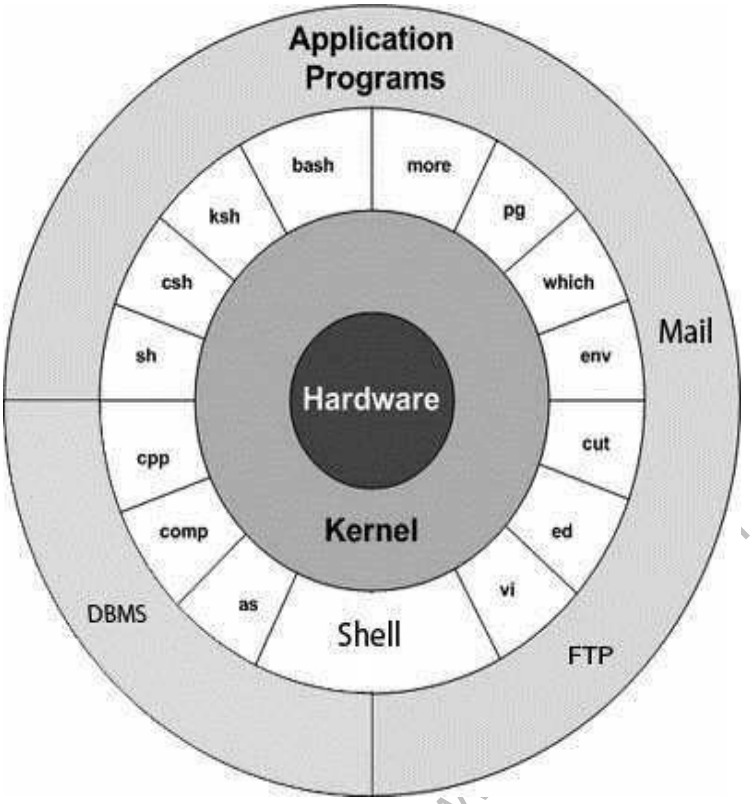
The computer programs that allocate the system resources and coordinate all the details of the computer's internals is called the operating system or kernel.

Users communicate with the kernel through a program known as the shell. The shell is a command line interpreter; it translates commands entered by the user and converts them into a language that is understood by the kernel.

- Unix was originally developed in 1969 by a group of AT&T employees at Bell Labs, including Ken Thompson, Dennis Ritchie, Douglas McIlroy, and Joe Ossanna.
- There are various Unix variants available in the market. Solaris Unix, AIX, HP Unix and BSD are few examples. Linux is also a flavor of Unix which is freely available.
- Several people can use a UNIX computer at the same time; hence UNIX is called a multiuser system.
- A user can also run multiple programs at the same time; hence UNIX is called multitasking

Unix Architecture:

Here is a basic block diagram of a UNIX system:



- **Shell:** The shell is the utility that processes your requests. When you type in a command at your terminal, the shell interprets the command and calls the program that you want. The shell uses standard syntax for all commands. C Shell, Bourne Shell and Korn Shell are most famous shells which are available with most of the Unix variants.
- **Commands and Utilities:** There are various command and utilities which you would use in your day to day activities. **cp, mv, cat** and **grep** etc. are few examples of commands and utilities. There are over 250 standard commands plus numerous others provided through 3rd party software. All the commands come along with various optional options.
- **Files and Directories:** All data in UNIX is organized into files. All files are organized into directories. These directories are organized into a tree-like structure called the filesystem

Accessing Unix:

- When you first connect to a UNIX system, you usually see a prompt such as the following

To log in:

1. Have your userid (user identification) and password ready. Contact your system administrator if you don't have these yet.
2. Type your userid at the login prompt, then press ENTER. Your userid is case-sensitive, so be sure you type it exactly as your system administrator instructed.
3. Type your password at the password prompt, then press ENTER. Your password is also case-sensitive.
4. If you provided correct userid and password then you would be allowed to enter into the system. Read the informatand messages that come up on the screen something as below.

```
login : amrood
amrood's password:
Last login: Sun Jun 14 09:32:32 2009 from 62.61.164.73
```

You would be provided with a command prompt (sometime called $ prompt) where you would type your all the commands. For example to check calendar you need to type **cal** command as follows:

```
$ cal
      June 2009
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
$
```

2. Enter your old password the one you're currently using.
3. Type in your new password. Always keep your password complex enough so that no body can guess it. But make sure, you remember it.
4. You would need to verify the password by typing it again.

$ passwd
Changing password for amrood
(current) Unix password:******
New UNIX password:*******
Retype new UNIX password:*******
passwd: all authentication tokens updated successfully
$

Listing Directories and Files:

All data in UNIX is organized into files. All files are organized into directories. These directories are organized into a tree-like structure called the filesystem.

You can use **ls** command to list out all the files or directories available in a directory. Following is the example of using **ls** command with **-l** option.

$ ls -l
total 19621
drwxrwxr-x 2 amrood amrood 4096 Dec 25 09:59 uml
-rw-rw-r— 1 amrood amrood 5341 Dec 25 08:38 uml.jpg
drwxr-xr-x 2 amrood amrood 4096 Feb 15 2006 univ
drwxr-xr-x 2 root root 4096 Dec 9 2007 urlspedia
-rw-r--r-- 1 root root 276480 Dec 9 2007 urlspedia.tar
drwxr-xr-x 8 root root 4096 Nov 25 2007 usr
-rwxr-xr-x 1 root root 3192 Nov 25 2007 webthumb.php
-rw-rw-r-- 1 amrood amrood 20480 Nov 25 2007 webthumb.tar

Who Are You

While you're logged in to the system, you might be willing to know : **Who am I**?
The easiest way to find out "who you are" is to enter the **whoami** command:

$ whoami
amrood
$ who
amrood ttyp0 Oct 8 14:10 (limbo)
bablu ttyp2 Oct 4 09:08 (calliope)
qadir ttyp4 Oct 8 12:09 (dent)

Just type **logout** command at command prompt, and the system will clean up everything and break the connection.

Listing Files:

To list the files and directories stored in the current directory. Use the following command:

$ls

```
bin hosts lib  res.03
ch07  hw1  pub  test_results
ch07 .bak hw2 res.01 users
docs hw3 res.02 work
```

The command **ls** supports the **-1** option which would help you to get more information about the listed files:

$ls -l

```
total 1962188
drwxrwxr-x 2 amrood amrood 4096 Dec 25 09:59 uml
-rw-rw-r-- 1 amrood amrood 5341 Dec 25 08:38 uml.jpg
drwxr-xr-x 2 amrood amrood 4096 Feb 15 2006 univ
drwxr-xr-x 2 root root 4096 Dec 9 2007 urlspedia
-rw-r--r-- 1 root root 276480 Dec 9 2007 urlspedia.tar
drwxr-xr-x 8 root root 4096 Nov 25 2007 usr
drwxr-xr-x   2 200 300 4096 Nov 25 2007 webthumb-1.01
-rwxr-xr-x    1 root root 3192 Nov 25 2007 webthumb.php
-rw-rw-r-- 1 amrood amrood 20480 Nov 25 2007 webthumb.tar
-rw-rw-r-- 1 amrood amrood 5654 Aug 9 2007 yourfile.mid
-rw-rw-r-- 1 amrood amrood 166255 Aug 9 2007 yourfile.swf
drwxr-xr-x 11 amrood amrood 4096 May 29 2007 zlib-1.2.3
```

Hidden Files:

An invisible file is one whose first character is the dot or period character (.). UNIX programs (including the shell) use most of these files to store configuration information.

Some common examples of hidden files include the files:

- **.profile:** the Bourne shell (sh) initialization script
- **.kshrc:** the Korn shell (ksh) initialization script
- **.cshrc:** the C shell (csh) initialization script
- **.rhosts:** the remote shell configuration file

To list invisible files, specify the -a option to ls:

$ ls -a

```
. .profile docs lib test_ results
.. .rhosts   hosts pub users
   .emacs   bin   hw1 res.01 work
   .exrc      ch07 hw2 res.02
   .kshrc    ch07 .bak hw3 res.03
```

edit mode. Once you are in edit mode you can start writing your content in the file as below:
This is unix file....I created it for the first time.....
I'm going to save this content in this file.

Once you are done, do the following steps:

- Press key esc to come out of edit mode.
- Press two keys Shift + ZZ together to come out of the file completely

Now you would have a file created with filemame in the current directory

$ vi filename

Editing Files:

You can edit an existing file using **vi** editor. We would cover this in detail in a separate tutorial. But in short, you can open existing file as follows:

$ vi filename

Once file is opened, you can come in edit mode by pressing key i and then you can edit file as you like. If you want to move here and there inside a file then first you need to come out of edit mode by pressing key esc and then you can use following keys to move inside a file:

- l key to move to the right side.
- h key to move to the left side.
- k key to move up side in the file.
- j key to move down side in the file.

So using above keys you can position your cursor where ever you want to edit. Once you are positioned then you can use i key to come in edit mode. Edit the file, once you are done press esc and finally two keys Shift + ZZ together to come out of the file completely.

Display Content of a File:

You can use **cat** command to see the content of a file. Following is the simple example to see the content of above created file:

$ cat filename

This is unix file....I created it for the first time.....
I'm going to save this content in this file.

Counting Words in a File:

You can use the **wc** command to get a count of the total number of lines, words, and characters contained in a file. Following is the simple example to see the information about above created file:

$ wc filename
2 19 103 filename

Here is the detail of all the four columns:

1. First Column: represents total number of lines in the file.
2. Second Column: represents total number of words in the file.
3. Third Column: represents total number of bytes in the file. This is actual size of the file

$ cp source_file destination_file
Following is the example to create a copy of existing file **filename**.
$ cp filename copyfile
Now you would find one more file **copyfile** in your current directory. This file would be exactly same as original file **filename**.

Renaming Files:

To change the name of a file use the **mv** command. Its basic syntax is:
$ mv old_file new_file
Following is the example which would rename existing file **filename** to **newfile**:
$ mv filename newfile
The **mv** command would move existing file completely into new file. So in this case you would fine only **newfile** in your current directory

Deleting Files:

To delete an existing file use the **rm** command. Its basic syntax is:
$ rm filename
rm command.
Following is the example which would completely remove existing file **filename**:
$ rm filename
You can remove multiple files at a tile as follows:
$ rm filename1 filename2 filename3

Unix Directories:

A directory is a file whose sole job is to store file names and related information. All files whether ordinary, special, or directory, are contained in directories.
UNIX uses a hierarchical structure for organizing files and directories. This structure is often referred to as a directory tree . The tree has a single root node, the slash character (/), and all other directories are contained below it.

Home Directory:

The directory in which you find yourself when you first login is called your home directory.
You will be doing much of your work in your home directory and subdirectories that you'll be creating to organize your files.
You can go in your home directory anytime using the following command:
$cd ~
Here **~** indicates home directory. If you want to go in any other user's home directory then use the following command:
$cd ~username
To go in your last directory you can use following command:
$cd -

Absolute/Relative Pathnames:

Directories are arranged in a hierarchy with root (/) at the top. The position of any file within the hierarchy is described by its pathname.
Elements of a pathname are separated by a /. A pathname is absolute if it is described in relation to root, so absolute pathnames always begin with a /.

with /. Relative to user amrood' home directory, some pathnames might look like this:
chem/notes
personal/res
To determine where you are within the filesystem hierarchy at any time, enter the command **pwd** to print the current working directory:

```
$pwd
/user0/home/amrood
```

Listing Directories:

To list the files in a directory you can use the following syntax:

```
$ls dirname
```

Following is the example to list all the files contained in /usr/local directory:

```
$ls /usr/local
X11      bin      imp          jikes      sbin
ace      doc      include      lib        share
atalk    etc      info         man        ami
```

Creating Directories:

Directories are created by the following command:

```
$mkdir dirname
```

Here, directory is the absolute or relative pathname of the directory you want to create. For example, the command:

```
$mkdir mydir
```

Creates the directory mydir in the current directory. Here is another example:

```
$ mkdir /tmp/test-dir
```

This command creates the directory test-dir in the /tmp directory. The **mkdir** command produces no output if it successfully creates the requested directory.

If you give more than one directory on the command line, mkdir creates each of the directories. For example:

```
$ mkdir docs pub
```

Creates the directories docs and pub under the current directory.

Creating Parent Directories:

Sometimes when you want to create a directory, its parent directory or directories might not exist. In this case, mkdir issues an error message as follows:

```
$mkdir /tmp/amrood/test
mkdir: Failed to make directory "/tmp/amrood/test";
No such file or directory
```

In such cases, you can specify the **-p** option to the **mkdir** command. It creates all the necessary directories for you. For example:

```
$mkdir -p /tmp/amrood/test
```

Above command creates all the required parent directories.

Removing Directories:

Directories can be deleted using the **rmdir** command as follows:

Changing Directories:

You can use the cd command to do more than change to a home directory: You can use it to change to any directory by specifying a valid absolute or relative path. The syntax is as follows:

$cd dirname

Here, dirname is the name of the directory that you want to change to. For example, the command:

$cd /usr/local/bin

Changes to the directory /usr/local/bin. From this directory you can cd to the directory /usr/home/amrood using the following relative path:

$cd ../../home/amrood

Renaming Directories:

The mv (move) command can also be used to rename a directory. The syntax is as follows:

$mv olddir newdir

You can rename a directory **mydir** to **yourdir** as follows:

$mv mydir yourdir

Unix File Permission:

File ownership is an important component of UNIX that provides a secure method for storing files. Every file in UNIX has the following attributes:

- **Owner permissions:** The owner's permissions determine what actions the owner of the file can perform on the file.
- **Group permissions:** The group's permissions determine what actions a user, who is a member of the group that a file belongs to, can perform on the file.
- **Other (world) permissions:** The permissions for others indicate what action all other users can perform on the file.

The Permission Indicators:

While using **ls -l** command it displays various information related to file permission as follows:

$ls -l /home/amrood

-rwxr-xr-- 1 amrood users 1024 Nov 2 00:10 myfile

drwxr-xr--- 1 amrood users 1024 Nov 2 00:10 mydir

Here first column represents different access mode ie. permission associated with a file or directory. The permissions are broken into groups of threes, and each position in the group denotes a specific permission, in this order: read (r), write (w), execute (x):

- The first three characters (2-4) represent the permissions for the file's owner. For example -rwxr-x**r--** represents that onwer has read (r), write (w) and execute (x) permission.
- The second group of three characters (5-7) consists of the permissions for the group to which the file belongs. For example -rwxr-x**r--** represents that group has read (r) and execute (x) permission but no write permission.

- The last group of three characters (8-10) represents the permissions for everyone else. For example -rwxr-x**r--** represents that other world has read (r) only permission.

