - Option 3: Chmod 666 file1.txt

iLEARN

- 49 -

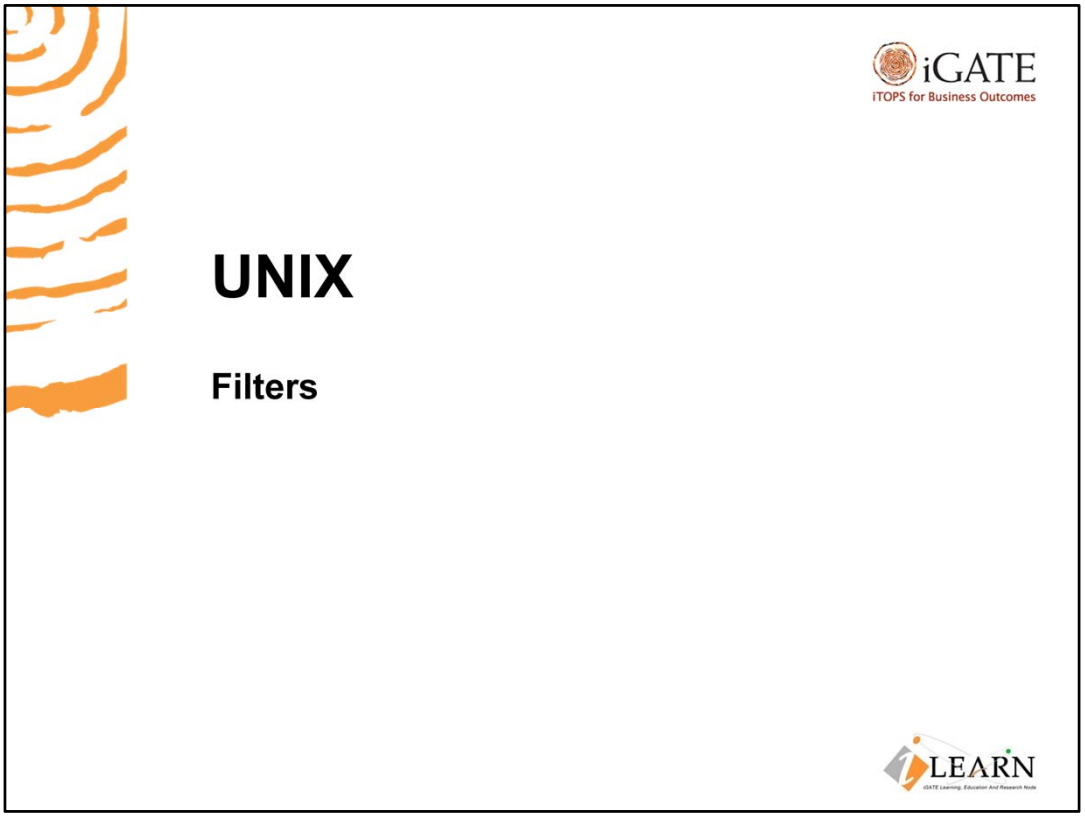
Review – Match the Following

iGATE
iTOPS for Business Outcomes

1. To change directory to home directory	a. rm *.dat
2. To remove all files with extension *.dat	b. cat <abc.txt
3. To display contents of file abc.txt	c. cat > abc.txt
4. To create abc.txt file	d. cd
	e. cd \
	f. mkdir mydir

iLEARN

- 50 -


iGATE
iTOPS for Business Outcomes
UNIX
Filters
iLEARN

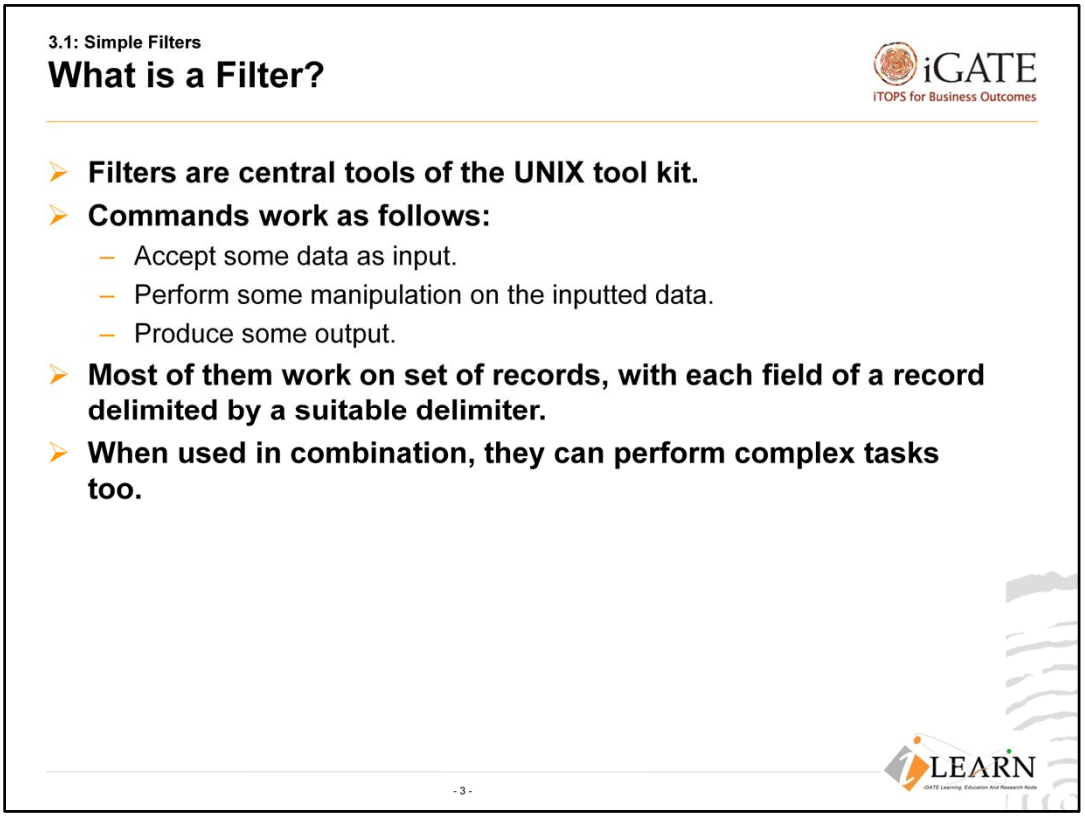
Lesson Objectives

iGATE
iTOPS for Business Outcomes

- **In this lesson, you will learn:**
  - Filter commands in UNIX:
    - Simple Filters
    - Advance Filters

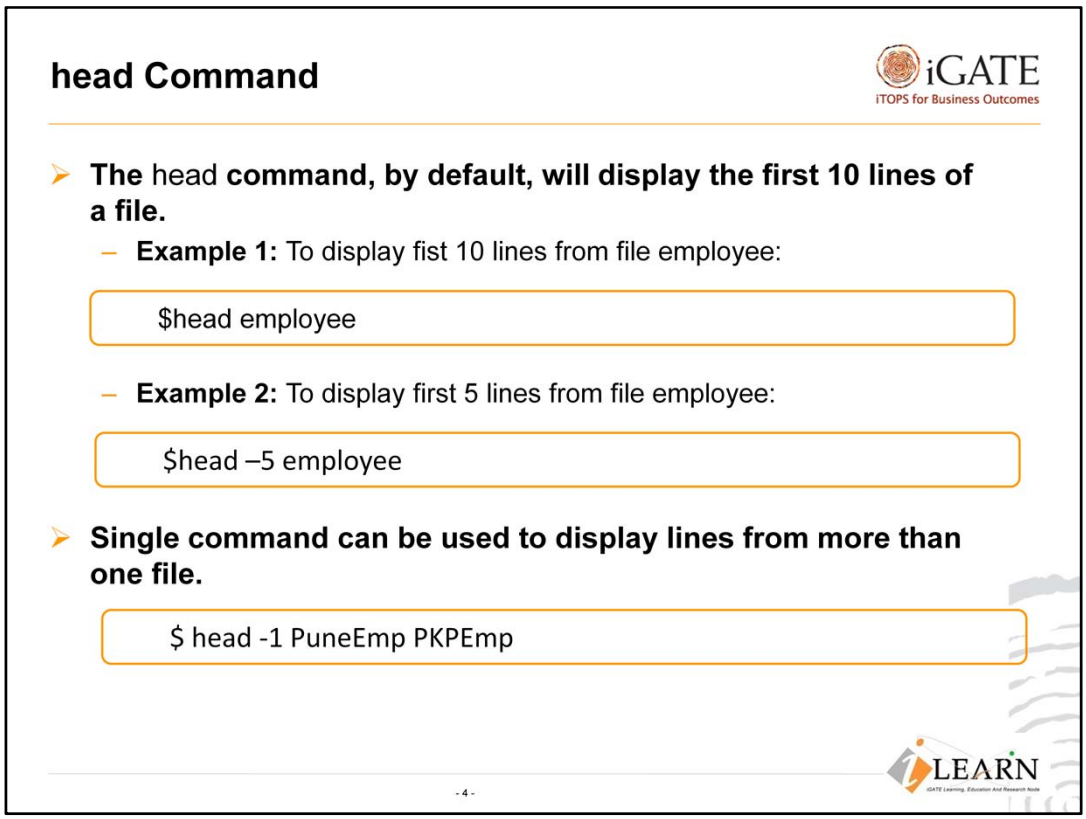
iLEARN

- 2 -



Simple Filters:

Filters are central tools of the UNIX tool kit. These commands accept some data as input, perform some manipulation on it and produce some output. Most of them work on set of records, with each field of a record delimited by a suitable delimiter. They serve as useful text manipulators. When used in combination, they can perform complex tasks as well. This lesson discusses some commonly used simple filters.



The head Commands:

These are simple horizontal filters.
Using head, it is possible to display beginning of one or more lines form files. By default, the first 10 lines are displayed. Incase a numeric line count argument is specified, the command would display those many lines from the beginning of the file.

To display first 10 lines of the file bigfile, use the following syntax**:**

```
$ head bigfile
```

Output:

```
cfile1.lst
cfile2.lst
errfile
file1.txt
file2.txt
file3.txt
mail
newdir1
newdir2
newfile.txt
```

To display first 3 lines of the file bigfile, use the following syntax:

```
$ head -3 bigfile
```

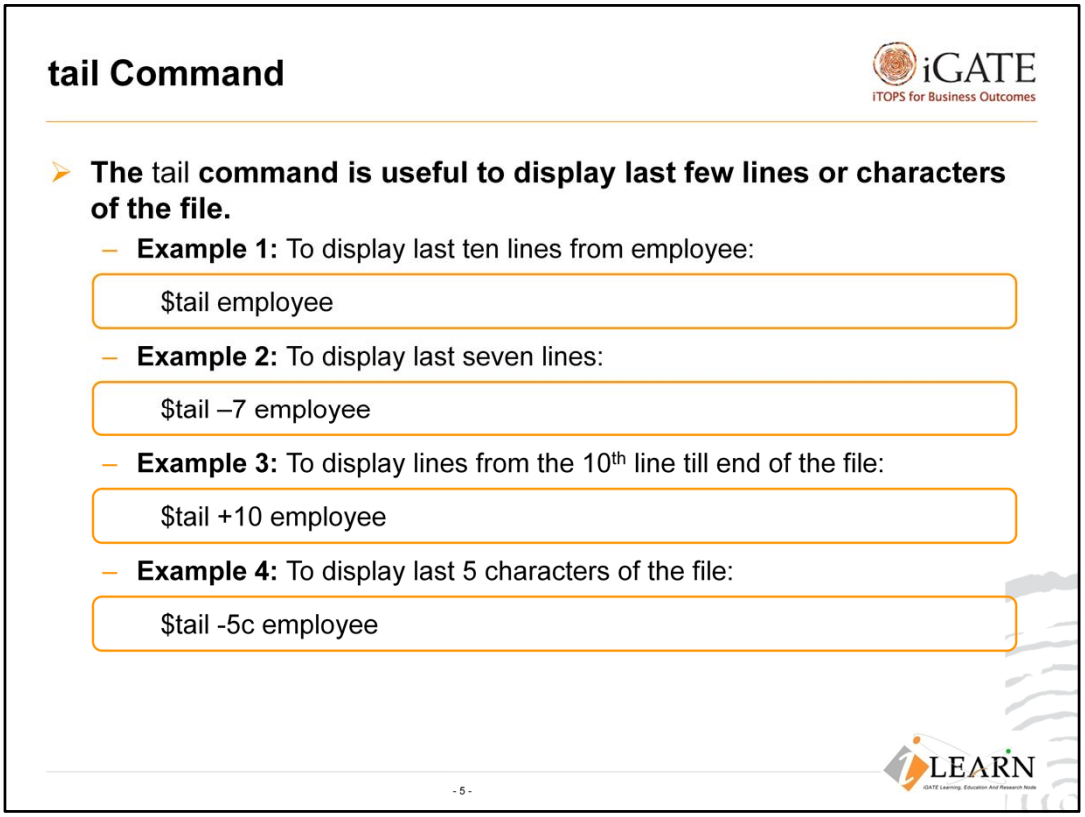
Output:

```
cfile1.lst
cfile2.lst
errfile
```

```
$ head -1 listing bigfile
```

Output:

```
==> listing <==
cfile1.lst
==> bigfile <==
cfile1.lst
```



The tail Commands :

Using the tail command:
Using tail, the end of file can be displayed – default being last 10 lines.

```
$ tail -2 listing
```

Output:
result
testdir2
mail
newdir1
newdir2
newfile.txt

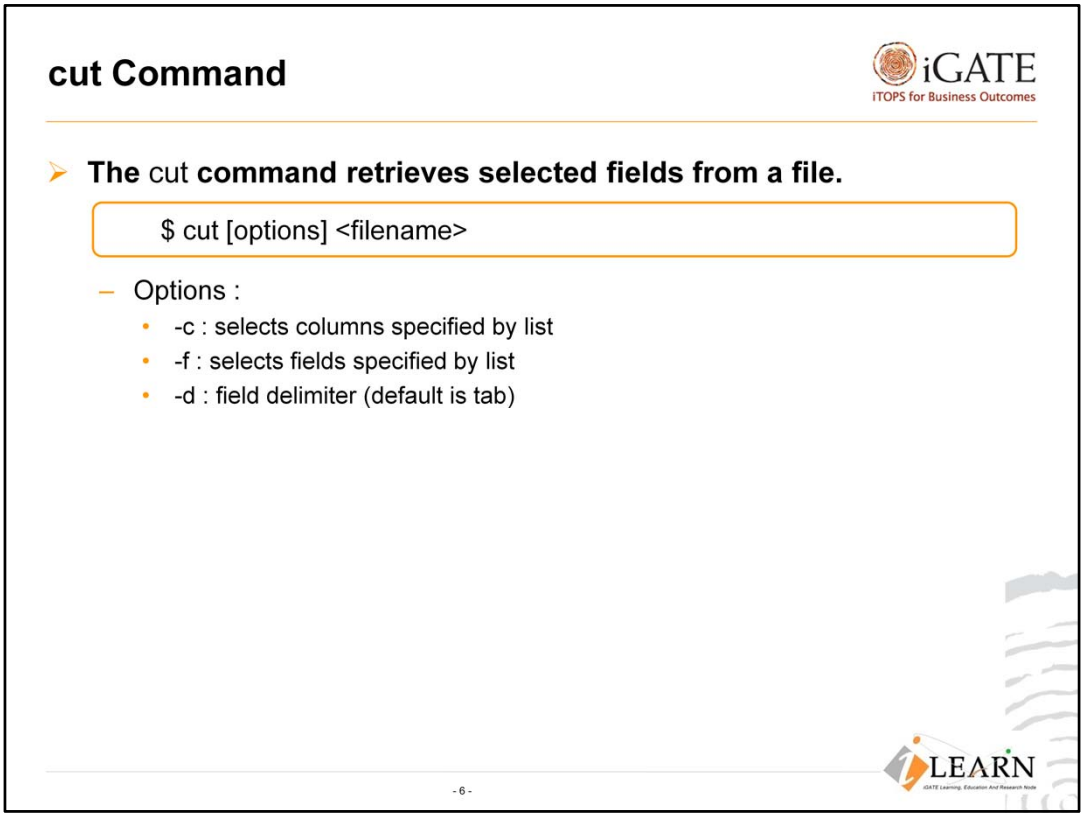
```
$ tail +20 bigfile
```

Output :
result
Testdir2

To display last 6 characters from bigfile, use the following syntax:

```
$ tail -6c bigfile
```

le.txt



cut Command:

You can slice the file vertically with the cut command, and paste laterally with the paste command.

The cut command can be used to retrieve specific column information from a file. In case of fixed record formats, the –c (columns) option can be used to specify column positions. If a delimiter has been used, -f (field) in conjunction with –d (delimiter) options can be used for retrieval. The default delimiter is tab.

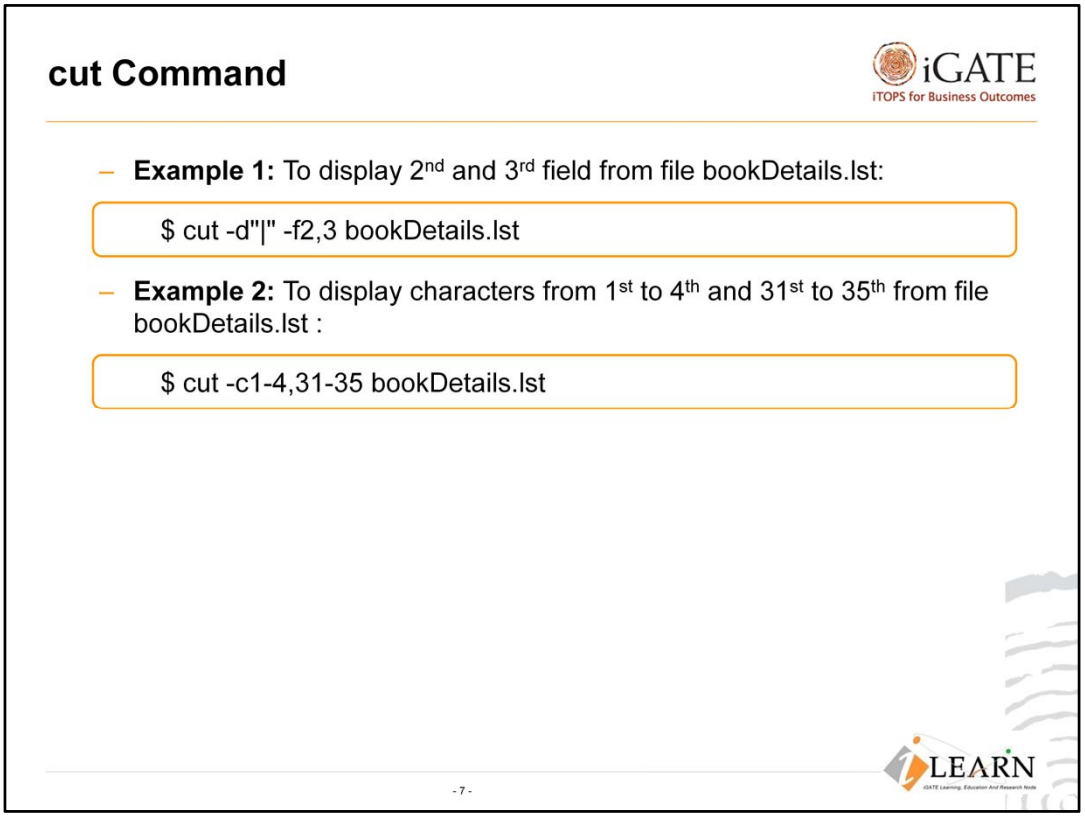
```
$ cat bookDetails.lst
```

Output:
```
1001|Unix for You                |375
1002|Learning Unix               |250
1003|Unix Shell Programming      |450
1004|Unix Device Drivers         |375
1005|Advanced Unix Concepts      |450
```

The following command will display first 4 characters followed by 31st to 35th characters:

```
$ cut -c1-4,31-35 bookDetails.lst
```

Output:
```
1001|375
1002|250
1003|450
1004|375
1005|450
```



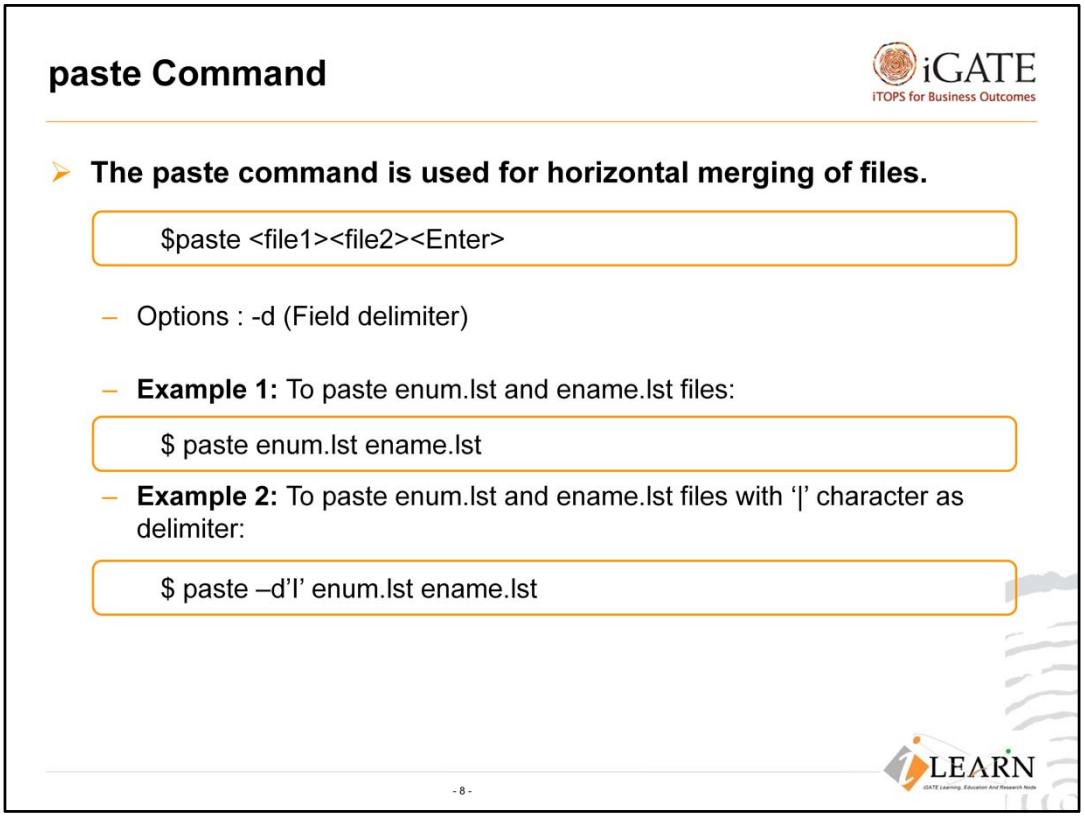
cut Commands:

To display 2nd and 3rd field from file bookDetails.lst :
In the given command on the above slide, the –d option specifies field delimiter is |. Hence it will consider that in bookDetails file there are 3 fields separated by '|' character. The –f option will specify to display 2nd and 3rd field.

```
$ cut -d"|" -f2,3 bookDetails.lst
```

Output:

```
Unix for You                |375
Learning Unix               |250
Unix Shell Programming      |450
Unix Device Drivers         |375
Advanced Unix Concepts      |450
```



paste Command:

Several files can be pasted laterally, with specific delimiters, with the paste command.

```
$  cat enum.lst
```

Output:
1010
2021
3718
4135
5765

```
$ cat ename.lst
```

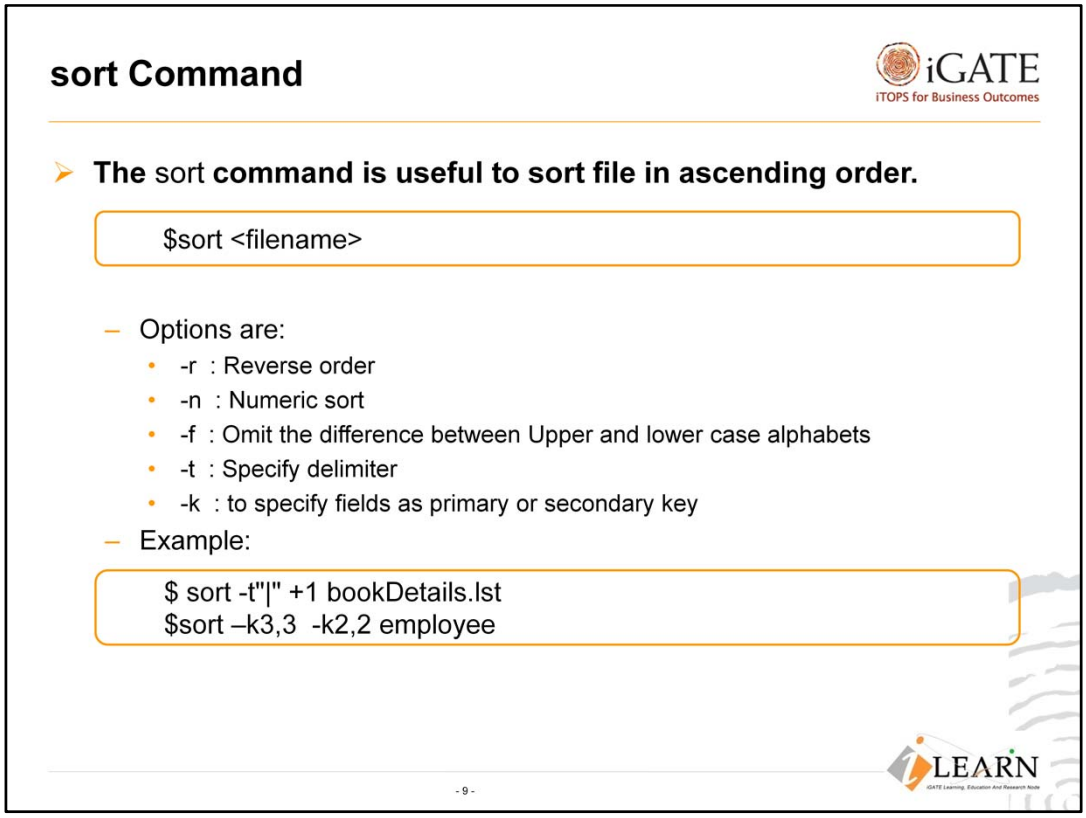
Output:
Abc
Zxc
Qwe
Jkl
Uio

```
$ paste enum.lst ename.lst
```

Output:
1010 Abc
2021 Zxc
3718 Qwe
4135 Jkl
5765 Uio

```
$ paste -d"|" enum.lst ename.lst
```

Output:
1010|Abc
2021|Zxc
3718|Qwe
4135|Jkl
5765|Uio



sort Command:

Sorting a file with the sort command:

The sort command sorts a file (which may or may not contain fixed length records) on line by line basis. Default sorting is in the ascending ASCII order, which can be reversed by using the –r option.

Sorting can be done on one or more fields by specifying the delimiter using –t option. It is also possible to specify character positions within fields.

Using the –m option, it is also possible to merge any number of sorted files.

Since the sorting is done on the basis of ASCII collating sequence, incase of sorting of numbers, -n option needs to be used.

```
$ sort -t"|" +1 bookDetails.lst
```

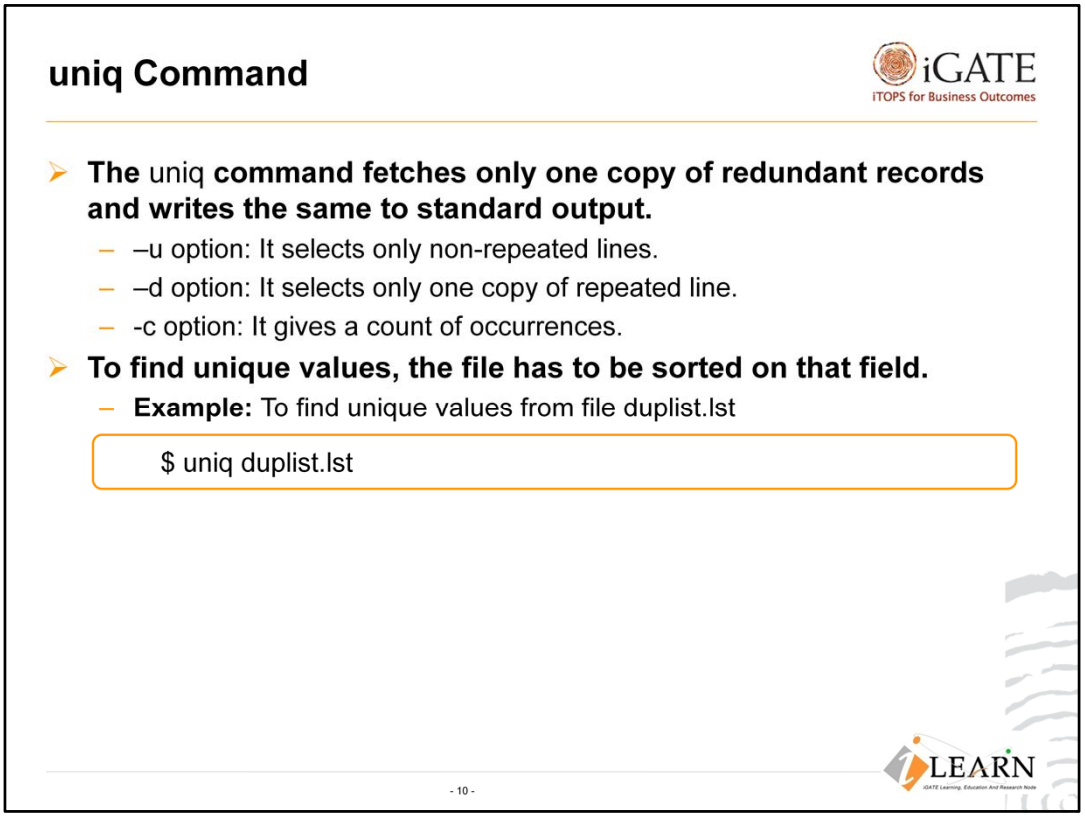
Output:
```
1005|Advanced Unix Concepts   |450
1002|Learning Unix            |250
1004|Unix Device Drivers       |375
1003|Unix Shell Programming    |450
1001|Unix for You              |375
```

To sort file employee on 3rd field as primary key and 2nd field as secondary key, use the following syntax:

```
$ sort –t"|" –k3,3 -k2,2 Employee
```

To consider only 3rd and 4th character from 2nd field for sorting employee file, use the following syntax:

```
$sort –t"|" –k2.3,2.4 employee
```



uniq Command:

The **uniq** command requires a sorted file as input. It fetches only one copy of redundant records and writes the same to standard output.

The **–u** option can be used to select only non-repeated lines, while the **–d** option can be used to select only one copy of repeated line. It is also possible to get a count of occurrences with the **–c** option.

Example 1:

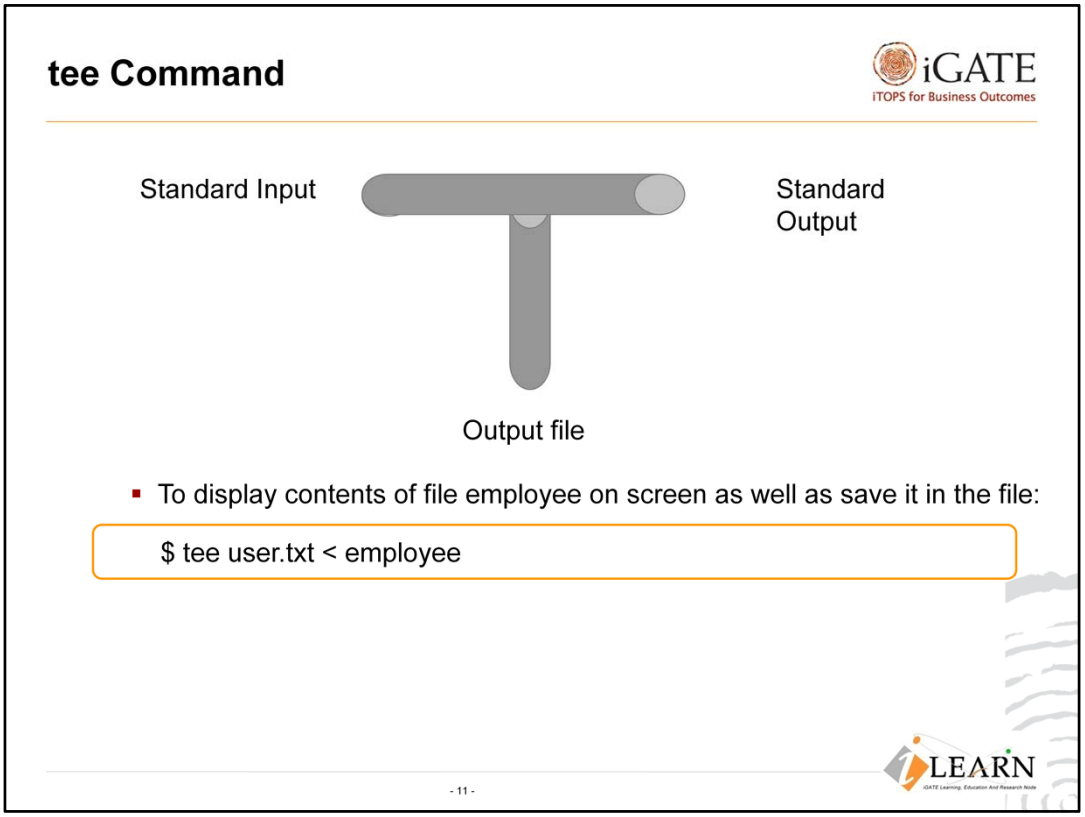
```
$ cat  duplist.lst
```

Output:
```
1
34
30
34
1
23
23
4
```

Example 2:

```
$ sort -n duplist.lst | uniq
```

Output:
```
1
4
23
30
34
```



tee Command:

The tee command copies the standard input to the standard output and also to the specified file.

If it is required to see the output on screen as well as to save output to a file, the **tee** command can be used. The **tee** command uses both standard input and standard output.

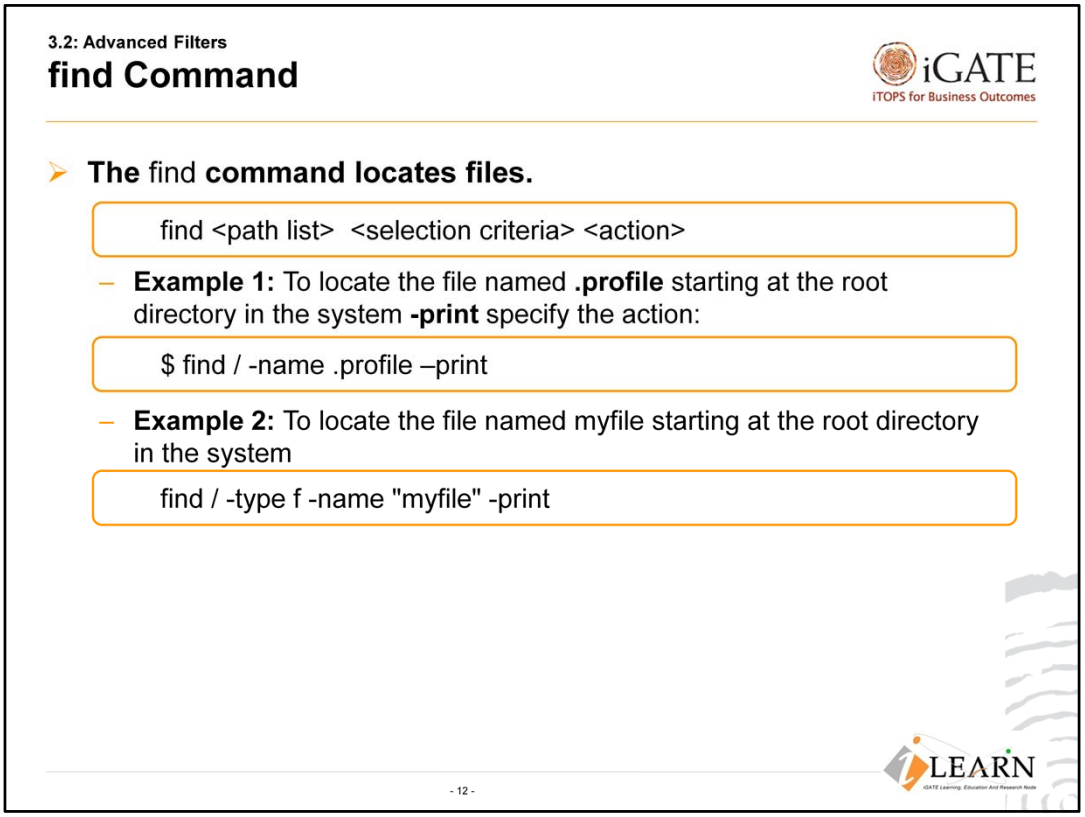
To display list of users and it's count both on screen, use the following:

```
Who| tee /dev/tty  | wc -l
```

If we give command as **who|wc –l** , it will display only number of users on screen. However, in the above command **tee** will save the o/p to /dev/tty file which is terminal device file. Hence the o/p of **who** will be displayed on screen and also will be transferred as i/p to **wc** command.

In the following command, **sort** will sort the file and o/p will be transferred to **tee** command. It will display o/p on screen as well as store it in file **sorted_file.txt**. The o/p will be given to **uniq** command which will give count of lines in the file. The head will find first 12 lines and store it in **top12.txt** file.

```
$sort somefile.txt | tee sorted_file.txt |  uniq -c |  head 12 > top12.txt
```



find Command (locating files with find):

The **find** command is used to find files matching a certain set of selection criteria. The **find** command searches recursively in a hierarchy, and also for each pathname in the pathname-list (a list of one or more pathnames specified for searching).

The syntax of the find command is given in the following format:

```
$ find <path list>  <selection criteria> <action>
```

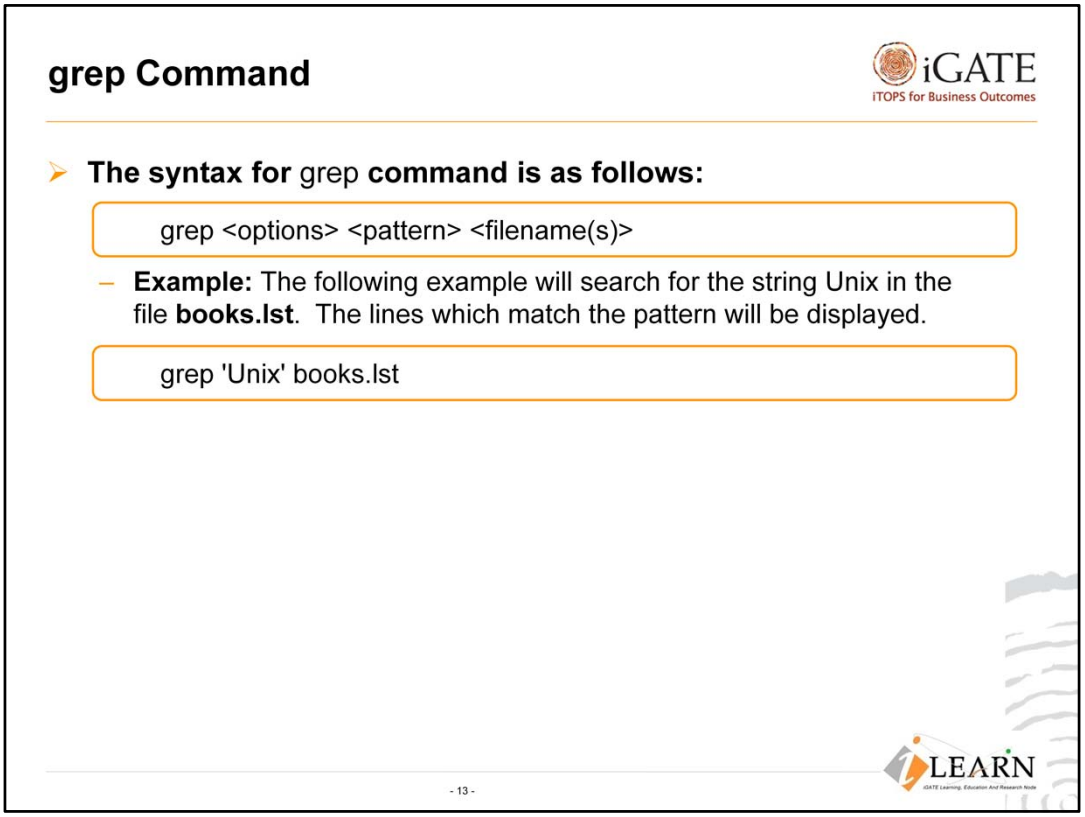
The find command first looks at all the files in the directories specified in the path list. Subsequently, it matches the files for one or more selection criteria. Finally it takes action on those selected files.

```
$ find / -name .profile -print
```

The above command will locate the **.profile** files in the system.

```
$find . –name *stat
```

The above command will locate all file names ending with **stat**.



grep Command:

The **grep** command is used to locate a pattern / expression in a file / set of files. There are many options that are available for obtaining different types of outputs.

The syntax for the grep command is as follows:

```
grep <options> <pattern> <filename(s)>
```

The grep command scans the file(s) specified for the required pattern, and outputs the lines containing the pattern. Depending on the options used, appropriate output is printed. The grep command compulsorily requires a pattern to be specified, and the rest of the arguments are considered as file names in which the pattern has to be searched.

grep Command

iGATE
iTOPS for Business Outcomes

- **Options of grep:**
  - c : It displays count of lines which match the pattern.
  - n : It displays lines with the number of the line in the text file which match the pattern.
  - v : It displays all lines which do not match pattern.
  - i : It ignores case while matching pattern.
  - -w : It forces grep to select only those lines containing matches that form whole words

iLEARN
iGATE Learning, Education And Research Node

- 14 -

grep Command

iGATE
iTOPS for Business Outcomes

- **Example 1:** To print all lines containing “rose” regardless of case:

```
$grep -i rose flower.txt
```

- **Example 2:** To print all lines containing “rose” as a word:

```
$grep -w rose flower.txt
```

- **Example 3:** To print all lines not containing “rose”:

```
$grep -v rose flower.txt
```

iLEARN
iGATE Learning, Education And Research Node

- 15 -

grep Command

iGATE
iTOPS for Business Outcomes

- **Regular Expression:**

Expression	Description
^ (Caret)	match expression at the start of a line, as in ^A.
$ (Question)	match expression at the end of a line, as in A$.
\ (Back Slash)	turn off the special meaning of the next character, as in \^.
[ ] (Brackets)	match any one of the enclosed characters, as in [aeiou]. Use Hyphen "-" for a range, as in [0-9].
[^]	match any one character except those enclosed in [ ], as in [^0-9].
. (Period)	match a single character of any value, except end of line.
* (Asterisk)	match zero or more of the preceding character or expression.
\{x,y\}	match x to y occurrences of the preceding.
\{x\}	match exactly x occurrences of the preceding.
\{x,\}	match x or more occurrences of the preceding.

iLEARN
iGATE Learning, Education And Research Node

- 16 -

grep Command

iGATE iTOPS for Business Outcomes

- **Examples of Regular Expression:**

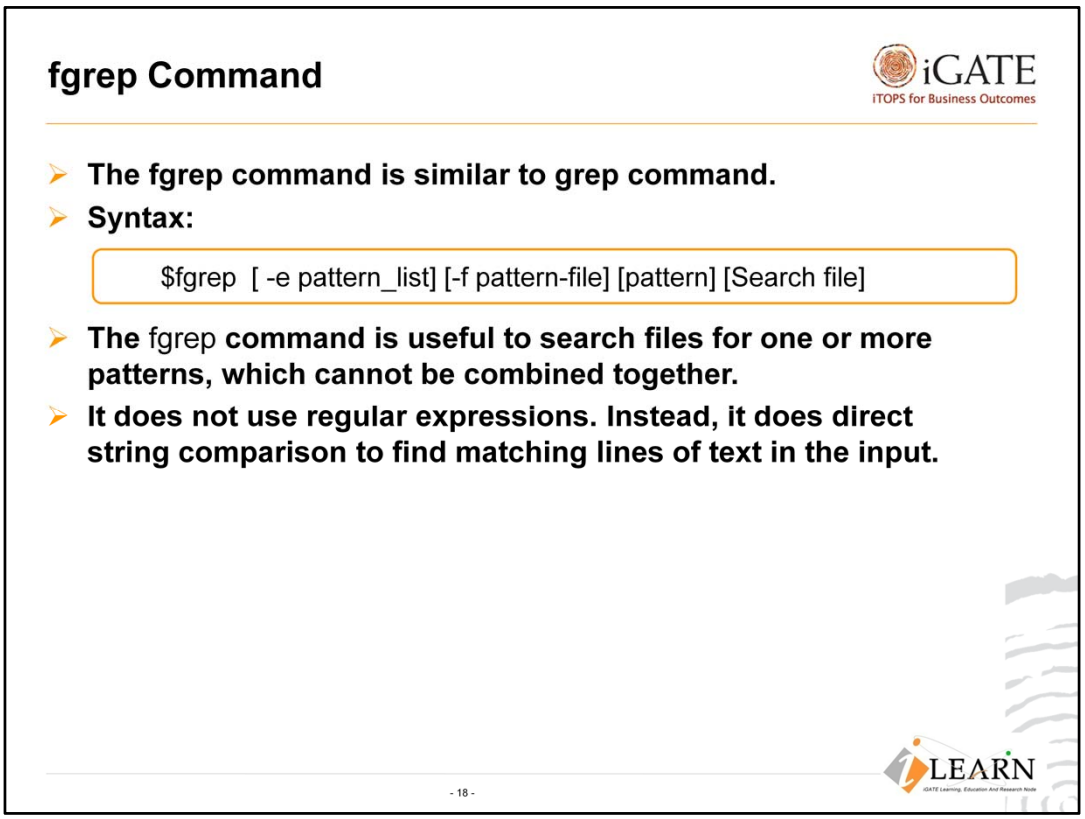
Example	Description
grep "smile" files	search *files* for lines with 'smile'
grep '^smile' files	'smile' at the start of a line
grep 'smile$' files	'smile' at the end of a line
grep '^smile$' files	lines containing only 'smile'
grep '\^s' files	lines starting with '^s', "\" escapes the ^
grep '[Ss]mile' files	search for 'Smile' or 'smile'
grep 'B[oO][bB]' files	search for BOB, Bob, BOb or BoB
grep '^$' files	search for blank lines
grep '[0-9][0-9]' file	search for pairs of numeric digits

iLEARN iGATE Learning, Education And Research Node

- 17 -

grep Command:
Some more examples:

Example	Description
grep '^From: ' /usr/mail/$USER	list your mail
grep '[a-zA-Z]'	any line with at least one letter
grep '[^a-zA-Z0-9]	anything not a letter or number
grep '[0-9]\{3\}-[0-9]\{4\}'	999-9999, like phone numbers
grep '^.$'	lines with exactly one character
grep '"smug"'	'smug' within double quotes
grep '"*smug"*'	'smug', with or without quotes
grep '^\.'	any line that starts with a Period "."
grep '^\.[a-z][a-z]'	line start with "." followed by 2 lowercase letters



fgrep Command:

The **fgrep** command can also accept multiple patterns from command line as well as a file. However, it does not accept regular expressions – only fixed strings can be specified. The **fgrep** command is faster than **grep** and **egrep**, and should be used while using fixed strings.

