6 Customizing the Environment

6.1 Shell variable and naming rules

In Unix, a user can create shell variable as per their requirement. A variable created within the shell is known as shell variable. A user can create or even destroy shell variable. It is also used to customize the shell which is allocated at the time of login. A user has to make attention during the naming of shell variable.

(i) A variable name begins with either alphabet (i.e. a-z or A-Z) or underscore (i.e. '_').
(ii) After initial letter, it consists of any combination of alphabets, digits (i.e. 0-9) and underscore.
(iii) In Unix, variable name is case-sensitive. For example, variables *a1* and *A1* are considered as different.
(iv) Comma and blank spaces are not allowed.
(v) It is recommended to avoid special characters like *,?,&,$ etc...in variable name.

6.2 Types of variable

Basically, variables in Unix are of two types: User defined variables and system/environmental variables.

6.2.1 User defined variable

A variable created by user known as user defined variable. Variables are widely used in the shell scripting. Like any other programming languages, shell allows you to define and use variables in both command-line and shell script. However, no type declarations or initialization are required to use them. A user can create a shell variable by typing generalized code at command prompt or in shell script as follow:

variable_name=[value/expression/command]

The generalized code consist of *variable_name*, *equal sign(i.e. =)* and *value* or *expression* or *any UNIX command*. An entity after equal sign is optional. There is no space on either side of equal sign.

✓ A user can create a variable in the current shell like this:

$myvar=1234 *#No space on either side of equal sign*

By default, any variables created in shell are of string type. That means, the value is stored in ASCII rather than in binary format. In above example, variable *myvar* is string type and store number 1234 as characters 1, 2, 3 and 4.

✓ Variables created within the shell are automatically removed as soon as shell is expired. Similarly, variables defined within the script are created as soon as script is started and removed when the script execution will be over.

✓ When the shell reads the command line, it interprets any word preceded by a $ as a variable, and replaces the word by the value of the variable.

To display the contents of *myvar* variable, a user can use **echo** command like this:

$echo $myvar <enter>

```
1234
$
```

✓ In Unix, if you try to evaluate value of an unassigned or undefined variable then it returns a null string.

e.g. `$echo $xyz<enter>`

```
                    #display only blank-line appended by echo
$
```

It displays nothing on a screen because variable *xyz* is not defined in the shell. Here, you can see a blank-line which is appended by **echo** command.

✓ A user can assign null string explicitly to a variable by any one of the following statement:

```
$x=<enter>          or
$x="<enter>         or          #single quote only
$x=""<enter>                    #double quote only
```

✓ A user can also assigned a string value to a variable and display it on the screen as follows:

```
$sname=Kush <enter>
$echo $sname <enter>
Kush
$
```

✓ A multiple-word string is also assigned to a variable by enclosing it in either single or double-quote as follow:

```
$name="Nirva Patel"<enter>      #multiple-words string must be
                                #enclosed within the quote
$echo $name <enter>
Nirva Patel
$
```

✓ The shell offers another notation, a variable name enclosed within a pair of curly braces ({}), to evaluate variable.

e.g.
```
$echo ${sname} <enter>
Kush
$
```

✓ This notation (pair of curly braces) is useful to combine the string with a value of a variable.

e.g.
```
$echo ${sname}Patel <enter>
KushPatel
$
```

✓ It is possible to concatenate value of variables by writing them adjacent to one another without any space between them.

e.g.
```
$x=Kush ; y=Patel
$z=$x$y             #joins value of x and y and stores it in z
$echo $z
KushPatel
$
```

✓ Variables are also used to speed up your interaction with the system. For example, suppose you have to move to the directory */home/bca/mydir* several times during the session, then assign this path to a shell variable and use it with the **cd** command like this:

```
$path='/home/bca/mydir' <enter>#path-name in quote
```

```
$cd $path <enter>          #switch to directory /home/bca/mydir
$
```

This feature reduces a lot of typing work and typing error.