File Access Modes: The permissions of a file are the first line of defense in the security of a Unix system. The basic building blocks of Unix permissions are the **read**, **write**, and **execute** permissions, which are described below:

User with execute permissions can run a file as a program.

Directory Access Modes:

Directory access modes are listed and organized in the same manner as any other file. There are a few differences that need to be mentioned:

1. Read:

Access to a directory means that the user can read the contents. The user can look at the filenames inside the directory.

2. Write:

Access means that the user can add or delete files to the contents of the directory.

3. Execute:

Executing a directory doesn't really make a lot of sense so think of this as a traverse permission.
A user must have execute access to the **bin** directory in order to execute ls or cd command.

Changing Permissions:

To change file or directory permissions, you use the **chmod** (change mode) command. There are two ways to use chmod: symbolic mode and absolute mode.

Using chmod in Symbolic Mode:

The easiest way for a beginner to modify file or directory permissions is to use the symbolic mode. With symbolic permissions you can add, delete, or specify the permission set you want by using the operators in the following table.

Chmod operator	Description
+	Adds the designated permission(s) to a file or directory.
-	Removes the designated permission(s) from a file or directory.
=	Sets the designated permission(s).

Here's an example using testfile. Running ls -1 on testfile shows that the file's permissions are as follows:

```
$ls -l testfile
-rwxrwxr-- 1 amrood users 1024 Nov 2 00:10 testfile
```

Then each example chmod command from the preceding table is run on testfile, followed by ls -l so you can see the permission changes:

```
$chmod o+wx testfile
$ls -l testfile
-rwxrwxrwx 1 amrood users 1024 Nov 2 00:10 testfile

$chmod u-x testfile
$ls -l testfile
-rw-rwxrwx 1 amrood users 1024 Nov 2 00:10 testfile
$chmod g=r-x testfile
$ls -l testfile
-rw-r-xrwx 1 amrood users 1024 Nov 2 00:10 testfile
```

The second way to modify permissions with the chmod command is to use a number to specify each set of permissions for the file.

Each permission is assigned a value, as the following table shows, and the total of each set of permissions provides a number for that set.

Number	Octal Permission Representation	Ref
0	No permission	---
1	Execute permission	--x
2	Write permission	-w-
3	Execute and write permission: 1 (execute) + 2 (write) = 3	-wx
4	Read permission	r--
5	Read and execute permission: 4 (read) + 1 (execute) = 5	r-x
6	Read and write permission: 4 (read) + 2 (write) = 6	rw-
7	All permissions: 4 (read) + 2 (write) + 1 (execute) = 7	rwx

Here's an example using testfile. Running ls -1 on testfile shows that the file's permissions are as follows:

$ls -l testfile

-rwxrwxr-- 1 amrood users 1024 Nov 2 00:10 testfile

Then each example chmod command from the preceding table is run on testfile, followed by ls -l so you can see the permission changes:

$ chmod 755 testfile

$ls -l testfile

-rwxr-xr-x 1 amrood users 1024 Nov 2 00:10 testfile

$chmod 743 testfile

$ls -l testfile

-rwxr---wx 1 amrood users 1024 Nov 2 00:10 testfile

$chmod 043 testfile

$ls -l testfile

----r---wx 1 amrood users 1024 Nov 2 00:10 testfile

Changing Owners and Groups:

While creating an account on Unix, it assigns a owner ID and a group ID to each user. All the permissions mentioned above are also assigned based on Owner and Groups.

Two commands are available to change the owner and the group of files:

1. **chown:** The chown command stands for "change owner" and is used to change the owner of a file.
2. **chgrp:** The chgrp command stands for "change group" and is used to change the group of a file.

Changing Ownership:

The chown command changes the ownership of a file. The basic syntax is as follows:
$ chown user filelist
The value of user can be either the name of a user on the system or the user id (uid) of a user on the system.
Following example:
$ chown amrood testfile

Changing Group Ownership:

The chrgp command changes the group ownership of a file. The basic syntax is as follows: Unix
$ chgrp group filelist
The value of group can be the name of a group on the system or the group ID (GID) of a group on the system.
Following example:
$ chgrp special testfile

UNIT-2

In this chapter, we will discuss in detail about file management in Unix. All data in Unix is organized into files. All files are organized into directories. These directories are organized into a tree-like structure called the filesystem.

When you work with Unix, one way or another, you spend most of your time working with files. This tutorial will help you understand how to create and remove files, copy and rename them, create links to them, etc.

In Unix, there are three basic types of files –

- **Ordinary Files** – An ordinary file is a file on the system that contains data, text, or program instructions. In this tutorial, you look at working with ordinary files.
- **Directories** – Directories store both special and ordinary files. For users familiar with Windows or Mac OS, Unix directories are equivalent to folders.
- **Special Files** – Some special files provide access to hardware such as hard drives, CD-ROM drives, modems, and Ethernet adapters. Other special files are similar to aliases or shortcuts and enable you to access a single file using different names.

ListingFiles

To list the files and directories stored in the current directory, use the following command:

```
$ls
```

Here is the sample output of the above command –

```
$ls

bin         hosts   lib      res.03
ch07        hw1     pub      test_results
```

The command **ls** supports the **-l** option which would help you to get more information about the listed files –

```
$ls -l
drwxrwxr-x  2 amrood amrood      4096 Dec  2 09:59 uml
-rw-rw-r--  1 amrood amrood      5341 Dec  2 08:38 uml.jpg
drwxr-xr-x  2 amrood amrood      4096 Feb  1  2006 univ
drwxr-xr-x  2 root   root        4096 Dec  9  2007 urlspedia
-rw-r--r--  1 root   root      276480 Dec  9  2007 urlspedia.tar
drwxr-xr-x  8 root   root        4096 Nov  2  2007 usr
drwxr-xr-x  2    200    300      4096 Nov  2  2007 webthumb-1.01
-rwxr-xr-x  1 root   root        3192 Nov  2  2007 webthumb.php
-rw-rw-r--  1 amrood amrood     20480 Nov  2  2007 webthumb.tar
-rw-rw-r--  1 amrood amrood      5654 Aug  9  2007 yourfile.mid
-rw-rw-r--  1 amrood amrood    166255 Aug  9  2007 yourfile.swf
drwxr-xr-x  1 amrood amrood      4096 May  2  2007 zlib-1.2.3
$
```

Here is the information about all the listed columns –

- **First Column**: Represents the file type and the permission given on the file. Below is the description of all type of files.
- **Second Column**: Represents the number of memory blocks taken by the file or directory.
- **Third Column**: Represents the owner of the file. This is the Unix user who created this file.
- **Fourth Column**: Represents the group of the owner. Every Unix user will have an associated group.
- **Fifth Column**: Represents the file size in bytes.
- **Sixth Column**: Represents the date and the time when this file was created or modified for the last time.
- **Seventh Column**: Represents the file or the directory name.

In the **ls -l** listing example, every file line begins with a **d**, **-**, or **l**. These characters indicate the type of the file that's listed.

Prefix	Description

-	Regular file, such as an ASCII text file, binary executable, or hard link
b	Block special file. Block input/output device file such as a physical hard drive
c	Character special file. Raw input/output device file such as a physical hard drive
d	Directory file that contains a listing of other files and directories
l	Symbolic link file. Links on any regular file
p	Named pipe. A mechanism for interprocess communications
s	Socket used for interprocess communication

Metacharacters

Metacharacters have a special meaning in Unix. For example, ***** and **?** are metacharacters. We use ***** to match 0 or more characters, a question mark (**?**) matches with a single character.

For Example –

```
$ls ch*.doc
```

Displays all the files, the names of which start with **ch** and end with **.doc** –

```
ch01-1.doc   ch010.doc  ch02.doc    ch03-2.doc
ch04-1.doc   ch040.doc  ch05.doc    ch06-2.doc
ch01-2.doc ch02-1.doc c
```

Here, ***** works as meta character which matches with any character. If you want to display all the files ending with just **.doc**, then you can use the following command –

```
$ls *.doc
```

Hidden Files

An invisible file is one, the first character of which is the dot or the period character (.). Unix programs (including the shell) use most of these files to store configuration information.

Some common examples of the hidden files include the files −

- **.profile** − The Bourne shell (sh) initialization script
- **.kshrc** − The Korn shell (ksh) initialization script
- **.cshrc** − The C shell (csh) initialization script
- **.rhosts** − The remote shell configuration file

To list the invisible files, specify the **-a** option to **ls** −

```
$ ls -a

.          .profile       docs      lib      test_results
..         .rhosts        hosts     pub      users
.emacs     bin            hw1       res.01   work
.exrc      ch07           hw2       res.02
.kshrc     ch07.bak       hw3       res.03
$
```

- Single dot (**.**) − This represents the current directory.
- Double dot (**..**) − This represents the parent directory.

Creating Files

You can use the **vi** editor to create ordinary files on any Unix system. You simply need to give the following command −

```
$ vi filename
```

The above command will open a file with the given filename. Now, press the key **i** to come into the edit mode. Once you are in the edit mode, you can start writing your content in the file a

```
I'm going to save this content in this file.
```

Once you are done with the program, follow these steps –

- Press the key **esc** to come out of the edit mode.
- Press two keys **Shift + Z** together to come out of the file completely. You will now

have a file created with **filename** in the current directory.

```
$ vi filename
$
```

Editing Files

You can edit an existing file using the **vi** editor. We will discuss in short how to open an existing file –

```
$ vi filename
```

Once the file is opened, you can come in the edit mode by pressing the key **i** and then you can proceed by editing the file. If you want to move here and there inside a file, then first you need to come out of the edit mode by pressing the key **Esc.** After this, you can use the following keys to move inside a file –

- **l** key to move to the right side.
- **h** key to move to the left side.
- **k** key to move upside in the file.
- **j** key to move downside in the file.

So using the above keys, you can position your cursor wherever you want to edit. Once you are positioned, then you can use the **i** key to come in the edit mode. Once you are done with the editing in your file, press **Esc** and finally two keys **Shift + ZZ** together to come out of the file completely.

DisplayContentofa File

You can use the **cat** command to see the content of a file. Following is a simple example to see the content of the above created file –

```
$ cat filename
This is unix file....I created it for the first time.....
I'm going to save this content in this file.
```

```
$
```

You can display the line numbers by using the **-b** option along with the **cat** command as follows –

```
$ cat -b filename
1  This is unix file....I created it for the first time.....
2  I'm going to save this content in this file.
$
```

Counting Words in a File

You can use the **wc** command to get a count of the total number of lines, words, and characters contained in a file. Following is a simple example to see the information about the file created above –

```
$ wc filename
2  19 103 filename
$
```


Here is the detail of all the four columns –

- **First Column**: Represents the total number of lines in the file.
- **Second Column**: Represents the total number of words in the file.
- **Third Column**: Represents the total number of bytes in the file. This is the actual size of the file.
- **Fourth Column**: Represents the file name.

You can give multiple files and get information about those files at a time. Following is simple syntax –

```
$ wc filename1 filename2 filename3
```

Copying Files

To make a copy of a file use the **cp** command. The basic syntax of the command is –

```
$ cp source_file destination_file
```

Following is the example to create a copy of the existing file **filename**.

```
$ cp filename copyfile
$
```

You will now find one more file **copyfile** in your current directory. This file will exactly be the same as the original file **filename**.

RenamingFiles

To change the name of a file, use the **mv** command. Following is the basic syntax –

```
$ mv old_file new_file
```

The following program will rename the existing file **filename** to **newfile.**

```
$ mv filename newfile
$
```

The **mv** command will move the existing file completely into the new file. In this case, you will find only **newfile** in your current directory.

DeletingFiles

To delete an existing file, use the **rm** command. Following is the basic syntax –

```
$ rm filename
```

Caution: A file may contain useful information. It is always recommended to be careful while using this **Delete** command. It is better to use the **-i** option along with **rm** command.

Following is the example which shows how to completely remove the existing file **filename**.

```
$ rm filename
$
```

You can remove multiple files at a time with the command given below –

```
$ rm filename1 filename2 filename3
$
```

Standard Unix Streams

Under normal circumstances, every Unix program has three streams (files) opened for it when it starts up −

- **stdin** − This is referred to as the *standard input* and the associated file descriptor is 0. This is also represented as STDIN. The Unix program will read the default input from STDIN.
- **stdout** − This is referred to as the *standard output* and the associated file descriptor is 1. This is also represented as STDOUT. The Unix program will write the default output at STDOUT
- **stderr** − This is referred to as the *standard error* and the associated file descriptor is 2. This is also represented as STDERR. The Unix program will write all the error messages at STDERR.

In this chapter, we will discuss in detail about directory management in Unix.

A directory is a file the solo job of which is to store the file names and the related information. All the files, whether ordinary, special, or directory, are contained in directories.

Unix uses a hierarchical structure for organizing files and directories. This structure is often referred to as a directory tree. The tree has a single root node, the slash character (**/**), and all other directories are contained below it.

HomeDirectory

The directory in which you find yourself when you first login is called your home directory.

You will be doing much of your work in your home directory and subdirectories that you'll be creating to organize your files.

You can go in your home directory anytime using the following command –

```
$cd ~
$
```

Here **~** indicates the home directory. Suppose you have to go in any other user's home directory, use the following command –

```
$cd ~username
$
```

To go in your last directory, you can use the following command –

```
$cd -
$
```


Absolute/Relative Pathnames

Directories are arranged in a hierarchy with root **(/)** at the top. The position of any file within the hierarchy is described by its pathname.

Elements of a pathname are separated by a **/**. A pathname is absolute, if it is described in relation to root, thus absolute pathnames always begin with a **/**.

Following are some examples of absolute filenames.

```
/etc/passwd
/users/sjones/chem/notes
/dev/rdsk/Os3
```

A pathname can also be relative to your current working directory. Relative pathnames never begin with **/**. Relative to user amrood's home directory, some pathnames might look like this –

```
chem/notes
personal/r
```

To determine where you are within the filesystem hierarchy at any time, enter the command **pwd** to print the current working directory –

```
$pwd
/user0/home/amrood

$
```

Listing Directories

To list the files in a directory, you can use the following syntax –

```
$ls dirname
```

Following is the example to list all the files contained in **/usr/local** directory –

```
$ls /usr/local
X11       bin       gimp      jikes     sbin
ace       doc       include   lib       share
atalk     etc       info      man       ami
```

Creating Directories

We will now understand how to create directories. Directories are created by the following command –

```
$mkdir dirname
```

Here, directory is the absolute or relative pathname of the directory you want to create. For example, the command –

```
$mkdir mydir
$
```

Creates the directory **mydir** in the current directory. Here is another example –

```
$mkdir /tmp/test-dir
$
```

This command creates the directory **test-dir** in the **/tmp** directory. The **mkdir** command produces no output if it successfully creates the requested directory.