The **egrep** and **fgrep** commands to some extent overcome the limitations of **grep**. However, the principal disadvantage of the grep family of filters is that there are no options available to identify fields. Also it is very difficult to search for an expression in a field. This is where the **awk** command is very useful.

fgrep Command

iGATE iTOPS for Business Outcomes

- **Options of fgrep command:**
  - -e pattern_list :
    - It searches for a string in pattern-list.
  - -f pattern-file :
    - It takes the list of patterns from pattern-file.
  - pattern
    - It specifies a pattern to be used during the search for input.
    - It is same as grep command.
  - E.g To search employee file for all patterns stored in mypattern file

    $ fgrep –f mypattern employee.lst

iLEARN

- 19 -

fgrep Command:
Example using fgrep:

```
$ cat stud.lst
```

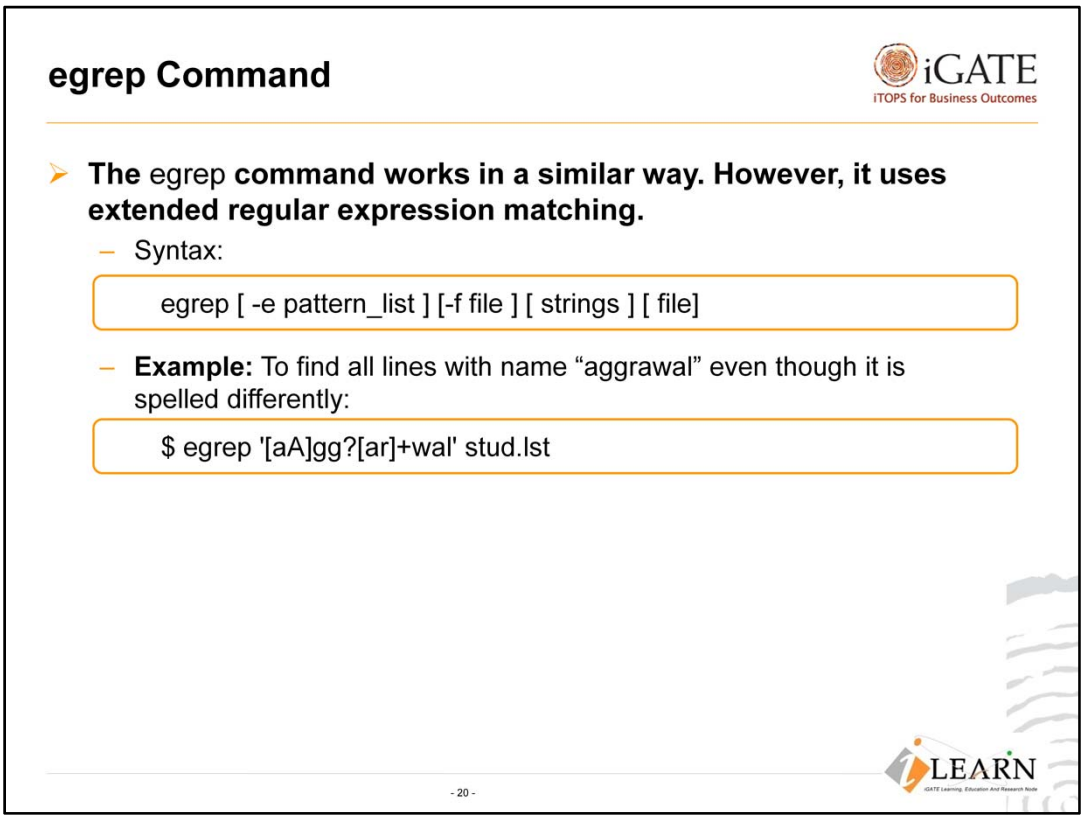
Output:
```
R001|Pratik Sharma      |425
R002|Pallavi V.         |398
R003|Pratibha Aggarwal  |400
R004|Preeti Agrawal     |390
R005|Prerana Agarwal    |421
R006|Pranita aggarwal   |380
```

```
$cat mypattern
```

Output:
```
Pratik
Pratibha
```

```
$ fgrep –f mypattern stud.lst
```

Output:
```
R001|Pratik Sharma      |425
R003|Pratibha Aggarwal  |400
```



egrep Command (extending grep):

The egrep command offers all the options of the grep command. In addition, it is possible to specify alternative patterns. The table given below gives the extended regular expression used by egrep.

Expression	Significance
ch+	Match with 1 or more occurrences of character ch
ch?	Match with 0 or more occurrences of character ch
exp1|exp2	Match with expressions exp1 or exp2
(a1|a2)a3	Match with expression a1a3 or a2a3

Some examples of using egrep are given:

```
$ cat stud.lst
```

Output:
```
R001|Pratik Sharma      |425
R002|Pallavi V.         |398
R003|Pratibha Aggarwal  |400
R004|Preeti Agrawal     |390
R005|Prerana Agarwal    |421
R006|Pranita aggarwal   |380
```

```
$ egrep '[aA]gg?[ar]+wal' stud.lst
```

Output:
```
R003|Pratibha Aggarwal  |400
R004|Preeti Agrawal     |390
R005|Prerana Agarwal    |421
R006|Pranita aggarwal   |380
```

Summary

iGATE
iTOPS for Business Outcomes

- **In this lesson, you have learnt:**
  - The head and tail filter commands filter the file horizontally.
  - The cut and paste commands filter the file vertically.
  - -m option of sort command is used to merge two sorted files.
  - The tee command helps us to send o/p to standard o/p as well as to file.
  - grep, fgrep, and egrep commands use to search files for some pattern.

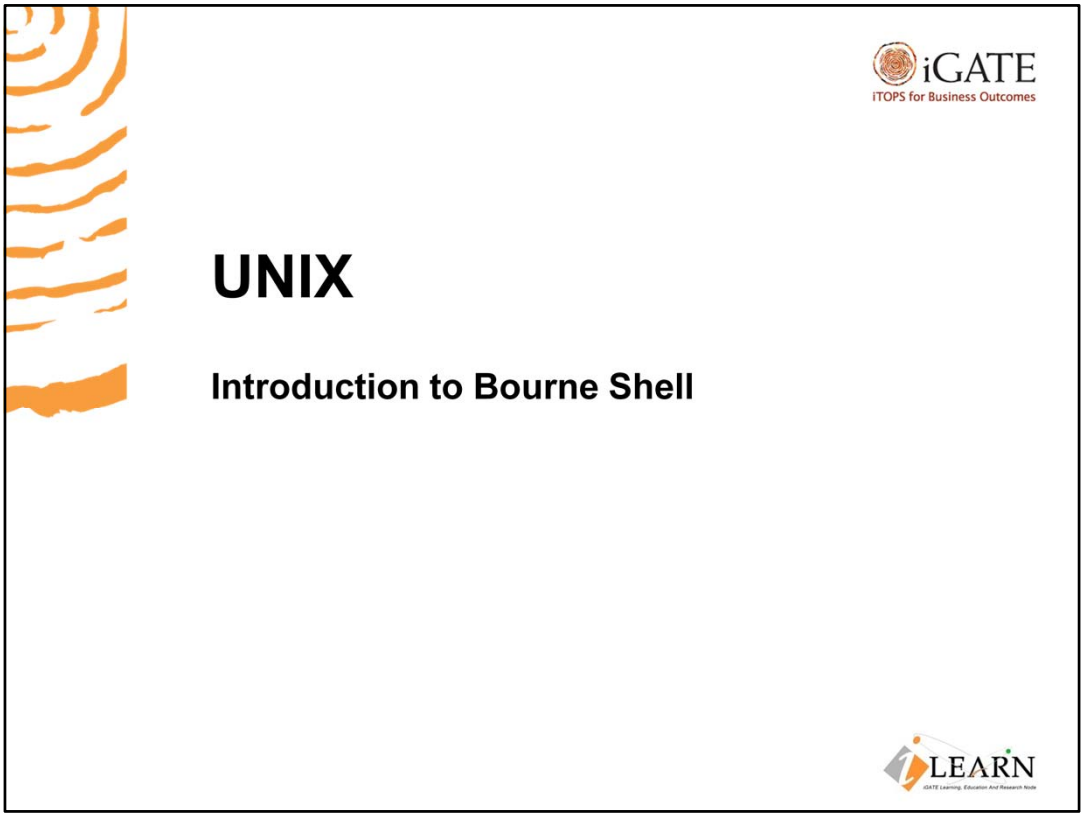
iLEARN

- 21 -

Review Questions

- Question 1: ___ command to display directory listing on screen as well as store it in dirlist.lst.
- Question 2: ___ filter commands filter file vertically?
- Question 3: ___ filter commands filter file horizontally?

- 22 -


iGATE
iTOPS for Business Outcomes
UNIX
Introduction to Bourne Shell
iLEARN

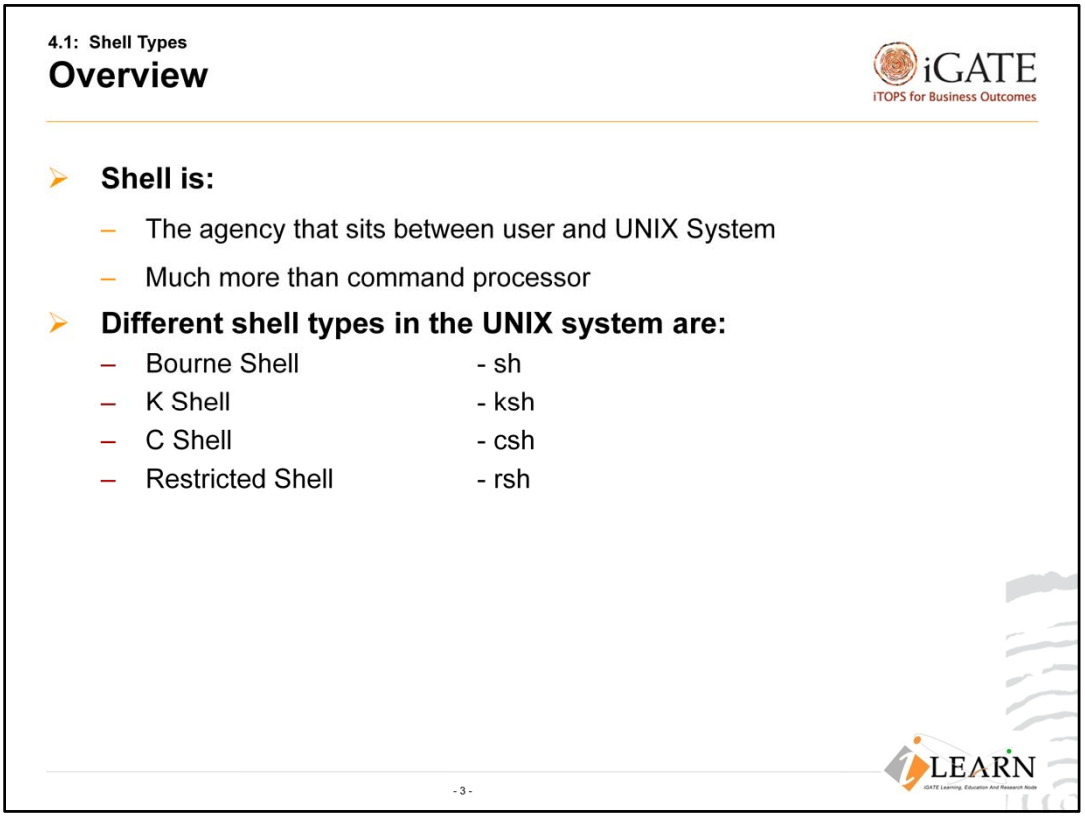
Lesson Objectives

iGATE
iTOPS for Business Outcomes

- **To understand following topics:**
  - Different shell types
  - Working of shell
  - Bourne shell metacharacters
  - Shell redirection
  - Command substitution

iLEARN

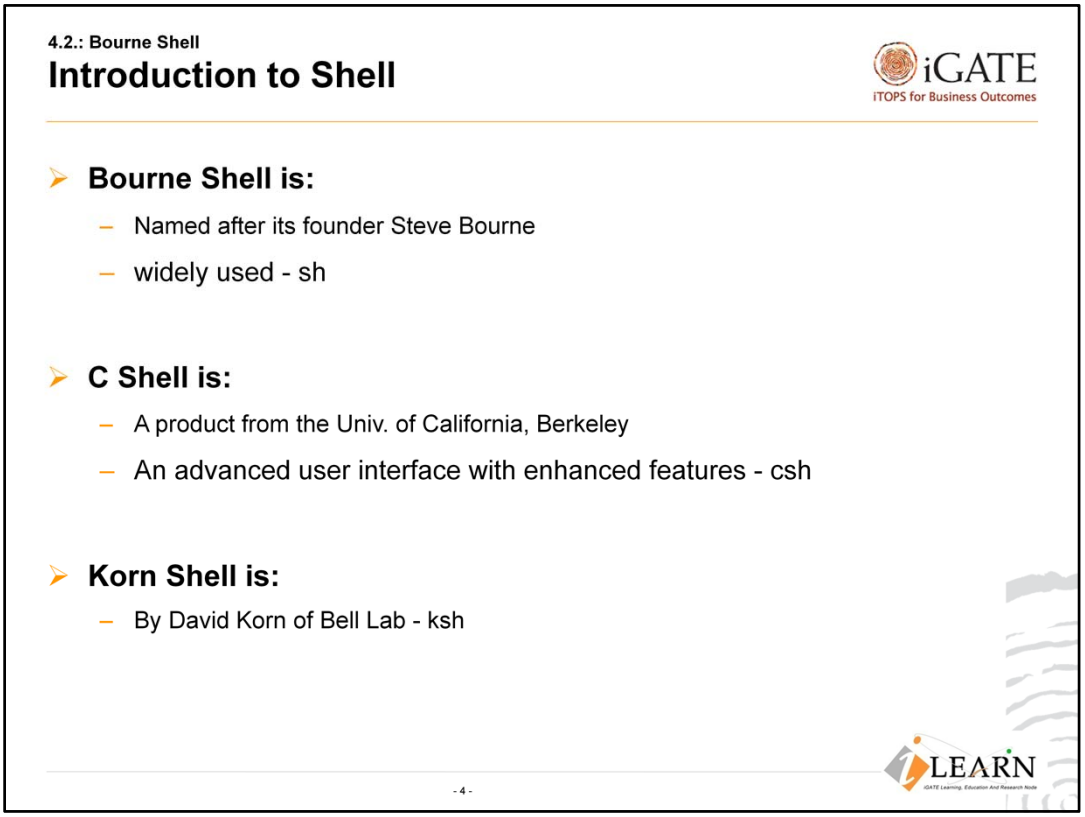
- 2 -



What is a Shell?

The shell is the agency that sits between the user and the Unix system. Whenever a command is issued, it is the shell that acts as the command interpreter. But the shell in Unix is much more than just a command processor. This chapter introduces some of the features of the Bourne Shell.

Shell is a process that creates an environment for you to work in.
When you login to UNIX machine you see a prompt because automatically a new shell process is started. This process will be terminated whenever user logs out.
How the command is executed
-The shell displays prompt and wait for you to enter a command
-When you type any command it scans the command line and processes all metacharacters to recreate simplified command. (e.g if the command is rm *, then * will be replaced by all file names in the current directory)
-Then it passes the command to kernel for execution
-And wait till execution of the command completes
-After command execution is complete, it displays prompt again to take up the next command



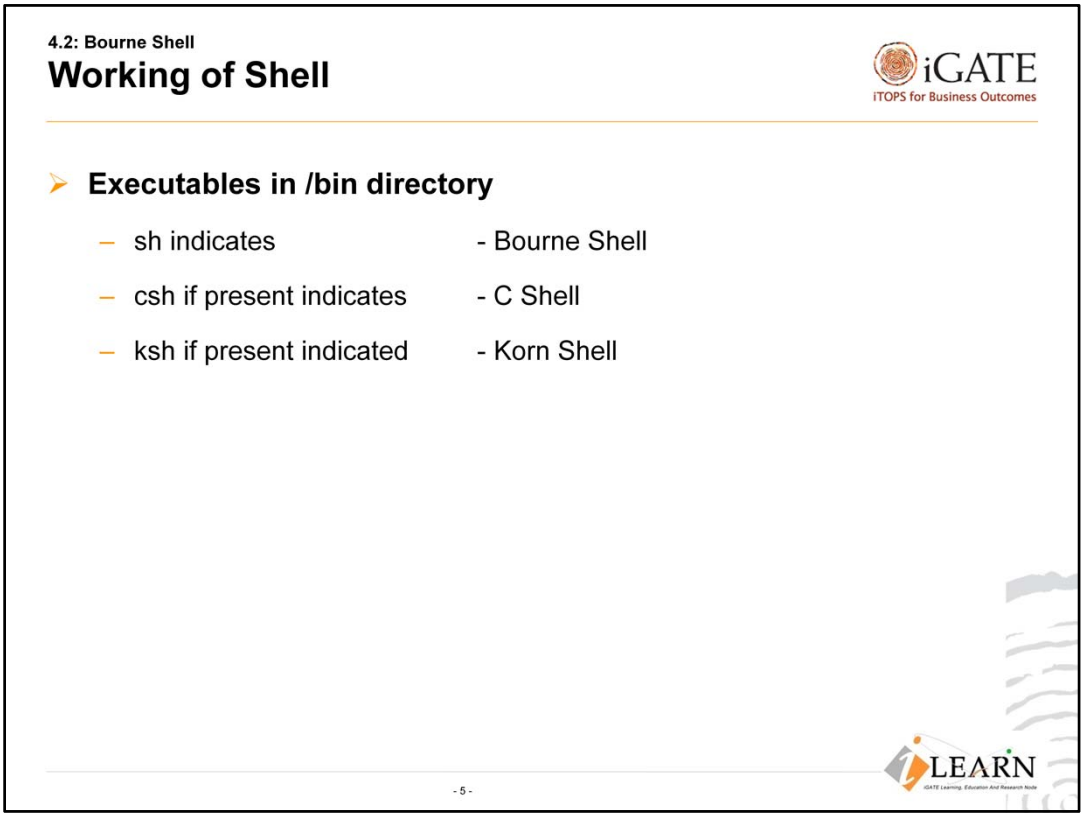
Introduction to the shell:

Bourne Shell is one of the earliest and most widely used Unix shells. It is named after its founder Steve Bourne. The executable program sh in the /bin directory is the Bourne shell.

There are other shells available on the Unix systems – popular amongst them are the C shell and the Korn shell.

The C shell is a product from the University of California, Berkeley. It has an advanced user interface with enhanced features. C shell, if present, is available as csh.

The Korn shell is from the Bell Labs. It is the most modern shell available currently, and is likely to become a standard. The Korn shell executable is ksh.



Working of Shell:

The shell is a Unix command – it is a program that starts when user logs in and terminates when user logs out. The job of the shell is to accept and interpret user requests (which are nothing but other Unix commands). Besides the shell is also programmable – this will be covered later.

The shell typically performs following activities in a cycle:

Issues a $ prompt, and waits for user to enter a command

Scans and processes the command after user enters command

The command is passed on to the Kernel for execution and the shell waits for its conclusion

The $ prompt appears so that user can enter next command. Hence the shell is in a continuous sleep – waking – waiting cycle

Some of the features of the shell are explored here.

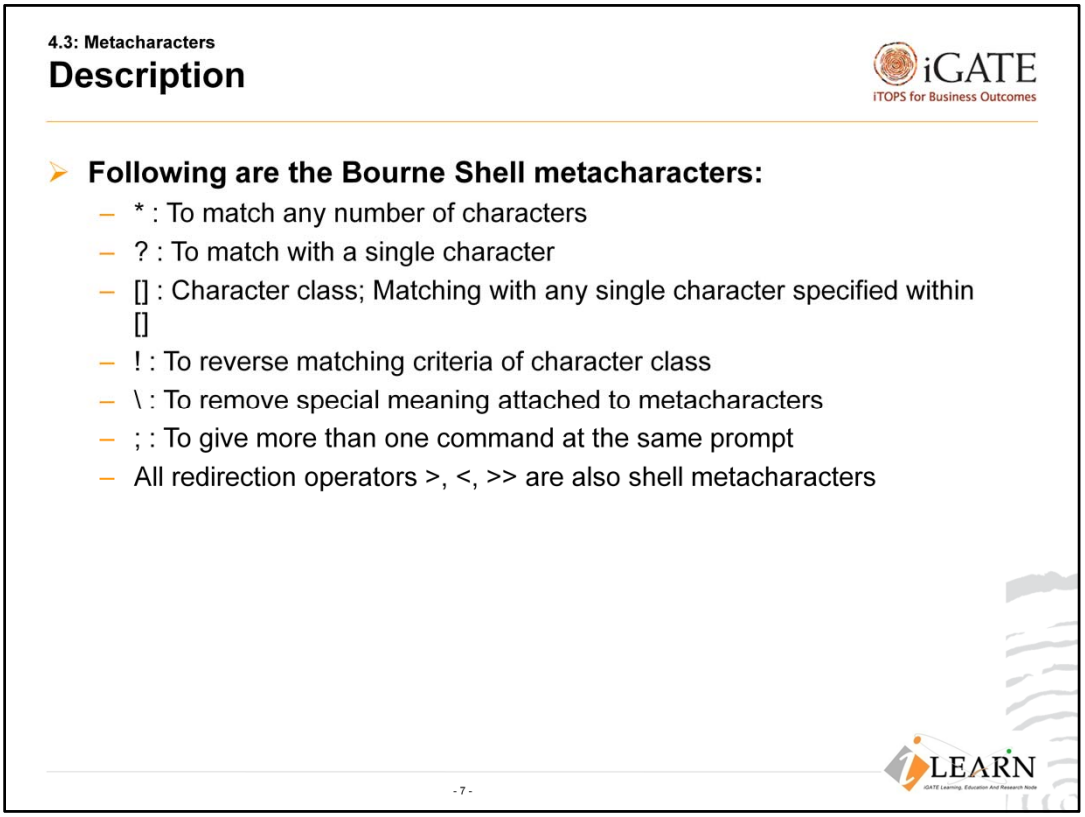
Working of Shell (contd..)

iGATE
iTOPS for Business Outcomes

- **Continuous sleep-waking-waiting cycle**
- **Performs following activities:**
  - Issues a $ prompt & waits for user to enter a command.
  - After user enters command, shell scans & processes the command.
  - The command is passed on to the Kernel for execution & the shell waits for its conclusion.
  - The $ prompt appears so that the user can enter next command.

iLEARN
iGATE Learning, Education And Research Node

- 6 -



Shell Metacharacters for pattern matching and combining commands:
Metacharacters are characters to which the shell attaches special meaning. The shell interprets these metacharacters and the command is rebuilt before it is passed on to the kernel.

Wildcards for Pattern Matching:
The wildcard * is used to match any number of characters (including none). It however, does not match patterns beginning with a dot (.).
The wildcard ? matches a single character.
Examples:
emp* : This would match all patterns that begin with emp, and may be followed by any number of characters (like emp, emppune, empttc, empseepz etc).
emp? : This would match all patterns that begin with emp and are followed by exactly one more character, which could be anything (like emp1, empa etc).
Patterns can be made more restrictive by using character class, represented by []. Any number of characters can be specified within the [], but the matching would occur for a single character within a class.
emp[abc]: This would match with patterns that begin with emp followed by a or b or c any one of it. (like empa, empb or empc)

Contd..

Escaping with the Backslash:

In order to remove the special meaning attached to metacharacters, the backslash can be used. For example, the expression emp* would match only with the pattern emp*. In the given example \ removes special meaning of *(i.e. 0 or more characters) and treat it as a character '*'

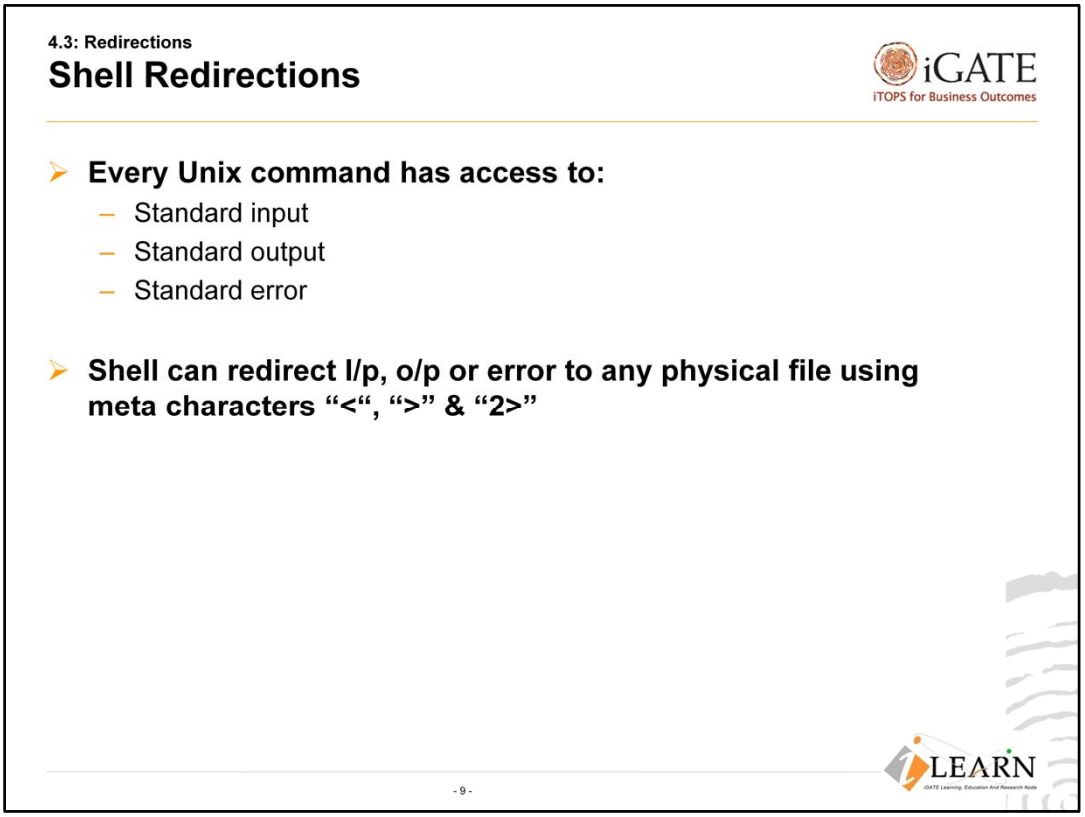
Combining command using the semicolon:

It is possible to give more than one command at the same prompt so that they will be executed in sequence one after the other. This is done with the character ; as in the example below:

Example: Using the wild cards:

```
$ ls -l file2.txt ; chmod u+x file2.txt ; ls -l file2.txt
-rwxr--r--   1 deshpavn group         61 Mar 29 10:44 file2.txt
$
```

Here it is possible to assign permissions and subsequently check the same.



Redirections by Shell:

Many commands work with character streams. The default is the keyboard for input (standard input, file number 0), and terminal for the output (standard output, file number 1). In case of any errors, the system messages get written to standard error (file number 2), which defaults to a terminal.

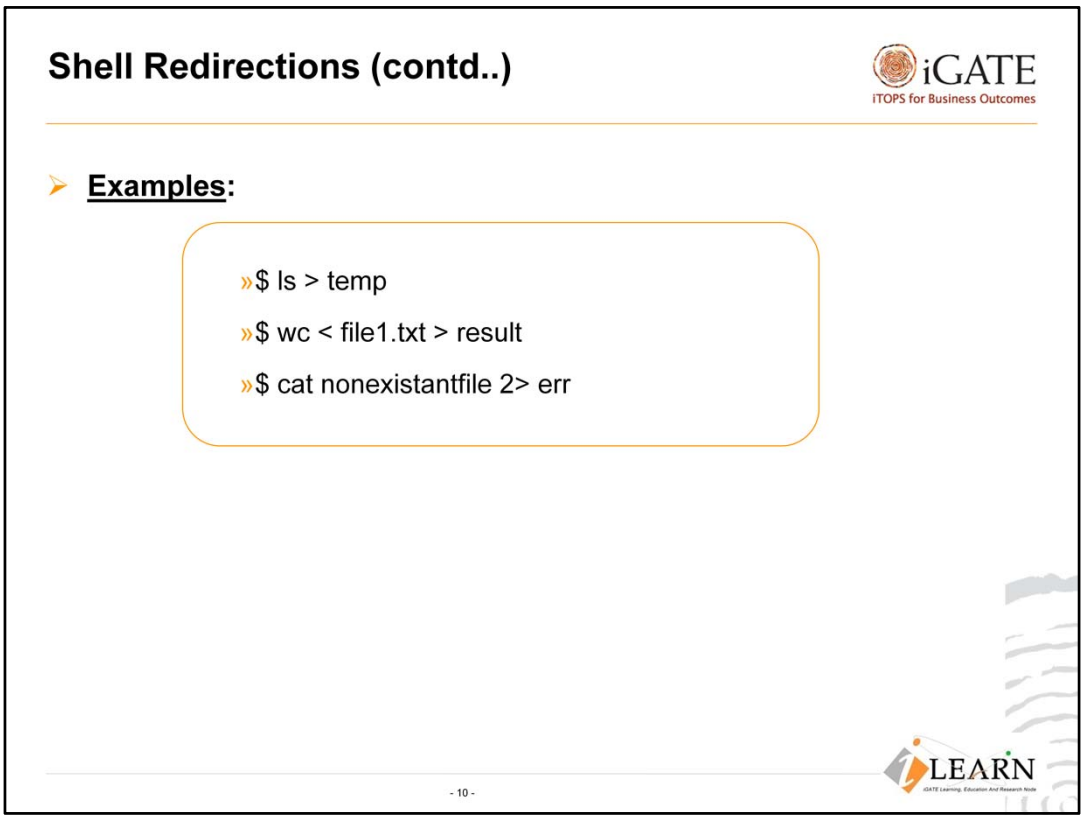
Unix treats each of these streams as files, and these files are available to every command executed by the shell. It is the shell's responsibility to assign sources and destinations for a command. The shell can also replace any of the standard files by a physical file, which it does with the help of meta characters for redirection.

In order to take the standard input from a file, the character < is used.

```
$ wc < file1.txt
    2    12    60
```

To redirect the output to a file, > is used. If outfile does not exist, it is first created; otherwise, the contents are overwritten. In order to append output to the existing contents, >> is used.

```
$ wc < file1.txt > result
$ cat result
    2    12    60
```

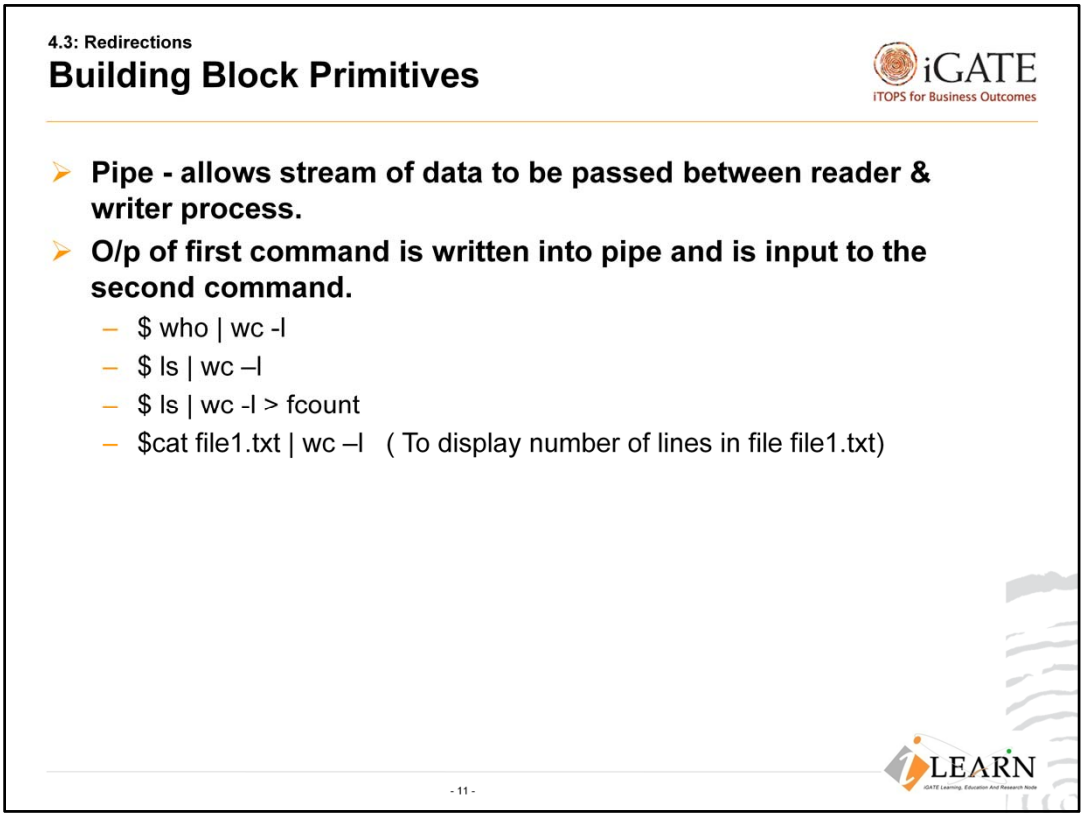


Using redirections:

```
$ wc < cfile1.lst >> result
$ cat result
    2    12    60
    4     4     8
```

As shown above, it is possible to combine redirection operators on a single command line. The order of the redirection symbols does not affect the working of the command.
Using redirections:
To redirect the standard error, 2> is used.

```
$ cat xyzfile
cat: cannot open xyzfile: No such file or directory (error 2)
$ cat xyzfile 2> errfile
$ cat errfile
cat: cannot open xyzfile: No such file or directory (error 2)
$ cat xxfile 2>> errfile
$ cat errfile
cat: cannot open xyzfile: No such file or directory (error 2)
cat: cannot open xxfile: No such file or directory (error 2)
```



Connecting commands with Pipes:

The shell has a special operator called pipe (|), using which the output of one command can be sent as an input to another. The shell sets up the interconnection between commands. It eliminates the need of temporary files for storing intermediate results.

Creating a Pipeline:

```
$ who | wc -l
       8
```

When a sequence of commands are combined this way, a pipeline is said to have been formed.

It is possible to combine redirection along with the pipe.

Combining pipe with redirection:

```
$ ls | wc -l > output
$ cat output
    13
```

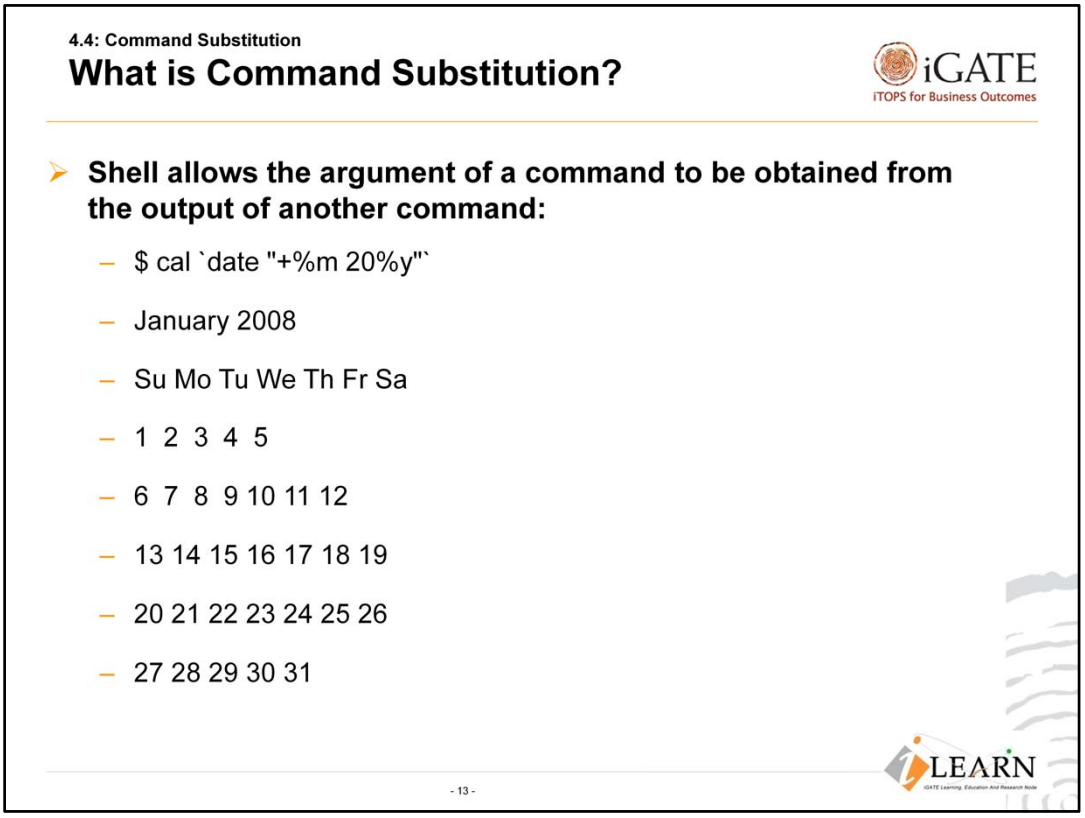
Building Block Primitives (contd..)

iGATE
iTOPS for Business Outcomes

- | - pipe symbol
- Any number of commands can be combined together to make a single command.

iLEARN

- 12 -

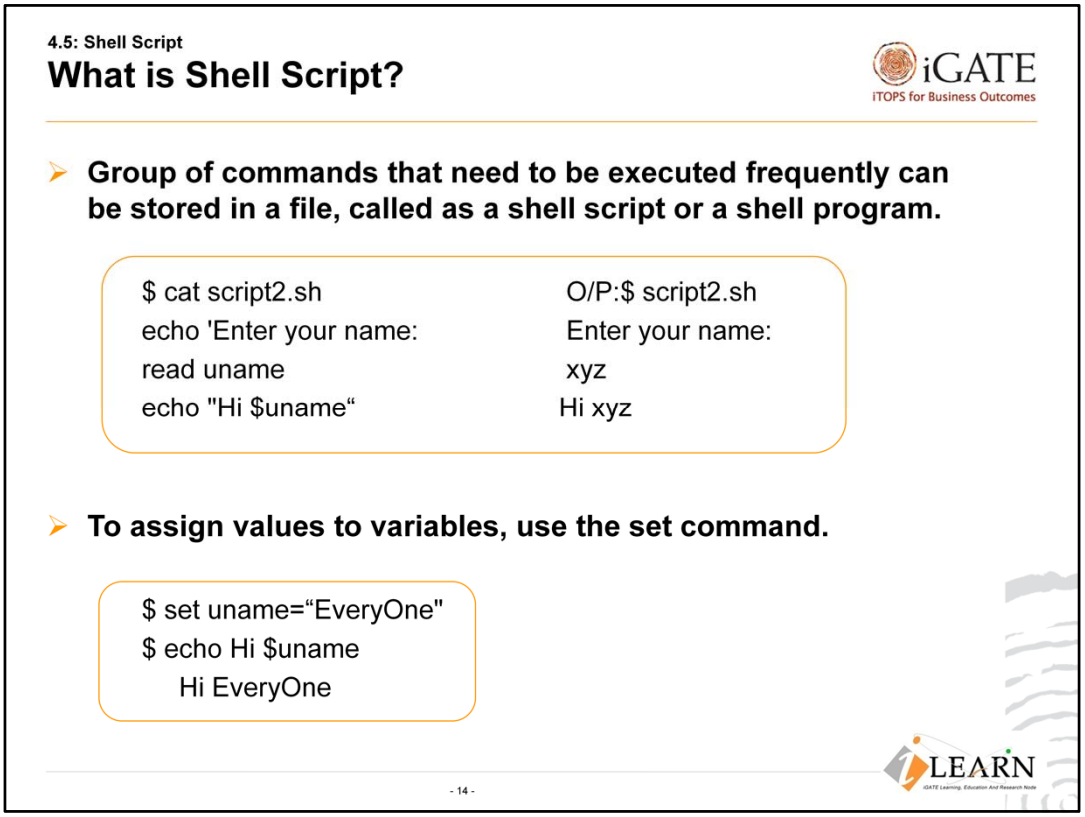


What is Command Substitution?

The shell allows the argument of a command to be obtained from the output of another command – this feature is called command substitution. This is done by using a pair of backquotes. The backquoted command is executed first. The output of command is substituted in place of command. Then outer command will get executed.

Example:

```
$ cal `date "+%m 20%y"`
   March 2001
Su Mo Tu We Th Fr Sa
          1  2  3
 4  5  6  7  8  9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24
25 26 27 28 29 30 31
```



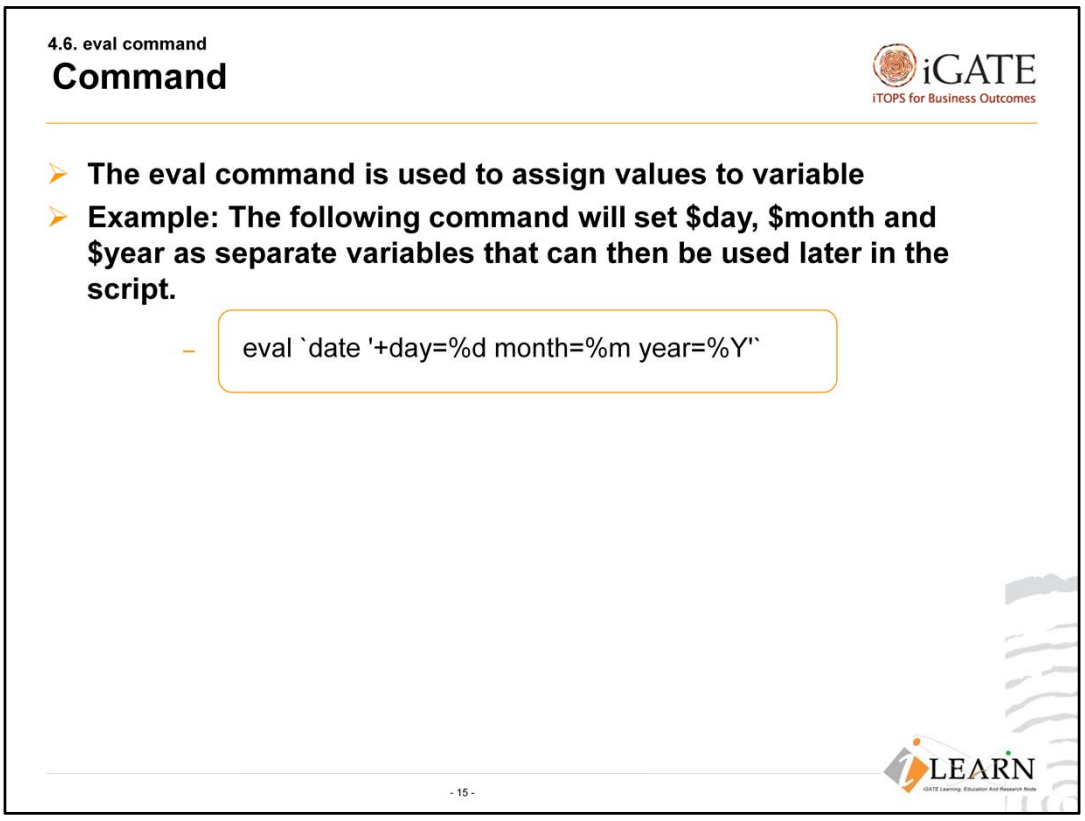
User defined shell variables and the echo command:

Since the shell is programmable, it is possible for users to define their own variables. No type is associated with the shell variables. The value of a variable is always string type.

Shell variables are assigned values with the = operator, in the form variable=value. [There should be no spaces on either side of =]. Variable names can be combination of letters, digits & underscore, but the first character has to be a letter. Shell is sensitive to case. Even though the value is a string, if it contains only numerals, it can be used for numerical computation.

In order to retrieve the value stored in a variable, the $ sign needs to be used. The command echo can be used to display messages, as also the values of variables.

```
$ echo hi
hi
$ echo hi there
hi there
$ echo "hi there"
hi there
$ set msg=“Hi there"
$ echo $msg
  Hi there
```



Using the shell eval command:

In shell scripts it is common to set variables using the output of a command, Example:

```
variable=`command`
```

The output from the command is normally a single value. If, however, the command returns multiple values you need to use some other program (e.g. awk) to split the combined result into separate parts. Alternatively, you could pass an argument to the command to specify which result should be returned.

A neater solution is to use the shell's eval command. For example, say you wanted to return day, month and year from the date command. You'd have to write either:

```
day=`date +%d`
month=`date +%m`
year=`date +%Y`
```

or:

```
result=`date '+%d %m %Y'`
```

and then break up $result with awk or cut for example. Instead you can write:

```
eval `date '+day=%d month=%m year=%Y'`
```

This will set $day, $month and $year as separate variables that can then be used later in the script.

Summary

iGATE
iTOPS for Business Outcomes

- In UNIX different types of shells are available: CSH, KSH and Bourne Sh.
- Redirection operator can be used to redirect i/p or o/p to files or printer.
- Pipeline character can be used to send o/p of one command as i/p of another command.
- Group of commands that need to be executed frequently are stored in a file, called as a shell script.

iLEARN

- 16 -

Add the notes here.

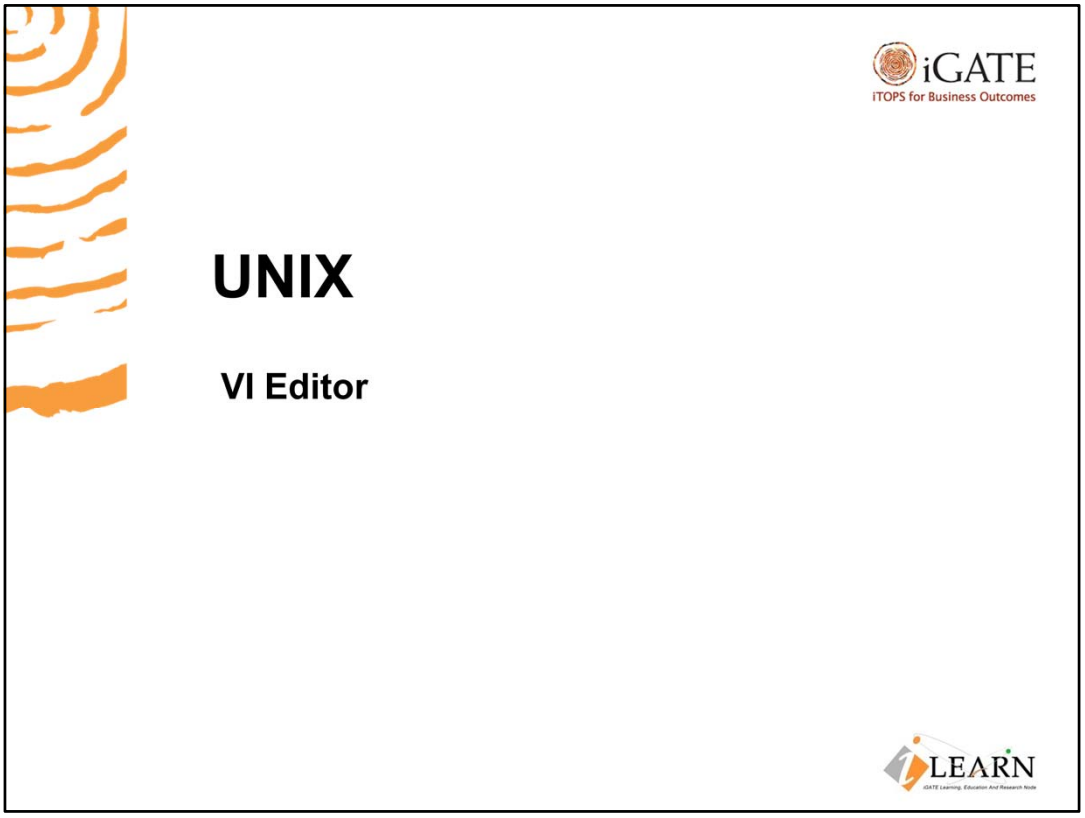
Review Questions

iGATE
iTOPS for Business Outcomes

- Question 1: In shell, what are the different metacharacters available?
- Question 2: _____ symbol is used as output redirection.
- Question 3: _____ symbol is used as command substitution operator.

iLEARN

- 17 -


iGATE
iTOPS for Business Outcomes
UNIX
VI Editor
iLEARN

Lesson Objectives

iGATE
iTOPS for Business Outcomes

- **Different modes of vi editor**
  - Input
  - Command
  - Esc mode
- **Input mode commands**
- **Vi editor – Save & Quit**
- **Navigation commands**
- **Paging functions**
- **Search and repeat commands**

iLEARN
iGATE Learning, Education And Research Node

- 2 -

Lesson Objectives

iGATE
iTOPS for Business Outcomes

- Vi editor – Other Features
- SED – Introduction to SED
- SED Commands

- 3 -

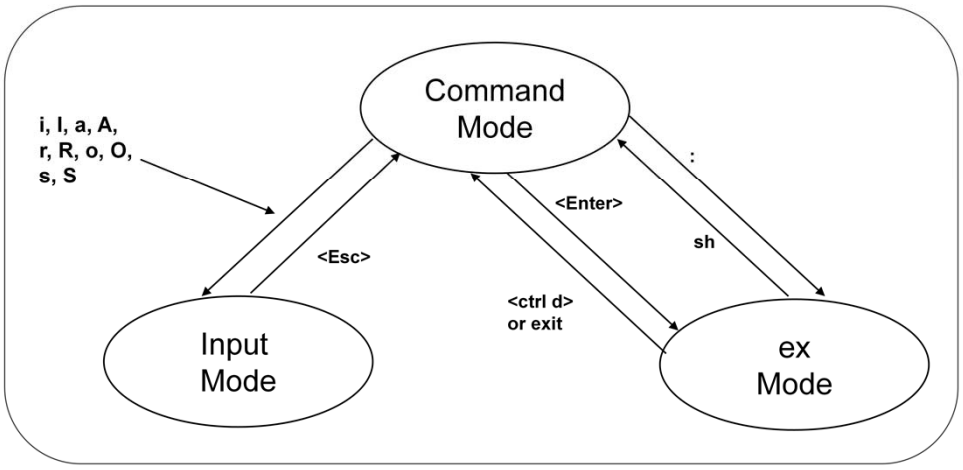
iLEARN
iGATE Learning, Education And Research Node

5.1: Modes of Vi Editor

Introduction

iGATE
iTOPS for Business Outcomes

- **Three Modes of Vi Editor are:**



iLEARN

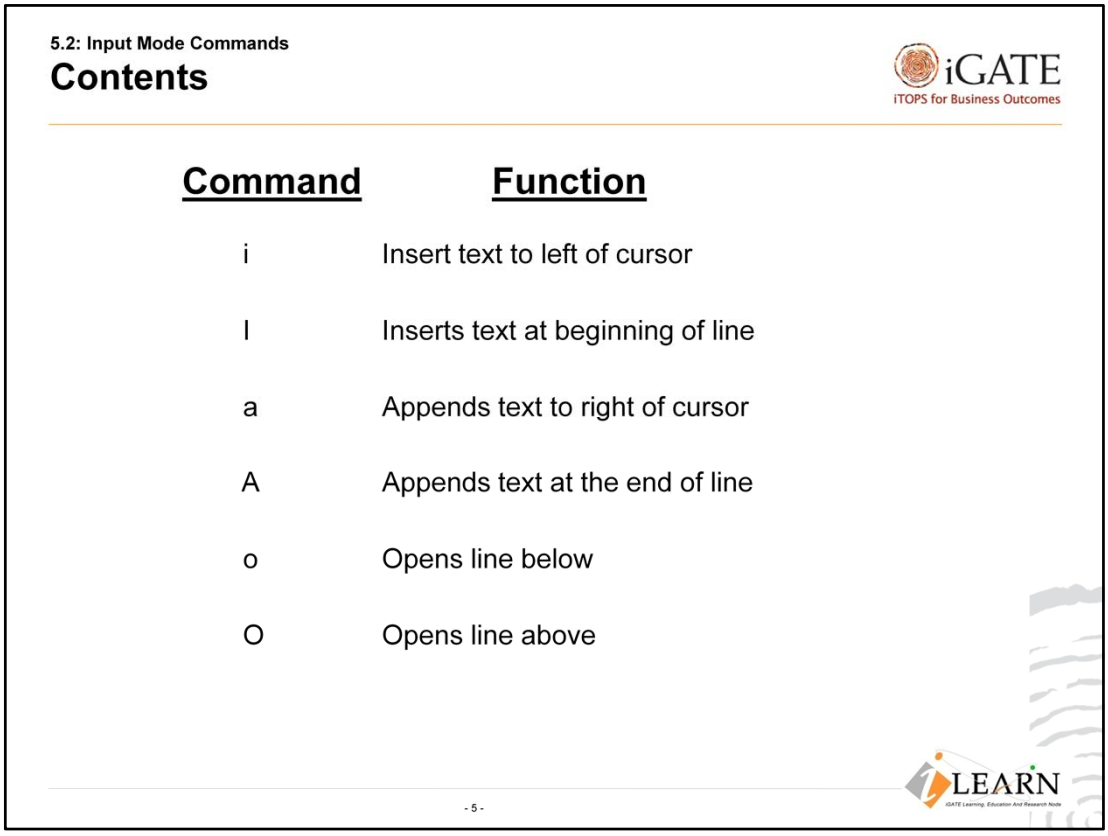
- 4 -

What is the vi Editor?

