9. INTRODUCTION TO SHELL PROGRAMMING

All shell statements and Linux commands can be entered in the command line itself. When a group of commands has to be executed regularly, they are better stored in a file. All such files are called shell scripts, shell programs, or shell procedures. A shell script is an executable file which contains shell commands. The script act as a program by sequentially executing each command in the file. A scripting language consists of control structures, shell commands, variables and expressions.

Uses of shell script:

- System boot scripts.
- Customizing administrative tasks.
- Creating simple applications.
- To kill or start multiple applications together.
- Data backup and creating snapshots.
- Monitoring your linux system.
- Whenever you find yourself doing the same task over and over again you should use shell scripting, i.e. repetitive task automation.

Shell special characters

We begin with the special set of characters that the shell uses to match filenames. We have used commands with more than one filename as arguments (e.g. cat chap01 chap02). Often, you may need to enter multiple filename in a command line. The most obvious solution is to specify all the filename separately:

ls chap chap01 chap02 chap03 chap04 chapx chapy chapz

If filenames are similar (as above) ,we can use the facility offered by the shell of representing them by a single pattern. For instance, the pattern chap* represents all filenames beginning with chap. This pattern is framed with ordinary characters (like chap) and a metacharacters (like *) using well-defined rules. The pattern can then be used as an argument to the command, and the shell will expand it suitably before the command is executed. The metacharacters that are used to construct the generalized pattern for matching filenames belong to a category called wild-cards.

The * and ? : The metacharacters,*, is one of the characters of the shell s wild-card set. It matches any number of characters

#ls chap*

Chap chap01 chap02 chap03 chap04 chapx chapy chapz

The next wild-card is the ?, which matches a single character. When used with same string chap (chap?) the shell matches all five-character filenames beginning with chap. Appending another ? creates the pattern chap??, which matches six-character filename.

#ls chap?

chapx chapy chapz
#ls chap??
chap01 chap02 chap03

Matching the dot (.): If you want to list all hidden filenames in your directory having at least three characters after the dot, the dot must be matched explicitly.

#ls .???*
.bash_profile .exrc .netscape .profile

The character class: The character class comprises a set of characters enclosed by the rectangular brackets, [&], but it matches a single character in the class. The pattern [abcd] is a character class and it matches a single character a, b, c or d. This can be combined with any string or another wild-card expression, so selecting chap01, chap02 and chap04

#ls chap0 [124]
chap01 chap02 chap04
#ls chap[1-4] lists chap01, chap02, chap03 and chap04
#ls chap[x-z] lists chapx chapy chapz

Comments

Comments help to make your code more readable. They do not affect the o/p of your program. They are specially made for you to read. All comments in bash begin with the hash symbol: # , except for the first line (#!/bin/bash). The first line is not a comment. Any lines after the first line that begin with # are a comment.

```
#!/bin/bash
#this program counts from 1 to 10
for i in 1 2 3 4 5 6 7 8 9 10 ; do
echo $i
done
```

Command Separator [Semicolon]

The semicolon (;) is used as a command separator. You can run more than one command on a single line by using the command separator, placing the semicolon between each command.

#date; who am i
Mon Mar 27 19:15:41 PDT 2017
Root tty/s Mar 27 17:50

Escaping and Quoting

Escaping: Providing a \ (backslash) before the wild-card to remove its special meaning. For instance, in the pattern *, the \ tells the shell that the asterisk has to be matched literally instead of being interpreted as a metacharacters.

rm chap* doesn t remove chap1 chap2

The \ suppresses the wild-card nature of the *, thus preventing the shell from performing filename expansion on it. This feature is known as escaping.

Quoting: Enclosing the wild-card, or even the entire pattern, within quotes (like chap*). Anything within these quotes are left alone by the shell and not interpreted. There s another way to turn off the meaning of a metacharacters. When a command argument is enclosed in quotes, the meaning of all enclosed special characters is turned off.

#echo \ displays a \
rm chap* removes file chap*

The following example shows the protection of four special characters using single quotes:

#echo the character |, <,> and $ are also special .
The character |, <,> and $ are also special.

Command substitution

Command substitution is used to combine more than one command in a command line. The command construct makes available the output of command for assignment to a variable. This is also known as back quotes or back ticks . The shell enables one or more command arguments to be obtained from the standard o/p of another command. This feature is called command substitution.

