CHARUTAR VIDYA MANDAL'S
SEMCOM
Vallabh Vidyanagar

Faculty Name: Ami D. Trivedi
Class: TYBCA (Semester-V)
Subject: US05EBCA01 (Basics of UNIX Operating System)

*UNIT – 2 (The Bourne Again Shell -bash)

PROMPTS

Prompt is a string consisting of one or more characters, showing the position of cursor. The appearance of a prompt generally indicates that previous command has completed its run. The prompt can be customized by setting the value of PS1 or prompt variable.

COMMAND LINE

Refer command line from Unit-1

ESCAPING AND QUOTING

Escaping

It's a generally accepted principle that filename should not contain the shell metacharacter. But let's see what happen if we do so.

Imagine a file named chap* created with the > symbol.

```
$ echo > chap*          Note: use chap\* if chap* does not work
$_
```

The silent returns of the prompt suggest that the file has been created. A suitable wild-card pattern used with ls confirms this.

```
$ ls –x chap*
chap   chap*   chap01   chap02   chap03   ….chapx   chapy   chapz
```

There is ia file with the name chap* in the current directory. The wild-card pattern matched this file along with the others.

This file can be a great nuisance and should be removed immediately. But that won't be easy. You can not use rm chap* because that would remove all files in this list, and not remove this one i.e. chap* only.

How do you remove this remove this file then, without deleting the other file? To make this possible the shell has to treat the asterisk literally instead of interpreting it as a metacharacter.

The answer lies in the \ (backslash), yet another metacharacter. Backslash removes meaning of any metacharacter placed after it.

Use the \ before the * and it solves the problem:

```
$ ls –x chap\*          Note: this will match chap*
chap*
$ rm chap\*
$ ls –x chap\*
chap* not found
```

The expression chap* literally matches the string chap*.This is necessary feature provided by the shell, and this concept can be extended to another area also.

The use of the \ in removing the magic from any special character is called **escaping or despecialization**.

If you have the files chap01 chap02 chap03 in your current directory, and then create a file chap[1-3] by using

```
$ echo > chap[1-3]
```

then you should escape the two rectangular bracket when accessing the file:

```
$ ls –x chap0\[1-3\]
chap[1-3]
$ rm chap0\[1-3\]
$ ls –xchap0\[1-3\]
chap0[1-3] not found
```

Sometimes, you would need to escape the \ character itself. Since the shell treats this as a special character, you need another \ to escape it:

```
$ echo \\
\
$ echo The newline character is \\n
echo The newline character is \n
```

Quoting

It is another way to turn off (suppress) the meaning of a special character. When a command argument is enclosed in quotes, the meanings of all enclosed special characters are turned off.

E.g.:

```
$ echo ‘*?[8-9]’      Note: We can use double quoters also
*?[8-9]
```

The argument above is said to be quoted.

Double quote would also have served purpose. But in some cases, use of double quotes does allows interpretation of some of special character (especially the $ and ` - the backquote).

For a beginner, single quotes are the safest as they protect all special characters except quote themselves.

Quoting Preserve Spacing

The space is another character that has special meaning to the shell. When shell finds contiguous spaces and tab in the command line, it compresses them to a single space.

When you issue the following echo command, you will find all space compressed.

```
$ echo The shell          compress    multiple     space
The shell will compress multiple spaces.
```

The above arguments to echo could have been preserved by escaping the space character wherever it occurs at least twice.

```
$ echo The shell \ \ \ compress \ \  multiple \ \ space”
The shell          compress    multiple     space
```

When we want to protect large number of characters from the shell, quoting is more preferable to escaping.

```
$ echo “The shell          compress    multiple     space”
The shell          compress    multiple     space
```

We used double quotes this time. It worked well. Quotes also protect the \.

```
$ echo ‘\’
\
```

INTERNAL AND EXTERNAL COMMANDS

Shell recognizes 3 types of commands.

1. **External command**

   The most commonly used ones are the UNIX utilities and program like cat, ls and so on. The shell creates a process for each of these commands that it executes and remain its parent.