Line editors, full screen editors and stream editors are all available on the Unix system. The editor “ed” was developed by Ken Thompson, and was the original editor that accompanied the Unix system. The line editor “ex” was created by William Joy, on the basis of “ed”. This chapter discusses the “vi” editor, which is a full screen editor, widely acknowledged as one of the most powerful editors available in any environment.
The vi editor is also created by William Joy, and is in fact simply the visual mode of the line editor “ex”. It offers innumerable functions, but the terseness of its commands is considered to be a major handicap.

Modes of vi editor:

The vi editor works in 3 modes: Input, Command and ex mode. The relation between the three modes is depicted in the figure in the slide above.
Command Mode, Input Mode, ex Mode



What is the Input Mode?

The Input mode is used to insert, append, replace or change text. A summary of input mode commands are given below:

Command	Function
• i	Inserts text to left of cursor
• I	Inserts text at beginning of line
• a	Appends text to right of cursor
• A	Appends text at end of line
• o	Opens line below
• O	Opens line above
• rch	Replaces single character at cursor with character ch
• R	Replaces text from cursor to right
• s	Replaces single character at cursor with any number of characters
•S	Replaces entire line

Contents (contd..)

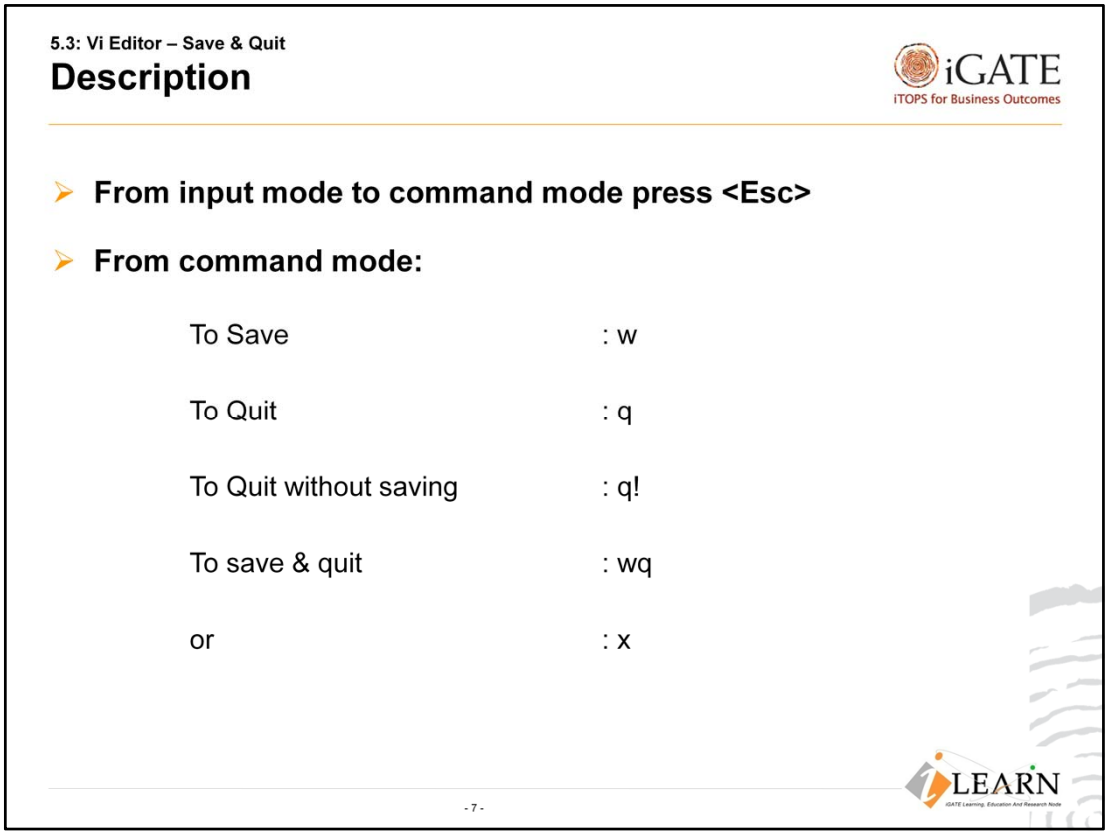
iGATE
iTOPS for Business Outcomes

Command	Function
r	Replaces single character under cursor with character (no<Esc>)
R	Replace text from cursor to right

iLEARN
iGATE Learning, Education And Research Node

- 6 -

Add the notes here.



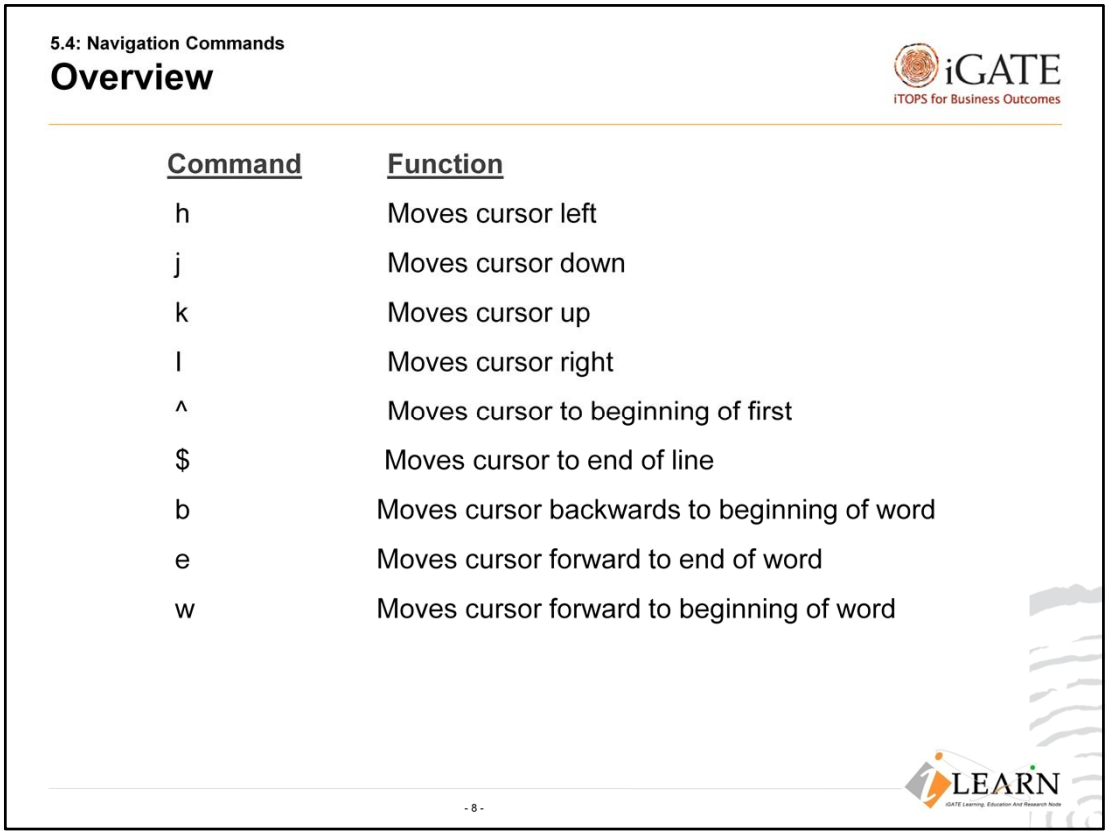
Saving and Quitting: The Last Line Mode

vi uses the ZZ command to save and quit editor. The 'ex' mode, also referred to as last line mode, can also be used.
To switch from command mode to ex mode, a colon (:) is pressed, which appears as ex prompt in the bottom line. Any ex command can be entered at this prompt. Following commands can be used for saving and quitting from the ex mode:

Command	Function
• w	Write buffer into disk and remain in editing mode
• x	Save and quit the editor
• wq	Write and quit editor
• q	Quit editor

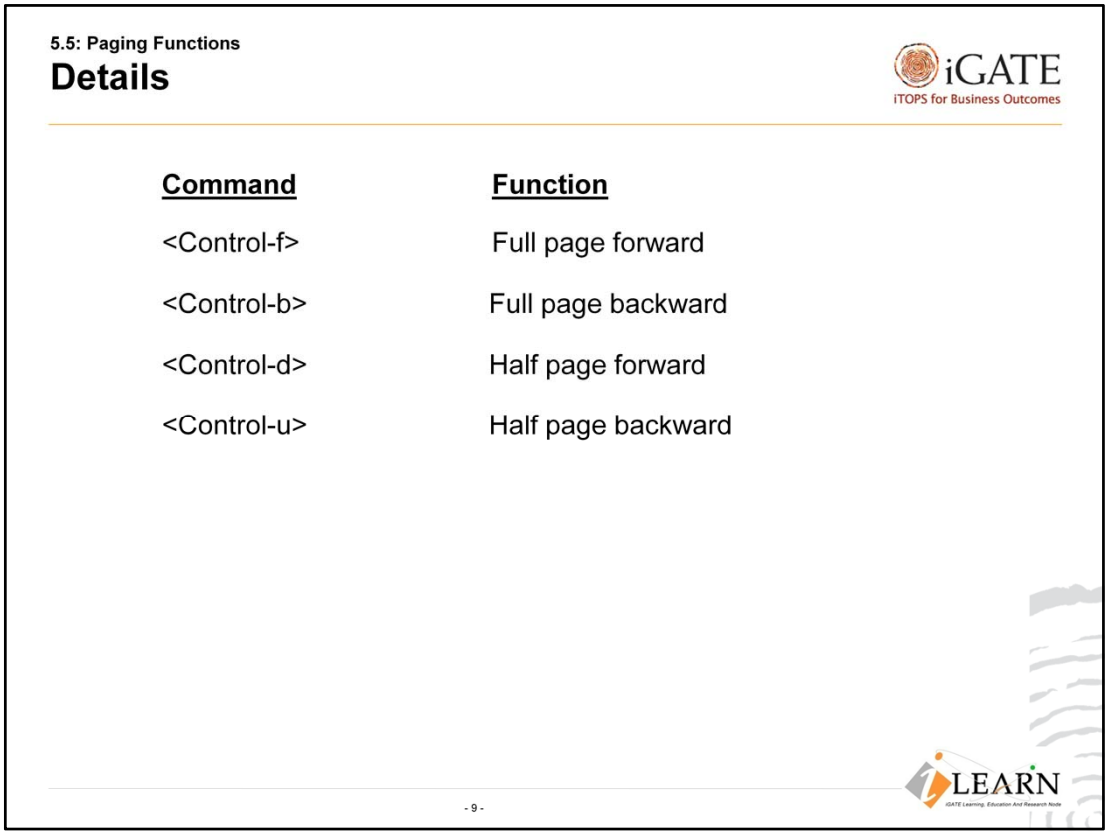
The Repeat Factor

A number can be prefixed to any command: most commands will interpret the instruction to repeat the command that many times. Hence this number is called as the repeat factor. For example, to insert a series of 30 asterisks in a line, 30i* can be used.
The repeat factor can be used with input as well as command mode.



Navigation (Cursor Movement commands):

Command	Function
• h (or backspace)	Move cursor left
• j	Move cursor down
• k	Move cursor up
• l (or spacebar)	Move cursor right
• ^ (no repeat factor)	Move cursor to beginning of first word of line
• 0 or |	Move cursor to beginning of line (no repeat factor with 0)
• $	Move cursor to end of line
• b	Move cursor back to beginning of word
• e	Move cursor forward to end of word
• w	Move cursor forward to beginning of word



Paging Functions:

Command	Function
• <Ctrl-f>	Full Page forward
• <Ctrl-b>	Full Page backward
• <Ctrl-d>	Half Page forward
• <Ctrl-u>	Half Page backward
• <Ctrl-l>	Redraw page screen (no repeat factor)

5.6: Search and Repeat Commands

Details

iGATE
iTOPS for Business Outcomes

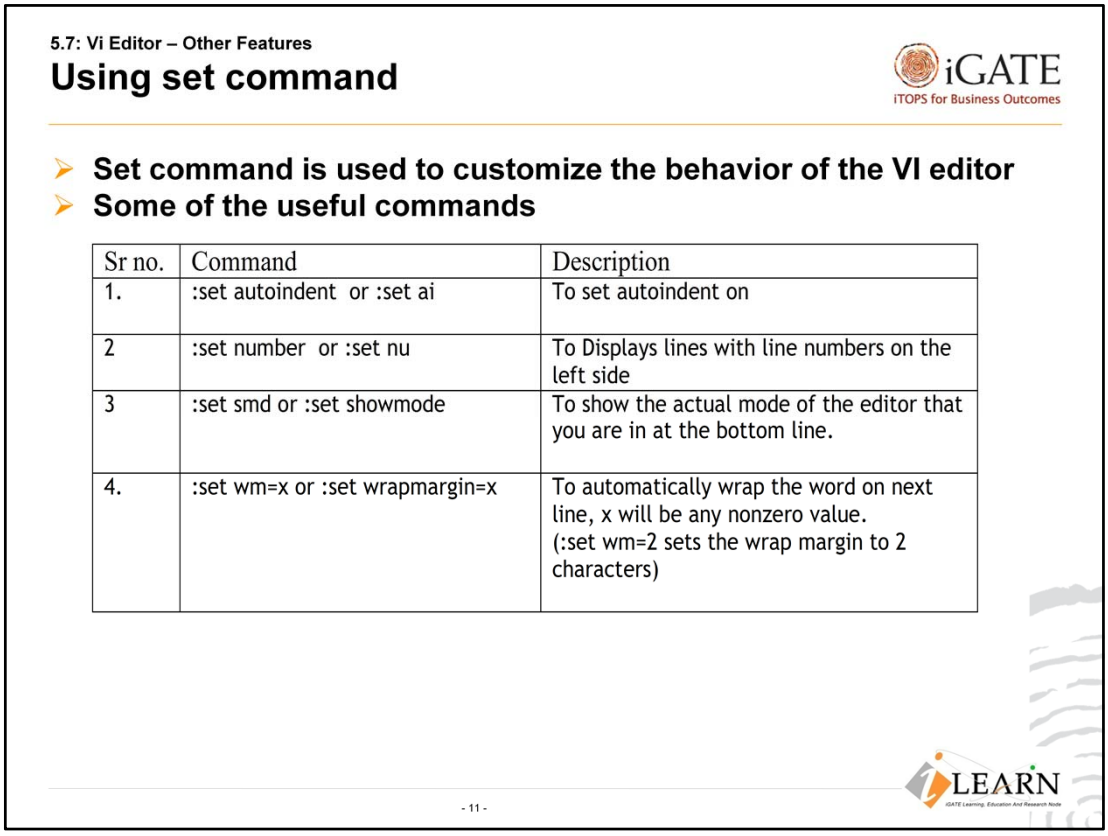
Commands	Functions
/pat	Searches forward for pat
?pat	Searches backward for pattern pat
n	Repeats search in the same direction along which the previous search was made (no repeat factor)
N	Repeats search in a direction opposite to that which the previous search was made (no repeat factor)

iLEARN
iGATE Learning, Education And Research Node

- 10 -

Search and repeat commands:

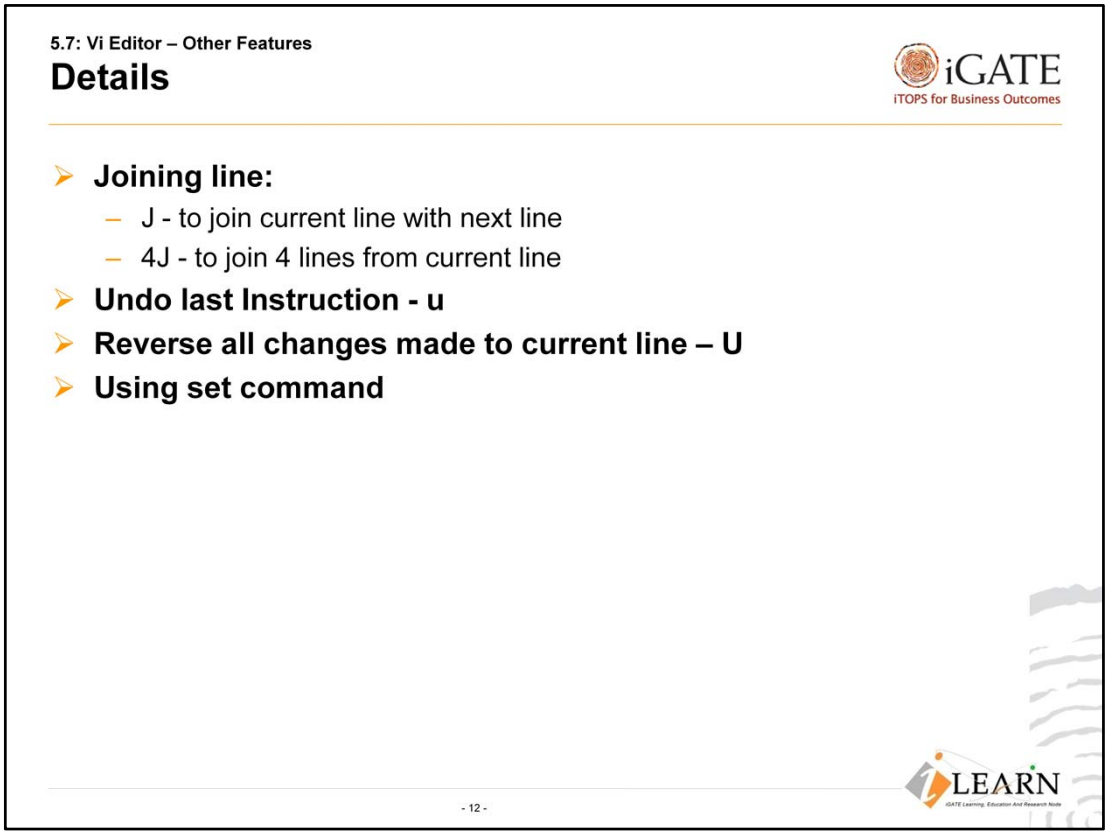
Command	Function
• */pat*	Searches forward for pattern *pat*
• *?pat*	Searches backward for pattern *pat*
• n	Repeats search in same direction as previous search (no repeat factor)
•N	Repeats search in opposite direction as previous search (no repeat factor)
•f*ch*	Moves cursor forward to first occurrence of character ch in current line



To edit the behavior of vi editor. There are many options which can be used with :set command

To get the list of all options in set command use
:set all
Some more commands

Sr no.	Command	Description
1.	:set noautoindent or :set noai	To unset autoindention
2.	:set nomesg	Turn off messages, so that nobody can bother you while using the editor.
3.	:set warn	To warns you if you have modified the file, but haven't saved it yet.
4.	:set tabstop=x or :set ts=x	To set the tabstop to x spaces (:set tabstop=8 the tab key will display 8 spaces)
5.	:set ignorecase or :set ic	To set ignore case by default while searching
6.	:set noignorecase or :set noic	To unset ignore case option
7.	:set linelimit=1048560	To set the maximum file size to edit
8.	:set list	To display hidden character like tabs or end of the line
9.	:set nolist	To display hide character like tabs or end of the line



Operators

vi uses a number of operators which can be used along with commands to perform complex editing functions. The most commonly used operators are:

d – delete
c – change
yy – yank (copy)
! – filter to act on text

Operators can work only when combined with a command or itself. The operators also take a repeat factor.

Some samples of using operators:

Command	Function
d$ or D	Deletes from cursor to end of line
5dd	Deletes five lines
d/*endif*	Deletes from cursor up to the first occurrence of the string *endif* in the forward direction
d30G	Deletes from cursor up to line number 30
df.	Deletes from cursor to first occurrence of a dot
c0	Changes from cursor to beginning of line
c$ or C	Changes from cursor to end of line
3cw or c3w	Changes three words

5.8: SED – Introduction to SED

- **SED("Stream EDitor") is a non-interactive stream oriented editor for filtering and transforming text.**
- **It reads input line by line, applying the operation which has been specified via the command line (or a *sed script*), and then outputs the line in a terminal or file.**
- **When to use SED?**
  - To automate editing actions to be performed on one or more files.
  - To simplify the task of performing the same edits on multiple files.
  - To write conversion programs.

5.9: SED Commands

Invoking SED using Command Line

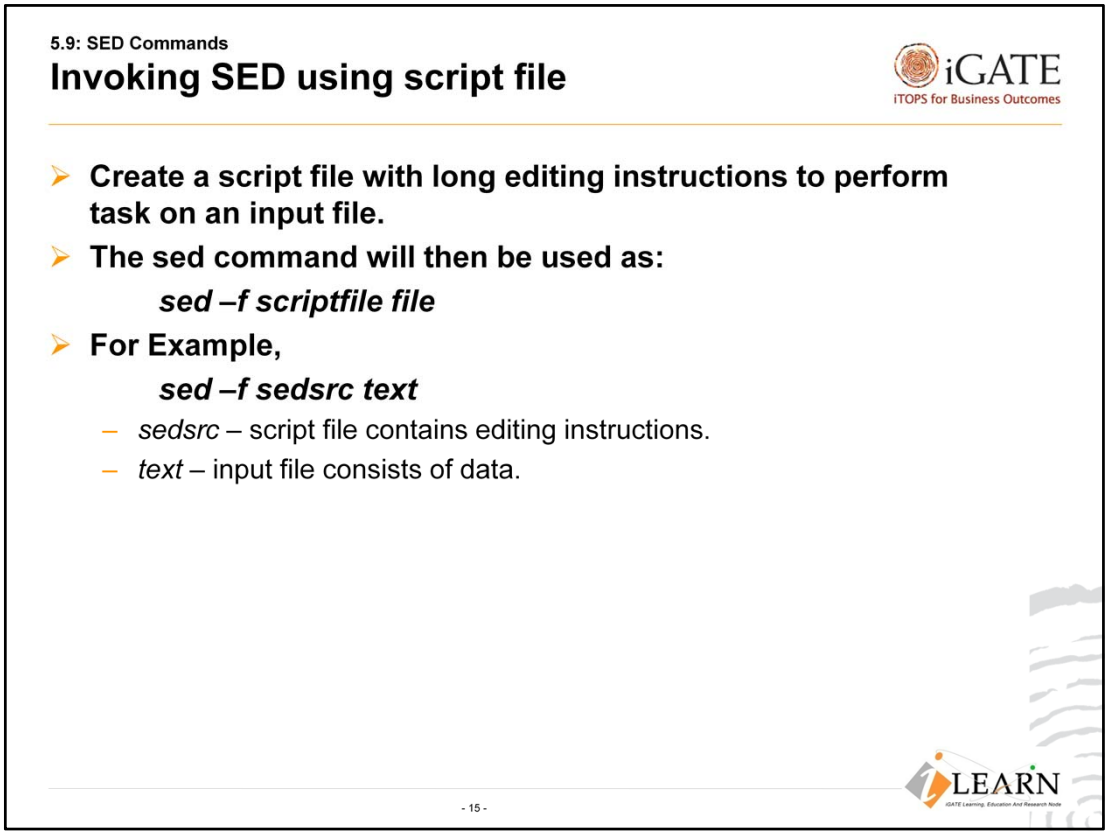
iGATE iTOPS for Business Outcomes

Syntax of SED Command

sed *options sed-script* filename

- sed-script -> sed can use regular expressions for manipulating text on the input file.
- Options:
  - -n Suppress the default output.
  - -e Script is an edit command for sed . Used to specify multiple instructions by preceding with –e.

iLEARN iGATE Learning, Education And Research Node



The sed command will then be used as:

sed –f scriptfile file

All the editing command that we need to execute are placed in a file, as shown:

```
$ cat sedsrc
s/ WB/, West Bengal/
s/ BH/, Bihar/
s/ MH/, Maharashtra/
```

The following command reads all of the substitution commands in the sedsrc and applies them to each line in the input file "text":

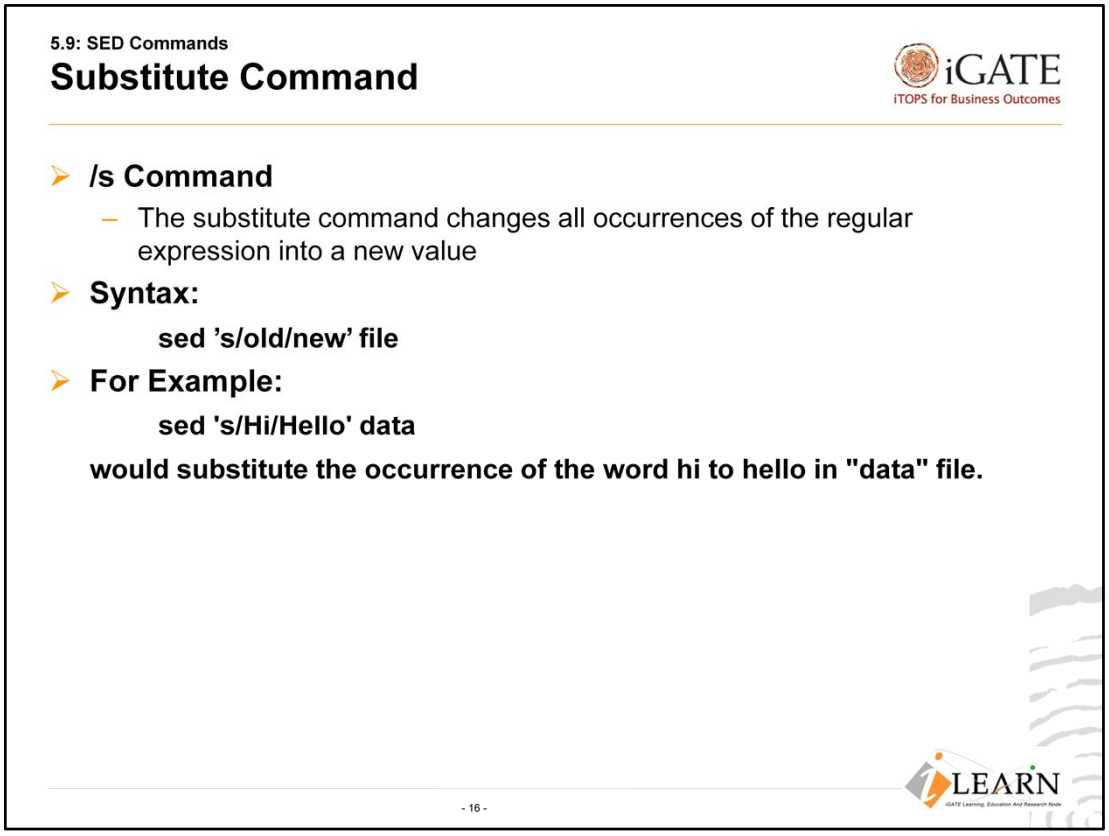
$ sed –f sedsrc text

```
Sidd B-1/250 Kalyani, West Bengal
Tito A-3/11 Thane, Maharashtra
Rayn D-17 LakeTown, West Bengal
Miter C/268 G.B.Road, Bihar
```

The above command will display the output in the Terminal.

The following command used for Redirecting the output to a file:

$ sed –f sedsrc text > newtext

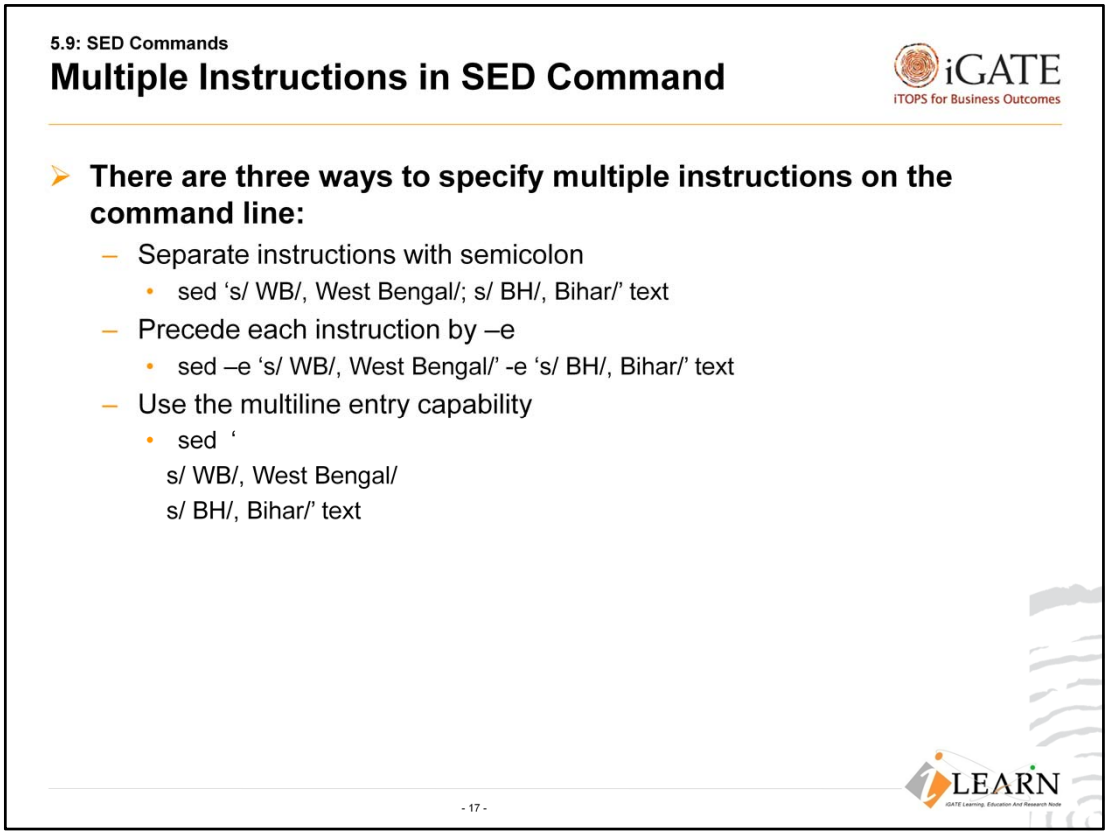


Let there be a text file called "text". The content of the file is as shown:
Sidd B-1/250 Kalyani WB
Tito A-3/11 Thane MH
Rayn D-17 LakeTown WB
Miter C/268 G.B.Road BH

The substitution command in sed:
$ sed 's/WB/WestBengal/' text

Two lines are affected by the instruction but in the above example all lines will be displayed. Enclosing the instruction in single quotes is not mandatory but its required if the substitution command contains spaces:
$ sed 's/ WB/, West Bengal/' text

g option: Used to make the command replace in all the instance of the word instead of first occurrence of the word in each input line.
$ sed 's/rat/cat/g' temp
cat cat
For example if we want to change 'rat' to 'cat' in lines that contain the word 'dog' we say:
$ sed '/dog/s/rat/cat/g' temp
$ sed 's/rat/cat/4' # replaces only 4th instance in a line



There are three ways to specify multiple instructions on the command line:
1. Separate instructions with semicolon
sed 's/ WB/, West Bengal/; s/ BH/, Bihar/' text

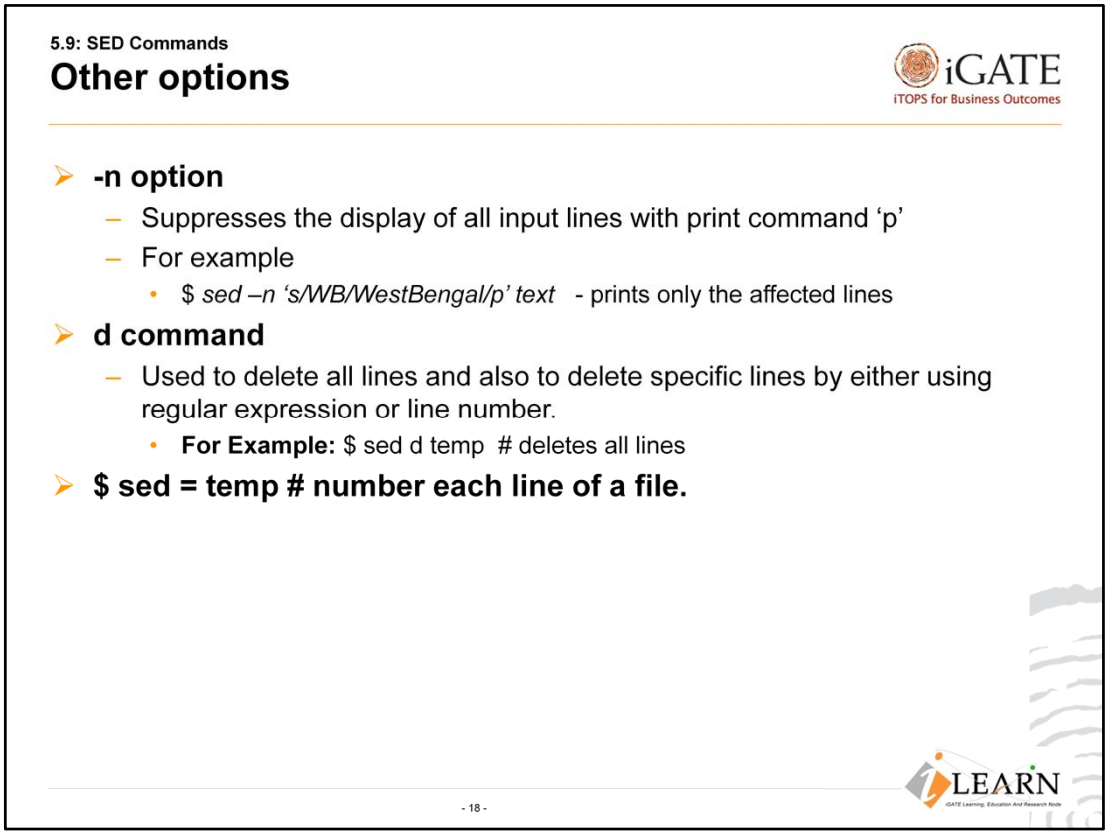
2. Precede each instruction by –e
sed –e 's/ WB/, West Bengal/' -e 's/ BH/, Bihar/' text

3. Use the multiline entry capability
sed '
s/ WB/, West Bengal/
s/ BH/, Bihar/' text

It is very easy to make mistake in the instruction or omit a required element.

$ sed 's/ WB/, West Bengal' text
sed: command garbled: s/ WB/, West Bengal

Notice the error message. Sed usually display any line that it cannot execute, but it does not tell what is wrong with the command. Here a slash at the end is missing.



The –n option suppresses the automatic output. When using this option each line needed to produce output must contain the print command, **p.**
$ sed –n 's/WB/WestBengal/p' text
Sidd B-1/250 Kalyani WestBengal
Rayn D-17 LakeTown WestBengal

Here only the lines that were affected were printed.
For printing only line 2 and 3, the command used is:

$ sed –n 2,3p text

If –n option is not present all the lines will get printed and line from 2 to 3 will get printed twice.
d used to delete all lines and also to delete specific lines by either using regular expression or line number.
$ sed d temp # deletes all lines
$ sed '$d' temp #delete last line.
$ sed '1d' temp #delete first line.
$ sed '/^$/d' temp #delete all blank lines.

# number each line of a file (simple left alignment).
$ sed = temp

5.9: SED Commands

More Commands

iGATE
iTOPS for Business Outcomes

Sl.No	Command	Description
1	sed 10q temp	print first 10 lines of file(emulates behavior of "head")
2	sed q temp	print first line of file(emulates "head -1")
3	sed '$!d' temp # method 1 sed -n '$p' temp # method 2	Prints last line of a file(emulates "tail -1")
4	sed '$!N;s/\n/ /' temp	join pairs of lines side-by-side (like "paste")
5	sed '$!N; s/^\(.*\)\n\1$/\1/; t; D' temp	Delete all lines except duplicate lines (emulates "uniq -d").
6	sed '1,10d' temp	delete the first 10 lines of a file

iLEARN
iGATE Learning, Education And Research Node

- 19 -

Summary

iGATE
iTOPS for Business Outcomes

- **In vi editor:**
  - esc key is used to change the mode.
  - esc - $ is used to move cursor at the end of the file.
  - Wq is used to write (save) and quit from the file.
  - q! is used to quit without saving.
- **SED**
  - Commands used to process the data.
    - Command line instruction
    - Script file based instruction

iLEARN
iGATE Learning, Education And Research Node

- 20 -

Add the notes here.

Review Questions

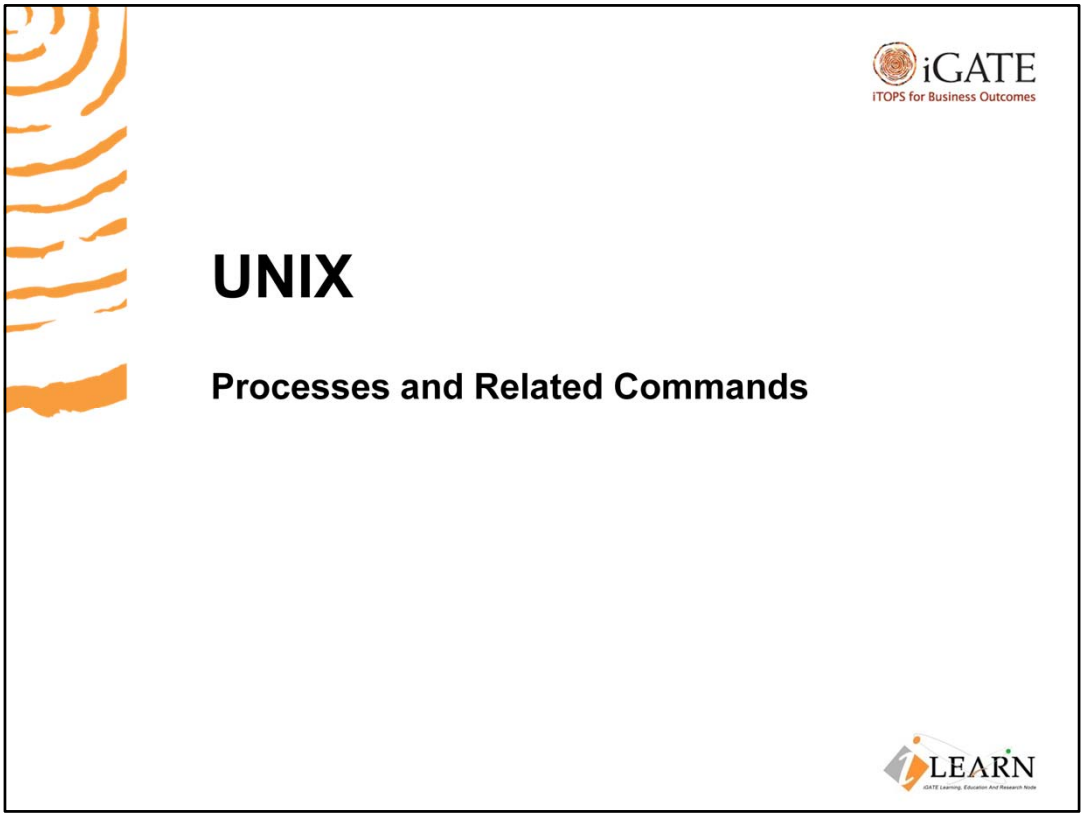
iGATE
iTOPS for Business Outcomes

- What command is used to copy the lines in vi editor?
- _____ command search for the pattern in vi editor in forward direction?
- What is the <control b> command used for?
- VI editor is stream Oriented?
  - True
  - False

iLEARN

- 21 -

Add the notes here.


iGATE
iTOPS for Business Outcomes
UNIX
Processes and Related Commands
iLEARN

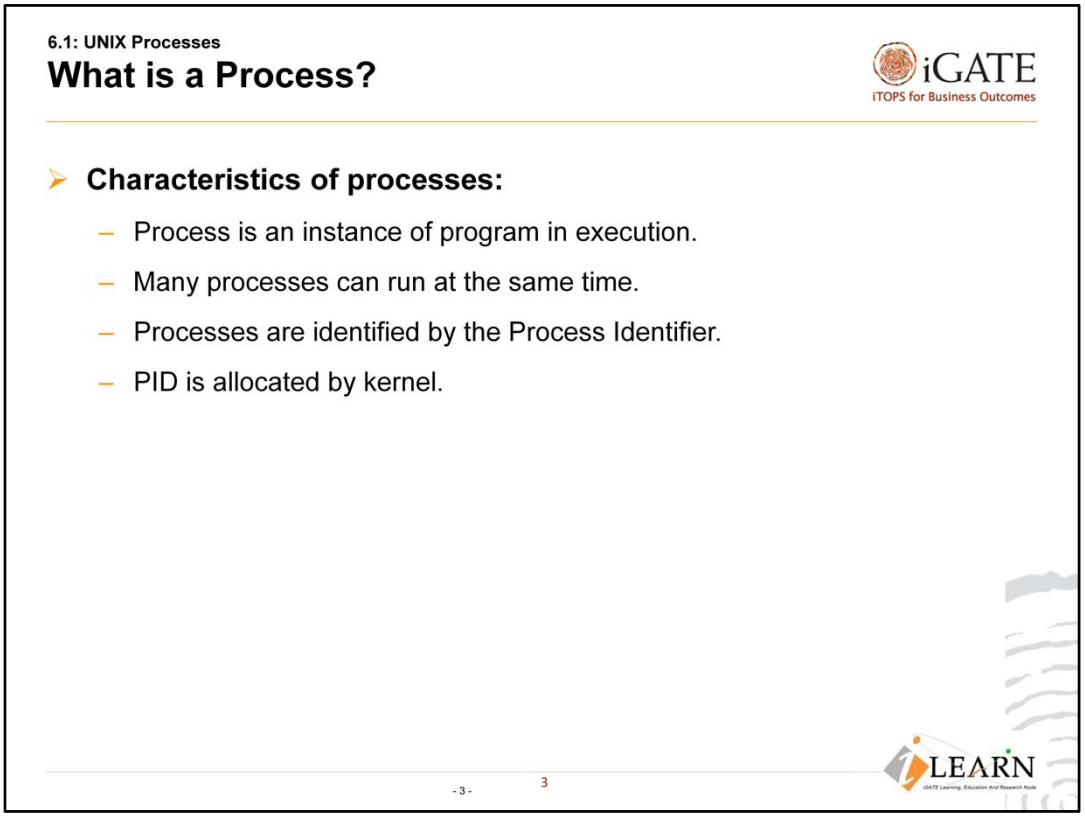
Lesson Objectives

iGATE
iTOPS for Business Outcomes

- **UNIX processes:**
  - Parent and Child processes
  - Process Status Command – ps
  - Running processes in background mode
  - Terminate process
  - Process scheduling

iLEARN

- 2 -



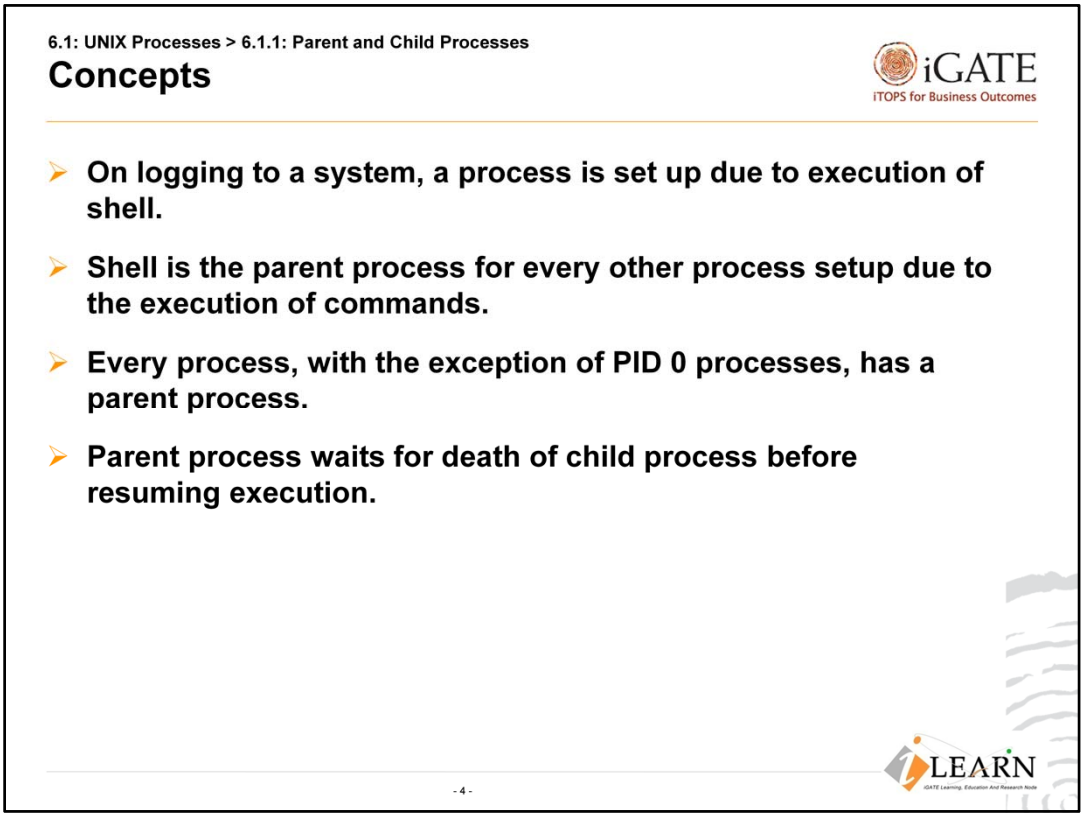
UNIX Processes:

When you execute a program on your UNIX system, the system creates a special environment for that program. This environment contains everything needed for the system to run the program as if no other program were running on the system.