#echo the today s date is `date`
The today s date is sat sep 7 19:01:59 IST 2002

You can use the two commands in a pipeline and then use the output as the argument to a third.

#echo there are `ls | wc - l` files in the current directory
There are 58 files in the current directory.

Command substitution is enabled when back quotes are used within double quotes. If you use single quotes, it s not.

Creating shell script

Following steps are required to write shell script:

1. Use any editor like vi or mcedit to write shell script.
   Syntax: vi script name.sh
   Examples: **vi pgm1.sh**
2. After writing shell script set execute permission for your script as follows
   Syntax: chmod permission your-script-name
   Examples: **#chmod +x your-script-name**
   #chmod +x pgm1.sh
3. Execute your script as
   Syntax: sh your-script-name
   . / Your s-script-name
   Examples: **#. /pgm1.sh**

Shell variables

The shell supports variables that are useful both in the command line and shell scripts. A variable assignment is of the form variable = value (no spaces around=), but its evaluation requires the $ as prefix to the variable name:-

#count = 5	no $ required for assignment
#echo $count	but needed for evaluation
5	

A variable can also be assigned the value of another variable:

#**total = $count**	Assigning a value to another variable
#echo $total	
5	

All shell variables are initialized to null strings by default. While explicit assignment of null strings with x= or x= is possible, you can also use this as a shorthand.

X=	a null string

A variable can be removed with unset and protected from reassignment by read-only. Both are shell internal commands.

Unset x	x is now undefined

Read-only x x can t be reassigned

Command line arguments-Positional parameters

Shell procedures accept arguments in another situation too-when you specify them in the command line itself. When arguments are specified with a shell procedure, they are assigned to certain special variables , or rather positional parameters. The first argument is read by the shell into the parameter **$1**, the second argument into **$2** and so on. You can t technically call them shell variables because all variables are evaluated with a **$** before the variable name. In the case of **$1**, you really don t have a variable 1 that is evaluated with **$**.

```
#cat emp2.sh
#!/bin/sh
#
echo  Program: $0                          #$0 contains the program name
The number of arguments specified is $#
The arguments are $*                       #All arguments stored in $*
grep  $1  $2
echo  \nJob Over
```

The parameter $* stores the complete set of positional parameters as a single string.
The parameter $# is set to the number of arguments specified. This lets you design scripts that check whether the right numbers of arguments have been entered.
The parameter $0 holds the command name itself.
Invoke this script with the pattern director and the filename emp1.lst as the two arguments:

```
#emp2.sh director emp1.lst
Program: emp2.sh
The number of arguments specified is 2
The arguments are director emp1.lst
1006|chanchal singhvi  |director|sales      |03/09/38|6700
6521|lalit chowdary     |director|marketing|26/09/45|8200
Job Over
```

When arguments are specified in this way, the first word(the command itself) is assigned to $0,the second word(the first argument) to $1,and the third word(the second argument) to $2.You can use more positional parameters in this way up to $9(and using shift statement you can go beyond).

The parameter $? :

The parameter $? Stores the exit status of the last command. It has the value 0 if the command succeeds and a non-zero value if it fails. For example, if grep fails to find a pattern the return value is 1, and if the file scanned is unreadable in the first place, the return value is 2.In any case, return values exceeding 0 are to be interpreted as failure of the command. Try using grep in different ways:

```
#grep director emp.lst > /dev/null; echo $?
0
# grep manager emp.lst > dev/null;echo $?
1
# grep manager emp3.lst > dev/null;echo $?
grep:can  t open emp3.lst
2
```

The logical operators && and || Conditional execution:

The shell provides two operators to control execution of a command depending on the success or failure of the previous command the && and ||

The && operator is used by the shell in the same sense as it is used in C. It delimits two commands; the second command is executed only when the first succeeds. You can use it with the grep command in this way:

```
#grep  director  emp1.lst && echo  pattern found in file
1006|chanchal singhvi  |director|sales      |03/09/38|6700
6521|lalit chowdury    |director|marketing|26/09/45|8200
Pattern found in a file
```

The || operator is used to execute the command following it only when the previous command fails. If you :grep a pattern from a file without success, you can notify the failure:

```
#grep  manager  emp2.lst || echo  Pattern not found
Pattern not found
```

Evaluating expression

expr: computation and string handling:

The Bourne shell can check whether an integer is greater than another, but it doesn t have any computing features at al. it has to rely on the external **expr** command for that purpose. This command combines two functions in one:

- Performs arithmetic operations on integers.
- Manipulates strings.

