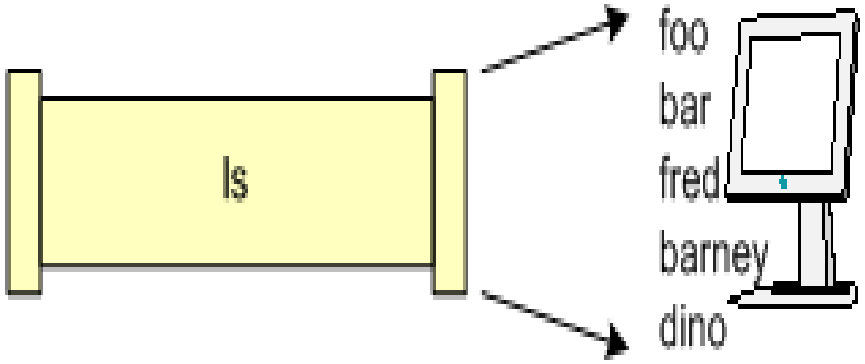
I/O redirection

Pipe and filters

Redirection

- In typical Unix installations, commands are entered at the keyboard and output resulting from these commands is displayed on the computer screen. Thus, input (by default) comes from the terminal and the resulting output (stream) is displayed on (or directed to) the monitor. Commands typically get their input from a source referred to as **standard input** (stdin) and typically display their output to a destination referred to as **standard output** (stdout) as pictured below:

- Input flows (by default) as a stream of bytes from standard input along a channel, is then manipulated (or generated) by the command, and command output is then directed to the standard output. The ls command can then be described as follows; there is really no input (other than the command itself) and the ls command produces output which flows to the destination of stdout (the terminal screen), as below:



Input/Output Redirection

Unix provides the capability to change where standard input comes from, or where output goes using a concept called Input/Output (I/O) redirection. I/O redirection is accomplished using a redirection operator which allows the user to specify the input or output data be redirected to (or from) a **file**. Note that redirection always results in the data stream going to or coming from a file (the terminal is also considered a file).

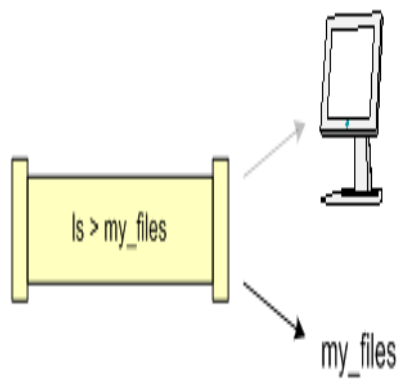
The simplest case to demonstrate this is basic **output redirection**. The output redirection operator is the > (greater than) symbol, and the general syntax looks as follows:

```
command > output_file_spec
```

Spaces around the redirection operator are not mandatory, but do add readability to the command. Thus in our ls example from above, we can observe the following use of output redirection:

```
$ ls > my_files [Enter]
$
```

Notice there is no output appearing after the command, only the return of the prompt. Why is this, you ask? This is because all output from this command was *redirected* to the file my_files. Observe in the following diagram, no data goes to the terminal screen, but to the file instead.



Examining the file as follows results in the contents of the my_files being displayed:

```
$ cat my_files [Enter]
foo
bar
fred
barney
dino
$
```

In this example, if the file my_files does not exist, the redirection operator causes its creation, and if it does exist, the contents are overwritten. Consider the example below:

```
$ echo "Hello World!" > my_files [Enter]
$ cat my_files [Enter]
Hello World!
```

Notice here that the previous contents of the my_files file are gone, and are replaced with the string "Hello World!" Note also that when using redirection, the output file is created first, following:

```
$ cat my_files [Enter]
Hello World!
$ cat my_files > my_files [Enter]
$ cat my_files [Enter]
$
```

Often we wish to add data to an existing file, so the shell provides us with the capability to append output to files. The append operator is the >>. Thus we can do the following:

```
$ ls > my_files [Enter]
$ echo "Hello World!" >> my_files [Enter]
$ cat my_files [Enter]
foo
bar
fred
barney
dino
Hello World!
```

```
$ 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
$
```

Upon execution, you will receive the following output. You can count the number of lines in the file by redirecting the standard input of the **wc** command from the file *users* −

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

Redirection Commands

Following is a complete list of commands which you can use for redirection –

Sr.No.	Command & Description
1	**pgm > file** Output of pgm is redirected to file
2	**pgm < file** Program pgm reads its input from file
3	**pgm >> file** Output of pgm is appended to file
4	**n > file** Output from stream with descriptor **n** redirected to file
5	**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

Pipes & filter

- A concept closely related to I/O redirection is the concept of piping and the pipe operator. The pipe operator is the | character (typically located above the enter key). This operator serves to join the standard output stream from one process to the standard input stream of another process.
- 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)
```

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”.

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 −

```
$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	**-l** Prints only the names of files with matching lines (letter "l")
4	**-c** Prints only the count of matching lines.
5	**-i** Matches either upper or lowercase.