What is a Process?

A process is an instance of a program in execution. When any executable file is executed, a process starts. It remains active while the program is executing. When the program terminates, the process dies. Generally the name of the process is the name of the executable.

Since Unix is a multi-tasking system, there can be many processes that run at the same time. A unique number called as the Process Identifier, PID, identifies each of these processes. The kernel allocates this PID, which is a number from 0 to 32767. It is the responsibility of the kernel to manage the processes – in terms of time allocated to process, its associated priorities and swapping etc.



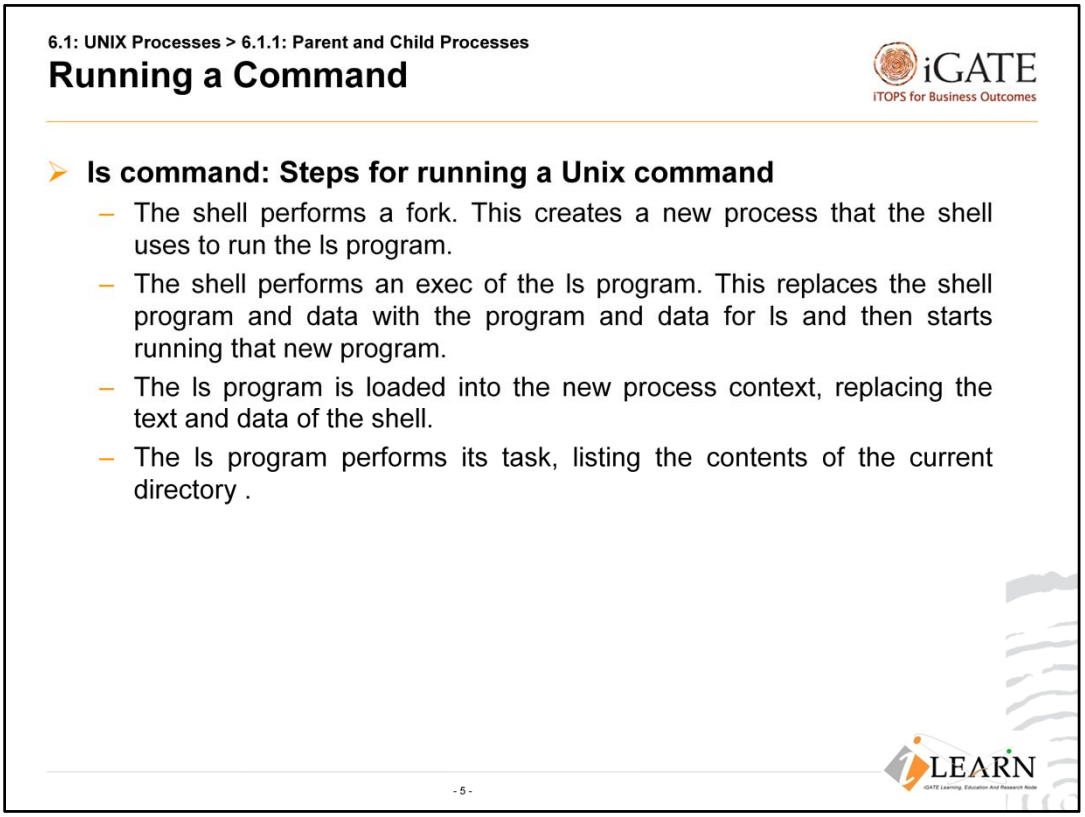
Parent and Child Processes:

When a user logs on to the system, the kernel sets up a process – this is actually the process set up due to execution of sh command. This process remains active till the user logs off. The sh process is identified by the special variable $$. Any logging off and logging on again will result in an assignment of a different PID. The knowledge of this PID is required for controlling the activities of the terminal.

Any command written at the prompt is actually the standard input to the sh program. When a external command is run from the command line, the shell process spawns a new process for the command, which remains active till the command is active. The shell here is the parent process while the new process is the child process. The child process inherits the environment of the parent process. The child process can alter the operating environment that it has inherited, but the modified environment will not be available to the parent after the death of the child process.

Every process has a parent. The exception is the first process, with PID 0, which does not have a parent. This is set up when the system boots. It can be treated as analogous to the “root” in the file system.
Every process can have only one parent process. However, a process can spawn multiple processes. For example, when a pipeline is set up between 2 commands, the sh process would set up two processes for each of the commands.

On the completion of the child process, a signal is sent to the parent, and the control is reverted back to the parent process. However, if the parent process dies, the child processes automatically die.



Running a Command:

When you enter ls to look at the contents of your current working directory, UNIX does a series of things to create an environment for ls and the run it.
Internally, Unix creates a process with the fork system call. This system call creates a copy of the process that invokes it. This child process is an image of the parent process, but with a new PID. The exec system call is then used – the parent process will overwrite the image of child with the copy of the program that is to be executed. After this, the parent executes the wait system call – parent will continue to wait till the death of the child. After this, parent can continue with its other activities.

Process is NOT set for all commands. The shell recognizes 2 types of commands – external and internal. External commands are commands like cat, ls etc or utilities. Shell scripts also come under the category of external commands. A process will be set up for the external commands. The internal commands are the shell's own built in statements, and commands like cd and echo etc. No process is set up for such commands.

It is necessary that some commands are built into the shell itself and no process is set up. That is because it is very difficult or sometimes impossible to implement some commands as external commands.

6.1: UNIX Processes > 6.1.2: Process Status Command - ps

PS Command

iGATE iTOPS for Business Outcomes

- **ps command displays characteristics of a process.**
- **Syntax:**

  ps [option [arguments] …]

- **Options:**
  - -f - full form
  - -u - details of only users processes
  - -a - all processes details
  - -l - detailed listing
  - -e - system processes

ps

```
$ ps
  PID    TTY        TIME CMD
  599    ttyp0   00:00:00 sh
  613    ttyp0   00:00:00 ps
$ _
```

iLEARN

- 6 -

PS Command:

The ps command can be used to display the characteristics of the processes. It has the knowledge of kernel built into it, using which it can read the process tables to get necessary information.

This command is one of the few Unix commands, which generates header information. It is also a highly variant command – the exact output obtained depends on the version of Unix as well as the hardware used.

When used without any options, ps command displays the following:

1. Process Id
2. The terminal with which the process is associated
3. Cumulative processor time of the process
4. The process name

6.1: UNIX Processes > 6.1.2: Process Status Command - ps

Example

iGATE iTOPS for Business Outcomes

➢ **Output of ps –l command:**

```
$ ps -l
  F S    UID   PID  PPID  C PRI NI     ADDR   SZ     WCHAN     TTY        TIME C
MD
 20 R    201   599   598  3  47 24 fb11c8b0   60         -    ttyp0    00:00:00 s
h
 20 O    201   625   599  1  48 24 fb11ca08  164         -    ttyp0    00:00:00 p
s
$ _
```

iLEARN

- 7 -

Process Status Command: Example:

• **l** – It displays Long format

•**F** - Octal flags which are added together to give more information about the current status of a process(20 – process is loaded in primary memory: it has not been swapped out to disk)

•**S** - State of the process (O – Process is running on a processor, R – Process is on run queue)

•**UID** - The Userid of the process owner (login name is printed using –f option)

•**PID** - The process ID of the process (this number is needed to kill a process)

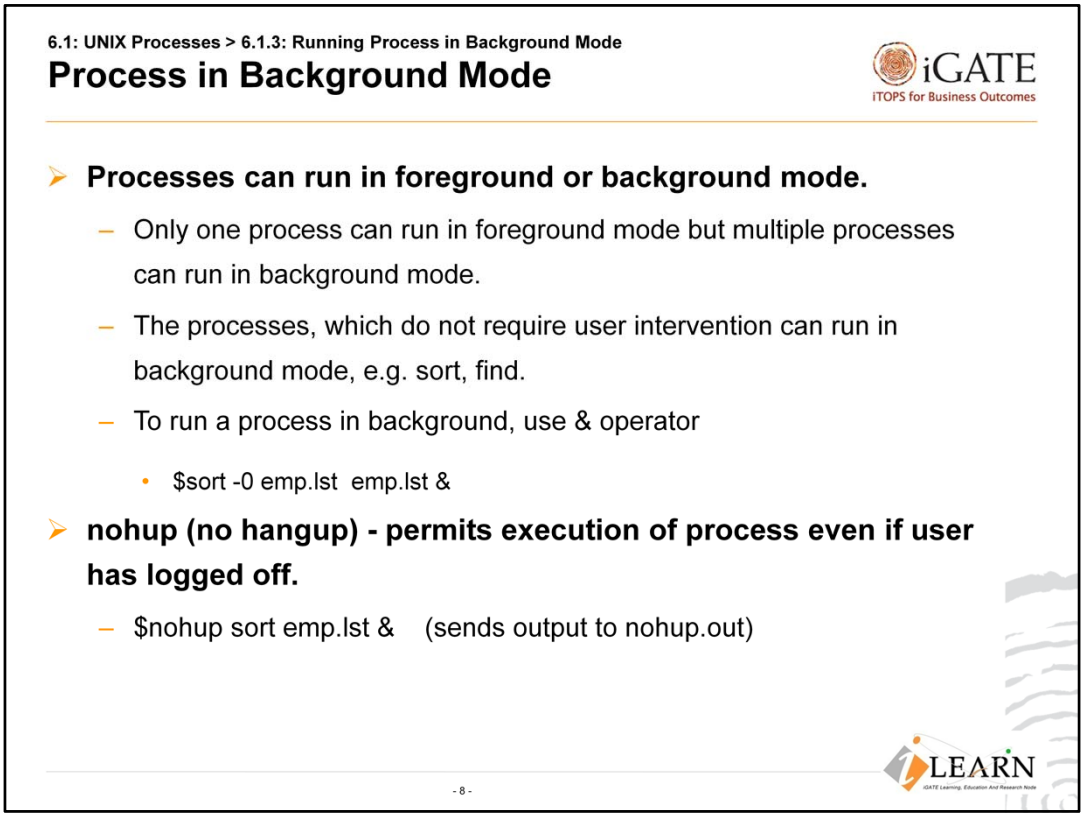
•**PPID** - The process ID of the parent process

•**C** - CPU usage by the process; combination of this value with nice value is used to calculate the priority

•**PRI** - The priority of the process (lower number mean lower priority)

•**NI** - The nice value of the process

•**ADDR** - The virtual address of the process entry in the process table



Process in Background Mode:

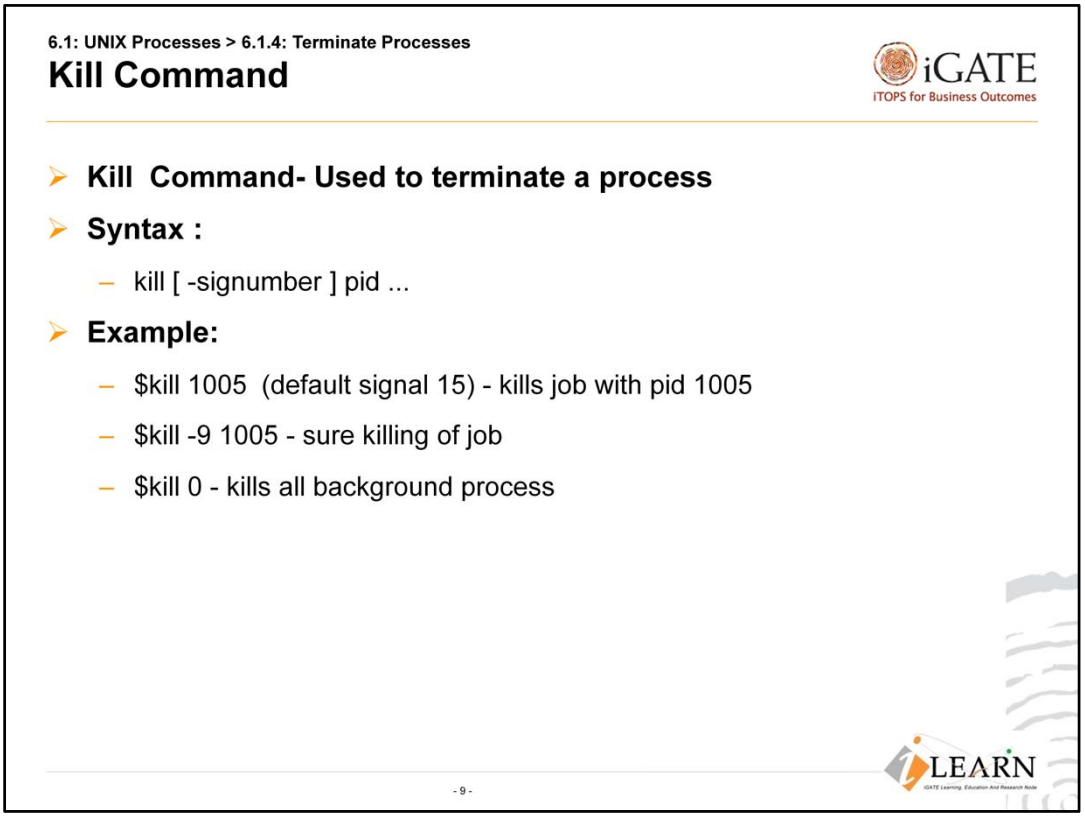
It is possible to have only one job working in the foreground. But the other jobs can be made to run at the background. Background execution is a useful feature if more important jobs are to be run in the foreground and less important ones relegated to the background.

The & operator is provided by the shell to run a process in the background. On invoking a command terminated with &, shell immediately returns a PID for this number; and the shell is ready to accept another command even though the previous command is not terminated yet. It will be used as follows:

```
$ sort –o emp.lst emp.lst &
```

Even if a command is run in the background, it is possible that the output as well as error messages are still coming to the standard output. Hence, care should be taken to suitably redirect this output.

However, too many jobs should not be run in the background as significant deterioration of CPU performance can occur.



The Kill Command:

It is possible to send signals to processes. When a process receives a signal, it can ignore it, terminate or do something else. Signals in Unix are identified by a number – each signal notifying that an event has occurred.

Other syntax

```
kill -s signame pid ...
kill -l [ exit_status ]
kill [ -signame ] pid ...
```

A signal number 15 is used by default by the kill command to terminate a process. The kill command takes one or more PID numbers as its arguments and facilities premature termination of these processes. It is used as follows (assumed that process numbered 117 is being killed):

```
$ kill 117
```

It is possible that some programs simply ignore this command and continue normal execution. In that case, a "sure kill", with signal number 9, has to be employed.

```
$ kill –9 117
```

In order to kill all processes except the login shell, an argument of 0 is passed to the kill command.

Of course, one can kill only one's own processes. Besides, some system processes cannot be killed at all.

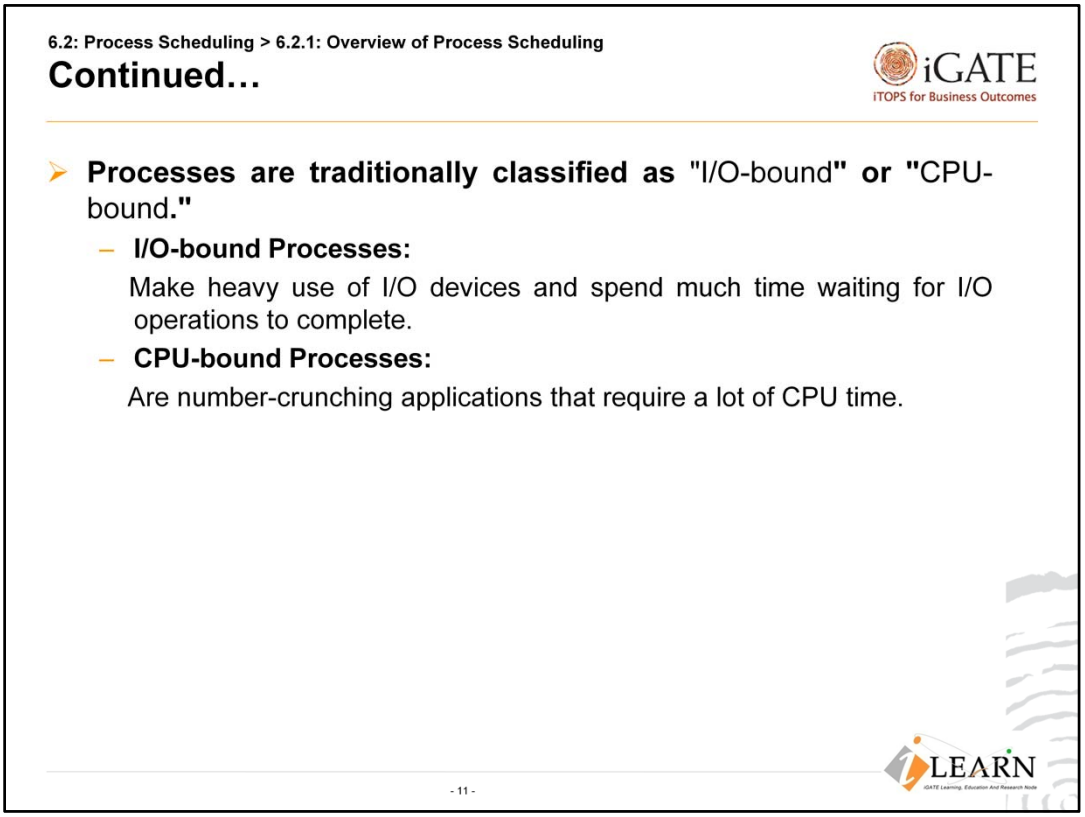
6.2: Process Scheduling > 6.2.1: Overview of Process Scheduling

Details

iGATE iTOPS for Business Outcomes

- **Scheduling Policy:**
  - *time-sharing* technique
  - Several processes are allowed to run "concurrently," which means that the CPU time is roughly divided into "slices," one for each runnable process.
  - The scheduling policy is also based on process priority
  - In UNIX, process priority is dynamic.

iLEARN iGATE Learning, Education And Research Node

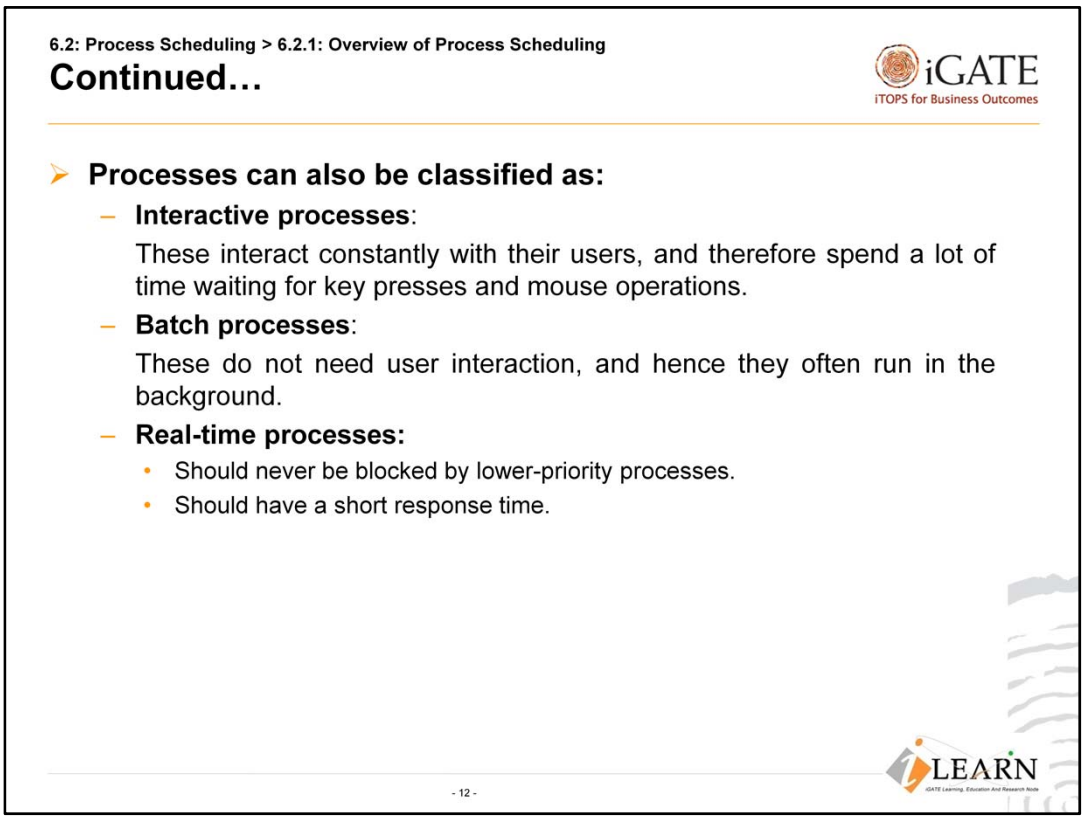


Overview of Process Scheduling:

When speaking about scheduling, processes are traditionally classified as "I/O-bound" or "CPU-bound."

•**I/O-bound Processes:** They make heavy use of I/O devices and spend much time waiting for I/O operations to complete.

•**CPU-bound Processes:** They are number-crunching applications that require a lot of CPU time.



Overview of Process Scheduling (contd..):

An alternative classification distinguishes three classes of processes:

Interactive processes

These interact constantly with their users, and therefore spend a lot of time waiting for key presses and mouse operations. When input is received, the process must be woken up quickly, or the user will find the system to be unresponsive. Typically, the average delay must fall between 50 and 150 ms. The variance of such delay must also be bounded, or the user will find the system to be erratic. Typical interactive programs are command shells, text editors, and graphical applications**.**

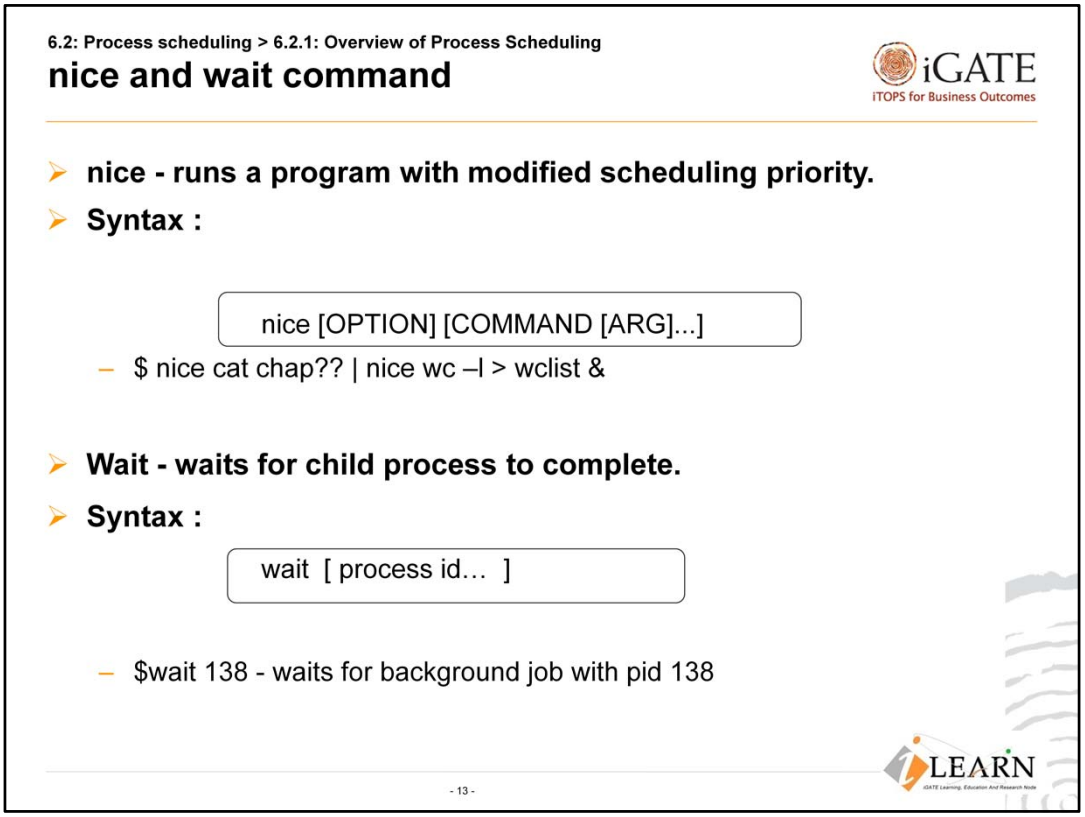
Batch processes

These do not need user interaction, and hence they often run in the background. Since such processes do not need to be very responsive, they are often penalized by the scheduler. Typical batch programs are programming language compilers, database search engines, and scientific computations.

Real-time processes:

These have very strong scheduling requirements. Such processes should never be blocked by lower-priority processes, they should have a short response time and, most important, such response time should have a minimum variance. Typical real-time programs are video and sound applications, robot controllers, and programs that collect data from physical sensors.

The two classifications we just offered are somewhat independent. For instance, a batch process can be either I/O-bound (e.g., a database server) or CPU-bound (e.g., an image-rendering program). While in UNIX real-time programs are explicitly recognized as such by the scheduling algorithm, there is no way to distinguish between interactive and batch programs. In order to offer a good response time to interactive applications, UNIX (like all Unix kernels) implicitly favors I/O-bound processes over CPU-bound ones.

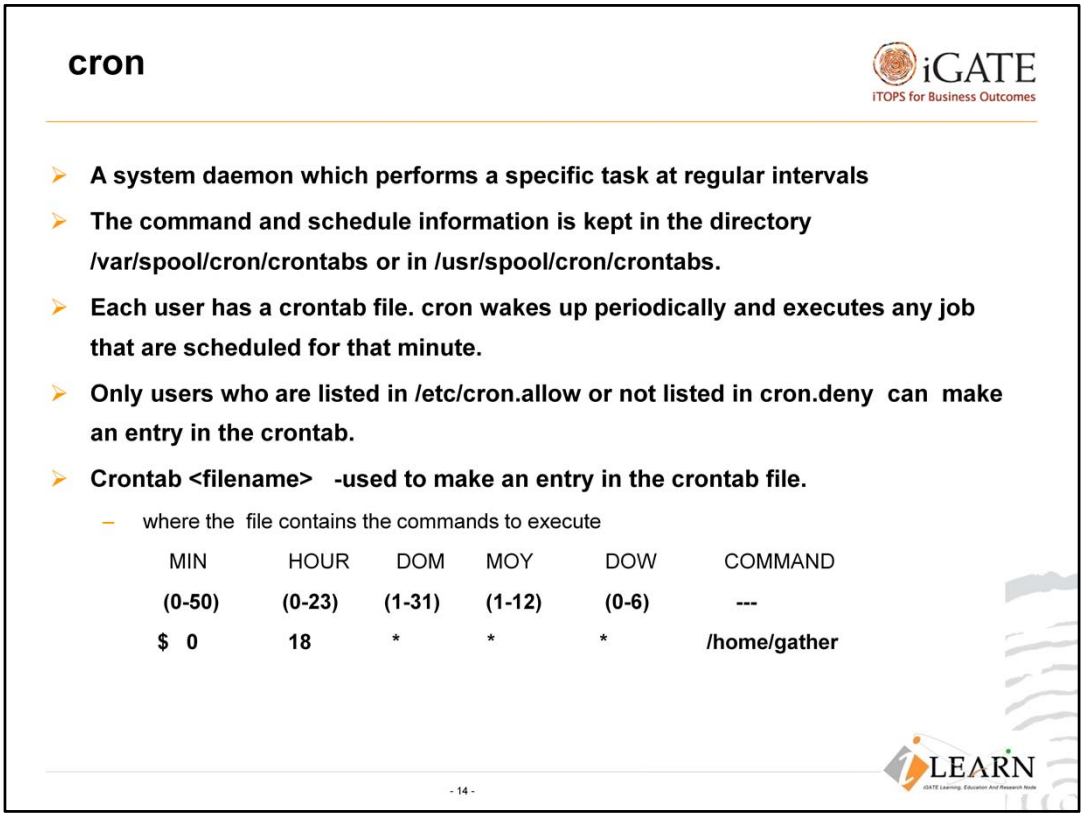


nice and wait Command:

All processes on Unix are usually executed with equal priority. In order to reduce the priority of a job, the command needs to be prefixed with nice. By default, nice reduces the priority of any process by 10 units. The amount of reduction can also be specified as an argument (value from 0 to 19) to the nice command. Incase of commands in a pipeline, to reduce the priority of all commands in the pipeline, nice needs to be used in each element in the pipeline.

```
$ nice cat chap?? | nice wc –l > wclist &
```

The wait command can be used to wait for the completion of a background process. This is a built-in shell command – no process is spawned for this command. It sends the shell into a wait state so that it can acknowledge the death of child processes.



You can view the contents of the global cron file by logging in as **root** and typing:
 # crontab -l
The cron process executes the jobs it contains at the times that are specified. Each "cron job" is composed of six fields. Fields one to five contain clock information which specify when the command (given in the sixth field) should be executed.
Minute Hour Day Month Day Task
Minute = Minute of the hour, 00 to 59. * Will indicate every minute (details later)
Hour = Hour of the day in 24-hour format, 00 to 23. * Will indicate every hour (details later)
Day = Day of the month, 1 to 31. * Will indicate every day (details later)
Month = Month of the year, 1 to 12. * Will indicate every month (details later)
Day = Day of the week, 3 chars - sun, mon, tue, or numeric (0=sun, 1=mon etc).... * Will indicate every day (details later)
Task = The command you want to execute

Note: each of the above must be separated by at least 1 space.

Summary

iGATE
iTOPS for Business Outcomes

- **Unix processes**
- **Process related commands**
  - ps
  - nohup
  - wait
  - kill
  - nice
- **Background processes**

iLEARN

- 15 -

Add the notes here.

Review Questions

iGATE
iTOPS for Business Outcomes

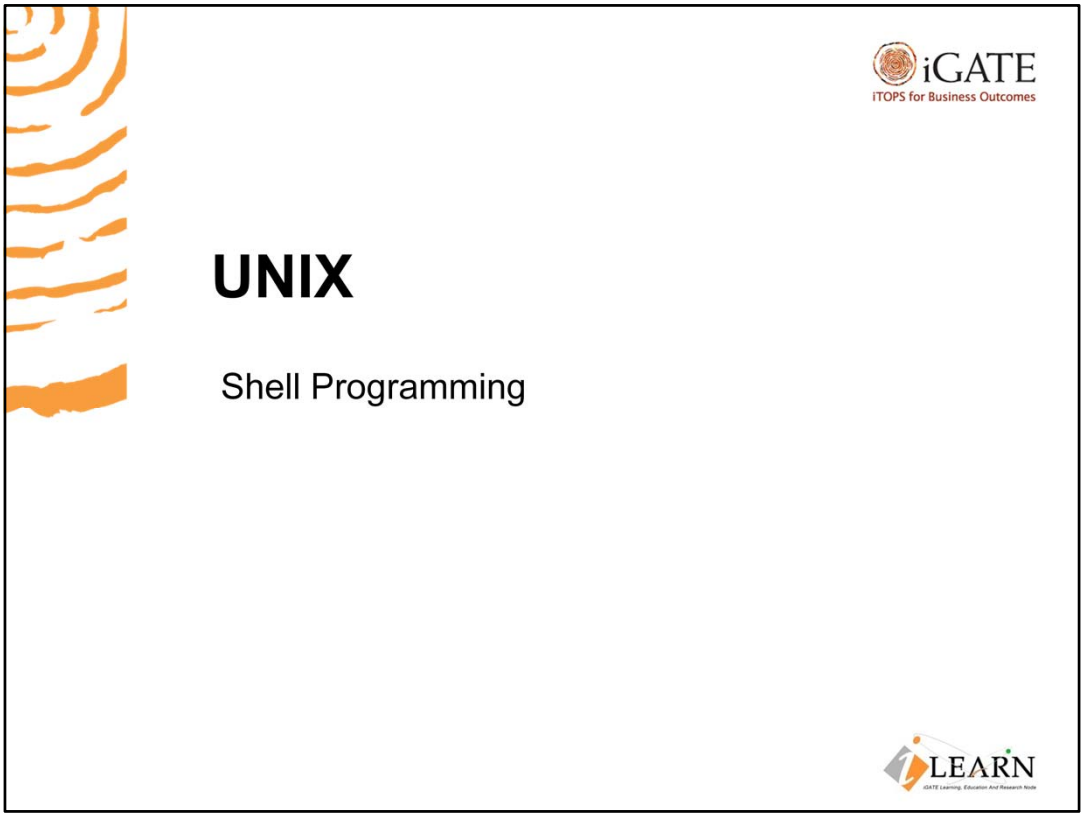
- **Complete The Following :**
  - A unique number called the _____ identifies each process.
  - Processes using heavy i/o are called as _____

- **True / False**
  - A signal number of 9 is used, by default, by the kill command to terminate a process.
  - You can kill any process, including the system process, using the kill command.

iLEARN

- 16 -

Add the notes here.


iGATE
iTOPS for Business Outcomes
UNIX
Shell Programming
iLEARN

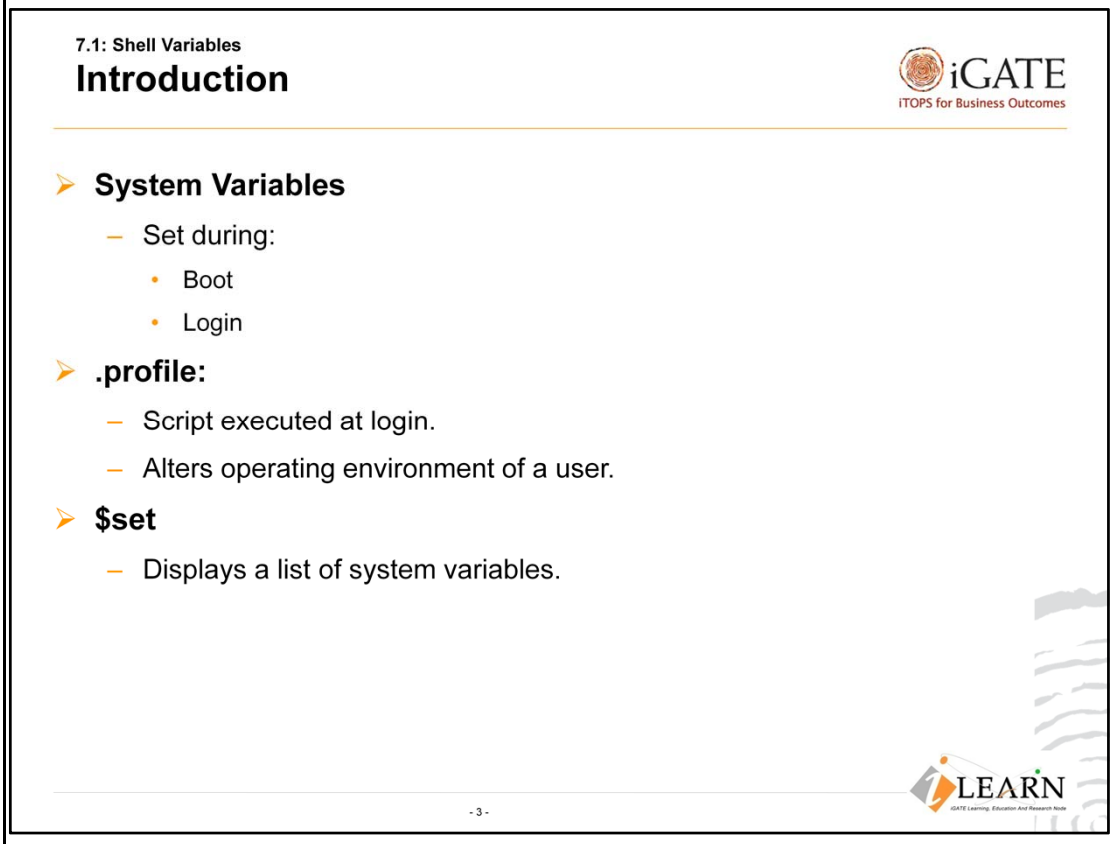
Lesson Objectives

iGATE
iTOPS for Business Outcomes

- **At the end of the session you will be able to understand:**
  - Shell variable
  - Environment variables
  - Shell script commands
  - Command substitution
  - Command line argument
  - Conditional statements
  - Iterative statements

iLEARN
iGATE Learning, Education And Research Node

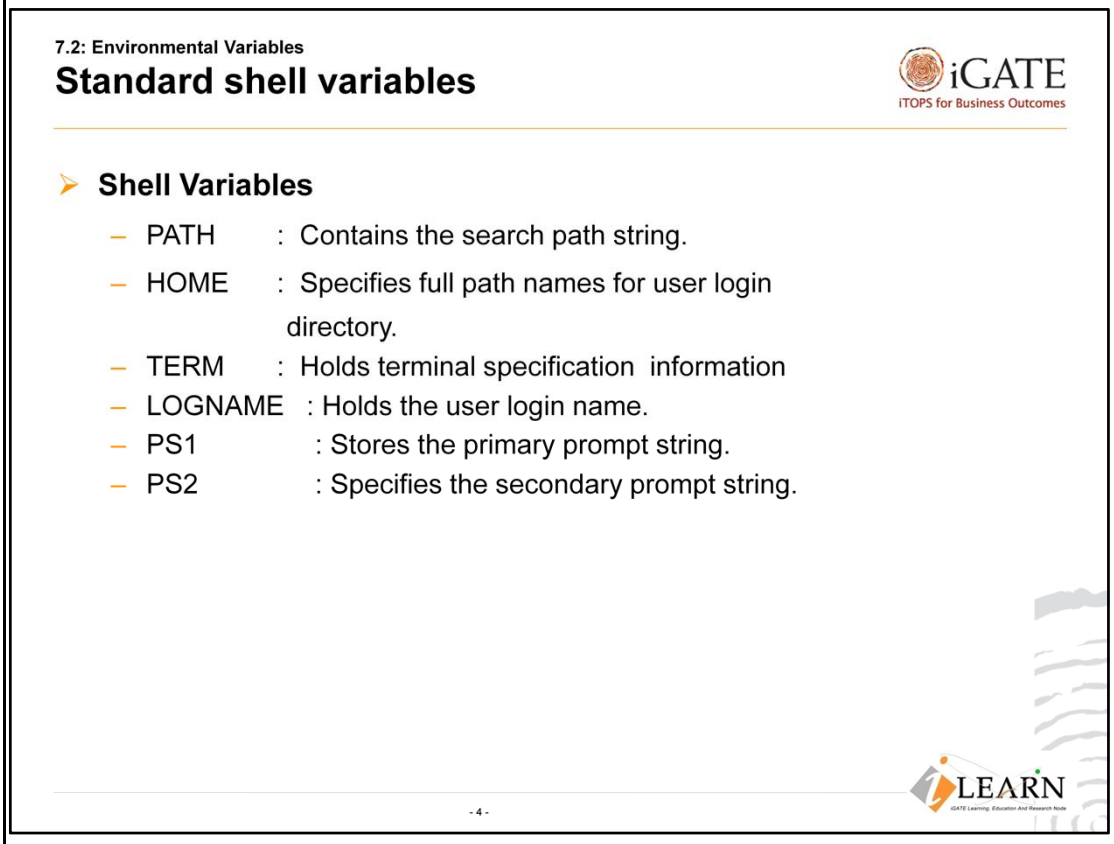
- 2 -



System Variables

There are several variables set by the system - some during booting and some after logging in. These are called the system variables, and they determine the environment one is working in. The user can also alter their values. The set statement can be used to display list of system variables.

```
$ set
HOME=/usr1/deshpavn
HUSHLOGIN=FALSE
HZ=100
IFS=
LOGNAME=deshpavn
MAIL=/usr/spool/mail/deshpavn
MAILCHECK=600
MF_ADM=adm.cat@Unix
MSG_MAIL=1
MS_PROFILE=1
OPTIND=1
PATH=/bin:/usr/bin:/usr1/deshpavn/bin:.
PS1=$
PS2=>
SHELL=/bin/sh
TERM=ansi
TZ=IST-5:30
```



Output of set command

Significance of some of these variables is explained below:

PATH Variable: Determines the list of directories (in order of precedence) that need to be scanned while you look for an executable command.

Path can be modified as:

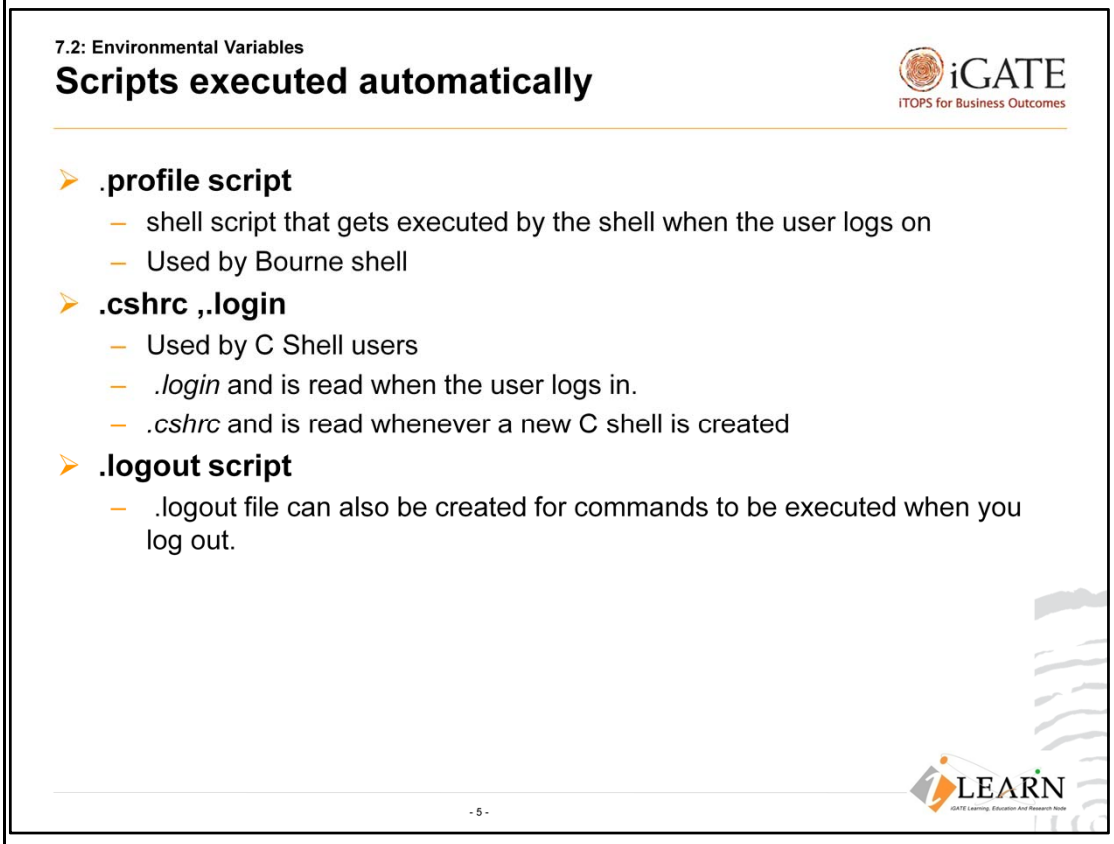
```
$ PATH=$PATH:/usr/user1/progs
```

This causes the /usr/user1/progs path to get added to the existing PATH list.

HOME Variable: This controls the login or Home directory for the user.

IFS Variable: It contains a string of characters that can be used as separators on command line.

PS1 and PS2 Variables: These determine the primary and secondary prompt.



.profile script

The *.profile* script is a shell script that gets executed by the shell when the user logs on. It contains settings for the operating environment of the user, and it remains in effect throughout the login session. Using this file, it is possible to customize operating environment.

.cshrc ,.login and .logout script

For the Bourne shell, the system reads the *.profile* file and executes the commands found there. C Shell users, however, have two files to read and execute. One is called *.login* and is read when the user logs in. The second is called *.cshrc* and is read whenever a new C shell is created, including the login shell. A .logout file can also be created for commands to be executed when you log out.

Page 07-5

7.3: Shell script Commands

Example

iGATE
iTOPS for Business Outcomes

- **Simple Shell Script: Accept Name & Display Message**

hello.sh

```
echo "Good Morning!"
echo "Enter your name?"
read name
echo "HELLO $name How are you?
```

- **To execute the shell script**

  $sh hello.sh

- **To debug the shell script use –x option**

  $sh –x hello.sh

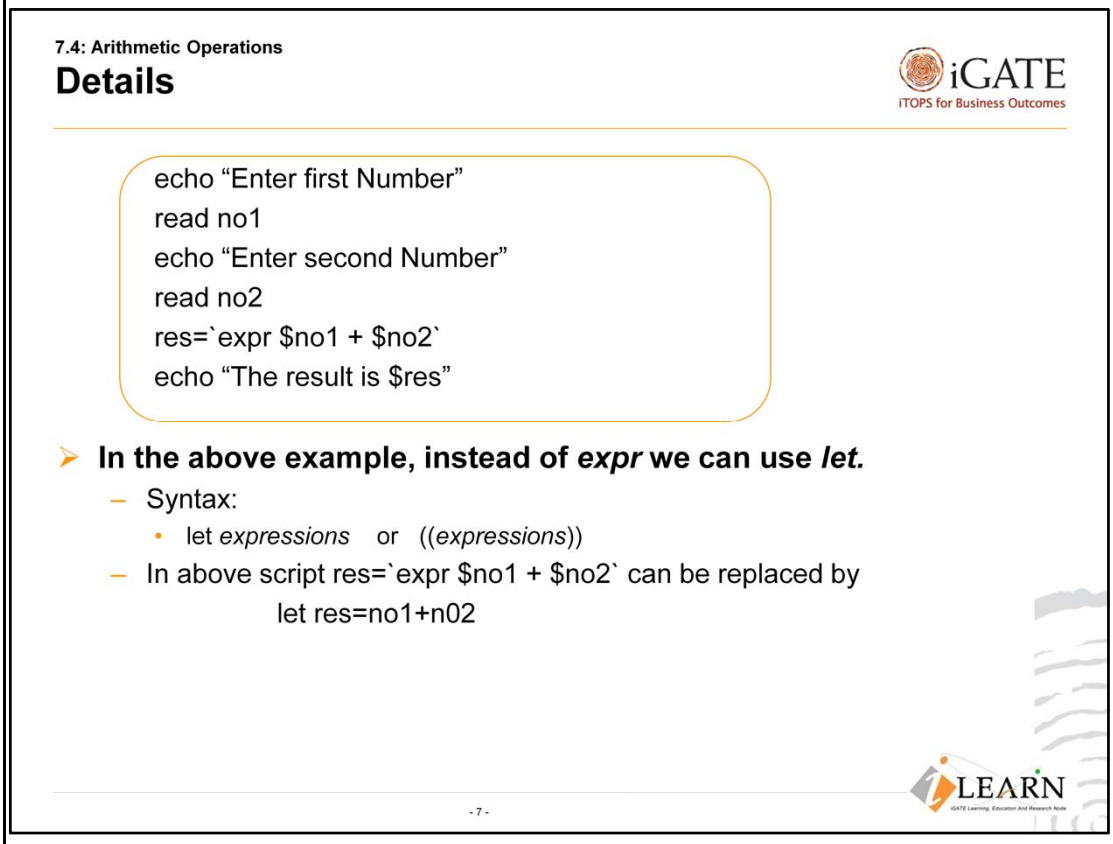
iLEARN

- 6 -

In above program, the *read* command accepts input from the user and stores it in name variable.

To display the variable value, you need to precede the variable name with a $ sign:

echo "HELLO **$name** How are you?

