Birla Institute of Technology & Science, Pilani
Second Semester 2014-2015, Computer Programming [CS F111]
Lab #3
Week #3

Objectives:

1. I/O Redirection
2. Filters
3. Piping
4. Introduction to Unix Shell and Shell Script

In UNIX the keyboard is defined as a standard input device and the computer monitor is defined as a standard output device. If a command is defined to take input from the standard input, it means it takes its input from the keyboard. Similarly, if a command gives its output to the standard output, it means it displays the output to the monitor. UNIX allows us to temporarily change the standard input and standard output by means of what is called as Indirection & Piping.

1. INDIRECTION OF OUTPUT

- Create a file named as ***file1*** using **vi editor**, write the following contents in the ***file1***:

  A programmer for whom computing is its own reward; may enjoy the challenge of breaking into other computers but does no harm; true hacker subscribe to a code of ethics and look down upon crackers.

- Create another file with a name ***file2*** using **vi editor**, write the following contents in the ***file2***:

  I am a white hat hacker. I have my own code of ethics

- Execute the following two commands:
  [f2012999@ prithvi ~] $ cat file1
  [f2012999@ prithvi ~] $ cat file1 > file3

  Observe the difference. The first command displays the contents of ***file1*** onto the standard output, i.e. the monitor. Instead, the second command redirected that output to another file, named ***file3***. Observe that this file did not exist but execution of the second command created it. You can look at the contents of ***file3*** using vi editor or cat command.
  Now execute the following command:
  [f2012999@ prithvi ~] $ cat file1 > file2

  Look at the contents of ***file2*** and observe that by executing the above command you have declared ***file2*** as the temporary standard output causing the contents of ***file1*** to be redirected to ***file2***. Since the file ***file2*** is not empty, it will be overwritten. To avoid overwriting the contents of ***file2***, instead of using **>** use **>>**. This will append the contents of ***file1*** to the contents of ***file2.*** Check it out yourself.

- Execute the following commands and observe the result
  [f2007999@ prithvi ~] $ cal 1 2010 > cal1
  Look at the contents of ***cal1***
  [f2007999@ prithvi ~] $ cal 2 2010 > cal2
  Look at the contents of ***cal2***
  [f2007999@ prithvi ~] $ cat cal2 >> cal1
  Again look at the contents of ***cal1***. Make a note of your observations

d2. INDIRECTION OF INPUT

- Execute the following two command and observe the result:
  [f2012999@ prithvi ~] $ cat
  [f2012999@ prithvi ~] $ cat file2
  In the first case, the cat command takes its input from standard input device, i.e. keyboard. It displays whatever you type on the keyboard. Press ctrl+c to terminate its execution.
  In the second case, the **cat** command takes its input from the file named ***file2*** and sends the result to the standard output (i.e. your monitor). You should note that this command does not take its input from standard input (keyboard); rather it takes the input from a file.

- Execute the following command and observe the result:
  [f2012999@ prithvi ~] $ cat < file2
  Do you find any difference between this command and the previous command? No difference, you are reading the contents of the file ***file2*** and outputting on the standard output (monitor), but the file ***file2*** doesn't become standard input for always, after the execution of the command UNIX will automatically make the standard input as keyboard.

- Execute the following command

Exercise -1 (Execute the following commands and record the observations)

Important – In place of whole prompt ***[f2012999@ prithvi ~] $*** only ***$*** symbol is used

Commands	**What does the command do?**
$ls > filelist	????
$ ls –l > longFileList	????
$date; cal 2 2010	????

[f2012999@ prithvi ~] $ cat < cal1 > testCal1
List the contents of your directory (use ***ls***), you will find a new file ***testCal1***. Look at the contents of file ***testCal1***.
Now execute the following command
[f2012999@prithvi ~] $ cat cal2 > testCal2
List the contents of your directory (use ***ls***), again you will find a new file ***testCal2***. Look at the contents of file ***testCal2***. So what difference do you find in previous two commands?

- The **indirection operators** and usage
  > filename - make file with a name ***filename*** as the standard output
  < file - make file with a name ***filename*** as the standard input
  >> file - make file with a name ***filename*** as the standard output, append to it if it exists

Now try to make out what the following commands achieve

$date; cal 2 2010; cal 3 2010	????
$(date ; ls) > complex	????

3. SOME MORE UNIX COMMANDS

Before moving on to filters and pipes, let us be aware about some useful Unix commands.

(a) wc (word count) - This command is used to count the number of lines, words and characters in a given file. Execute the command as shown below and observe the result.

