hitbullseye

Introduction, shell responsibilities, pipes, and input Redirection

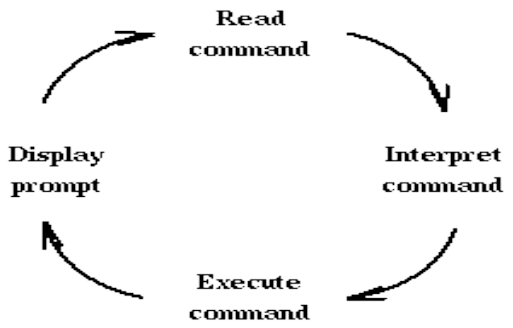
A **Shell** provides you with an interface to the Unix system. It gathers input from you and executes programs based on that input. When a program finishes executing, it displays that program's output.

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 Interpretive Cycle

The shell is perhaps the most important program on the Unix system, from the end-user's standpoint. The shell is your interface with the Unix system, the middleman between you and the kernel.

CONCEPT: The shell is a type of program called an *interpreter*. An interpreter operates in a simple loop: It accepts a command, interprets the command, executes the command, and then waits for another command. The shell displays a "prompt," to notify you that it is ready to accept your command.



Interpreter loop

The shell recognizes a limited set of commands, and you must give commands to the shell in a way that it understands: Each shell command consists of a command name, followed by command options (if any are desired) and command arguments (if any are desired). The command name, options, and arguments are separated by blank spaces.

The Shell's Responsibilities

Now you know that the shell analyzes each line you type in and initiates the execution of the selected program. But the shell also has other responsibilities, as outlined in Figure A

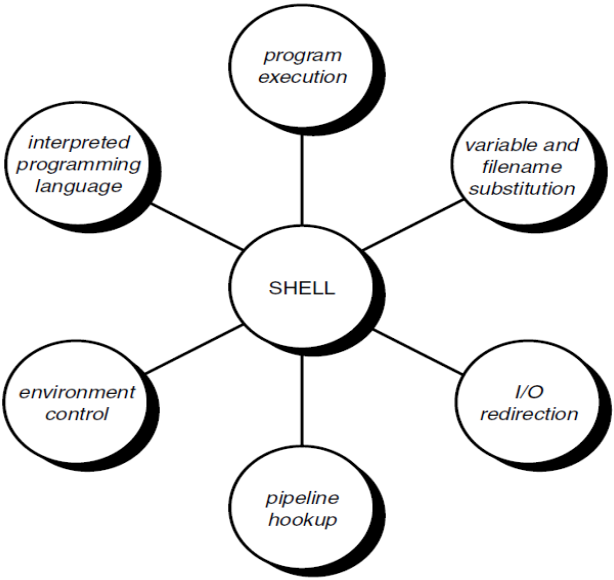


Fig: (A) The shell's Responsibilities

Program Execution

The shell is responsible for the execution of all programs that you request from your terminal. Each time you type in a line to the shell, the shell analyzes the line and then determines what to do. As far as the shell is concerned, each line follows the same basic format:

program-name arguments:

The line that is typed into the shell is known more formally as the command line. The shell scans this command line and determines the name of the program

to be executed and what arguments to pass to the program. The shell uses special characters to determine where the program name starts and ends, and where each argument starts and ends. These characters are collectively called *whitespace characters*, and are the space character, the horizontal tab character, and the end-of-line character, known more formally as the *newline character*. Multiple occurrences of whitespace characters are simply ignored by the shell. When you type the command

mv tmp/mazewars games the shell scans the command line and takes everything from the start of the line to the first whitespace character as the name of the program to execute: mv. The set of characters up to the next whitespace character is the first argument to mv: **tmp/mazewars**. The set of characters up to the next whitespace character (known as a word to the shell)—in this case, the newline—is the second argument to mv: games. After analyzing the command line, the shell then proceeds to execute the mv command, giving it the two arguments tmp/mazewars and games (see fig-B)

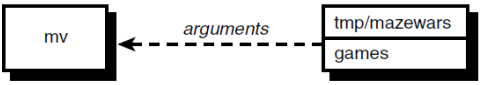


Fig: (B) Execution of mv with two arguments.