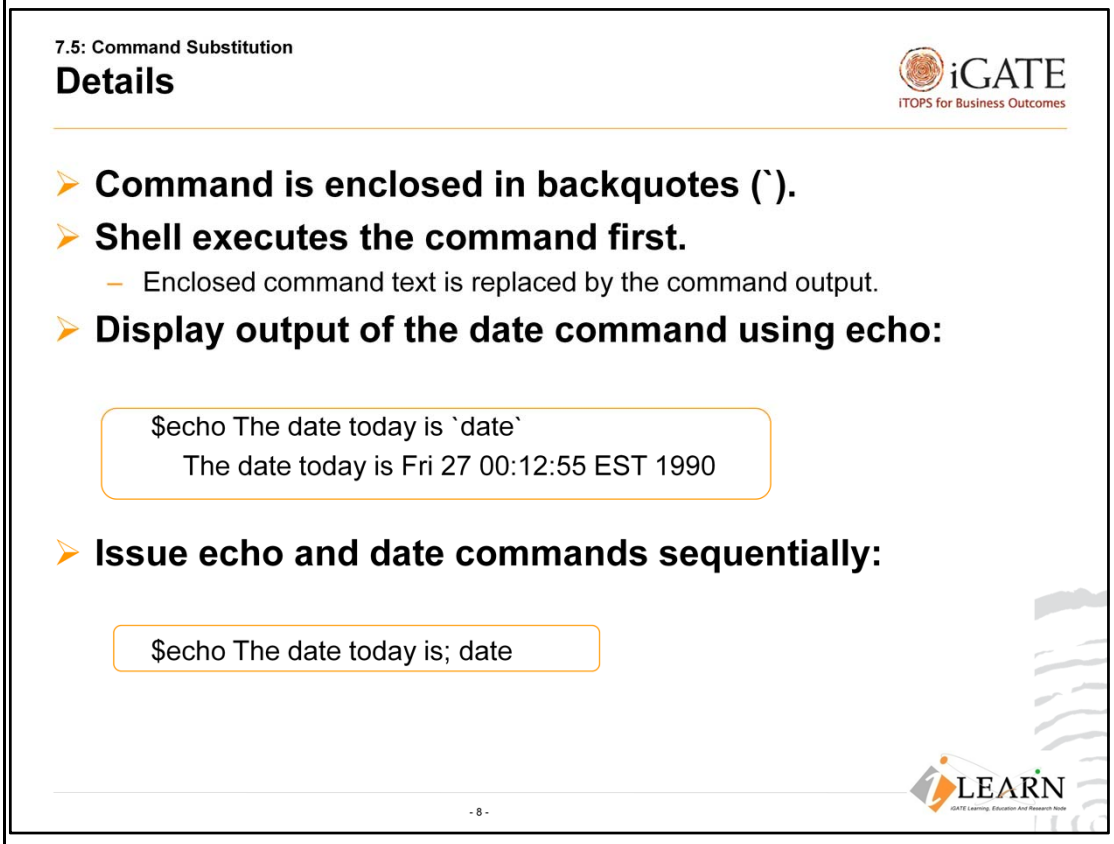


The above program accepts two numbers and displays their sum as a result. Instead of the *expr* command, we can use the *let* command.

Example:
Add *one* to variable i. Using expr statement:
- i=`expr $i + 1`

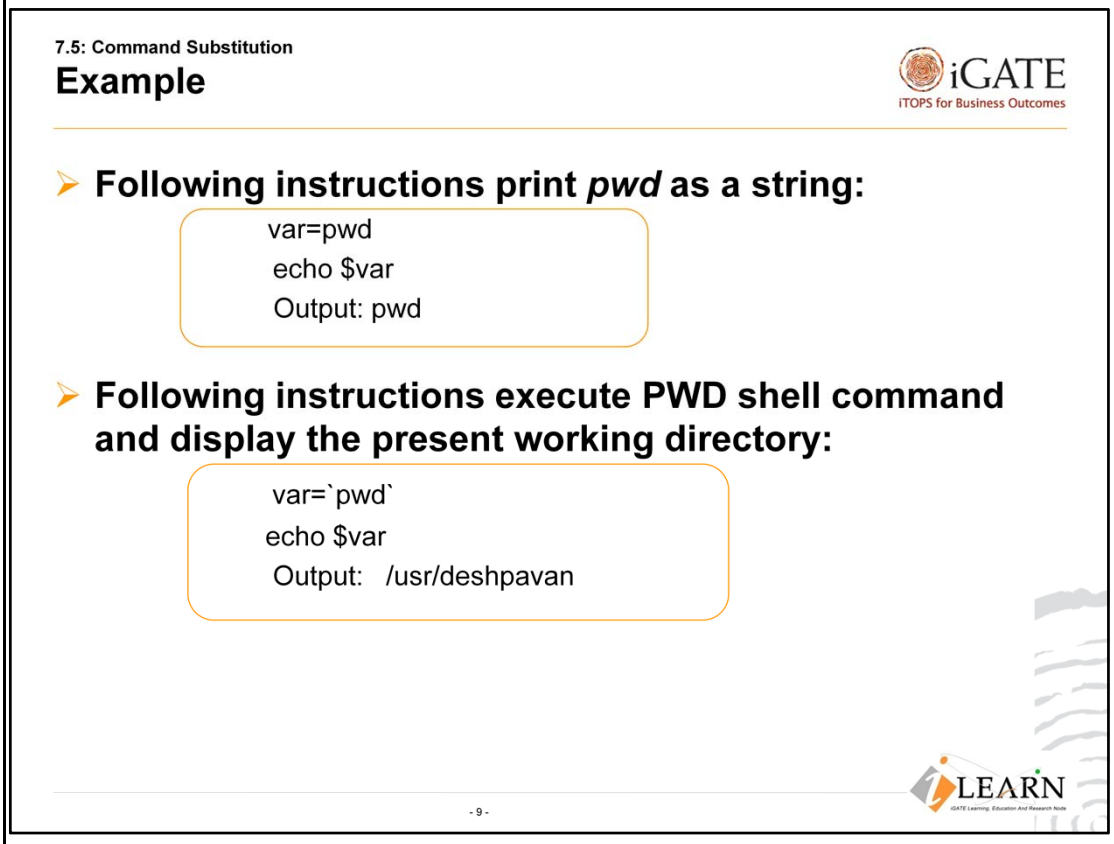
Add *one* to variable i. Using let statement:
- let i=i+1 If no spaces in expression
- let "i = i + 1" enclose expression in “… “ if expression includes spaces
- ((i = i + 1))

Expr is generally used but let is more user-friendly. It is used in Bash and Korn shell



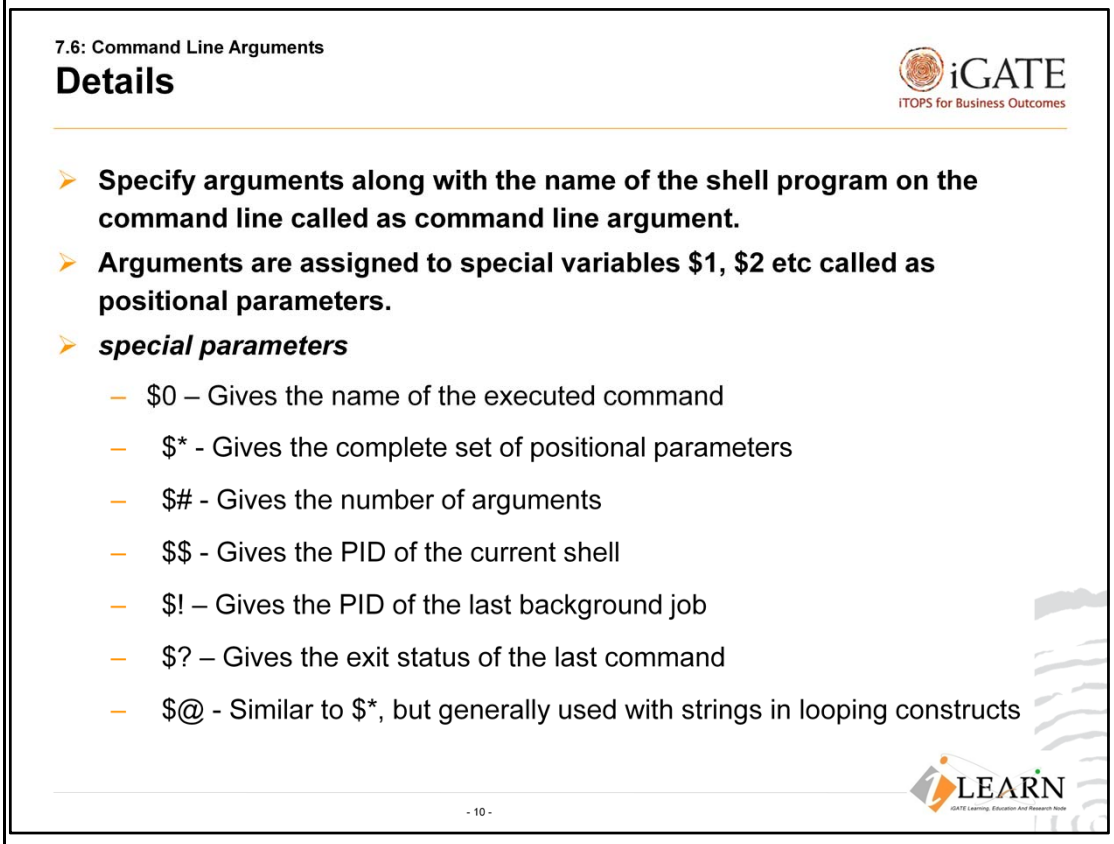
$echo The date today is `date`
In this command date is a command which is enclosed in backquotes and hence will get replaced by its output and then echo command will display message

$echo The date today is; date
In above command echo and date commands are separated by ; hence will get executed sequentially.



In the first example pwd is a string which is assigned to var variable. Hence o/p of echo $var will be pwd

But in second example 'pwd' string is assigned to var variable
Hence echo $var command will is
echo`pwd'
Since pwd is enclosed in backquotes it will get replaced by present working directory. echo will display name of current working directory.



You can pass values to shell programs while you execute shell scripts. These values entered through command line are called as *command line arguments*.

Parameters Related to Command Line Arguments

When you specify argument along with the name of the shell procedure, they are assigned into parameters $1, $2 etc. They are called as positional parameters. There are also some other *special parameters* you can use. Some of them are:

- $0 – Gives the name of the executed command
- $* - Gives the complete set of positional parameters
- $# - Gives the number of arguments
- $$ - Gives the PID of the current shell
- $! – Gives the PID of the last background job
- $? – Gives the exit status of the last command
- $@ - Similar to $*, but generally used with strings in looping constructs

Page 07-10

7.6: Command Line Arguments

Details

iGATE iTOPS for Business Outcomes

- **Arguments are assigned to special variables (positional parameters).**
  - $1 - First parameter , $2 - Second parameter,….
  - Example:

```
echo Program: $0
echo Number of arguments are $#
echo arguments are $*
grep "$1" $2
echo "\n End of Script“
```

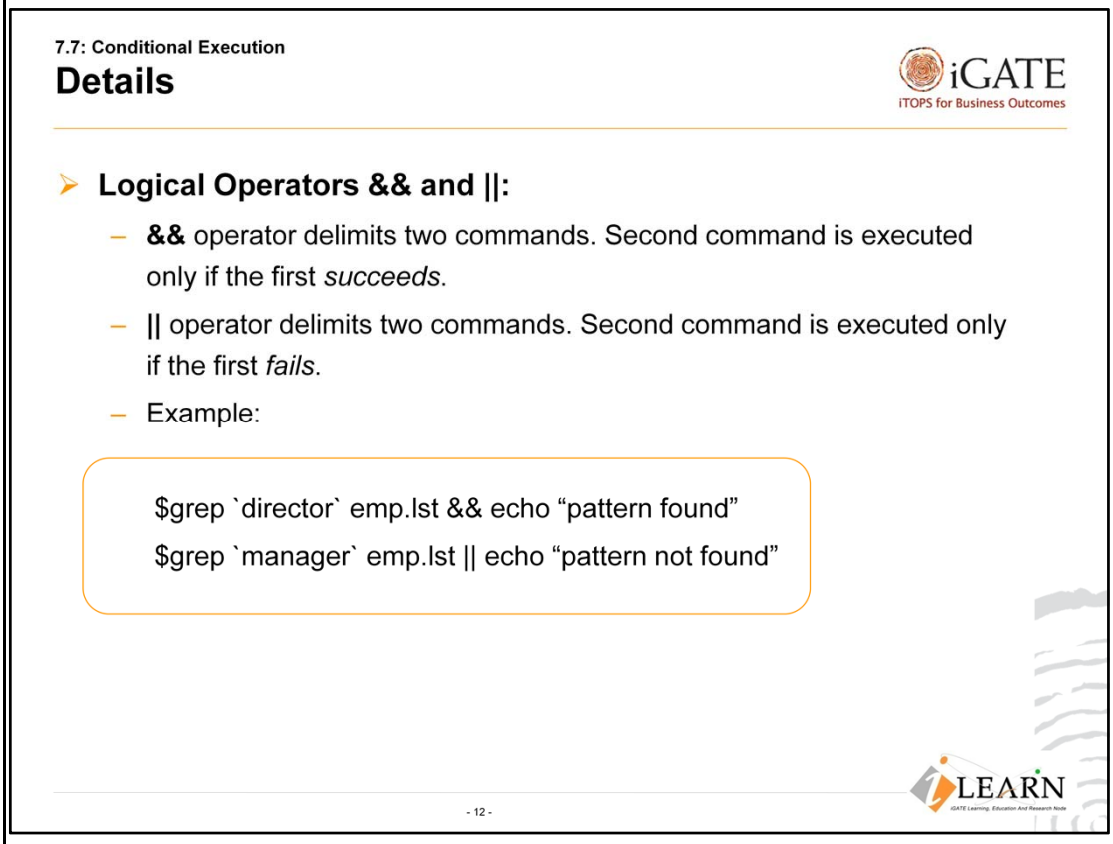
  - Run script:

```
$ scr1.sh "Unix" books.lst          --$1 is UNIX  , $2 –books.lst
```

iLEARN

- 11 -

In above example
$ scr1.sh "Unix" books.lst - The output only has lines with UNIX as substring from book.lst file .
Program: scr1.sh
Number of arguments are 2.
Arguments are Unix books.lst.
1001|Learning Unix |Computers |01/01/1998| 575
1004|Unix Device Drivers |Computers |09/08/1995| 650
1007|Unix Shell Programming |Computers |03/02/1993| 536
End of Script.



Conditional Execution using && and ||

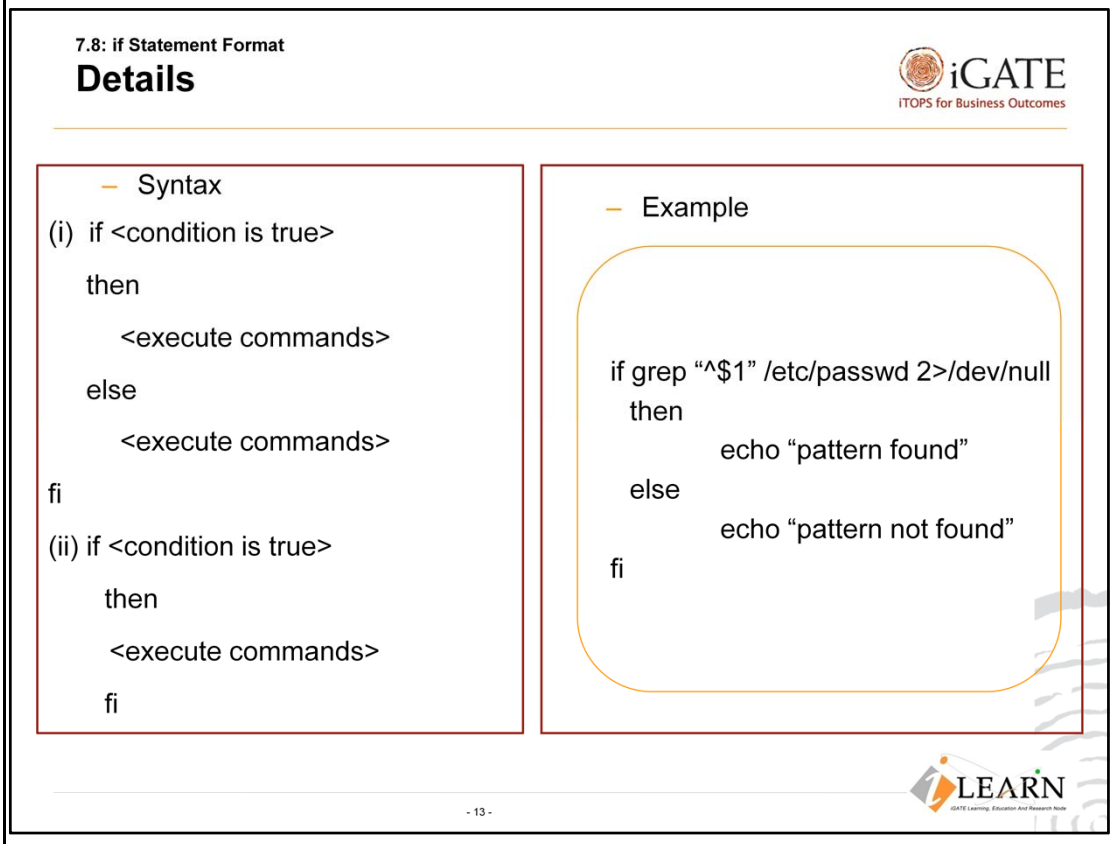
The shell provides && and || operators to control the execution of a command depending on the success or failure of previous command. In case of &&, the second command executes only if the first has succeeded. Similarly, || will ensure that the second command is executed only if the first has failed.

The following command displays “Found!” only if the XML pattern is found in the *books.lst* file at least once.

```
$ grep "XML" books.lst && echo "Found!"
1003|XML Unleashed          |Computers     |20/02/2000| 398
1006|XML Applications       |Fiction       |09/08/2000| 630
Found!
```

The following command displays “Not Found …”. If *grep* does not find the “WAP” pattern in the *books.lst* file.

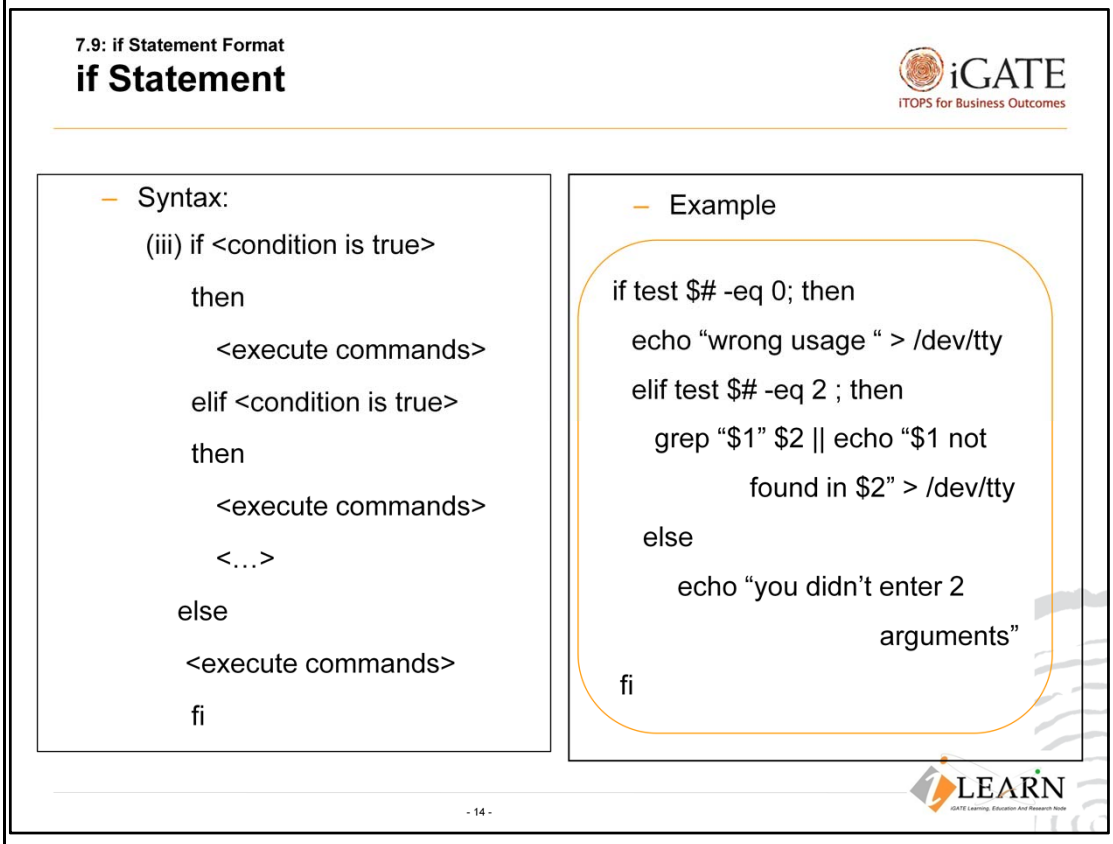
```
$ grep "WAP" books.lst || echo "Not Found..."
“Not Found...”
```



In UNIX **/dev/null** or **the null device** is a special file that discards all data written to it.

The null device is typically used to dispose the unwanted output stream of a process.

In given example, if *grep* returns any error and you wish to discard error messages, use /dev/null device.



In the example, test command is use to specify condition
The shell scripts checks for *two* command line arguments. If the number of arguments is *zero*, then the output is:
Wrong Usage

If it is *two*, then the first argument is used as a pattern and the second one is used as the file name to search in the *grep* command.

If the pattern is found, then the output of the *grep* command is displayed. Otherwise, the output of the echo command is displayed.

If the number of arguments are not *two,* then the output is as follows:
“you didn’t enter 2 arguments”.

7.9: test Statement

Relational Operator for numbers

iGATE
iTOPS for Business Outcomes

- **Specify condition either using *test* or *[condition]***
  - Example: test $1 –eq $2 same as [$1 –eq $2]
- **Relational Operator for Numbers:**
  - eq: Equal to
  - ne: Not equal to
  - gt: Greater than
  - gc: Greater than or equal to
  - lt: Less than
  - lc: Less than or equal to

iLEARN
iGATE Learning, Education And Research Node

- 15 -

7.9: test Statement

Relational Operator for strings and logical operators

iGATE iTOPS for Business Outcomes

- **String operators used by *test:***
  - -n str — True, if str not a null string
  - -z str — True, if str is a null string
  - S1 = S2 — True, if S1 = S2
  - S1 != S2 — True, if S1 ≠ S2
  - str — True, if str is assigned and not null
- **Logical Operators**
  - -a .AND.
  - -o .OR.
  - ! Not

iLEARN iGATE Learning, Education And Research Node

- 16 -

7.9: test Statement

File related operators

iGATE
iTOPS for Business Outcomes

- **File related operators used by test command**
  - -f <file> True, if file exists and it is regular file
  - -d<file> True, if file exist and it is directory file
  - -r <file> True, if file exist and it is readable file
  - -w <file> True, if file exist and it is writable file
  - -x <file> True, if file exist and it is executable file
  - -s <file> True, if file exist and it's size > 0
  - -e <file> True, if file exist

iLEARN
iGATE Learning, Education And Research Node

- 17 -

Page 07-17

7.9: test Statement

Example

iGATE
iTOPS for Business Outcomes

- **Check whether user has entered a filename or not:**
  - Example:

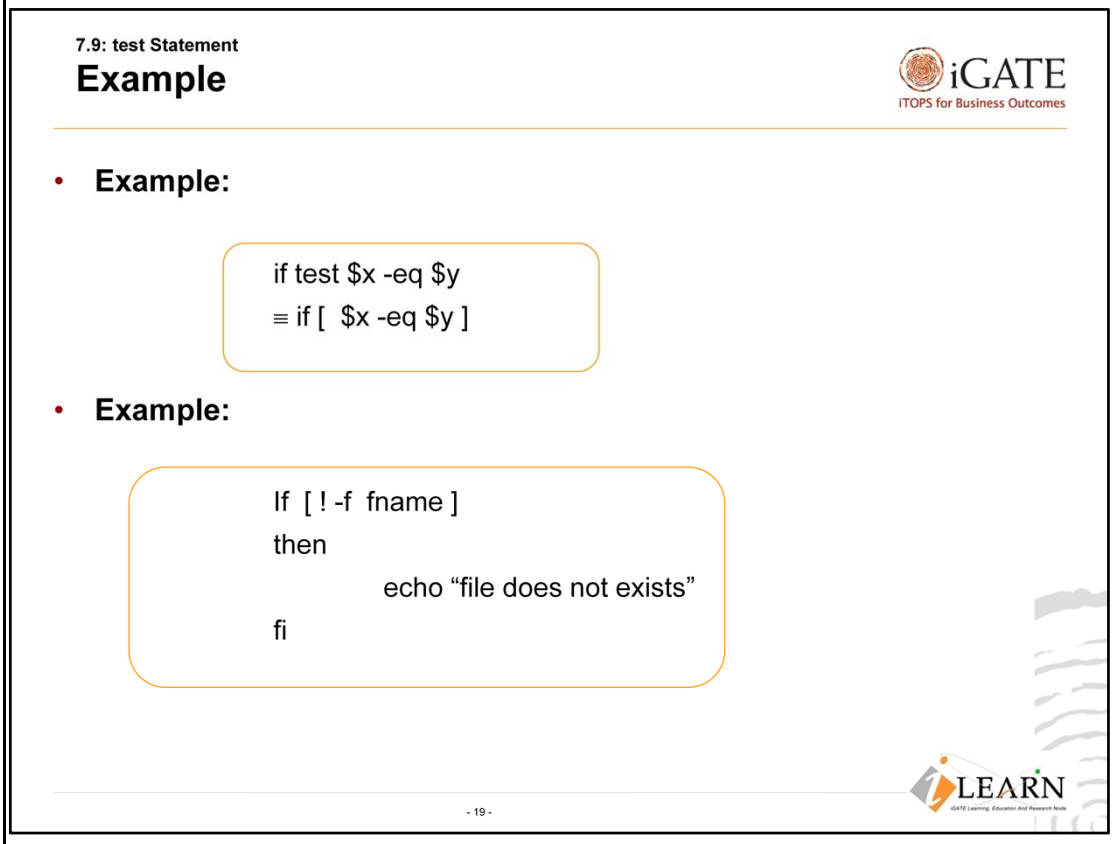
```
echo "Enter File Name:\c "
read fn
if [ -z “$fn” ]
then
   echo “You have not entered file name“
fi
```

iLEARN
iGATE Learning, Education And Research Node

- 18 -

In the given example **–z** checks whether **$fn** is empty or not. If users do not enter the file name, then the output is as follows:

“You have not entered file name“.



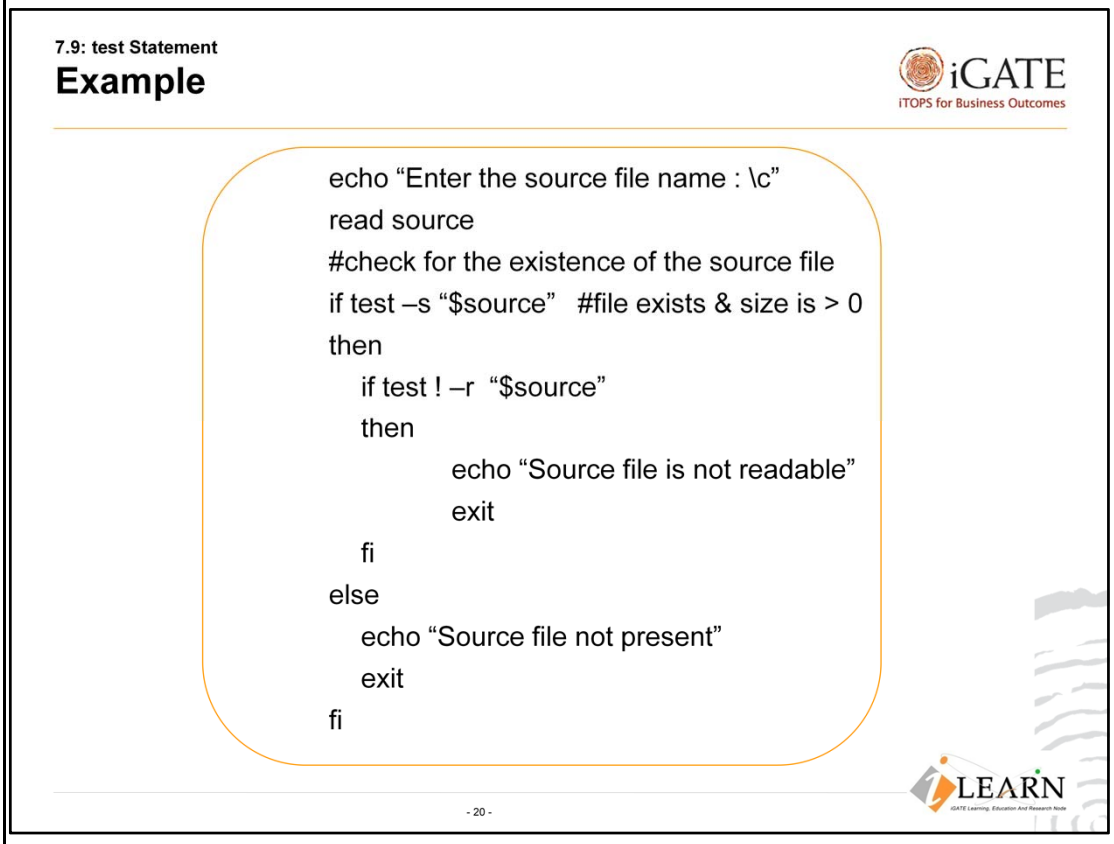
if test $x -eq $y
≡ if [$x -eq $y]

In above command both the conditions are the same. You can use the “[“ bracket to check the condition in place of the *test* command.

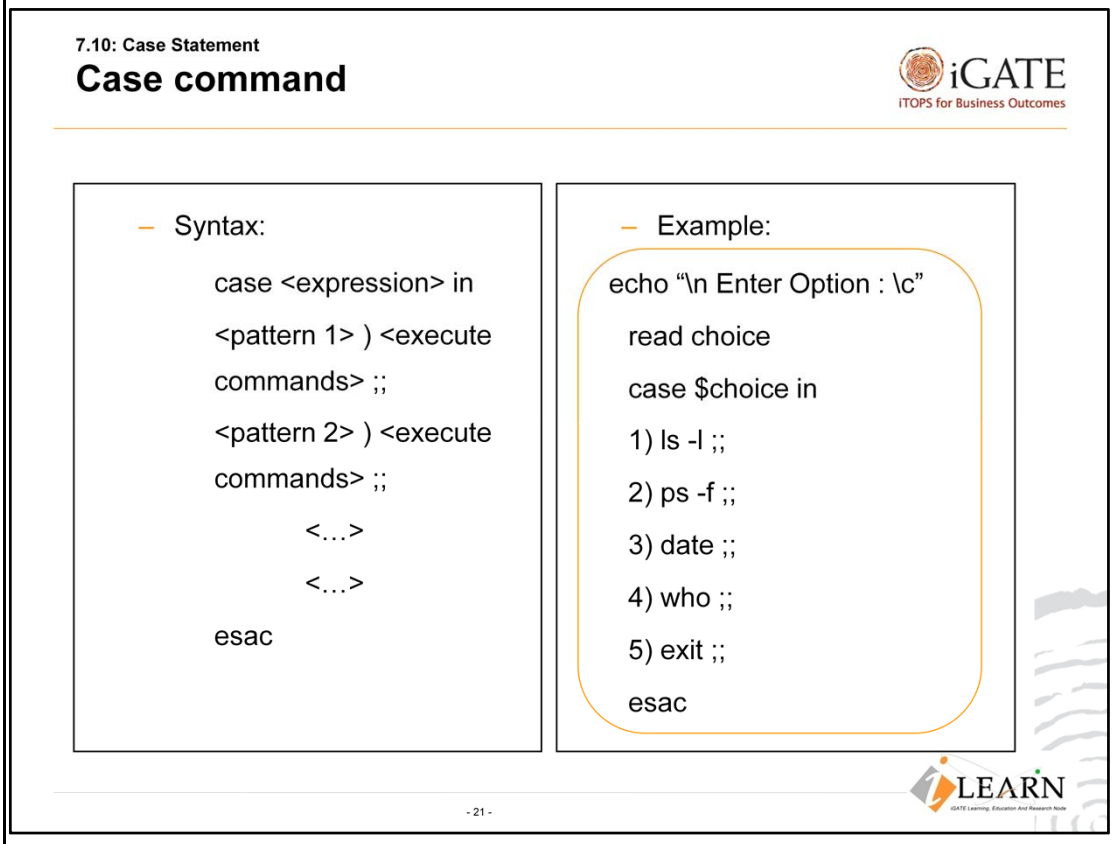
test $x –eq $y returns true if the values of variables x and y are equal. You can write the same condition as [$x –eq $y]. Here, instead of *test* command we use “[“ (square bracket).

If [! -f fname]

You can also write this condition as:
test !-f fname



The above example checks whether a given source file exists and displays appropriate messages.



In a *case* statement you can also use commands enclosed in *backquotes*. The given example executes command **`date | cut -d " " –f1`** which returns only the day part. The output is used to execute the appropriate case.

Example:

```
case `date | cut -d" " –f1` in
Mon ) <commands> ;;
Tue ) <commands> ;;
        :
esac
```

Example:

```
#display the options to the user
        echo "1. Date and time          2. Directory listing"
        echo "3. Users information      4. Current directory"
        echo "Enter choice (1,2,3,4) :\c"
read choice
case $choice in
1)      date;;
2)      ls –l;;
3)      who;;
4)      pwd;;
*) echo wrong choice;;
esac
#end of script
```

7.10: Case Statement

Example

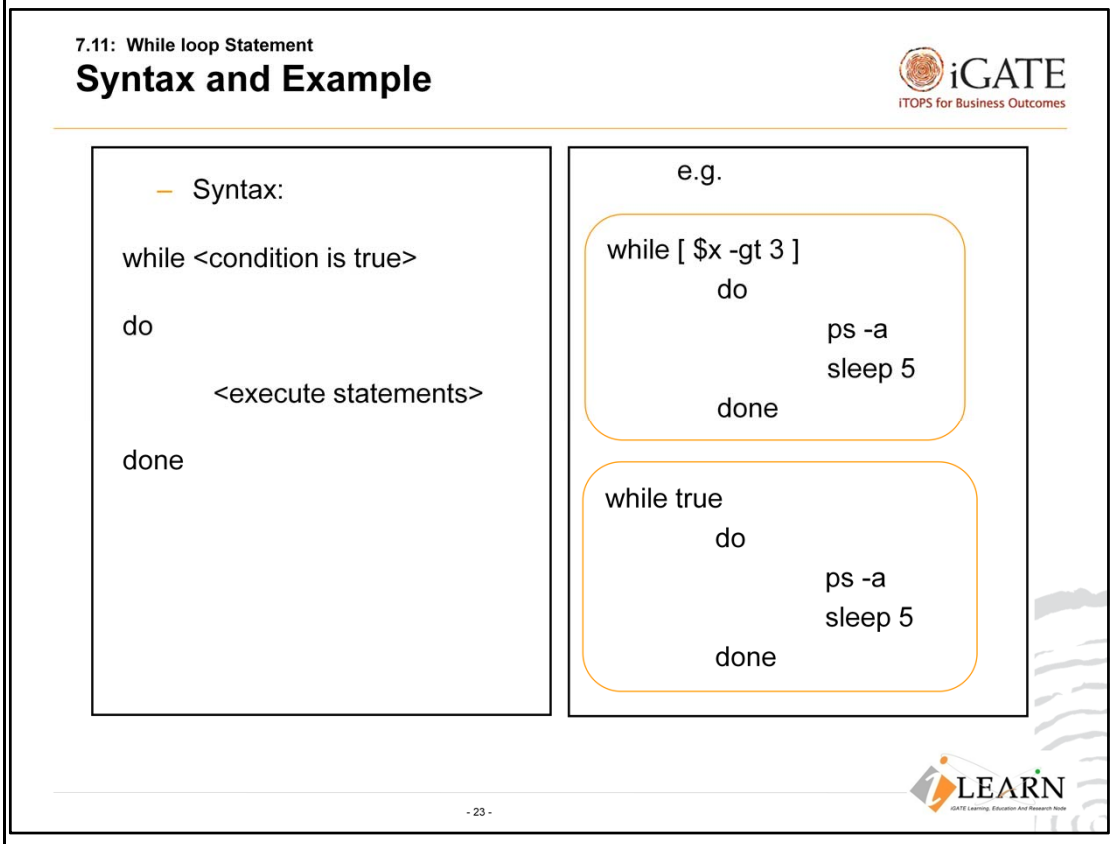
iGATE
iTOPS for Business Outcomes

```
echo “do you wish to continue?”
read ans
   Case “$ans” in
   [yY] [eE] [sS]) ;;
            [nN] [oO]) exit ;;
                     *) “invalid option” ;;
esac
```

iLEARN

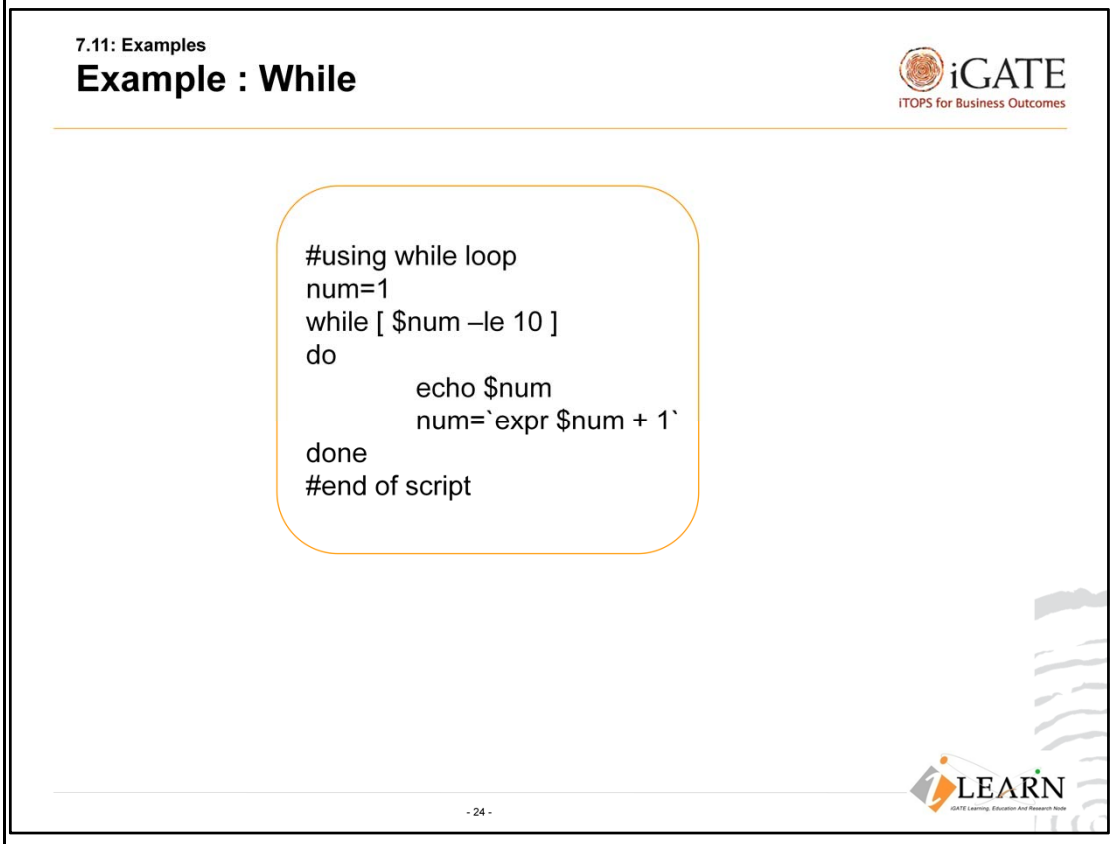
- 22 -

In the above example, the first case matches with “yes” or “YES”. Similarly, the second case matches with “no” or “NO”.



Example: Script to edit, compile and execute a program.

```
while true
Do
  cc $1
case $? In
o) echo “Compilation Successful”
    echo “Executing a.out”
    a.out ; exit ;;
*) echo “Compilation Error”
    echo “Press <Enter> to edit”
    read pause
    vi $1 ;;
Esac
done
```



In the above example, the loop executes till the condition is true. This is till the value of the variable num is < 10.

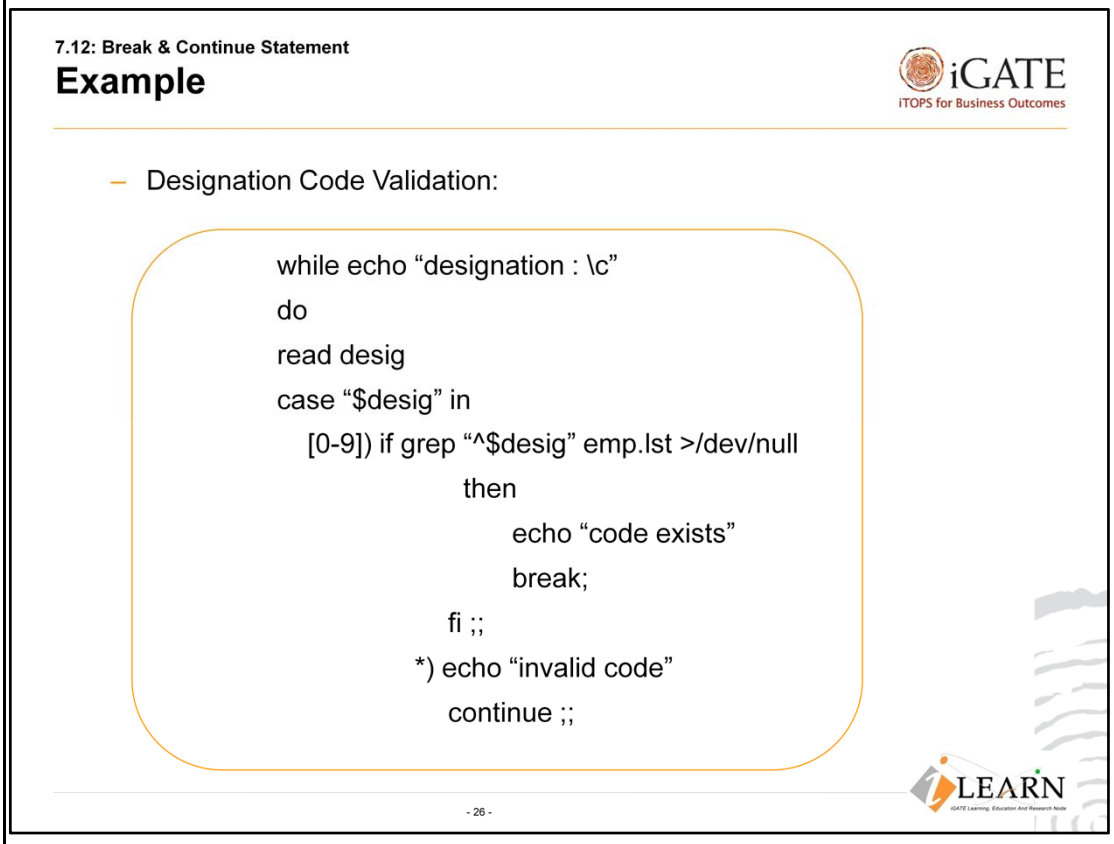
7.12: Break & Continue Statement

break and continue statement

iGATE iTOPS for Business Outcomes

- **Continue:**
  - Suspends statement execution following it.
  - Switches control to the top of loop for the next iteration.
- **Break:**
  - Causes control to break out of the loop.

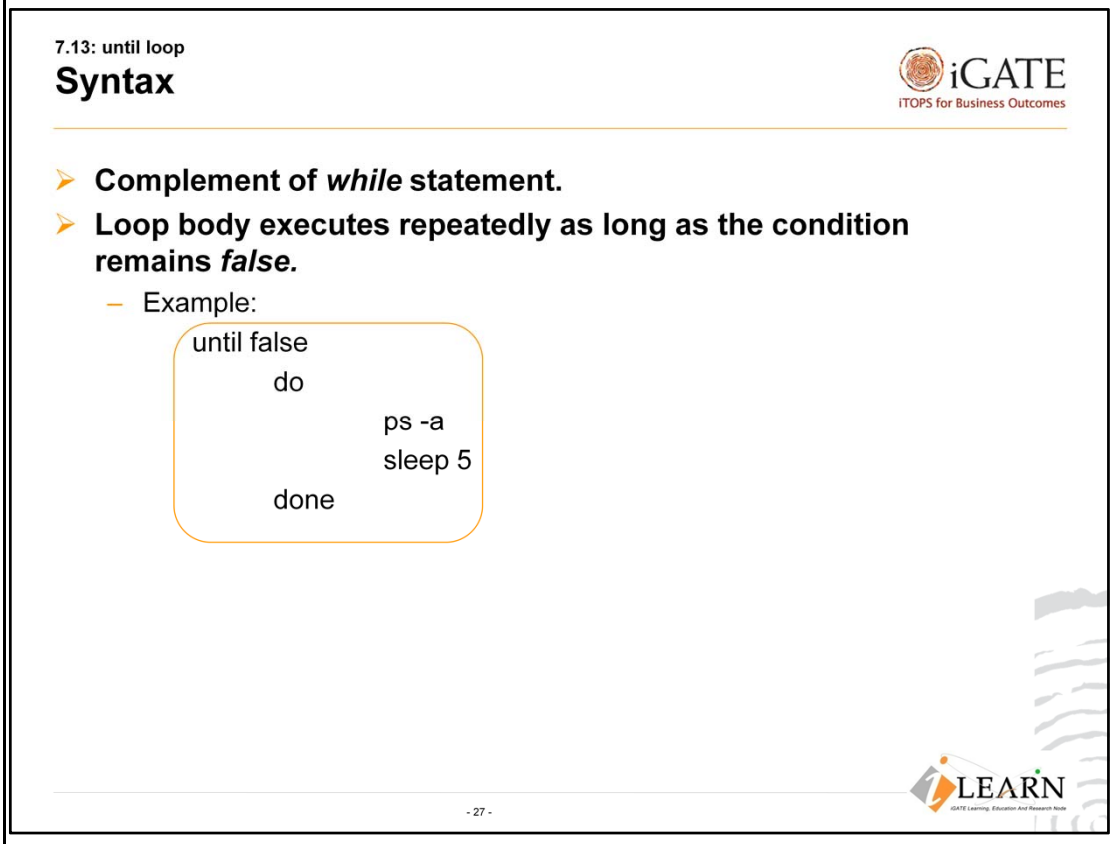
iLEARN iGATE Learning, Education And Research Node



In above example, the *while* loop is an unending loop as **echo “ designation : \c ”** statement (which is put as a condition in the while loop) always returns an exit status of success (condition becomes true).

Hence, it is more efficient if you write the following as a single statement:
while true
echo “designation : \c”

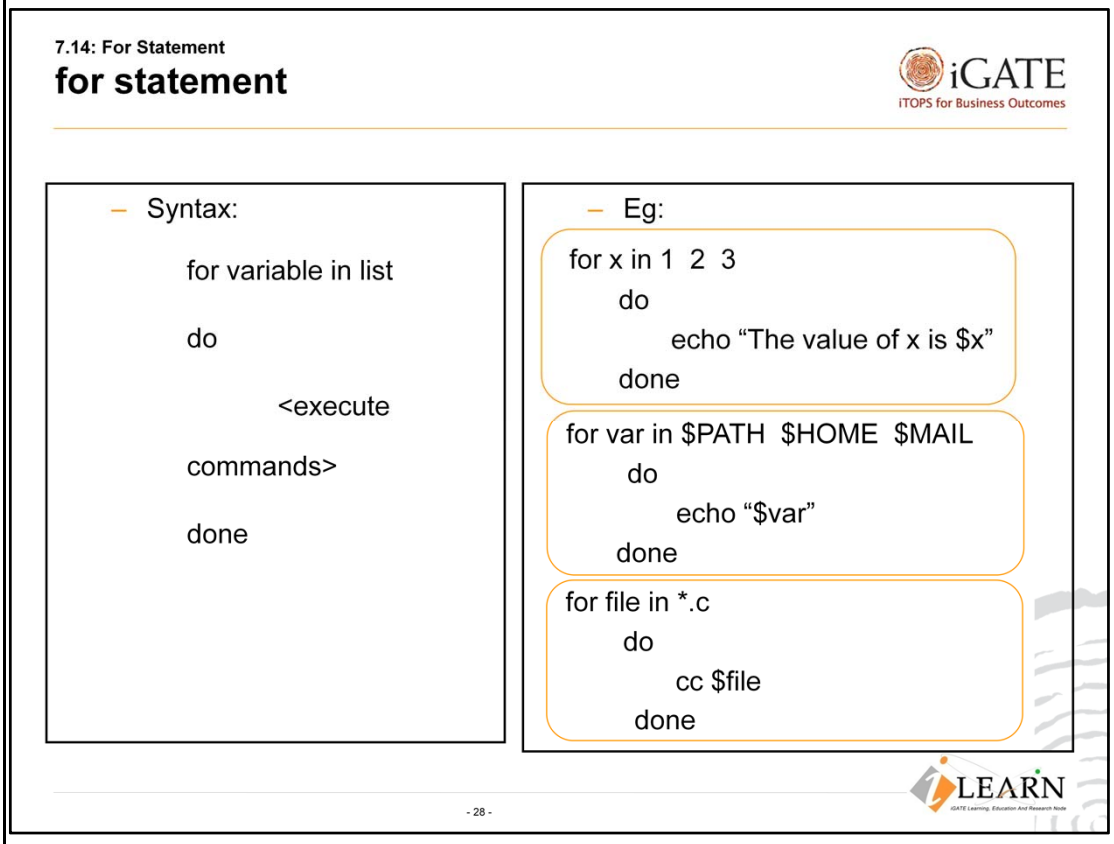
In the above program if you enter a designation as a two digit number, it matches with case [0-9][0-9]. If the designation found in the file break statement is executed, control comes out of the loop and the program halts. Otherwise, the default case is executed. Continue statement transfers the control at the beginning of the loop.