If you give more than one directory on the command line, **mkdir** creates each of the directories. For example, –

```
$mkdir docs pub
$
```

Creates the directories docs and pub under the current directory.

CreatingParentDirectories

We will now understand how to create parent directories. Sometimes when you want to create a directory, its parent directory or directories might not exist. In this case, **mkdir** issues an error message as follows –

```
$mkdir /tmp/amrood/test
mkdir: Failed to make directory
"/tmp/amrood/test"; No such file or
directory
```

In such cases, you can specify the **-p** option to the **mkdir** command. It creates all the necessary directories for you. For example –

```
$mkdir -p /tmp/amrood/test
$
```

The above command creates all the required parent directories.

Removing Directories

Directories can be deleted using the **rmdir** command as follows –

```
$rmdir dirname
$
```

Note – To remove a directory, make sure it is empty which means there should not be any file or sub-directory inside this directory.

You can remove multiple directories at a time as follows –

```
$rmdir dirname1 dirname2 dirname3
$
```

The above command removes the directories dirname1, dirname2, and dirname3, if they are empty. The **rmdir** command produces no output if it is successful.

Changing Directories

You can use the **cd** command to do more than just change to a home directory. You can use it to change to any directory by specifying a valid absolute or relative path. The syntax is as given below –

```
$cd dirname
$
```

Here, **dirname** is the name of the directory that you want to change to. For example, the command –

```
$cd /usr/local/bin
$
```

Changes to the directory **/usr/local/bin**. From this directory, you can **cd** to the directory **/usr/home/amrood** using the following relative path –

```
$cd ../../home/amrood
$
```

Renaming Directories

The **mv (move)** command can also be used to rename a directory. The syntax is as follows:

```
$mv olddir newdir
$
```

You can rename a directory **mydir** to **yourdir** as follows –

```
$mv mydir yourdir
$
```

The directories. (dot) and .. (dot dot)

The **filename .** (dot) represents the current working directory; and the **filename ..** (dot dot) represents the directory one level above the current working directory, often referred to as the parent directory.

If we enter the command to show a listing of the current working directories/files and use the **-a option** to list all the files and the **-l option** to provide the long listing, we will receive the following result.

```
$ls -la
drwxrwxr-x    4    teacher   class    2048  Jul 16 17.56 .
drwxr-xr-x    60   root               1536  Jul 13 14:18 ..
----------    1    teacher   class    4210  May 1 08:27 .profile
-rwxr-xr-x    1    teacher   class    1948  May 12 13:42 memo
$
```

In this chapter, we will discuss in detail about file permission and access modes in Unix. File ownership is an important component of Unix that provides a secure method for storing files. Every file in Unix has the following attributes –

- **Owner permissions** – The owner's permissions determine what actions the owner of the file can perform on the file.
- **Group permissions** – The group's permissions determine what actions a user, who is a member of the group that a file belongs to, can perform on the file.
- **Other (world) permissions** – The permissions for others indicate what action all other users can perform on the file.

The Permission Indicators

While using **ls -l** command, it displays various information related to file permission as follows –

```
$ls -l /home/amrood
-rwxr-xr--1 amrood users 1024Nov 2 00:10  myfile
drwxr-xr--- 1 amrood         users 1024 Nov 2 00:10  mydir
```

Here, the first column represents different access modes, i.e., the permission associated with a file or a directory.

The permissions are broken into groups of threes, and each position in the group denotes a specific permission, in this order: read (r), write (w), execute (x) –

- The first three characters (2-4) represent the permissions for the file's owner. For example, **-rwxr-xr--** represents that the owner has read (r), write (w) and execute (x) permission.
- The second group of three characters (5-7) consists of the permissions for the group to which the file belongs. For example, -**rwxr-xr--** represents that the group has read (r) and execute (x) permission, but no write permission.
- The last group of three characters (8-10) represents the permissions for everyone else. For example, **-rwxr-xr--** represents that there is **read (r)** only permission.

FileAccessModes

The permissions of a file are the first line of defense in the security of a Unix system. The basic building blocks of Unix permissions are the **read**, **write**, and **execute** permissions, which have been described below –

Read

Grants the capability to read, i.e., view the contents of the file.

Write

Grants the capability to modify, or remove the content of the file.

Execute

User with execute permissions can run a file as a program.

DirectoryAccessModes

Directory access modes are listed and organized in the same manner as any other file. There are a few differences that need to be mentioned:

Read

Access to a directory means that the user can read the contents. The user can look at the **filenames** inside the directory.

Write

Access means that the user can add or delete files from the directory.

Execute

Executing a directory doesn't really make sense, so think of this as a traverse permission.

A user must have **execute** access to the **bin** directory in order to execute the **ls** or the **cd** command.

ChangingPermissions

To change the file or the directory permissions, you use the **chmod** (change mode) command. There are two ways to use chmod — the symbolic mode and the absolute mode

The easiest way for a beginner to modify file or directory permissions is to use the symbolic mode. With symbolic permissions you can add, delete, or specify the permission set you want by using the operators in the following table.

chmod Operator	Description
+	Adds the designated permission(s) to a file or directory.
-	Removes the designated permission(s) from a file or directory.
=	Sets the designated permission(s).

Here's an example using **testfile**. Running **ls -1** on the testfile shows that the file's permissions are as follows –

```
$ls -l testfile
-rwxrwxr--1 amrood users 1024Nov 2 00:10  testfile
```

Then each example **chmod** command from the preceding table is run on the testfile, followed by **ls –l**, so you can see the permission changes –

```
$chmod o+wx testfile
$ls -l testfile
-rwxrwxrwx  1 amrood   users 1024  Nov  2 00:10  testfile
$chmod u-x testfile
$ls -l testfile
-rw-rwxrwx  1 amrood   users 1024  Nov  2 00:10  testfile
$chmod g=rx testfile
$ls -l testfile
-rw-r-xrwx  1 amrood   users 1024  Nov  2 00:10  testfile
```

Here's how you can combine these commands on a single line:

```
$chmod o+wx,u-x,g=rx testfile
$ls -l testfile
-rw-r-xrwx1 amrood users 1024Nov 2 00:10  testfile
```

Using chmod withAbsolute Permissions

The second way to modify permissions with the chmod command is to use a number to specify each set of permissions for the file.

Each permission is assigned a value, as the following table shows, and the total of each set of permissions provides a number for that set.

Number	Octal Permission Representation	Ref
0	No permission	---
1	Execute permission	--x
2	Write permission	-w-
3	Execute and write permission: 1 (execute) + 2 (write) = 3	-wx
4	Read permission	r--
5	Read and execute permission: 4 (read) + 1 (execute) = 5	r-x
6	Read and write permission: 4 (read) + 2 (write) = 6	rw-
7	All permissions: 4 (read) + 2 (write) + 1 (execute) = 7	rwx

Here's an example using the testfile. Running **ls -1** on the testfile shows that the file's permissions are as follows –

```
$ls -l testfile
-rwxrwxr--1 amrood users 1024Nov 2 00:10  testfile
```

Then each example **chmod** command from the preceding table is run on the testfile, followed by **ls –l**, so you can see the permission changes –

```
$ chmod 755 testfile
$ls -l testfile
-rwxr-xr-x1 amrood users 1024Nov 2 00:10  testfile
$chmod 743 testfile
```

```
$ls -l testfile
-rwxr---wx1 amrood users 1024Nov 2 00:10  testfile
$chmod 043 testfile
$ls -l testfile
----r---wx1 amrood users 1024Nov 2 00:10  testfile
```

Changing Owners and Groups

While creating an account on Unix, it assigns a **owner ID** and a **group ID** to each user. All the permissions mentioned above are also assigned based on the Owner and the Groups.

Two commands are available to change the owner and the group of files −

- **chown** − The **chown** command stands for **"change owner"** and is used to change the owner of a file.
- **chgrp** − The **chgrp** command stands for **"change group"** and is used to change the group of a file.

Changing Ownership

The **chown** command changes the ownership of a file. The basic syntax is as follows −

```
$ chown user filelist
```

The value of the user can be either the **name of a user** on the system or the **user id (uid)** of a user on the system.

The following example will help you understand the concept −

```
$ chown amrood testfile
$
```

Changes the owner of the given file to the user **amrood**.

NOTE: The super user, root, has the unrestricted capability to change the ownership of any file but normal users can change the ownership of only those files that they own.

Changing Group Ownership

The **chgrp** command changes the group ownership of a file. The basic syntax is as follows:

```
$ chgrp group filelist
```

The value of group can be the **name of a group** on the system or **the group ID (GID)** of a group on the system.

Following example helps you understand the concept:

```
$ chgrp special testfile
$
```

Changes the group of the given file to **special** group.

SUID and SGID File Permission

Often when a command is executed, it will have to be executed with special privileges in order to accomplish its task.

As an example, when you change your password with the **passwd** command, your new password is stored in the file **/etc/shadow**.

As a regular user, you do not have **read** or **write** access to this file for security reasons, but when you change your password, you need to have the write permission to this file. This means that the **passwd** program has to give you additional permissions so that you can write to the file **/etc/shadow**.

Additional permissions are given to programs via a mechanism known as the **Set User ID (SUID)** and **Set Group ID (SGID)** bits.

When you execute a program that has the SUID bit enabled, you inherit the permissions of that program's owner. Programs that do not have the SUID bit set are run with the permissions of the user who started the program.

This is the case with SGID as well. Normally, programs execute with your group permissions, but instead your group will be changed just for this program to the group owner of the program.

The SUID and SGID bits will appear as the letter **"s"** if the permission is available. The SUID **"s"** bit will be located in the permission bits where the owners' **execute** permission normally resides.

For example, the command -

```
$ ls -l /usr/bin/passwd
-r-sr-xr-x  1   root   bin  19031 Feb 7 13:47  /usr/bin/passwd*
$
```

one of the following users –

- The owner of the sticky directory
- The owner of the file being removed
- The super user, root

To set the SUID and SGID bits for any directory try the following command –

```
$ chmod ug+s dirname
$ ls -l
drwsr-sr-x 2 root root    4096 Jun 19 06:45 dirname
$
```

UNIT-3

A **Shell** provides you with an interface to the Unix system. It gathers input from you and executes programs based on that input. When a program finishes executing, it displays that program's output.

Shell is an environment in which we can run our commands, programs, and shell scripts. There are different flavors of a shell, just as there are different flavors of operating systems. Each flavor of shell has its own set of recognized commands and functions.

Shell Prompt

The prompt, **$**, which is called the **command prompt**, is issued by the shell. While the prompt is displayed, you can type a command.

Shell reads your input after you press **Enter**. It determines the command you want executed by looking at the first word of your input. A word is an unbroken set of characters. Spaces and tabs separate words.

Following is a simple example of the **date** command, which displays the current date and time –

```
$date
Thu Jun 25 08:30:19 MST 2009
```


You can customize your command prompt using the environment variable PS1 explained in the Environment tutorial.

Shell Types

In Unix, there are two major types of shells –

- Korn shell (ksh)
- Bourne Again shell (bash)
- POSIX shell (sh)

The different C-type shells follow –

- C shell (csh)
- TENEX/TOPS C shell (tcsh)

The original Unix shell was written in the mid-1970s by Stephen R. Bourne while he was at the AT&T Bell Labs in New Jersey.

Bourne shell was the first shell to appear on Unix systems, thus it is referred to as "the shell".

Bourne shell is usually installed as **/bin/sh** on most versions of Unix. For this reason, it is the shell of choice for writing scripts that can be used on different versions of Unix.

In this chapter, we are going to cover most of the Shell concepts that are based on the Borne Shell.

Shell Scripts

The basic concept of a shell script is a list of commands, which are listed in the order of execution. A good shell script will have comments, preceded by **#** sign, describing the steps.

There are conditional tests, such as value A is greater than value B, loops allowing us to go through massive amounts of data, files to read and store data, and variables to read and store data, and the script may include functions.

We are going to write many scripts in the next sections. It would be a simple text file in which we would put all our commands and several other required constructs that tell the shell environment what to do and when to do it.

Shell scripts and functions are both interpreted. This means they are not compiled.

Example Script

Assume we create a **test.sh** script. Note all the scripts would have the **.sh** extension. Before you add anything else to your script, you need to alert the system that a shell script is being started. This is done using the **shebang** construct. For example –


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

This tells the system that the commands that follow are to be executed by the Bourne shell. *It's called a shebang because the # symbol is called a hash, and the ! symbol is called a bang*.

Shell Comments

You can put your comments in your script as follows –

```
#!/bin/bash

# Author : Zara Ali
# Copyright (c) Tutorialspoint.com
# Script follows here:
pwd
ls
```

Save the above content and make the script executable –

```
$chmod +x test.sh
```

The shell script is now ready to be executed –

```
$./test.sh
```

Upon execution, you will receive the following result –

```
/home/amrood
index.htm  unix-basic_utilities.htm  unix-directories.htm
test.sh    unix-communication.htm    unix-environment.htm
```

Note – To execute a program available in the current directory, use **./program_name**

Extended Shell Scripts

Shell scripts have several required constructs that tell the shell environment what to do and when to do it. Of course, most scripts are more complex than the above one.

The shell is, after all, a real programming language, complete with variables, control structures, and so forth. No matter how complicated a script gets, it is still just a list of commands executed sequentially.

The following script uses the **read** command which takes the input from the keyboard and assigns it as the value of the variable PERSON and finally prints it on STDOUT.

```
#!/bin/sh

# Author : Zara Ali
# Copyright (c) Tutorialspoint.com
# Script follows here:

echo "What is your name?"
read PERSON
echo "Hello, $PERSON"
```

s chapter, we will learn how to use Shell variables in Unix. A variable is a character string to which we assign a value. The value assigned could be a number, text, filename, device, or any other type of data.

A variable is nothing more than a pointer to the actual data. The shell enables you to create, assign, and delete variables.

Variable Names

The name of a variable can contain only letters (a to z or A to Z), numbers (0 to 9) or the underscore character (_).

By convention, Unix shell variables will have their names in UPPERCASE.

The following examples are valid variable names –

```
_ALI
TOKEN_A
VAR_1
VAR_2
```

Following are the examples of invalid variable names –

```
2_VAR
-VARIABLE
VAR1-VAR2
VAR_A!
```

The reason you cannot use other characters such as **!**, *****, or **-** is that these characters have a special meaning for the shell.

Defining Variables

Variables are defined as follows –

```
variable_name=variable_value
```

For example –


```
NAME="Zara Ali"
```

The above example defines the variable NAME and assigns the value "Zara Ali" to it. Variables of this type are called **scalar variables**. A scalar variable can hold only one value at a time.

cess the value of defined variable, prefix its name with the dollar sign ($) –

For example, the following script will access the value of defined variable NAME and print it on STDOUT –

Live Demo

```
#!/bin/sh

NAME="Zara Ali"
echo $NAME
```

The above script will produce the following value –

```
Zara Ali
```

Read-only Variables

Shell provides a way to mark variables as read-only by using the read-only command. After a variable is marked read-only, its value cannot be changed.

