Background:

```
sed OPTIONS... [SCRIPT] [INPUTFILE...]
```

For example, to replace all occurrences of ‘hello’ to ‘world’ in the file input.txt:

```
sed 's/hello/world/' input.txt > output.txt
```

If you do not specify *INPUTFILE*, or if *INPUTFILE* is -, sed filters the contents of the standard input. The following commands are equivalent:

```
sed 's/hello/world/' input.txt > output.txt

sed 's/hello/world/' < input.txt > output.txt

cat input.txt | sed 's/hello/world/' - > output.txt
```

sed writes output to standard output. Use -i to edit files in-place instead of printing to standard output. See also the W and s///w commands for writing output to other files. The following command modifies file.txt and does not produce any output:

```
sed -i 's/hello/world/' file.txt
```

By default sed prints all processed input (except input that has been modified/deleted by commands such as d). Use -n to suppress output, and the p command to print specific lines. The following command prints only line 45 of the input file:

```
sed -n '45p' file.txt
```

Command to Delete pattern matching line

Syntax:

```
$ sed '/pattern/d' filename.txt
```

Example:

```
$ sed '/abc/d' filename.txt
```

Program 1:

```
echo "enter the filename"
read fname
echo "enter the starting line number"
read s
echo "enter the ending line number"
read n
sed -n $s, $n\p $fname | cat > newline
cat newline
```

Introduction to Shell

```
$echo $$
```

The above command writes the PID of the current shell −

```
29949
```

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.

```
#!/bin/sh

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

For loop to be executed up to number of arguments given through command line: -

Program2: - ShellScipt.sh (for example specified word to delete is Kolkata)

```
#!/bin/bash
for i in $*
do
sed '/Kolkata/d' $i
done
```

Command: - sh ShellScript.sh file1.txt file2.txt …… filen.txt

Then the Kolkata will be deleted from all the files given as a command line argument.

Program3: Write a shell script that displays a list of all the files in the current working directory.

```
echo "enter the directory name"
read dir
if [ -d $dir ]
then
cd $dir
ls > f
exec < f
while read line
do
if [ -f $line ]
then
if [ -r $line -a -w $line -a -x $line ]
then
echo "$line has all permissions"
else
echo "files not having all permissions"
fi
fi
done
```

Example: - Command Line Arguments

I. Command Line Arguments are passed by the user from the terminal.
II. int argc keep track of the count of command line argument
III. argv[] basically represents an array of strings that stores each argument in string format

Program 4(a): Program to check the arguments and print the first command line argument

```
#include <stdio.h>

void main(int argc, char *argv[] ){
printf("Example of Command Line Arguments in C!\n");
printf("Program name is: %s\n",argv[0]);
printf("Arguments passes: %d\n", argc);
if(argc < 2){
printf("You did not pass any arguments\n");
}
else{
printf("First argument: %s\n", argv[1]);
}
}
```

Program4(b): Program Print all the arguments received as command line Arguments.

```
#include<stdio.h>
int main(int argc,char* argv[])
{
int i;
printf("Program Name is: %s\n",argv[0]);
printf("Using Command Line Arguments in C!\n\n");
if(argc==1)  //Count is 1 that is Program name only
printf("You did not provide any arguments!\n");
if(argc>=2)
{
printf("Arguments Passed: %d.\n\n",argc);
printf("Below are the arguments that is received by the Program!\n");
for(i=0;i<argc;i++)
printf("argv[%d]: %s\n",i,argv[i]);
}
return 0;
}
```

File and Directory:

Given a pathname, the stat function returns a structure of information about the named file. The fstat function obtains information about the file that is already open on the descriptor filedes. The lstat function is similar to stat, but when the named file is a symbolic link, lstat returns information about the symbolic link, not the file referenced by the symbolic link. The second argument is a pointer to a structure that we must supply. The function fills in the structure pointed to by buf. The definition of the structure can differ among implementations, but it could look like

```
struct stat {
mode_t st_mode; /* file type & mode (permissions) */
ino_t st_ino; /* i-node number (serial number) */
dev_t st_dev; /* device number (file system) */
dev_t st_rdev; /* device number for special files */
nlink_t st_nlink; /* number of links */
uid_t st_uid; /* user ID of owner */
gid_t st_gid; /* group ID of owner */
off_t st_size; /* size in bytes, for regular files */
time_t st_atime; /* time of last access */
time_t st_mtime; /* time of last modification */
time_t st_ctime; /* time of last file status change */
```

```
 blksize_t st_blksize; /* best I/O block size */
 blkcnt_t st_blocks; /* number of disk blocks allocated */
 };
```

File Types: -

Most files on a UNIX system are either regular files or directories, but there are additional types of files. The types are:

1. **Regular file.** The most common type of file, which contains data of some form. There is no distinction to the UNIX kernel and whether this data is text or binary. Any interpretation of the contents of a regular file is left to the application processing the file. One notable exception to this is with binary executable files. To execute a program, the kernel must understand its format. All binary executable files conform to a format that allows the kernel to identify where to load a program's text and data.

2. **Directory file.** A file that contains the names of other files and pointers to information on these files. Any process that has read permission for a directory file can read the contents of the directory, but only the kernel can write directly to a directory file. Processes must use the functions described in this chapter

to make changes to a directory.

3. **Block special file.** A type of file providing buffered I/O access in fixed-size units to devices such as disk drives.

4. **Character special file.** A type of file providing unbuffered I/O access in variable-sized units to devices. All devices on a system are either block special files or character special files.

5. **FIFO.** A type of file used for communication between processes. It's sometimes called a named pipe.

6. **Socket.** A type of file used for network communication between processes. A socket can also be used for non-network communication between processes on a single host. We use sockets for interprocess communication.

7. **Symbolic link.** A type of file that points to another file.

File type Macro <sys/stat.h>