The syntax is as follows:
until condition
do
commands
Done

This loop is a complement of the *while* loop. In the *while* loop statements are repeated till the condition is *true*. But in an *until* loop, statements inside loop are repeated till the condition is *false.* As soon as the condition becomes true, the iteration stops.

In the above example given until loop is infinite loop.



Example 1:
In this example, *for* loop executes *three* times because three numbers are there in the list . In every iteration *x* is assigned 1, 2 and 3 respectively.

Example 2:
In this example also, *for* loop executes 3 times. In each iteration, *var* takes values from system variables in the list $PATH, $HOME and $MAIL respectively.

Example 3:
In this example, the for loop iterates equal to the number of files with extension c in the current working directory. This is because *.c is replaced with a list of all files with extension *c* in the current working directory.
Some more examples are:

```
for i in 1 2 3 4 5 6 7 8 9 0
do
                echo $i
done
```

Example : for

iGATE
iTOPS for Business Outcomes

```
for file in chap20 chap21 chap22 chap23; do
    cp $file ${file}.bak
    echo $file copied to $file.bak
done
```

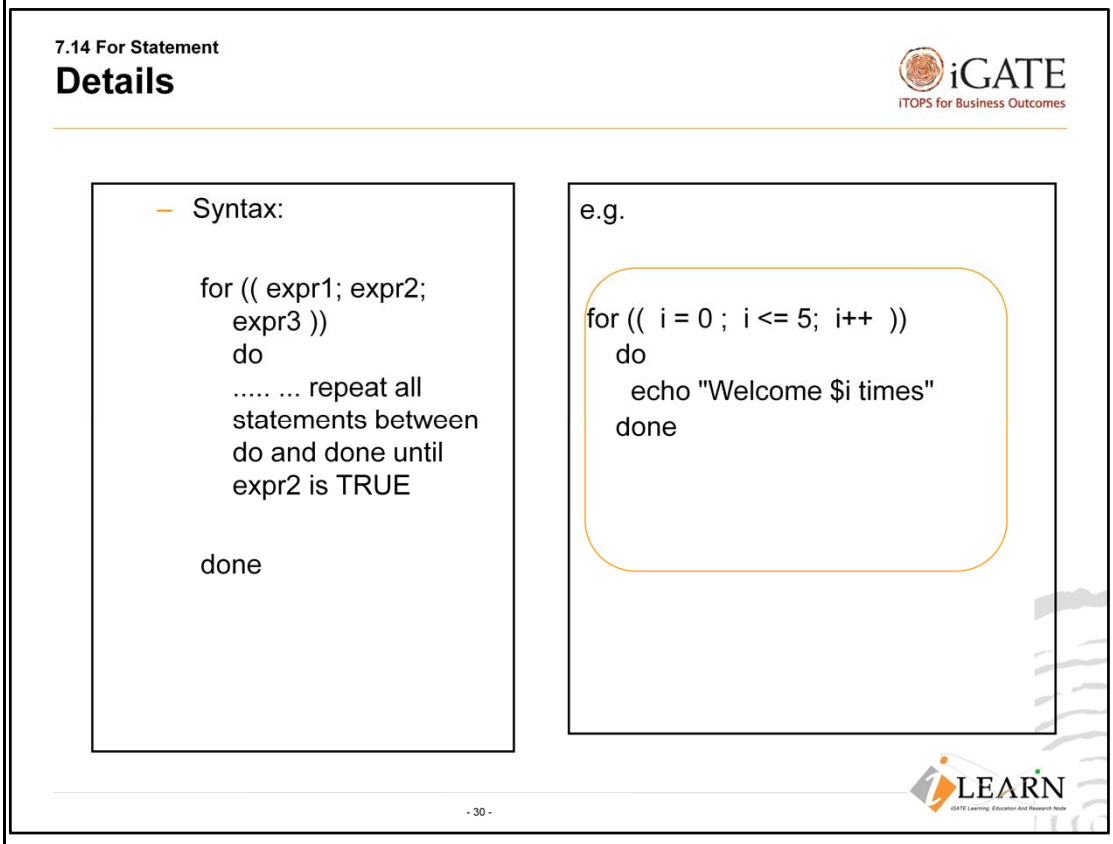
```
for file in 'cat clist'.....
```

```
for file in *.htm *.html;
do
    # do something
done
```

```
for pattern in "$@"; do
    grep "$pattern" emp.lst || echo "$pattern not found"
done
```

iLEARN
iGATE Learning, Education And Research Node

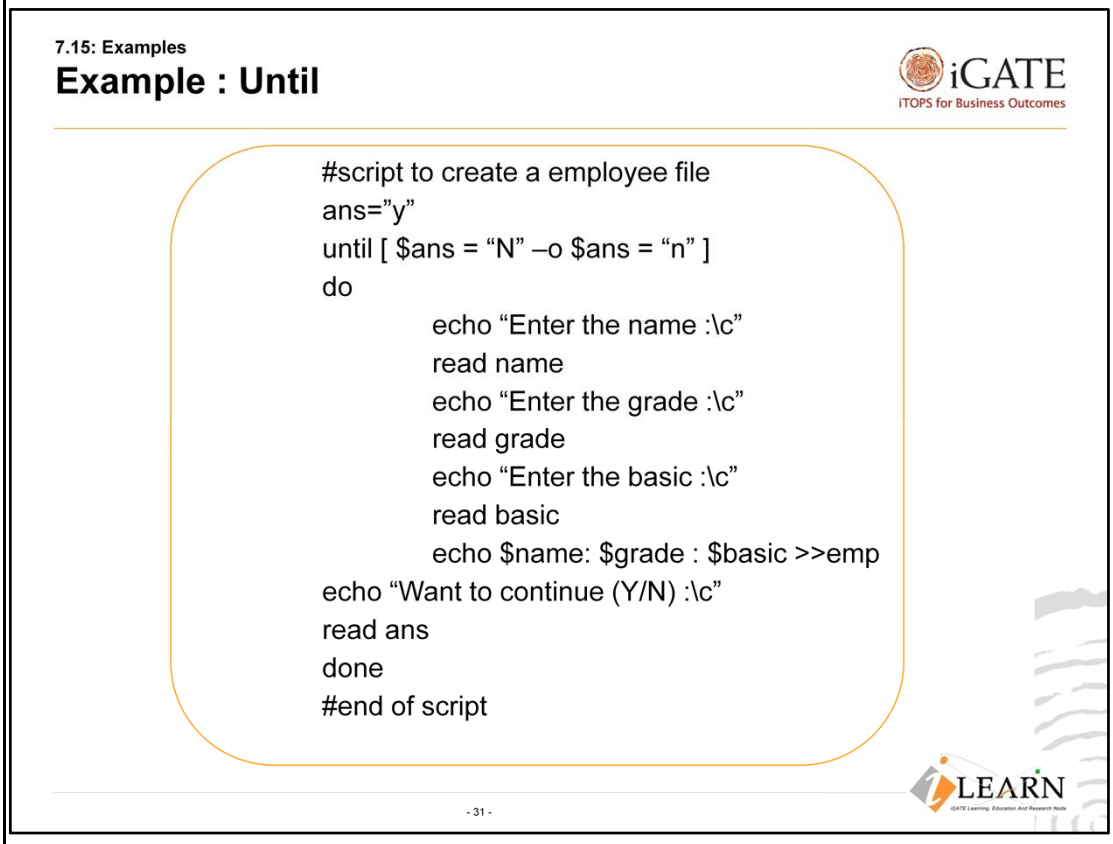
- 29 -



In above example, syntax before the first iteration, *expr1* is evaluated. This is usually used to initialize variables for the loop. All statements between *do* and *done* are executed repeatedly until the value of *expr2* is true.

After each iteration of the loop, *expr3* is evaluated. This is usually used to increment a loop counter.

The output of the given example is:
Welcome 0 times
Welcome 1 times
Welcome 2 times
Welcome 3 times
Welcome 4 times
Welcome 5 times



In above example the loop executes till the condition is false. This is as soon as the user enters "N" or "n" for *ans*, the condition is true and the loop stops iteration.

Some more examples of shell script are:

Script to accept five numbers and display their sum:

```
        echo the parameters passed are : $1, $2, $3, $4, $5
        echo the name of script is      : $0
        echo the number of parameters passed are : $#
        #calculate the sum
        sum=`expr $1 + $2 + $3 + $4 + $5`
        echo the sum is $sum
#end of script
```

Invoke this script as follows:

$sh disp_sum 10 12 13 14 15

The above command is to be followed by 5 different number as shown.

7.16: Shell functions

Functions in Shell Script

iGATE iTOPS for Business Outcomes

- Use shell functions to modularize the script.
- These are also called as script module
- Normally defined at the beginning of the script.
- Syntax (Function Definition):

```
functionname(){
        commands
        }
```

- Example: Function to create a directory and change directories:
- Use mkcd mydir to call the function. mydir is used as $1 in the function.

```
mkcd()
{
mkdir $1        --$1 is the argument we pass while calling function
cd $1
}
```

iLEARN iGATE Learning, Education And Research Node

- 32 -

You can also call the shell function *script module* as it makes a whole script section available under a single name. Normally, shell functions are defined at the beginning of the script. Or several functions can be stored in a file and read whenever they are needed. Files are stored in the *bin* directory. Function name can be any combination from the regular character string.

7.16 : Shell functions

Using return statement

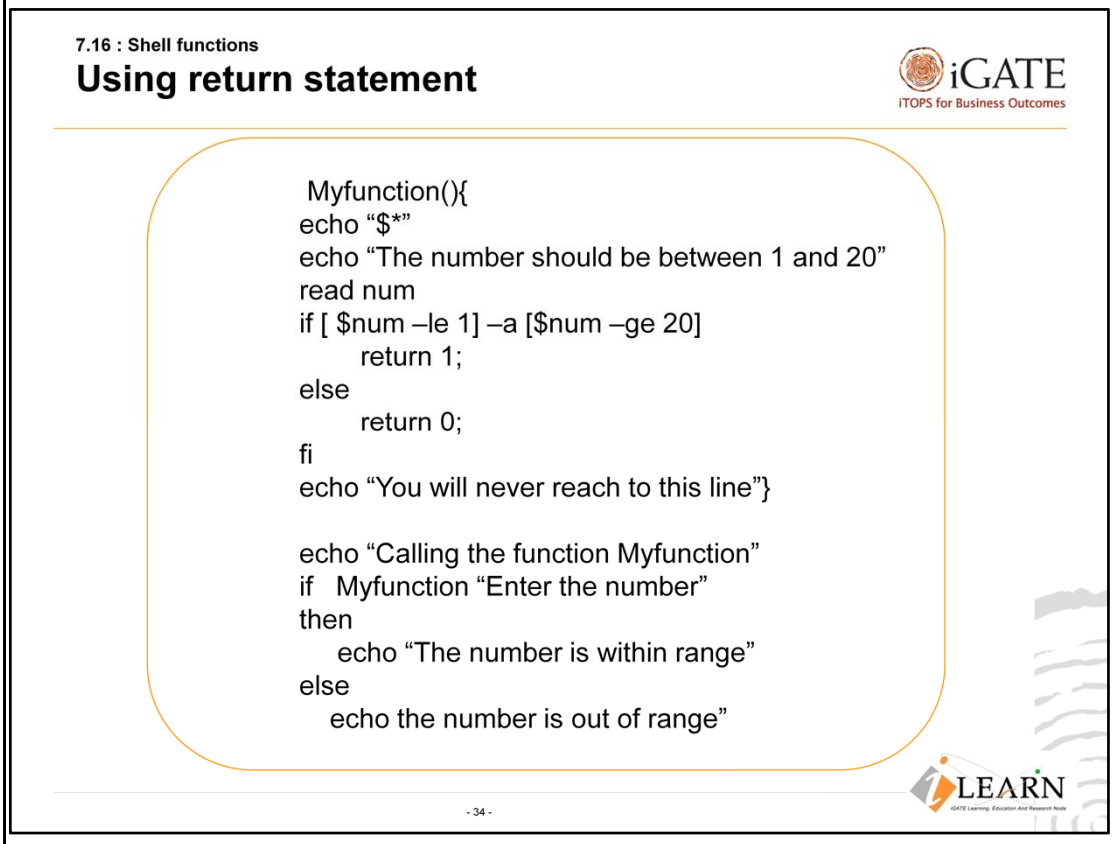
iGATE iTOPS for Business Outcomes

- **Used to come out of a function from within.**
  - If called *without* an argument, function return value is the same as *exit* status of the last command executed within the function
  - If called *with* an argument it returns the argument specified.
  - Example:

```
functret()
{
command1
if …….
then
      return 1
else
      return 0
fi
Command2
}
```

iLEARN iGATE Learning, Education And Research Node

- 33 -



In the above example, *Myfunction* is called in the *if* statement with message "enter the number". This message is passed as three arguments.

In *Myfunction,* the first line is echo $*.
Hence, it display message "Enter the number".
Read num accepts the number.
If the number is between 1 and 20, the function returns 1, otherwise it returns 0.

If *Myfunction* returns 1, then the output is:
The number is within range.

Otherwise the output should be as follows:
The number is out of range.

7.16 : Arrays

Using arrays

iGATE iTOPS for Business Outcomes

- **Contains a collection of values accessible by individuals or groups**
  - Subscript of array element indicates their position in the array.
    - arrayname[subscript]
- **First element is stored at subscript 0.**
  - Assign a value in *flowers* array at the first position.
    - Flowers[0]=Rose
- **Assign values in an array with a single command:**
  - $ set -A Flowers Rose Lotus
- **Access individual array elements**
  - ${arrayname[subscript]}

iLEARN iGATE Learning, Education And Research Node

- 35 -

7.16 : Arrays

Using arrays

iGATE iTOPS for Business Outcomes

- **To print values from array we can use while loop**

```
flowers[0]=Rose
 flowers[1]=Lotus
flowers[2]=Mogra
i=0
while [ $i -lt 3 ]
do
echo ${flowers[$i]}
 i=`expr $i+1`
done
```

- **Access all elements:**

```
${array_name[*]}
${array_name[@]}
```

iLEARN iGATE Learning, Education And Research Node

- 36 -

You can display all elements from the array using * or @ symbol:

```
Num[0]="Zero"
Num[1]="One"
Num[2]="Two"
Num[3]="Three"
echo "First Method: ${NAME[*]}"
echo "Second Method: ${NAME[@]}"
```

Summary

iGATE
iTOPS for Business Outcomes

- **.profile:**
  - Script executed during login time.
- ***Command enclosed in backquotes (`):***
  - Shell executes the command first
  - Enclosed command text is replaced by the command the output.
- **Test:**
  - Command used to check the condition in an if statement.
- **Different loop statements in Unix are:**
  - For
  - While
  - Until

iLEARN
iGATE Learning, Education And Research Node

- 37 -

Review Questions

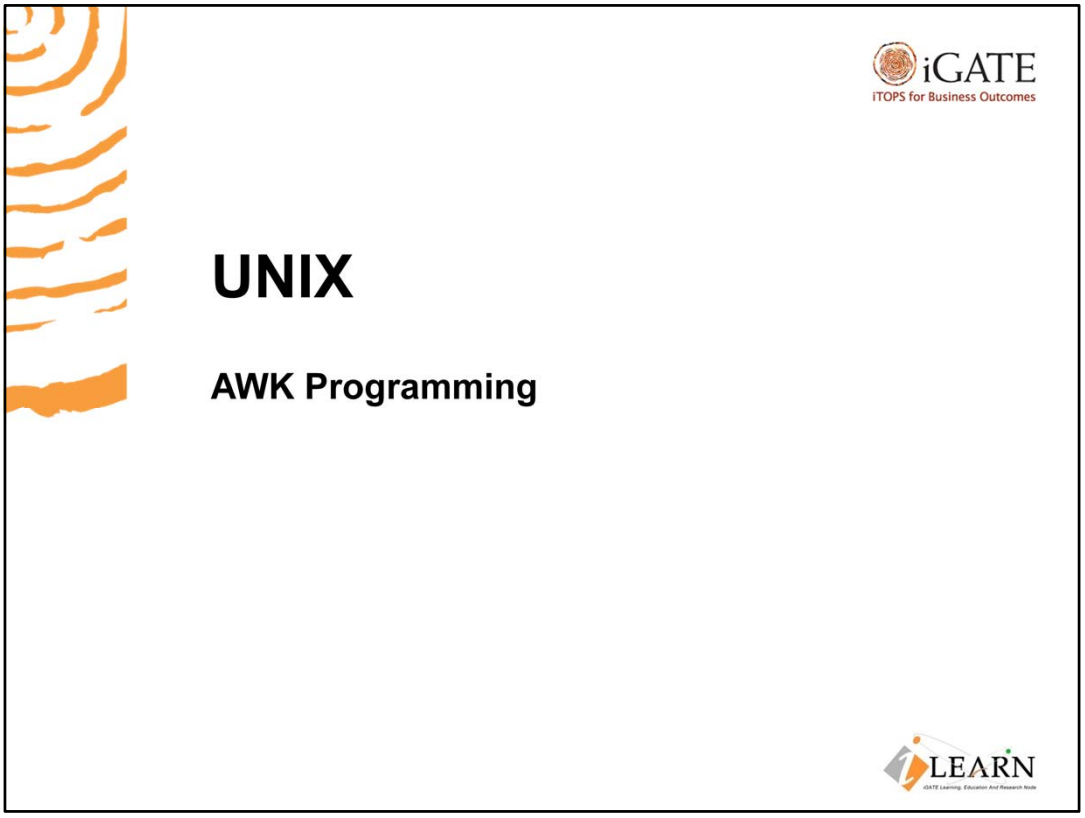
iGATE
iTOPS for Business Outcomes

- **Complete The Following**
  - ------- command can be replaced by test command.
  - ------ condition checks whether two strings are equal or not.
  - ------ loop terminates as soon as condition becomes true.

- **TRUE OR FALSE**
  - PS1 stores primary cursor string:

iLEARN

- 38 -

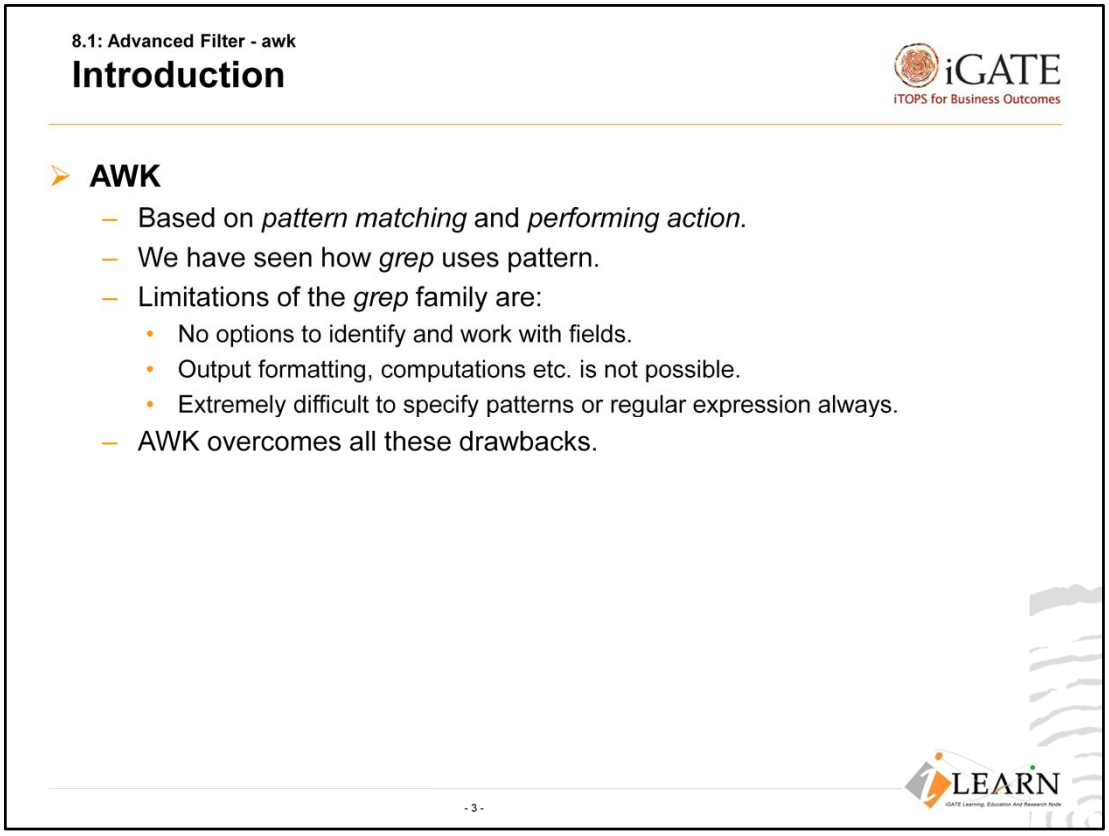

iGATE
iTOPS for Business Outcomes
UNIX
AWK Programming
iLEARN

Lesson Objectives

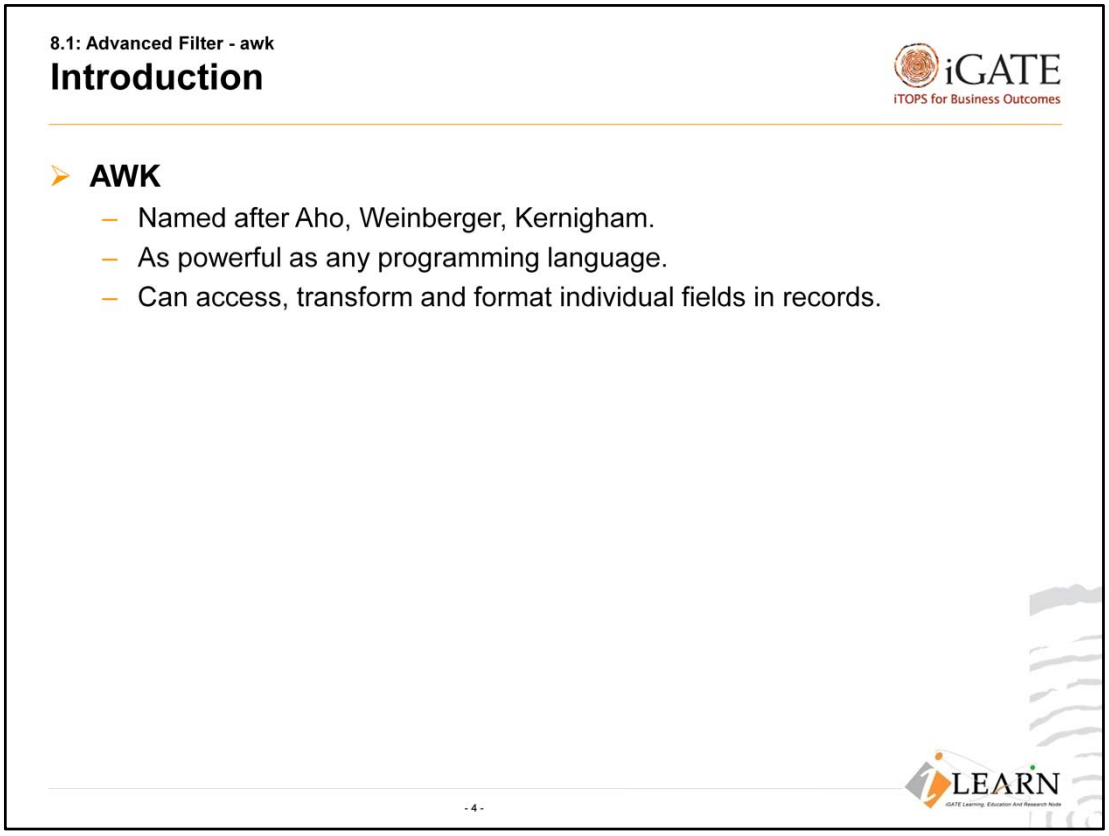
iGATE
iTOPS for Business Outcomes

- **At the end of the session you is able to understand:**
  - How to write simple *awk* scripts.

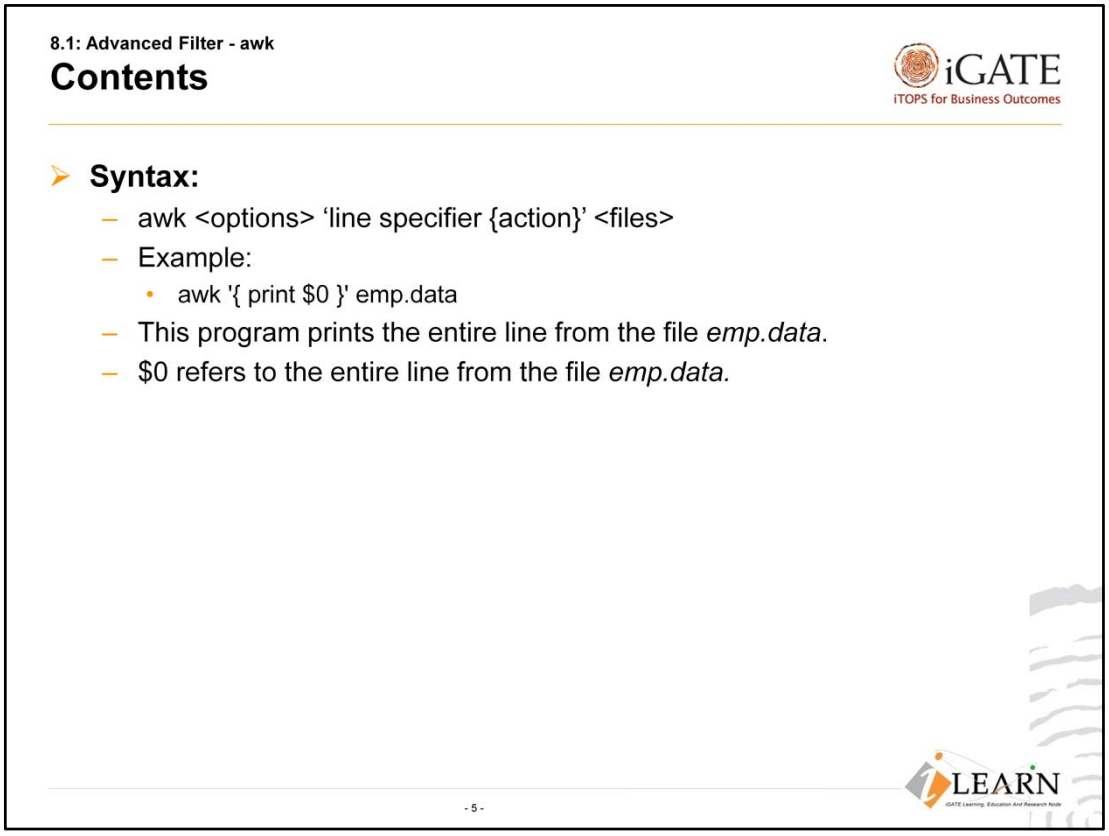
iLEARN
iGATE Learning, Education And Research Node



The *awk* command, named after its authors Aho, Weinberger and Kernighan, is one of the most powerful utilities for text manipulation. It combines features of many other filters. It can access, transform and format individual fields in a record – it is also called as a report writer. It can accept regular expressions for pattern matching, has “C” type programming constructs, variables and in-built functions. In fact, awk is nearly as powerful as any other programming language. However, awk programs are generally slow, if any alternative commands are available to do the same use them rather than using AWK.



Filters accept some data as input, perform some manipulation on it and produce some output. Some simple filters have been discussed in the previous chapters. This chapter discusses advanced filter –*awk*.



Advanced Filter awk.

The general syntax of the awk command is:

awk <options> 'line specifier {action}' <file(s)>

Simple awk Filtering

Following is an example of a simple awk command:

It prints all lines from file *books.lst* in which pattern 'Computer' is found.

$ awk '/Computer/ {print}' books.lst

Output:

```
1001|Learning Unix              |Computers    |01/01/1998| 575
1003|XML Unleashed              |Computers    |20/02/2000| 398
1004|Unix Device Drivers         |Computers    |09/08/1995| 650
1007|Unix Shell Programming     |Computers    |03/02/1993| 536
```

8.1: Advanced Filter - awk

AWK variables

iGATE
iTOPS for Business Outcomes

- **Variable List:**
  - **$0:** Contains contents of the full current record.
  - **$1..$n:** Holds contents of individual fields in the current record.
  - **NF:** Contains number of fields in the input record.
  - **NR:** Contains number of record processed so far.
  - **FS:** Holds the field separator. Default is *space* or *tab.*
  - **OFS:** Holds the output field separator.
  - **RS:** Holds the input record separator. Default is a new line.
  - **FILENAME:** Holds the input file name.
  - **ARGC:** Holds the number of Arguments on Command line
  - **ARGV:** The list of arguments

iLEARN

- 6 -

8.1: Advanced Filter - awk

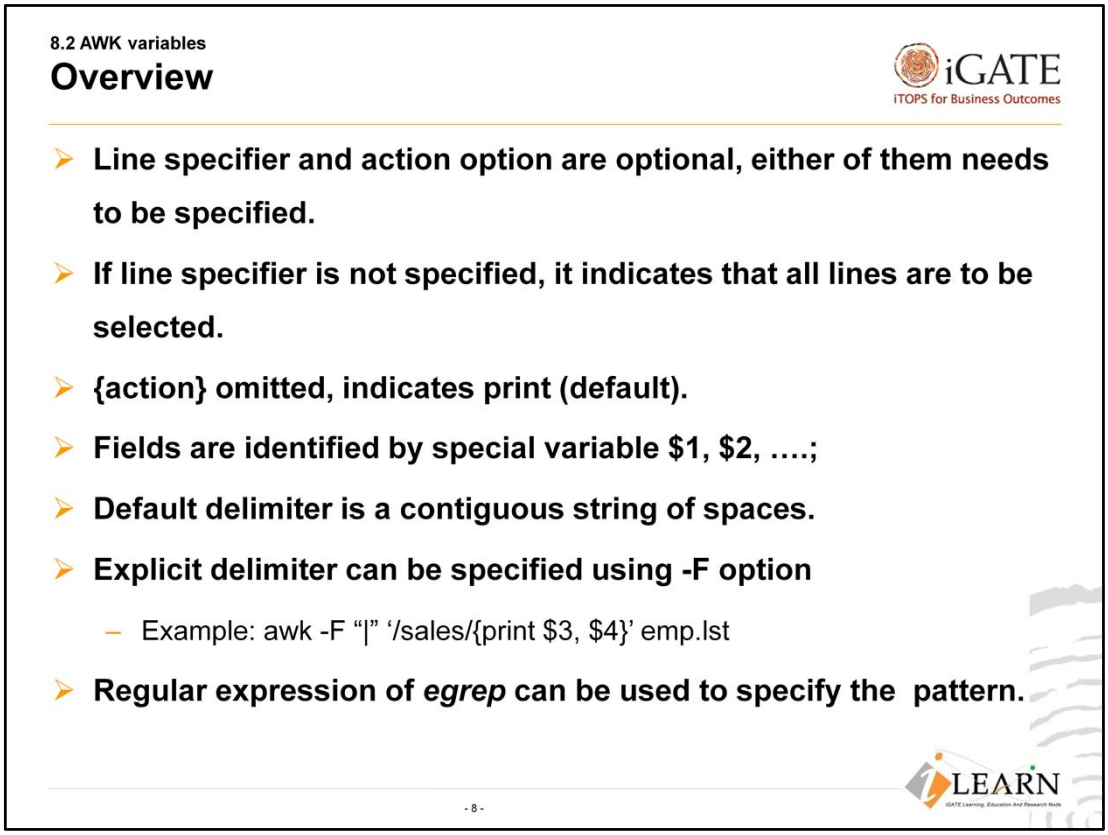
Example

iGATE
iTOPS for Business Outcomes

- **awk '{ print $1 $2 $3 }' emp.data**
  - This prints the *first, second* and *third* column from file *emp.data.*
- **awk '{ print }' emp.data**
  - Prints all lines (all the fields) from file *emp.data.*

iLEARN
iGATE Learning, Education And Research Node

- 7 -



The line specifier uses a context address to specify the lines that need to be taken up for processing in the action section. If the line specifier is missing, then the action is applicable to all lines of the file.

In the action part, statement {print} indicates that selected lines are to be printed. The statement {print} is equivalent to {print $0} - $0 being the variable for the entire line. The awk command is also capable of breaking each line into fields – each field is identified as $1, $2 etc.

For the purpose of identification of fields, awk uses a contiguous string of spaces as field delimiter. However, it is possible to use a different delimiter. This is specified using the –F option in awk.

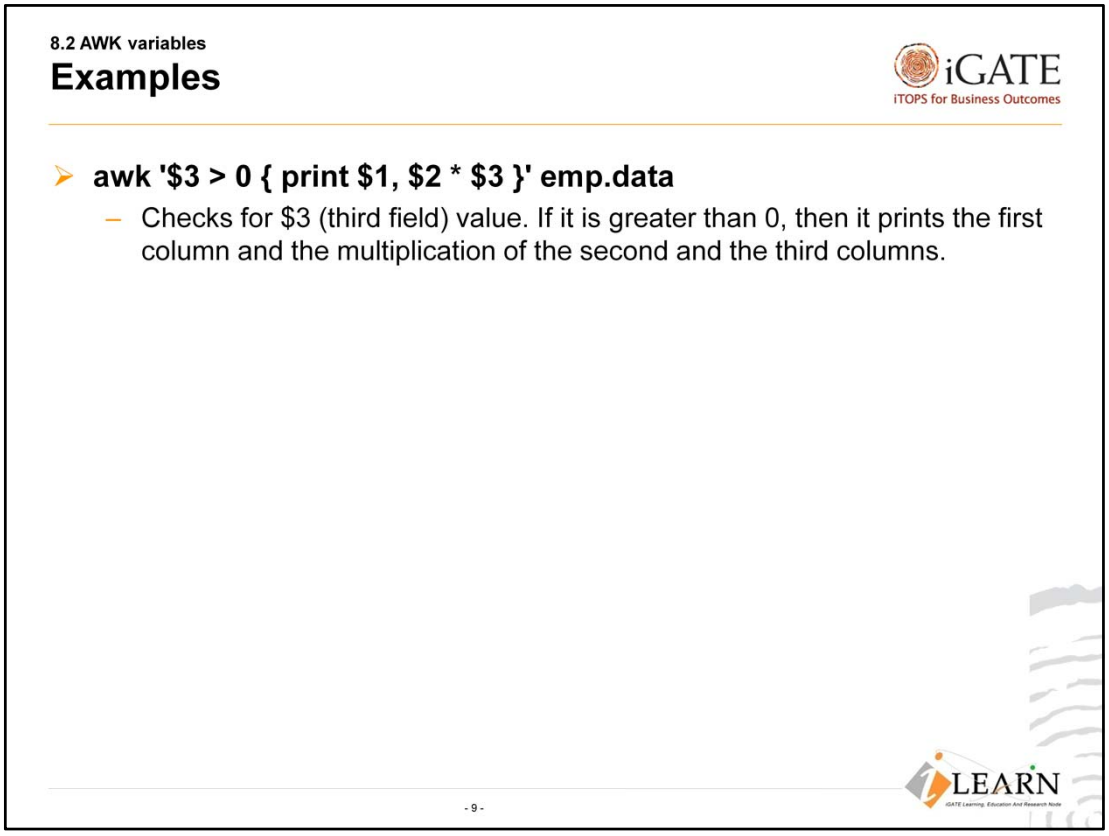
$ awk -F"|" '/Computer/ {print $2,$5}' books.lst
Output:
Learning Unix 575
XML Unleashed 398
Unix Device Drivers 650
Unix Shell Programming 536

For pattern matching, awk uses regular expressions of *egrep* variety.
$ awk -F"|" '/XML|Unix/ {print $2, $5}' books.lst
Output:
Learning Unix 575
XML Unleashed 398
Unix Device Drivers 650
XML Applications 630
Unix Shell Programming 536

It is possible to specify line numbers in file using the inbuilt NR variable. Also, awk can use the C-like *printf* statement to format the output.

$ awk -F"|" '$1=="1002" {printf "%2d,%-20s",NR,$2}' books.lst
Output:
2,Moby Dick

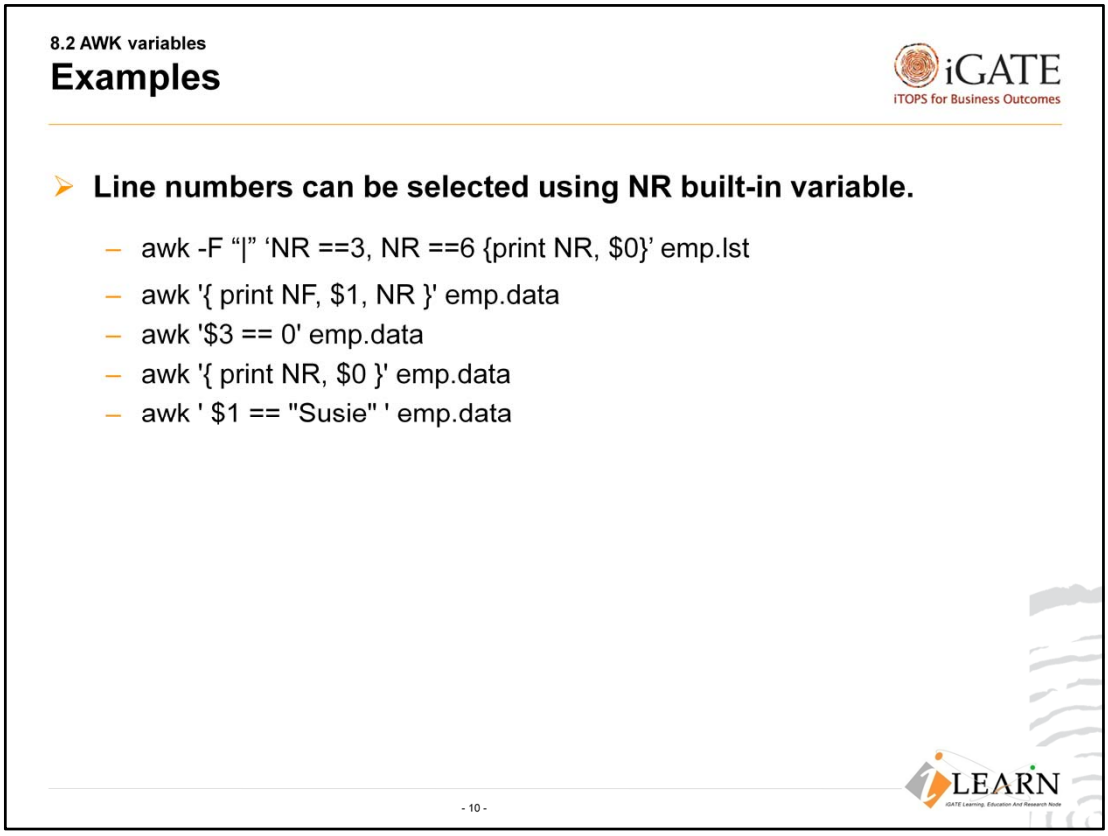
The –f option of awk is useful if you wish to store the line specifier or action in a separate file.



The logical operators || (or) and && (and) are used by the awk command to combine conditions in the line specifier. A relational operator can be used in the line specifier, also in the action component.

The operators == (equal) and != (not equal) can handle only fixed length strings and not regular expressions. To match regular expressions, ~ (match) and !~ (negate) are used. The characters ^ and $ can be used for looking for a pattern in the beginning and end of field.

To work with numbers, operators like < (less than), > (greater than), <= (less than or equal), >= (greater than or equal), == (equal) and != (not equal) can be used.
It is possible to perform computations on numbers using C like arithmetic operators (+, -, *, /, %, ++, --, += etc).
$ awk -F"|" '/Unix/ && $5 < 600 {printf "%s,%d\n",$2,$5}' books.lst
Output:
Learning Unix ,575
Unix Shell Programming ,536
$ awk -F"|" '$2=="Learning Unix"' books.lst
$ awk -F"|" '$2~/Learning Unix/' books.lst
1001|Learning Unix |Computers |01/01/1998| 575
$ awk -F"|" '$5<500 {
> cnt=cnt+1
> printf "%d %s\n",cnt,$2}' books.lst
1 Moby Dick
2 XML Unleashed



awk -F "|" 'NR ==3, NR ==6 {print NR, $0}' emp.lst

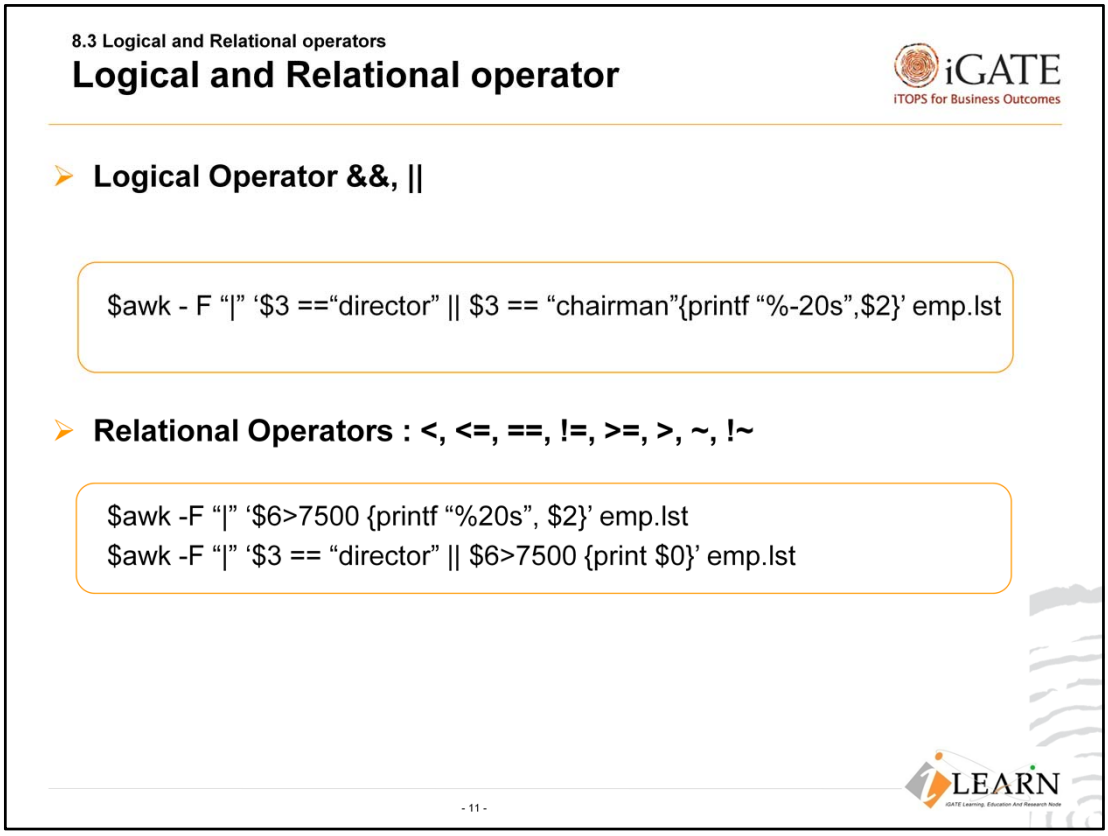
In this example, NR represents record number. It prints records from *third record* to *sixth* record.
-F is use to specify field separator.

awk '{ print NF, $1, NR }' emp.data
This prints number of fields, contents of field 1 and record number for all records.

awk '$3 == 0' emp.data
It prints all lines in which the value in the third field is 0.

awk '{ print NR, $0 }' emp.data
It will print record number and record for all records

awk ' $1 == "Susie" ' emp.data
It prints all lines in which the value in the first field is Susie.



The logical operators || (or) and && (and) are used by the awk command to combine conditions in the line specifier. A relational operator can be used in the line specifier, also in the action component.

The operators == (equal) and != (not equal) can handle only fixed length strings and not regular expressions. To match regular expressions, ~ (match) and !~ (negate) are used. The characters ^ and $ can be used for looking for a pattern in the beginning and end of field.

For working with numbers, operators like < (less than), > (greater than), <= (less than or equal), >= (greater than or equal), == (equal) and != (not equal) can be used.

It is possible to perform computations on numbers using C like arithmetic operators (+, -, *, /, %, ++, --, += etc).

Print all lines with Unix pattern in the line and value of 5 th field should be < 600:

$ awk -F"|" '/Unix/ && $5 < 600 {printf "%s,%d\n",$2,$5}' books.lst

Output

```
Learning Unix                ,575
Unix Shell Programming   ,536
$ awk -F"|" '$2=="Learning Unix"' books.lst
$ awk -F"|" '$2~/Learning Unix/' books.lst
1001|Learning Unix           |Computers     |01/01/1998| 575
$ awk -F"|" '$5<500 {
> cnt=cnt+1
> printf "%d %s\n",cnt,$2}' books.lst
1 Moby Dick
2 XML Unleashed
```

Example of awk command using relational, logical expressions and computations.

8.3 Logical and Relational operators

Logical and Relational operators

iGATE
iTOPS for Business Outcomes

- **== tries to find perfect match.**
  - String may have trailing spaces.
  - To overcome this you can use "~" and "!~" (match and negate of match) with the regular expression.
  - Example:

```
$awk -F "|" '$2~/director/||$2~/g.m/{printf $0}' emp.lst
$3 ~/^g.m/    # Beginning with g.m
```

iLEARN
iGATE Learning, Education And Research Node

- 12 -

8.4 Arithmetic operators

Arithmetic operators

iGATE iTOPS for Business Outcomes

- **Computation on numbers is done:**
  - +, -, *, /, % operator available.
  - No type-declaration for variables.
  - Variables are initialized to zero.

```
$awk -F "|" '$3 == "director" || $6>7500 {
>kount = kount+1
>printf "%3d%-20s\n", kount,$2}' emp.lst
```

- **Can also use C constructs like kount++, Kount+=sum etc.**

iLEARN iGATE Learning, Education And Research Node

- 13 -

8.5 awk command

awk command

iGATE
iTOPS for Business Outcomes

- **-f option**
  - awk program can be written in a separate file and used in awk.
    - Example:

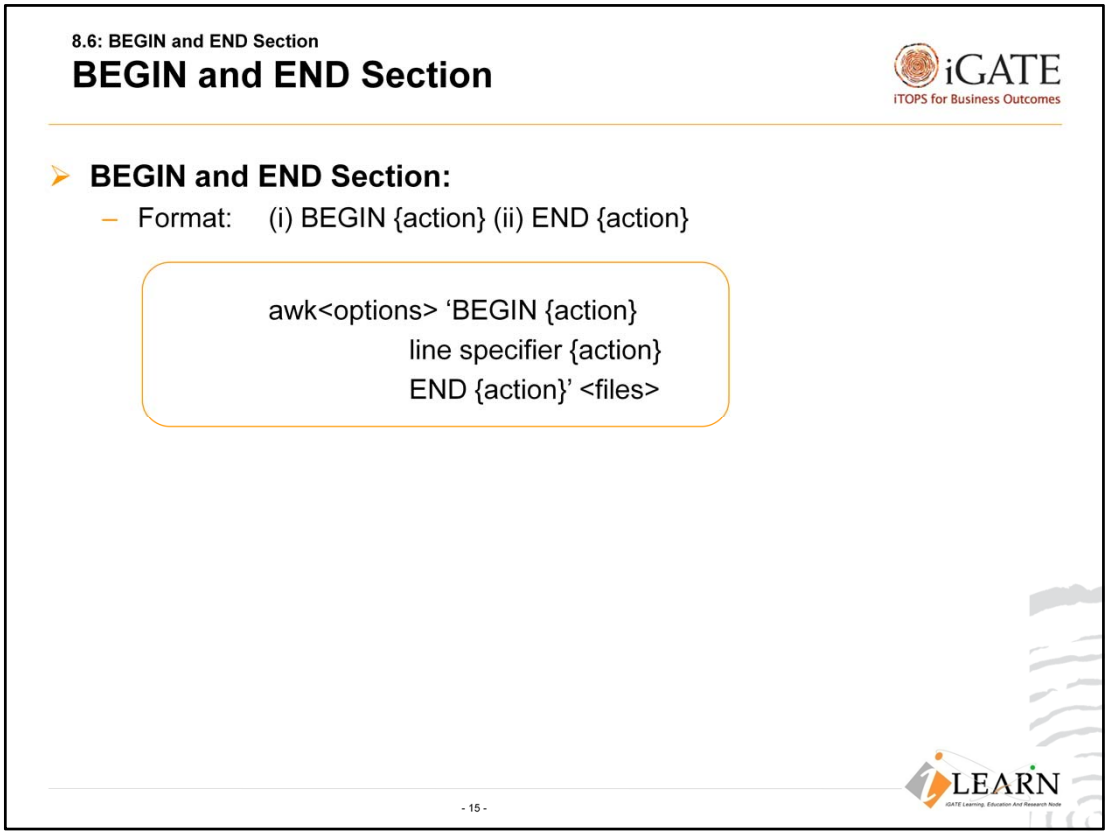
```
$awk -F "|" -f emp.awk    emp.lst
```

  - Single quoted contents are written in this file without quotes.

```
$cat emp.awk
$3 == "Director" {print $2,$4,$5}
$
```

iLEARN
iGATE Learning, Education And Research Node

- 14 -



BEGIN and END Sections

In case, something is to be printed before processing the first line begins, BEGIN section can be used. Similarly, to print at the end of processing, END section can be used.