[f2012999@prithvi ~] $wc file2

You will get the following output:

1 35 197 file2 (This means the file ***file2*** has 1 lines, 35 words and 197 characters)

This command can use options like –l, -w, -c to get the number of lines, words characters individually. Execute the following commands and observe the result:

[f2012999@prithvi ~] $wc –l file2
[f2012999@prithvi ~] $wc –c file2
[f2012999@prithvi ~] $wc –w file2

(b) sort - The **sort** program arranges lines of text alphabetically or numerically. Create a file with a name ***food.txt*** and write the following lines into that file:

Afghani Cuisine
Bangkok Wok
Big Apple Deli
Isle of Java
Mandalay

Execute the following command and observe the result

[f2012999@prithvi:~]$sort food.txt

- **sort** doesn't modify the file itself; it reads the file and writes the sorted text to the standard output
- Now redirect the output of **sort** command to a file ***sortedFood.txt***, execute the following command:
  [f2012999@prithvi:~]$sort food.txt > sortedFood.txt
- See the contents of file **sortedFood.txt**
- There are many options that control the sorting. **Use man command to see these options.** Two of these options are given below:

Option	Description
-n	Sort numerically and ignore blanks and tabs
-r	Reverse the order of sort

- Create a file ***student.txt*** and write the following contents into the file:
  2005A7PS634P : DENNIS Peter
  2007A3PS222P : Reedam Dwivedi
  2008A4PS212G : KIRTI Singh
  2007B4A7896G : Deepak SHARMA
  2005A2PS343H : Gaurav SINGH

Now look at the contents of file student.txt (use ***cat***)

To understand sort options in detail solve Exercise: 3 provide below

Exercise:3 (Execute the below mentioned commands and record your observations)

Command	Observation
$sort –n student.txt	
$sort –r student.txt	
$cat student.txt | grep "200" | sort	
$cat student.txt | grep "A7" | sort -r	

(c) head - The head command reads the first few lines of any file given to it as an input and writes them to the display. By default, head returns the first ten lines of each file name that is provided to it. For example, the following will display the first ten lines of the file named ***myFile.txt***

[f2012999@prithvi ~] $head myFile.txt

If more than one input file is provided, head will return the first ten lines from each.
Below is an example:

[f2012999@prithvi ~] $head myFile.txt file1

You can specify the number of lines that you want to display from the file by using an option ***–n***. This ***-n*** option is used followed by an integer indicating the number of lines desired. For example, the above example could be modified to display the first 2 lines from the file ***myFile.txt.***

[f2012999@prithvi ~] $head -n2 myFile.txt

We can have more files in the command, in which case the specified number of lines will be displayed from each file

[f2012999@prithvi ~] $head –n2 myFile.txt file1

(d) tail - The tail command is similar to the head command except that it reads the lines from the end of file.

[f2012999@prithvi ~] $tail –n3 myFile.txt

4. PIPES AND FILTERS

- The indirection operator (**|**) is called as pipe symbol, which helps to join two commands. Basically it feeds the output of one command as an input to other command. Two or more commands connected in this way forms a **pipe**.

- When a command takes its input from another command, performs some operation on that input, and writes the result to the standard output (which may be piped to yet another program), it is referred to as a **filter**.

Hence the commands that make up the **pipe** are called **filters**. One of the most common uses of filters is to modify output. Just as a common filter cuts unwanted items, the UNIX filters can be used to restructure output. Putting a vertical bar (|) on the command line between two commands makes a **pipe**. When a **pipe** is set up between two commands, the standard output of the command to the left of the pipe symbol becomes the standard input of the command to the right of the pipe symbol.

Solve the following exercise and the concept of pipe will become clearer to you

Exercise -2 (Execute the following commands and record the observation)

Commands	What does the command do?
$ ls | wc myFile.txt	
$ ls > myFile.txt | wc	

$ ls | wc > myFile.txt	
$ cat myFile.txt file1 | sort	
$ cat file1 | sort | head -1 | wc -w	

5. <u>GREP COMMAND</u> -

Now, let us make the concept of pipes and filters more clearly to us. For this, understand one more important command in Unix, i.e. Grep command.

The **grep** command searches a file(s) for lines that contains certain pattern. It can be used in a pipe so that only those lines of the input files containing a given pattern are sent to the standard output.

- Create three file with a name ***gp1.dat***, ***gp2.doc***, and ***gp3.txt***
- Suppose you want to search for files whose name contains these two characters **gp**. Execute the following command and see the output it should come something like what follows:
  [f2012999@prithvi:~]$ls -l | grep "gp"

```
-rw-r--r-- 1 f2009777 ug2009      0 2010-02-04 15:18 gp1.dat
-rw-r--r-- 1 f2009777 ug2009      0 2010-02-04 15:19 gp2.txt
-rw-r--r-- 1 f2009777 ug2009      0 2010-02-04 15:19 gp3.doc
```

<u>Exercise:1</u> (Execute the below mentioned commands and record your observations)	
Command	**Observation**
$ls -l | grep "dat"	
$ls -l | grep "txt"	
$ls -l | grep "doc"	

*Note: In place of **[f2012999@prithvi:~]$** only **$** is used here to save space, so please don't panic*

- **grep** command has several options which let you modify the search. Below is a list of some of the options.