Computation:

expr can perform the four basic arithmetic operations as well as the modulus functions:

```
# x= 3 y= 5                    multiple assignments without a;
#expr  3 + 5
8
#expr  $x - $y
-2
#expr  3 \*  5                  asterisk has to be escaped
15
#expr  $y / $x                  decimal portion truncated
1
#expr  13 % 5
3
```

String handling:

For manipulating strings, expr uses two expressions separated by a colon. The string to be worked upon is placed on the left of the:, and a regular expression is placed on its right. Depending on the composition of the expression, expr can perform three important string functions:

- Determine the length of the string
- Extract a substring
- Locate the position of a character in a string.

The length of a string: The length of a string is a relatively simple matter; the regular expression (.*) signifies to expr that it has to print the number of characters matching the pattern, i.e., the length of the entire string:

```
#expr  abcdefghijkl  :  .*          space on either side of : required
12
```

Here, expr has counted the number of occurrences of any character (.*).

Extracting a substring: expr can extract a string enclosed by the escaped characters \ (and \). If you wish to extract the 2-digit year from a 4-digit string, you must create a pattern group and extract it this way:

```
#stg=2003
#expr  $stg  :  ..\(..\)                    extracts last two characters
03
```

It signifies that the first two characters in the value of $stg have to be ignored and two characters have to be extracted from the third character position

Locating position of a character: expr can also return the location of the first occurrence of a character inside a string. To locate the position of the character d in the string value of $stg, you have to count the number of characters which are not d ([^d]*), followed by a d:

```
#stg = abcdefgh ; expr  $stg  :  [^d]*d
4
```

10. SHELL CONTROL STRUCTURES

THE if CONDITIONAL:

The if statement makes two-way decisions depending on the fulfillment of a certain condition. In the shell, the statement uses the following forms, much like the one used in other languages:

```
if condition is successful
then
        execute commands
else
        execute commands
fi
```
(Form 1)

```
if condition is successful
then
        execute commands
fi
```
(Form2)

If also requires a then. It evaluates the success or failure of the command that is specified in its command line . If command succeeds, the sequence of commands following it is executed. If command fails, then the else statement is executed.

```
#cat emp3.sh
#!/bin/sh
#
# if grep    ^$1     /etc/passwd   2>/dev/null
then
          echo    pattern found - job Over
else
         echo    Pattern not found
fi
```

```
#emp3.sh   ftp
ftp:*:325:15: FTP User:/users1/home/ftp:/bin/true
Pattern found     job over
#emp3.sh mail
Pattern not found
```

THE case CONDITIONAL:

The case statement is the second conditional offered by the shell. The statement matches an expression for more than one alternative, and uses a compact construct to permit multi-way branching. The general syntax of the case statement is as follows:

```
case expression in
     pattern1) execute commands1   ;;
```

```
        pattern2) execute commands2   ;;
        pattern3) execute commands3   ;;
                ..
    esac
```

case first matches the expression for a pattern1,and if successful, executes the commands associated with it. If it doesn t, then it falls through and matches pattern2, and so on. Each command list is terminated by a pair of semi-columns, and the entire construct is closed with esac (reverse of case).

```
#cat menu.sh
#
echo    MENU\n
1. List of files\n 2. Processes of user\n 3. Today  s date
4. Users of system\n 5. Quit to UNIX\n Enter your option: \c
read choice
case   $choice   in
        1) ls   l ;;
        2) ps   f ;;
        3) date ;;
        4) who ;;
        5) exit ;;
esac
```

#menu.sh

```
                MENU
1. List of files
2. Processes of user
3. Today  s date
4. Users of system
5. Quit to UNIX
Enter your option: 3
Tue Jan 8  18:09:06  IST 2003
```

for: LOOPING WITH A LIST

The **for** loop is different in structure from the ones used in other programming languages. Unlike while and until, for doesn t test a condition, but uses a list instead. The syntax for this construct is as follows:

Syntax1:

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

The loop body also uses the keywords **do** and **done,** but the additional parameters here are variable and list. Each whitespace-separated word in list is assigned to variable in turn, and commands are executed until list is exhausted.

```
#for file in chap20 chap21 chap22 chap23; do
> cp $file $(file).bak
>echo $file copied to $file.bak
>done
```

```
Chap20 copied to chap20.bak
Chap21 copied to chap21.bak
Chap22 copied to chap22.bak
Chap23 copied to chap23.bak
```

The list here comprises a series of character strings separated by whitespace.

while: LOOPING

Loops let you perform set of instructions repeatedly. The shell features three types of loops- while, until and for. All of them repeat the instruction set enclosed by certain keywords. While performs a set of instructions till the control command returns true exit status. The general syntax of this command is as follows:

```
while condition is true
do                                  Note the do keyword
        commands
done                                Note the done keyword
```

The commands enclosed by **do** and **done** are executed repeatedly as long as condition remains true.

```
#!/bin/sh
#
answer=y
while [   $answer   =   y   ]
do
        echo   enter the code and description: \c   >/dev/tty
        read code description
        echo   $code|$description   >> newlist
        echo   enter any more (y/n)? \c   >/dev/tty
        read anymore
        case $anymore in
                y*|Y*) answer=y ;;          #Also accepts yes, YES etc
                n*|N*) answer=n ;;          #Also accepts no, NO etc
                    *) answer=y ;;          #Any other reply means y
        esac
done
```

#emp5.sh

```
Enter the code and description: 03 analgesics
Enter anymore (y/n)? y
Enter the code and description: 04 antibiotics
Enter anymore (y/n)? y
Enter the code and description: 05 OTC drugs
Enter anymore (y/n)? n
#cat newlist
03| analgesics
04| antibiotics
05| OTC drugs
```

RELATIONAL AND LOGICAL OPERATORS

Relational operators:

Operators	Meaning
-eq	Equal to
-ne	Not equal to
-gt	Greater than
-ge	Greater than or equal to
-lt	Less than
-le	Less than or equal to

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.

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

Relational operators:

Here is an example which uses all the relational operators. Assume variable a holds 10 and variable b holds 20:

```
#cat rel.sh
#!/bin/sh
a=10
b=20
if [ $a -eq $b ]
then
        echo    $a  -eq  $b : a is equal to b
else
        echo    $a  -eq  $b : a is not equal to b
fi
if [ $a -ne $b ]
then
        echo    $a  -ne  $b : a is not equal to b
else
```

```
        echo    $a  -ne  $b : a is equal to b
fi
if [ $a  -gt  $b ]
then
        echo    $a  -gt  $b : a is greater than b
else
        echo    $a  -gt  $b : a is not greater than b
fi
if [ $a  -lt  $b ]
then
        echo    $a  -lt  $b : a is less than b
else
        echo    $a  -lt  $b : a is not less than b
fi
if [ $a  -ge  $b ]
then
        echo    $a  -ge  $b : a is greater than or equal to b
else
        echo    $a  -ge  $b : a is not greater than or equal to b
fi
if [ $a  -le  $b ]
then
        echo    $a  -le  $b : a is less than or equal to b
else
        echo    $a  -le  $b : a is not less than or  equal to b
fi
```

#chmod +x rel.sh

#./rel.sh

The above script will generate the following result:

```
10    eq  20: a is not equal to b
10    ne  20: a is not equal to b
10    gt  20: a is not greater than b
10    lt   20: a is less than b
10    ge  20: a is not greater than or equal to b
10    le   20: a is less than or equal to b
```

Logical operators:

Here are logical operators that are used during shell script:

Operator	Description
cond1 -a cond2	True if condition1 and condition2 are true(perform AND operation)
cond1 -o cond2	True if condition1 and condition2 are true(perform OR operation)
!cond1	True if condition1 is false

Example:

#cat log.sh

```
#!/bin/bash
a=25
b-29
if [ $a    gt 20    a $b    gt 25 ]; then
echo    both condition satisfied
fi
if [ $a    gt 25    o $b    gt 25 ]; then
echo    only one condition is satisfied
fi
```

#chmod +x log.sh
#./log.sh
Both condition satisfied
Only one condition is satisfied.

ADVANCED FILTER SED AND AWK

sed – THE STREAM EDITOR:

sed is a multipurpose tool which combines the work of several filters. sed performs non-interactive operations on a data stream, hence its name. It uses very few options but has a host of features that allow you to select lines and run instructions on them.