- ✓ It is possible to assign output of Unix command to a variable. The command substitution feature allows you to set variable by the output of Unix command.

```
e.g.  $allfiles=`ls`       #command enclosed within back quotes
      $echo $allfiles      #display all files of current directory.
```

- ✓ The shell also offers a facility to remove variables from the shell. A user can remove shell variables using **unset** command as follow:

```
unset var1 [var2 var3...]
```

For example, to remove shell variables *a1*, *a2* and *a3*, then the command is:

```
unset a1 a2 a3
```

- ✓ Like any other programming language, Unix offers a feature to make a variable read only. A **readonly** keyword makes a variable read only as follow:

```
$sname=Kush                #or   readonly sname=Kush
$readonly sname
```

After making a variable read only, its value can not be altered by user. All read only variables are listed by issuing the command **readonly** at the shell prompt.

```
$readonly<enter>           #list all readonly variables
declare -r sname="Kush"
$
```

A user cannot remove **readonly** variable(s) explicitly from the shell. It removes automatically from the shell whenever user logged out from the Unix system.

NOTE: set command display list of user as well as system defined variables.

6.2.2 Environmental variable

When you start a Unix system, some variables are set at the time of system startup and some are after logging in. These variables are known as environmental variables. They are also known as Unix-defined or System variables. A user can control the Unix system using these variables. The environmental variables are available to the base/parent shell as well as any number of new sub-shell created under the base shell. In short, environment variables are inherited by the sub-shell from its base shell. The scope of the environment variables are extended to user environment,i.e the sub shell that runs shell script, Unix utilities such as **mail**, **vi** etc.

In Unix, most of the environmental variables are in upper-case letter. *HOME, PATH, PS1, PS2, IFS, LOGNAME* etc... are an example of the Unix environmental variable. A user can view all these variables by applying the **env** command or **export** statement at the shell prompt. Let us discuss some of the environmental variables in the following section.

(i) HOME:

HOME variable contains the home directory path as its value. This is the default directory of user and will be placed on it at the time of logging in. A user can view a value of this variable using **echo** command like this:

```
$echo $HOME <enter>
/home/bharat
$
```

It displays an absolute pathname of user's home directory. The value of this variable is set from the file */etc/passwd* during the login time of user. A user can not change his home directory. If a user changes the value of *HOME* variable, then it will not change the home directory of the user but switches to this directory when **cd** command used without arguments. This means that, if user apply a **cd** command without any argument, *$HOME* append implicitly i.e. *cd $HOME* which is equivalent to **cd** without argument.

✓ For example, a user set *HOME* variable like this

```
$HOME=/home/bharat/script
$
```

Now, if user applies **cd** command at command line, every time he switches to */home/bharat/script* directory instead of user's home directory.

(ii) PATH:

It is a variable that provides a search path to locate any executable command submitted at command line. *PATH* variable contains list of directories that are searched by the shell to locate a command. A user can know the search path for executable file or command as follow:

```
$echo $PATH<enter>
/usr/local/bin:/bin:/usr/bin
$
```

It displays list of search path for executable file. Each pathname is separated by delimiter colon (i.e. :). DOS and Windows users also use a *PATH* variable which is stored in AUTOEXEC.BAT file to specify the search path, but the field delimiter here is the semi-colon (i.e. ;). If user wish to include the directory */home/bharat* in a search list, then the variable is redefine as follow:

```
$PATH=$PATH:/home/bharat <enter>
$ echo $PATH<enter>
/usr/local/bin:/bin:/usr/bin:/home/bharat
$
```

(iii) IFS:

IFS contains a sequence of characters that are used as word separators during the interpretation of the command line. It contains invisible characters like the space, tab and new-line. A user can view the content of this variable by taking the octal dump as follow:

```
$echo "$IFS" | od–bc <enter>        #quote must be required to display value
0000000      040    011    012    012
                    \t     \n     \n
0000004
```

The output denotes that an ASCII octal value 040 specify space character. \t and \n represent the tab and new-line character respectively. Last new-line character is appended by **echo** command.