Example-1:

To Search for the given string in a single file test.sh

```
$ cat test.sh
#!/bin/bash
fun()
        echo "This is a test."
        # Terminate our shell script with success message
        exit 1

fun()
```

from above file grep exit:

```
$ grep "exit" demo_file
```

output:

```
        exit 1
```

Example-2:

To Checking for the given string in multiple files: in this case test.sh and test1.sh

```
$ cat test.sh
#!/bin/bash
fun()
        echo "This is a test."
        # Terminate our shell script with success message
        exit 1

fun()
```

```
$ cat test1.sh
#!/bin/bash
fun()
        echo "This is a test1."
        # Terminate our shell script with success message
        exit 0

fun()
```

grep exit in both files test.sh and test1.sh:

```
$ grep exit test*
```

output:

```
test1.sh:       exit 0
test.sh:        exit 1
```

Example-3:

To Case insensitive search using grep -i, added EXIT in test1.sh

```
$ cat test1.sh
#!/bin/bash
fun()
        echo "This is a test1."
        # Terminate our shell script with success message, EXIT with 0
        exit 0

fun()
```

```
$ grep exit test1.sh
test1.sh:       exit 0
```

```
$ grep -i exit test*
```

output:

```
test1.sh:       # Terminate our shell script with success message, EXIT with 0
test1.sh:       exit 0
```

To Invert match using grep -v

```
$ cat test.sh
#!/bin/bash
fun()
        echo "This is a test."
        # Terminate our shell script with success message
        exit 1

fun()


$ grep -v  exit test.sh
```

output:

```
#!/bin/bash
fun()
        echo "This is a test."
        # Terminate our shell script with success message
```

To print line number of searched string

$ grep -n exit test.sh

output:

5: exit 1

```
$ cat file.txt
ostechnix
Ostechnix
o$technix
linux
linus
unix
technology
hello world
HELLO world
```

```
$ grep 'hello world' file.txt
hello world
```

You can also use **-n** flag to show the line numbers in the output:

```
$ grep -n nix file.txt
```

Sample output:

```
1:ostechnix
2:Ostechnix
3:o$technix
6:unix
```

To search for the words that matches the pattern "tech" at the beginning of the line in a file, run:

```
$ grep ^tech file.txt
technology
```

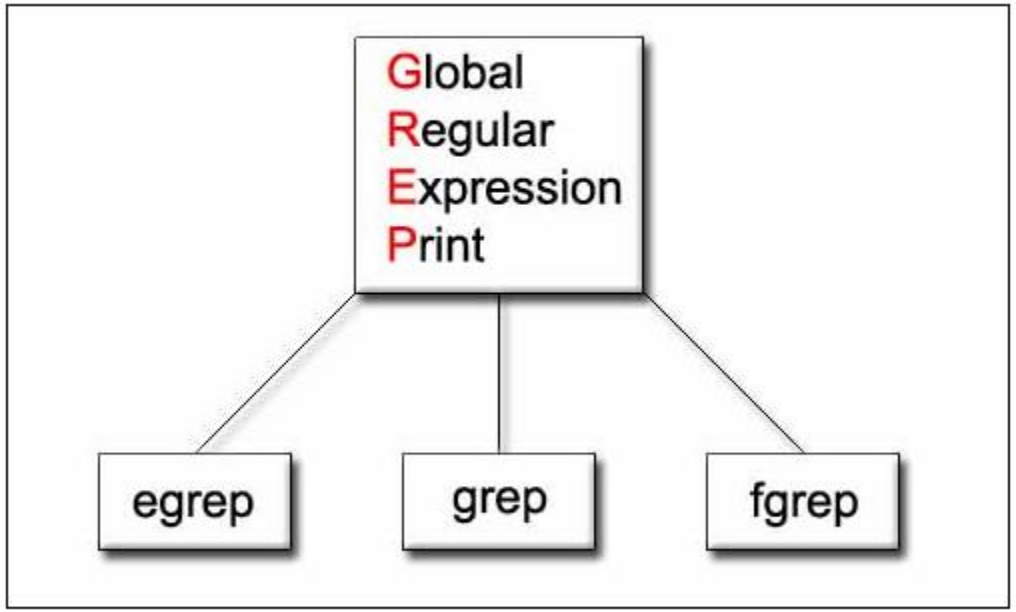
Similarly, we can search for the words that ends with a particular letter(s), for example **"x"**, like below.

```
$ grep x$ file.txt
ostechnix
Ostechnix
o$technix
linux
unix
```

Also, you can find the words that contains any character using **.** (dot).

For example, let us search for any word that has **"n"** in the file.

```
$ grep .n file.txt
ostechnix
Ostechnix
o$technix
linux
linus
unix
technology
```



egrep stands for **e**xtended **grep**. It is similar to **"grep -E"** command. It will do all the things that grep will do. However, It provides some additional functionalities, such as using complicated regex, than the normal grep command does out of the box.

fgrep stands for **f**ast **grep**. It is similar to **"grep -F"**. fgrep can't recognize regular expressions or special characters. fgrep can be used where you want regular expressions to be evaluated.

```
$ egrep '^(l|o)' file.txt
ostechnix
o$technix
linux
linus
```

Similarly, We can search for the lines that start with any character range between "l" to "u". That means, we will get the lines that start with l, m, n, o, p, q, r, s, t, and u. Everything else will be omitted from the result.

```
$ egrep '^[l-u]' file.txt
ostechnix
o$technix
linux
linus
unix
technology
```

For example, we use the following command to find the words end in **"x"**.

```
$ grep x$ file.txt
ostechnix
Ostechnix
o$technix
linux
unix
```

Now run the same command with fgrep.

```
$ fgrep x$ file.txt
```

It will display nothing, because it couldn't evaluate the special characters.

Now let us see another example. To search for the words that matches the string **"o$"** with grep command, we do:

```
$ grep o$ file.txt
```

But, we get nothing, why? Because, as per the above command, we use the dollar symbol($) to find the words that ends in "o". Since there were no characters ends with "o" in file.txt, we get nothing.

Now, run the same command with fgrep.

```
$ fgrep o$ file.txt
o$technix
```