For example, the following script generates an error while trying to change the value of NAME –

Live Demo

```
#!/bin/sh

NAME="Zara Ali"
readonly NAME
NAME="Qadiri"
```

The above script will generate the following result –

```
/bin/sh: NAME: This variable is read only.
```

Unsetting Variables

Unsetting or deleting a variable directs the shell to remove the variable from the list of variables that it tracks. Once you unset a variable, you cannot access the stored value in the variable.

Following is the syntax to unset a defined variable using the **unset** command –


```
unset variable_name
```

The above command unsets the value of a defined variable. Here is a simple example that demonstrates how the command works –

Variable Types

When a shell is running, three main types of variables are present –

- **Local Variables** – A local variable is a variable that is present within the current instance of the shell. It is not available to programs that are started by the shell. They are set at the command prompt.
- **Environment Variables** – An environment variable is available to any child process of the shell. Some programs need environment variables in order to function correctly. Usually, a shell script defines only those environment variables that are needed by the programs that it runs.
- **Shell Variables** – A shell variable is a special variable that is set by the shell and is required by the shell in order to function correctly. Some of these variables are environment variables whereas others are local variables.

In this chapter, we will discuss in detail about special variable in Unix. In one of our previous chapters, we understood how to be careful when we use certain non alphanumeric characters in variable names. This is because those characters are used in the names of special Unix variables. These variables are reserved for specific functions.

For example, the **$** character represents the process ID number, or PID, of the current shell –

```
$echo $$
```

The above command writes the PID of the current shell –

```
29949
```

The following table shows a number of special variables that you can use in your shell scripts –

Sr.No.	Variable & Description
1	**$0:**The filename of the current script.
2	**$n:**These variables correspond to the arguments with which a script was invoked. Here **n** is a positive decimal number corresponding to the position of an argument (the first argument is $1, the second argument is $2, and so on).
3	**$#:**The number of arguments supplied to a script.
4	**$*:**All the arguments are double quoted. If a script receives two arguments, $* is equivalent to $1 $2.
5	**$@:**All the arguments are individually double quoted. If a script receives two arguments, $@ is equivalent to $1 $2.
6	**$?:**The exit status of the last command executed.
7	**$$:**The process number of the current shell. For shell scripts, this is the process ID under which they are

The command-line arguments $1, $2, $3, ...$9 are positional parameters, with $0 pointing to the actual command, program, shell script, or function and $1, $2, $3, ...$9 as the arguments to the command.

Following script uses various special variables related to the command line −

```
#!/bin/sh

echo "File Name: $0"
echo "First Parameter : $1"
echo "Second Parameter : $2"
echo "Quoted Values: $@"
echo "Quoted Values: $*"
echo "Total Number of Parameters : $#"
```

Here is a sample run for the above script −

```
$./test.sh Zara Ali
File Name : ./test.sh
First Parameter : Zara
Second Parameter : Ali
Values: Zara Ali
Quoted Values: Zara Ali
Total Number of Parameters : 2
```

Special Parameters $* and $@

There are special parameters that allow accessing all the command-line arguments at once. **$*** and **$@** both will act the same unless they are enclosed in double quotes, **""**.

Both the parameters specify the command-line arguments. However, the "$*" special parameter takes the entire list as one argument with spaces between and the "$@" special parameter takes the entire list and separates it into separate arguments.

We can write the shell script as shown below to process an unknown number of commandline arguments with either the $* or $@ special parameters −

```
#!/bin/sh

for TOKEN in $*
do
   echo $TOKEN
done
```


Here is a sample run for the above script −

```
$./test.sh Zara Ali 10 Years Old
Zara
Ali
```

The $? variable represents the exit status of the previous command.

Exit status is a numerical value returned by every command upon its completion. As a rule, most commands return an exit status of 0 if they were successful, and 1 if they were unsuccessful.

Some commands return additional exit statuses for particular reasons. For example, some commands differentiate between kinds of errors and will return various exit values depending on the specific type of failure.

Following is the example of successful command –

```
$./test.sh Zara Ali
File Name : ./test.sh
First Parameter : Zara
Second Parameter : Ali
Quoted Values: Zara Ali
Quoted Values: Zara Ali
Total Number of Parameters : 2
$echo $?
0
$
```

In this chapter, we will discuss how to use shell arrays in Unix. A shell variable is capable enough to hold a single value. These variables are called scalar variables.

Shell supports a different type of variable called an **array variable**. This can hold multiple values at the same time. Arrays provide a method of grouping a set of variables. Instead of creating a new name for each variable that is required, you can use a single array variable that stores all the other variables.

All the naming rules discussed for Shell Variables would be applicable while naming arrays.

Defining Array Values

The difference between an array variable and a scalar variable can be explained as follows.

Suppose you are trying to represent the names of various students as a set of variables. Each of the individual variables is a scalar variable as follows –


```
NAME01="Zara"
NAME02="Qadir"
NAME03="Mahnaz"
NAME04="Ayan"
NAME05="Daisy"
```


We can use a single array to store all the above mentioned names. Following is the simplest method of creating

example, the following commands –

```
NAME[0]="Zara"
NAME[1]="Qadir"
NAME[2]="Mahnaz"
NAME[3]="Ayan"
NAME[4]="Daisy"
```

If you are using the **ksh** shell, here is the syntax of array initialization –

```
set -A array_name value1 value2 ... valuen
```

If you are using the **bash** shell, here is the syntax of array initialization –

```
array_name = (value1 ... valuen)
```

Accessing Array Values

After you have set any array variable, you access it as follows –

```
${array_name[index]}
```

Here *array_name* is the name of the array, and *index* is the index of the value to be accessed. Following is an example to understand the concept –

Live Demo

```
#!/bin/sh

NAME[0]="Zara"
NAME[1]="Qadir"
NAME[2]="Mahnaz"
NAME[3]="Ayan"
NAME[4]="Daisy"
echo "First Index: ${NAME[0]}"
echo "Second Index: ${NAME[1]}"
```

The above example will generate the following result –

```
$./test.sh
First Index: Zara
Second Index: Qadir
```


You can access all the items in an array in one of the following ways –

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

```
NAME[2]="Mahnaz"
NAME[3]="Ayan"
NAME[4]="Daisy"
echo "First Method: ${NAME[*]}"
echo "Second Method: ${NAME[@]}"
```


The above example will generate the following result –

```
$./test.sh
First Method: Zara Qadir Mahnaz Ayan Daisy
Second Method: Zara Qadir Mahnaz Ayan Daisy
```

There are various operators supported by each shell. We will discuss in detail about Bourne shell (default shell) in this chapter.

We will now discuss the following operators –

- Arithmetic Operators
- Relational Operators
- Boolean Operators
- String Operators
- File Test Operators

Bourne shell didn't originally have any mechanism to perform simple arithmetic operations but it uses external programs, either **awk** or **expr**.

The following example shows how to add two numbers.

```
#!/bin/sh

val=`expr 2 + 2`
echo "Total value : $val"
```

The above script will generate the following result –

```
Total value : 4
```

The following points need to be considered while adding –

- There must be spaces between operators and expressions. For example, 2+2 is not correct; it should be written as 2 + 2.
- The complete expression should be enclosed between ‘ ‘, called the backtick.

Arithmetic Operators

Operator	Description	Example
+ (Addition)	Adds values on either side of the operator	`expr $a + $b` will give 30
- (Subtraction)	Subtracts right hand operand from left hand operand	`expr $a - $b` will give -10
* (Multiplication)	Multiplies values on either side of the operator	`expr $a \* $b` will give 200
/ (Division)	Divides left hand operand by right hand operand	`expr $b / $a` will give 2
% (Modulus)	Divides left hand operand by right hand operand and returns remainder	`expr $b % $a` will give 0
= (Assignment)	Assigns right operand in left operand	a = $b would assign value of b into a
== (Equality)	Compares two numbers, if both are same then returns true.	[$a == $b] would return false.
!= (Not Equality)	Compares two numbers, if both are different then returns true.	[$a != $b] would return true.

It is very important to understand that all the conditional expressions should be inside square braces with spaces around them, for example **[$a == $b]** is correct whereas, **[$a==$b]** is incorrect.

All the arithmetical calculations are done using long integers.

Relational Operators

Bourne Shell supports the following relational operators that are specific to numeric values. These operators do not work for string values unless their value is numeric.

For example, following operators will work to check a relation between 10 and 20 as well as in between "10" and "20" but not in between "ten" and "twenty".

Assume variable **a** holds 10 and variable **b** holds 20 then –

Show Examples

Operator	Description	Example
-eq	Checks if the value of two operands are equal or not; if yes, then the condition becomes true.	[$a -eq $b] is not true.
-ne	Checks if the value of two operands are equal or not; if values are not equal, then the condition becomes true.	[$a -ne $b] is true.
-gt	Checks if the value of left operand is greater than the value of right operand; if yes, then the condition becomes true.	[$a -gt $b] is not true.
-lt	Checks if the value of left operand is less than the value of right operand; if yes, then the condition becomes true.	[$a -lt $b] is true.

s around them. For example, **[$a <= $b]** is correct whereas, **[$a <= $b]** is incorrect.

Boolean Operators

The following Boolean operators are supported by the Bourne Shell.

Assume variable **a** holds 10 and variable **b** holds 20 then –

Show Examples

Operator	Description	Example
!	This is logical negation. This inverts a true condition into false and vice versa.	[! false] is true.
-o	This is logical **OR**. If one of the operands is true, then the condition becomes true.	[$a -lt 20 -o $b -gt 100] is true.
-a	This is logical **AND**. If both the operands are true, then the condition becomes true otherwise false.	[$a -lt 20 -a $b -gt 100] is false.

String Operators

The following string operators are supported by Bourne Shell.

Assume variable **a** holds "abc" and variable **b** holds "efg" then –

Show Examples

Operator	Description	Example
=	Checks if the value of two operands are equal or not; if yes, then the condition becomes true.	[$a = $b] is not true.
!=	Checks if the value of two operands are equal or not; if values are not equal then the condition becomes true.	[$a != $b] is true.
-z	Checks if the given string operand size is zero; if it is zero length, then it returns true.	[-z $a] is not true.
-n	Checks if the given string operand size is non-zero; if it is nonzero length, then it returns true.	[-n $a] is not false.
str	Checks if **str** is not the empty string; if it is empty, then it returns false.	[$a] is not false.

File Test Operators

www.FirstRanker.com

ator	Description	Example
-b file	Checks if file is a block special file; if yes, then the condition becomes true.	[-b $file] is false.
-c file	Checks if file is a character special file; if yes, then the condition becomes true.	[-c $file] is false.
-d file	Checks if file is a directory; if yes, then the condition becomes true.	[-d $file] is not true.
-f file	Checks if file is an ordinary file as opposed to a directory or special file; if yes, then the condition becomes true.	[-f $file] is true.
-g file	Checks if file has its set group ID (SGID) bit set; if yes, then the condition becomes true.	[-g $file] is false.
-k file	Checks if file has its sticky bit set; if yes, then the condition becomes true.	[-k $file] is false.
-p file	Checks if file is a named pipe; if yes, then the condition becomes true.	[-p $file] is false.
-t file	Checks if file descriptor is open and associated with a terminal; if yes, then the condition becomes true.	[-t $file] is false.
-u file	Checks if file has its Set User ID (SUID) bit set; if yes, then the condition becomes true.	[-u $file] is false.
-r file	Checks if file is readable; if yes, then the condition becomes true.	[-r $file] is true.
-w file	Checks if file is writable; if yes, then the condition becomes true.	[-w $file] is true.
-x file	Checks if file is executable; if yes, then the condition becomes true.	[-x $file] is true.
-s file	Checks if file has size greater than 0; if yes, then condition becomes true.	[-s $file] is true.
-e file	Checks if file exists; is true even if file is a directory but exists.	[-e $file] is true.

C Shell Operators

Following link will give you a brief idea on C Shell Operators –

C Shell Operators

Korn Shell Operators

Following link helps you understand Korn Shell Operators –

Korn Shell Operators

The if...else statements

If else statements are useful decision-making statements which can be used to select an option from a given set of options.

Unix Shell supports following forms of **if…else** statement –

- if...fi statement
- if...else...fi statement
- if...elif...else...fi statement

Most of the if statements check relations using relational operators discussed in the previous chapter.

The case...esac Statement

You can use multiple **if...elif** statements to perform a multiway branch. However, this is not always the best solution, especially when all of the branches depend on the value of a single variable.

Unix Shell supports **case...esac** statement which handles exactly this situation, and it does so more efficiently than repeated **if...elif** statements.

There is only one form of **case...esac** statement which has been described in detail here –

- case...esac statement

The **case...esac** statement in the Unix shell is very similar to the **switch...case** statement we have in other programming languages like **C** or **C++** and **PERL**, etc.

In this chapter, we will discuss shell loops in Unix. A loop is a powerful programming tool that enables you to execute a set of commands repeatedly. In this chapter, we will examine the following types of loops available to shell programmers –

- The while loop
- The for loop
- The until loop
- The select loop

You will use different loops based on the situation. For example, the **while** loop executes the given commands until the given condition remains true; the **until** loop executes until a given condition becomes true.

All the loops support nesting concept which means you can put one loop inside another similar one or different one. This nesting can go up to unlimited number of times based on your requirement.

Here is an example of nesting **while** loop. The other loops can be nested based on the programming requirement in a similar way –

Nesting while Loops

It is possible to use a while loop as part of the body of another while loop.

Syntax

```
while command1 ; # this is loop1, the outer loop
do
   Statement(s) to be executed if command1 is true

   while command2 ; # this is loop2, the inner loop
   do
      Statement(s) to be executed if command2 is true
   done

   Statement(s) to be executed if command1 is true
done
```

Example

Here is a simple example of loop nesting. Let's add another countdown loop inside the loop that you used to count to nine –

```
#!/bin/sh

a=0
while [ "$a" -lt 10 ]    # this is loop1
do
   b="$a"
   while [ "$b" -ge 0 ]  # this is loop2
   do
      echo -n "$b "
      b=`expr $b - 1`
   done
   echo
   a=`expr $a + 1`
done
```


This will produce the following result. It is important to note how **echo -n** works here. Here **-n** option lets echo avoid printing a new line character.

```
0
1 0
2 1 0
3 2 1 0
```

In this chapter, we will discuss the Loop control in Unix. So far you have looked at creating loops and working loops to accomplish different tasks. Sometimes you need to stop a loop or skip iterations of the loop.