2. **Shell Scripts**

   The shell executes these scripts by spawning (initiating) another shell (i.e. a sub shell). This sub shell executes the commands listed in the script. The sub shell becomes the parent of the commands that feature in the script.

3. **Internal command**

   Shell is a programming language on its own. So the shell has a number of built in commands. Some of them like cd and echo don't generate a process. They are executed directly by the shell. Similarly, variable assignment with the statement x=5 doesn't generate a process.

PATH

Many UNIX commands use file and directory names as arguments. It is assumed that these files and directories are present in current directory. For eg. The command

```
cat script1.sh
```

Note: Our home or login directory is /home/student

will work only if the file script1.sh exists in current directory. Suppose you are in /faculty directory and you want to access script1.sh which is placed in /home/student. We can not use **cat script1.sh** command.

There are 2 ways to access this file from your current directory /faculty.

1. **With absolute pathname**
2. **With relative pathname**

1. Absolute Path

It uses root directory as the ultimate reference for the file. All path reference here originate from root.

We have to use following command from current directory i.e. /faculty to display script1.sh. This command makes use of an absolute pathname for script1.sh.

```
$ cat  /home/student/script1.sh
```

Here cat command makes use of absolute pathname. Location of script1.sh is specified with reference to the root (the first /).

When you have more than one / in the pathname, for each such / you have to descend one level in the file system. Thus student is one level below home, and two levels below root.

Note:

No two files in UNIX file system can have identical absolute path name.

If you have two files with same name, they must be in different directories. It means that their pathnames will also be different.

2. **Relative Path (. and ..)**

It use current directory as a point of reference and specify the path relative to it.

Absolute path method can be tedious when absolute path name is long i.e. when you are located at number of generations away from the root. For this UNIX offers a shortcut – the Relative pathname.

. (a single dot) - This represents the current directory
.. (two dots) - This represents parent directory.

We can use **..** to frame relative pathnames.

```
$ pwd
/home/student/progs
$ cd..                    Note: Moves one level up
/home/student
$ pwd
/home/student
```

This method is compact and more useful when ascending the hierarchy. The command **cd..** means change your directory to the parent of current directory.

We can combine any number of **..** separated by /. When / is used with **..**, it moves one level up.

```
$ pwd
/home/student/progs
$ cd ../..                  Note: Moves two level up
/home
```

Now, how to move from /home/student to /home/faculty ?

```
$ pwd
/home/student
$ cd../faculty
$ pwd
/home/faculty.
```

So weather you should use an absolute or a relative pathname depends solely on the comparative number of keystrokes required to describe the pathname.

SHELL VARIABLES

Shell variables are an integral part of shell Programming. They provide the ability to store and manipulate information within a shell program.

The variables you use are completely under your control. You can create and destroy any number of variables as needed to solve the problem at hand.

The rules for building shell variables are as follows:

1. A variable name is any combination of alphabets, digits and an underscore('_')
2. No commas or blanks are allowed within a variable name.
3. The first character of a variable name must either be an alphabet or an underscore.
4. Variable names should be of any reasonable length.
5. Variable names are case sensitive.

Here are few sample variable names: si_int, hra, mark_1

It is good practice to use this huge choice in naming variables by using meaningful variable names. It is always advisable to construct meaningful variable names like rollno, basic_salary etc.

BASIC COMMAND LINE PROCESSING

It is necessary to understand the sequence of steps which are followed when a command is processed.

After the command line is terminated by pressing [Enter] at prompt, the shell does processing of command line in one or more passes (steps). The sequence of processing is as follows:

1. Parsing
2. Variable evaluation
3. Command substitution
4. Redirection
5. Wild card interpretation
6. Path evaluation

1. **Parsing**

   Shell first breaks up the command line into words using spaces and tabs as delimiters. Quoted words are not broken. All consecutive occurrences of a space or a tab are replaced with a single space.

2. **Variable evaluation**

   All words preceded by a $ are evaluated as variables. Quoted or escaped words preceded by $ are not evaluated.

3. **Command substitution**

   Any command surrounded by backquote is executed by shell and its output is inserted at the place where it found in command.