A sed instruction:

Everything in sed is an instruction. An instruction combines an address for selecting lines, with an **action** to be taken on them, as shown by the syntax:

sed *options address action file(s)*

sed has two ways of addressing lines:

*By one or two line number
*By specifying a /- enclosed pattern which occurs in a line

In the first form, address specifies either one line number to select a single line, or a set of two to select a group of contiguous lines. Likewise; the second form can use one or two patterns. The action can be either insertion, deletion or substitution of text (table).sed has its own family of internal commands which transform selected lines/text in several ways.

Line addressing:

sed follows the line addressing as, you either specify a line or a pair of them ,to limit the boundaries of the selection. The action is appended to this address. For instance, the instruction 3q can be broken to the address 3 and action q (quit).When the instruction is enclosed within quotes and followed by one or more filenames, you can simulate head -3 in this way:

```
# sed    3q    emp.lst                                    #quits after line no. 3
2233|a. k. shukla          |g.m       |sales       |12/12/52|6000
9867|jai sharma           |director  |production|12/03/50|4000
3456|sumit chakrobarty|d.g.m      |marketing |19/04/43|6000
```

sed also uses the p(print) command to print the output.But notice what happens when you use two line addresses along with p command:

```
#sed 1,2p emp.lst
2233|a. k. shukla |g.m |sales |12/12/52|6000
2233|a. k. shukla |g.m |sales |12/12/52|6000
9867|jai sharma |director |production|12/03/50|4000
9867|jai sharma |director |production|12/03/50|4000
3456|sumit chakrobarty|d.g.m |marketing |19/04/43|6000
2345|n.k.gupta |chairman|personnal |11/03/47|7800
1008|chanchal singhvi |director |sales |03/02.90|4500
2456|anil aggarwal |manager |sales |01/09/90|4000
```

.more lines with each line displayed only once.

You probably intended to select first two lines of emp.lst with the instruction 1,2p. Instead, the first two lines have been printed twice. Hence we can know that sed with its p command by default prints all lines on the standard output, in addition to all lines affected by action.

The n option: suppressing duplicate line printing

To overcome problem of printing duplicate lines, you should use the n option whenever you use the p command. Thus previous command should have been written as follows:

```
#sed n 1,2p emp.lst
2233|a. k. shukla |g.m |sales |12/12/52|6000
9867|jai sharma |director |production|12/03/50|4000
```

And to select last line of file, use $

```
#sed n $p emp.lst
2456|anil aggarwal |manager |sales |01/09/90|4000
```

sed also has a negation operator(!),which can be used with any action. For instance, selecting the first 2 lines is just same as not selecting lines 3 through the end. The next to previous command sequence can be written in this way too:

```
#sed n 3,$!p emp.lst                    #Dont print lines 3 to the end
```

The address and action are enclosed with a pair of single quotes.

Selecting lines from middle:

sed can also select lines from the middle of a file,to select lines from 9 through 11,you have

```
#sed n 9-11p emp.lst
```

sed is not restricted to selecting contiguous groups of lines.

```
#sed n 1-2p                    #3 addresses in one command, using only a single
7,9p                           #pair of quotes
$p emp.lst                     #selects last line
```

There is also an alternative method of entering previous sequence of instructions. The e option allows you to enter as many instructions as you wish

```
#sed n e 1-2p e 7-9p e $p emp.lst
```

Inserting and changing text:

sed can also insert new text and change existing text in a file. If you remember doing similar things in vi.sed uses i(insert),a(append) and c(change) commands. But there are important differences too. Let s append 2 records to input on emp.lst and store new output in empnew.lst

```
#sed $a\
>2033|a. k. bose|g.m |sales |1612/52|6500\
```

>9877|jayant |director |production|12/03/50|4500
> emp.lst > empnew.lst

First enter instruction $a which appends text at the end of file. Then enter \ before pressing the <enter> key. You can key in as many lines as you wish. Each line except the last line has to be terminated by \ before hitting the <enter> key.sed identifies line without \ as the last line of input.

Double spacing text:

Consequence of not using an address with these commands. The inserted or changed text then is placed after or before every line of file

```
sed  i\                                          #inserts before every line
                                                 #this is blank line
  emp.lst
```

Inserts a blank line before each line of file printed. This is another way of double spacing text, **a** would insert a blank line after each selected line.

Context addressing:

The second form of addressing lets you specify a pattern as well. When you specify a single pattern all lines containing the pattern are selected.

#**sed -n /director/p emp.lst**

This method of addressing lines by specifying context is known as context addressing. You can also specify a comma-separated pair of context addresses to select group of lines. Line and context addresses can also be mixed.

#**sed n /dasgupta/,/saxsena/p emp.lst**
#sed n 1,/dasgupta/p emp.lst

Deleting lines:

Using d(delete) command, sed can evaluate grep s v option to select lines not containing the pattern. Either of following commands

#**sed /director/d emp.lst > olist** # -n option not to be used with d
sed -n /director/!p emp.lst > olist

Selects all but lines containing director and save them in list.

Writing selected lines to a file:

The (w) write command makes it possible to write selected lines in a separate file. Save the records of directors in a disk in this way.

#sed n /director/w dlist emp.lst #no display when using n option

Since sed accepts more than one address, you can perform a full context splitting of file emp.lst. You can also have

```
#sed  n  /director/w dlist
          /manager/w mlist
         /executive/w elist   emp.lst
```

The f option: Instructions from a file:

When there are numerous editing instructions to be performed, it will be better to use f option to accept instructions from a file. For e.g., previous instructions could have been stored in a file instr.fil

```
#cat >  instr.fil
/director/w dlist
/manager/w mlist
```

/executive/w elist

And then sed used with f file name option.

#**sed n f instr.fil emp.lst**

sed is quite literal in that it allows a great deal of freedom in using and repeating options. You can use f options with multiple files. You can also combine e and f option as many times as you want.

#**sed n f instr.fil1 f instr.fil2 emp.lst**

Substitution:

sed s strongest feature is substitution, achieved with its s(substitute) command. It lets you replace a pattern in its input with something else. The syntax for such a command can be described as

[Address] s/expression1/expression2/flags

Here string1 will be replaced preferred by string2 in all lines specified by the address. If address is not specified substitution will be preferred by all lines containing string1. Using this syntax let s first replace word director by member in first five lines of emp.lst

#**sed 1,5s/director/member/ emp.lst.**

To broaden search so that it affects all lines you may either use global address 1, $ or simply drop address altogether. In the absence of an address sed acts on all lines. i.e.

#**sed 1,$s/director/member/ emp.lst**

#sed s/director/member/ emp.lst

Remember that in absence of n and p options, all lines are displayed whether substitution has been preferred or not.

sed also uses regular expressions for patterns to be substituted. To replace all occurrences of agarwal , aggrawal and aggarwal by simply Agarwal use regular expression as

#**sed n s/[Aa]gg*[ar][ra]wal/Agarwal/p emp.lst**

This time n and p command ensured that only lines affected by substitution are displayed; unaffected lines are suppressed.

Global substitution:

Suppose you require to replace all occurrence of director character in first line by #. See what happens when you use following command.

#sed s/ | / # / emp.lst | head -3

2233#a. k. shukla |g.m |sales |12/12/52|6000
9867#jai sharma |director |production|12/03/50|4000
9867#jai sharma |director |production|12/03/50|4000

This only replaces first occurrence of the | in all lines other pipes remain unaffected. All the previous substitutions also acted on first occurrences in the line, you couldn t know that because string director never occurred more than once in same line. To replace all occurrences, you need to use g (global) flag at the end of instruction, referred to as global substitution. i.e.

#**sed s/ | / # /g emp.lst | head -3**

2233#a. k. shukla #g.m #sales #12/12/52#6000
9867#jai sharma #director #production#12/03/50#4000
9867#jai sharma #director #production#12/03/50#4000

Compressing multiple spaces:

How do you delete trailing spaces from second, third and fourth fields? Regular expression required in source string needs to signify one or more occurrences of a space, followed by a |:

#**sed s/ *|/ | /g emp.lst | head -2**

2233|a.k.agarwal|g.m|sales|12/12/1990|1200
9876|jai sharma|director|production|13/12/2000|5000

The remembered pattern:

There s another way of performing substitution. Scan entire file for say string director and replace first occurrence in each line matched.