In this chapter, we will learn following two statements that are used to control shell loops–

- The **break** statement
- The **continue** statement

The infinite Loop

All the loops have a limited life and they come out once the condition is false or true depending on the loop.

A loop may continue forever if the required condition is not met. A loop that executes forever without terminating executes for an infinite number of times. For this reason, such loops are called infinite loops.

Example

Here is a simple example that uses the **while** loop to display the numbers zero to nine –

```
#!/bin/sh

a=10

until [ $a -lt 10 ]
do
   echo $a
   a=expr $a + 1`
done
```

This loop continues forever because **a** is always **greater than** or **equal to 10** and it is never less than 10.

The break Statement

The **break** statement is used to terminate the execution of the entire loop, after completing the execution of all of the lines of code up to the break statement. It then steps down to the code following the end of the loop.

Syntax

The following **break** statement is used to come out of a loop –

```
break
```

The break command can also be used to exit from a nested loop using this format –

```
break n
```

```
#!/bin/sh
while [ $a -lt 10 ]
do
   echo $a
   if [ $a -eq 5 ]
   then
      break
   fi
   a=`expr $a + 1`
done
```

Upon execution, you will receive the following result –

```
0
1
2
3
4
5
```

Here is a simple example of nested for loop. This script breaks out of both loops if **var1 equals 2** and **var2 equals 0 –**

```
#!/bin/sh

for var1 in 1 2 3
do
   for var2 in 0 5
   do
      if [ $var1 -eq 2 -a $var2 -eq 0 ]
      then
         break 2
      else
         echo "$var1 $var2"
      fi
   done
done
```

Upon execution, you will receive the following result. In the inner loop, you have a break command with the argument 2. This indicates that if a condition is met you should break out of outer loop and ultimately from the inner loop as well.


```
1 0
1 5
```

The continue statement

continue

with the break statement, an integer argument can be given to the continue command to skip commands from nested loops.


```
continue n
```

Here **n** specifies the **n**th enclosing loop to continue from.

Example

The following loop makes use of the **continue** statement which returns from the continue statement and starts processing the next statement –

Live Demo

```
#!/bin/sh

NUMS="1 2 3 4 5 6 7"

for NUM in $NUMS
do
   Q=`expr $NUM % 2`
   if [ $Q -eq 0 ]
   then
      echo "Number is an even number!!"
      continue
   fi
   echo "Found odd number"
done
```

Upon execution, you will receive the following result –

```
Found odd number
Number is an even number!!
Found odd number
Number is an even number!!
Found odd number
Number is an even number!!
Found odd number
```

What is Substitution?

The shell performs substitution when it encounters an expression that contains one or more special characters.

Example

Here, the printing value of the variable is substituted by its value. Same time, **"\n"** is substituted by a new line –

```
Value of a is 10
```

wing is the result without **-e** option –

```
Value of a is 10\n
```

The following escape sequences which can be used in echo command –

Sr.No.	Escape & Description
1	**** backslash
2	**\a** alert (BEL)
3	**\b** backspace
4	**\c** suppress trailing newline
5	**\f** form feed
6	**\n** new line
7	**\r** carriage return
8	**\t** horizontal tab
9	**\v** vertical tab

You can use the **-E** option to disable the interpretation of the backslash escapes (default).

You can use the **-n** option to disable the insertion of a new line.

Command Substitution

When performing the command substitution make sure that you use the backquote, not the single quote character.

Example

Command substitution is generally used to assign the output of a command to a variable. Each of the following examples demonstrates the command substitution –

Live Demo

```
#!/bin/sh

DATE=`date`
echo "Date is $DATE"

USERS=`who | wc -l`
echo "Logged in user are $USERS"

UP=`date ; uptime`
echo "Uptime is $UP"
```

Upon execution, you will receive the following result –

```
Date is Thu Jul  2 03:59:57 MST 2009
Logged in user are 1
Uptime is Thu Jul  2 03:59:57 MST 2009
03:59:57 up 20 days, 14:03,  1 user,  load avg: 0.13, 0.07, 0.15
```

Variable Substitution

Variable substitution enables the shell programmer to manipulate the value of a variable based on its state.

Here is the following table for all the possible substitutions –

Sr.No.	Form & Description
1	**${var}** Substitute the value of *var*.
2	**${var:-word}** If *var* is null or unset, *word* is substituted for **var**. The value of *var* does not change.
3	**${var:=word}** If *var* is null or unset, *var* is set to the value of **word**.

Example

Following is the example to show various states of the above substitution –


```
#!/bin/sh

echo ${var:-"Variable is not set"}
echo "1 - Value of var is ${var}"

echo ${var:="Variable is not set"}
echo "2 - Value of var is ${var}"

unset var
echo ${var:+"This is default value"}
echo "3 - Value of var is $var"

var="Prefix"
echo ${var:+"This is default value"}
echo "4 - Value of var is $var"

echo ${var:?"Print this message"}
echo "5 - Value of var is ${var}"
```

Upon execution, you will receive the following result –

```
Variable is not set
1 - Value of var is
Variable is not set
2 - Value of var is Variable is not set

3 - Value of var is
This is default value
4 - Value of var is Prefix
Prefix
5 - Value of var is Prefix
```


In this chapter, we will discuss in detail about the Shell quoting mechanisms. We will start by discussing the metacharacters.

The Metacharacters

Unix Shell provides various metacharacters which have special meaning while using them in any Shell Script and causes termination of a word unless quoted.

For example, **?** matches with a single character while listing files in a directory and an ***** matches more than one character. Here is a list of most of the shell special characters (also called metacharacters) –

Following example shows how to print a ' or a ; –


```
#!/bin/sh

echo Hello; Word
```

Upon execution, you will receive the following result –

```
Hello
./test.sh: line 2: Word: command not found

shell returned 127
```

Let us now try using a quoted character –

```
#!/bin/sh

echo Hello\; Word
```

Upon execution, you will receive the following result –

```
Hello; Word
```

The **$** sign is one of the metacharacters, so it must be quoted to avoid special handling by the shell –

```
#!/bin/sh

echo "I have \$1200"
```

Upon execution, you will receive the following result –

```
I have $1200
```

The following table lists the four forms of quoting –

Sr.No.	Quoting & Description
1	**Single quote** All special characters between these quotes lose their special meaning.
2	**Double quote** Most special characters between these quotes lose their special meaning with these exceptions –

Backslash

Any character immediately following the backslash loses its special meaning.

Back quote

4

Anything in between back quotes would be treated as a command and would be executed.

The Single Quotes

Consider an echo command that contains many special shell characters –

```
echo <-$1500.**>; (update?) [y|n]
```

Putting a backslash in front of each special character is tedious and makes the line difficult to read –

```
echo \<-\$1500.\*\*\>\; \(update\?\) \[y\|n\]
```

There is an easy way to quote a large group of characters. Put a single quote (') at the beginning and at the end of the string –

```
echo '<-$1500.**>; (update?) [y|n]'
```

Characters within single quotes are quoted just as if a backslash is in front of each character. With this, the echo command displays in a proper way.

If a single quote appears within a string to be output, you should not put the whole string within single quotes instead you should precede that using a backslash (\) as follows –

```
echo 'It\'s Shell Programming'
```

The Double Quotes

Try to execute the following shell script. This shell script makes use of single quote –

```
VAR=ZARA
echo '$VAR owes <-$1500.**>; [ as of (`date +%m/%d`) ]'
```

Upon execution, you will receive the following result –

57

```
$VAR owes <-$1500.**>; [ as of (`date +%m/%d`) ]
```

This is not what had to be displayed. It is obvious that single quotes prevent variable substitution. If you want to substitute variable values and to make inverted commas work as expected, then you would need to put your commands in double quotes as follows –

le quotes take away the special meaning of all characters except the following –

- $ for parameter substitution
- Backquotes for command substitution
- \$ to enable literal dollar signs
- \` to enable literal backquotes
- \" to enable embedded double quotes
- \\ to enable embedded backslashes
- All other \ characters are literal (not special)

Characters within single quotes are quoted just as if a backslash is in front of each character. This helps the echo command display properly.

If a single quote appears within a string to be output, you should not put the whole string within single quotes instead you should precede that using a backslash (\) as follows –

```
echo 'It\'s Shell Programming'
```

The Backquotes

Putting any Shell command in between **backquotes** executes the command.

Syntax

Here is the simple syntax to put any Shell **command** in between backquotes –

```
var=`command`
```

Example

The **date** command is executed in the following example and the produced result is stored in DATA variable.

Live Demo

```
DATE=`date`

echo "Current Date: $DATE"
```

Upon execution, you will receive the following result –

```
Current Date: Thu Jul  2 05:28:45 MST 2009
```


In this chapter, we will discuss in detail about the Shell input/output redirections. Most Unix system commands take input from your terminal and send the resulting output back to your terminal. A command normally reads its input from the standard input, which happens to be your terminal by default. Similarly, a command normally writes its output to standard output, which is again your terminal by default.

t command will be written to file instead of your terminal.

Check the following **who** command which redirects the complete output of the command in the users file.


```
$ who > users
```

Notice that no output appears at the terminal. This is because the output has been redirected from the default standard output device (the terminal) into the specified file. You can check the users file for the complete content –

```
$ cat users
oko          tty01    Sep 12 07:30
ai           tty15    Sep 12 13:32
ruth         tty21    Sep 12 10:10
pat          tty24    Sep 12 13:07
steve        tty25    Sep 12 13:03
$
```

If a command has its output redirected to a file and the file already contains some data, that data will be lost. Consider the following example –

```
$ echo line 1 > users
$ cat users
line 1
$
```

You can use >> operator to append the output in an existing file as follows –

```
$ echo line 2 >> users
$ cat users
line 1
line 2
$
```

Input Redirection

Just as the output of a command can be redirected to a file, so can the input of a command be redirected from a file. As the **greater-than character >** is used for output redirection, the **less-than character <** is used to redirect the input of a command.

The commands that normally take their input from the standard input can have their input redirected from a file in this manner. For example, to count the number of lines in the file *users* generated above, you can execute the command as follows –


```
$ wc -l users
2 users
$
```

Note that there is a difference in the output produced by the two forms of the wc command. In the first case, the name of the file users is listed with the line count; in the second case, it is not.

In the first case, wc knows that it is reading its input from the file users. In the second case, it only knows that it is reading its input from standard input so it does not display file name.

Here Document

A **here document** is used to redirect input into an interactive shell script or program.

We can run an interactive program within a shell script without user action by supplying the required input for the interactive program, or interactive shell script.

The general form for a **here** document is –

```
command << delimiter
document
delimiter
```

Here the shell interprets the **<<** operator as an instruction to read input until it finds a line containing the specified delimiter. All the input lines up to the line containing the delimiter are then fed into the standard input of the command.

The delimiter tells the shell that the **here** document has completed. Without it, the shell continues to read the input forever. The delimiter must be a single word that does not contain spaces or tabs.

Following is the input to the command **wc -l** to count the total number of lines –

```
$wc -l << EOF
   This is a simple lookup program
        for good (and bad) restaurants
        in Cape Town.
EOF
3
$
```

You can use the **here document** to print multiple lines using your script as follows –

```
#!/bin/sh

cat << EOF
This is a simple lookup program
for good (and bad) restaurants
in Cape Town.
EOF
```

```
in/sh
filename=test.txt
vi $filename <<EndOfCommands
i
This file was created automatically from
a shell script
^[
ZZ
EndOfCommands
```

If you run this script with vim acting as vi, then you will likely see output like the following –

```
$ sh test.sh
Vim: Warning: Input is not from a terminal
$
```

After running the script, you should see the following added to the file **test.txt** –

```
$ cat test.txt
This file was created automatically from
a shell script
$
```

Discard the output

Sometimes you will need to execute a command, but you don't want the output displayed on the screen. In such cases, you can discard the output by redirecting it to the file **/dev/null** –

```
$ command > /dev/null
```

Here command is the name of the command you want to execute. The file **/dev/null** is a special file that automatically discards all its input.

To discard both output of a command and its error output, use standard redirection to redirect **STDERR** to **STDOUT** –

```
$ command > /dev/null 2>&1
```

Here **2** represents **STDERR** and **1** represents **STDOUT**. You can display a message on to STDERR by redirecting STDOUT into STDERR as follows –

```
$ echo message 1>&2
```


Redirection Commands

pgm >> file: Output of pgm is appended to file

n > file: Output from stream with descriptor **n** redirected to file

n >> file: Output from stream with descriptor **n** appended to file

6 **n >& m:** Merges output from stream **n** with stream **m**

7 **n <& m:** Merges input from stream **n** with stream **m**

8 **<< tag:** Standard input comes from here through next tag at the start of line

|

9 Takes output from one program, or process, and sends it to another

Note that the file descriptor **0** is normally standard input (STDIN), **1** is standard output (STDOUT), and **2** is standard error output (STDERR).

UNIX PROGRAMMING-FREQUENTLY ASKED QUESTIONS

UNIT-I

1)Explain the structure of UNIX operating system with the help of neat diagram.(5M)
2)What is command substitution? What is the token for command substitution in the korn shell?(2M)
// What is command substitution? Give examples.(3M)
3)What is an operating system? Mention the characteristics of Unix operating system(2M)
4)Explain the unix kernel architecture
//What is kernel? Is it similar to the operating system? Explain
5)Explain different unix versions
6)Explain the following unix commands-who,lp,date,cat,echo,man
7)Explain about UNIX features?

UNIT-II

1)What is file? What are different types of files? Explain.
2)Explain the implementation details of UNIX file system
3)What are the possible file system security levels? How to change permissions of a file?(5M)
4)What is a directory? Explain Significance of Sticky bit permissions on a directory.(5M)
//What is meant by security? Explain the different levels of security provided by UNIX(5M)
5)Explain file handling utilities
6)Describe usage of cp and mv commands.(2M)
// Is cp is similar to mv? Explain with examples(3M)
7)Write syntax for changing ownership and group name on a given files(2M)
8)What would be the effect of the following commands:
a)mkdir /usr/local/src/bash/{old,new,dist,bugs} b)type printf
c)echo –e "\t\tHellothere\c" d)echo –n "Hello there"
9)What would be the effects of the following commands
a)cat users b)echo$PS1 c)mv x.c y d)ls
10)Explain the following unix commands-cp.mv,rm
11)Explain the following System Call with syntax-chown,chmod
12)Explain the following Directory API with example
i)umask ii)mkdir
13)What would be the effect of the following commands (15M)
a) mkdir -p works/xyz/unix/book b) rm -f –letter
c) ls -l /usr/bin/vi d) uname e) echo $PATH

UNIT-III

1)List out 'here' document and append redirection operators with example.(2M)
2)What is meant by redirection? Discuss about the different types of redirection.(5M)
3)Explain shell variables
4)Define redirection. Explain how input, output and error redirection is done?(8M)
//What is Redirection? Explain various commands used for redirection(7M)

UNIT-4

Filters:

In this chapter, we will discuss in detail about pipes and filters in Unix. You can connect two commands together so that the output from one program becomes the input of the next program. Two or more commands connected in this way form a pipe.To make a pipe, put a vertical bar (|) on the command line between two commands.When a program takes its input from another program, it performs some operation on that input, and writes the result to the standard output. It is referred to as a filter.