These sections are optional. When present, they are delimited by the body of the awk program.

Normally if you want to print any header lines or want to set field separator then use BEGIN section

And If you want to display total or any summarized information at the end of the report then use END section

8.6 BEGIN and END Section

Example

iGATE
iTOPS for Business Outcomes

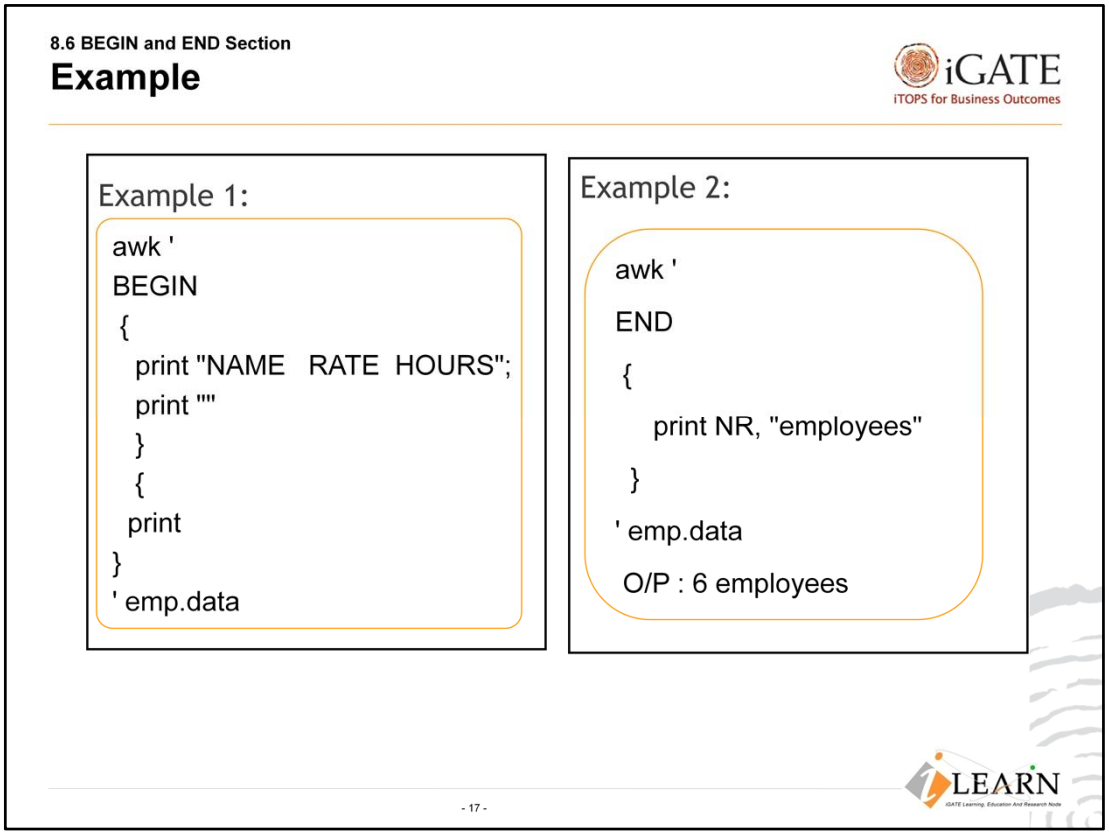
```
$cat emp.awk
BEGIN {  printf “\n\t Employee details \n\n”
}
$6>7500{
# increment sr. no. and sum salary
kount++; tot+=$6
printf “%d%-20s%d\n”, kount, $2, $6
}
END {  printf “\n The Avg. Sal. Is %6d\n”, tot/kount
}
$awk -F “|” -f emp.awk emp.lst
```

iLEARN

- 16 -

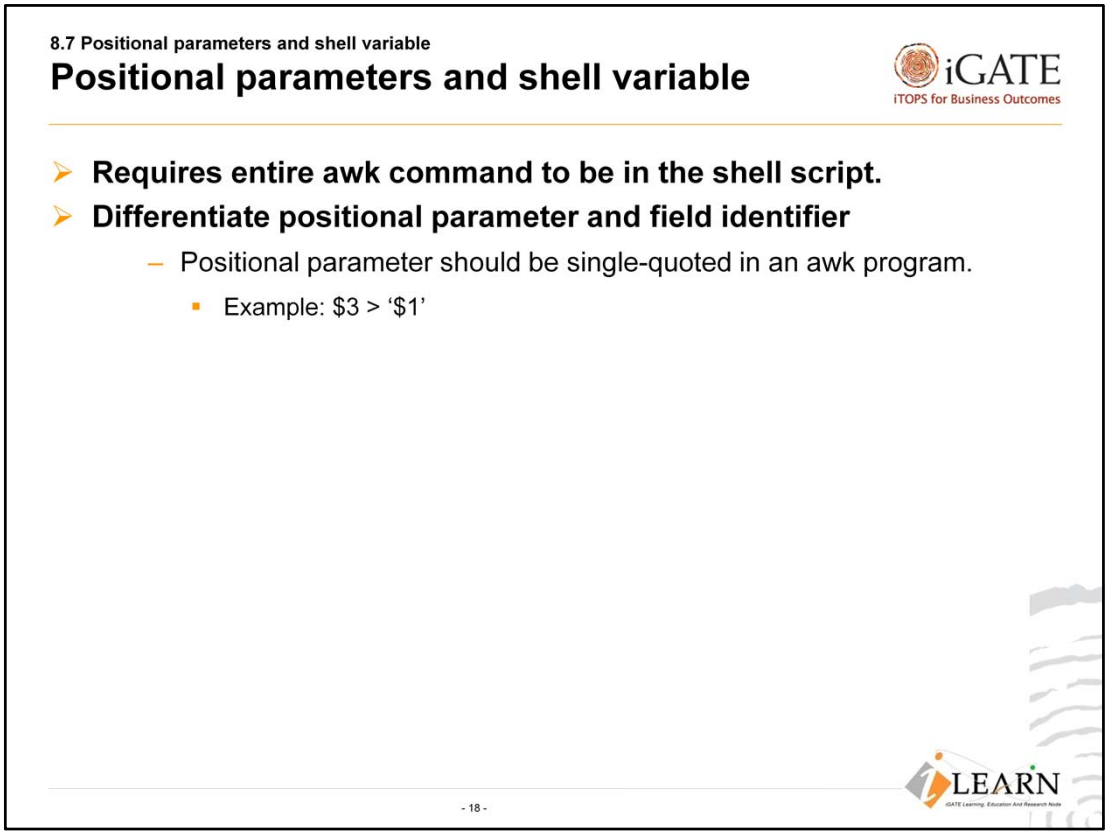
$ cat awk1.awk
BEGIN {
printf "\n\t Unix Book Details \n\n"
}
$2~/Unix/ {
cnt++; tot+=$5
printf "%d %-15s\n",cnt,$2
}
END {
printf "\n Total Cost is %d\n",tot
}
$ awk -F"|" -f awk1.awk books.lst
Output:

Unix Book Details
1 Learning Unix
2 Unix Device Drivers
3 Unix Shell Programming
Total Cost is 1761



Example 1:
In this example It will perform action enclosed in BEGIN section (It prints headings) and then it will perform print action on all lines in the file emp.dat (because no line specifier is mentioned)

Example 2 :
In this example BEGIN section is not given. The main action part is also empty hence no action is performed on all the records and then it will execute End section which prints last record number which will give you total number of records.



Positional Parameters and Shell Variables

It is possible to store an entire awk command into a file as a shell script, and pass parameters as arguments to the shell script. The shell will identify these arguments as $1, $2 etc based on the order of their occurrence on the command line.

Within awk, since $1, $2 etc. indicate fields of data file, it is necessary to distinguish between the positional parameters and field identifiers. This is done by using single quotes for the positional parameters used by awk.

```
$ cat awk2.awk
awk -F"|" '$5>'$1' {
cnt++
printf "%d %s %d\n",cnt,$2,$5 }
END {
printf "\n No. of books costing more than specified amount is:%d\n",cnt
}' books.lst
$ awk2.awk 600
```

Output:

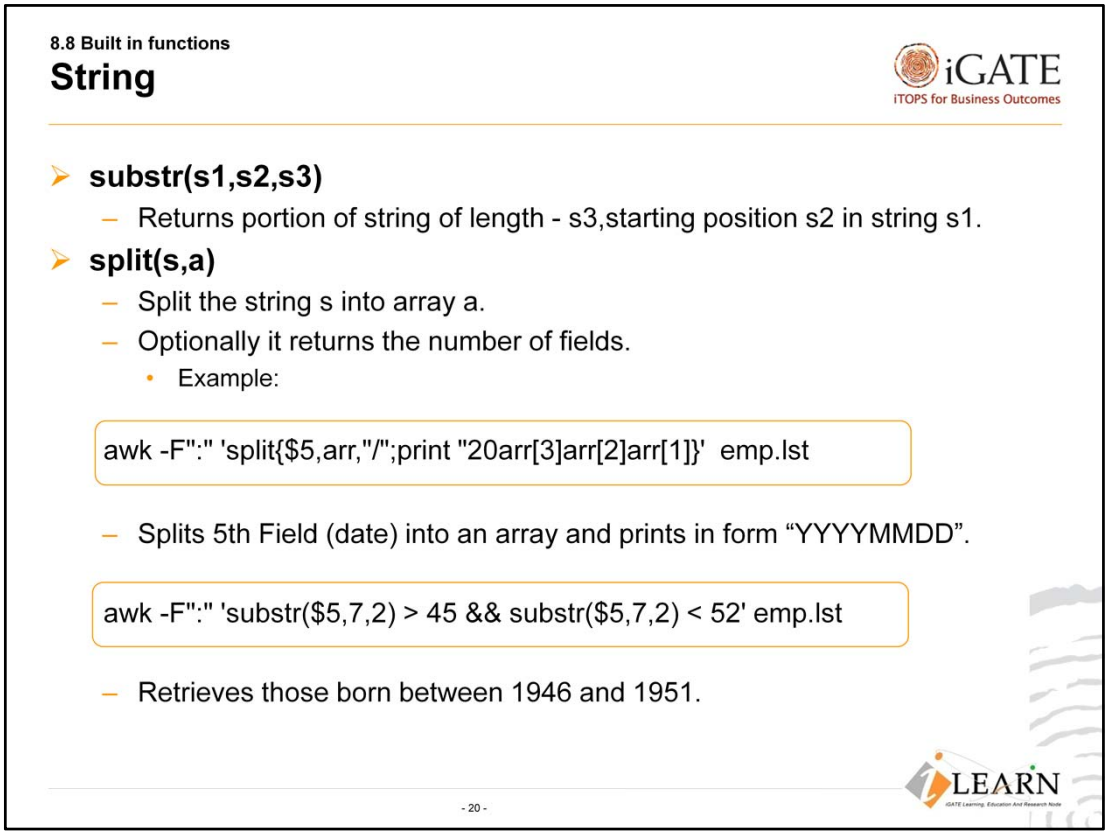
```
1 Unix Device Drivers       650
2 Complete Works:Sherlock H 1290
3 XML Applications          630
No. of books costing more than specified amount is:3
$ awk2.awk 1000
1 Complete Works:Sherlock H 1290
No. of books costing more than specified amount is:1
```

(Example of awk command using positional parameter.)

8.8 Built in functions

Numeric Functions

- **int(x)**
  - Returns integer value of x.
- **sqrt(x)**
  - Returns square root of x.
- **index(s1,s2)**
  - Returns the position of string s2 in s1.
- **length()**
  - Returns length of the argument.



awk -F":" 'length > 1024' emp.lst
Retrieves all lines > 1024.
awk -F":" 'length($2) > 11' emp.lst
Retrieves all lines whose characters in 2nd col < 11.

awk -F":" 'substr($5,7,2) > 45 && substr($5,7,2) < 52' emp.lst
Retrieves those born between 1946 and 1951.

8.9: Control statements

Control statements

iGATE
iTOPS for Business Outcomes

```
if (<condition>)
{
        <Block of
statements>
}
else
{
        <Block of
statements>
}
```

```
While (Condition)
{
    <Block of statements>
}
for
(<expr1>;<cond1>;<expr2>)
{
        <Block of
statements>
}
```

iLEARN
iGATE Learning, Education And Research Node

- 21 -

8.9: Control statements

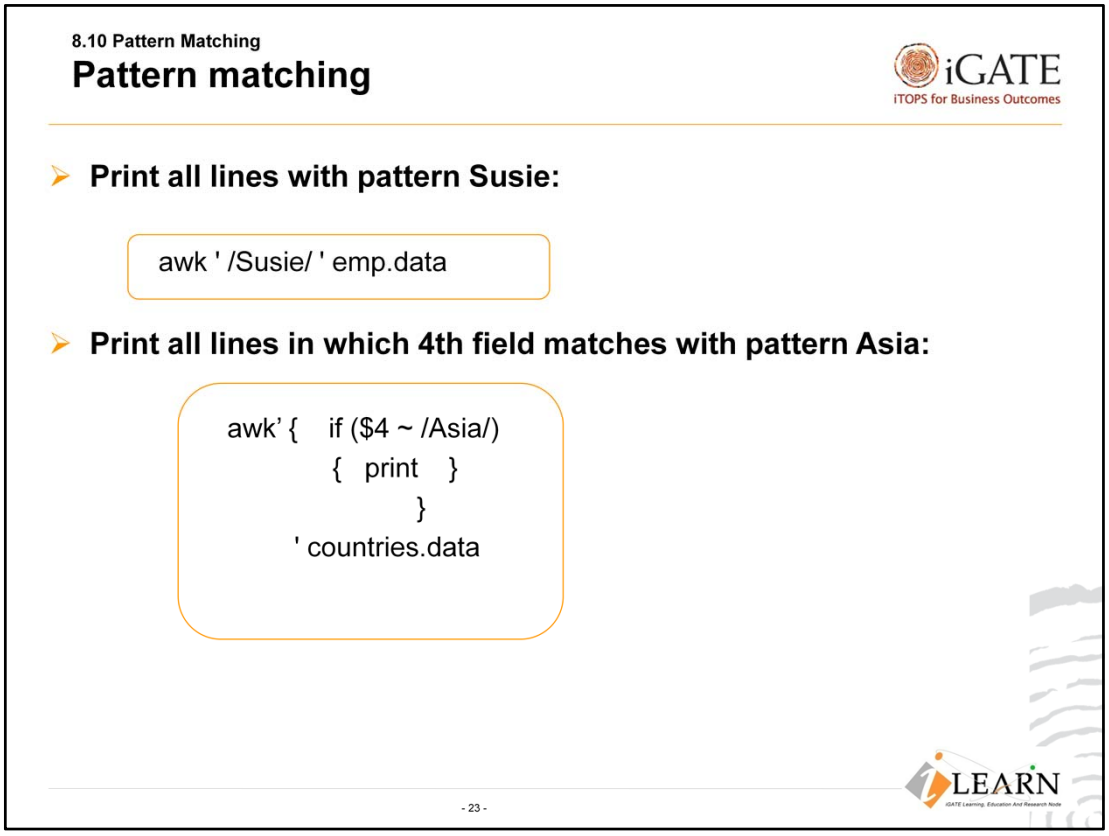
Example

iGATE
iTOPS for Business Outcomes

```
awk '
{
   if (NF != 3)  {
      print $0, "number of fields is not equal to 3“
   if ($2 < 3.35)  {
      print $0, "rate is below minimum wage"   }
   if ($2 > 10)  {
      print $0, "rate exceeds $ 10 per hour"   }
   if ($3 < 0)  {
      print $0, "negative hours worked"   }
   if ($3 > 60)  {
      print $0, "too many hours worked"   }
}
' emp.data
```

iLEARN

- 22 -



To print all lines in which 4 th field does not matches with pattern Asia:

```
awk '
{
   if ($4 !~ /Asia/)
   {
      print
   }
}
' countries.data
```

8.10 Pattern Matching

Example

iGATE
iTOPS for Business Outcomes

Print any line that matches exactly three digits

```
awk '
{
  if (/^[0-9][0-9][0-9]$/)
    {
      print
    }
}
' numbers.data
```

print any line that consists only digits

```
awk '
{
   If (/^[0-9]+$/)
    {
      print
    }
}
' numbers.data
```

iLEARN
iGATE Learning, Education And Research Node

- 24 -

Summary

iGATE
iTOPS for Business Outcomes

- AWK is based on pattern matching and performing action.
- Commands enclosed in BEGIN section gets executed first.
- Then. it performs main action part on all records.
- Commands enclosed in END section gets executed.

iLEARN

- 25 -

Add the notes here.

Review Questions

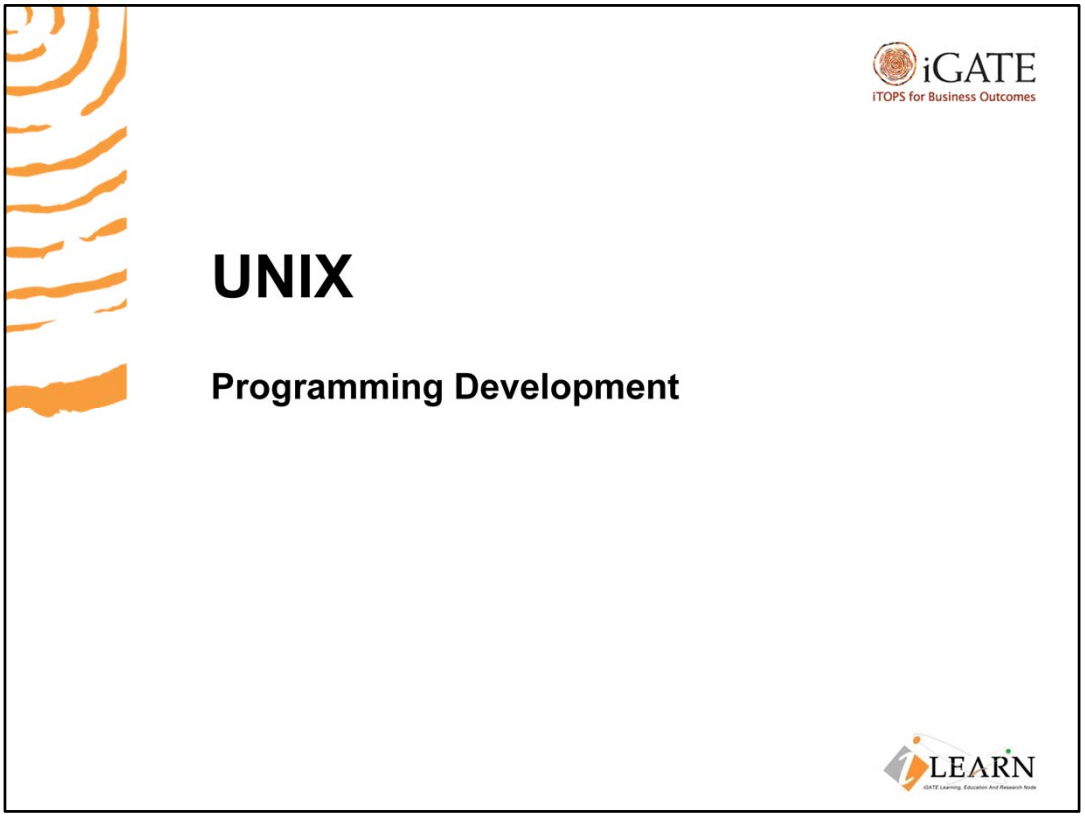
iGATE
iTOPS for Business Outcomes

- In ------ section report header can be printed
- ------ section helps to print totals
- Print $0 prints whole record
  - TRUE
  - FALSE

iLEARN

- 26 -

Add the notes here.


iGATE
iTOPS for Business Outcomes
UNIX
Programming Development
iLEARN

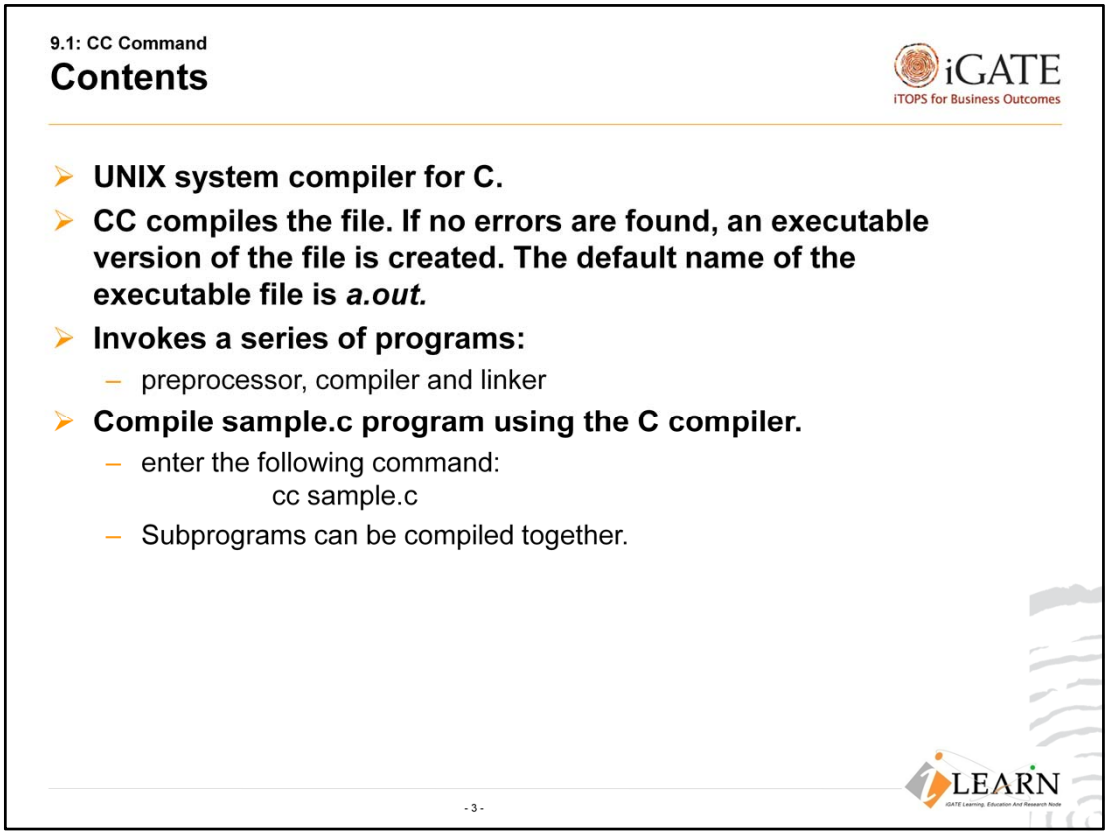
Lesson Objectives

iGATE
iTOPS for Business Outcomes

- CC
- MAKE
- SCCS
- Introduction to System administration

iLEARN
iGATE Learning, Education And Research Node

- 2 -



C Program development (CC , Make, SCCS)

An essential resource for C Program development is the C Compiler. A compiler is a special program that translates program source to executable code. If it encounters a error, it generates the error messages. The compiler also provides a suitable programming environment – file handling, basic I/O, other connections with the operating system.

In Program development there are source files, header files and library files. Library files are precompiled files which contain object code. The source and header files are created by programmers using an editor.(e.g. VI editor). Source files in C have .c extension, c++ has *.cpp* extension, header files have *.h* extension.

Overview of Compilation and Linking

Compilation refers to the processing of source code files (.c,.cpp) and the creation of an 'object' file. This step doesn't create anything the user can actually run. Instead, the compiler merely produces the machine language instructions that correspond to the source code file that was compiled. For instance, if you compile (but do not link) three separate files, you will have three object files created as output, each with the name <filename>.o or <filename>.obj (the extension will depend on your compiler). Each of these files contains a translation of your source code file into a machine language file. But you cannot run them yet! You need to turn them into executables your operating system can use. That is where the linker comes in.

9.1: CC Command

Options with CC Command

iGATE
iTOPS for Business Outcomes

- **Specify your own output (.o) file name**
  - By default cc creates output file by name a.out.
  - To specify own file name use –o option
    - Example: cc –o sample.o sample.c
  - CC compiler creates sample.o file instead of a.out
  - Execute the file:

    $ ./sample.o --- "." represents current directory.

iLEARN
iGATE Learning, Education And Research Node

- 4 -

9.1: CC Command

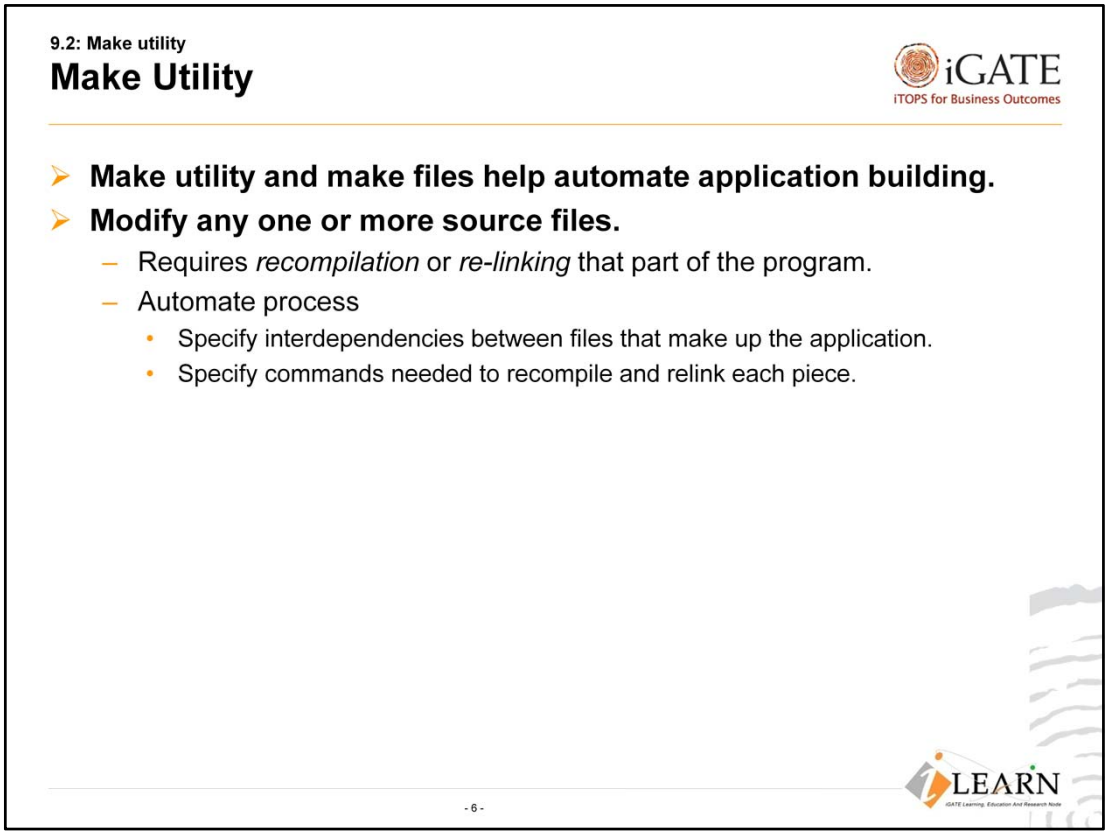
Options with CC Command

iGATE iTOPS for Business Outcomes

- **Some more options with cc**
  - **-c:** Compiles only; does not attempt to link source files.
  - **-l <Library>:** When linking, it adds the indicated library to the list of libraries to be linked.
    - C program includes mathematical functions such as *sin* or *tan.*
    - During compilation you must specify math library usage while linking.
    - Example: cc sample.c –lm --- Uses math library while linking.
  - **-o output:** If linking, places executable output in the file output.

iLEARN iGATE Learning, Education And Research Node

- 5 -



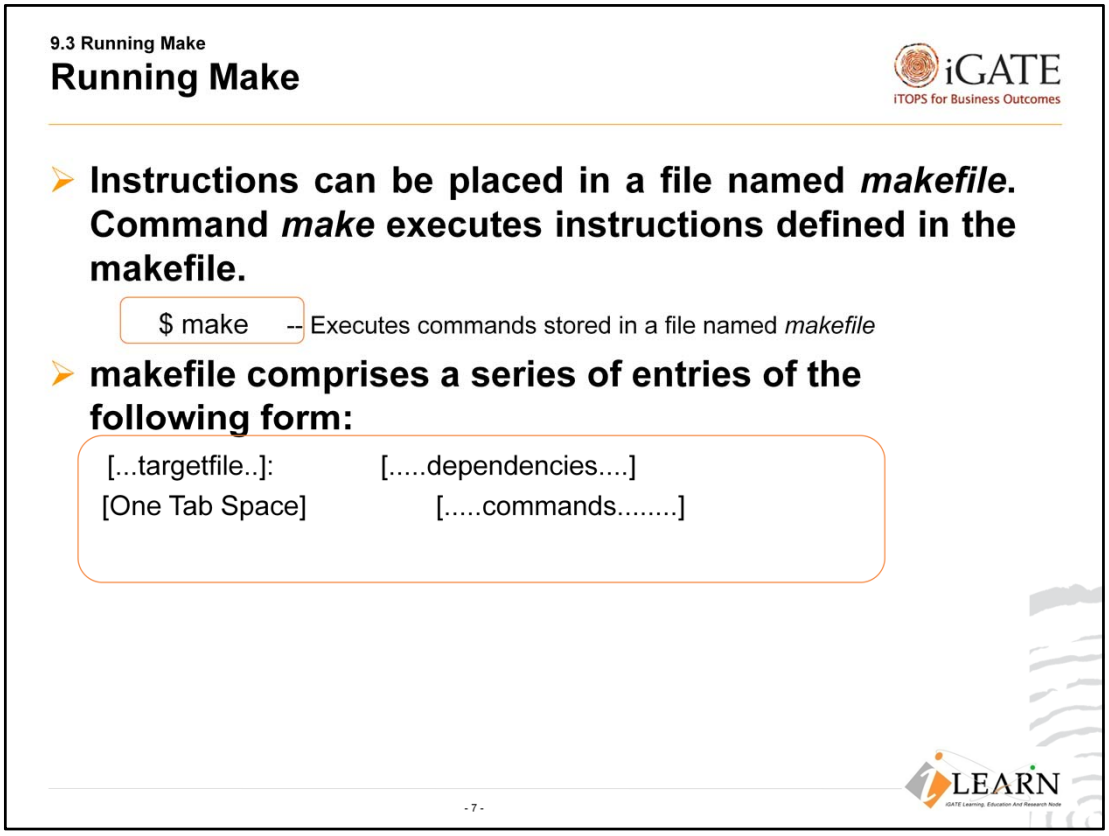
Managing the process of compiling large applications can be difficult. During program development a number of subroutines may be developed separately and linked at the end. However keeping a track of the updated versions is difficult. Further, files that are changed may not be recompiled. UNIX provides a tool that takes care of this for you. **make** looks for a makefile, which includes directions for building the application.

Make Utility

You can use the make utility and makefiles to help automate building of an application.

The make utility applies intelligence to the task of program compilation and linking.

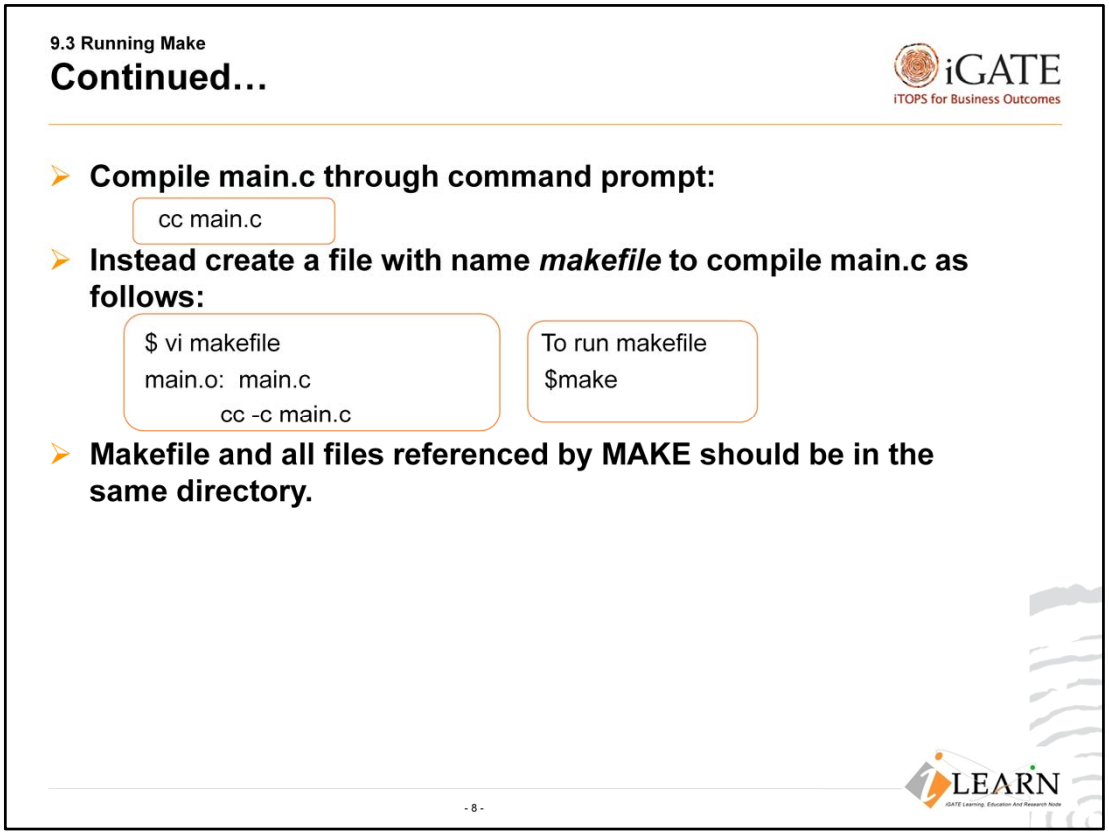
Typically, a large application might exist as a set of source files and INCLUDE files, which require linking with a number of libraries. Modifying any one or more of the source files requires recompilation of that part of the program and relinking. You can automate this process by specifying the interdependencies between files that make up the application along with the commands needed to recompile and relink each piece. With these specifications in a file of directives, **make** ensures that only the files that need recompiling are recompiled and that relinking uses the options and libraries you want.



You can think of the makefile as being its own programming language. The syntax is as follows:

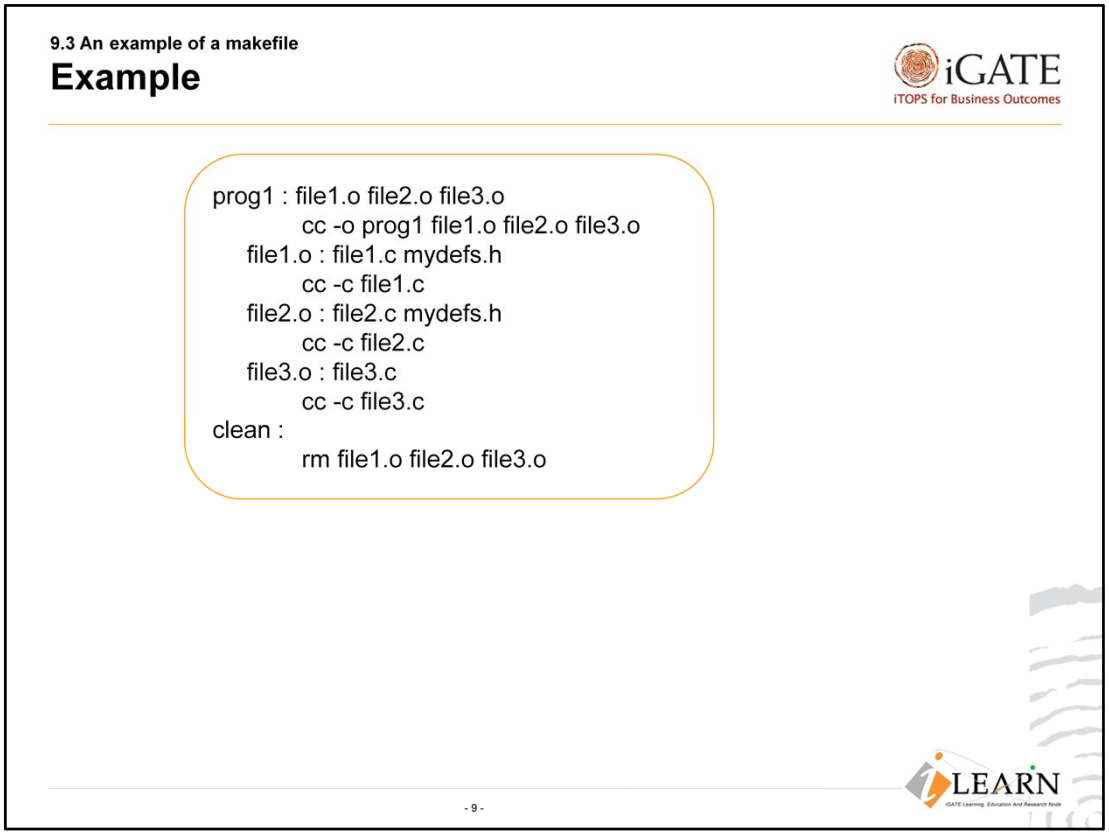
target: dependencies
Command list

Dependencies can be targets declared elsewhere in the makefile, and they can have their own dependencies. When a make command is issued, the target on the command line is checked; if no targets are specified on the command line, the first target listed in the file is checked.



When make tries to build a target, first the dependencies list is checked. If any of them requires rebuilding, it is rebuilt. Then, the command list specified for the target itself is executed.

make has its own set of default rules, which are executed if no other rules are specified. One rule specifies that an object is created from a C source file using $(cc) $(CFLAGS) -c (source file). CFLAGS is a special variable; a list of flags that will be used with each compilation can be stored there. These flags can be specified in the makefile, on the make command line, or in an environment variable. make checks the dependencies to determine whether a file needs to be made. It uses the mtime field of a file's status. If the file has been modified more recently than the target, the target is remade.



This is an example descriptor file to build an executable file called prog1. It requires the source files file1.c, file2.c, and file3.c. An include file, mydefs.h, is required by files file1.c and file2.c. If you wanted to compile this file from the command line using C the command is:

% CC -o prog1 file1.cc file2.cc file3.cc

This command line is rather long to be entered many times as a program is developed and is prone to typing errors. A descriptor file could run the same command better by using the simple command:

% make prog1

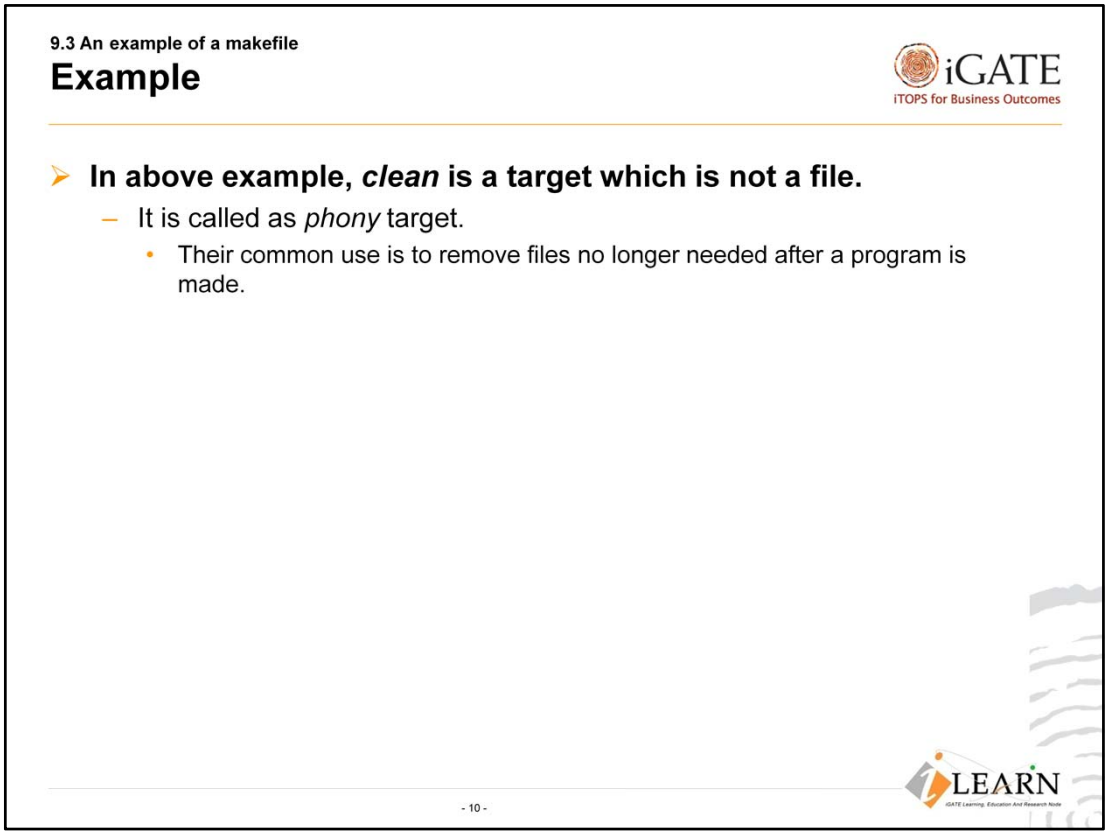
or if prog1 is the first target defined in the descriptor file

% make

This first example descriptor file is much longer than necessary but is useful for describing what is going on.

Let's go through the example to see what make does by executing with the command make prog1 and assuming the program has never been compiled.

1. *Make* finds the target prog1 and sees that it depends on the object files file1.o file2.o file3.o
2. *Make* next looks to see if any of the three object files are listed as targets. So make looks at each target to see what it depends on. Make sees that file1.o depends on the files file1.c and mydefs.h.
3. Now *make* looks to see if either of these files are listed as targets and since they aren't it executes the commands given in file1.o's rule and compiles file1.c to get the object file.
4. *Make* looks at the targets file2.o and file3.o and compiles these object files in a similar fashion.
5. *Make* now has all the object files required to make prog1 and does so by executing the commands in its rule.



You probably noticed we did not use the target, clean, it is called a *phony target*.

A phony target is one that is not really the name of a file. It will only have a list of commands and no prerequisites. One common use of phony targets is for removing files that are no longer needed after a program has been made. The following example simply removes all object files found in the directory containing the descriptor file.
clean : rm *.o

9.3 Running Make

Continued…

iGATE
iTOPS for Business Outcomes

- **Table below displays results of executing make with these options.**

Command	Result
`make`	use the default descriptor file, build the first target in the file
`make myprog`	use the default descriptor file, build the target `myprog`
`make -f mymakefile`	use the file `mymakefile` as the descriptor file, build the first target in the file
`make -f mymakefile myprog`	use the file `mymakefile` as the descriptor file, build the target `myprog`

iLEARN

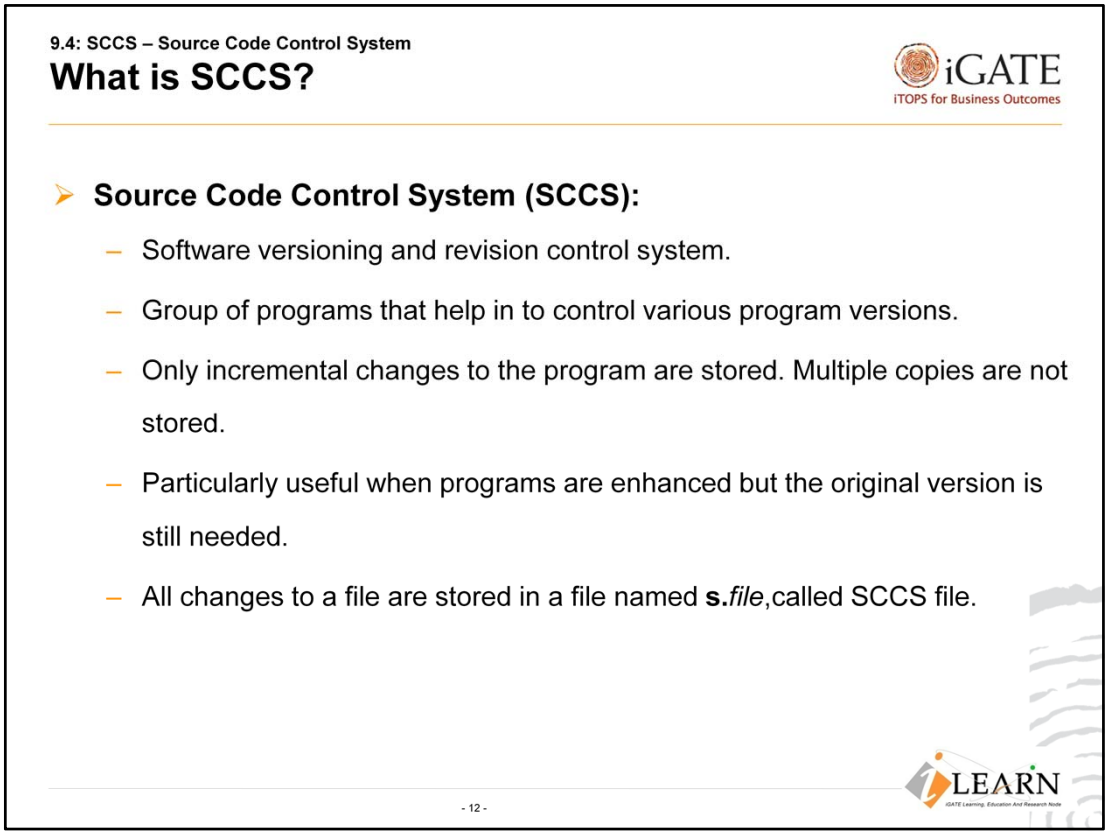
- 11 -

$make
It will use default descriptor file i.e. makefile

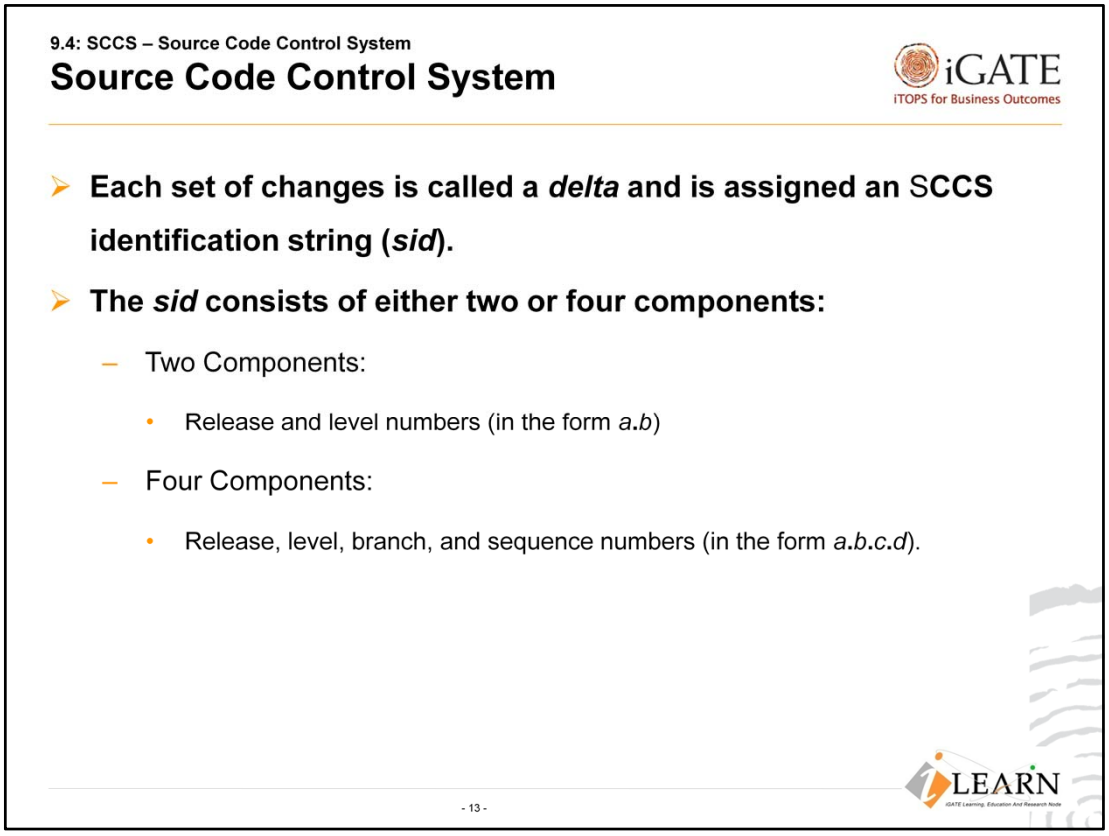
$make myprog
It will use file named makefile and create target myprog.
In makefile if myprog is not the first target then you can specify it wile executing makefile

$make –f mymakefile
If your descriptor file name is different than the default name i.e. makefile the to execute your own file use –f option
In above example make will use mymakefile file instead of using makefile.