As mentioned, multiple occurrences of whitespace characters are ignored by the shell. This means that when the shell processes this command line:

echo when do we eat?

it passes four arguments to the echo program: when, do, we, and eat? See fig: (c)

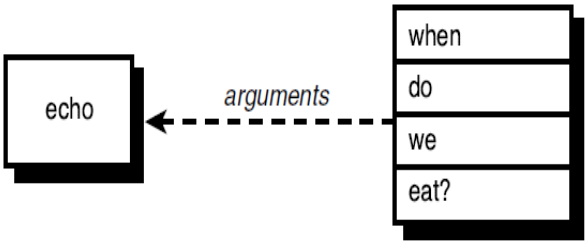


Fig: (c) Execution of echo with four arguments.

Because echo takes its arguments and simply displays them at the terminal, separating each by a space character, the output from the following becomes easy to understand:

$ **echo when do we eat?**

when do we eat?

$

The fact is that the echo command never sees those blank spaces **I/O Redirection:**

It is the shell's responsibility to take care of input and output redirection on the command line. It scans the command line for the occurrence of the special redirection characters <, >, or >>. When you type the command

echo Remember to tape Law and Order > reminder

the shell recognizes the special output redirection character > and takes the next word on the command line as the name of the file that the output is to be redirected to. In this case, the file is a reminder.

Before the shell starts execution of the desired program, it redirects the standard output of the program to the indicated file. As far as the program is concerned, it never knows that its output is being redirected. It just goes about its merry way writing to standard output. Let's take another look at two nearly identical commands:

$ **wc -l users**

5 users

$ **wc -l < users**

5

$

In the first case, the shell analyzes the command line and determines that the name of the program to execute is wc and it is to be passed two arguments: -l and users. See fig(d)

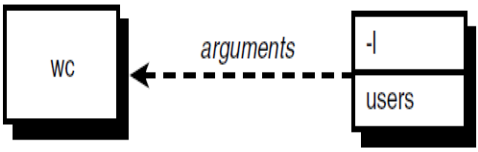


Fig: (d) Execution of wc -l users.

When wc begins execution, it sees that it was passed two arguments. The first argument, -l, tells it to count the number of lines. The second argument specifies the name of the file whose lines are to be counted. So wc opens the file users, counts its lines, and then prints the count together with the filename at the terminal.

Operation of wc in the second case is slightly different. The shell spots the input redirection character < when it scans the command line. The word that follows on the command line is the name of the file input is to be redirected from. Having “gobbled up” the < users from the command line, the shell then starts execution of the wc program, redirecting its standard input from the file users and passing it the single argument -l. See following fig(e)

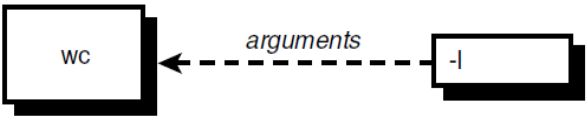


Fig: (e) Execution of wc -l < users.

When wc begins execution this time, it sees that it was passed the single argument -l. Because no filename was specified, wc takes this as an indication that the number of lines appearing on standard input is to be counted. So wc counts the number of lines on standard input, unaware that it's actually counting the number of lines in the file users. The final tally is displayed at the terminal—without the name of a file because wc wasn't given one.

Pipe in Linux:

A pipe is a form of redirection (transfer of standard output to some other destination) that is used in Linux and other Unix-like operating systems to send the output of one command/program/process to another command/program/process for further processing. The Unix/Linux systems allow stdout of a command to be connected to stdin of another command. You can make it do so by using the pipe character '|'.

Pipe is used to combine two or more commands, and in this, the output of one command acts as input to another command, and this command's output may act

as input to the next command and so on. It can also be visualized as a temporary connection between two or more commands/ programs/ processes. The command line programs that do the further processing are referred to as filters.