The grep Command

The grep command searches a file or files for lines that have a certain pattern.

The syntax is –$grep pattern file(s)

1.The name "grep" comes from the ed (a Unix line editor) command g/re/p which means “globally search for a regular expression and print all lines containing it”.

2.A regular expression is either some plain text (a word, for example) and/or special characters used for pattern matching.

The simplest use of grep is to look for a pattern consisting of a single word. It can be used in a pipe so that only those lines of the input files containing a given string are sent to the standard output. If you don't give grep a filename to read, it reads its standard input; that's the way all filter programs work –

EXAMPLES:

$ls -l | grep "Aug"

-rw-rw-rw- 1 john doc 11008 Aug 6 14:10 ch02

-rw-rw-rw- 1 john doc 8515 Aug 6 15:30 ch07

-rw-rw-r-- 1 john doc 2488 Aug 15 10:51 intro

-rw-rw-r-- 1 carol doc 1605 Aug 23 07:35 macros

$

There are various options which you can use along with the grep command –

Sr.No.	Option & Description
1	-v Prints all lines that do not match pattern.
2	-n Prints the matched line and its line number.

3 -lPrints only the names of files with matching lines (letter "l")

4 -cPrints only the count of matching lines.

5 -iMatches either upper or lowercase.

Let us now use a regular expression that tells grep to find lines with "carol", followed by zero or other characters abbreviated in a regular expression as ".*"), then followed by "Aug".–Here, we are using the -i option to have case insensitive search –

$ls -l | grep -i "carol.*aug"

-rw-rw-r-- 1 carol doc 1605 Aug 23 07:35 macros

$

The sort Command:The sort command arranges lines of text alphabetically or numerically. The following example sorts the lines in the food file –

$sort food

Afghani Cuisine

Bangkok Wok

Big Apple Deli

Isle of Java

Mandalay

Sushi and Sashimi

Sweet Tooth

Tio Pepe's Peppers

$

The sort command arranges lines of text alphabetically by default. There are many options that control the sorting –

Sr.No. Description

1 -nSorts numerically (example: 10 will sort after 2), ignores blanks and tabs.

2 -rReverses the order of sort.

3 -fSorts upper and lowercase together.

4 +xIgnores first x fields when sorting.

More than two commands may be linked up into a pipe. Taking a previous pipe example using grep, we can further sort the files modified in August by the order of size.

The following pipe consists of the commands ls, grep, and sort –

```
$ls -l | grep "Aug" | sort +4n
-rw-rw-r--  1 carol doc        1605 Aug 23 07:35 macros
-rw-rw-r--  1 john  doc        2488 Aug 15 10:51 intro
-rw-rw-rw-  1 john  doc        8515 Aug  6 15:30 ch07
-rw-rw-rw-  1 john  doc       11008 Aug  6 14:10 ch02
$
```

This pipe sorts all files in your directory modified in August by the order of size, and prints them on the terminal screen. The sort option +4n skips four fields (fields are separated by blanks) then sorts the lines in numeric order.

The pg and more Commands

A long output can normally be zipped by you on the screen, but if you run text through more or use the pg command as a filter; the display stops once the screen is full of text.

Let's assume that you have a long directory listing. To make it easier to read the sorted listing, pipe the output through more as follows –

```
$ls -l | grep "Aug" | sort +4n | more
-rw-rw-r--  1 carol doc        1605 Aug 23 07:35 macros
-rw-rw-r--  1 john  doc        2488 Aug 15 10:51 intro
-rw-rw-rw-  1 john  doc        8515 Aug  6 15:30 ch07
-rw-rw-r--  1 john  doc       14827 Aug  9 12:40 ch03
```

.

.

.

-rw-rw-rw- 1 john doc 16867 Aug 6 15:56 ch05

--More--(74%)

The screen will fill up once the screen is full of text consisting of lines sorted by the order of the file size. At the bottom of the screen is the more prompt, where you can type a command to move through the sorted text.Once you're done with this screen, you can use any of the commands listed in the discussion of the more program.

SED COMMAND

In this chapter, we will discuss in detail about regular expressions with SED in Unix.A regular expression is a string that can be used to describe several sequences of characters. Regular expressions are used by several different Unix commands, including ed, sed, awk, grep, and to a more limited extent, vi.Here SED stands for stream editor. This stream-oriented editor was created exclusively for executing scripts. Thus, all the input you feed into it passes through and goes to STDOUT and it does not change the input file.

Invoking sed:Before we start, let us ensure we have a local copy of /etc/passwd text file to work with sed.As mentioned previously, sed can be invoked by sending data through a pipe to it as follows –

$ cat /etc/passwd | sed

Usage: sed [OPTION]... {script-other-script} [input-file]...-n, --quiet, --silent

suppress automatic printing of pattern space -e script, --expression = script

The cat command dumps the contents of /etc/passwd to sed through the pipe into sed's pattern space. The pattern space is the internal work buffer that sed uses for its operations.

The sed General Syntax:

Following is the general syntax for sed –/pattern/action

Here, pattern is a regular expression, and action is one of the commands given in the following table. If pattern is omitted, action is performed for every line as we have seen above.The slash character (/) that surrounds the pattern are required because they are used as delimiters.

Sr.No.	Range & Description
1	pPrints the line
2	dDeletes the line
3	s/pattern1/pattern2/Substitutes the first occurrence of pattern1 with pattern2

Deleting All Lines with sed

We will now understand how to delete all lines with sed. Invoke sed again; but the sed is now supposed to use the editing command delete line, denoted by the single letter d −

$ cat /etc/passwd | sed 'd'

$

Instead of invoking sed by sending a file to it through a pipe, the sed can be instructed to read the data from a file, as in the following example.

The following command does exactly the same as in the previous example, without the cat command −$ sed -e 'd' /etc/passwd

$The sed AddressesThe sed also supports addresses. Addresses are either particular locations in a file or a range where a particular editing command should be applied. When the sed encounters no addresses, it performs its operations on every line in the file.

The following command adds a basic address to the sed command you've been using −

$ cat /etc/passwd | sed '1d' |more

daemon:x:1:1:daemon:/usr/sbin:/bin/sh

bin:x:2:2:bin:/bin:/bin/sh

sys:x:3:3:sys:/dev:/bin/sh

sync:x:4:65534:sync:/bin:/bin/sync

games:x:5:60:games:/usr/games:/bin/sh

man:x:6:12:man:/var/cache/man:/bin/sh

mail:x:8:8:mail:/var/mail:/bin/sh

backup:x:34:34:backup:/var/backups:/bin/sh

$

Notice that the number 1 is added before the delete edit command. This instructs the sed to perform the editing command on the first line of the file. In this example, the sed will delete the first line of /etc/password and print the rest of the file.

The sed Address Ranges"We will now understand how to work with the sed address ranges. So what if you want to remove more than one line from a file? You can specify an address range with sed as follows −

$ cat /etc/passwd | sed '1, 5d' |more

games:x:5:60:games:/usr/games:/bin/sh

man:x:6:12:man:/var/cache/man:/bin/sh

mail:x:8:8:mail:/var/mail:/bin/sh

backup:x:34:34:backup:/var/backups:/bin/sh

$The above command will be applied on all the lines starting from 1 through 5. This deletes the first five lines.

ot.

Replacing with Empty Space

Use an empty substitution string to delete the root string from the /etc/passwd file entirely −

$ cat /etc/passwd | sed 's/root//g'

:x:0:0::/:/bin/sh

daemon:x:1:1:daemon:/usr/sbin:/bin/sh

Address Substitution

If you want to substitute the string sh with the string quiet only on line 10, you can specify it as follows −

$ cat /etc/passwd | sed '10s/sh/quiet/g'

root:x:0:0:root user:/root:/bin/sh

daemon:x:1:1:daemon:/usr/sbin:/bin/sh

bin:x:2:2:bin:/bin:/bin/sh

```
sys:x:3:3:sys:/dev:/bin/sh

sync:x:4:65534:sync:/bin:/bin/sync

games:x:5:60:games:/usr/games:/bin/sh

man:x:6:12:man:/var/cache/man:/bin/sh

mail:x:8:8:mail:/var/mail:/bin/sh

backup:x:34:34:backup:/var/backups:/bin/quiet
```

Similarly, to do an address range substitution, you could do something like the following −

```
$ cat /etc/passwd | sed '1,5s/sh/quiet/g'

root:x:0:0:root user:/root:/bin/quiet

daemon:x:1:1:daemon:/usr/sbin:/bin/quiet

bin:x:2:2:bin:/bin:/bin/quiet

sys:x:3:3:sys:/dev:/bin/quiet

sync:x:4:65534:sync:/bin:/bin/sync

games:x:5:60:games:/usr/games:/bin/sh

man:x:6:12:man:/var/cache/man:/bin/sh

mail:x:8:8:mail:/var/mail:/bin/sh

backup:x:34:34:backup:/var/backups:/bin/sh
```


As you can see from the output, the first five lines had the string sh changed to quiet, but the rest of the lines were left untouched.

The Matching Command

You would use the p option along with the -n option to print all the matching lines as follows −

```
$ cat testing | sed -n '/root/p'
root:x:0:0:root user:/root:/bin/sh
[root@ip-72-167-112-17 amrood]# vi testing
root:x:0:0:root user:/root:/bin/sh
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/bin/sh
man:x:6:12:man:/var/cache/man:/bin/sh
mail:x:8:8:mail:/var/mail:/bin/sh
news:x:9:9:news:/var/spool/news:/bin/sh
backup:x:34:34:backup:/var/backups:/bin/sh
```

Using Regular Expression

While matching patterns, you can use the regular expression which provides more flexibility.

Check the following example which matches all the lines starting with daemon and then deletes them −

```
$ cat testing | sed '/^daemon/d'
root:x:0:0:root user:/root:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
```

sync:x:4:65534:sync:/bin:/bin/sync

games:x:5:60:games:/usr/games:/bin/sh

man:x:6:12:man:/var/cache/man:/bin/sh

mail:x:8:8:mail:/var/mail:/bin/sh

news:x:9:9:news:/var/spool/news:/bin/sh

backup:x:34:34:backup:/var/backups:/bin/sh

Following is the example which deletes all the lines ending with sh −

$ cat testing | sed '/sh$/d'

sync:x:4:65534:sync:/bin:/bin/sync

The following table lists four special characters that are very useful in regular expressions.

Sr.No. Character & Description

1

^

Matches the beginning of lines

2

$

Matches the end of lines

3

.

Matches any single character

4

*

Matches zero or more occurrences of the previous character

5

[chars]

Matches any one of the characters given in chars, where chars is a sequence of characters. You can use the - character to indicate a range of characters.

Matching Characters

Look at a few more expressions to demonstrate the use of metacharacters. For example, the following pattern –

Sr.No. Expression & Description

1

/a.c/

Matches lines that contain strings such as a+c, a-c, abc, match, and a3c

2

/a*c/

Matches the same strings along with strings such as ace, yacc, and arctic

3

/[tT]he/

Matches the string The and the

4

/^$/

Matches blank lines

5

/^.*$/

Matches an entire line whatever it is

6

/ */

Matches one or more spaces

7

/^$/

Matches blank lines

Following table shows some frequently used sets of characters −

Sr.No. Set & Description

1

[a-z]

Matches a single lowercase letter

2

[A-Z]

Matches a single uppercase letter

3

[a-zA-Z]

Matches a single letter

4

[0-9]

Matches a single number

5

[a-zA-Z0-9]

Matches a single letter or number

Character Class Keywords

Some special keywords are commonly available to regexps, especially GNU utilities that employ regexps. These are very useful for sed regular expressions as they simplify things and enhance readability.

For example, the characters a through z and the characters A through Z, constitute one such class of characters that has the keyword [[:alpha:]]

Using the alphabet character class keyword, this command prints only those lines in the /etc/syslog.conf file that start with a letter of the alphabet –


```
$ cat /etc/syslog.conf | sed -n '/^[[:alpha:]]/p'
authpriv.*                      /var/log/secure
mail.*                          -/var/log/maillog
cron.*                          /var/log/cron
uucp,news.crit                  /var/log/spooler
local7.*                        /var/log/boot.log
```

The following table is a complete list of the available character class keywords in GNU sed.

Sr.No. Character Class & Description

1

[[:alnum:]]

Alphanumeric [a-z A-Z 0-9]

2

[[:alpha:]]

Alphabetic [a-z A-Z]

3

[[:blank:]]

Blank characters (spaces or tabs)

4

[[:cntrl:]]

Control characters

5

[[:digit:]]

Numbers [0-9]

6

[[:graph:]]

Any visible characters (excludes whitespace)

7

[[:lower:]]

Lowercase letters [a-z]

8

[[:print:]]

Printable characters (non-control characters)

9

[[:punct:]]

Punctuation characters

10

[[:space:]]

Whitespace

11

[[:upper:]]

Uppercase letters [A-Z]

12

[[:xdigit:]]

Hex digits [0-9 a-f A-F]

Aampersand Referencing

The sed metacharacter & represents the contents of the pattern that was matched. For instance, say you have a file called phone.txt full of phone numbers, such as the following −

5555551212

5555551213

5555551214

6665551215

6665551216

7775551217

You want to make the area code (the first three digits) surrounded by parentheses for easier reading. To do this, you can use the ampersand replacement character −

$ sed -e 's/^[[:digit:]][[:digit:]][[:digit:]]/(&)/g' phone.txt

(555)5551212

(555)5551213

(555)5551214

(666)5551215

(666)5551216

(777)5551217

Here in the pattern part you are matching the first 3 digits and then using & you are replacing those 3 digits with the surrounding parentheses.

Using Multiple sed Commands

You can use multiple sed commands in a single sed command as follows −

$ sed -e 'command1' -e 'command2' ... -e 'commandN' files

Here command1 through commandN are sed commands of the type discussed previously. These commands are applied to each of the lines in the list of files given by files.

Using the same mechanism, we can write the above phone number example as follows −

```
$ sed -e 's/^[[:digit:]]\{3\}/(&)/g'    \
    -e 's/)[[:digit:]]\{3\}/&-/g' phone.txt
```

(555)555-1212

(555)555-1213

(555)555-1214

(666)555-1215

(666)555-1216

(777)555-1217

Note − In the above example, instead of repeating the character class keyword [[:digit:]] three times, we replaced it with \{3\}, which means the preceding regular expression is matched three times. We have also used \ to give line break and this has to be removed before the command is run.

Back References

The ampersand metacharacter is useful, but even more useful is the ability to define specific regions in regular expressions. These special regions can be used as reference in your replacement strings. By defining specific parts of a regular expression, you can then refer back to those parts with a special reference character.

To do back references, you have to first define a region and then refer back to that region. To define a region, you insert backslashed parentheses around each region of interest. The first region that you surround with backslashes is then referenced by \1, the second region by \2, and so on.

Assuming phone.txt has the following text −

(555)555-1212

(555)555-1213

(555)555-1214

(666)555-1215

(666)555-1216

(777)555-1217

Try the following command −

```
$ cat phone.txt | sed 's/\(.*)\)\(.*-\)\(.*$\)/Area \
    code: \1 Second: \2 Third: \3/'
Area code: (555) Second: 555- Third: 1212
Area code: (555) Second: 555- Third: 1213
Area code: (555) Second: 555- Third: 1214
Area code: (666) Second: 555- Third: 1215
Area code: (666) Second: 555- Third: 1216
Area code: (777) Second: 555- Third: 1217
```

Note − In the above example, each regular expression inside the parenthesis would be back referenced by \1, \2 and so on. We have used \ to give line break here. This should be removed before running the command.

UNIT-5

A shell script is a computer program designed to be run by the Unix/Linux shell which could be one of the following:

The Bourne Shell
The C Shell
The Korn Shell
The GNU Bourne-Again Shell

A shell is a command-line interpreter and typical operations performed by shell scripts include file manipulation, program execution, and printing text.
Extended Shell Scripts

Shell scripts have several required constructs that tell the shell environment what to do and when to do it. Of course, most scripts are more complex than the above one.

The shell is, after all, a real programming language, complete with variables, control structures, and so forth. No matter how complicated a script gets, it is still just a list of commands executed sequentially.

The following script uses the read command which takes the input from the keyboard and assigns it as the value of the variable PERSON and finally prints it on STDOUT.


```
#!/bin/sh

# Author : Zara Ali
# Copyright (c) Tutorialspoint.com
# Script follows here:

echo "What is your name?"
read PERSON
echo "Hello, $PERSON"
```

Here is a sample run of the script ?

```
$./test.sh
What is your name?
Zara Ali
Hello, Zara Ali
$
```

Subsequent part of this tutorial will cover Unix/Linux Shell Scripting in detail.

There are various operators supported by each shell. We will discuss in detail about Bourne shell (default shell) in this chapter.

We will now discuss the following operators −

- Arithmetic Operators
- Relational Operators
- Boolean Operators
- String Operators
- File Test Operators

Bourne shell didn't originally have any mechanism to perform simple arithmetic operations but it uses external programs, either awk or expr.

The following example shows how to add two numbers −
Live Demo

```
#!/bin/sh

val=`expr 2 + 2`
echo "Total value : $val"
```

The above script will generate the following result −

```
Total value : 4
```

The following points need to be considered while adding −

- There must be spaces between operators and expressions. For example, 2+2 is not correct; it should be written as 2 + 2.

- The complete expression should be enclosed between ‘ ‘, called the backtick.

Arithmetic Operators

The following arithmetic operators are supported by Bourne Shell.

Assume variable a holds 10 and variable b holds 20 then −

Show Examples

Operator	Description	Example
+ (Addition)	Adds values on either side of the operator	`expr $a + $b` will give 30
- (Subtraction)	Subtracts right hand operand from left hand operand	`expr

$a - $b` will give -10
* (Multiplication) Multiplies values on either side of the operator `expr $a \* $b` will give 200
/ (Division) Divides left hand operand by right hand operand`expr $b / $a` will give 2
% (Modulus) Divides left hand operand by right hand operand and returns remainder `expr $b % $a` will give 0
= (Assignment) Assigns right operand in left operand a = $b would assign value of b into a
== (Equality) Compares two numbers, if both are same then returns true. [$a == $b] would return false.
!= (Not Equality) Compares two numbers, if both are different then returns true. [$a != $b] would return true.

It is very important to understand that all the conditional expressions should be inside square braces with spaces around them, for example [$a == $b] is correct whereas, [$a==$b] is incorrect.

All the arithmetical calculations are done using long integers.
Relational Operators

Bourne Shell supports the following relational operators that are specific to numeric values. These operators do not work for string values unless their value is numeric.

For example, following operators will work to check a relation between 10 and 20 as well as in between "10" and "20" but not in between "ten" and "twenty".

Assume variable a holds 10 and variable b holds 20 then −

Show Examples
Operator Description Example
-eq Checks if the value of two operands are equal or not; if yes, then the condition becomes true. [$a -eq $b] is not true.
-ne Checks if the value of two operands are equal or not; if values are not equal, then the condition becomes true. [$a -ne $b] is true.
-gt Checks if the value of left operand is greater than the value of right operand; if yes, then the condition becomes true. [$a -gt $b] is not true.
-lt Checks if the value of left operand is less than the value of right operand; if yes, then the condition becomes true. [$a -lt $b] is true.
-ge Checks if the value of left operand is greater than or equal to the value of right operand; if yes, then the condition becomes true. [$a -ge $b] is not true.
-le Checks if the value of left operand is less than or equal to the value of right operand; if yes, then the condition becomes true. [$a -le $b] is true.

It is very important to understand that all the conditional expressions should be placed inside square braces with spaces around them. For example, [$a <= $b] is correct whereas, [$a <= $b] is incorrect.

Boolean Operators

The following Boolean operators are supported by the Bourne Shell.

Assume variable a holds 10 and variable b holds 20 then −

Show Examples

Operator	Description	Example
!	This is logical negation. This inverts a true condition into false and vice versa.	[! false] is true.
-o	This is logical OR. If one of the operands is true, then the condition becomes true.	[$a -lt 20 -o $b -gt 100] is true.
-a	This is logical AND. If both the operands are true, then the condition becomes true otherwise false.	[$a -lt 20 -a $b -gt 100] is false.

String Operators

The following string operators are supported by Bourne Shell.

Assume variable a holds "abc" and variable b holds "efg" then −

Show Examples

Operator	Description	Example
=	Checks if the value of two operands are equal or not; if yes, then the condition becomes true.	[$a = $b] is not true.
!=	Checks if the value of two operands are equal or not; if values are not equal then the condition becomes true.	[$a != $b] is true.
-z	Checks if the given string operand size is zero; if it is zero length, then it returns true.	[-z $a] is not true.
-n	Checks if the given string operand size is non-zero; if it is nonzero length, then it returns true.	[-n $a] is not false.
str	Checks if str is not the empty string; if it is empty, then it returns false.	[$a] is not false.

File Test Operators

We have a few operators that can be used to test various properties associated with a Unix file.

Assume a variable file holds an existing file name "test" the size of which is 100 bytes and has read, write and execute permission on −

Show Examples

Operator	Description	Example
-b file	Checks if file is a block special file; if yes, then the condition	

becomes true. [-b $file] is false.
-c file Checks if file is a character special file; if yes, then the condition becomes true. [-c $file] is false.
-d file Checks if file is a directory; if yes, then the condition becomes true. [-d $file] is not true.
-f file Checks if file is an ordinary file as opposed to a directory or special file; if yes, then the condition becomes true. [-f $file] is true.
-g file Checks if file has its set group ID (SGID) bit set; if yes, then the condition becomes true. [-g $file] is false.
-k file Checks if file has its sticky bit set; if yes, then the condition becomes true. [-k $file] is false.
-p file Checks if file is a named pipe; if yes, then the condition becomes true. [-p $file] is false.
-t file Checks if file descriptor is open and associated with a terminal; if yes, then the condition becomes true. [-t $file] is false.
-u file Checks if file has its Set User ID (SUID) bit set; if yes, then the condition becomes true. [-u $file] is false.
-r file Checks if file is readable; if yes, then the condition becomes true. [-r $file] is true.
-w file Checks if file is writable; if yes, then the condition becomes true. [-w $file] is true.
-x file Checks if file is executable; if yes, then the condition becomes true. [-x $file] is true.
-s file Checks if file has size greater than 0; if yes, then condition becomes true. [-s $file] is true.
-e file Checks if file exists; is true even if file is a directory but exists. [-e $file] is true.

C Shell Operators

Following link will give you a brief idea on C Shell Operators −

C Shell Operators
Korn Shell Operators

Following link helps you understand Korn Shell Operators −

Korn Shell Operators

In this chapter, we will learn how to use Shell variables in Unix. A variable is a character string to which we assign a value. The value assigned could be a number, text, filename, device, or any other type of data.

A variable is nothing more than a pointer to the actual data. The shell enables you to create, assign, and delete variables.
Variable Names

The name of a variable can contain only letters (a to z or A to Z), numbers (0 to 9) or the underscore character (_).

By convention, Unix shell variables will have their names in UPPERCASE.

The following examples are valid variable names −

_ALI
TOKEN_A
VAR_1
VAR_2

Following are the examples of invalid variable names −

2_VAR
-VARIABLE
VAR1-VAR2
VAR_A!

The reason you cannot use other characters such as !, *, or - is that these characters have a special meaning for the shell.
Defining Variables

Variables are defined as follows −

variable_name=variable_value

For example −

NAME="Zara Ali"

The above example defines the variable NAME and assigns the value "Zara Ali" to it. Variables of this type are called scalar variables. A scalar variable can hold only one value at a time.

Shell enables you to store any value you want in a variable. For example −

VAR1="Zara Ali"
VAR2=100

Accessing Values

To access the value stored in a variable, prefix its name with the dollar sign ($) −

For example, the following script will access the value of defined variable NAME and print it on STDOUT −
Live Demo

```
#!/bin/sh

NAME="Zara Ali"
echo $NAME
```

The above script will produce the following value −

```
Zara Ali
```

Read-only Variables

Shell provides a way to mark variables as read-only by using the read-only command. After a variable is marked read-only, its value cannot be changed.

For example, the following script generates an error while trying to change the value of NAME −
Live Demo

```
#!/bin/sh

NAME="Zara Ali"
readonly NAME
NAME="Qadiri"
```

The above script will generate the following result −

```
/bin/sh: NAME: This variable is read only.
```

Unsetting Variables

Unsetting or deleting a variable directs the shell to remove the variable from the list of variables that it tracks. Once you unset a variable, you cannot access the stored value in the variable.

Following is the syntax to unset a defined variable using the unset command −

```
unset variable_name
```

The above command unsets the value of a defined variable. Here is a simple example that demonstrates how the command works −

```
#!/bin/sh

NAME="Zara Ali"
unset NAME
echo $NAME
```

The above example does not print anything. You cannot use the unset command to unset variables that are marked readonly.

Variable Types

When a shell is running, three main types of variables are present −

Local Variables − A local variable is a variable that is present within the current instance of the shell. It is not available to programs that are started by the shell. They are set at the command prompt.

Environment Variables − An environment variable is available to any child process of the shell. Some programs need environment variables in order to function correctly. Usually, a shell script defines only those environment variables that are needed by the programs that it runs.

Shell Variables − A shell variable is a special variable that is set by the shell and is required by the shell in order to function correctly. Some of these variables are environment variables whereas others are local variables.

In this chapter, we will discuss in detail about special variable in Unix. In one of our previous chapters, we understood how to be careful when we use certain nonalphanumeric characters in variable names. This is because those characters are used in the names of special Unix variables. These variables are reserved for specific functions.

For example, the $ character represents the process ID number, or PID, of the current shell −

```
$echo $$
```

The above command writes the PID of the current shell −

```
29949
```

The following table shows a number of special variables that you can use in your shell scripts

Sr.No. Variable & Description

1

$0

The filename of the current script.
2

$n

These variables correspond to the arguments with which a script was invoked. Here n is a positive decimal number corresponding to the position of an argument (the first argument is $1, the second argument is $2, and so on).
3

$#

The number of arguments supplied to a script.
4

$*

All the arguments are double quoted. If a script receives two arguments, $* is equivalent to $1 $2.
5

$@

All the arguments are individually double quoted. If a script receives two arguments, $@ is equivalent to $1 $2.
6

$?

The exit status of the last command executed.
7

$$

The process number of the current shell. For shell scripts, this is the process ID under which they are executing.
8

$!

The process number of the last background command.
Command-Line Arguments

The command-line arguments $1, $2, $3, ...$9 are positional parameters, with $0 pointing to the actual command, program, shell script, or function and $1, $2, $3, ...$9 as the arguments to the command.

Following script uses various special variables related to the command line −

```
#!/bin/sh

echo "File Name: $0"
echo "First Parameter : $1"
echo "Second Parameter : $2"
echo "Quoted Values: $@"
echo "Quoted Values: $*"
echo "Total Number of Parameters : $#"
```

Here is a sample run for the above script −

```
$./test.sh Zara Ali
File Name : ./test.sh
First Parameter : Zara
Second Parameter : Ali
Quoted Values: Zara Ali
Quoted Values: Zara Ali
Total Number of Parameters : 2
```

Special Parameters $* and $@

There are special parameters that allow accessing all the command-line arguments at once. $* and $@ both will act the same unless they are enclosed in double quotes, "".

Both the parameters specify the command-line arguments. However, the "$*" special parameter takes the entire list as one argument with spaces between and the "$@" special parameter takes the entire list and separates it into separate arguments.

We can write the shell script as shown below to process an unknown number of commandline arguments with either the $* or $@ special parameters −

```
#!/bin/sh

for TOKEN in $*
do
    echo $TOKEN
done
```

Here is a sample run for the above script −

```
$./test.sh Zara Ali 10 Years Old
Zara
Ali
10
Years
Old
```

Note − Here do...done is a kind of loop that will be covered in a subsequent tutorial.

Exit Status

The $? variable represents the exit status of the previous command.

Exit status is a numerical value returned by every command upon its completion. As a rule, most commands return an exit status of 0 if they were successful, and 1 if they were unsuccessful.

Some commands return additional exit statuses for particular reasons. For example, some commands differentiate between kinds of errors and will return various exit values depending on the specific type of failure.

Following is the example of successful command −

```
$./test.sh Zara Ali
File Name : ./test.sh
First Parameter : Zara
Second Parameter : Ali
Quoted Values: Zara Ali
Quoted Values: Zara Ali
Total Number of Parameters : 2
$echo $?
0
$
```


In this chapter, we will discuss how to use shell arrays in Unix. A shell variable is capable enough to hold a single value. These variables are called scalar variables.