(iv) PS1:

PS1 variable contains the system prompt i.e. the $ symbol. This is the primary prompt of the system. The system prompt may be changed by setting the value of this variable to the desired prompt.

e.g.

```
$PS1="HELLO> " <enter>
HELLO>                              #new prompt
```

The bash shell uses number of escape sequences to make prompt string more informative. Some of these

Table-(a.6): Escape sequence characters

Character	meaning
\u	The username of the current user
\h	Hostname of computer
\W	current directory relative to the home directory
\@	current time in am/pm format
\t	current time in HH:MM:SS format
\T	current time in 12-hour format
\s	Name of the shell

✓ For example, a user set a value of *PS1* as follow:

```
$PS1="\@?"
02:22 PM?                    #new prompt.
```

It shows a new prompt which display current time in am/pm format followed by '?' character.
The *PS1* variable can be set to a new value only at the UNIX prompt and not within a shell script.

NOTE: The primary prompt for system administrator is '#' sign. That means, when a system administrator logged in then he gets "#" prompt instead of "$".

(v) PS2:

It contains secondary prompt string. If commands have a lengthy syntax then we have to continue its syntax into multiple lines at command line. In that case, the shell will issue a secondary prompt, usually >, which indicates that the command line is not complete.

e.g.

```
$echo "This is
> a three-lines                #secondary prompt (>) appears
> text message"<enter>
This is
a three-lines
text message
$
```

✓ A user can change secondary prompt by issuing the following command as:

```
$PS2=<cont...>
$echo "This is a first line
<cont...>This is a second line
<cont...>This is a last line"<enter>
This is a first line
This is a second line
This is a last line
$
```

(vi) LOGNAME:

This variable contains the login/user name of the user. This variable is also set when a user is successfully logged in to the Unix system. A user can view his login name by applying following command at shell prompt.

```
$echo $LOGNAME <enter>
```

```
nirva
$
```

Here, *nirva* is a login name of the current user.

(vii) SHELL:

This variable is also set at the time of logging in. The value of this variable is set from file */etc/passwd*. *SHELL* determines the type of shell that a user sees on logging in. It stores the name of user's login shell. It should be noted that both login shell and current shell may be same or different.

- ✓ A user can view his login shell by issuing command as follow:

```
$ echo $SHELL          #display login shell
/bin/bash
$
```

The output denotes that login shell of the user is *bash* which is stored in */bin* directory.

- ✓ A user can display name of his current shell as follow:

```
$ echo $0              #display current shell
bash
$
```

The output shows that the current shell is also *bash*.

- ✓ Consider the following command sequences in which both login shell and current shell must be different.

```
$ echo $SHELL          #display login shell
/bin/bash
$ csh                  #switch to another shell
$ echo $0              #display current shell
csh
$
```

(viii) PWD:

It contains absolute pathname of current directory. The value of this variable is changed as soon as a user switches to another directory. A user can view the value of this variable using **echo** command as follow:

```
$ echo $PWD
/home/bharat/y2013
$
```

(ix) MAIL:

It contains absolute pathname of user's mailbox file. A user can view the value of this variable using **echo** command as follow:

```
$ echo $MAIL
/var/spool/mail/bharat
$
```

The output denotes that */var/spool/mail/bharat* is a mailbox location of user *bharat*. Here, the mailbox filename is same as the name of username. That means all off-line messages sends by any users will be stored in the file *bharat* under the directory */var/spool/mail*. The mailbox location is varied from system to system.

(x) MAILCHECK:

It stores mail-checking interval for incoming mail. The value of this variable is in terms of seconds.

```
$ echo $MAILCHECK
60
$
```

It displays 60 which indicate that after 60 seconds the shell checks the file for the arrival of new mail. If the new mail is arrived, it informs user about new mail by the message as follow:

You have new mail in /var/spool/mail/bharat

Here, if a user *bharat* is running a command, he will get new arrival mail message on his terminal only after the command has accomplished its task.

(xi) HISTSIZE:

It indicates number of last executed commands stored in memory.

```
$ echo $HISTSIZE
1000
$
```

The value of *HISTSIZE* indicates that last 1000 commands executed at command-line will be stored in the history file. In bash, history file is stored in home directory of user and the name of history file is *.bash_history*. A user can view the content of history file using the **cat** command.

6.3 Aliases

Alias is a nick name of any existing Unix command. In other words, alias is a facility provided by Unix using which a user can give a shorthand new name to the existing command. This feature is useful to customize a command. There are some advantages to use this feature such as a user can give alias name to the commands which are most frequently used and are lengthy. So, it reduces typing work and typographical errors.

In Unix, **alias** statement is used to create a new name of existing command. The syntax of alias command is as follow:

Syntax:

alias [argument]

Here, argument is optional. It should be an alias name or an alias name with definition.

✓ Without argument, **alias** display definition of all the alias names.

```
$ alias
alias cp='cp -i'
alias ls='ls -F --show-control-chars --color=auto'
alias md='mkdir'
alias mv='mv -i'
alias rd='rmdir'
alias rm='rm -i'
$
```

✓ An alias name is created with definition like this:

```
$ alias mydir='cd /home/bharat/script'     #creates alias mydir
$ pwd
/home/bharat
$ mydir
$ pwd
/home/bharat/script
$
```

The output denotes that an alias name *mydir* is created. It should be noted that the definition of alias must

be enclosed either in single or double quote.

- ✓ A user can view a definition of specific alias like this:

```
$ alias mydir
alias mydir='cd /home/bharat/script'
$
```

- ✓ A user can create an alias in an alias definition

```
$alias dir="ls -l"
$ alias pgdir="dir|more"
$ pgdir
```

It creates an alias *dir* which display files of current directory, each filename in separate line. The second line creates another alias named *pgdir* which uses alias *dir* in its definition. Now, pgdir displays file listing page-wise.

- ✓ A user can pass argument to an alias command like this:

```
$dir f*                    #display all files that begins with 'f'
```

Here, alias name *dir* uses argument *f** to display files of current directory that begins with character 'f'.

- ✓ An ambiguous alias command display wrong output

```
$pgdir f*                  #generates wrong output
```

- ✓ A user can unset an alias with the **unalias** statement. The general form of **unalias** statement is:

Syntax:

```
unalias [-a] name [name ...]
```

Without any argument, **unalias** statement display usage of it.

- ✓ A user create alias should be removed like this:

```
$unalias dir               #removes alias dir
```

- ✓ A user can remove system defined aliases using –a option like this:

```
unalias –a                 #removes system defined as well as user-define aliases
```

6.4 command history

We know that Bourne shell does not supports history feature whereas other shells like bash shell, C shell and Korn shell provides it. The advantage of this feature is that a user can recall previously typed commands, edit them and re-execute it.

The **history** command displays list of commands which are executed on command line. A bash user's get last 1000-commands executed at the command prompt. Table-(b.6) shows the history commands used in bash shell.

Table-(b.6): history command

Command	Meaning
history 5	It displays last 5-commands
!!	It repeats last command.
!5	It repeats event number 5
!5:p	It prints only event number i.e. command not its output.
!?man?	It repeats last command in which pattem 'man' anywhere in the command.
!cat	It repeats last command that begins with cat

- ✓ A user can list last 5-commands as follow:

```
$ history 5
 1002  sort f1|uniq -u
 1003  cut -d'' -f1 f1
 1004  paste f1 f2
```

```
1005  cmp f1 f2
1006  history 5
$
```

The output shows a number before each command known as the event number. Every commands executed on command line are added to history list.

- ✓ A user can run specific event using exclaimation mark (!) as follow:

```
$ !1005
cmp f1 f2
f1 f2 differ: byte 7, line 1
$
```

It executes event 1005 which is *cmp f1 f2*.

- ✓ A user can run last command that begins with cut as follow:

```
$ !cut
cut -d' ' -f1 f1
hello
hello
unix
data
vb.net
$
```

There are many commands used in history. Here, we discussed few of them.

6.5 Exporting shell variables

We can create any number of shell variables within the shell. The scope of these shell variables is limited to the shell in which it is defined. They are not accessible in it's beneath child shell. So, as per scope of the user-defined shell variable, it is further sub-divided into two categories: local variable and global variable.

Local variable

A user can create a sub-shell or child shells within the shell by typing **sh** at shell prompt. When a variable is defined, it is known only to the shell which created it. If a new shell is created by typing **sh**, this shell is unaware of shell variables of its parent shell. This variable is called a local variable.

A value of variable is accessible only in the shell in which it is created known as local variable. In other words, a variable whose scope is limited to current shell is known as local variable. It is not known by child shell. The following example illustrates the this concept:

```
$city=surat
$echo $city
surat
$sh                    #create a new shell
$echo $city            #display nothing
$city=Baroda           #Give new value 'Baroda' to city
$echo $city
Baroda
$clrl + d              #return to parent shell
$echo $city            #parent unaware of 'Baroda'
```

```
surat
$sh                    #create a child shell
$echo $city            #value 'Baroda' destroyed
$ctrl + d              #Return to parent
```

It is obvious from the above statements that local variables are removed from the shell as soon as a shell is destroyed.

Global variable

In many instances, it may be necessary for all the child shells to know about the shell variables of the parent shell. The Bourne shell provides the **export** command to achieve this. The general syntax of **export** command is as follow:

Syntax:

export [variable/expression]

Without argument, **export** command displays all environment variables as well as user-defined variables.

- ✓ A user can make defined variable global as follow:

```
$A=10 <enter>
$export A                    #variable A is exported to child shell
```

- ✓ A user can also create global variable as follow: (works on bash shell)

```
$export B=15<enter>#global variable is created
```

Any shell variable given as an argument to this command will be passed on to all subsequent child shells. Shell variables are accessible to all subsequent child shell known as global variables. These variables are available recursively to child shells. The following example illustrates the above concept:

```
$city=surat
$export city
$echo $city
surat
$sh                    #create a new child shell
$echo $city            #child shell has the variable city
surat
$city=Baroda           #give new value to city
$echo $city
Baroda
$ctrl d                #return to parent shell
$echo $city            #parent shell continues to have value surat
surat
$
```

The above example shows that variables can be exported or passed on to sub-shells, but the reverse is not true. This is because the **export** command causes a copy of the variable name and values to be passed onto a child shell process. The value of the copy can be changed by the child shell but when it dies, so does the copy. The original variable remains untouched.

NOTE: Exported variables are recursively available to all child shell. However, when the child shell alters the value of the variable, the change is not seen in the parent shell.

Exercise

Answer the following questions

1. Explain shell variable with its naming rules.
2. Write a short note on User-defined variable in Unix.
3. What is shell variable? What is its default type?
4. What is null string? Which are the different ways to create null variable?
5. Explain any two methods to evaluate value of shell variable.
6. How can we make variable readonly in Unix?
7. How can we list only system defined variables?
8. Write a short note on system defined variables.
9. Assume that user does not know about existing search path. Is it possible to include path */home/bcal* in search list? Justify your answer.
10. Which are the different ways to know the home directory of a user?
11. List any two environmental variables that are set by reading the file /etc/passwd.
12. What is the difference between login shell and current shell?
13. Explain alias feature of bash shell with its advantages.
14. Explain the purpose of MAILCHECK variable.
15. How can you know the location of your mailbox?
16. Explain the significance of IFS variable.
17. List any two ways to display system defined variables.
18. Differentiate between local variable and global variable.
19. How can you run last command of history that contains 'more' any where in the command?
20. Predict the output of following code:

(i)
```
a=*
echo $a
echo "$a"
echo *
echo "*"
```

(ii)
```
a='*'
echo $a
```

(iii)
```
a="*"
echo $a
```

(iv)
```
a=\*
echo $a
```