4. **Redirection**

   The shell looks for characters >, < and >> to open the files pointed by these characters.

5. **Wild card interpretation**

   The shell scans the command line for wild cards and replaces them with a list of filenames that matches the pattern.

6. **Path evaluation**

   It finally looks for the PATH variable to determine the sequence of directories to be searched to follow the command.

USING THE ECHO COMMAND

Refer echo command from Unit-3.

A QUICK INTRODUCTION TO BASIC FILTERS – cat AND cut

cat - displaying & creating files

cat is universal file viewer and it is one of the most well-known command of UNIX file system.

It is mainly used to display the content of a small file on the terminal. As usually before you copy or remove files, you will often want to see the content.

```
$ cat note1
This is file containing simply this statement.
```

Like many other UNIX commands cat also accepts more than one filename as an argument.

```
$ cat note1 note2
```

Here content of second file will be shown immediately after the first file without any heading. Here cat concatenates content of both the files.

If you have non-printing ASCII characters in your input, you can use cat with –v option to display these characters.

Using cat to Create File

cat is also useful for creating a file.

```
$ cat >  test1
cat used in this way represents a
rudimentary (simple or basic) editor.
[ctrl+d]
$_
```

When the command line is terminated after hitting enter, the prompt vanish and cat waits to take input from the user. Finally press [ctrl+d] to signify the end of input to the system. The file is written and the prompt returned. To verify this simply 'cat' this file.

```
$ cat test1
cat used in this way represents a
rudimentary (simple or basic) editor.
```

cut – Slitting (cutting) a file vertically

While head and tail commands are used to slice file horizontally, you can slice a file vertically with cut command. cut identifies both column and fields. We'll take column first.

Cutting column (-c)

Let's see use of cut to extract the first four column of the group file. This will use –c option followed by the column specification.

friend	bharat:male:Delhi ganga:female:Bombay yagna:male:Calcutta shanti:female:Chennai

```
$ cut –c1-5 friend
bhara
ganga
yagna
shant
```

The specification –c1-4 cuts columns 1 to 4.

We can also use cut with more than one column specification. Ranges are permitted and commas can be used to separate column chunks:

```
$ cut –c -3,6-22,28-34,55- relatives
```

Observe that cut also use a special form of selecting a column from beginning and upto end of a line. The expression 55- indicates column number 55 to end of the line. Similarly, -3 indicate 1-3.

Cutting fields(-f)

The -c option is useful for fixed length lines. Most UNIX files don't contain fixed length lines. Hence you'll need to cut fields rather than columns.

cut uses the tab as the default field as delimiter, but can also work with a different delimiter.

We require two options here: **-d** for delimiter and **–f** for specifying the field list. This is how you can cut out first and third field.

$ cut –d: -f1,3 friend OR **$ cut –d ":" -f1,3 friend**

So when you use the –f option, you shouldn't forget to use the –d option too, unless the file has default delimiter (the tab).

cut can be used to extract the first word of a line by specifying the space as the delimiter.

$ who | cut –d " " –f1

So cut is powerful text manipulator often used in combination with other commands or filters.

THE BUILDING BLOCKS APPROACH

UNIX is popular with programmers for a variety of reasons.

A primary reason for its popularity is the building-block approach, where a suite of simple tools can be streamed together to produce very sophisticated results.

Another reason is the philosophy that 'everything is a file', which means that a standardized set of operations and functionality can be performed on different file types (directories vs. regular files), hardware devices, and even system processes.

INPUT/OUTPUT REDIRECTION

Many commands send their output to terminal. Command like cat also takes input from the keyboard. These commands designed to use a **character stream**. A stream is just a sequence of bytes that many commands see as input and output.

UNIX treats these streams as files. Group of UNIX commands reads from and writes to these files.

Shell sets up 3 standard files for input, output and error.

1. Standard file for input is known as **standard input**.
2. Standard file for output is known as **standard output**.
3. Standard file for error is known as **standard error**.

Shell attaches these files to user's terminal while logging in. Any program that uses streams will find them open and available.