Shell supports a different type of variable called an array variable. This can hold multiple values at the same time. Arrays provide a method of grouping a set of variables. Instead of creating a new name for each variable that is required, you can use a single array variable that stores all the other variables.

All the naming rules discussed for Shell Variables would be applicable while naming arrays.

Defining Array Values

The difference between an array variable and a scalar variable can be explained as follows.

Suppose you are trying to represent the names of various students as a set of variables. Each of the individual variables is a scalar variable as follows −

```
NAME01="Zara"
NAME02="Qadir"
NAME03="Mahnaz"
NAME04="Ayan"
NAME05="Daisy"
```

We can use a single array to store all the above mentioned names. Following is the simplest method of creating an array variable. This helps assign a value to one of its indices.

```
array_name[index]=value
```

Here array_name is the name of the array, index is the index of the item in the array that you want to set, and value is the value you want to set for that item.

As an example, the following commands −

```
NAME[0]="Zara"
NAME[1]="Qadir"
NAME[2]="Mahnaz"
NAME[3]="Ayan"
NAME[4]="Daisy"
```


If you are using the ksh shell, here is the syntax of array initialization −

```
set -A array_name value1 value2 ... valuen
```

If you are using the bash shell, here is the syntax of array initialization −

```
array_name=(value1 ... valuen)
```

Accessing Array Values

After you have set any array variable, you access it as follows −

```
${array_name[index]}
```

Here array_name is the name of the array, and index is the index of the value to be accessed. Following is an example to understand the concept −
Live Demo

```
#!/bin/sh

NAME[0]="Zara"
NAME[1]="Qadir"
NAME[2]="Mahnaz"
NAME[3]="Ayan"
NAME[4]="Daisy"
echo "First Index: ${NAME[0]}"
echo "Second Index: ${NAME[1]}"
```

The above example will generate the following result −

```
$./test.sh
First Index: Zara
Second Index: Qadir
```

You can access all the items in an array in one of the following ways −

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

Here array_name is the name of the array you are interested in. Following example will help you understand the concept −
Live Demo

```
#!/bin/sh

NAME[0]="Zara"
NAME[1]="Qadir"
NAME[2]="Mahnaz"
NAME[3]="Ayan"
NAME[4]="Daisy"
echo "First Method: ${NAME[*]}"
echo "Second Method: ${NAME[@]}"
```

The above example will generate the following result −

```
$./test.sh
First Method: Zara Qadir Mahnaz Ayan Daisy
Second Method: Zara Qadir Mahnaz Ayan Daisy
```

In this chapter, we will understand shell decision-making in Unix. While writing a shell script, there may be a situation when you need to adopt one path out of the given two paths. So you need to make use of conditional statements that allow your program to make correct decisions and perform the right actions.

Unix Shell supports conditional statements which are used to perform different actions based on different conditions. We will now understand two decision-making statements here −

The if...else statement

The case...esac statement

The if...else statements

If else statements are useful decision-making statements which can be used to select an option from a given set of options.

Unix Shell supports following forms of if...else statement −

if...fi statement
if...else...fi statement
if...elif...else...fi statement

Most of the if statements check relations using relational operators discussed in the previous chapter.
The case...esac Statement

You can use multiple if...elif statements to perform a multiway branch. However, this is not always the best solution, especially when all of the branches depend on the value of a single variable.

Unix Shell supports case...esac statement which handles exactly this situation, and it does so more efficiently than repeated if...elif statements.

There is only one form of case...esac statement which has been described in detail here −

case...esac statement

The case...esac statement in the Unix shell is very similar to the switch...case statement we have in other programming languages like C or C++ and PERL, etc

In this chapter, we will discuss shell loops in Unix. A loop is a powerful programming tool that enables you to execute a set of commands repeatedly. In this chapter, we will examine the following types of loops available to shell programmers −

The while loop
The for loop
The until loop
The select loop

You will use different loops based on the situation. For example, the while loop executes the given commands until the given condition remains true; the until loop executes until a given condition becomes true.

Once you have good programming practice you will gain the expertise and thereby, start using appropriate loop based on the situation. Here, while and for loops are available in most of the other programming languages like C, C++ and PERL, etc.

Nesting Loops

All the loops support nesting concept which means you can put one loop inside another similar one or different loops. This nesting can go up to unlimited number of times based on your requirement.

Here is an example of nesting while loop. The other loops can be nested based on the programming requirement in a similar way −

Nesting while Loops

It is possible to use a while loop as part of the body of another while loop.

Syntax

```
while command1 ; # this is loop1, the outer loop
do
    Statement(s) to be executed if command1 is true

    while command2 ; # this is loop2, the inner loop
    do
        Statement(s) to be executed if command2 is true
    done

    Statement(s) to be executed if command1 is true
done
```

Example

Here is a simple example of loop nesting. Let's add another countdown loop inside the loop that you used to count to nine −

```
#!/bin/sh

a=0
while [ "$a" -lt 10 ]       # this is loop1
do
    b="$a"
    while [ "$b" -ge 0 ]    # this is loop2
    do
        echo -n "$b "
        b=`expr $b - 1`
    done
    echo
    a=`expr $a + 1`
done
```

This will produce the following result. It is important to note how echo -n works here. Here -n option lets echo avoid printing a new line character.

```
0
1 0
2 1 0
3 2 1 0
4 3 2 1 0
5 4 3 2 1 0
6 5 4 3 2 1 0
7 6 5 4 3 2 1 0
8 7 6 5 4 3 2 1 0
9 8 7 6 5 4 3 2 1 0
```


In this chapter, we will discuss shell loop control in Unix. So far you have looked at creating loops and working with loops to accomplish different tasks. Sometimes you need to stop a loop or skip iterations of the loop.

In this chapter, we will learn following two statements that are used to control shell loops−

The break statement

The continue statement

The infinite Loop

All the loops have a limited life and they come out once the condition is false or true depending on the loop.

A loop may continue forever if the required condition is not met. A loop that executes forever without terminating executes for an infinite number of times. For this reason, such loops are called infinite loops.
Example

Here is a simple example that uses the while loop to display the numbers zero to nine −

```
#!/bin/sh

a=10

until [ $a -lt 10 ]
do
    echo $a
    a=expr $a + 1`
done
```


This loop continues forever because a is always greater than or equal to 10 and it is never less than 10.
The break Statement

The break statement is used to terminate the execution of the entire loop, after completing the execution of all of the lines of code up to the break statement. It then steps down to the code following the end of the loop.
Syntax

The following break statement is used to come out of a loop −

```
break
```

The break command can also be used to exit from a nested loop using this format −

```
break n
```

Here n specifies the nth enclosing loop to the exit from.
Example

Here is a simple example which shows that loop terminates as soon as a becomes 5 −

```
#!/bin/sh

a=0

while [ $a -lt 10 ]
do
   echo $a
   if [ $a -eq 5 ]
   then
      break
   fi
   a=`expr $a + 1`
done
```

Upon execution, you will receive the following result −

```
0
1
2
3
4
5
```

Here is a simple example of nested for loop. This script breaks out of both loops if var1 equals 2 and var2 equals 0 −
Live Demo


```
#!/bin/sh

for var1 in 1 2 3
do
   for var2 in 0 5
   do
      if [ $var1 -eq 2 -a $var2 -eq 0 ]
      then
         break 2
      else
         echo "$var1 $var2"
      fi
   done
```

done

Upon execution, you will receive the following result. In the inner loop, you have a break command with the argument 2. This indicates that if a condition is met you should break out of outer loop and ultimately from the inner loop as well.

```
1 0
1 5
```

The continue statement

The continue statement is similar to the break command, except that it causes the current iteration of the loop to exit, rather than the entire loop.

This statement is useful when an error has occurred but you want to try to execute the next iteration of the loop.
Syntax

```
continue
```

Like with the break statement, an integer argument can be given to the continue command to skip commands from nested loops.

```
continue n
```

Here n specifies the nth enclosing loop to continue from.
Example

The following loop makes use of the continue statement which returns from the continue statement and starts processing the next statement −
Live Demo

```
#!/bin/sh

NUMS="1 2 3 4 5 6 7"

for NUM in $NUMS
do
    Q=`expr $NUM % 2`
    if [ $Q -eq 0 ]
    then
        echo "Number is an even number!!"
        continue
    fi
    echo "Found odd number"
done
```

Upon execution, you will receive the following result −

```
Found odd number
Number is an even number!!
Found odd number
Number is an even number!!
Found odd number
Number is an even number!!
Found odd number
```

In this chapter, we will discuss in detail about process management in Unix. 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.

Whenever you issue a command in Unix, it creates, or starts, a new process. When you tried out the ls command to list the directory contents, you started a process. A process, in simple terms, is an instance of a running program.

The operating system tracks processes through a five-digit ID number known as the pid or the process ID. Each process in the system has a unique pid.

Pids eventually repeat because all the possible numbers are used up and the next pid rolls or starts over. At any point of time, no two processes with the same pid exist in the system because it is the pid that Unix uses to track each process.

Starting a Process

When you start a process (run a command), there are two ways you can run it −

- Foreground Processes
- Background Processes

Foreground Processes

By default, every process that you start runs in the foreground. It gets its input from the keyboard and sends its output to the screen.

You can see this happen with the ls command. If you wish to list all the files in your current directory, you can use the following command −

UNIT-6

Processes in Unix

As part of process management, first need to know how to create processes.

Fork

In Unix the system call fork creates new processes. Fork has the following semantics:

• it creates an exact copy of the forking process

• it returns:

– an error (-1) if unsuccessful; the global variable errno gives the specific failure

– 0 to the child process

– process id of child to the parent

• child does not share any memory with the parent

• child and parent share open file descriptors

• the child is said to inherit its environment from its parent.

690 CASE STUDY 1: UNIX AND LINUX CHAP. 10

10.3 PROCESSES IN UNIX

In the previous sections, we started out by looking at UNIX as viewed fromthe keyboard, that is, what the user sees at the terminal. We gave examples ofshell commands and utility programs that are frequently used. We ended with abrief overview of the system structure. Now it is time to dig deeply into the kernel and look more closely at the basic concepts UNIX supports, namely, processes,memory, the file system, and input/output. These notions are important becausethe system calls—the interface to the operating system itself—manipulate them.

For example, system calls exist to create processes, allocate memory, open files,and do I/O.Unfortunately, with so many versions of UNIX in existence, there are somedifferences between

them. In this chapter, we will emphasize the features common to all of them rather than focus on any one specific version. Thus in certainsections (especially implementation sections), the discussion may not apply equally to every version.

10.3.1 Fundamental Concepts

The only active entities in a UNIX system are the processes. UNIX processesare very similar to the classical sequential processes that we studied in Chap 2.Each process runs a single program and initially has a single thread of control. Inother words, it has one program counter, which keeps track of the next instructionto be executed. Most versions of UNIX allow a process to create additionalthreads once it starts executing.UNIX is a multiprogramming system, so multiple, independent processes maybe running at the same time. Each user may have several active processes atonce, so on a large system, there may be hundreds or even thousands of processesrunning. In fact, on most single-user workstations, even when the user is absent,dozens of background processes, called daemons, are running. These are startedautomatically when the system is booted. (‘‘Daemon’’ is a variant spelling of‘‘demon,’’ which is a self-employed evil spirit.)A typical daemon is the cron daemon. It wakes up once a minute to check ifthere is any work for it to do. If so, it does the work. Then it goes back to sleepuntil it is time for the next check.This daemon is needed because it is possible in UNIX to schedule activitiesminutes, hours, days, or even months in the future. For example, suppose a userhas a dentist appointment at 3 o’clock next Tuesday. He can make an entry in thecron daemon’s database telling the daemon to beep at him at, say, 2:30. When theappointed day and time arrives, the cron daemon sees that it has work to do, andstarts up the beeping program as a new process.The cron daemon is also used to start up periodic activities, such as making daily disk backups at 4 A.M., or reminding forgetful users every year on OctoberCS 3013 1 week1-proc.tex 21

Shell Arithmetic

• shell uses expr for integer arithmetic

o num + num addition

o num - num subtraction

o num * num multiplication

o num / num integer division

o num % num remainder

• real arithmetic

o can be kludged using bc or dc

o not pretty !

expr

The shell treats all shell variables as strings. We have to use the expr command (/bin/expr) to perform arithmetic in the shell. It takes two integer arguments and an operand and writes the result to standard output. The result from expr is usually assigned to a shell variable using command substitution.There must be spaces between the operand and the arguments. Note that since expr uses * as the multiplication operand it must be shielded in shell scripts (usually using the \ character) as the * has special meaning to the shell.

For example, here is a shell script which will multiply two arguments:-

```
#!/bin/sh

#multiply.expr - multiply

#first arg by second

Result=`expr $1 \* $2`

echo Result of $1 \* $2 is $Result
```

expr's arguments must be integers: if given non-integer arguments it will not perform the calculation. (The isanum example script can be used to determine if a given argument is a number - the constructs used in this script will be explained later in the course).

Real Arithmetic

expr only provides integer arithmetic. It is possible to implement real arithmetic, ie arithmetic performed on real numbers, using either the bc or dc arithmetic utilities. Though considered slightly outwith the scope of this course a division would be

performed as follows:-

```
#!/bin/sh

#divide.bc - divide first arg by

#second

Result=`echo " scale=3 ; $1 / $2 " | bc`

echo Result of $1 / $2 is $Result
```

Frequently asked questions

1 a) With a neat sketch, explain the architecture of UNIX operating system. [7]

b) Explain the following UNIX commands

(i) uname (ii) ls (iii) more (iv) cat [8]

2 a) Explain the implementation details of UNIX file system. [6]

b) Explain the following UNIX commands

(i) grep (ii) sort (iii) diff [9]

3 a) What is Redirection? Explain various commands used for redirection. [7]

b) What is a filter in UNIX? List out various filters in UNIX. Explain any two of them. [8]

4 a) Define the 'grep' family. Mention the primary difference between fgrep and the other two members of the grep family. [7]

b) Write a 'sed' command to

i) display the lines through 10 to 15 in a given text file

ii) replace the word 'UNIX' with 'LINUX' in a given text file [8]

5 a) Define an associative array and explain the steps in processing an associative array. [8]

b) With a neat diagram, describe an awk utility's view of a file and also explain the file buffers and record buffers of awk. [7]

6 a) What is an Environment variable? List out the environment variables that control the user environment in Korn Shell. [8]

b) Demonstrate the execution of Loop redirection with a suitable example. [7]

7 a) Explain the echo command in C shell. Demonstrate the use of C shell character codes for each command. [8]

b) What is the application of ‘eval’ command in C shell and also explain the execution of ‘eval’ command with suitable example. [7]

8 a) Explain the commands that are available in UNIX file system to change the permissions of a file. [7]

b) Explain the following directory API with example

(i) opendir (ii) readdir (iii) closedir