#sed n /director/s/director/member/p emp.lst

Here the address /director/ appears to be redundant; without using it will have same output. However, you must understand this form also because it widens scope of substitution. It s possible that you may like to replace a string in all lines containing a different string. For example, you may like to change the designation of only that director who heads the following marketing function. Use this form to scan the file for pattern marketing and then perform required substitution.

#sed n marketing/s/director/member/p emp.lst

1234|lalit chowdury |member |marketing |09/12/45|2500

The above sequence however be condensed if the scanned pattern is same as the substituted pattern i.e. source string. This was the case in next to previous example.sed remembers scanned patterns and lets you use // (two front slashes) as source string if its same as the scanned pattern. You then need not to specify pattern there. Two forms are then equivalent.

#sed director/s//member emp.lst

#sed s/director/member emp.lst

The second form, though require two fewer keystrokes. This empty (or null) regular expression enclosed within the two front slashes and is understood by sed to represent search pattern.

The repeated patterns:

When a pattern is the source string also occurs in replacement, you can use special characters & to represent it. Both the following commands do same thing.

#sed /director/s//executive director/ emp.lst

#sed /director/s//executive &/ emp.lst

The &, known as repeated pattern, expands to the entire source string. The & is the only other special character you can use in replacement string.

Internal commands used by sed:-

Commands	Description
i	inserts before line
a	appends after line
c	changes line(s)
d	deletes line(s)
p	prints line(s)on standard o/p
q	quits after reading up to addressed line
=	prints line number addressed
s/s1/s2	substitute string1 by string2
r flname	places contents of the file flname after line
w flname	writes addressed lines of file flname

awk: AN ADVANCED FILTER

awk is an interpreted programming language which focuses on processing text. It was designed to execute complex pattern-matching operations on streams of textual data. It makes heavy use of strings, associative arrays and regular expressions.

SIMPLE awk FILTERING

awk is a little awkward to use at first, but if you feel comfortable with sed, then you ll find a friend in awk. The syntax is as follows:

awk options ***selection_criteria {action} file(s)***

The selection_criteria filters input and selects lines for the action component to act upon. This component is enclosed within curly braces. The selection_criteria and actions constitute an awk program that is surrounded by a set of single quotes.

A typically complete awk command specifies the selection criteria and action. The following command selects the directors from emp.lst

```
#awk   /director/ {print}   emp.lst
9876| jai Sharma          |director |production| 12/03/50|7000
2345| barun sengupta     |director |personnel  | 11/05/47|7800
1006| chanchal singhvi  |director |sales        | 03/09/38|6700
6521| lalit chowdary     |director |marketing  | 26/09/45|8200
```

The selection_criteria section (/director/) selects lines that are processed in the action section ({print}). An awk program must have either the selection criteria or the action, or both, but within single quotes. Double quotes will create problems unless used judiciously. For pattern matching, awk uses regular expressions in sed-style:

```
#awk  -F  |      /sa[kx]s*ena /    emp.lst
3212| shyam saxena      |d.g.m |accounts    | 12/03/50|6000
2345| j.b. saksena       |g.m    |marketing  | 11/05/47|8000
```

SPLITTING A LINE INTO FIELDS

awk uses the special parameter, $0, to indicate the entire line. It also identifies fields by $1, $2, $3. awk uses a contiguous sequence of spaces and tabs as a single delimiter. You can use awk to print the name, designation, department and salary of all the sales people.

```
#awk  -F  |      /sales/ { print $2, $3, $4, $6 }    emp.lst
a.k.shukla              g.m             sales   6000
chanchal singhvi        director        sales   6700
s.n.dasgupta            manager         sales   5600
anil aggarwal           manager         sales   5000
```

Notice that a, (comma) has been used to delimit the field specifications. This ensures that each field is separated from the other by a space.

If you want to select lines 3 to 6, all you have to do is use the built-in variable NR to specify the line numbers:

```
#awk  -F  |      NR==3, NR==6  { print NR,  $2, $3, $6 }     emp.lst
3 n.k.gupta             chairman        5400
4 v.k.agrawal           g.m             9000
5 j.b.saxena            g.m             8000
6 sumit chakraborty     d.g.m           6000
```

NR is one of those built-in variables used in awk programs, and == is one of the many operators employed in comparison tests.

printf: FORMATTING OUTPUT

The C-like **printf** statements, you can use awk as a stream formatter, **awk** accepts most of the formats used by the **printf** function used in C, but the %s format will be used for string data, and %d for numeric.

```
#awk  -F  |      /[aA]gg?[ar]+wal/ {
>printf   %3d  %-20s  %-12s  %d\n   , NR, $2, $3, $6 }    emp.lst
4 v.k. aggarwal         g.m             9000
9 sudhir agarwal        executive       7500
15 anil aggarwal        manager         5000
```

Number Comparison

sed can also handle numbers both integer and floating type and make relational tests on them. You can now print pay- slips for those people whose basic pay exceeds 7500:

```
#awk -F | $6 > 7500 {
>printf %-20s %-12s %d\n , $2, $3, $6 } emp.lst
v.k.agrawal g.m 9000
j.b.saxena g.m 8000
lalit chowdary director 8200
barun sengupta director 7800
```

You can also combine regular expression matching with numeric comparison to locate those, either born in 1955 or drawing a basic pay greater than 8000:

```
#awk -F | $6 >8000 || $5 -/45$/ emp.lst
0110|v.k.agrawal |g.m |marketing |31/12/40 |9000
2345|j.b.saxena |g.m |marketing |12/03/45 |8000
6521|lalit chowdary |director |marketing |26/09/45 |8200
```

THE BEGIN AND END SECTIONS

If you have to print something before processing the first line, for example, a heading, then the BEGIN section can be used quite gainfully. Similarly, the END section is useful in printing some totals after processing is over.

The begin and end sections are optional and take the form

BEGIN { action } *both require curly braces*

END { action }

```
#vi empawk.awk
BEGIN {
printf \t\t Employee Abstract \n\n
} $6 > 7500 {
kount++ ; tot+= $6
printf %3d %-20s %-12s %d\n , kount, $2, $3, $6
}
END {
printf \n\t The average basic pay is %6d\n , tot/kount
}
#chmod +x empawk.awk
```

The BEGIN section here prints a suitable heading, offset by two tabs (\t\t) , while the END section prints the average pay(tot/kount) for the selected lines. To execute this program, use the f option:

```
#awk F | -f empawk.awk emp.lst
Employee abstract
1 v.k.agrawal g.m 9000
2 j.b.saxena g.m 8000
```

ARRAYS

An array is also a variable except that this variable can store a set of value or elements. Each element is accessed by a subscript called the index. Array are indexed using numbers, they usually start at 0 and go to N-1 the number of element in an array.

Example:

```
#cat array.awk
#!/usr/bin/awk -f
```

```
begin {
some_array [1] =    hello
some_array [2] =    everybody
some_array [3] =    !
print some_array [1], some_array [2], some_array [3]
}
```

Output: ./array.awk

Hello everybody!

Associative (HASH) Arrays

awk arrays are associative, where information is held as key-value pairs often referred to as hash. The index is the key that is saved internally as a string. Instead of using a fixed integer to index the array, you can use a value, a string, to identify each element in the associative array. When we set an array element using mon [1] = mon , awk converts the number 1 to a string.

```
#awk  BEGIN {
>direction [  N  ] =  North  ; direction [  S  ] =  South  ;
>direction [  E  ] =  East  ; direction [  W  ] =  West  ;
>printf (  N is %s and W is %s\n  , direction [  N  ], direction [  W  ]);
>mon[1] =  jan  ; mon[  1  ] =  January  ; mon[  01  ] =  JAN  ;
>printf (  mon[1] is %s\n  , mon[1]) ;
>printf (  mon[01] is also %s\n  , mon[01]) ;
>printf (  mon[ \  1\  ] is also  %s\n  , mon["1"]);
>printf (  but mon[\  01\  ] is %s\n  , mon[  01  ]);
>}
```

Output:

```
N is North and W is West
mon [1] is January
mon [01] is also January
mon [  1  ] is also January
But mon [  01  ] is JAN
```

ENVIRON [ ]: The Environment Array

awk maintains the associative array, ENVIRON [ ], to store all environment variables. This POSIX requirement is met by recent versions of awk including nawk (new awk) and gawk (GNU awk).

```
#nawk  BEGIN {
>print  HOME  =  environ [  HOME  ]
>print  PATH  =  environ [  PATH  ]
>}
```

Output:

```
HOME=/users1/home/staff/sumit
PATH=/usr/bin:: /usr/local/bin:: /usr/ccs/bin
```