Shell has set some physical devices as defaults:

- Standard input – Default source is keyboard
- Standard output – Default destination is terminal
- Standard error – Default destination is terminal

Instead of taking input from keyboard it can be taken from any disk file or some other device. And instead of output and error going to terminal, it can go to any disk file or some other device.

1. Standard Output

Commands like cat and who send their output as a character stream. This stream is called **standard output** of the command.

3 destinations of standard output:

- Terminal is first and default destination of standard output.
- Disk file is second destination.
- Input to other program (through pipe) is third destination.

By default it appears on the terminal. Using the symbols **> and >>**, we can redirect the output to a disk file.

<u>Write to '>'</u>

Syntax: command > filename

A statement like this says to LINUX to put whatever comes out of command (except errors) and place it in file filename.

Example
$who > newfile

Shell looks at the > symbol and understands that standard output has to be redirected. It will open the file **newfile**, writes the stream into it and then closes the file. Nothing appears on screen.

If the output file does not exist then shell creates it before executing the command. If the output file exists then shell overwrites it.

<u>Append to '>>'</u>

Syntax: command >> filename

>> symbol take the output from this command and attach (append) it to the end of this file. We can use this symbol to retain the old content of output file.

Example
$who >> newfile   Note: Doesn’t disturb existing content of newfile.

If the output file does not exist then shell creates it before executing the command. If the output file exists then the output will be appended to end of **newfile**.

2. <u>Standard input</u>

Some commands are designed to take their input as a stream. This stream represents the **standard input** to a command.

3 sources of standard input:

- Keyboard is default source.
- A file using redirection with <.
- Another program using a pipeline.

$wc
Above command with no filename expects input from the keyboard because file name is omitted here. Input can be redirected to originate from a file.

<u>Redirect from '<'</u>

The single redirect symbol (<) reassigns the standard input of the program to the left of the <. So, for example:

Syntax: command < filename

Example

$wc < emp.lst	**$wc emp.lst**
3 9 27	3 9 27 emp.lst

Here with the use of < (left chevron), shell redirects **wc**’s standard input coming from file **emp.lst.** wc will not open the file emp.lst. wc without < will open the file emp.lst.

Why to redirect the standard input from a file if command can read the file itself? When you need to keep the command ignorant about source of its input then we can use <.

3. Standard error

When you enter an incorrect command or try to open a nonexistent file, some diagnostic message will be displayed on screen. This is **standard error** stream.

Destination for standard error is terminal.

Example

```
$cat stud.lst
cat : cannot open stud.lst: No such file or directory
```

COMMAND SUBSTITUTION

The shell allows the argument of a command to be obtained from the standard output of another. This feature is called command substitution.

In simple word, output of one command can become input for another command. Here, pipe (|) connects the standard output of one command to standard input of another.

Consider a simple example. Suppose we want to echo today's date with statement like this:

```
The date today is Wed Sep 10 10:10:10 EST 2012
```

Now the last part of the statement beginning with Wed represents the output of date command. How can we incorporate this date command into echo command?

```
$ echo The date today is `date`
The date today is Wed Sep 10 10:10:10 EST 2012
```

When scanning the command line, the `(backquote) is another metacharacter that the shell looks for.

There is special key on your keyboard that generate this character, and should not be confused with single quote (').

The shell than executes the enclosed command, and replace the enclosed command line with the output of the command.

For command substitution to work, the command "backquoted" must be standard output. date does; that's why command substitution worked.

You can use this features to generate useful messages. For example you can use 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
```

The command worked properly even though the arguments were double-quoted. It's a different story altogether when single quotes are used:

```
$ echo ‘There are `ls | wc -l` files in the current directory
There are `ls | wc -l` files in the current directory.
```

We encounter the first difference between the use of single and double quotes.

So, Command substitution has interesting applications possibilities. It speeds up work by letting you combine a number of instructions in one.

Disclaimer

The study material is compiled by Ami D. Trivedi. The basic objective of this material is to supplement teaching and discussion in the classroom in the subject. Students are required to go for extra reading in the subject through library work.