$make –f mymakefile myprog
In above example make will use mymakefile instead of makefile to create target myprog



SCCS was developed by AT&T as a system to control source code development. It has features in it that help support a production environment, including freezing of released code and hooks for integration of a problem-tracking system.
The Source Code Control System (SCCS) lets you keep track of each revision of a document, avoiding the confusion that often arises from having several versions of one file on line. SCCS is particularly useful when programs are enhanced but the original version is still needed.



All changes to a file are stored in a file named **s.***file*, which is called an SCCS file. Each time a file is "entered" into SCCS, SCCS notes which lines have been changed or deleted since the most recent version. From that information, SCCS can regenerate the file on demand. Each set of changes depends on all previous sets of changes.

Each set of changes is called a *delta* and is assigned an **S**CCS **id**entification string (*sid*). The *sid* consists of either two components, release and level numbers (in the form *a***.***b*), or of four components: the release, level, branch, and sequence numbers (in the form *a***.***b***.***c***.***d*). The branches and sequences are for situations when two on-running versions of the same file are recorded in SCCS. For example, *delta 3.2.1.1* refers to release 3, level 2, branch 1, sequence 1.

9.5 Commands

Commands with SCCS

iGATE iTOPS for Business Outcomes

- **Following commands are used with SCCS.**
  - admin: used to create initial versions of the file.
  - get: to check out any version
  - delta : to check in
  - comb: used to combine two or more versions and remove redundant versions.
  - remdel: used to remove a version.
  - prs: removes information about a SCCS file.

iLEARN iGATE Learning, Education And Research Node

- 14 -

Consider the file myfile.c

```
#define MSG "hello world!!!"
#include<stdio.h>
int main()
{
      printf(MSG);
      printf("\nhi!!!\n");
return 1;
}
```

If we want to maintain all versions of this program in single encoded file use following sequence of commands

This file doesnot exist in SCCS currently. Hence to check in to SCCS
File with name s.myfile.c use admin command with –I option as follows

$ admin –i myfile.c s.myfile.c

It will create s.myfile.c which contains contents of myfile.c as well as control information like who is owner of the file, time of creation
All versions of myfile.c must be checked in to this file.
If you want to modify the contents of the myfile.c we need to checkout for editting the file using –e option it also creates p.mefile.c as lock file to prevent user to checking out the same file again.

use following command to check out

```
$get –e s.myfile.c
1.1
New delta 1.2
8 lines
```

You can now edit the file myfile.c because we have checked out the file.

Let's change myfile.c as follows

```
#define MSG "hello world!!!"
#include<stdio.h>
int main()
{
        printf(MSG);
        printf("\nhi!!!\n");
        printf("This line is added in 1.2 version");       //this line Is added
return 1;
}
```

Now to save these changes as new version we need to check in the file
Use delta command as follows. It will prompt you for adding comments.

```
$ delta s.myfile.c
Comments? Added one line in new version
No id keywords(cm7)
1.2
1 inserted
```

Now if you use ls –l command you will not find myfile.c because it is in SCCS custody and if you want to modify it you have to check it out again so that changes will be saved as 1.3 version

9.5 Pseudo commands

Commands with SCCS

iGATE
iTOPS for Business Outcomes

- **Pseudo-commands**
  - Equivalent SCCS actions are indicated in parentheses.
    - **Create:** Create SCCS files (**admin -i** followed by **get**).
    - **Deledit:** Same as **delta** followed by **get -e**.
    - **Delget:** Same as **delta** followed by **get**.
    - **Diffs:** Compare file's current version and SCCS version (like **sccs diff**).
    - **Edit:** Get a file to edit (**get -e**).
    - **Print:** Print information (like **prs -e** followed by **get -p –m**)

iLEARN
iGATE Learning, Education And Research Node

- 16 -

9.6 Example

Example

iGATE
iTOPS for Business Outcomes

- **You create program.c file and want to maintain its version.**
  - Initialize the history file for a source file named program.c.
    - Make the SCCS subdirectory, and then use `sccs create‘
    - Example:

```
S mkdir SCCS
    $ sccs create program.c
     program.c: 1.1
    14 lines
```

iLEARN
iGATE Learning, Education And Research Node

- 17 -

9.6 Example

Example

iGATE
iTOPS for Business Outcomes

- **Edit file:**
  - First *checkout* the file and then *checkin* the file.
  - Version 1.2 for program.c is created.
  - Steps:
    - 1. Checkout the file.

```
$ sccs edit program.c
      1.1 new delta 1.2
      14 lines
```

    - 2. Edit file with **vi editor**: Insert line at third position. Delete second line. Checkin file using the following command.

```
$ sccs delget program.c comments? clarified cryptic diagnostic 1.2
                    3 inserted
                    2 deleted
                    12 unchanged
                   1.2 15 lines
```

iLEARN
iGATE Learning, Education And Research Node

- 18 -

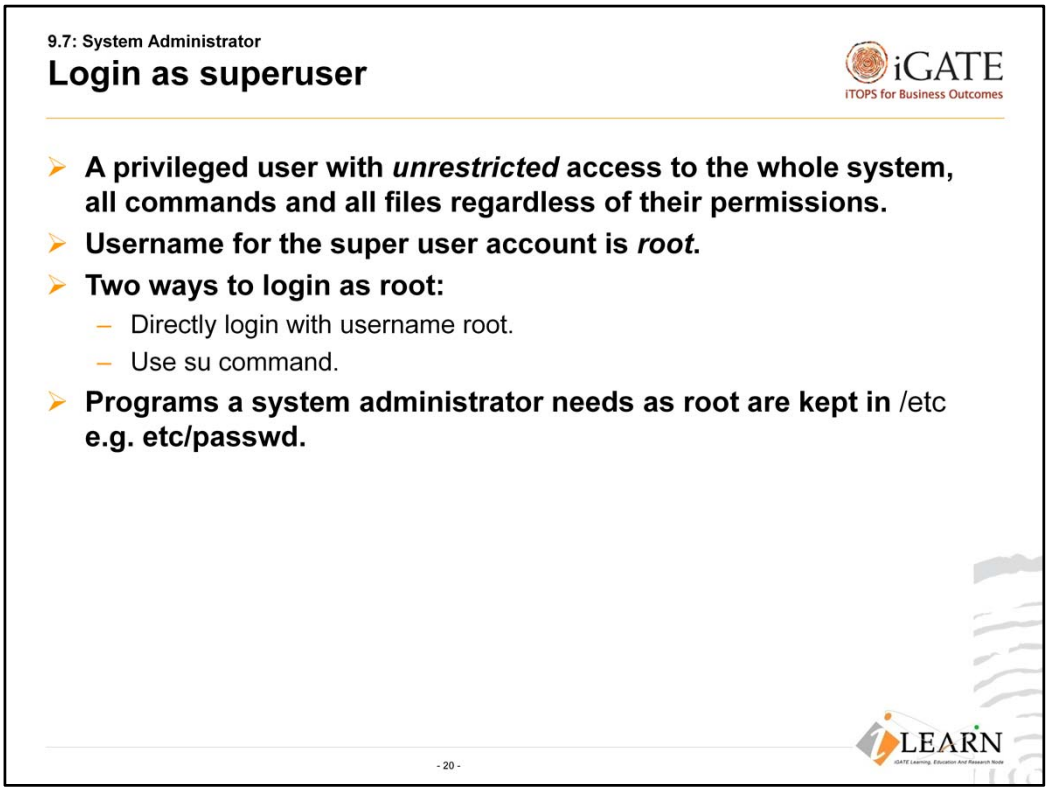
9.7: System Administration

Definition

iGATE
iTOPS for Business Outcomes

- **Task to maintain the system.**
- **Task performed by Administrator:**
  - Maintain files.
  - Maintain user login ids.
  - Maintain system log, hardware, software and network administration.
  - Configure the kernel.
  - Install and patch the UNIX operating system.

- 19 -



System Administrator
The superuser is a privileged user who has unrestricted access to the whole system; all commands and all files regardless of their permissions. By convention the username for the superuser account is root.

The root account is necessary as many system administration files and programs need to be kept separate from the executables available to non-privileged users. Unix allow users to set permissions on the files they own. A system administrator may need to override those permissions.

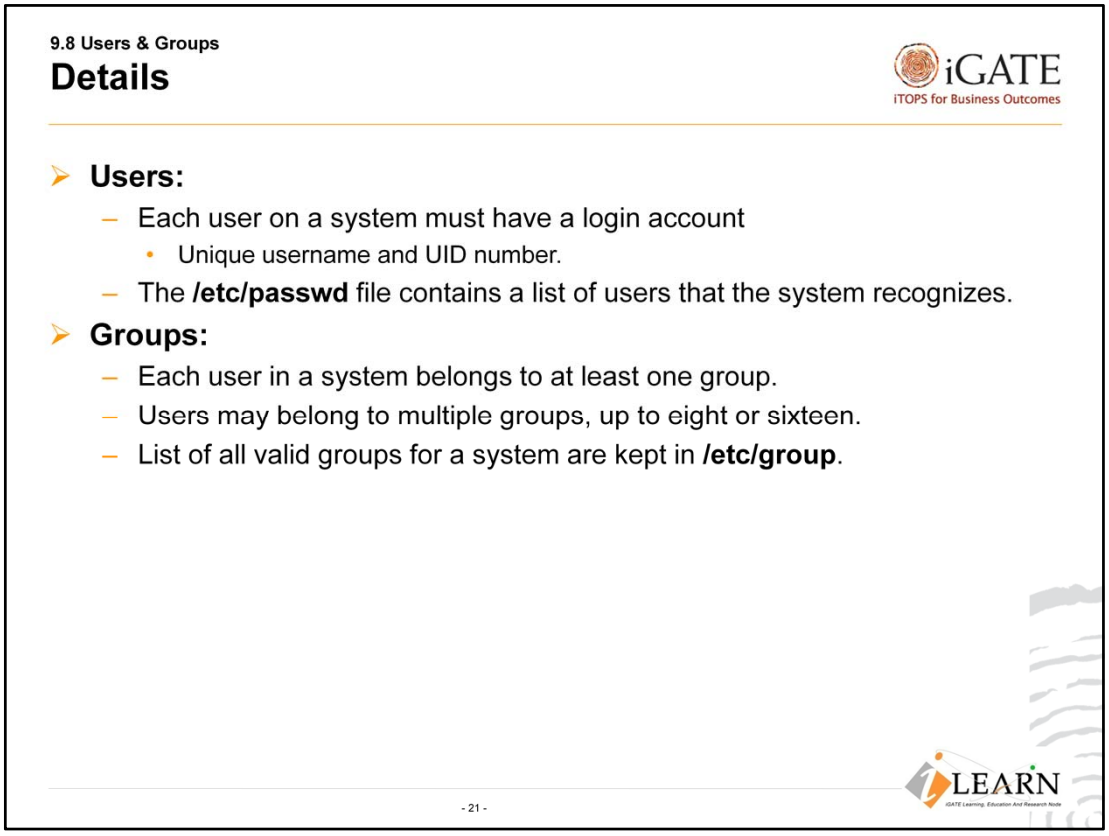
Because the superuser has the potential to affect the security of the entire system, it is recommended that this password be given only to people who absolutely need it, such as the system administrator. It is also a good idea to change the password on this account often.

There are several ways to log into the root account. If a system comes up in single user mode, whoever is logged in automatically has root privileges. When a system is already up in multi user mode, a user can log in directly as root. When a user is already logged in, issuing the su command, without options, will cause the system to prompt for the root password. Once it is given the user becomes root.

The root account has its own shell and frequently displays a prompt that is different from the normal user prompt. Commands and programs that a system administrator will need as root are kept in /etc to decrease the chances of a user trying them by accident. For example, /etc contains /etc/passwd, which holds a list of all users who have permission to use the system.

System Administrator (cont'd)
Because the root account has extensive privileges it has an equal potential for destruction. For example, the superuser may change another user's password without knowing the old password. The superuser can also mount and unmount file systems, remove any file or directory, and shut down the entire system. The root account should be used with caution and only when necessary to perform a given task. A misplaced keystroke in this mode can have disastrous results.
Various Unix systems will have different utilities for system administration.

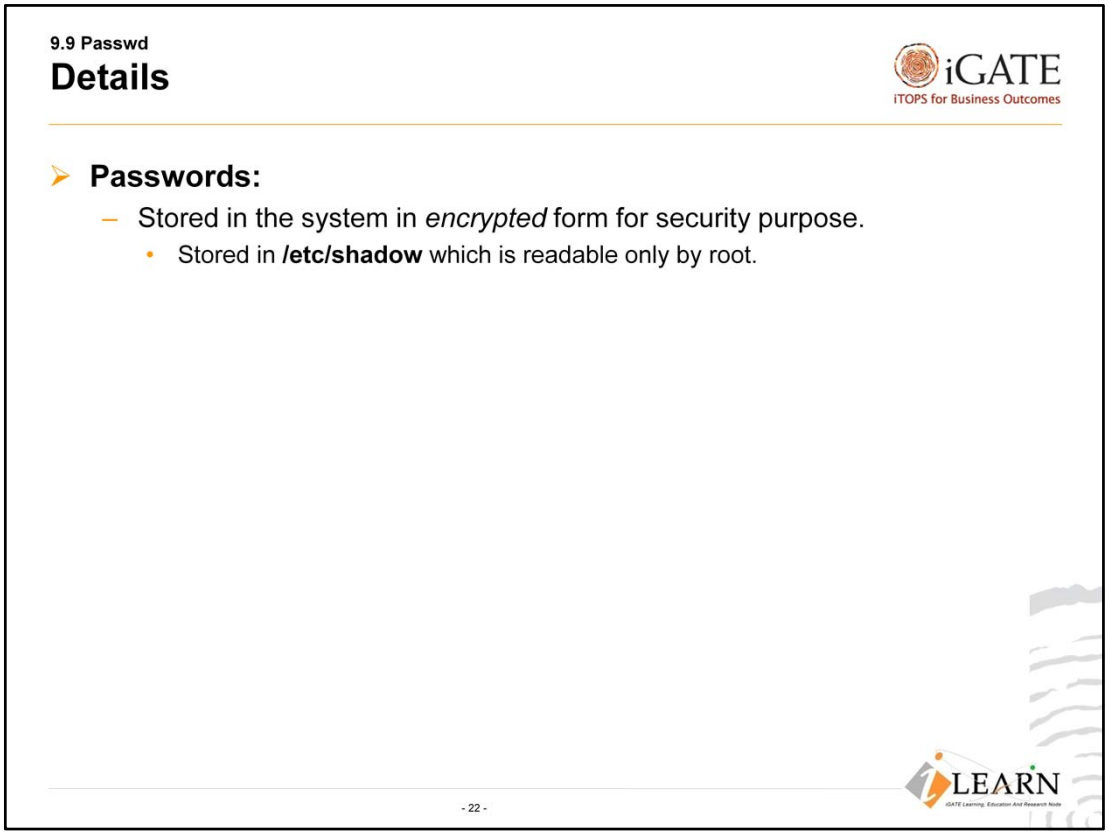
Super user can log in with the su command and can run utilities for system administration for creating, deleting and modifying groups and accounts.



Users
Each user on a system must have a login account identified by a unique username and UID number.
The /etc/passwd file contains a list of users that the system recognizes.

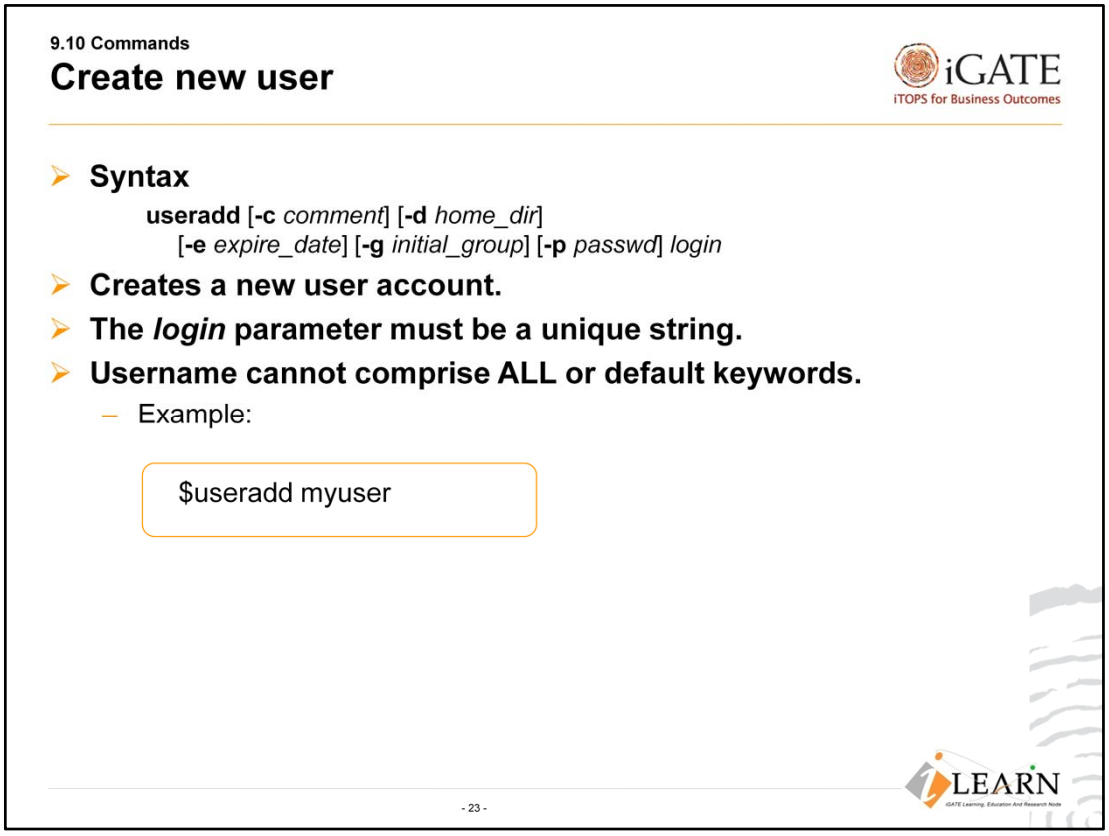
Groups
Each user in a system belongs to at least one group. Users may belong to multiple groups, up to a limit of eight or 16. A list of all valid groups for a system are kept in /etc/group.
This file contains entries such as:
work:*:15:trsmith,pmayfiel,arushkin

Each entry consists of four fields separated by a colon. The first field holds the name of the group. The second field contains the encrypted group password and is frequently not used. The third field contains the GID (group ID) number. The fourth field holds a list of the usernames of group members separated by commas.



Security
Passwords
Unix deals with passwords by not storing the actual password anywhere on the system. When a password is set, what's stored is a value generated by taking the password that was typed in and using it to encrypt a block of zeros. When a user logs in, /bin/login takes the password the user types in and uses it to encrypt another block of zeros. This is then compared with the stored block of zeros. If the two match the user is permitted to log in.

Decryption of passwords is possible - this means that even encrypted passwords are not secure if kept in a world-readable file like /etc/passwd. /etc/passwd needs to be world-readable for a number of reasons, including the user's ability to change their own passwords. So passwords are stored in /etc/shadow which is readable only by root. Even this does not make passwords absolutely secure, it just makes them harder to get to.



The useradd command does not create password information for a user. It initializes the password field with an asterisk (*). Later, this field is set with the passwd
The useradd command always checks the target user registry to make sure the ID for the new account is unique to the target registry.

Different Options
-c *comment*
Supplies general information about the user specified by the *login* parameter. The *comment* parameter is a string with no embedded colon (:) characters and cannot end with the characters '#!'.
-d *dir*
Identifies the home directory of the user specified by the *login* parameter. The *dir* parameter is a full path name.
-e *expire*
Identifies the expiration date of the account. The *expire* parameter is a 10-character string in the *MMDDhhmmyy* form, where *MM* is the month, *DD* is the day, *hh* is the hour, *mm* is the minute, and *yy* is the last 2 digits of the years 1939 through 2038. All characters are numeric. If the *expire* parameter is 0, the account does not expire. The default is 0. See the **date** command for more information.

Useradd Command’s Option

-g *group*
Identifies the user's primary group. The *group* parameter must contain a valid group name and cannot be a null value.

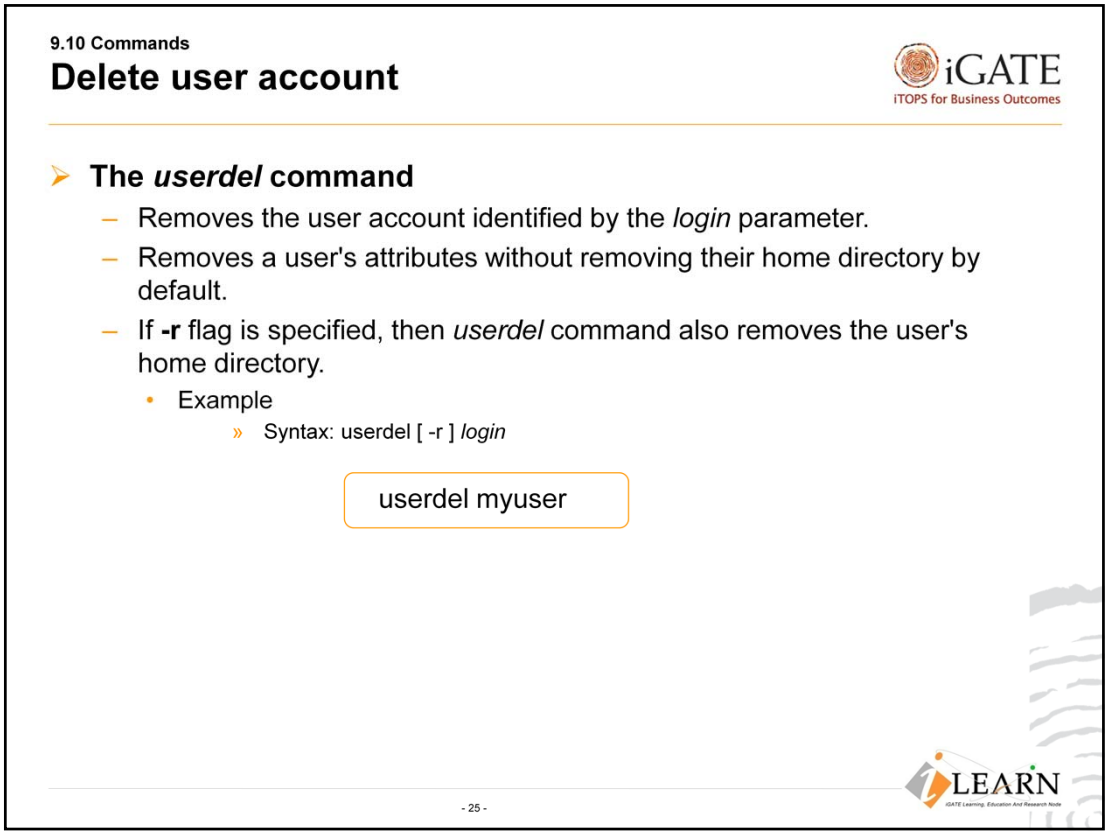
-G *group1,group2,...*
Identifies the groups the user belongs to. The *group1,group2,...* parameter is a comma-separated list of group names.
with **-m** flag.

-m
Makes user's home directory if it does not exist. The default is not to make the home directory.

-r *role1,role2,...*
Lists the administrative roles for this user. The *role1,role2,...* parameter is a list of role names, separated by commas.

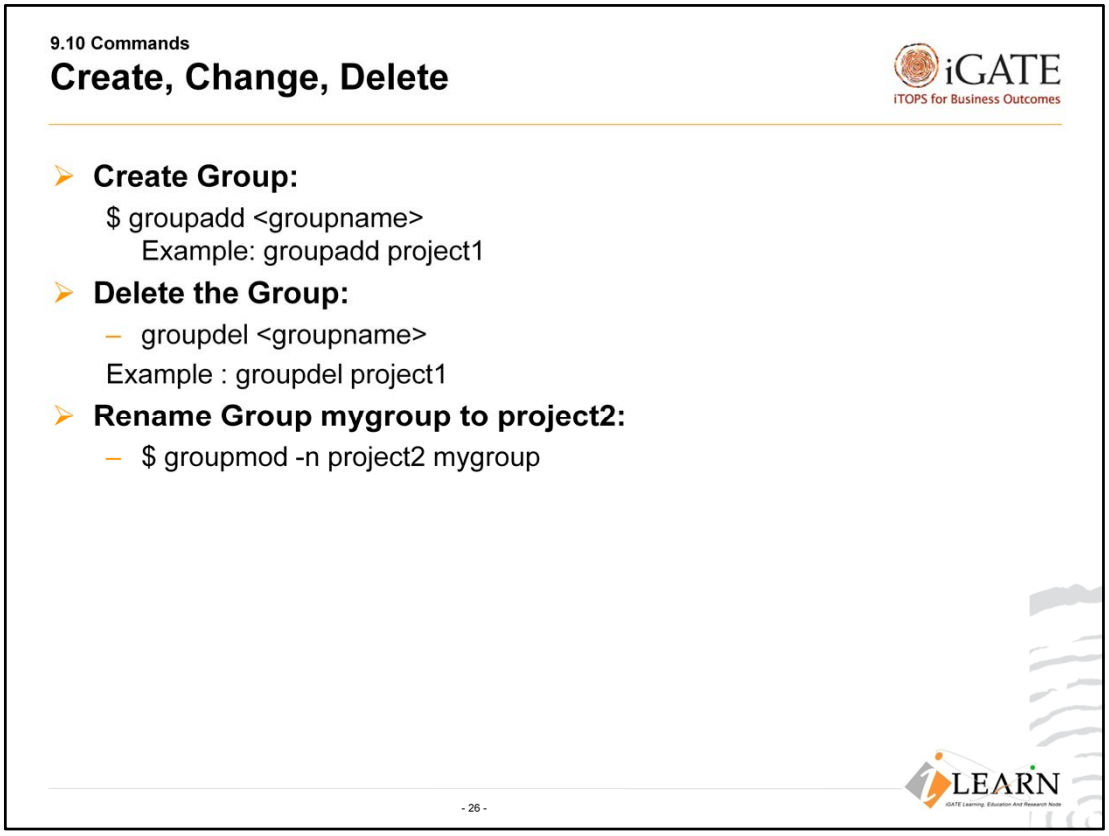
-s *shell*
Defines the program run for the user at session initiation. The *shell* parameter is a full path name.

-u *uid*
Specifies the user ID. The *uid* parameter is a unique integer string. Avoid changing this attribute so that system security will not be compromised.



Userdel command

The userdel command removes the user account identified by the *login* parameter. The command removes a user's attributes without removing the user's home directory by default. The user name must already exist. If the -r flag is specified, the userdel command also removes the user's home directory.



If the group is not assigned during creation of the user account, the administrator can run this command to assign the new group to the new user account:
usermod -g <groupname> <username>

9.10 Commands

Change Passwd

iGATE
iTOPS for Business Outcomes

- **Change your own password.**

```
$passwd
```

- **Change password for user *myuser*.**

```
$passwd myuser
```

- **Delete passwd.**

```
$passwd -d myuser
```

iLEARN
iGATE Learning, Education And Research Node

- 27 -

Summary

iGATE
iTOPS for Business Outcomes

- CC
- MAKE
- SCCS
- Introduction to system administrator

iLEARN
iGATE Learning, Education And Research Node

- 28 -

Review Questions

iGATE
iTOPS for Business Outcomes

- _____ is the UNIX system compiler for C.
- Modifying any one or more of the source files requires recompilation of that part of the program and relinking is possible because of make utility.
  - True
  - False
- In _____ multiple copies are not stored only changes are stored
- List any 3 commands used in SCCS?

iLEARN

- 29 -

Reference Books

iGATE
iTOPS for Business Outcomes

- **The Unix Programming Environment**
  - Kerningham & Pike, Prentice Hall
- **Unix System V.4 Concepts and Applications**
  - Sumitabha Das, Tata McGraw-Hill
- **Advanced Unix Programmer's Guide**
  - Stephen Prata, BPB
- **Introducing Unix System V**
  - Vijay Mukhi, Tata McGrawHill

iLEARN
iGATE Learning, Education And Research Node

- 30 -

iGATE
iTOPS for Business Outcomes

iLEARN
iGATE Learning, Education And Research Node

UNIX
(Version: 1.3)
Lab Book

Document Revision History

Date	Revision No.	Author	Summary of Changes
	1	Veena Deshpande	New course creation
30-Sept-2009	2	Kishori Khadilkar	Revamped as per new template
20-June-2011	3	Rathnajothi Perumalsamy	Revamped as per Integrated syllabus

Table of Contents

Getting Started

Overview

This lab book is a guided tour for learning Unix. It comprises 'To Do' assignments. Follow the steps provided and work out the 'To Do' assignments.

Setup Checklist

Here is what is expected on your machine in order for the lab to work

Minimum System Requirements

- Intel Pentium 90 or higher (P166 recommended)
- Microsoft Windows 95, 98, or NT 4.0, 2k, XP.
- Memory: 32MB of RAM (64MB or more recommended)

Please ensure that the following is done:

- A text editor like Notepad is installed.
- Participants should be able to connect to UNIX server through telnet (IP address : 192.168.224.34)

Instructions

- For all coding standards refer Appendix A. All lab assignments should refer coding standards.
- Create a directory by your name in drive <drive>. In this directory, create a subdirectory html_assgn. For each lab exercise create a directory as lab <lab number>

Learning More (Bibliography if applicable)

- UNIX Concepts and Application by Sumitabha Das
- The Unix Programming Environment", by Kernighan and Pike.
- UNIX Primer Plus, Third Edition. Don Martin, Stephen Prata, Mitchell Waite, Michael Wessler, and Dan Wilson
- Advanced Unix : a programmer's guide / Stephen Prata

Lab 1. Connecting to the Unix Server

Goals	• Learn to connect to the Unix server • Learn to log out of the Unix server
Time	5 min

1.1: Connecting to the Unix Server

Step 1: Enter your login name and password to login to the UNIX system.

1.2: Logging out of the system

Step 1: Type the exit command at $ prompt or else, press ctrl and d together to log out.

Lab 2. Unix Basic Command

Goals	• Learn to use basic Unix commands
Time	120 min
Lab Setup	Telnet with Unix Server

2:1 Single Row Functions:

1. To display the current working directory, the command is:
   pwd
   The output is as follows.
   /home/trg1
2. Display the path to and name of your HOME directory.
3. Display the login name using which you have logged into the system
4. Display the hidden files of your current directory.
5. List the names of all the files in your home directory.
6. Using the long listing format to display the files in your directory.
7. List the files beginning with chap followed by any number or any lower case alphabet. (Example, it should display all files whose names are like chap1, chap2, chap3 ……., chapa,ahapb,chapc,……..)
8. Give appropriate command to create a directory called C_prog under your home directory. (Note: Check the directory using ls)
9. Create the following directories under your home directory. (Note: Check using ls)

   newdir

   newdirectory
10. List the names of all the files, including the contents of the sub directories under your home directory.
11. Remove the directory called newdirectory from your working directory.
12. Create a directory called temp under your home directory.
13. Remove the directory called newdir under your home directory and verify the above with the help of the directory listing command.
14. Create another directory directorynew under the temp directory.
15. Change the directory to your home directory.
16. From your home directory, change the directory to directorynew using relative and absolute path.
17. Remove the directory called c_prog, which is in your home directory.
18. Change to the directory /etc and display the files present in it.
19. List the names of all the files that begin with a dot in the /usr/bin directory.

20. Create a file first.unix with the following contents.

    Hi! Good Morning everybody.

    Welcome to the First exercise on UNIX.

    Hope you enjoy doing the assignments.

21. Copy the file first.unix in your home directory to first.unics.

    (Note: checked using ls, first.unix file also should exist along with first.unics)

22. List the contents of first.unix and first.unics with a single command.
23. Create a new directory under the temp directory.
24. From your home directory, copy all the files to the directory created under the temp sub directory.
25. Move the file first.unix to the directory temp as second.unix
26. Remove the file called first.unics from the home directory.
27. Change your directory to temp and issue the command rm *. What do you observe?
28. Move all files whose names end with a, c and o to the HOME directory.
29. Copy all files that end with a 'UNIX' to the temp directory.
30. Issuing a single command, remove all the files from the directory temp and the directory itself.
31. Try commands cp and mv with invalid number of arguments and note the results.
32. Use the cat command to create a file friends, with the following data:

Madhu	6966456	09/07/68
Jamil	2345215	08/09/67
Ajay	5546785	01/04/66
Mano	7820022	09/07/68
David	8281292	09/09/60
Simmi	7864563	12/12/70
Navin	2224311	30/05/68

    The fields should be separated by a tab.

33. Display contents of the file friends.
34. Copy contents of friends to newfriend without using the cp command.
35. Display contents of the file friends and newfriends in a single command.
36. Find all users currently working on the system and store the output in a file named as users.
37. Append contents of friends file to the file, users.
38. Display current system date and time and record your observations. How is the time displayed?
39. Display calendar for the month and year of your birth.
40. Try following commands and record your observations.

    date "+ %"

date “+%m”

date “+%D”

date “+%/%Training Activity”

date “+%Training Activity”

date “+%r”

Lab 3. UNIX File System & Permissions

Goals	• Learn to grant and to remove permissions and to view the file system
Time	15 min
Lab Setup	Telnet with Unix Server

3.1: Viewing the File System and Granting/Removing Permissions

(Note: Create required files if doesn't exists**.)**

1. Give the execute permission for the user for a file chap1
2. Give the execute permission for user, group and others for a file add.c
3. Remove the execute permission from user, give read permission to group and others for a file aa.c
4. Give execute permission for users for a.c, kk.c, nato and myfile using single command
5. Change the directory to root directory. Check the system directories, like bin, etc, usr etc

Lab 4. JOINS AND SUBQUERIES

Goals	• Learn to use Pipes & Filters in UNIX
Time	120 min
Lab Setup	Telnet with Unix Server

4.1: Using Pipes and Filters:

1: Redirect the content of the help document ls, into a file called as lsdoc.

2: Display the content of the lsdoc page wise.

3: Display only the first 4 lines of the lsdoc file.

4: Display only the last 7 lines of the file lsdoc.

5: Remove the file lsdoc.

6: There will be B'day celebration from the friends file, find how many B'day parties will be held. If two of the friends have the B'date on the same day, then we will be having one party on that day.

7: Display the lines starting with Ma, in the file friends.

8: Display the lines starting with Ma, ending with i or ending with id, in the file friends.

9: Print all the files and the directory files from the current directory across all the sub directories, along with its path

10: Print only the Directory files.

11: Display the files starting with chap, along with its path.

12: Sort the file friends in ascending order of names.

13: Display the contents of the file friends in uppercase letters.

14: Store the contents of your home directory in a file called dir.

15: From the above file dir, display the file permissions and the name of the file only.

16: From the same dir file, store only the file names in a file called files.

17: From the same dir file, store only the permissions of files in a file called perms.

18: From the same dir file, store only the file sizes in a file called sizes.

19: Display the file names, sizes and permissions from your directory in that order.

20: Display the number of users working on the system.

21: Find out the smallest file in your directory.

22: Display the total number of lines present in the file friends.

23:Create the following fixed record format files (with "|" delimiter between fields) with the structure given below, and populate them with relevant data use these files to solve following questions

emp.lst: Empid(4),Name(18),Designation(9),Dept(10),Date of Birth(8),Salary(5)

dept.lst : Dept.Code(2),Name(10),Head of Dept's id(4)

desig.lst: Designation Abbr.(2), Name (9)

1. Find the record lengths of each file.
2. Display only the date of birth and salary of the last employee record.
3. Extract only employee names and designations. (Use column specifications). Save output as cfile1.
4. Extract Emp.id, dept, dob and salary. (Use field specifications). Save output as cfile2.
5. Fix the files cfile1 and cfile2 laterally, along with the delimiter.
6. Sort the emp.lst file in reverse order of Emp. Names.
7. Sort the emp.lst file on the salary field, and store the result in file srtf.
8. Sort the emp.lst file on designation followed by name.
9. Sort the emp.lst file on the year of birth.
10. Find out the various designations in the employee file. Eliminate duplicate listing of designations.
11. Find the non-repeated designation in the employee file.
12. Find the number of employees with various designations in the employee file.
13. Create a listing of the years in which employees were born in, along with number of employees born in that year.
14. Use nl command to create a code table for designations to include designation code (Start with dept. code 100, and subsequently 105, 110 …).

24: PCS has its offices at Pune, TTC and Mumbai. The employees' data is stored separately for each office. Create appropriate files (with same record structure as in previous assignment) and populate with relevant data.

1. List details about an employee 'Manu Sharma' in the Mumbai office.

2. List only the Emp.Id. And Dept. of Manu Sharma.
3. List details of all managers in all offices. (O/P should not contain file names.).
4. Find the number of S.E. in each office.
5. List only the Line Numbers and Employee names of employees in 'H/W' in Pune file.
6. Obtain a listing of all employees other than those in 'HR' in the Mumbai file and save contents in a file 'nonhr'.
7. Find the name and designation of the youngest person who is not a manager.
8. Display only the filename(s) in which details of employee by the name 'Seema Sharma' can be found.
9. Locate the lines containing saxena and saksena in the Mumbai office.
10. Find the number of managers who earn between 50000 and 99999 in the Pune office.
11. List names of employees whose id is in the range 2000 – 2999: in Pune Office; in all offices.
12. Locate people having same month of birth as current month in Pune office.
13. List details of all employees other than those of HR and Admin in file F1.
14. Locate for all Dwivedi, Trivedi, Chaturvedi in Pune file.
15. Obtain a list of people in HR, Admin and Recr. depts. sorted in reverse order of the dept.

Stretched assignments:

25: Write a command sequence that prints out date information in this order: time, day of week, day number, month, year:
13:44:42 IST Sun 16 Sept 1994

26: Write a command sequence that prints the names of the files in the current directory in the descending order of number of links

27: Write a command sequence that prints only names of files in current working directory in alphabetical order

28: Write a command sequence to print names and sizes of all the files in current working directory in order of size

29: Determine the latest file updated by the user

Lab 5. Vi Editor

Goals	Work with Vi Editor in Unix
Time	30 min
Lab Setup	Telnet with Unix Server

5.1: Working wth Vi Editor

1. Create a file using Vi. Enter the following text:
   A network is a group of computers that can communicate with each other, share resources, and access remote hosts or other networks. Netware is a computer network operating system designed to connect, manage, and maintain a network and its services. Some of the network services are Netware Directory Services (NDS), file system, printing and security.

   a. Change the word “Netware” in the second line to “Novell Netware”.
   b. Insert the text “(such as hard disks and printers)” after “share resources” in the first line.
   c. Append the following text to the file:
   “Managing NDS is a fundamental administrator role because NDS provides a single point for accessing and managing most network resources.”

2: Create the data files, used in the previous lab sessions using vi editor.

Lab 6. SED Commands

Goals	Learn to use SED Commands in Unix
Time	15 min
Lab Setup	Telnet with Unix Server

6.1: Using SED Commands

1. Create a file “Employee.dat” with text as follows.

James	76382	PACE	Chennai
John	34228	GRIT	Hyderabad
Peter	22321	GE	Bangalore
Albert	32342	GRIT	Pune
Mathew	23222	PACE	Mumbai
Richard	23232	ACS	Pune

a) Write a sed command to print only the lines starting at line 2 and ending with the letters “Pune”

b) Write a sed command that will display the top 5 lines from the file

c) Write a sed command that will substitute the word “Chennai” for "Pune" used in all instance of the word

d) Write a sed command that will replace occurrence of the character e with the string UNIX in all lines. (Use –e option)

e) Write a sed command to delete blank lines

f) Write a sed command to delete lines from 3 to 5

2**: Create a new file “PACE.dat which has only the lines that contain the word “PACE” from Employee.dat**

Lab 7. Process Related Commands

Goals	Learn to use process-related commands in Unix
Time	15 min
Lab Setup	Telnet with Unix Server

7.1: Using Process-Related Commands

1. Find out the PID of the processes that are activated by you
2. Find out the information about all the processes that are currently active
3. Start a different process in the background. Find out the status of the background process using the PID of the same.

Lab 8. Shell Script

Goals	Learn to write simple shell scripts
Time	3 Hrs
Lab Setup	Telnet with Unix Server

7.1: Writing Shell-Scripts

1. Display the Primary and Secondary prompt. Change the primary prompt to your name: temporarily

2: As soon as you login, the prompt should be changed to your name: also the name of the home directory should be automatically displayed.

3: Check the content of the Environmental variable SHELL.

4: Try the below exercise and check the output.

Note: Type every line and press enter, do not type the entire code in a vi editor.

```
$continent=”Africa”
$echo “$continent”
                    ----------→ Africa
$sh
$echo “$continent”
                    ----------→ No Response
$continent=”Asia”
$echo “$continent”
                    ----------→ Asia
$ctrl + d
$echo “$continent”
                    ----------→ Africa
$sh
$echo “$continent”
                    ----------→ No Response
$ctrl + d
```

5: Try the below exercise and check the output. (Export variables)

Note: Type every line and press enter, do not type the entire code in a vi editor.

```
$continent=”Africa”
export continent
$echo “$continent”
                    ----------→ Africa
```

```
$sh
$echo “$continent”
                    ----------→ Africa
$continent=”Asia”
$echo “$continent”
                    ----------→ Asia
$ctrl + d
$echo “$continent”
                    ----------→ Africa
```

6: Write a shell script that takes the user name as input and reports whether he / she has logged in or not.

7: Write a shell script to display the file name and its contents of all the files that is there in the current directory.

8: Write a shell script, which will take a file name as argument and check whether the file exists and display its access permissions for user.

9: Pass three numbers as command line arguments and display the largest number in the given three numbers.

10: Write a shell script which will accept a pattern and a file name. The pattern will be searched in the file provided. Display appropriate messages and perform necessary validations on file.

11: To create a menu program for a) creating a file, b) Creating a directory, c) copying a file, d) moving a file. (use functions)

- **a.** If the file exists already give the appropriate message
- **b.** If the dir exists already give the appropriate error message
- **c.** Source file should exist if not give a message, It should have read permission if not another message, Destination file either there or not, if not there then create it and copy it. If there, then ask whether to overwrite or not, if yes then overwrite it or else give a message file exists already and not overwritten.

12: Write a function yesno() to display question to user and accept answer as y/n. If answer to the question is y the function should return 0 otherwise 1.

Use yesno functions for asking different questions. Question will be passed as parameter to the function.

Accept filename from user check whether it is file or directory. Use yesno() function to display question do you really want to delete file? If the ans is y, then delete the file or directory.

13: Write a shell script to store names of four employees and check whether those employees are currently logged in or not. Display appropriate message.

14: Accept the user's first and last name and the echo the entire name along with some suitable comment.

15: List all files that have been modified today.

16: Display long listing of only the regular files in the current directory.

17: Display details of all files in the 2 “paths” accepted from user. The display should be screen by screen.

18: Let the script display its name and its PID.

19: Get the concatenated o/p of 2 files into a third file: Take 3 command line arguments: The first argument is the name of a destination file, and the other two arguments are names of files whose contents are to be placed in the destination file.

Stretched Assignments:

20: Write a menu driven shell program to:
a. Display calendar of current month
b. Search for a pattern in all the files/subdirectories from current directory.
c. Count the no. of directories / sub directories in current directory

21: Display day of week for a given date. (ddmmyyyy)
If day is Monday, display message “Monday Blues”
Friday display message “yeh! It’s week end.”
Similarly display different messages for each day of the week.

22: Display the contents of all .lst files in the current directory.

23: Design a simple calculator, which will add/subtract/multiply/divide 2 numbers.
eg. cal 10 20 + will give o/p as 30.

24: For a student file with the following fields, rollno, name, marks, Generate 2 files ‘Pass’ and ‘Fail’ containing records of student who have passed or failed. Also count the number of students who have passed or failed.

25: Accept a date string from terminal and display employees born after the input date.

Lab 9. Writing and Executing C and C++ Programs

Goals	• Learn to use process-related commands in Unix Learn how to write, compile, link and execute simple C and C++ programs • Understand the steps involved from writing program to executing it
Time	60 min
Lab Setup	Linux Operating System and gcc and g++ compilers

9.1: Write, Compile, Link and Execute a Simple C and C++

Program:

Step 1: Login to the **Linux Operating** System.

Login to the home directory. Use the login details provided by faculty to login in the system

Step 2: Use vi editor to write following C programs:

pi.c

```
#include <math.h>

mypi()
{
printf("pi = %.5f\n", 4 * atan(1.0));
}
```

hello.c

```
#include <stdio.h>
main ()
{
mypi();
}
```

Step 3: Compile the source files.

Use the gcc command to compile pi.c and hello.c source files to create pi.o and hello.o object files.

```
$ gcc -c pi.c
$ gcc -c hello.c
```

Step 4: Link the object files.

Execute the gcc command with –lm –o command option to link the object files to create executable file hello (-lm option will include library maths).

```
$ gcc -lm -o hello hello.o pi.o
```

Step 5: Execute the hello file.

Execute the hello program to see the output by giving following command (./ indicate search hello file in current directory otherwise the file will be searched in bin directory)

```
$  ./hello
pi = 3.14159
```

Stretched assignments

Write AWK scripts for the following:

(Use emp.lst file created in Lab 4)

1: Find the number of employees belonging to a particular department specified by user

2: Find the count of people in each dept. of the employee file

3: Generate a list of all S.E. who earn more than the amount specified by user

4: View the employee records in order of designations

5: List employee details of all employees who earn more than the average salary of all employees.

Lab 10. Understanding Makefile and Make Utility

Goals	• Learn how to create a makefile to automate the build process • Understand the steps involved in creating and executing Makefile using make utility
Time	60 min
Lab Setup	Linux Operating System and gcc and g++ compilers

10.1: Create and Execute a Makefile:

1. Create a make file.
2. Using vi editor create Makefile as shown below:

```
$ cat Makefile
pi.o:   pi.c
        gcc -c pi.c

hello.o: hello.c
        gcc -c hello.c

hello:  hello.o pi.o
         gcc -lm -o hello hello.o pi.o
```

Note: You need TABS after each ":" and before the gcc on the lines that follow each "rule". Basically, a makefile says "to make hello file, you need hello.o and pi.o files, and you run this command"

Step 2: Execute Makefile.

3. To execute Makefile use the make command as shown below (in the command hello indicates the target):

```
$ make hello
gcc -c hello.c
gcc -c pi.c
gcc -lm -o hello hello.o pi.o
$ ./hello
pi = 3.14159
$
```

Note: Leave out tabs in Makefile, make will complain:

Makefile:5: * missing separator. Stop**

10.2: <TODO> Compiling and executing c++ programs

Write picpp.cpp and hellocpp.cpp c++ programs to do same task as that of pi.c and hello.c respectively.

Write Makefile1 to build hellocpp executable file and execute hellocpp executable program.

Note: You need to use g++ compiler to compile the C++ program. Also explore how to invoke Makefile1 from make utility.