Option	**Description**
-v	Print all lines that do not match pattern
-n	Print the matched line and its line number
-c	Print only the count of matching lines
-i	Match for both upper or lower case letters

To understand **grep** command options in detail solve Exercise: 2 provided below:

<u>Exercise:2</u> (Execute the below mentioned commands and record your observations)	
Command	**Observation**
$ls -l | grep –v "gp"	

Exercise:2 (Execute the below mentioned commands and record your observations)	
Command	**Observation**
$ls -l | grep –n "gp"	
$ls -l | grep –c "gp"	
$ls -l | grep –i "GP"	

6. INTRODUCTION TO UNIX SHELL AND PATTERN MATCHING

A UNIX ***shell*** is a program that understands the commands given by the user and allows the user to execute commands by typing them manually at a terminal. It also allows the users to write scripts which can execute these commands automatically; such scripts are known as ***shell scripts***.
Anything written to the right of the UNIX command is an argument passed to that command i.e. filenames or a piped command.

Look at the following examples:

- **$ls main.c**
  In this example **main.c** is the filename which goes as an argument to **ls** which does a long listing for this file name if at all it is present in the working directory.

The shell provides a mechanism for generating a list of file names that match a pattern.

For example,

- **$ls gp***

```
-rw-r--r-- 1 f2009777 ug2009      0 2010-02-04 15:18 gp1.dat
-rw-r--r-- 1 f2009777 ug2009      0 2010-02-04 15:19 gp2.txt
-rw-r--r-- 1 f2009777 ug2009      0 2010-02-04 15:19 gp3.doc
```

This command does a listing for all the file names in the current directory that start with **gp**. The character ***** also know a **wildcard** finds all the string patterns that are prefixed/suffixed with it.
*Note: If there is a directory with a name **gprog (i.e. directory name starts with gp)** then above command will display its contents.*

$ls *.txt

```
-rw-r--r-- 1 f2009777 ug2009      0 2010-02-04 15:19 gp2.txt
```

This command does a listing for all the file names in the current directory which ends with **.txt** extension.

The file name in UNIX has usually two parts separated by a dot (**.**) called as first name and extension. Let us consider a file name as ***filen.ext.*** Here ***filen*** is the first name of the file and ***ext*** is the extension which tells about the type of the file. Say **txt** (for text file), **doc** (document file), **c** (c program file) etc.

Following are the special characters interpreted by the shell called as *Wild cards*

Wildcards	**Meaning**
*	Matches any number of characters including null character
?	Matches a single character
[ijk]	Matches a single character either i,j or k.
[!ijk]	Matches a single character which is neither of i, j or k

[x-z]	Matches for a single character within this ASCII range of characters
[!x-z]	Matches for a single character not within this ASCII range of characters

(i) * stands for zero or more characters.

$ls gp (outputs file with a name **gp**, if exists)
$ls gp* (outputs all the files with a name that start with **gp**)
$ls * (outputs all the files in the working directory)

(ii) ? matches a single character

$ls a?t
Matches all those files of three characters with **a** as the first and **t** as the third character (and any character in between)
$ls ?le
Matches all three character files and those that end with **le**)

(iii) [ijk] matches a single character out of i, j, or k.

$ls [ijk].*
Matches all those files of which has first character as i, or, k.

Exercise:1 What will be output of the following commands?

Command	Observation
ls ??i*	Displays all files with name starting with any first two characters, i as the third character, and followed by any number of characters
ls ?	
ls *?	
ls ?*	
ls "*"	

7. QUOTING

Characters that have a special meaning to the shell, such as **< > * ? | &** ( ) are called **metacharacters**. Any character preceded by a **** is *quoted* and loses its special meaning, if any. The **** is elided so that

$ **echo \?**

will echo a single **?,** and

$ **echo **

will echo a single **\.** The **** is convenient for quoting single character. When more than one character needs quoting, the above mechanism is clumsy and error prone. A string of characters may be quoted by enclosing the string between single quotes. For example,

$ **echo xx'****'xx ;** *Is equivalent to $ echo xx******xx*

will echo

xx**xx**

8. SHELL SCRIPTS

Unix uses shells to accept commands given by the user. Shell accepts commands from the user, interprets and executes them. Instead of giving commands one by one, if it required to execute a set of commands together (in sequence), then we can store these commands in a text file and tell the shell to execute it. This is known as *shell script*.

Write the following commands in the file named as ***loc.sh***

```
date
ls –l | grep "gp"
echo "Lines in file food.txt containing the pattern java"
grep "java" food.txt
```

Save and execute the file using the command:
$ sh loc.sh

3.1 Performing Arithmetic operations

Type the following in the file named ***arithmetic.sh***. *Observe the operator (`), it is not a single quote operator. Rather this operator is found at the key in your keyboard along with (~) operator.*

```
a=20
b=12
echo `expr $a  +  $b`
echo `expr $a  -  $b`
echo `expr $a  \*  $b`  #for multiplication
echo `expr $a  /  $b`
echo `expr $a  %  $b` #for modulus division
echo `expr \( $a \* \( $a + $b \) / 2 \)` # to compute (a*(a+b)/2)
```

Now execute this script file by using the command

$ **sh arithmetic.sh**

You will get answer as follows:

```
32
8
240
1
8
320
```

Note: Line 1 and 2 assigns some values to variables a and b. In the line echo ***echo `expr $a + $b`***
expr is the key word which causes the expression to be evaluated. Anything follows # is a comment and will be only for improving readability.