This direct connection between commands/ programs/ processes allows them to operate simultaneously and permits data to be transferred between them continuously rather than having to pass it through temporary text files or through the display screen. Pipes are unidirectional **i.e data flows from left to right through the pipeline.**

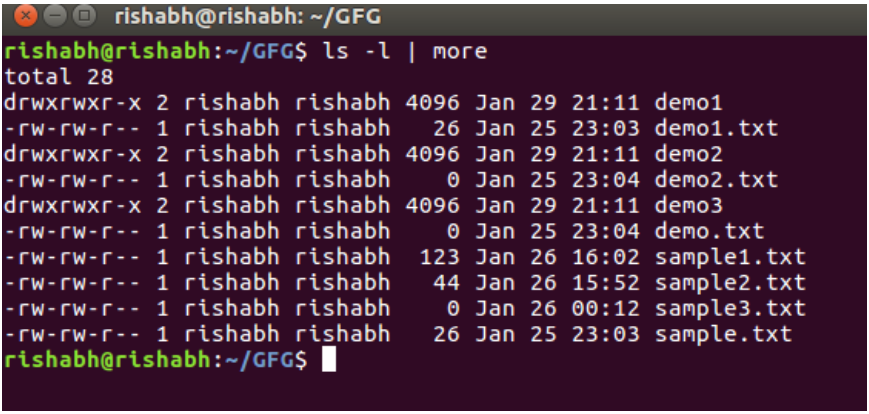
Syntax:

Example:

1. **Listing all files and directories and give it as input to more command.**

   $ ls -l | more

Output:



The more command takes the output of $ ls -l as its input. The net effect of this command is that the output of ls -l is displayed one screen at a time. The pipe acts as a container which takes the output of ls -l and gives it to more as input. This command does not use a disk to connect the standard output of ls -l to the standard input of more because pipe is implemented in the main memory.

In terms of I/O redirection operators, the above command is equivalent to the following command sequence.

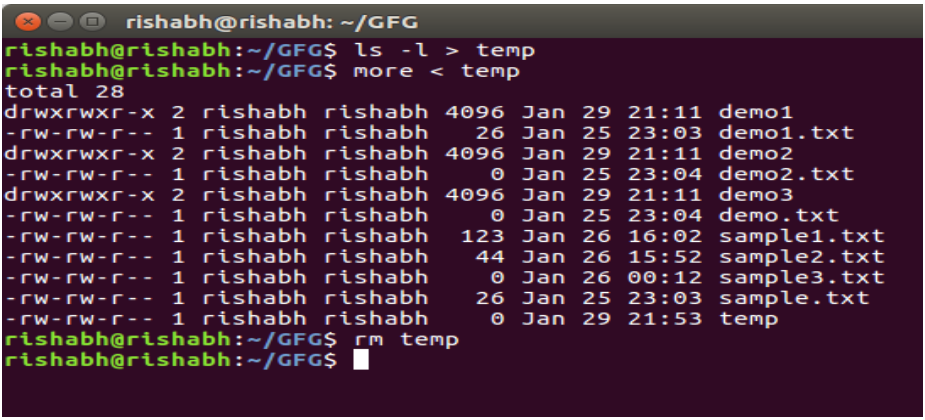
$ ls -l -> temp

more -> temp (or more temp)

[contents of temp]

rm temp

Output:



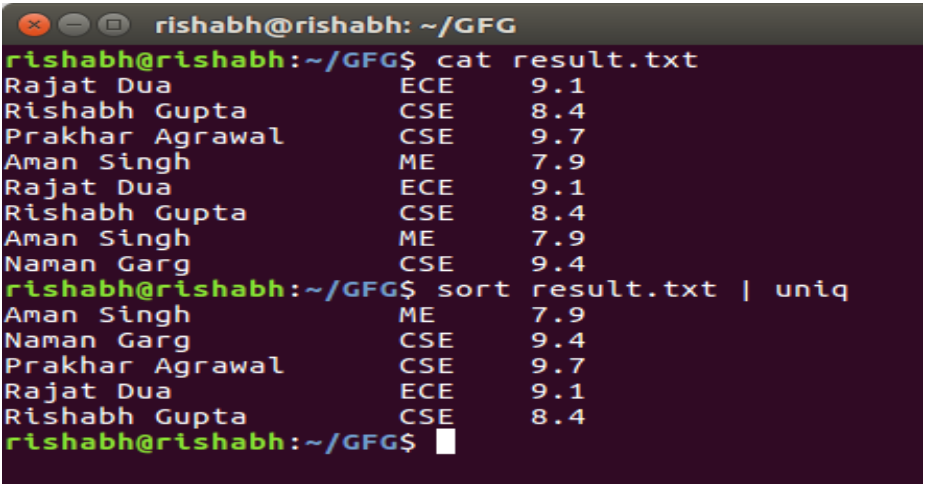
Output of the above two commands is same.

2. **Use sort and uniq command to sort a file and print unique values.**

   $ sort record.txt | uniq

This will sort the given file and print the unique values only.

Output:

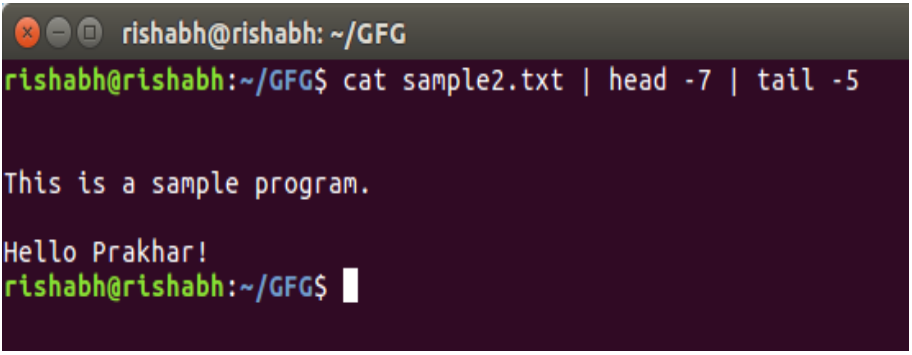


3. **Use head and tail to print lines in a particular range in a file.**

```
$ cat sample2.txt | head -7 | tail -5
```

This command select first 7 lines through (head -7) command and that will be input to (tail -5) command which will finally print last 5 lines from that 7 lines.

Output:

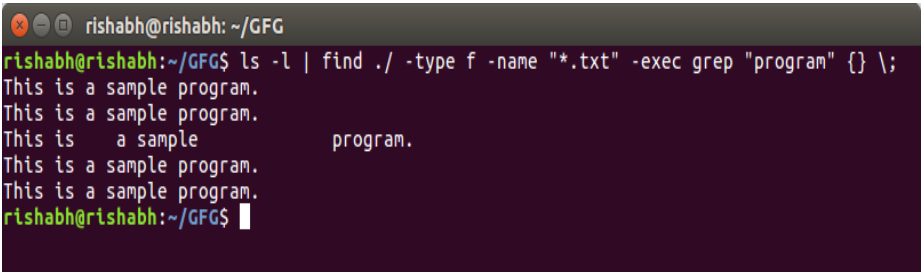


4. **Use ls and find to list and print all lines matching a particular pattern in matching files.**

   $ ls -l | find ./ -type f -name "*.txt" -exec grep "program" {} \;

This command select files with **.txt** extension in the given directory and search for pattern like “program” in the above example and print those in which have program in them.

Output:



5. **Use cat, grep, tee and wc command to read the particular entry from the user and store in a file and print the line count.**

   $ cat result.txt | grep "Rajat Dua" | tee file2.txt | wc -l

This command select **Rajat Dua** and store them in file2.txt and print the total number of lines matching **Rajat Dua**

Output: