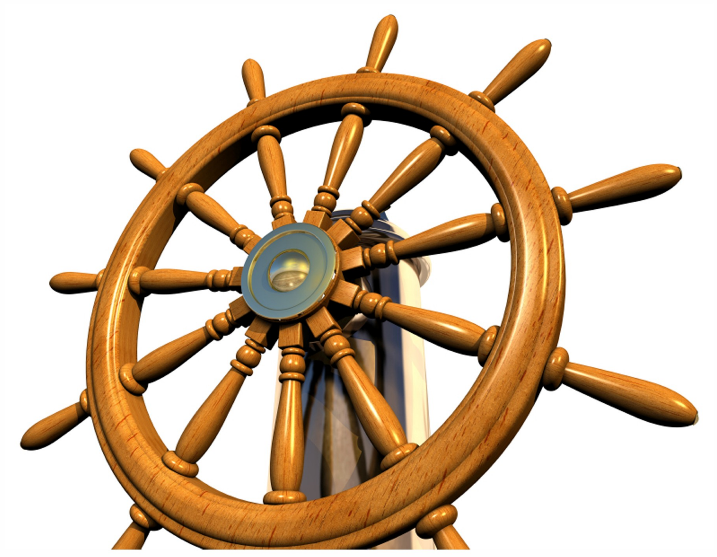
Bash 101 Hacks

Take Control of Your Bash Command Line and Shell Scripting



Ramesh Natarajan

www.thegeekstuff.com

Table of Contents

Introduction

"Take Control of Your Bash Command Line and Shell Scripting"

Bash is the default shell for most Linux distributions. If you are spending significant time in the Linux environment, you should master the bash command line to become productive. Using bash shell scripting, System administrator, DBAs, and developers can quickly automate several routine tasks.

This book is organized as follows:

- Chapters 1 - 3 are an introduction to using the bash command line. It explains fundamentals including command line editing, history, jobs, etc, and different types of bash parameters with several examples.
- Chapters 4 – 6 cover various bash conditional commands, looping structures, builtins, and array manipulation. Clear examples are provided.
- Chapter 7 - 9 explain bash I/O redirection, several bash additional features, and bash shell expansions with clear examples.

A note on the examples: Most examples are identified in the following way.

Example Description

```
Lines of code for you to type, with the result you will
see on screen.
```

Additional clarification or discussion will appear below the code section in plain text.

About the Author

I'm Ramesh Natarajan, author of The Geek Stuff blog thegeekstuff.com and numerous ebooks including this one.

I have done extensive programming in several languages and C is my favorite. I have done a lot of work on the infrastructure side including Linux system administration, DBA, Networking, Hardware and Storage (EMC).

I also developed passworddragon.com — a free, easy and secure password manager that runs on Windows, Linux and Mac.

I wrote the following ebooks:

- Linux 101 Hacks – linux.101hacks.com (free)
- Vim 101 Hacks – vim.101hacks.com
- Nagios Core 3 – thegeekstuff.com/nagios-core-ebook
- Sed and Awk 101 Hacks – thegeekstuff.com/sed-awk-101-hacks-ebook

If you have any feedback about this eBook, please use this contact form - thegeekstuff.com/contact – to get in touch with me.

Copyright & Disclaimer

Version

Version	Date	Revisions
1.0	21 – Sep – 2011	First Edition

Chapter 1. Introduction

1. Bash Introduction

Using a command shell such as Bash, you can give instructions to your operating system. There are two ways to work with a shell:

- Interactively from the command line prompt.
- Automating a task by executing a shell script.

BASH stands for "Bourne Again Shell". This is based on the Bourne shell developed by GNU. This is the default shell on most Linux operating system distributions.

The following are some different types of shells:

- sh - Bourne Shell
- bash - Bourne Again Shell
- csh - C shell
- tcsh - Turbo C shell
- ksh - Korn shell

The /etc/shells file lists all available shells on your system. Please note that on Linux, /bin/sh is a symbolic link to bash.

```
$ cat /etc/shells
/bin/sh
/bin/bash
/sbin/nologin
/bin/tcsh
/bin/csh
```

On Linux, the last field of every user record in /etc/passwd indicates the default shell for that user. You may notice that bash is set as default shell for user accounts.

```
$ grep ramesh /etc/passwd
ramesh:x:511:1004::/home/ramesh:/bin/bash
```

Bash Command Line Mode

Once you login to a typical Linux system, any command that you type at the prompt is executed using the bash shell command line mode. Lines you type contain the command followed by optional arguments. The arguments should be separated by one or multiple spaces.

In the following simple example the “tar” command is executed by the bash shell command line using three arguments that are separated by spaces.

```
$ tar cvfz etc.tar.gz /etc/
```

You can execute from the command line any one of the following:

- Bash builtins
- Bash functions
- Bash aliases
- External non-bash programs located somewhere in your system

Bash Shell Script Mode

A bash shell script is a file that contains a sequence of commands that the bash shell can understand. When you run a bash shell script file, commands in that bash file are executed one by one in sequential order.

Using bash scripting you can automate tasks. For example, you can write a shell script to backup your application files to a specific location.

2. First Shell Script

Let us create a simple shell script that takes a backup of your home directory.

```
$ vi backup1.sh
tar cvfz /backup/ramesh.tar.gz /home/ramesh
```

When you execute this shell script you just created, it will give you a permission denied error message.

```
$ ./backup1.sh
-bash: ./backup1.sh: Permission denied
```

As you notice below, this file doesn't have the execute permission. So, first use the chmod command to assign execute permission to the file, and then execute the shell script from the command line, as shown below.

```
$ ls -l backup1.sh
-rw-r--r--. 1 ramesh ramesh 3 Aug 06 23:22 backup1.sh


$ chmod u+x backup1.sh


$ ./backup1.sh
```

You can execute a shell script using any one of the following three methods:

1. cd to the directory where the shell script is located, and type "./" followed by the file name to execute the script. For example:

```
cd /home/ramesh
./backup1.sh
```

2. Execute the shell script from anywhere by giving the full path of the script.

```
/home/ramesh/backup1.sh
```

3. Add the directory that contains the shell scripts to the PATH environment variable, and just give the name of the shell script to execute it. In this case, you don't need to give either "./" in front of the script or call the script with the full path name.

```
export PATH=$PATH:/home/ramesh
backup1.sh
```

The advantage of methods 2 and 3 is that you don't need to cd to the directory where the script resides in order to execute it.

Let us enhance the above shell script a little bit.

```
$ vi backup2.sh
#!/bin/bash
# Take a backup of a specific user home directory
USERNAME=ramesh
BACKUP_LOCATION=/backup #Location to store the backup
tar cvfz $BACKUP_LOCATION/$USERNAME.tar.gz /home/$USERNAME
echo "Backup of $USERNAME home directory completed."
```

- #!/bin/bash: This should be the first line of your shell script. "#!" is called sha bang, which is followed by the full path of the interpreter that should be used to execute the rest of the commands in this script.
- Each line starting with a "#" is a comment. You can have comments on a full or partial line. If a line starts with #, it is treated as a comment line. At the end of a command, you can also give # and comment the rest of the line, as shown in the line where the BACKUP_LOCATION variable is defined in the above example

- USERNAME is the variable that holds the username that needs to be backed up.
- BACKUP_LOCATION is the variable that holds the location of the backup.

Regarding the line “#!/bin/bash”: in Linux all of the following are the same.

- #!/bin/bash as the first line, which will use bash shell to execute the commands in the shell script. As a best practice, always use this method.
- #!/bin/sh as the first line, which will use /bin/sh (which is a symbolic link to bash in Linux) to execute the commands in the shell script.
- Completely eliminating the “#!” sha bang first line from your shell script. In this case, it will use bash, as it is the default shell in Linux.

Chapter 2. Bash Command Line Fundamentals

3. Command Line Editing

In bash command line mode, when you type a command, you can make corrections to it by using the arrow keys to navigate to a specific location in the line and editing it; when changes are complete you can execute the command just by pressing the <Enter> key. It is not required that the cursor be at the end of the line when you press <Enter> to execute the command.

Using the arrow keys to navigate character by character can be painful. The readline library provides two efficient modes to edit the commands in the command line.

- **emacs mode.** This is the default mode. In this mode, you can use emacs style shortcuts to navigate and edit the commands.
- **vi mode.** In this mode, you can use the vi style shortcuts to navigate and edit the commands. Since I'm a fan of vim editor, I prefer this mode.

The following indicates that the shell is currently using emacs mode.

```
$ set | grep SHELLOPTS
SHELLOPTS=braceexpand:emacs:hashall:histexpand:history:int
eractive-comments:moni
```

To change to the vi mode, do the following:

```
$ set -o vi


$ set | grep SHELLOPTS
SHELLOPTS=braceexpand:hashall:histexpand:history:interacti
ve-comments:monitor:vi
```

To go back to the emacs mode, do the following:

```
$ set -o emacs


$ set | grep SHELLOPTS
SHELLOPTS=braceexpand:emacs:hashall:histexpand:history:int
eractive-comments:monitor
```

In the vi mode, after you use the arrow key to the previous command, use <Esc> key before you use the navigation commands. When you are ready to insert a text or make modifications, use <Esc> key again to go to the edit mode.

The following table contains Emacs and Vim key bindings for common text operations that can be used in the bash command line.

Emacs Mode	**Vim Mode**	**Description**
<Ctrl>+b	h	Move left by one character
<Ctrl>+f	l	Move right by one character
<Ctrl>+d	x	Delete the character highlighted by the cursor
Backspace		Delete the character left to the cursor
Just start typing	<Esc>	Insert characters from the current cursor location
<Ctrl>+a	^	Jump to the beginning of the line
<Ctrl>+e	$	Jump to the end of the line
<Alt>+b	b or B	Move left by one word
<Alt>+f	w or W	Move right by one word
<Ctrl>+k	D	Delete all the characters from current cursor position to end of the line

<Alt>+d	dw	Delete the rest of the current word. If current character is space, delete the next word.
<Ctrl>+w	dB	Delete up to the previous white space. i.e delete the first part of the current word until current cursor position.

~/.inputrc File

You can customize the readline default key bindings by adding your own bindings in a file. Following are some of the readline behaviors that you can change by setting the corresponding variables in the .inputrc file.

Variable	Description
bell-style	Terminal bell ringing style. Possible values are: none, visible, audible (default)
comment-begin	The string that needs to be prefixed to an line for commenting. Default is #
completion-ignore-case	Ignore case during filename completion when set to 'on'. Default is off.
completion-query-items	For completions it displays "Display all xxx possibilities? (y or n)" when the possibilities are above 100 by default. Change that number using this parameter.
disable-completion	Disable word completion (when you press tab) by setting this to on. Default is 'off'.
editing-mode	Possible values are: emcas, vi. Default is emacs.
expand-tilde	Set this to 'on' to convert tilde to full path for word completion. Default is 'off'

history-size	Set this to 0 to store unlimited commands in the history.
horizontal-scroll-mode	When the line you type at the command line is longer, it will scroll horizontally (instead of wrapping it), when this value is set to 'on'. Default is 'off'
mark-directories	When directory name is completed (when you press tab), a slash will be appended by default to the directory. This is enabled by default, set this value to 'off' to disable this.

The user default inputrc file is ~/.inputrc, but if the INPUTRC variable is pointing to a file, bash will try to read the values from that file instead. If ~/.inputrc does not exist, and there is no file pointed to by the INPUTRC variable, then bash will use the global default file /etc/inputrc.

If you want to change the default editing mode to vi anytime you login to bash shell, .inputrc file is a good place to set it.

```
$ vi ~/.inputrc
set editing-mode vi
```

Note: You can also add 'set -o vi' to your .bash_profile file.

The ~/.inputrc file is a good place to set several customized user settings. For example, set the history size as shown below.

```
$ cat ~/.inputrc
set editing-mode vi
set history-size 1500
set completion-ignore-case on
```

4. Command History

Search the history using Control+R

This may be your most frequently used feature of command history. If you’'ve already executed a very long command, you can simply search history using a keyword and re-execute the same command without having to type it fully. Just press Control+R and type the keyword.

In the following example, I searched for the keyword “red”, which displayed the previous command that contained the word “red” (cat /etc/redhat-release”).

```
$ [Press Ctrl+R from the command prompt]
(reverse-i-search)`red': cat /etc/redhat-release

[Press enter when you see your command to execute it]

$ cat /etc/redhat-release
Red Hat Enterprise Linux Server release 6.0 (Santiago)
```

Sometimes you want to edit a command from history before executing it. For example, you can search for the keyword “httpd”, which will display “service httpd stop” command from the history, select this command and change the stop to start and re-execute it again as shown below.

```
$ [Press Ctrl+R from the command prompt]
(reverse-i-search)`httpd': service httpd stop

[Press left arrow key when you see your command, edit it,
and press Enter to execute it]

$ service httpd start
```

Repeat Previous Commands

The following are the 5 different ways to repeat the last executed command.

1. Up arrow and press enter
2. Type !! and press enter
3. Type !-1 and press enter
4. In emacs mode, press Control+P and press enter
5. In vi mode, press <Esc>-k and press enter

Execute a Specific Command From History

In the following example, If you want to repeat the command #4, you can do !4 as shown below.

```
$ history | more
1  service network restart
2  exit
3  id
4  cat /etc/redhat-release

$ !4
cat /etc/redhat-release
Red Hat Enterprise Linux Server release 6.0 (Santiago)
```

Execute a Command that Starts with a Specific Word

Type ! followed by the starting few letters of the command that you would like to re-execute.

In the following example, typing !ps and pressing enter, executed the previous command starting with ps, which is 'ps aux | grep yp '

```
$ !ps
ps aux | grep yp
root 16947 0.1 1264 ?     0:00 ypbind
```

Clear All the Previous History using option -c

Sometime you may want to clear all the previous history, but want to keep the history moving forward.

```
$ history -c
```

Substitute Words from History Commands

Sometimes when searching through history, you may want to execute a different command but use the same parameters from the command that you have just located in history.

In the example below, the !!:$ next to the vi command gets the argument from the previous command to the current command.

```
$ ls anaconda-ks.cfg


$ vi !!:$
vi anaconda-ks.cfg
```

In the example below, the !^ next to the vi command gets the first argument from the previous command (i.e cp command) to the current command (i.e vi command).

```
$ cp anaconda-ks.cfg anaconda-ks.cfg.bak


$ vi  !^
vi anaconda-ks.cfg
```

Substitute a Specific Argument for a Specific Command

In the example below, !cp:2 searches for the previous command in history that starts with cp and takes the second argument of cp and substitutes it for the ls -l command as shown below.

```
$ cp ~/longname.txt /very/long/path/long-filename.txt

$ ls -l !cp:2
ls -l /very/long/path/long-filename.txt
```

In the example below, !cp:$ searches for the previous command in history that starts with cp and takes the last argument (in this case, which is also the second argument as shown above) of cp and substitutes it for the ls -l command as shown below.

```
$ ls -l !cp:$
ls -l /very/long/path/long-filename.txt
```

5. History Related Environment Variables

Display Timestamp using HISTTIMEFORMAT

Typically when you type history from the command line, it just displays the commands. For auditing purposes, it may be beneficial to display the timestamp along with the command as shown below.

```
$ export HISTTIMEFORMAT='%F %T '


$ history | more
1  2011-08-06 19:02:39 service network restart
2  2011-08-06 19:02:39 exit
3  2011-08-06 19:02:39 id
4  2011-08-06 19:02:39 cat /etc/redhat-release
```

Control the History Size using HISTSIZE

You can control the total number of lines in the history using HISTSIZE. Append the following two lines to the .bash_profile and relogin to the bash shell again to see the change.

In this example, only 450 commands will be stored in the bash history.

```
$ vi ~/.bash_profile
HISTSIZE=450
HISTFILESIZE=450
```

Change the History File Name using HISTFILE

By default, history is stored in ~/.bash_history file. Add the following line to the .bash_profile and relogin to the bash shell to store the history command in .mycommands file instead of the .bash_history file.

This can be helpful when you want to track commands executed from different terminals using different history file name.

```
$ vi ~/.bash_profile
HISTFILE=/root/.mycommands
```

Eliminate consecutive repeated history entries using HISTCONTROL

In the following example the “pwd” command was executed three times. You can see all the 3 continuous occurrences of the “pwd” command in history:

```
$ pwd
$ pwd
$ pwd

$ history | tail -4
44  pwd
45  pwd
46  pwd
[Note that there are three pwd commands in history, after
executing pwd 3 times as shown above]
47  history | tail -4
58  history | tail -4
```

To eliminate such duplicates, set HISTCONTROL to ignoredups as shown below. In this example there is only one pwd command in the history, even after executing pwd.

```
$ export HISTCONTROL=ignoredups


$ pwd
$ pwd
$ pwd


$ history | tail -3
56  export HISTCONTROL=ignoredups
57  pwd
```

Erase Duplicates Across the History using HISTCONTROL

The ignoredups control described above removes duplicates only if they are consecutive commands. To eliminate duplicates across the whole history, set the HISTCONTROL flag erasedups.

Example of erasedups usage:

```
$ export HISTCONTROL=erasedups


$ pwd
$ service httpd stop


$ history | tail -3
38  pwd
39  service httpd stop
40  history | tail -3


$ ls -ltr
$ service httpd stop
$ history | tail -6
35  export HISTCONTROL=erasedups
36  pwd
37  history | tail -3
38  ls -ltr
39  service httpd stop
[Note that the previous service httpd stop after pwd got erased]
40  history | tail -6
```

Force History not to Remember a Command using HISTCONTROL

You can instruct history to ignore certain commands by setting HISTCONTROL flag ignorespace. Then type a space in front of any command you wish to be left out of history, as shown below.

```
$ export HISTCONTROL=ignorespace
$ ls -ltr
$ pwd
$  service httpd stop [Note that there is a space at the beginning of service, to ignore this command from history]

$ history | tail -3
67  ls -ltr
68  pwd
69  history | tail -3
```

Note: I can see lot of junior sysadmins getting excited about ignorespace, as they can hide a command from the history. It is good to understand how ignorespace works, but as a best practice, don't purposefully hide anything from history.

Ignore Specific Commands from the History using HISTIGNORE

Sometimes you may not want to clutter your history with basic commands such as pwd and ls. Use HISTIGNORE to specify all the commands that you want to ignore from the history. Multiple commands can be specified in HISTIGNORE by delimiting them with colon.

Note that both “ls” and “ls –ltr” are added to HISTIGNORE in the example. You have to provide the exact command line to be ignored.

```
$ export HISTIGNORE="pwd:ls:ls -ltr:"
$ pwd
$ ls
$ ls -ltr
$ service httpd stop

$ history | tail -3
79  export HISTIGNORE="pwd:ls:ls -ltr:"
80  service httpd stop
81  history
[Note that history did not record pwd, ls and ls -ltr]
```

Disable History using HISTSIZE

If you want to disable history altogether, set HISTSIZE to 0 as shown below and bash will not remember any commands you type.

```
$ export HISTSIZE=0
$ history
$ [Note that history did not display anything]
```

6. Edit Command History with fc

History command

List all the commands from history:

```
history
```

List the last 10 commands from the history:

```
history 10
```

Clear history. This deletes all the commands from the history list:

```
history -c
```

Delete a specific line number from history (the following example deletes history line item #15):

```
history -d 15
```

Write the current history to the default file as shown below. (Normally history from the current session is written to the default history file only when you log off; -w forces the write immediately.) The default file is ~/.bash_history.

```
history -w
```

Add a command to history without really executing it.

```
history -s "tar cvf /tmp/newfile.tar file"
```

fc command

Using fc you can list the history commands. This is just like the history command but with more control. But the power of the fc command is that you can also execute the commands from the history.

List history commands 2 through 5.

```
fc -l 2 5
```

List history commands 2 through 5 (without the line numbers).

```
fc -ln 2 5
```

List history commands 2 through 5 (in reverse order without line numbers).

```
fc -lnr 2 5
```

Without the “-l” which sends the commands to your screen, fc will open them in an editor (uses vi editor as default). You can make appropriate changes to the commands, and when you save and exit the editor, the new commands will be automatically executed in the command line.

Open history commands from 2 through 5 in the default editor vi.

```
fc 2 5
```

Open the history commands in the nano editor.

```
fc -e nano 2 5
```

The fc -s option shown next is very powerful if you use it appropriately, but be careful! You might end-up running some command from the history that you really don't want to repeat.

The following example searches for the pattern "ls" in the command history and executes the previous command that matches this pattern. In this example, it finds "ls -altr" and executes it automatically.

```
$ fc -s ls
ls -altr
```

You can also replace a specific pattern in the history; this is useful for executing a different command with arguments you used before.

Let us assume that you executed the following ls command from the command line.

```
$ ls ~/.inputrc
/home/ramesh/.inputrc
```

The following example searches for the previous command containing the keyword "ls" and changes that pattern to "cat" before executing. So, the following command executes "cat ~/.inputc" because the previous command in the history was "ls ~/.inputrc"

```
$ fc -s ls=cat
cat ~/.inputrc
set editing-mode vi
```

7. History Expansion Examples

Using history expansion you can pick a specific command from the history, execute it as it is, or modify it and execute it based on your needs. The ! starts the history expansion.

Command	Description
!!	Repeats the previous command
!10	Repeat the 10th command from the history
!-2	Repeat the 2nd command (from the last) from the history
!string	Repeat the command that starts with “string” from the history
!?string	Repeat the command that contains the word “string” from the history
^str1^str2^	Substitute str1 in the previous command with str2 and execute it
!!:$	Gets the last argument from the previous

	command.
!string:n	Gets the nth argument from the command that starts with “string” from the history.

!?string Example

Let us assume that you've executed the following command at some point and it is somewhere in the history.

```
$ /usr/local/apache2/bin/apachectl restart
```

Later when you want to execute the same command, if you try the following it will fail because it is looking for a line that starts with “apache”.

```
$ !apache
-bash: !apache: event not found
```

However if you do the following, it will look for any command that **contains** the string “apache” and execute it as shown below.

```
$ !?apache
/usr/local/apache2/bin/apachectl restart
```

^str1^str2^ Example

Sometimes you might check whether a file exists using a quick ls command as shown below.

```
$ ls /etc/sysconfig/network
```

Once you verify that the file exists, to view the content of the file using vi, you don't need to type the whole file name again. Instead do the following, which will replace the word 'ls' in the previous command with the word 'vi' and execute the command.

```
$ ^ls^vi
vi /etc/sysconfig/network
```

!!:$ Example

In this example, the following command takes a copy of the /etc/passwd file to the home directory as passwd.bak.

```
$ cp /etc/passwd /home/ramesh/passwd.bak
```

Once you create the backup of the file, if you want to open the backup file, you don't need to type the whole backup file name again. Instead, you can use the last argument of the previous command along with 'vi' command as shown below.

```
$ vi !!:$
vi /home/ramesh/passwd.bak
```

Please note that “!:$” is exactly same as “!!:$”. So, the above example can also be executed as shown below.

```
$ vi !:$
vi /home/ramesh/passwd.bak
```

!string:n Example

When you execute a command that has multiple arguments (as shown in the tar command example below), you can extract only a specific argument from it for later use.

```
$ tar cvfz ~/sysconfig.tar.gz /etc/sysconfig/*
```

Now if you want to ls the newly created tar.gz file, you can do the following, which will take the 2nd argument of the previous tar command.

```
$ ls -l !tar:2
ls -l ~/sysconfig.tar
```

8. Manage Background Jobs using &, fg, bg

You can execute a shell script or a program in the background using the bash job control commands.

Executing a background job

Appending an ampersand (&) to the command executes the job in the background.

For example, when you execute a find command that might take a long time to execute, you can put it in the background as shown below.

The following example finds all the files under the root file system that changed within the last 24 hours.

```
$ find / -ctime -1 > /tmp/changed-file-list.txt &
[1] 5378
```

Anytime you put a job in the background, it will display an output like "[1] 5378".

- [1] - Indicates the job id. In this example, the job id is 1
- 5378 - The number next to the job id indicates the PID of the process. In this example, the process id of the background job is 5378.

Send a Foreground job to the Background using CTRL-Z and bg

You can send the current foreground job to the background as explained below:

- Press CTRL+Z, which will suspend the current foreground job.
- Execute bg, to put that job in the background.

For example, if by mistake you started a job in the foreground, you don’t need to kill the current job and start a new background job.

Instead, suspend the current job and put it in the background as shown below.

```
$ find / -ctime -1 > /tmp/changed-file-list.txt
$ [Press CTRL-Z]
[2]+ Stopped find / -ctime -1 > /tmp/changed-file-list.txt


$ bg
```

After pressing CTRL-Z, instead of typing bg, you can also simply use the job number followed by & to put the job in the background as shown below. In the above example, when you pressed Ctrl-Z, it stopped the foreground process, and displayed the job number [2]. So, use that job number to put the process in the background.

```
$ %2 &
```

View Background Jobs using jobs Command

Type 'jobs' to view all the background jobs.

```
$ jobs
[1]   Running                         bash download-file.sh &
[2]-  Running                         evolution &
[3]+  Done                            nautilus .
```

- The + after a job number indicates that it is the current job.
- The - after a job number indicates that it is the previous job.
- In the above example, job number 3 is the current job, and job number 2 is the previous job.

Move a Job from the Background to the Foreground using fg

You can bring a background job to the foreground using the fg command. When executed without arguments, fg will take the most recent background job and move it to the foreground.

```
$ fg
```

When there are multiple background jobs running, and you would like to move a particular job to the foreground, you should use the job id.

In the following example, fg %1 will bring the job#1 (i.e download-file.sh) to the foreground.

```
$ jobs
[1]   Running                 bash download-file.sh &
[2]-  Running                 evolution &
[3]+  Done                    nautilus .

$ fg %1
```

You can also simply use the job number to bring the job to the foreground (i.e without giving fg command), as shown below.

```
$ %1
```

%%, %+, %-

Let us assume the following jobs are running in the background:

```
$ jobs
[1]   Running                 ./jobs.sh &
[2]-  Running                 ./jobs1.sh &
[3]+  Running                 ./jobs2.sh &
```

%% , %+ , and % are all the same, indicating the current background job. In this example, it is job number 3.

```
$ jobs %%
[3]+  Running                 ./jobs2.sh &
$ jobs %+
[3]+  Running                 ./jobs2.sh &
$ jobs %
[3]+  Running                 ./jobs2.sh &
```

%- indicates the previous background job. In this example, it is job number 2.

```
$ jobs %-
[2]-  Running                 ./jobs1.sh &
```

Kill a Specific Background Job using Kill %

If you want to kill a specific background job, use “kill %job-number”. For example, to kill the job 2 use:

```
$ kill %2
```

9. Job Command Options

Use "jobs -l" to display PID along with the job id.

```
$ jobs -l
[1]   6580 Done                 ./jobs.sh
[2]-  6625 Running              ./jobs1.sh &
[3]+  6688 Running              ./jobs2.sh &
```

Display only the status of a specific job number.

```
$ jobs -l 3
```

```
[3]+  6688 Running                 ./jobs2.sh &
```

When you are running multiple jobs, if you want to know only the status of the jobs that were changed since it was informed to the user last time, do the following.

```
$ jobs -n
[1]   6580 Done                    ./jobs.sh
```

Display only the PID of all the background jobs.

```
$ jobs -p
6625
6688
```

Bring the job that contains the letter j to the foreground

```
$ %?j
./jobs.sh
```

Bring the job that starts with string "./j" to the foreground

```
$ %./j
./jobs.sh
```

$! contains the PID of the last background job.

```
$ echo $!
6922
```

The following table indicates the various job strings that you can use to refer the background jobs.

Job String	Description
%n	Indicates the nth background job
%%, $+, %	Indicates the current job
%-	Indicates the previous job
%string	Indicates a job that starts with the specified string
%?string	Indicates a job that contains the specified string

Disown Command

Using the disown command, you can instruct bash to disown a specific job.

When a job is disowned by the bash shell, it is be removed from the background job table that bash maintains, so that it will not be displayed by the jobs command, but the job is still running. You can view it using the ps command.

In the following example, a background job is running.

```
$ jobs
[1]+  Running                 ./jobs.sh &
```

Now, instruct bash shell to disown this background job.

```
$ disown %1
```

After disowning, the job is not displayed in the jobs command output.

```
$ jobs
$ jobs -l
```

However, when you do “ps -ef” to show all processes, you'll see the job running in the background.

```
$ ps -ef | grep 6922
ramesh    6922  6513  0 08:11 00:00:00 bash
ramesh    6998  6922  0 08:16 00:00:00 ./jobs.sh
ramesh    7000  6513  0 08:16 00:00:00 grep -i 6922
```

Wait and Suspend

When a job is running, you can execute the wait command to instruct bash to wait until all the jobs are terminated.

In this example, the following 3 jobs are running.

```
$ jobs -l
[1]   6580 Done                    ./jobs.sh
[2]-  6625 Running                 ./jobs1.sh &
[3]+  6688 Running                 ./jobs2.sh &
```

The wait command without any argument will wait for all the jobs owned by this session to be completed.

```
wait
```

You can also pass either the PID or a job number to instruct the bash shell to wait for a particular job. The following example waits for job#2 to complete.

```
wait %2
(or)
wait 6625
```

Note: You can also use the wait command in a shell script, which will defer the shell script from continuing until all the background jobs are completed.

The suspend command suspends the execution of the current bash shell and waits for the SIGCONT signal. You cannot suspend a login shell:

```
$ suspend
-bash: suspend: cannot suspend a login shell
```

So, let us create a new bash shell, and suspend that:

```
$ bash
$ suspend
[1]+  Stopped                 bash
```

The bash shell we just created using the "bash" command is now suspended (i.e Stopped). To bring it to the foreground, do the following.

```
$ fg
bash
```

10. Bash Aliases

An alias command is a simple string substitution of one text for another that can be used as the first word of a simple command.

How to Set an Alias

Aliases can be defined on the command line, or in the .bash_profile, or in the .bashrc, using the following syntax:

```
alias name='unix command with options'
```

- alias – is a shell built-in
- name – any user-defined name for the alias.
- ‘unix command with options’ stands for any unix command, which may include options.

This means that name will be an alias for the command. Whenever name is typed, bash will substitute the corresponding command with its options.

No spaces are needed on either side of the equal sign. Quotes around the command are necessary if the substituted string has of more than one word.

Executing the alias command from the command line creates a temporary alias, that is, it is available until you exit the shell. Storing the alias in a bash startup files makes it a permanent alias.

Common Alias Examples

Some useful bash alias examples are shown below that you can use in your bash startup files.

Open last modified file in vim:

```
alias Vim="vim `ls -t | head -1`"
```

Find top 5 big files:

```
alias findbig="find . -type f -exec ls -s {} \; | sort -n
-r | head -5"
```

Grep for a bash process:

```
alias psg="ps -aux ¦ grep bash"
```

List including hidden files with indicator and color:

```
alias ls='ls -aF --color=always'
```

List in long format:

```
alias ll='ls -l'
```

Clear all the history and screen:

```
alias hcl='history -c; clear'
```

Make basic commands interactive, and verbose:

```
alias cp="cp -iv"
alias rm="rm -i"
alias mv="mv -iv"
alias grep="grep -i"
```

Exit this command shell:

```
alias x="exit"
```

Clear the screen and list files:

```
alias cls='clear;ls'
```

Filesystem diskspace usage:

```
alias dus='df -h'
```

Navigate to the different directories:

```
alias ..='cd ..'

alias 2..='cd ../..'
```

How to view all the Aliases

Execute alias without arguments to view list of aliases set in a shell.

```
$ alias
alias ..='cd ..'
alias ll='ls -l'
..
```

To view a particular alias, enter the command format “alias aliasname” as shown below.

```
$ alias ll
alias ll='ls -l'
```

Using alias to redefine a system command

You can use alias to redefine a command; this is useful for such things as requiring confirmation before executing a destructive copy operation:

```
$ alias cp="cp -iv"
```

Note: Shell functions are faster. Aliases are looked up after functions and thus resolving is slower. While aliases are easier to understand, shell functions are preferred over aliases for almost every purpose. You should be very careful replacing a standard command with an alias or a function.

How to temporarily stop using aliases

If you want to call the native command instead of your alias, then you have to type \ in front of it to ‘escape’ the conversion.

```
$ \aliasname
```

For example, alias cp=”cp -iv”, will ask you confirmation if you are about to overwrite a file. This can be annoying when you are copying lot of files that you already know you are going to overwrite. You might want to temporarily use the regular cp command instead of the cp-alias. You would type:

```
\cp * /backup/files/
```

How to remove an Alias

unalias is a shell built-in to remove an alias. To remove a particular alias specify its name as the argument to the unalias command.

Remove the alias for hcl, if you previously defined it.

```
$ hcl
[history and screen are cleared]
$ unalias hcl
$ hcl
-bash: hcl: command not found
```

How to remove all Aliases

unalias with -a option, removes all the aliases.

```
$ unalias -a
$ alias
```

11. Customize Bash Prompt using PS1

Display Username, Hostname and Current Directory in the Prompt

The PS1 in this example displays the following information in the prompt:

- \u – Username
- \h – Hostname
- \w – Full path of the current working directory

```
$ export PS1="\u@\h \w$ "

ramesh@dev-db /etc/mail$
```

Different Codes for the PS1 Variable to define a Custom Prompt

Use the following codes and create your own personal PS1 Linux prompt that is functional and suites your taste.

Custom Prompt Codes	
Code	**Description**
\a	ASCII bell character (07)
\d	Date in "Weekday Month Date" format (e.g., "Tue May 26")
\D{format}	Format is passed to strftime(3) and the result is inserted into the prompt string; an empty format results in a locale-specific time representation. The braces are required
\e	ASCII escape character (033)
\h	Hostname up to the first part
\H	Full Hostname
\j	Total number of jobs currently managed by the shell
\l	Basename of the shell's terminal device name
\n	Newline
\r	Carriage return
\s	Name of the shell, the basename of $0 (the portion following the final slash)
\t	Current time in 24-hour HH:MM:SS format
\T	Current time in 12-hour HH:MM:SS format
\@	Current time in 12-hour am/pm format
\A	Current time in 24-hour HH:MM format

Custom Prompt Codes	
Code	**Description**
\u	Username of the current user
\v	Bash version (e.g., 4.1)
\V	Full bash version. I.e version + patch level (e.g., 4.1.2)
\w	Current working directory, with $HOME abbreviated with a tilde
\W	Basename of the current working directory, with $HOME abbreviated with a tilde
\!	History number of this command
\#	Command number of this command
\$	If the effective UID is 0, a #, otherwise a $
\nnn	Character corresponding to the octal number nnn
\\	Backslash
\[	Begin a sequence of non-printing characters, which could be used to embed a terminal control sequence into the prompt
\]	End a sequence of non-printing character

Display Current Time in the Prompt

In the PS1 environment variable, you can directly execute any Linux command, by specifying in the format $(linux_command).

In the following example, the command $(date) is executed to display the current time inside the prompt.

```
$ export PS1="\u@\h [\$(date +%k:%M:%S)]$ "
```

```
ramesh@dev-db [11:09:56]$
```

You can also use \t to display the current time in “hh:mm:ss” format:

```
$ export PS1="\u@\h [\t]$ "

ramesh@dev-db [12:42:55]$
```

You can also use \@ to display the current time in 12-hour am/pm format:

```
$ export PS1="[\@] \u@\h$ "

[04:12 PM] ramesh@dev-db$
```

Show host, kernel type, and last command status in the Prompt

The following example displays three items separated by | (pipe) in the command prompt:

- \!: The history number of the command
- \h: hostname
- $kernel_version: The output of the uname -r command from $kernel_version variable
- \$?: Status of the last command

```
$ kernel_version=$(uname -r)

$ export PS1="\!|\h|$kernel_version|\$?$ "

473|dev-db|2.6.25-14.fc9.i686|0$
```

Use Bash Shell Function inside PS1 Variable

You can also invoke a bash shell function in the PS1 as shown below. This displays the total number of running httpd processes.

```
$ function httpdcount {
>  ps aux | grep httpd | grep -v grep | wc -l
> }

$ export PS1="\u@\h [`httpdcount`]$ "

ramesh@dev-db [12]$
```

You can add the following lines to .bash_profile or .bashrc to make this change permanent:

```
function httpdcount {
  ps aux | grep httpd | grep -v grep | wc -l
}
export PS1='\u@\h [`httpdcount`]$ '
```

Use a Shell Script Inside PS1 Variable

You can also invoke a shell script inside the PS1 variable. In the example below, the ~/bin/totalfilesize.sh, which calculates the total filesize of the current directory, is invoked inside the PS1 variable.

```
$ cat ~/bin/totalfilesize.sh
for filesize in $(ls -l . | grep "^-" | awk '{print $5}')
do
  let totalsize=$totalsize+$filesize
done
echo -n "$totalsize"
```

Set the PS1 variable to call the above totalfilesize.sh file.

```
$ export PATH=$PATH:~/bin

$ export PS1="\u@\h [\$(totalfilesize.sh) bytes]$ "

ramesh@dev-db [167997 bytes]$
```

Note: This executes the totalfilesize.sh to display the total file size of the current directory in the PS1 prompt

12. Color Your Bash Prompt

Change the Foreground Color of the Prompt

Display prompt in blue color, along with username, host and current directory information.

This is for a light blue prompt:

```
$ export PS1="\e[0;34m\u@\h \w$ \e[m"
```

This is for a dark blue prompt:

```
$ export PS1="\e[1;34m\u@\h \w$ \e[m"
```

In the above two PS1 examples:

- \e[- Indicates the beginning of color prompt
- x;ym - Indicates color code as described below.
- \e[m - indicates the end of color prompt

The following are few standard color codes:

- Black 0;30
- Blue 0;34
- Green 0;32

- Cyan 0;36
- Red 0;31
- Purple 0;35
- Brown 0;33

Note: Replace 0 with 1 in the above color codes for dark color. For example, “1,33” is for dark brown.

Another method of setting the color is shown below. Make the color change permanent by adding the following lines to .bash_profile or .bashrc file.

```
STARTCOLOR='\033[31m';
ENDCOLOR="\033[0m"
export PS1="$STARTCOLOR\u@\h \w$ $ENDCOLOR"
```

The following are few values you can use for the above STARTCOLOR:

- Black \033[30m
- Blue \033[34m
- Green \033[32m
- Cyan \033[36m
- Red \033[31m
- Purple \033[35m
- Brown \033[33m

Change the Background Color of the Prompt

Change the background color by specifying **\e[{code}m** in the PS1 prompt as shown below.

This is for Light Gray background:

```
$ export PS1="\e[47m\u@\h \w$ \e[m"
```

Combine of background and foreground. This is for Light Blue foreground and Light Gray background:

```
export PS1="\e[0;34m\e[47m\u@\h \w$ \e[m"
```

Add the following to the .bash_profile or .bashrc to make the above background and foreground color permanent.

```
STARTFGCOLOR='\e[0;34m';
STARTBGCOLOR="\e[47m"
ENDCOLOR="\e[0m"
export PS1="$STARTFGCOLOR$STARTBGCOLOR\u@\h \w$ $ENDCOLOR"
```

Play around by using the following background colors (for the above STARTBGCOLOR) and choose the one that suits your taste:

- \e[40m
- \e[41m
- \e[42m
- \e[43m
- \e[44m
- \e[45m
- \e[46m
- \e[47m

Display Multiple Colors in the Prompt

You can also display multiple colors in the same prompt. Add the following function to .bash_profile

```
function prompt {
  local BLUE="\[\033[0;34m\]"
  local DARK_BLUE="\[\033[1;34m\]"
  local RED="\[\033[0;31m\]"
  local DARK_RED="\[\033[1;31m\]"
  local NO_COLOR="\[\033[0m\]"
  case $TERM in
    xterm*|rxvt*)
      TITLEBAR='\[\033]0;\u@\h:\w\007\]'
      ;;
    *)
      TITLEBAR=""
      ;;
  esac
  PS1="\u@\h [\t]$ "
  PS1="${TITLEBAR}\
  $BLUE\u@\h $RED[\t]>$NO_COLOR "
  PS2='continue-> '
  PS4='$0.$LINENO+ '
}
```

You can re-login for the changes to take effect or source the .bash_profile as shown below.

```
$. ./.bash_profile
$ prompt
ramesh@dev-db [13:02:13]>
```

Change the Prompt Color using Tput

You can also change color of the PS1 prompt using tput as shown below:

```
$ export PS1="\[$(tput bold)$(tput setb 4)$(tput setaf
7)\]\u@\h:\w $ \[$(tput sgr0)\]"
```

tput Color Capabilities:

- tput setab [1-7] - Set a background color using ANSI escape
- tput setb [1-7] - Set a background color
- tput setaf [1-7] - Set a foreground color using ANSI escape
- tput setf [1-7] - Set a foreground color

tput Text Mode Capabilities:

- tput bold - Set bold mode
- tput dim - turn on half-bright mode
- tput smul - begin underline mode
- tput rmul - exit underline mode
- tput rev - Turn on reverse mode
- tput smso - Enter standout mode (bold on rxvt)
- tput rmso - Exit standout mode
- tput sgr0 - Turn off all attributes

Color Code for tput:

- 0 – Black
- 1 – Red
- 2 - Green
- 3 - Yellow
- 4 - Blue
- 5 - Magenta
- 6 – Cyan
- 7 - White

13. PS1, PS2, PS3, PS4 and PROMPT_COMMAND

PS1 – Default Interactive Prompt

As explained in the previous section, the default interactive prompt on your Linux can be modified to something useful and informative using PS1.

In the following example, the default PS1 was “\s-\v\$”, which displays the shell name and the version number. Let us change this default behavior to display the username, hostname and current working directory name:

```
bash-4.1$ export PS1="\u@\h \w$ "


ramesh@dev-db /etc/mail$
```

PS2 – Continuation interactive prompt

A very long unix command can be broken down to multiple lines by giving \ at the end of the line. I find it very helpful and easy to read to break long commands into multiple lines using \. I have seen others who don’t like to break up long commands.

The default interactive prompt for a multi-line command is “> “. Let us change this default behavior to display “continue->” by using the PS2 environment variable as shown below.

This uses the default ">" for continuation prompt:

```
$ myisamchk --silent --force --fast --update-state \
> --key_buffer_size=512M --sort_buffer_size=512M \
> --read_buffer_size=4M --write_buffer_size=4M \
> /var/lib/mysql/bugs/*.MYI
```

This uses the modified "continue-> " for continuation prompt:

```
$ export PS2="continue-> "


$ myisamchk --silent --force --fast --update-state \
continue-> --key_buffer_size=512M
--sort_buffer_size=512M \
continue-> --read_buffer_size=4M --write_buffer_size=4M \
continue-> /var/lib/mysql/bugs/*.MYI
```

PS3 – Prompt used by “select” Inside Shell Script

You can define a custom prompt for a select loop inside a shell script, using the PS3 environment variable, as explained below.

Shell script and output WITHOUT PS3: (This displays the default "#?" for select command prompt)

```
$ cat ps3.sh
select i in mon tue wed exit
do
  case $i in
    mon) echo "Monday";;
    tue) echo "Tuesday";;
    wed) echo "Wednesday";;
    exit) exit;;
  esac
done

$ ./ps3.sh
1) mon
2) tue
3) wed
4) exit
```

```
#? 1
Monday
#? 4
```

Shell script and output WITH PS3:

```
$ cat ps3-1.sh
PS3="Select a day (1-4): "
select i in mon tue wed exit
do
  case $i in
    mon) echo "Monday";;
    tue) echo "Tuesday";;
    wed) echo "Wednesday";;
    exit) exit;;
  esac
done

$ ./ps3-1.sh
1) mon
2) tue
3) wed
4) exit
Select a day (1-4): 1
Monday
Select a day (1-4): 4
```

Note: This displays the modified "Select a day (1-4): " for the select command prompt

PS4 – Prompt used by “set -x” to prefix tracing output

The PS4 shell variable defines the prompt that gets displayed when you execute a shell script in debug mode as shown below.

Shell script and output WITHOUT custom PS4 displays the default "++":

```
$ cat ps4.sh
set -x
echo "PS4 demo script"
ls -l /etc/ | wc -l
du -sh ~

$ ./ps4.sh
++ echo 'PS4 demo script'
PS4 demo script
++ ls -l /etc/
++ wc -l
243
++ du -sh /home/ramesh
48K     /home/ramesh
```

The example shell script and output WITH custom PS4 displays the modified "{script-name}.{line-number}+" while tracing the output. The PS4 defined below in the ps4.sh has the following two codes:

- $0 – indicates the name of script
- $LINENO – displays the current line number within the script

```
$ cat ps4.sh
export PS4='$0.$LINENO+ '
set -x
echo "PS4 demo script"
ls -l /etc/ | wc -l
du -sh ~

$ ./ps4.sh
../ps4.sh.3+ echo 'PS4 demo script'
```

```
PS4 demo script
../ps4.sh.4+ ls -l /etc/
../ps4.sh.4+ wc -l
243
../ps4.sh.5+ du -sh /home/ramesh
48K     /home/ramesh
```

PROMPT_COMMAND

Bash shell executes the content of PROMPT_COMMAND before displaying the PS1. This example displays the PROMPT_COMMAND and PS1 output on different lines.

```
$ export PS1="\u@\h \w$ "
$ export PROMPT_COMMAND="date +%k:%m:%S"

22:08:42
ramesh@dev-db /etc/mail$
```

If you want to display the value of PROMPT_COMMAND in the same line as the PS1, use the echo -n as shown below:

```
$ export PROMPT_COMMAND="echo -n [$(date +%k:%m:%S)]"

[22:08:51]ramesh@dev-db /etc/mail$
```

14. Bash Startup Files

For Interactive Bash Shell

The following is the sequence of execution of bash startup files, when you login through Bash interactive login shell:

- If /etc/profile exists, execute it and go to the next step
- If ~/.bash_profile exists, execute it and go to the end

- If ~/.bash_login exists, execute it and go to the end
- If ~/.profile exists, execute it and go to the end
- End

In the above steps ~/.bashrc is not directly checked by bash during the login shell. However, in the ~/.bash_profile file, you'll typically have the following code at the top, which will invoke the ~/.bashrc during the login shell. If you don't have the following lines in your ~/.bash_profile, add them.

```
if [ -f ~/.bashrc ]; then
  . ~/.bashrc
fi
```

When the /etc/profile is executed, it checks all the *.sh file under /etc/profile.d and executes them. If you are a system administrator, and would like to add some global environment variable setting for all the users, you can create a shell script called env.sh under /etc/profile.d directory. When you logout, if ~/.bash_logout exits, it executes it at that time.

For Interactive Non-login Bash Shell

For Bash interactive non-login shell, if ~/.bashrc exists, it is executed.

Typically you'll have the following lines in the ~/.bashrc which will execute the /etc/bashrc. Add them if not.

```
if [ -f /etc/bashrc ]; then
        . /etc/bashrc
fi
```

Test the Sequence of Execution

One of the ways to test the sequence of execution is by adding different PS1 values to these files, then logging in again to the shell to see which PS1 value got picked up by the Linux prompt. Earlier we discussed how to use PS1 to make your Linux prompt both functional and stylish.

/etc/profile gets executed. Add following PS1 line to /etc/profile and re-login to make sure the Linux prompt changes to the PS1 value set inside the /etc/profile.

```
$ grep PS1 /etc/profile
PS1="/etc/profile> "
```

Re-login to see the prompt change as shown below. Please make sure ~/.bash_profile doesn't have any PS1 for this to work properly.

```
/etc/profile>
```

~/.bash_profile gets executed: Add following PS1 to ~/.bash_profile, ~/.bash_login, ~/.profile and ~/.bashrc. Re-login to make sure the Linux prompt changes to the PS1 value set inside the ~/.bash_profile as shown below.

```
/etc/profile> grep PS1 ~/.bash_profile
export PS1="~/.bash_profile> "
/etc/profile> grep PS1 ~/.bash_login
export PS1="~/.bash_login> "
/etc/profile> grep PS1 ~/.profile
export PS1="~/.profile> "
/etc/profile> grep PS1 ~/.bashrc
export PS1="~/.bashrc> "
```

Upon re-login, bash executed /etc/profile first and ~/.bash_profile next. So, it took the PS1 from ~/.bash_profile as shown below. It also did not execute ~/.bash_login, as ~/.bash_profile exists.

```
~/.bash_profile>
```

~/.bash_login gets executed. Rename the .bash_profile to something else. Re-login to make sure the Linux prompt changes to the PS1 value set inside the ~/.bash_login as shown below.

```
mv .bash_profile bash_profile_not_used
```

Upon re-login, it executed /etc/profile first. Since it cannot find ~/.bash_profile, it executed ~/.bash_login.

```
~/bash_login>
```

~/.profile gets executed. Rename the .bash_login to something else. Re-login to make sure the Linux prompt changes to the PS1 value set inside the ~/.profile as shown below.

```
~/.bash_login> mv .bash_login bash_login_not_used
```

Upon re-login, it executed /etc/profile first. Since it cannot find ~/.bash_profile and ~/.bash_login, it executed ~/.profile.

```
~/.profile>
```

~/.bashrc gets executed for non-login shell testing. Executing "bash" at the command prompt will give another non-login shell, which will invoke .bashrc as shown below.

```
~/.profile> bash
```

This displays PS1 from .bashrc as shown below.

```
~/.bashrc> exit
```

After exiting from non-login shell, we are back to login shell.

```
~/.profile>
```

Chapter 3. Bash Parameters

15. Three Types of Bash Parameters

Bash has the following three types of parameters.

1. Variables

These are created by the user either in the command-line or from a shell script to store values (string, integer, etc.,), required for any custom processing.

2. Special parameters

These are pre-defined by bash and cannot be changed by the user, i.e. they are read-only.

3. Positional parameter

These are the arguments given to your shell scripts when it is invoked. It could be from **$1** to **$N**. When N consists of more than a single digit, it must be enclosed in a braces like **${N}**.

We'll be discussing more about these in the later hacks.

16. Variable Basics

Following are the key points to remember about bash variables.

No Space Before or After the “=” Sign.

- Correct: When you use VAR=VALUE, shell sees the “=” as a symbol and treats the command as a variable assignment.
- Wrong: When you use VAR = VALUE, shell assumes that VAR must be the name of a command and tries to execute the command VAR.

- Correct: The following example assigns the value "ramesh" to the variable USERNAME.

```
$ USERNAME=ramesh
```

Wrong: The following example tries to execute the command "USERNAME" with two arguments "=" and "ramesh". Since there is not a command called "USERNAME", bash displays "command not found".

```
$ USERNAME = ramesh
-bash: USERNAME: command not found
```

Wrong: The following example tries to execute the command "USERNAME" with the argument "=ramesh". Since there is not a command called "USERNAME", it displays "command not found".

```
$ USERNAME =ramesh
-bash: USERNAME: command not found
```

Wrong: The following example is little different from the above two wrong examples, since bash will recognize that USERNAME is a variable not a command. But this is still wrong because it assigns an empty value to the variable USERNAME and then tries to execute the command "ramesh". Since there is not a command called "ramesh", it displays "command not found".

```
$ USERNAME= ramesh
-bash: ramesh: command not found
```

Refer to a Variable Using $ or ${ }

```
$ USERNAME=ramesh
$ echo "Value of USERNAME variable is: $USERNAME"
Value of USERNAME variable is: ramesh
```

In most cases, **$** and **${}** are the same. For example:

```
$ echo "Value of USERNAME variable is: ${USERNAME}"
Value of USERNAME variable is: ramesh
```

However, when you concatenate the value of a variable with another value, you should use the **${}** referencing form. For example, suppose you want to append the static value "home" to the value of the **$USERNAME** variable.

The following will not work. This is really trying to display the value of the variable **$USERNAMEhome**. Since there is no such variable, nothing will be displayed

```
$ echo $USERNAMEhome
```

However, the following will work. Now, we've clearly separated the variable name from the rest of the string using **${}**.

```
$ echo ${USERNAME}home
rameshhome
```

*Note: Using the **${}** form is called parameter substitution. There is lot more to parameter substitution, which we'll cover in later hacks.*

Quoting Vs Not-Quoting ("$VAR" vs $VAR)

In simple terms:

- If you quote, it is considered as one word.
- If you don't quote, it will be split into multiple words.

In the following example, $states in the for loop is not quoted. As you see below, the values will not be quoted, when it is used.

In this example, (for i in $states) is same as (for i in CA NY UT TX).

```
$ cat var.sh
states="CA NY UT TX"
```

```
for i in $states
do
 echo $i
done
```

When you execute this example, you'll see the value of $states is split into multiple words and displayed individually as shown below.

```
$ ./var.sh
CA
NY
UT
TX
```

In the following example, $states in the for loop is quoted. As you see below, the values will be quoted, when it is used.

In this example, (for i in “$states”) is same as (for i in "CA NY UT TX").

```
$ cat var1.sh
states="CA NY UT TX"
for i in "$states"
do
 echo $i
done
```

When you execute this example, you'll see the value of $states is NOT split into multiple words. It is treated as a single word and displayed in one line.

```
$ ./var1.sh
CA NY UT TX
```

Bash shell does not care about the type of variables. i.e variables can store strings, integers, or real numbers.

Variables do not have to be declared. But, when you access the variable which is not used so far, you will not get any warning or error message. Instead, it will display a blank value as shown below.

```
$ echo "VAR1 Value: $VAR1"
VAR1 Value:
```

When the value contains space in them, use double quotes around the value. More on quoting later.

```
$ VAR1="The Geek Stuff"


$ echo "VAR1 Value: $VAR1"
VAR1 Value: The Geek Stuff
```

Getting Variable from the Environment to the Bash Shell Script

First, define a variable in the command line.

```
$ VAR1="The Geek Stuff"
```

Next, create a shell script that uses the variable we just defined:

```
$ vi variable.sh
#!/bin/bash
echo "VAR1 Value: $VAR1"
```

Now, execute this shell script.

```
$ ./variable.sh
VAR1 Value:
```

Executing this shell script displays a blank value, even though we assigned a value to the VAR1 from the command line. Why?

The answer is: During the execution of variable.sh, it spawns a new bash shell and it executes the script. So the variable VAR1 will not have the value in the spawned shell. You need to export the variable for it to be inherited by another program – including a shell script, as shown below.

```
$ export VAR1="The Geek Stuff"
$ ./variable.sh
VAR1 Value: The Geek Stuff
```

Variables are Case Sensitive

USERNAME is not same as username. They are different variables.

```
$ USERNAME=ramesh
$ username=john

$ echo "USERNAME=$USERNAME username=$username"
USERNAME=ramesh username=john
```

17. Positional Parameters - Shell Script Arguments

Arguments that are passed to the shell scripts can be referenced as positional parameters.

- **$1** - 1st argument
- **$2** - 2nd argument
- ..
- **${10}** - 10th argument
- **${11}** - 11th argument

When you have two digits, you should use **${ }** format to reference them. **$0** is the name of the script.

```
$ cat args.sh
echo "Script Name: $0"
echo "First Parameter: $1"
echo "Second Parameter: $2"
echo "Third Parameter: $3"
echo "All Parameters: $*"
echo "All Parameters: $@"
echo "Total Number of Parameters: $#"

$ ./args.sh CA NY UT
Script Name: ./args.sh
First Parameter: CA
Second Parameter: NY
Third Parameter: UT
All Parameters: CA NY UT
All Parameters: CA NY UT
Total Number of Parameters: 3
```

Note: $0, $, $@ are not positional parameters. They are special parameters. We'll discuss more about special parameters in the later hack.*

Also, please note that $0's value is set depending on how the script is called:

- For "./args.sh CA NY UT", $0 is "./args.sh".
- For "/home/ramesh/args.sh CA NY UT", $0 is "/home/ramesh/args.sh".

For debugging purposes, if you want to display all the parameters, use the following single line code-snippet.

```
$ vi args1.sh
printf '"%b"\n' "$@" | cat -n

$ ./args1.sh CA NY UT
     1  "CA"
     2  "NY"
     3  "UT"
```

Difference between $* and $@

They behave exactly the same except when you double quote them. When you double quote "$*", it is converted to "${1}x${2}x${3}x...". In this, x is the first character of the IFS variable.

So, you should always use $@, unless you have a specific reason. The following example shows this difference.

```
$ vi args2.sh
IFS="<^"
echo -n '[$*] = '
echo $*
echo -n '[$@] = '
echo $@
echo -n '["$*"] = '
echo "$*"
echo -n '["$@"] = '
echo "$@"

$ ./args2.sh one two three
[$*] = one two three
[$@] = one two three
["$*"] = one<two<three
["$@"] = one two three
```

18. Getting Range of Parameters

As you already know, either $@ or $* gives all the arguments that was passed to the shell script. However, if you want get only a range of arguments (without explicitly calling them using $1, $2, $3 etc.,), you can use the method shown here.

```
${@:$start:$count}
```

- $start - is the starting position of the argument from which to extract. When $start is negative number, the starting position is counted from the end, i.e a start value of 3 means start with the 3rd argument from the beginning, and a start value of -3 means start with the 3rd argument from the end.
- $count - indicates how many arguments to extract from the $start position. This is optional. When you don't provide $count, bash extracts all the remaining arguments from $start.

Example of argument extraction:

```
$ cat args3.sh
echo "\$@        = $@"
start=2
count=3
echo "\${@:2}    = ${@:$start}"
echo "\${@:2:3}  = ${@:$start:$count}"
start=-4
echo "\${@:-4}   = ${@:$start}"
echo "\${@:-4:3} = ${@:$start:$count}"
echo "Looping range of arguments"
for i in "${@:$start:$count}"
do
 echo "$i"
done


$ ./args3.sh 1 2 3 4 5 6
$@       = 1 2 3 4 5 6
```

```
${@:2}    = 2 3 4 5 6
${@:2:3}  = 2 3 4
${@:-4}   = 3 4 5 6
${@:-4:3} = 3 4 5
Looping range of arguments
3
4
5
```

*Note: In the above example, you can replace @ with * to get the same results.*

19. Shift Command

```
shift [n]
```

The Shift command moves the arguments (positional parameters) to the left by n positions. If you don't specify n, the arguments will be moved by 1.

The following script shows how this works.

```
$  cat shift.sh
if [ "$#" -ne "4" ]; then
 echo "Usage: ./shift one two three four"
fi
echo "Before shift:"
printf '"%b"\n' "$@" | cat -n
shift 2
echo "After shift 2:"
printf '"%b"\n' "$@" | cat -n
```

After 'shift 2', arguments "one" and "two" are ignored". "three" become $1, and "four" becomes $2.

```
$  ./shift.sh one two three four
Before shift:
     1  "one"
     2  "two"
     3  "three"
     4  "four"
After shift 2:
     1  "three"
     2  "four"
```

You cannot pass a negative number to shift; once you have shifted you cannot shift back. For example, "shift -2" will display the following error:

```
./shift.sh: line 6: shift: -2: shift count out of range
```

Note: When you are trying to shift more than the total number of arguments, the return status will be 1. In the above example "shift 2" return status is 0. However, if you try to do 'shift 5', the return status will be 1.

20. Bash Special Parameters

The following table explains various bash special parameters.

Special Parameter	Description
$*	Contains all the positional parameters.
$@	Same as above. Contains all the positional parameters.
$#	Total number of positional parameters. i.e total number of arguments passed to the shell script.
$$	Contains the PID of the shell.
$!	Contains the PID of the most recently executed background process.

$?	Contains the exit status of the most recently executed command.
$-	Contains all the options set using the bash set builtin command.
$_	Gives the last argument to the previous command. At the shell startup, it gives the absolute filename of the shell script being executed.

Use $# to Count Positional Parameters

$# is the special parameter, which gives you the total number of positional parameters. Use this at the beginning of a shell script to make sure users are passing the required arguments to the shell script, and display a usage message when they don't provide the required number of arguments.

```
$ cat args-count.sh
#!/bin/bash
if [ $# -lt 2 ]
then
  echo "Usage: $0 arg1 arg2"
  exit
fi
echo -e  "\$1=$1"
echo -e "\$2=$2"
let add=$1+$2
let sub=$1-$2
let mul=$1*$2
let div=$1/$2
echo -e
"Addition=$add\nSubtraction=$sub\nMultiplication=$mul\nDiv
ision=$div\n"
```

If the total number of arguments passed to the script is less than 2, it will throw the usage information as shown below,

```
$ ./args-count.sh 10
Usage: ./args-count.sh arg1 arg2
```

Use $$ to Create Temporary Files and Directories

Inside the shell script, you might want to create some temporary files or directories to do some manipulation. Instead of creating tmp.log file, you can create temp[PID].log, where PID will the process of the bash shell that is spawned to execute the shell script. This is helpful, when you want to create a temporary file (or directory) that needs to be unique when the shell script is executed multiple times simultaneously.

The following script creates a temporary log file and directory under /tmp.

```
$ cat create-temp.sh
echo "Temporary file = /tmp/temp$$.log"
touch /tmp/temp$$.log
echo "Temporary directory = /tmp/temp.$$"
mkdir /tmp/temp.$$

$ ./create-temp.sh
Temporary file = /tmp/temp17134.log
Temporary directory = /tmp/temp.17134
```

After executing the shell script, you'll see the temporary file and directory:

```
$ ls -l /tmp/
drwxr-xr-x.  2 ramesh 4096 Aug 6 23:30 temp.17134/
-rw-r--r--.  1 ramesh    0 Aug 6 23:30 temp17134.log
```

If you create a temporary file or directory inside a shell script that is needed only within the shell script, make sure to delete it at the end of the shell script:

```
rm /tmp/temp$$.log
rm -rf /tmp/temp.$$
```

$?, $-, $_ Examples

```
$ cat special-params.sh
#!/bin/bash
echo "1. [\$_] = $_"
/usr/local/bin/dbhome 2> /dev/null
if [ "$?" -ne "0" ]; then
  echo "2. Previous command execution failed! Checked
using [\$?] "
fi
echo "3. [\$-] = $-"
touch /tmp/test.$$
cp /tmp/test.$$ /tmp/test.log
echo "4. [\$_] = $_"
rm /tmp/test.$$ /tmp/test.log
```

Now, execute the above script.

```
$ ./special-params.sh
1. [$_] = ./special-params.sh
2. Previous command execution failed! Checked using [$?]
3. [$-] = hB
4. [$_] = /tmp/test.log
```

- $_ at the beginning of the script displays the absolute name of the file that was used to execute the script.
- $? is used to check whether the previous command executed successfully. In this case, the command /usr/local/bin/dbhome does not exist, so $? returned 1.
- $- displays the options that were set earlier using the set command.

- $_ used anywhere else in the script (other than the beginning) gives the value of the last argument of the previous command. In this example, the previous command was "cp /tmp/test.$$ /tmp/test.log". The last argument in this command is "/tmp/test.log".

21. Double Quotes (Partial Quoting/Easy Quoting)

Quoting is very important in Bash and you can get into lot of trouble if you don't understand all the nuances of quoting.

First, let us take the simple example of not quoting. Both the following examples produce the same output, as multiple spaces between arguments are treated as just one single space.

```
$ echo Hello World
Hello World

$ echo Hello    World
Hello World
```

When you want to preserve spaces inside a string, you should double quote it, as shown below.

```
$ echo "Hello    World"
Hello    World
```

Basically double quotes keeps the text intact and do not allow bash to split the text at spaces and treat it as multiple arguments. However, double quotes allow variable substitution inside the quoted text using $varname.

Variable assignment

When you are assigning a value to a variable, it is good practice to always double quote the value on the right hand side.

The following are exactly the same, as the value on the right is just a single word. But still use quotes; it is a good habit to develop.

```
$ USERNAME=ramesh

$ USERNAME="ramesh"
```

When you have space in your values, you have to use double quotes.

```
$ USERNAME="ramesh natarajan"
```

If you don't use the double quotes, bash will assign the value "ramesh" to variable USERNAME, and try to execute the command "natarajan".

```
$ USERNAME=ramesh natarajan

-bash: natarajan: command not found
```

Double Quote Inside Double Quotes

If you want to use a double quote inside the double quotes, use \ to escape it as shown below.

```
echo "Hello World"

echo "Hello \"World\""
```

Ignore the Special Meaning of $ Inside Double Quotes

If you don't want the $ symbol to indicate a variable that is to be replaced, use \ to escape it as shown below.

```
$ dollar=100
$ echo "This will not escape dollar: $dollar"

This will not escape dollar: 100


$ echo "This will escape dollar: \$dollar"

This will not escape dollar: $dollar
```

Note: The first echo command showed the value, the second showed the text because the $ symbol was escaped using back slash.

22. Single Quotes (Full Quoting/Strict Quoting)

Just like double quotes, single quotes will preserve spaces:

```
$ USERNAME='ramesh natarajan'
```

However, single quotes will not expand variable values, i.e the $ symbol will be treated as a literal value. In this example, var1 contains "$USERNAME" instead of the value of the variable USERNAME.

```
$ USERNAME="ramesh natarajan"
$ var1='$USERNAME'
$ echo $var1
$USERNAME
```

Anything you use inside single quotes will be used exactly as it is:

```
$ echo 'this has "double quotes" in it'
this has "double quotes" in it
```

Basically single quotes keeps the text intact, do not allow bash to split the text into multiple arguments, and do not allow variable substitution. All the special characters are treated as literal values.

The backslash can be used to escape a special character in order to ignore its meaning even without using either double quotes or single quotes.

```
echo This will not expand \$dollar
```

Note: The above works the same as the following versions which use double quotes and single quotes.

```
echo "This will not expand \$dollar"
echo 'This will not expand $dollar'
```

23. Variable Declaration using Declare

Note: Both typeset and declare builtins are the same, but use declare. Typeset is deprecated and is available only for backward compatibility.

For normal situations, you don't need to declare a variable type to use it.

```
total=3
name=ramesh
echo $total
echo $name
```

However, bash lets you declare integer, read only, array, associative array, and export type variables.

Declare an Integer using declare -i

```
declare -i variablename
```

The following will work whether or not you declare the variable as integer.

```
total=3
let total=total+2
echo $total
```

When you declare a variable as integer, you cannot use that variable to store other types of values. For example, you cannot store a string value to a variable that is declared as integer.

The undeclared variable total can be used to store either number or string:

```
$ total=3
$ echo $total
```

```
3
$ total="Some string value"
$ echo $total
Some string value
```

The following example gives an error message when you try to store a string value to an integer variable.

```
$ declare -i total
$ total=3
$ echo $total
3

$ total="Some string value"
-bash: Some string value: syntax error in expression
(error token is "string value")
$ echo $total
3
```

Read only Variable using declare –r

```
declare -r variablename
```

Use this when you want to assign a value to a variable that should not be allowed to change.

In the following example, the variable BESTSHELL is defined as read-only using the declare command. A value is assigned in the same declare statement.

```
$ declare -r BESTSHELL=Bash
$ echo $BESTSHELL
Bash
```

Once you define a read-only variable, you cannot change its value, as bash will throw an error message:

```
$ BESTSHELL=Korn
-bash: BESTSHELL: readonly variable
```

declare -x

A variable defined in a shell will not be carried over to a child process. For example, if you define a variable in the command line it will not be available to a shell script executed from the command line, as the shell script spawns a new bash to execute the script.

In the following example, a variable global is defined in the command line.

```
$ global=world
$ echo $global
world
```

If you spawn a new bash shell, the value of the global variable is not defined.

```
$ bash
$ echo $global
```

When you go back to the parent shell, you can again see the global variable.

```
$ exit
$ echo $global
world
```

To export a variable to a subshell or child process, use declare -x. In the following example, the variable global is declared with -x.

```
$ declare -x global=world
$ echo $global
world
```

Note: You can also use the export command for this.

Now, when you spawn a new bash shell, the value of the global variable is still available, as it was declared using 'declare -x' in the parent shell.

```
$ bash
$ echo $global
world
```

Note: This is a one way process, i.e the parent shell can send the value to the child process, but any changes made to it in the child will not be sent back to the parent. The child process has its own copy of the variable.

Modify the value of the global in the child process, and exit to the parent process to check the value.

```
$ global=country
$ echo $global
country
$ exit
$ echo $global
world
```

Note: When the declare command is used without any arguments, it will display both the environment variables and user defined local variables.

declare –a and declare -A

Bash allows you to declare arrays (-a option) or associative arrays (-A option).

We discuss these at some length later on.

24. let command

Bash supports arithmetic expressions using the let command.

```
Syntax:
let expression1 expression2 expression3
```

From the command line, you can perform arithmetic without the let command.

```
$ total=3
$ total=$total+3
$ echo $total
6
```

However, this sequence would not work properly in a shell script, as shown in the next example. Therefore, inside a shell script you should use the let command. Also shown in the example is the use of let to do multiple arithmetic expressions on one line

```
$ cat let.sh
total=3
total=total+3
echo "This is wrong: $total"
total=3
let total=total+3
echo "This is correct: $total"
# Multiple arithmetic expressions in a single let command
let total=total+5 sum=5 group=group+5

$ ./let.sh
This is wrong: total+3
This is correct: 6
total=11 sum=5 group=5
```

Note: when using let the variable on the right side of the = doesn't really need to have a $ in front of it. Both of the following will give the same result.

```
let total=total+3
let total=$total+3
```

25. ((expression))

The (( )) syntax tells bash to evaluate the contents as an expression. This is very much like let. Both of the following examples work exactly the same.

```
let total=total+3
((total=total+3))
```

However, within the (( )) you can have spaces:

```
$ (( total = total + 3 ))
$ echo $total
21
```

Spaces in the let command would separate the line into multiple expressions:

```
$ let total = total + 1
-bash: let: =: syntax error: operand expected (error token
is "=")
```

Since (( )) supports spaces, you cannot use multiple expressions on one line:

```
$ let total=total+1 sum=sum+2
$ ((total=total+1 sum=sum+2))
-bash: ((: total=total+1 sum=sum+2: syntax error in
expression (error token is "sum=sum+2")
```

(( )) allows you to use pre- and post- increment/decrement:

```
((total=0))
((total=total+3))
((total++))
((total--))
((++total))
```

You can also write ((total=total+3)) as shown below. Both are the same.

```
((total=total+3))
total=$((total+3))
```

Note: (( )) is helpful for conditional comparisons which are explained in the if statement section.

26. expr command

expr is a unix command (not a bash builtin) which can also be used to evaluate an expression.

```
$ expr total+2
total+2
$ expr $total+2
23+2
```

To evaluate an expression and store it in a variable using expr:

```
total=`expr $total + 2`
```

Let or (( )) or expr -- which one to use?

All of the following are the same:

```
let total=total+2
((total=total+2))
```

```
total=`expr $total + 2`
```

In general, use either let or (( )). Avoid the expr command because it is really calling a unix command to evaluate the arithmetic expression; that means it will unnecessarily spawn a new process.

27. Arithmetic Operators

Basic Arithmetic Operators

- **+** Plus
- **-** minus
- ***** multiplication
- **/** division
- **%** modulo, or mod (remainder of an integer division, e.g. 7%2 = 1)
- ****** exponentiation (for example, 10**4 = 10*10*10*10)

Shortcut Arithmetic Operators

Instead of saying 'total=$total+5', you can use the shortcut arithmetic operator += as shown below.

```
total+=5
```

- **+=** plus-equal (increment variable by a constant)
- **-=** minus-equal (decrement variable by a constant)
- ***=** times-equal (multiply variable by a constant)
- **/=** slash-equal (divide variable by a constant)
- **%=** mod-equal (remainder of dividing variable by a constant)

Auto increment, decrement:

```
let total++
let total--
```

Here is a sample shell script that demonstrates various arithmetic operators:

```
$ cat arithmetic.sh
#!/bin/bash
echo "Basic Arithmetic Operators"
total=10
echo "total = $total"
let total=$total+1
echo "total+1 = $total"
(( total=$total-1 ))
echo "total-1 = $total"
total=`expr 10 \* 2`
echo "total*2 = $total"
let total=$total/2
echo "total/2 = $total"
((total=$total**4))
echo "total%5 = $total"
total=`expr $total % 3`
echo "total%3 = $total"
echo "Shortcut Arithmetic Operators"
total=10
echo "total = $total"
let total+=5
echo "total+=5 is $total"
((total-=5))
echo "total-=5 is $total"
let total*=5
echo "total*=5 is $total"
let total/=5
```

```
echo "total/=5 is $total"
let total%=3
echo "total%=3 is $total"
echo "Auto increment and decrement"
let total++
echo "total++ is $total"
((total--))
echo "total-- is $total"
```

The following is the output of the above arithmetic shell script.

```
$ ./arithmetic.sh
Basic Arithmetic Operators
total = 10
total+1 = 11
total-1 = 10
total*2 = 20
total/2 = 10
total%5 = 10000
total%3 = 1
Shortcut Arithmetic Operators
total = 10
total+=5 is 15
total-=5 is 10
total*=5 is 50
total/=5 is 10
total%=3 is 1
Auto increment and decrement
total++ is 2
total-- is 1
```

Bash does not understand floating point arithmetic. It treats numbers containing a decimal point as strings. This restriction applies even if you use let, (( )), or expr.

```
$ total=1.5
$ let total=$total+1
-bash: let: total=1.5+1: syntax error: invalid arithmetic
operator (error token is ".5+1")
```

Also, note that if you use expr for arithmetic expressions, you need to specify an escape character (\) for most of the arithmetic operators. For example, * needs an escape char as shown below.

```
$ expr 2 * 3
expr: syntax error
$ expr 2 \* 3
6
```

Operator Precedence

Bash evaluates an expression in a particular order based on the 'precedence' of the operators. The following table lists highest precedence operators first. To be completely safe, use parentheses to control the order of evaluation!

Operator	Description
var++, var--	Post auto increment and auto decrement
++var, --var	Pre auto increment and auto decrement
-, +	Unary minus and plus
!, ~	Logical and bitwise negation
**	Exponentiation
***, /, %**	Multiply, divide, and modulo operator
+, -	Add and subtract
<<, >>	Bit-wise left and right shift

<=, >=, <, >	Comparisons
==, !=	Equal-to and not-equal-to
& ^ |	Operators in the same order mentioned here. AND, XOR, OR
&&	Logical AND
||	Logical OR
expr ? expr : expr	Ternary operator
=, *=, /=, %=, +=, -=, <<=, >>=, &=, ^=, |=	Assignments

Chapter 4. Bash Conditional Commands and Loops

28. Number Comparison

Before we start discussing if conditions, for loops, and while loops, let us understand what operators we can use for numeric and string comparisons in the [ ] and test command of a if statement.

To compare numbers (integers) use the following operators.

Operator	Description
-eq	Equal to
-ne	Not Equal to
-gt	Greater than
-ge	Greater than or equal to
-lt	Less than
-le	Less than or equal to

The following example shows how to use the number comparison operators in an if statement.

```
$ cat operators-integer.sh
total=${1}
if [ $total -eq 100 ]; then
 echo "-eq: total is equal to 100"
fi
if [ $total -ne 100 ]; then
 echo "-ne: total is NOT equal to 100"
fi
if [ $total -lt 100 ]; then
```

```
 echo "-lt: total is less than 100"
fi
if [ $total -gt 100 ]; then
 echo "-gt: total is greater than 100"
fi
if [ $total -le 100 ]; then
 echo "-le: total is less than or equal to 100"
fi
if [ $total -ge 100 ]; then
 echo "-le: total is greater than or equal to 100"
fi
```

Execute the above shell script by passing an argument as shown below to see how the various test conditions work:

```
$ ./operators-integer.sh 80
-ne: total is NOT equal to 100
-lt: total is less than 100
-le: total is less than or equal to 100
```

29. String Comparison

To compare strings use the following operators.

Operator	Description
=	Equal to
==	Equal to
!=	Not Equal to
<	Less than
>	Greater than
-z	Zero byte? Is the given string empty?

-n	Not empty? Is the given string not empty?

The following example shows how to use the string comparison operators in an if statement.

```
$ cat operators-string.sh
state=${1}
if [ "$state" == "california" ]; then
  echo "==: state is california"
fi
if [ "$state" != "california" ]; then
  echo "==: state is not california"
fi
if [ "$state" \< "indiana" ]; then
 echo "<: state comes before indiana"
fi
if [ "$state" \> "indiana" ]; then
 echo ">: state comes after indiana"
fi
if [ -n "$state" ]; then
 echo "-n: state is not null"
fi
if [ -z "$city" ]; then
 echo "-z: city is null"
fi
```

Execute the above shell script by passing an argument to see how the various test conditions work:

```
$ ./operators-string.sh california
==: state is california
<: state comes before indiana
-n: state is not null
```

```
-z: city is null
```

Please note: In the above example, a single equal sign (=), is the same as a double equal sign (==). Both do string comparison inside the if […]; test construct. But, to avoid any confusion, I recommend that you always use == for comparison and = for assignment.

Additional notes on string comparison and the example

- -n and -z need to be added before the variable to be tested. Otherwise bash won't have anything to test.
- -z stands for zero-byte. While using -z, we are really checking whether the given variable is either zero-byte or not null.
- The variable city was never defined anywhere before. So, when you use -z to check whether it is null, the test will return true.
- When using < or >, bash does an ASCII comparison. For example, "apple" < "ball" is true, as "a" comes before "b".
- While using < or > inside an if test construct, you should escape it using a backslash.

You can use the combination of -z and &&, to start a process when it is not running. The following example starts apache only if it is not already running.

```
[ -z "`/bin/ps ax | /bin/grep httpd | /bin/grep -v
/bin/grep`" ] && /usr/local/bin/apache2/apachectl start
```

When you are referencing a variable, you should always double quote it to avoid any errors. In the following example, $city in the "[-z $city]" is not double quoted. This will not give any error when the city doesn't contain any space in it as shown below.

```
$ city="Vegas"

$ [ -z $city ] && echo "City is null"
```

However when you have space in the value of the variable city, this will fail and throw an error message as shown below.

```
$ city="Las Vegas"
$ [ -z $city ] && echo "City is null"
-bash: [: Las: binary operator expected
```

If you double quote $city, bash will not throw any error messages.

```
$ city="Las Vegas"
$ [ -z "$city" ] && echo "City is null"
```

30. if command

if then fi

As shown below, the if statement can be used to evaluate the output of any bash statement. In the following syntax, statement1 is executed and if the output is true, bash will execute statements 2 and 3.

General Syntax:

```
if statement1
then
     statement2
     statement3
     ..
fi
```

In most cases, statement 1 will be a conditional expression in brackets, also called the [command; this is called a simple if statement. If the conditional expression is true (nonzero), bash executes the statements enclosed between the keywords “then” and “fi”, otherwise the statement list is skipped.

Simple Syntax using [command:

```
if [ conditional expression ]
then
     statement1
     statement2
     ..
fi
```

The keyword "then" should be in a separate line. You can use the keyword “then” in the same line as "if" by adding a semicolon as shown below:

```
if [ conditional expression ]; then
     statement1
     statement2
     ..
fi
```

The keyword “test” may be used instead of brackets.

Syntax using test command:

```
if test conditional expression
then
     statement1
     statement2
     ..
fi
```

The following example shows the three variations of the simple if statement.

```
$ cat if-simple.sh
total=${1}
if [ $total -eq 100 ]
```

```
then
 echo "total is equal to 100"
fi
if test $total -eq 100
then
 echo "total is equal to 100"
fi
if [ $total -eq 100 ]; then
 echo "total is equal to 100"
fi
```

Execute the above shell script to view the results.

```
$ ./if-simple.sh 100
total is equal to 100
total is equal to 100
total is equal to 100
```

if then else fi

You can include an "else" branch if you need statements to be executed when the conditional expression is false.

If – else Syntax:

```
if [ conditional expression ]
then
     statement1
     statement2
     ..
else
     statement3
     statement4
     ..
fi
```

If the conditional expression is true, bash executes the statements 1 and 2. If the conditional expression is false, it executes statements 3 and 4. In either case, execution resumes with the subsequent statements. This example shows the usage of the if-then-else-fi statement:

```
$ cat if-else.sh
total=${1}
if [ $total -eq 100 ]
then
 echo "total is equal to 100"
else
 echo "total is not equal to 100"
fi
if test $total -eq 100
then
 echo "total is equal to 100"
else
 echo "total is not equal to 100"
fi
if [ $total -eq 100 ]; then
 echo "total is equal to 100"
else
 echo "total is not equal to 100"
fi
```

Execute the above shell script to view the results.

```
$ ./if-else.sh 101
total is not equal to 100
total is not equal to 100
total is not equal to 100
```

if then elif then else fi

Elif is short for “else if”. This construct allows you to to select one of many blocks of code to execute by testing two or more conditional expressions.

If-elif-else Syntax:

```
if [ conditional expression1 ]
then
     statement1
     statement2
     ..
elif [ conditional expression2 ]
then
     statement3
     statement4
     ..
else
     statement5
fi
```

- If conditional expression1 is true, bash executes statements 1 and 2. Otherwise, if conditional expression2 is true, it executes statements 3 and 4. Otherwise, it executes statement 5.
- You can use as many elif blocks as you need, but remember that the expressions will be tested in order from top to bottom so it will take longer to get to the final else block. In time critical applications this can be important.

The following example shows the usage of the if-elif-else-fi statement.

```
$ cat if-elif.sh
total=${1}
if [ $total -eq 100 ]
```

```
then
 echo "total is equal to 100"
elif [ $total -lt 100 ]
then
 echo "total is less than 100"
else
 echo "total is greater than 100"
fi

$ ./if-elif.sh 101
total is greater than 100
```

if then else if then fi fi (Nested if)

The if statement and else statement can be nested in bash. This is essentially identical to using elif, but with an important difference: each if statement must end with a corresponding fi.

Syntax:

```
if [ conditional expression1 ]
then
     statement1
     statement2
     ..
else
     if [ conditional expression2 ]
     then
           statement3
           ..
     fi
fi
```

The following example shows the the usage of if-else-fi statement.

```
$ cat if-else-if.sh
total=${1}
if [ $total -eq 100 ]
then
 echo "total is equal to 100"
else
 if [ $total -lt 100 ]
 then
  echo "total is less than 100"
 else
  echo "total is greater than 100"
 fi
fi

$ ./if-else-if.sh 101
total is greater than 100
```

Arithmetic Expressions in an if Statement

You have to be very careful when you are evaluating arithmetic expressions in the conditional part of your if statement.

For the shell, you already know the following exit status:

- 0 means the last command executed successfully (i.e true).
- 1 means the last command execution failed (i.e false).

However inside an if statement, when an arithmetic expression's result is 0, it is considered false, and when an arithmetic expression's result is anything except 0, it is considered true.

This is explained in the following example:

```
$ cat if-arithmetic-expression.sh
if ((total=total+0)); then
 echo "Expression total=total+0 returned true."
else
 echo "Expression total=total+0 returned false."
fi
if ((total=total+1)); then
 echo "Expression total=total+1 returned true."
else
 echo "Expression total=total+1 returned false."
fi

$ ./if-arithmetic-expression.sh
Expression total=total+0 returned false.
Expression total=total+1 returned true.
```

31. File Test Operators in If Condition

In Linux everything is treated as a file, including the directory, a block device, a character device, or a traditional regular file. So, all the tests below are called file test operators.

In all the explanations below, "regular file" has a special meaning: it is a file in the file system, such as /etc/passwd. The word "file" mentioned by itself could be any types of file (directories, block devices, regular files, etc.,)

File Test Operator	Description
-e	File exists (this could be regular file, directory, block device, character device, etc.,)
-f	It's a regular file (for example: /etc/passwd)

File Test Operator	Description
-d	It's a directory (for example: /etc)
-b	It's a block device (for example: /dev/sda1)
-c	It's a character device (for example: /dev/tty1)
-s	File is not empty
-p	It's a pipe
-S	It's a socket
-h	It's a symbolic link
-t	Checks whether the given FD is opened in a terminal.
-r	File read permission
-w	File write permission
-x	File execute permission
-u	suid set on the file
-g	sgid set on the file
-k	Sticky bit set on the file
-O	You own the file
-G	File group id and my group id are the same.
-N	Did the file got modified since last read?
file1 -nt file2	File1 is newer than file2
file1 -ot file2	File1 is older than file2
file1 -ef file2	Both file1 and file2 are hard linked to the same file

Note: You can reverse the meaning of any of the above conditions by adding a ! in front of them.

Check if the file exists

As we explained above, file could be anything. So, all of the following will return true.

- -e /etc
- -e /etc/passwd
- -e /dev/sda

Some file exists examples:

```
$ cat if-exists.sh
if [ -e /etc/ ]; then
 echo "/etc directory exists"
fi
if [ -e /etc/passwd ]; then
 echo "/etc/passwd regular file exists"
fi
if [ -e /dev/sda1 ]; then
 echo "/dev/sda1 block device file exists"
fi
if [ -e /dev/tty1 ]; then
 echo "/dev/tty1 character device file exists"
fi
if [ -e /etc/rc.local ]; then
 echo "/etc/rc.local symbolic link file exists"
fi
```

Execute the above shell script to view the output.

```
$ ./if-exists.sh
/etc directory exists
```

```
/etc/passwd regular file exists
/dev/sda1 block device file exists
/dev/tty1 character device file exists
/etc/rc.local symbolic link file exists
```

Check Whether a Specific File Type Exists

You might want to know whether a specific directory, or link, or block device exists. Use the appropriate file test condition as shown in the next example below for this testing.

Note: For most file exists scenarios, you'll be testing whether a directory exists, or a regular file exists. Instead of using -e, I always prefer to specify the exact type of the file that I'm looking for, with –d or –f or –b or –h.

The following example explains how to test for a specific file type.

```
$ cat if-file-type-exists.sh
if [ -d /etc/ ]; then
 echo "/etc exists and it is a directory"
fi
if [ -f /etc/passwd ]; then
 echo "/etc/passwd exists and it is a regular file"
fi
if [ -b /dev/sda1 ]; then
 echo "/dev/sda1 exists and it is a block device"
fi
if [ -c /dev/tty1 ]; then
 echo "/dev/tty1 exists and it is a character device"
fi
if [ -h /etc/rc.local ]; then
 echo "/etc/rc.local exists and it is asymbolic link"
fi
```

Check for Permissions

-r, -w, and -x check whether you have read, write, and execute permission, respectively, on the specified file (or directory):

```
$ cat if-file-permission-exists.sh
filename=/etc/passwd
if [ -r $filename ]; then
 echo "You have read permission on $filename";
fi
if [ -w $filename ]; then
 echo "You have write permission on $filename";
fi
if [ -x $filename ]; then
 echo "You have execute permission on $filename";
fi
```

Note: Try the permission testing example from both root and non-root accounts to see different levels of access.

Reverse Test Conditions with !

You can reverse the meaning of any test condition by using ! in front of it.

For example, -s checks whether the file is not empty. If you want to check whether the file empty, use ! –s:

```
$ cat if-file-empty.sh
if [ -s /etc/passwd ]; then
 echo "/etc/passwd is not empty"
fi
touch /tmp/file1
if [ ! -s /tmp/file1 ]; then
 echo "/tmp/file1 is empty!"
fi
```

```
$ ./if-file-empty.sh
/etc/passwd is not empty
/tmp/file1 is empty!
```

32. AND, OR, Not Inside If Statement Using -a, -o, !

You can evaluate multiple expressions using -a or –o.

AND Comparison using -a

-a does “and” comparison inside a conditional expression

```
if [ "expression1" -a "expression2" ]
then
 statement1
 statement2
 ..
fi
```

Expressions can be either number comparison or string comparison. The following example demonstrates “and” numeric comparison:

```
$ cat and-comparision-numeric.sh
total=${1}
if [ $total -ge 50 -a $total -le 100 ]
then
 echo "total between 50 and 100"
fi
if [ $total -ge 1 -a $total -le 49 ]
then
 echo "total between 1 and 49"
```

```
fi

$ ./and-comparision-numeric.sh 10
total between 1 and 49
```

The following example demonstrates “and” string comparison:

```
$ cat and-comparision-string.sh
state=${1}
capital=${2}
if [ $state == "CA" -a "$capital" == "sacramento" ]
then
 echo "Yes. California capital is sacramento."
fi
if [ $state == "CA" -a "$capital" != "sacramento" ]
then
 echo "No. California capital is not $capital"
fi

$ ./and-comparision1.sh CA "los angeles"
No. California capital is not los angeles
```

OR Comparison using -o

-o does “or” comparison inside a conditional expression.

```
$ cat or-comparision.sh
input=${1}
if [ "$input" == "apple" -o "$input" == "orange" ]
then
 echo "Yes. $input is a fruit."
fi
```

```
$ ./or-comparision.sh orange
Yes. orange is a fruit.
```

Note: The above example uses string comparison but –o can be used with numeric expressions, just like –a.

Logical !

"! expression" is a logical NOT. We demonstrated the use of “! –s” earlier, but the ! may be used to reverse the result of any expression, for example:

```
$ cat not-comparision.sh
input=${1}
if [ ! "$input" == "apple" ]
then
 echo "No. $input is not a fruit."
fi

$ ./not-comparision.sh chicken
No. chicken is not a fruit.
```

33. && - Combine Multiple Commands with AND

&& is used to execute sequence of commands only when the previous command is successfully executed with a return status of zero.

Syntax:

```
command1 && command2 && command3 && ..
```

- If command1 executes successfully (with a return status of 0), bash moves on and executes command2.
- If command1 fails (with a return status of non zero), bash doesn't execute command2 (or command3 or anything else on that line).
- If command2 executes successfully, bash executes command3.

- If command2 fails, bash skips the rest of the line.

There are 5 commands separated by && in the following example, and all of them are executed because none of them fail:

```
$ ls -ld /etc && x=10 && echo "value of x is: $x" && pwd
&& echo "finished"

drwxr-xr-x. 86 root root 4096 Aug 6 06:37 /etc

value of x is: 10

/root

finished
```

The following example also has 5 commands separated by &&. However, the second command failed (because we've given space before and after the equal sign in variable assignment). Since command2 failed, command3 and 4 and 5 never got executed.

```
$ ls -ld /etc && x = 10 && echo "value of x is: $x" && pwd
&& echo "finished"

drwxr-xr-x. 86 root root 4096 Apr 6 06:37 /etc

-bash: x: command not found
```

Note: The return status of the whole list is exactly the same as the exit status of the last command in the list that failed.

You can use && in an if statement as shown below. When all the commands execute successfully, you can further execute one or more commands in the if command body.

In this example, several commands are separated by && in an if statement:

```
$ cat and-list1.sh

passwd=/etc/passwd

group=/etc/group
```

```
if [ -f $passwd ] && x=`wc -l $passwd|awk '{print $1}'` &&
[ -f $group ] && y=`wc -l $group|awk '{print $1}'` && let
total=$x+$y

then

 echo "Total number of lines in $passwd and $group files
are: $total"

fi

echo "Finished"


$ ./and-list1.sh

Total number of lines in /etc/passwd and /etc/group files
are: 88

Finished
```

Since we have only one command in the body (then part) of the if statement, we can remove the if command altogether and make the echo statement the last command in the && chain. But if we had more commands in the body of the if statement the form shown previously would be best.

The next example shows how to use && for controlling all the commands:

```
$ cat and-list2.sh

passwd=/etc/passwd

group=/etc/group

[ -f $passwd ] && x=`wc -l $passwd|awk '{print $1}'` && [
-f $group ] && y=`wc -l $group|awk '{print $1}'` && let
total=$x+$y && echo "Total number of files in $passwd and
$group files are: $total"


$ ./and-list2.sh

Total number of files in /etc/passwd and /etc/group files
are: 88
```

The commands used in the last 2 examples were:

- [-f $passwd] - Check whether the passwd file exists
- x=`wc -l $passwd|awk '{print $1}'` - Calculate the total number of lines in the passwd file and store it in variable x
- [-f $group] - Check whether the group file exists
- y=`wc -l $group|awk '{print $1}'` - Calculate the total number of lines in the group file and store it in variable y
- let total=$x+$y - Add x and y and stores the result in variable total.

To summarize: to use the test command [ ], you really don't need to use the if command; you can use && to test the result and control program flow.

A simple example of if replacement is to change the following:

```
if [ "$BACKUP" = "yes" ]
then
  tar cvf home.tar /home
fi
```

to:

```
[ "$BACKUP" = "yes" ] && tar cvf backup.tar /home
```

Another good use of && is shown below: cd only when the mkdir succeeds.

```
mkdir /home/ramesh/test && cd /home/ramesh/test
```

34. || - Combine Multiple Commands with OR

|| is called the OR list construct, which is used to execute exactly one command in a sequence of commands. Each command in the “or” list

will be executed only if the previous command failed (i.e. returned a non-zero status). The behavior of || is exactly the opposite of &&.

Syntax:

```
command1 || command2 || command3 || ..
```

- If command1 executes successfully (return status of 0), bash doesn't execute command2 or command3
- If command1 fails (return status of non zero), it executes command2.
- If command2 executes successfully, bash doesn't execute command3.
- If command2 failed, it executes command3.

The following example has 5 commands separated by ||. However, since command 1 got executed successfully, the rest of the commands in this list will never be executed.

```
$ ls -ld /etc || x=10 || echo "value of x is: $x" || pwd
|| echo "finished"
drwxr-xr-x. 86 root root 4096 Apr 28 06:37 /etc
```

One good use for the OR list is to replace a simple if ! statement:

```
if [ ! -f /.datafile ]
then
 touch /.datafile
fi
```

The code snippet below does exactly the same as the one above.

```
[ -f /.datafile ] || touch /.datafile
```

Another use of || is to exit a shell script with appropriate status when a necessary operation fails, as shown below.

```
touch /tmp/temp.dat || { echo 'Unable to create
/tmp/temp.dat file' >&2; exit 1; }
```

35. [[ ]] - Extended Test Command

As we know already, [ ] is the test command, typically used with the if statement.

[[ ]] is the extended test command, an advanced variation of the [ ] command that does everything [ ] does, plus the following:

- Pattern matching in the string comparison, when you use **=**, **==**, or **!=**
- **=~** inside extended test command
- **&&** and || inside the extended test command
- **<** and **>** for comparison inside the extended test command
- enclosing an expression in **( )** to change its evaluation precedence
- Tilde expansion, arithmetic expansion, and parameter expansion
- Command substitution and process substitution

The following example shows how to use pattern matching (using =, ||, and =~), using the extended test command.

```
$ cat extended-test1.sh
name=${1}
# 1. You can use pattern matching in [[ ]]
if [ $name = bon* ]; then
 echo "[test] command: Good Morning, Mr. Bond"
fi
if [[ $name = bon* ]]; then
 echo "[[test]] command: Good Morning, Mr. Bond"
```

```
fi
# 2. You can use || && inside [[ ]]
if [[ $name = super || $name = bon* ]]; then
 echo "[[test]] command: Good Morning, Mr. Hero"
fi
# 3. You can use =~ for regular expression inside [[ ]]
if [[ $name =~ ^b ]]; then
 echo "[[test]] command: Mr. $name, your name starts with
"b""
fi


$ ./extended-test1.sh bon
[[test]] command: Good Morning, Mr. Bond
[[test]] command: Good Morning, Mr. Hero
[[test]] command: Mr. bon, your name starts with b
```

The following example shows how to use pattern matching (using =), and regex comparison (using =~), using the extended test command.

```
$ cat extended-test2.sh
i=1
cd ~
for item in *
do
 if [[ $item = *.jpg ]]; then
  echo "Image $((i++)) : $item"
 fi
done
cd /etc
for item in *.conf
do
 if [[ $item =~ ^a ]]; then
```

```
   echo "Conf  $((i++)) : $item"
  fi
done

$ ./extended-test2.sh
Conf  1 : asound.conf
Conf  2 : autofs_ldap_auth.conf
```

36. Bash For Loops – Cycle through a List of values

For loops allow repeated execution of a command sequence based on an iteration variable. Bash supports two kinds of for loop, a “list of values” and a “traditional” c-like method that we discuss in a later section.

“list of values” Syntax:

```
for varname in list
do
     commands ##Body of the loop
done
```

- for, in, do and done are keywords
- list contains a list of items, which can be in the statement or fetched from a variable that contains several words separated by spaces. If list is missing from the for statement, then bash uses positional parameters that were passed into the shell.
- varname is any Bash variable name.

In this form, bash executes the command(s) enclosed in the body (between do and done) once for each item in the list. If the list contains 5 items, the for loop will be executed a total of 5 times. Each time through, the current item from the list will be stored in variable “varname” before the body executes. So, this varname can be processed in the body of the loop.

The following several examples show how to use the bash for loop “list of values” method.

Static Values as For List Items

In the following example, the list of values (Mon, Tue, Wed, Thu and Fri) is directly given after the keyword "in".

```
$ cat for1.sh
i=1
for day in Mon Tue Wed Thu Fri
do
 echo "Weekday $((i++)) : $day"
done
```

Execute this example:

```
$ ./for1.sh
Weekday 1 : Mon
Weekday 2 : Tue
Weekday 3 : Wed
Weekday 4 : Thu
Weekday 5 : Fri
```

The list of values should not be separated by comma (Wrong: Mon, Tue, Wed, Thu, Fri). The comma will be treated as part of the value, i.e. instead of "Mon", bash will use "Mon," as the value as shown below.

```
$ cat for1-wrong1.sh
i=1
for day in Mon, Tue, Wed, Thu, Fri
do
 echo "Weekday $((i++)) : $day"
done
```

```
$ ./for1-wrong1.sh
Weekday 1 : Mon,
Weekday 2 : Tue,
Weekday 3 : Wed,
Weekday 4 : Thu,
Weekday 5 : Fri
```

The list of values should not be enclosed in quotes. (Wrong: "Mon Tue Wed Thu Fri"). If you enclose in double quotes, it will be treated as a single value (instead of 5 different values), as shown below.

```
$ cat for1-wrong2.sh
i=1
for day in "Mon Tue Wed Thu Fri"
do
 echo "Weekday $((i++)) : $day"
done


$ ./for1-wrong2.sh
Weekday 1 : Mon Tue Wed Thu Fri
```

Using a Variable as List

Instead of providing the values directly in the for loop, you can store the values in a variable, and use the variable in the for loop after the "in" keyword, as shown in the following example.

```
$ cat for2.sh
i=1
weekdays="Mon Tue Wed Thu Fri"
for day in $weekdays
do
 echo "Weekday $((i++)) : $day"
done
```

```

$ ./for2.sh
Weekday 1 : Mon
Weekday 2 : Tue
Weekday 3 : Wed
Weekday 4 : Thu
Weekday 5 : Fri
```

Caution: As a best practice, you should always quote bash variables when you are referring to them, except in special cases. This is one of the special cases. if you double quote the variables in this for loop, the list of values will be treated as single value. Lots of people fall into this trap! Be careful and do not double quote your variables in for loops.

```
$ cat for2-wrong.sh
i=1
weekdays="Mon Tue Wed Thu Fri"
for day in "$weekdays"
do
 echo "Weekday $((i++)) : $day"
done

$ ./for2-wrong.sh
Weekday 1 : Mon Tue Wed Thu Fri
```

Get List Values from Positional Parameters

If you don't specify the keyword "in" followed by a list of values in the bash for loop, it will use the positional parameters passed to the shell script:

```
$ cat for3.sh
i=1
for day
do
```

```
 echo "Weekday $((i++)) : $day"
done

$ ./for3.sh Mon Tue Wed Thu Fri
Weekday 1 : Mon
Weekday 2 : Tue
Weekday 3 : Wed
Weekday 4 : Thu
Weekday 5 : Fri
```

Note: be careful if you use this method. You should not include the keyword "in" when specifying the for loop, because bash would look see an empty list and never execute the loop. Here's an example:

```
$ cat for3-wrong.sh
i=1
for day in
do
 echo "Weekday $((i++)) : $day"
done

$ ./for3-wrong.sh Mon Tue Wed Thu Fri
```

Use a Command Output For List of Values

You can use the output of any UNIX / Linux command as the list of values to the for loop by enclosing the command in back-ticks ` ` as shown in the next example.

List the usernames from the passwd file:

```
$ cat for4.sh
i=1
for username in `awk -F: '{print $1}' /etc/passwd`
```

```
do
 echo "Username $((i++)) : $username"
done

$ ./for4.sh
Username 1 : ramesh
Username 2 : john
Username 3 : preeti
Username 4 : jason
..
```

Looping Through Files and Directories

You can loop through the files and directories under a specific directory with a for loop. Just cd to that directory, and give * in the for loop as shown below.

Loop through all the files and directories under your home directory:

```
$ cat for5.sh
i=1
cd ~
for item in *
do
 echo "Item $((i++)) : $item"
done

$ ./for5.sh
Item 1 : positional-parameters.sh
Item 2 : backup.sh
Item 3 : emp-report.awk
Item 4 : item-list.sed
Item 5 : employee.db
```

The usage of * in the bash for loop is similar to the file globbing that we use in the linux command line for the ls command (and several other commands).

For example, the following will display all the files and directories under your home directory. This is the concept that is used in the above for5.sh example.

```
cd ~
ls *
```

The following will display all the *.conf files that begin with either a, b, c, or d under the /etc directory.

```
$ ls -1 /etc/[abcd]*.conf
/etc/asound.conf
/etc/autofs_ldap_auth.conf
/etc/cas.conf
/etc/cgconfig.conf
/etc/cgrules.conf
/etc/dracut.conf
```

The same argument that is used for an ls command can be used in a bash for loop, as shown in the example below.

```
$ cat for5-1.sh
i=1
for file in /etc/[abcd]*.conf
do
 echo "File $((i++)) : $file"
done

$ ./for5-1.sh
File 1 : /etc/asound.conf
File 2 : /etc/autofs_ldap_auth.conf
File 3 : /etc/cas.conf
File 4 : /etc/cgconfig.conf
File 5 : /etc/cgrules.conf
File 6 : /etc/dracut.conf
```

Break Out of a Bash For Loop

You can break out of a for loop using 'break' command as shown below.

```
$ cat for6.sh
i=1
for day in Mon Tue Wed Thu Fri
do
 echo "Weekday $((i++)) : $day"
 if [ $i -eq 3 ]; then
   break;
 fi
done

$ ./for6.sh
Weekday 1 : Mon
Weekday 2 : Tue
```

Continue a Bash For Loop

Under certain conditions, you can ignore the rest of the commands in the body of a for loop, and continue the loop from the top again (using the next value in the list), using the continue command as shown below.

The following example adds "(WEEKEND)" to Sat and Sun, and "(weekday)" to rest of the days.

```
$ cat for7.sh
i=1
```

```
for day in Mon Tue Wed Thu Fri Sat Sun
do
 echo -n "Day $((i++)) : $day"
 if [ $i -eq 7 -o $i -eq 8 ]; then
   echo " (WEEKEND)"
   continue;
 fi
 echo " (weekday)"
done
```

The output from this example:

```
$ ./for7.sh
Day 1 : Mon (weekday)
Day 2 : Tue (weekday)
Day 3 : Wed (weekday)
Day 4 : Thu (weekday)
Day 5 : Fri (weekday)
Day 6 : Sat (WEEKEND)
Day 7 : Sun (WEEKEND)
```

You may have noticed that an ‘else’ statement containing the second ‘echo’ would have worked too. There are usually many ways to do the same thing.

Range of numbers using "in"

You can loop through a range of numbers in the for loop by using brace expansion after the ‘in’ keyword.

The following example loops through the values 1 through 10.

```
$ cat for8.sh
for num in {1..10}
do
```

```
 echo "Number: $num"
done

$ ./for8.sh
Number: 1
Number: 2
Number: 3
Number: 4
Number: 5
...
```

Range of Numbers With Increment using "in"..

The following example loops through a set of numbers but each time through the loop variable is incremented by two instead of one. The loop will execute five times because the loop variable will be out of range after that.

Loop from 1 to 10 by 2, i.e. show the odd numbers from 1 to 10:

```
$ cat for9.sh
for num in {1..10..2}
do
 echo "Number: $num"
done

$ ./for9.sh
Number: 1
Number: 3
Number: 5
Number: 7
Number: 9
```

37. Bash for loops using C like syntax

The second form of bash for loop is similar to the ‘C’ programming language for loop, which has three expressions (initialization, condition and update).

```
for (( expr1; expr2; expr3 ))
do
     commands
done
```

- Before the first iteration, expr1 is evaluated. This is usually used to initialize variables for the loop.
- All the statements between do and done are executed repeatedly as long as the value of expr2 is TRUE.
- After each loop iteration, expr3 is evaluated. This is usually used to increment a loop counter.

The following examples show how to use this syntax in the bash for loop.

Loop using C-Style

Generate and display 5 random numbers using the bash C-style for loop:

```
$ cat for10.sh
for (( i=1; i <= 5; i++ ))
do
 echo "Random number $i: $RANDOM"
done

$ ./for10.sh
Random number 1: 23320
Random number 2: 5070
Random number 3: 15202
Random number 4: 23861
Random number 5: 23435
```

Infinite Loop Using Bash For

When you don't provide the start, condition, and increment in a C-style for loop, it will execute forever. You need to press Ctrl-C to stop the loop.

```
$ cat for11.sh
i=1;
for (( ; ; ))
do
   sleep $i
   echo "Number: $((i++))"
done
```

Note: Don’t forget you will need to press Ctrl-C to break from this example:

```
$ ./for11.sh
Number: 1
Number: 2
Number: 3
```

Increment Two Values Using Comma in C-style for loop

In the bash c-style loop, in addition to incrementing the value that is used in the condition, you can also increment some other value or perform some other action. In both the initialization section and the increment section of the C-style for loop, you can use multiple statements separated with a comma. This example uses i for control and manipulates j separately:

```
$ cat for12.sh
for ((i=1, j=10; i <= 5 ; i++, j=j+5))
do
 echo "Number $i: $j"
done
```

```
$ ./for12.sh
Number 1: 10
Number 2: 15
Number 3: 20
Number 4: 25
Number 5: 30
```

38. While Loop

Another iteration statement offered by the bash shell is the while statement.

```
while expression
do
     commands
done
```

- while, do, done are keywords
- Expression is any expression which returns a scalar value
- Commands between do and done are executed while the provided conditional expression is true.

This example reads data from stdin (the keyboard) and writes it into a file. EOF (Ctrl-Z) is needed to terminate the input:

```
$ cat while-writefile.sh
#! /bin/bash
echo -e "Enter absolute path of the file you want to
create"
read file
while read line
do
```

```
   echo $line >> $file
done
```

Note: In this example the parts you type are in bold. Ctrl-z means you type control z to terminate input.

```
$ sh while-writefile.sh
Enter absolute path of the file you want to create
/tmp/a
while
for
until
Ctrl-Z

$ cat /tmp/a
while
for
until
```

In the previous example, bash reads the filename from the user, and then reads lines of data from stdin until it sees EOF, appending each line to the file specified by filename. Bash redirection of the echo command output is used to append the lines. EOF terminates the loop because the read fails (i.e. it returns a nonzero value).

The next example has bash write data from a file to stdout (similar to ‘cat’). The input is still taken from stdin in this example, but bash redirection is used so that stdin comes from the specified file rather than from the keyboard.

Note how both of these examples have made use of the very powerful technique of redirection that we discuss more in a later section.

Read a file and echo it to stdout:

```
$ cat while-read.sh
#! /bin/bash
echo -e "Enter absolute path of the file name you want to
read"
read file
exec <$file # redirects stdin to a file
while read line
do
echo $line
done
```

Note: In this example the parts you type are in bold.

```
$ ./while-read.sh
Enter absolute path of the file name you want to read
/tmp/a
while
for
until
```

In this example, bash gets the filename to read from stdin (the keyboard) and then uses exec to redirects stdin, getting further input from a file instead of the keyboard. As in the previous example, the read command fails when EOF occurs, terminating the loop.

Infinite Loop Using Bash While

You can use “while true” to create an infinite loop, just like the for (;;) syntax demonstrated earlier.

The next example uses an infinite loop to test whether a server is up and running. The sleep command is used to do the ping test at a periodic interval (in this case sixty seconds). The script will be executed with the command “nohup” or no hangup, and with & at the end of the command line so that it is detached from the current shell. In this way the script will continue as long as the system is still running, even if you log out.

Example: Test the server with ip-address 192.168.100.10 every 60 seconds, and display a message if it fails.

```
$ cat until-loop.sh
IP=192.168.101.10
while true
do
 ping -c 2 $IP > /dev/null
 if [ $? -ne 0 ]; then
   echo "Server $IP is not responding"
 fi
 sleep 2
done

$ nohup until-loop.sh &
```

39. Until Loop

The bash until loop is very similar in syntax and function to the while loop. The only difference between the two is that the until statement executes its code block while its conditional expression is false, and the while statement executes its code block while its conditional expression is true.

```
until expression
do
     commands #body of the loop
done
```

- until, do, and done are keywords
- expression is any conditional expression

This bash until loop example monitors the size of the logfile. Once the logfile size reaches 2000 bytes, it makes a copy of that logfile.

```
$ cat until-monitor.sh
file=/tmp/logfile
until [ $(ls -l $file | awk '{print $5}') -gt 2000 ]
do
   echo "Sleeping for next 5 seconds"
   sleep 5
done
date=`date +%s`
cp $file "$file-"$date.bak

$ ./until-monitor.sh
Sleeping for next 5 seconds
Sleeping for next 5 second
```

The following bash until example is used to wait till a remote machine comes up before connecting via ssh to that machine. The until loop ends only when the ping succeeds (returns 0).

```
$ cat until-ping-ssh.sh
read -p "Enter IP Address:" ipadd
echo $ipadd
until ping -c 1 $ipadd
do
 sleep 60;
done
ssh $ipadd

$./until-ping-ssh.sh
Enter IP Address:192.143.2.10
PING 192.143.2.10 (192.143.2.10) 56(84) bytes of data.
--- 192.143.2.10 ping statistics ---
```

```
1 packets transmitted, 0 received, 100% packet loss, time
0ms

PING 192.143.2.10 (192.143.2.10) 56(84) bytes of data.

64 bytes from 192.143.2.10: icmp_seq=1 ttl=64 time=0.059
ms

--- 192.143.2.10 ping statistics ---

1 packets transmitted, 1 received, 0% packet loss, time
0ms

rtt min/avg/max/mdev = 0.059/0.059/0.059/0.000 ms

ramesh@192.143.2.10's password:
```

The bash until loop is quite useful at the command line as a way of waiting for certain events to occur.

40. Case Command

Syntax:

```
case var in
 pattern1 )
   command1
   command2
   ...
   ;;
 pattern2 )
   command3
   command4
   ...
   ;;
esac
```

- If the value of the "var" matches "pattern1", it will execute command1, command2, and any other commands in the "pattern1" block.

- If the value of the "var" matches "pattern2", it will execute command3, command4, and any other commands in the "pattern2" block.
- The case statement ends with "esac", which is nothing but the reverse of the word case.
- Each set of commands should end with two semi-colons (;;).

This simple case example sets variables state and capital based on the argument given on the command line (which should be the state abbreviation for either California or Nevada).

```
$ cat case1.sh
stateabbr=${1}
case $stateabbr in
 CA )
  state="California"
  capital="Sacramento"
  ;;
 NV )
  state="Nevada"
  capital="Carson City"
  ;;
esac
echo "$state Capital is $capital"

$ ./case1.sh CA
California Capital is Sacramento
```

In the above example, when the value of stateabbr is "CA", bash executes the two commands that are under the pattern "CA". It skips over the other cases after that, continuing after the ‘esac’ statement.

Case is often used to create a menu for user selection:

```
$ cat case2.sh
```

```
echo "MAIN MENU"
echo "---------"
echo "a. Backup Log files"
echo "b. Backup Config files"
echo "c. Backup Home Directory"
echo "d. Exit"
echo "Enter your choice:"
read choice
case $choice in
a | A ) echo "Backing up log files..";;
b | B ) echo "Backing up config files..";;
c | C ) echo "Backing up home directory..";;
d | D ) exit;;
    * ) echo "Invalid choice!";;
esac
```

In the above example, the "pattern" doesn't just contain one value. It has multiple values. For example "**a | A)**", indicates that irrespective of whether users enters a or A, the commands in this pattern will be executed.

The last pattern in the above list is "***)**", which matches any character. When you get to this point, it means that none of the patterns until this point is matched. So, this should be an invalid value entered by the user.

41. New Case Command Features: ;& and ;;&

As we already know, ;; is used as the last statement in the pattern block of the case statement, and bash doesn't attempt to test any further pattern match after it encounters the ;; (i.e After executing ;; bash terminates the case statement).

Instead of double semi-colon (;;), you can also use ;& or ;;&. The following table explains the three case terminators.

Case Terminator	Description
;;	After executing the statements in the current pattern block, bash terminates the case statement and exits out of it.
;&	After executing the statements in the current pattern block, bash continues jumps to the statements in the next pattern block and executes them (without testing the pattern condition in the next pattern block), then proceeds based on the next pattern block’s terminator.
;;&	After executing the statements in the current pattern block, bash continues to the next pattern block and tests the rest of the pattern conditions; if a pattern matches, bash executes the statements in the associated pattern block.

Case Statement Example using ;;

In the example below, the other cases are skipped; only the CA case is executed.

```
$ cat case3.sh
stateabbr=${1}
case $stateabbr in
 NY )
  echo "NY is New York"
  ;;
 CA )
  echo "CA is California"
  ;;
 NV )
  echo "NV is Nevada"
  ;;
 *  )
```

```
  echo "Unknown State Name"
  ;;
esac

$ ./case3.sh CA
CA is California
```

Case Statement Example using ;&

In the example below, since ;& is used to terminate the CA pattern block, NV is printed too. But since ;; is used to terminate the NV block, the rest of the cases after NV are skipped.

```
$ cat case4.sh
stateabbr=${1}
case $stateabbr in
 NY )
  echo "NY is New York"
  ;&
 CA )
  echo "CA is California"
  ;&
 NV )
  echo "NV is Nevada"
  ;;
 *  )
  echo "Unknown State Name"
  ;;
esac

$ ./case4.sh CA
CA is California
NV is Nevada
```

Case Statement Example using ;;&

In the example below, since ;;& is used to terminate the CA pattern block, the NV case is tested, but it fails. So, the * case is tested, and since it always succeeds “Unknown State Name” is printed.

```
$ cat case5.sh
stateabbr=${1}
case $stateabbr in
 NY )
  echo "NY is New York"
  ;;&
 CA )
  echo "CA is California"
  ;;&
 NV )
  echo "NV is Nevada"
  ;;
 *  )
  echo "Unknown State Name"
  ;;
esac

$ ./case5.sh CA
CA is California
Unknown State Name
```

42. Select Command

```
select var in list
do
 command1
 command2
```

```
done
```

Using select, we can quickly generate menu items:

```
$ cat select.sh
select state in CA NV NY
do
 echo "Your selection is: $state"
done

$ ./select.sh
1) CA
2) NV
3) NY
#? 1
Your selection is: CA
#? 2
Your selection is: NV
#?
```

There are some problems with the above script:

- Once you make a selection, it keeps looping, and doesn't exit. (You have to press Ctrl-C to exit the loop).
- The prompt displays the ugly "#?".

Following is a better version of the same script:

```
$ cat select1.sh
PS3="Enter your choice: "
select state in CA NV NY
do
 echo "Your selection is: $state"
 break
done

$ ./select1.sh
1) CA
2) NV
3) NY
Enter your choice: 1
Your selection is: CA
```

Note: The environment variable PS3 is used to set the display prompt for case statements.

When you have a menu item that contains spaces in it, you should use double quotes to enclose the individual items in the list of values.

In the following example, if you don't enclose "New York" in double quotes, New and York will be treated as two separate menu items, not what we want.

```
$ cat select1-space.sh
PS3="Enter your choice: "
select state in "California" "Nevada" "New York" "Exit"
do
 echo "Your selection is: $state"
 break
done

$ ./select1-space.sh
1) California
2) Nevada
3) New York
4) Exit
Enter your choice: 3
```

```
Your selection is: New York
```

Instead of giving the list of values directly after the "in" keyword, you can also use a variable that contains the list, as shown in the example below.

```
$ cat select2.sh
PS3="Enter your choice: "
menuitems="CA NV NY Exit"
select state in $menuitems
do
 echo "Your selection is: $state"
 break
done
```

Caution: Make sure you don't enclose the variable that you specify after the "in" keyword in double quotes, as the value will be treated as single word.

```
$ cat select2-wrong.sh
PS3="Enter your choice: "
menuitems="CA NV NY Exit"
select state in "$menuitems"
do
 echo "Your selection is: $state"
 break
done

$ ./select2-wrong.sh
1) CA NV NY Exit
Enter your choice: 1
Your selection is: CA NV NY Exit
```

You can also omit the keyword "in" and the list of values. In that case, the select command will use the positional parameters as list of values as shown in the example below.

```
$ cat select3.sh
PS3="Enter your choice: "
select state
do
 echo "Your selection is: $state"
 break
done

$ ./select3.sh CA NV NY Exit
1) CA
2) NV
3) NY
4) Exit
```

Chapter 5. Additional Bash Built-in Commands

Bash builtins are commands that are built as part of bash itself. Certain builtins are available only in bash and are not available in other shells or as an external command. However, some bash builtins replicate an existing shell commands. For example, the echo bash builtin behaves similarly to the /bin/echo external program. When you use echo inside a shell script, bash will be using the builtin command which is faster than creating a new process to call the external /bin/echo command.

43. List of bash builtins

The following table contains list of available shell builtins. Some of them are explained in separate hack with examples. To know more about a specific builtin use the help builtin command. For example, "help test" will give you information about the test builtin.

The following are various builtins available in the bash shell.

Builtin	Description
:	Colon is null command. It does nothing.
.	Period executes the given file
break	Exit from a loop
cd	Change working directory
continue	Continue a loop
eval	Execute commands
exec	Replace current shell with the given command
exit	Exit from the shell
export	Export a variable to the environment
getopts	Manipulate positional parameters

Builtin	Description
hash	List of recently executed commands with the full path
pwd	Print current working directory
readonly	Set the variable as read only
return	Return from a function with a specified exit value
shift	Move the positional parameters to the left
test [	Evaluate a conditional expression
times	Display user and system times used by shell and its children
trap	Execute commands when a signal is trapped
umask	Set file creation mask
unset	Remove the variable or function
alias	Manipulate command aliases
bind	Manipulate readline key bindings and variables
builtin	Execute a shell builtin
caller	Return the context of the current subroutine call
command	Execute the given command (ignores shell function with the same name)
declare	Declare a variable
echo	Display to stdout
enable	Enable or disable builtin commands
help	Display help about a specific builtin
let	Evaluate arithmetic expression

Builtin	Description
local	Create a local variable
logout	Exit from a login shell
mapfile	Read values from stdin and store in an array
printf	Display formatted output
read	Read values and assign it to variable
readarray	Same as mapfile builtin
source	Same as . (period) builtin
type	Identify how a command will be treated
typset	Deprecated. Use declare instead.
ulimit	Control resources allocated to a process
unalias	Remove an alias

44. Help Builtin

Get quick help about bash builtin commands. For example:

```
help set
help test
help if
help for
```

One line help description:

```
$ help -d if
if - Execute commands based on conditional.
```

One line usage synopsis of the command:

```
$ help -s if
if: if COMMANDS; then COMMANDS; [ elif COMMANDS; then
COMMANDS; ]... [ else COMMANDS; ] fi
```

Display builtin help similar to man

```
$ help -m if
```

45. Echo Builtin

The echo builtin command supports –n and –e options to suppress newline or insert a special character. The following table explains some of the special character codes that you can use with –e:

Code	Description
\a	Alert with the bell sound
\b	Backspace
\c	Further output is suppressed
\e	Escape
\f	Formfeed
\n	Newline
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
\\	Backslash
\0nnn	Represents octal value nnn
\xHH	Represents hexadecimal value HH

To print special characters use echo –e, to suppress newline use echo –n. Examples are on the next page.

It is very helpful to suppress newline when you have multiple echo commands inside a shell script, but you would like to see the output of those echo commands on a single line (instead of each echo command printing the output on a different line).

In the following example, when you use -n, you can notice that the bash prompt $ is printed on the same line as the output, as there is no newline printed after the text.

```
$ echo "Hello World"
Hello World
$ echo -n "Hello World"
Hello World$
```

In the following example, when you don't use -e, the special character for tab (\t) is printed as \t. When you use echo -e, it is treated as a tab character.

```
$ echo "hello\tworld"
hello\tworld
$ echo -e "hello\tworld"
helloworld
```

46. Color Your echo Output

You can color the text that is used in the echo statement as shown in the example below.

```
$ cat color.sh
# Define the color codes
BLUE="\033[0;34m"
DARK_RED="\033[1;31m"
GREEN="\033[0;32m"
NO_COLOR="\033[0m"
```

```
shell="Bash"
echo -e "The best shell is: $shell"
echo -e "The best shell is: $BLUE$shell$NO_COLOR"
echo -e "The best shell is: $DARK_RED$shell$NO_COLOR"
echo -e "The best shell is: $GREEN$shell$NO_COLOR"
```

Use the following color code table for using additional colors.

Color	Code
Black	**\033[0;30**
Blue	\033[0;34
Green	033[0;32
Cyan	033[0;36
Red	\033[0;31
Purple	\033[0;35
Brown	\033[0;33

Note: Replace [0 with [1 in the above table for dark color. For example, for dark blue, use \033[1;34

In addition to using the color codes, you can do additional text manipulation using tput. For example, bold or underline the text:

```
$ cat tput.sh
shell="Bash"
echo -e "The best shell is [normal]: $shell"
echo -e "The best shell is [bold]: `tput bold`$shell`tput
sgr0`"
echo -e "The best shell is [background]: `tput setb
4`$shell`tput setb 0`"
echo -e "The best shell is [underline]: `tput
smul`$shell`tput rmul`"
```

47. Printf Builtin

Bash has a formatted print command printf. These tables have codes for the format; usage will be described on the next page.

Format Identifier	Description
%s	String
%d	Decimal
%x	Hexadecimal
%o	Octal
%f	Floating point
%e	Double
%c	Prints only one character

Format Modifier	Description
Number	Minimum field width
- (hyphen)	Left align
0 (zero)	Numbers are padded with zero
+ (plus)	A + or - sign is printed
.number	Exact field width
*	Give number as argument

In the bash printf builtin, ‘format’ is a double quoted string with a mixture of literal text and ‘format identifiers’ (modified by ‘format modifiers’) that will be replaced with the ‘arguments’ following the format string:

```
printf format arguments
```

The following example prints a string in a field that is at least 5 characters wide. When the supplied string value is less than 5 characters, bash will add leading spaces:

```
$ printf "%5s\n" "123"
  123
```

When the value is more than 5 characters wide, it will print the full string:

```
$ printf "%5s\n" "123456"
123456
```

If you want an exact fixed width, even when the printed value contains more characters, use a . (period) along with the number. For example, %.5s will give you exactly 5 characters, truncating the rest of the characters:

```
$ printf "%.5s\n" "123456"
12345
```

You can also use a * in the printf format modifier, instead of including the field width or precision in the format string. You then pass the modifying number as an argument. Both of the following examples do exactly the same.

```
printf "%5s\n" "123"
printf "%*s\n" 5 "123"
```

Use a – (hyphen) to pad spaces on the right side of the value (instead of the default left side). This is helpful when printing a record with multiple fields.

```
$ printf "%-5s %s\n" "123" "ramesh"
123   ramesh
```

When you use 0 along with a numeric format identifier, printf adds leading 0s to the printed numbers, instead of spaces. The following example prints “00123” instead of “123” since we've specified “%05d” as the identifier.

```
$ printf "%05d %s\n" "123" "ramesh"
00123 ramesh
```

Use a + in the format modifier, to print the number along with its sign.

```
$ printf "%+05d %-05d %s\n" 123 -90 "ramesh"
+0123 -90   ramesh
```

All of the above are shown in the following example.

```
$ cat printf.sh
echo -e "\n1. Print Fixed Width (space before):"
printf "%5s %s\n" "123" "ramesh"
printf "%5s %s\n" "1234" "john"
printf "%5s %s\n" "12345" "jason"
printf "%5s %s\n" "123456" "preeti"
echo -e "\n2. Print Exact Width (truncate characters):"
printf "%.5s\n" "123456"
echo -e "\n3. Print Fixed Width (using argument):"
printf "%*s\n" 5 "123"
echo -e "\n4. Print Fixed Width (space after):"
printf "%-5s %s\n" "123" "ramesh"
printf "%-5s %s\n" "1234" "john"
printf "%-5s %s\n" "12345" "jason"
echo -e "\n5. Print Numbers:"
printf "%20s %5d\n" "Decimal" 123
printf "%20s %+5d\n" "Decimal (with sign)" 123
printf "%20s %+5d\n" "Decimal (with sign)" -90
printf "%20s %5o\n" "Octal" 80
printf "%20s %5x\n" "Hexadecimal" 1298
echo -e "\n6. Print Fixed Width Numbers (zeros before):"
printf "%05d %s\n" "123" "ramesh"
printf "%05d %s\n" "1234" "john"
printf "%05d %s\n" "12345" "jason"
```

48. Read Builtin

The bash read command is used to get input from the user interactively (or even non-interactively, from a file) and store it in a variable.

You can read one value, or multiple values (space, comma, or tab separated) in the same read statement. When you enter multiple values, the 1st word goes to the 1st variable, 2nd word to 2nd variable, etc. When there are more values than variables, the last variable will hold the remaining words; when there are more variables than values, the additional variables will have null.

Here are some of the most frequently used variations of the read command:

```
$ cat read.sh
echo -n "Enter your nick name: "
read name
echo "Good morning, $name"
echo -n "Enter your first name and last name: "
read fname lname
echo "Good morning, $fname $lname"
echo -n "Enter your first name, last name, and something
else: "
read fname lname remaining
echo "Good morning, $fname $lname. You said: $remaining"
echo -n "Enter anything: "
read -a arr
```

```
echo "Array values: $arr"
read -p "Enter your name: " name1
echo $name1
read -s -p "Enter your password: " password
echo "Your password is: $password"
read -t 5 -p "Enter the value within 5 seconds: " timeout
echo "You entered: $timeout"
read -N 5 -p "Read will read exactly 5 character: "
charact
echo "You entered: $charact"
read -r -p "Enter a value with back slash: " good
echo $good
```

Key points to remember:

- read -a arrayname : The values that are read are assigned to the array starting from 0.
- read -p prompt : Displays the prompt before reading the value. This is easier and quicker than giving a echo statement.
- read -s : This doesn't echo the value as you enter. This is useful for reading any sensitive information from the user.
- read -t 10 : If the user doesn't enter anything, read waits for 10 seconds and times out.
- read -N 5 : The read command returns after you've entered exactly 5 characters, no matter how many variables are being assigned. When you try to enter more than 5, read will not accept them; it will automatically stop at the 5th character input, and continue with the rest of the shell script logic.
- read -r: In a read command, if you want to enter the values in multiple lines, you can use \ while entering the values. However, if you want read to treat the \ literally (not as new line), you should use read –r.
- read without a variable name: If you don't specify a variable name, bash will store the value in the $REPLY bash internal variable.

You can also read value(s) from a file, as shown below.

```
$ cat test.txt
this is good

$ read var <test.txt

$ echo $var
this is good
```

49. Getopts Builtin

Use getopts in a shell script to get command line options typed when calling the script. You can pass options with or without any arguments.

Please note that getopt is not the same as getopts. getopt is an external program located under /usr/bin and getopts is a bash builtin. However, the getopts builtin is functionally similar to the getopt external program.

Here is a simple usage example of passing options to a shell script:

```
script -c -v -f /tmp/1.sh -z
```

In the above example, there are 4 options (c, v, f and z). Option names have to be only a single character. Option f also has an argument: /tmp/1.sh

You can combine multiple options together. For example, the c, v, and z options can be combined as “cvf” as shown below.

```
script -cvz -f /tmp/1.sh
```

In the shell script, the getopts command will use the following syntax:

```
getopts "optionstring" var [arguments]
```

- "optionstring" contains a list of all the options that the script might expect. For the above example, the optionstring would be "cvf:z". The colon after option f indicates that option f will have an argument.
- var – The variable where the option string will be stored.
- arguments – The given arguments are processed instead of the positional parameters. This is optional.

For getopts to work properly, each single character option passed to the script should have - as the prefix. When bash it encounters an argument that doesn't have hyphen as a prefix, it will terminate getopts parsing immediately.

Typically you would use getopts along with “while” and “case” statements as shown in the example below.

```
$ cat getopts.sh
while getopts ":cvf:z" Option
do
  case $Option in
    c ) echo -n "Option c is used. "
        echo "OPTIND=$OPTIND"
        ;;
    v ) echo -n "Option v is used. "
        echo "OPTIND=$OPTIND"
        ;;
    f ) echo -n "Option f is used. "
        echo "OPTIND=$OPTIND OPTARG=$OPTARG"
        ;;
    z ) echo -n "Option z is used. "
        echo "OPTIND=$OPTIND"
        ;;
  esac
done
```

```
shift $(($OPTIND - 1))
```

Execute the above getopts example shell script to view the output.

```
$ ./getopts.sh -c -v -f /tmp/1.sh -z
Option c is used. OPTIND=2
Option v is used. OPTIND=3
Option f is used. OPTIND=5 OPTARG=/tmp/1.sh
Option z is used. OPTIND=6
```

In the following example, the 2nd option "v" is given without the hyphen. So, getopts processes only the 1st option (-c) and stops.

```
$ ./getopts.sh -c v -f /tmp/1.sh -z
Option c is used. OPTIND=2
```

The first colon in the optionstring is optional. In the above example, you can also specify the optionstring as shown below.

```
while getopts "cvf:z" Option
```

But, when you pass an argument to the script that doesn't exist in the optionstring, it will throw error message. So, using a colon in front will not display that error message.

- $OPTIND points to the current option it is processing, and the
- $OPTARG contains the argument for that index. Both $OPTIND and
- $OPTARG are bash internal variables.

50. Hash Builtin

The bash shell has a hash table where the commands you execute are stored. The full path is included in the hash table.

When you execute the same command next time, bash won't search the PATH variable, instead it will pick up the command from the hash table and run it.

This is demonstrated in the sequence of examples shown below. The hash command without any argument displays the content of the hash table. In this example, it is empty, as we just started the session.

```
$ hash
hash: hash table empty
```

Execute a command, for example, ls.

```
$ ls
```

Now the hash command will display the hash table that contains two fields:

1. Total number of hits for the command
2. The command itself with the full path.

The following output shows that /bin/ls has been executed 1 time so far, and it's the only command in the hash table now. Next time when you execute the ls command, bash will not search the PATH variable, it will just pick up the full-path of the command from the hash table and execute it.

```
$ hash
hits    command
   1    /bin/ls
```

Execute some other command, for example, cat.

```
$ cat /etc/passwd
```

View the hash table again. Now, you'll see two entries.

```
$ hash
hits    command
   1    /bin/cat
   1    /bin/ls
```

Execute the ls command again, a couple of times.

```
$ ls
$ ls -l
```

View the hash table again. You'll still see two entries, but the ls command hits counter will show 3, as we executed ls 3 times so far in this session.

```
$ hash
hits    command
   1    /bin/cat
   3    /bin/ls
```

You can also add a command to the hash table without executing it. The following will add the tar command to the hash table.

```
$ hash tar
```

Since the tar command was added to the hash table without executing it, the hits will be 0 for the tar command as shown below.

```
$ hash
hits    command
   1    /bin/cat
   0    /bin/tar
   3    /bin/ls
```

51. pwd - Print Working Directory

As you already know, pwd is a simple command to print the current directory.

The one option that is useful is pwd -P, which prints the real directory structure (without symbolic links). *Note: it's a capital P!*

As you see below, /etc/rc0.d is a symbolic link to /etc/rc.d/rc0.d

```
$ ls -ld /etc/rc0.d
lrwxrwxrwx. root Aug 6 /etc/rc0.d -> rc.d/rc0.d/
```

Now, use the symbolic link to cd to that directory, and do a pwd. This will show only the symbolic link directory that you used to cd into it.

```
$ cd /etc/rc0.d

$ pwd
/etc/rc0.d
```

If you really want to know the original directory where you are currently located, you should use pwd -P, as shown below. In this example, pwd -P will display /etc/rc.d/rc0.d (not the /etc/rc0.d that was shown by pwd).

```
$ cd /etc/rc0.d

$ pwd -P
/etc/rc.d/rc0.d
```

52. Commands to Manipulate Builtins

builtin

The “builtin” command is used to explicitly call bash builtins.

Let us assume that you want to add "===>" in front of the pwd output. You can create a function called mypwd and add it to the .bash_profile to achieve this.

```
$ vi ~/.bash_profile
function mypwd () {
  echo -n "===>"
  pwd
}

$ source ~/.bash_profile

$ mypwd
===>/home/ramesh
```

Now, let us assume that you want to override the default pwd, and call your custom function anytime you execute pwd. Of course you can rename the mypwd function in your .bash_profile to pwd, but there is a problem with that approach. If you rename the custom mypwd function to pwd, then when you execute the pwd command, it will go into an infinite loop because it is calling itself. It will forever keep printing "===>".

```
$ vi ~/.bash_profile
function pwd () {
  echo -n "===>"
  pwd
}
```

So, when you are referring to a bash builtin from a custom function that has the same name as the builtin, you should explicitly call the original bash builtin using the "builtin" command.

Here’s an example demonstrating the use of the “builtin” command.

```
$ vi ~/.bash_profile
function pwd () {
  echo -n "===>"
  builtin pwd
}

$ source ~/.bash_profile

$ pwd
===>/home/ramesh
```

enable

You can enable or disable a bash builtin using enable.

By default all the builtins are enabled. For example, when you type alias, it will execute the “alias” builtin and display the output.

To disable the bash alias builtin, use enable with the -n option. Once you disable a builtin, when you try to execute it, bash will display “command not found” as shown below.

```
$ alias
$ enable -n alias
$ alias
-bash: alias: command not found
```

To enable the alias builtin again, use enable without the ‘-n’ option.

```
$ enable alias
$ alias
```

To view all the enabled shell builtins, execute 'enable -p'.

53. Type Builtin

A command that you execute from the command line can be an alias, a shell builtin, a shell keyword, or an external program. Use the type command to identify the type of a particular command.

The following example shows that “ls” is an alias, “pwd” is a builtin, “if” is a shell keyword, and “file” is an external program.

```
$ type ls
ls is aliased to `ls --color=auto'

$ type pwd
pwd is a shell builtin

$ type if
if is a shell keyword

$ type file
file is /usr/bin/file
```

If a command is available both as builtin and external program, and you would like to view both, you can use "type -a" as shown below.

```
$ type -a pwd
pwd is a shell builtin
pwd is /bin/pwd

$ type -a ls
```

```
ls is aliased to `ls --color=auto'
ls is /bin/ls
```

54. Ulimit Builtin

Using ulimit you can control the amount of resources that can be assigned to processes that are started by the bash shell.

Syntax:

```
ulimit [-SHacdefilmnpqrstuvx] [limit]
```

To view all the current limits, use ulimit -a, as shown below. From this output, you can see that the total number of "open files" allowed for a process that is created by the shell is 1024, the "max memory size" available for a process that is created by the shell is "unlimited", etc.

```
$ ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
scheduling priority             (-e) 0
file size               (blocks, -f) unlimited
pending signals                 (-i) 63822
max locked memory       (kbytes, -l) 64
max memory size         (kbytes, -m) unlimited
open files                      (-n) 1024
pipe size            (512 bytes, -p) 8
POSIX message queues     (bytes, -q) 819200
real-time priority              (-r) 0
stack size              (kbytes, -s) 8192
cpu time               (seconds, -t) unlimited
max user processes              (-u) 1024
virtual memory          (kbytes, -v) unlimited
```

```
file locks                            (-x) unlimited
```

You can view a specific resource using the short option for that resource. The short option for a particular resource is displayed in the above “ulimit -a” output within ( ). For example, the short option for open files is –n:

```
$ ulimit -n
1024
$ ulimit -m
unlimited
$ ulimit -i
63822
```

To change the value of a specific resource, specify the new value as an argument to the ulimit command. The following example changes the number of "open files" from the default “1024” to “2048”.

```
$ ulimit -n 2048

$ ulimit -n
2048

$ ulimit -a
core file size          (blocks, -c) 0
data seg size           (kbytes, -d) unlimited
scheduling priority             (-e) 0
file size               (blocks, -f) unlimited
pending signals                 (-i) 70000
max locked memory       (kbytes, -l) 64
max memory size         (kbytes, -m) unlimited
open files                      (-n) 1024
pipe size            (512 bytes, -p) 8
POSIX message queues     (bytes, -q) 819200
```

```
real-time priority              (-r) 0
stack size              (kbytes, -s) 8192
cpu time               (seconds, -t) unlimited
max user processes              (-u) 1024
virtual memory          (kbytes, -v) unlimited
file locks                      (-x) unlimited
```

Use S or H to an option to display the soft or Hard limit, respectively.

View the soft limits of all the resources:

```
ulimit -aS
```

View the hard limits of all the resources:

```
ulimit -aH
```

55. dirs, pushd, and popd

Use dirs, pushd and popd to manipulate the bash directory stack. You can use the directory stack to save and restore your current working directory by pushing and popping, respectively.

- dirs: Display the directory stack
- pushd: Push directory onto the stack
- popd: Pop directory from the stack and cd to it

Dirs will always print the current directory followed by the content of the stack. Even when the directory stack is empty, the dirs command will still print the current directory. *Remember: ~ stands for your home directory.*

```
$ popd
-bash: popd: directory stack empty

$ dirs
```

```
~


$ pwd
/home/ramesh
```

How to use pushd and popd?

Let us first create some temporary directories.

```
mkdir /tmp/dir1
mkdir /tmp/dir2
mkdir /tmp/dir3
mkdir /tmp/dir4
```

Push these temporary directories onto the directory stack as shown below.

```
cd /tmp/dir1
pushd .
cd /tmp/dir2
pushd .
cd /tmp/dir3
pushd .
cd /tmp/dir4
pushd .
```

After pushing the above 4 directories to the stack, let us view the content of the directory stack using the dirs command.

```
$ dirs
/tmp/dir4 /tmp/dir4 /tmp/dir3 /tmp/dir2 /tmp/dir1
```

Note: The first directory (/tmp/dir4) of the dirs command output is always the current directory, not the content from the stack.

At this stage, the directory stack contains the following directories:

```
/tmp/dir4
/tmp/dir3
/tmp/dir2
/tmp/dir1
```

The last directory that was pushed to the stack will be at the top. When you perform popd, bash will cd to the top directory entry in the stack and remove it from the stack. As shown above, the last directory that we pushed onto the stack is /tmp/dir4. So, when we do a popd, we will switch to /tmp/dir4 and remove it from the directory stack as shown below.

```
$ popd
$ pwd
/tmp/dir4
```

After the above popd, the directory stack contains the following directories:

```
/tmp/dir3
/tmp/dir2
/tmp/dir1
```

Now, do a popd to get pop another directory from the stack and switch to it.

```
$ popd
$ pwd
/tmp/dir3
```

After that popd, the directory stack contains the following directories:

```
/tmp/dir2
```

```
/tmp/dir1
```

Now, do another popd to get pop a directory from the stack and switch to it.

```
$ popd
$ pwd
/tmp/dir2
```

Now, do another popd to get pop a directory from the stack and switch to it.

```
$ popd
$ pwd
/tmp/dir1
```

After this popd, the directory stack is empty!

```
$ popd
-bash: popd: directory stack empty
```

Dirs

By default, dirs command prints the content of the directory stack in a single line as shown below.

```
$ dirs
~ /tmp/dir4 /tmp/dir3 /tmp/dir2 /tmp/dir1
```

Use dirs -p to print all the entries in the directory stack in a separate line:

```
$ dirs -p
~
/tmp/dir4
```

```
/tmp/dir3
/tmp/dir2
/tmp/dir1
```

Use dirs -l to show the full path of each entry:

```
$ dirs -l
/home/ramesh/.ssh /tmp/dir4 /tmp/dir3 /tmp/dir2 /tmp/dir1
```

Use dirs -c to delete all the entries on the directory stack.

```
$ dirs -c
```

Use dirs -v to print all the entries in the directory stack, each on a separate line along with its position in the stack:

```
$ dirs -v
 0  ~
 1  /tmp/dir4
 2  /tmp/dir3
 3  /tmp/dir2
 4  /tmp/dir1
```

Use dirs +n to display the nth element from the directory stack. For example, to display the 3rd element, do the following:

```
$ dirs +3
/tmp/dir2
```

Use dirs -n to display the nth element (counted in reverse order) from the directory stack. For example, to display the 3rd element from the bottom, do the following:

```
$ dirs -3
```

```
/tmp/dir4
```

Note: when bash is counting from the bottom, the last element in the list is treated as the 0th element.

pushd

By default when you execute the pushd command, in addition to putting that directory onto the directory stack, bash also does a cd to that directory.

Using the pushd -n option, you can add the directory to the stack without switching to it.

Push home directory onto the stack but don’t change your working directory:

```
$ cd ~
$ pushd -n /etc/
$ pwd
/home/ramesh
```

You can also use +n or –n with pushd to rotate the directory stack upward or downward, respectively.

Let us assume the directory stack contains the following list.

```
$ dirs -v
 0  /tmp/dir4
 1  /tmp/dir4
 2  /tmp/dir3
 3  /tmp/dir2
 4  /tmp/dir1
```

Using pushd +2, you can rotate the stack upward, making the 2nd element of the list (/tmp/dir3) the new top of the stack, as shown below.

```
$ pushd +2
/tmp/dir3 /tmp/dir2 /tmp/dir1 /tmp/dir4 /tmp/dir4

$ dirs -v
 0  /tmp/dir3
 1  /tmp/dir2
 2  /tmp/dir1
 3  /tmp/dir4
 4  /tmp/dir4
```

Using pushd -2, you can rotate the stack downward, making the 2nd element from the end of the list the new top of the stack.

Popd

By default when you execute the popd command, bash will pick the top entry from the directory stack, cd to it, and delete that entry from the stack.

Let us assume the directory stack contains the following items, and the current directory is the home directory.

```
$ pwd
/home/ramesh

$ dirs -v
 0  /tmp/dir4
 1  /tmp/dir4
 2  /tmp/dir3
 3  /tmp/dir2
 4  /tmp/dir1
```

Using popd -n, you can delete the top entry from the stack without switching to that directory.

```
$ popd -n
/tmp/dir4 /tmp/dir3 /tmp/dir2 /tmp/dir1

$ dirs -v
 0  /tmp/dir4
 1  /tmp/dir3
 2  /tmp/dir2
 3  /tmp/dir1

$ pwd
/home/ramesh
```

Using popd +2, you can delete the 2nd directory from the top of the list (/tmp/dir3 in this example). Bash will not switch to that directory.

```
$ dirs -v
 0  /tmp/dir4
 1  /tmp/dir4
 2  /tmp/dir3
 3  /tmp/dir2
 4  /tmp/dir1

$ popd +2
/tmp/dir4 /tmp/dir4 /tmp/dir2 /tmp/dir1

$ dirs -v
 0  /tmp/dir4
 1  /tmp/dir4
 2  /tmp/dir2
 3  /tmp/dir1
```

Using popd -2, you can delete the 2nd directory from the bottom of the list (/tmp/dir4 in this example). Bash will not switch to that directory.

Please note that when you use -, the last item is counted as 0.

```
$ dirs -v
 0  /tmp/dir4
 1  /tmp/dir4
 2  /tmp/dir2
 3  /tmp/dir1

$ popd -2
/tmp/dir4 /tmp/dir2 /tmp/dir1

$ dirs -v
 0  /tmp/dir4
 1  /tmp/dir2
 2  /tmp/dir1
```

56. Shopt Builtin

shopt stands for shell options.

To view all the shell options and their current status (whether the option is enabled or disabled), do the following:

```
$ shopt
autocd          off
cdable_vars     off
cdspell         off
checkhash       off
checkjobs       off
checkwinsize    on
..
```

To view the current status of a specific shell option (for example, cdspell), do the following:

```
$ shopt cdspell
cdspell         off
```

Use shopt -s, to enable (set) a particular shell option as shown below.

```
$ shopt -s cdspell


$ shopt cdspell
cdspell         on
```

Use shopt -u, to disable (unset) a particular shell option as shown below.

```
$ shopt -u checkwinsize


$ shopt checkwinsize
checkwinsize    off
```

To view all the available shell options and their current status using the set/unset syntax (shopt -s or shopt –u), use shopt –p. This is helpful when you want to take the shell options from one system replicate them on some other system. Just copy the output of shopt -p on one system, and paste it in to the other system, or redirect the shopt –p output to a file that you execute on the other system.

shopt –p example:

```
$ shopt -p
shopt -u autocd
shopt -u cdable_vars
shopt -s cdspell
shopt -u checkhash
shopt -u checkjobs
shopt -u checkwinsize
..
```

Some shopt options can also be controlled using the set builtin command. shopt -o displays all those shell options that can also be controlled using the set command.

```
$ shopt -o
allexport         off
braceexpand       on
emacs             off
errexit           off
errtrace          off
functrace         off
hashall           on
..
```

The following table displays all shopt options and what they mean. By default, all of these options are off (except those that are in bold).

Shopt Option	Description
autocd	Use only the directory name to cd to a directory. i.e no need to type 'cd /etc', just type '/etc'.
cdable_vars	Treat the argument to the cd command as a variable. i.e 'junk=/tmp', 'cd junk' will change to /tmp directory.
cdspell	Minor spelling mistakes are corrected automatically. i.e 'cd /temp' will automatically do 'cd /tmp'

Shopt Option	**Description**
checkhash	Checks whether a command is available in the hash table first (before checking the $PATH variable)
checkjobs	When jobs are running, first exit command will list all running jobs, second exit command will exit the shell.
checkwinsize	Check windows sizes and update LINES and COLUMNS variable accordingly.
cmdhist	Multiple line command are stored in the same history entry
compat31	Bash version 3.1 compatibility mode
compat32	Bash version 3.2 compatibility mode
compat40	Bash version 4.0 compatibility mode
dirspell	Minor spelling corrections will be performed for directory name for glob matching
dotglob	Path name expansion will include files starting with a . (dot)
execfail	Shell will not exit when exec command fails
expand_aliases	Aliases are expanded. Default value for this is on for interactive shell, and off for non-interactive shell.
extdebug	Enable extended debugging option
extglob	Enable extended pattern matching
extquote	For parameter expansions $'var' and $"var" quoting are performed

Shopt Option	**Description**
failglob	Display error message for failed file glob matching
force_fignore	Ignore the suffixes available in the FIGNORE shell variable during word completion.
globstar	Enable Recursive globbing (* *)
gnu_errfmt	Display error message in the GNU error message format
histappend	History list is appended instead of getting overwritten to the history file.
histreedit	Re-edit a failed history substitution
histverify	Verify the history substitution by loading it in an editor, instead of directly executing it.
hostcomplete	Bash completion also completes hostnames
huponexit	Send SIGHUP to all jobs when exiting an interactive shell
interactive_comments	Ability to use # for comments in interactive shell
lithist	lithist+cmdhist option saves multiline commands to history with embedded newlines as appropriate
login_shell	Value cannot be changed. This option is set when bash is a login shell.
mailwarn	Displays the message that the mailfile has been checked, if it was checked since bash last checked it.

Shopt Option	Description
no_empty_cmd_completion	For empty command completion, don't attempt to search the directories in the PATH
nocaseglob	For pathname expansions, match filename by ignoring the case
nocasematch	When executing 'case' or '[[' command, match the pattern by ignoring the case
nullglob	Pattern that doesn't match anything expands to null
progcomp	Enable programmable completion
promptvars	Prompt strings will be subjected to parameter expansion, arithmetic expansion, command substitution, and quote removal.
restricted_shell	Value cannot be changed. Value is set to 'on' when bash runs in restricted mode.
shift_verbose	Displays error message when shift command has to shift more positional parameters than available
sourcepath	Source command will search through the directories in the PATH variable for the given filename.
xpg_echo	Echo command will expand the backslash escape characters

Chapter 6. Bash Array Manipulation

57. Declare and Assign Values to an Array

In bash, an array is automatically created when you assign a value to a variable in the following format:

```
name[index]=value
```

- name is any name for an array
- index could be any number or expression that must evaluate to a number greater than or equal to zero.

To access an element from an array use curly brackets: ${name[index]}.

The following example demonstrates array usage:

```
$ cat array-declare.sh
city[0]="Los Angeles"
city[1]="San Francisco"
city[5]="New York"
city[12]="Beverly Hills"
echo ${city[5]}

$ ./array-declare.sh
New York
```

You can also declare an array explicitly using declare -a arrayname.

```
declare -a linux
```

Note: An array doesn't need to have continuous indexes. Also, when you try to print an array element that is not initialized, you'll get a null value.

Initializing an array during declaration

Instead of initializing each element of an array separately, you can declare and initialize an array by specifying the list of elements (separated by white space) with in paranthesis.

```
declare -a arrayname=(element1 element2 element3)
```

Note: The array elements should be separated by a space. If an individual element has embedded white space, enclose the whole value within quotes.

Array example with initialization

```
$ cat array-initialize.sh
declare -a city=("Los Angeles" "San Francisco" "New York"
"Beverly Hills")
echo ${city[2]}
state=(CA NY NV)
echo ${state[2]}
```

Note that you can omit the "declare -a" while assigning values, as shown above for the state array.

When the array elements contains a space in the value, use double quotes, as shown in the city array above.

When you initialize an array, normally the array indexes start from 0, with an increment of 1, i.e. 0, 1, 2, 3, 4, etc. However if you include an index in [ ] brackets, bash will use that instead.

In the following example, array1 has indexes starting from 0 with an increment of 1. However array2 has user-defined non sequential indexes.

```
$ cat array-declare1.sh
array1=("one" "two" "three" "four")
echo ${array1[@]}
array2=([0]="one" [5]="two" [11]="three" [13]="four")
```

```
echo ${array2[@]}
```

58. Access Array Elements

As shown in the above examples, to access an individual array element, you should use the following syntax:

```
${array[index]}
```

The curly braces while accessing the array element are mandatory.

Print the Whole Bash Array

There are different ways to print all the elements of an array. If the index number is @ or *, all members of an array are referenced.

```
$ cat array-access.sh
declare -a city=("Los Angeles" "San Francisco" "New York"
"Beverly Hills")
echo "city[2] = ${city[2]}"
echo "city[@] = ${city[@]}"
echo "city[*] = ${city[*]}"

$ ./array-access.sh
city[2] = New York
city[@] = Los Angeles San Francisco New York Beverly Hills
city[*] = Los Angeles San Francisco New York Beverly Hills
```

The name of the array refers to the 1st element of the array, i.e. referencing “city” is same as referencing “city[0]”.

When you echo just the array variable (without the index), bash displays the 1st element as shown below.

```
$ array=("one" "two" "three" "four")
$ echo $array
```

```
one
```

When you assign a value to an array variable (without the index), it stores the value to the 1st array element as shown below.

```
$ echo ${array[@]}
one two three four

$ array="zero"
$ echo ${array[@]}
zero two three four
```

You can use patterns instead of the index when accessing an element of the array. To do this you specify a search string:

```
${array[@]/searchstring/}
```

Searchstring can be an exact match for the entire array element, or may have a trailing * to match elements starting with the string. For example /NV/ would match only an array element “NV” but /N*/ would match “NV” and “NY” as shown in this example:

```
$ declare -a state=(CA NV NY UT TX)
$ echo "state[@]/NV/ = ${state[@]/NV/}"
$ echo "state[@]/N*/ = ${state[@]/N*/}"
```

The full syntax of searches is covered in the section **Find and Replace using Parameter Expansion.**

59. Add an Array Element

If you know exactly what index you want when adding a value, just use the index number and assign the value as shown below.

```
city[4]="Miami"
```

However if you just want to append several values at the end of an array, use the method shown in the next example. "Miami" is added at location 4, but both "Boston" and "Las Vegas" are just appended to the end of the array, and we don't need to specify the location for these.

```
$ cat array-add-element.sh
declare -a city=("Los Angeles" "San Francisco" "New York"
"Beverly Hills")
city[4]="Miami"
echo "city[@] = ${city[@]}"
city=("${city[@]}" "Boston" "Las Vegas")
echo "city[@] = ${city[@]}"


$ ./array-add-element.sh
city[@] = Los Angeles San Francisco New York Beverly Hills
Miami
city[@] = Los Angeles San Francisco New York Beverly Hills
Miami Boston Las Vegas
```

60. Remove Array Elements

unset is used to remove an element from an array. unset will have the same effect as assigning null to an element.

```
$ cat array-remove-element.sh
declare -a state=(CA NV NY UT TX)
echo "state[@] before delete = ${state[@]}"
unset state[1]
echo "state[@] after delete = ${state[@]}"
echo "state[0] = ${state[0]}"
echo "state[1] = ${state[1]}"
echo "state[2] = ${state[2]}"
echo "state[3] = ${state[3]}"
echo "state[4] = ${state[4]}"
```

Note that unset doesn't really delete the array element, instead it just assigns null value to the array element. The other indices stay the same.

```
$ ./array-remove-element.sh
state[@] before delete = CA NV NY UT TX
state[@] after delete = CA NY UT TX
state[0] = CA
state[1] =
state[2] = NY
state[3] = UT
state[4] = TX
```

Use this workaround to delete an element from the array:

```
state=(${state[@]:0:$pos} ${state[@]:$(($pos + 1))})
```

When the value of the variable “pos” is set to 1,

- ${state[@]:0:1} will give you one element starting from index 0.
- ${state[@]:2) will give you all the elements starting from index 2.
- Concatenate these and assign the result back to the original array to eliminates position 1 from the array.

Here we show the statement in an actual example:

```
$ cat array-remove-element1.sh
declare -a state=(CA NV NY UT TX)
echo "state[@] before delete = ${state[@]}"
pos=1
state=(${state[@]:0:$pos} ${state[@]:$(($pos + 1))})
echo "state[@] after delete = ${state[@]}"
echo "state[0] = ${state[0]}"
echo "state[1] = ${state[1]}"
echo "state[2] = ${state[2]}"
echo "state[3] = ${state[3]}"

$ ./array-remove-element1.sh
state[@] before delete = CA NV NY UT TX
state[@] after delete = CA NY UT TX
state[0] = CA
state[1] = NY
state[2] = UT
state[3] = TX
```

Remove Bash Array Elements using Patterns

```
$ cat array-remove-element2.sh
declare -a state=(CA NV NY UT TX)
declare -a newstate=(${state[@]/NV*/} )
echo "newstate[@] after delete = ${newstate[@]}"
echo "newstate[0] = ${newstate[0]}"
echo "newstate[1] = ${newstate[1]}"
echo "newstate[2] = ${newstate[2]}"
echo "newstate[3] = ${newstate[3]}"

$ ./array-remove-element2.sh
newstate[@] after delete = CA NY UT TX
newstate[0] = CA
newstate[1] = NY
newstate[2] = UT
newstate[3] = TX
```

Deleting an Entire Array

unset is used to delete an entire array.

```
$ cat array-remove.sh
```

```
declare -a state=(CA NV NY UT TX)
echo "state array before: ${state[@]}"
unset state
echo "state array after: ${state[@]}"

$ ./array-remove.sh
state array before: CA NV NY UT TX
state array after:
```

61. Array Length

A hash # in front of the array name gives the length.

- ${#array[@]} or ${#array[*]} returns array length, i.e. the total number of elements in the array.
- ${#array[n]} returns the length of the nth element, i.e. the total number of characters of the nth element in the array.

```
$ cat array-length.sh
array=("one" "two" "three" "four")
echo "\${array[@]} = ${array[@]}"
echo "Total number of elements in the array = $
{#array[@]}"
echo "Total number of elements in the array = $
{#array[*]}"
echo "Length of the 1st element = ${#array[0]}"

$ ./array-length.sh
${array[@]} = one two three four
Total number of elements in the array = 4
Total number of elements in the array = 4
Length of the 1st element = 3
```

62. Loop through an Array

You can loop through an array using any one of the bash for loop methods:

```
$ cat array-for1.sh
array=("one" "two" "three" "four")
for i in "${array[@]}"
do
 echo $i
done
total=${#array[@]}
for (( i = 0 ; i < $total ; i++ ))
do
 echo "array[$i] = ${array[$i]}"
done

$ ./array-for1.sh
one
two
three
four
array[0] = one
array[1] = two
array[2] = three
array[3] = four
```

To manipulate a group of files, you can use globbing and put those files in an array. In the following example, an array with filenames is created, and the individual filenames are listed using a for loop.

```
$ cat array-for-files.sh
i=1
files=(/etc/[abcd]*.conf)
```

```
for item in ${files[@]}
do
 echo "$((i++)): $item"
done
```

63. Load File Content to an Array

Create a sample file called states.txt

```
$ vi states.txt
CA
NV
LA
TX
```

The following example loads all the words from the states.txt as individual array elements.

```
$ cat array-file.sh
array=( `cat states.txt` )
for i in "${array[@]}"
do
 echo $i
done

$ ./array-file.sh
CA
NV
LA
TX
```

Note: When the file contains a line that has multiple words, each and every word will be loaded into a separate array element.

64. Associative Arrays in Bash

Associative arrays are like traditional arrays except they uses strings as their indexes rather than numbers. In all the previous array examples, array indexes contained only numbers, i.e. array[0], array[1], etc.

However, in associative arrays, instead of a number we can use a string index. Associative arrays can be declared using the declare statement as shown below. Please note that this uses upper case -A.

```
declare -A array
```

The following simple example shows how to declare and assign values to an associative array with string indexes.

```
$ cat aarray.sh
declare -A states
states=( ["CA"]="California" ["NY"]="New York" )
echo "You entered: ${states[${1}]}"

$ ./aarray.sh CA
You entered: California
```

To loop through the indexes (and not the values), use ${!array[@]} as shown:

```
$ cat aarray1.sh
declare -A states
i=1
states=( ["CA"]="California" ["NY"]="New York"
["NV"]="Nevada" ["UT"]="Utah")
for index in ${!states[@]}
do
 echo "$((i++)): $index"
done
```

```
$ ./aarray1.sh
1: NY
2: UT
3: NV
4: CA
```

Note: The order in which the values are returned by ${!array[@]} is not predictable. As you see in the above example, when we declared the states array, the index of the 1st element is "CA". However when we looped through the indexes using ${!array[@]}, it returned "NY" as the 1st index.

In a traditional array, when you refer to an element, you can directly give a variable name in the index without giving a $ in front of it. Both array[$i] and array[i] will work in a traditional array. However, the variable name without $ does not work with an associative array. Only array[$i] will work.

In this example, the 2nd echo statement that uses ind2 will not work, however the 3rd echo statement that uses $ind2 will work.

```
$ cat aarray2.sh
declare -A states2
states1=("California" "New York" "Nevada" "Utah")
states2=( ["CA"]="California" ["NY"]="New York"
["NV"]="Nevada" ["UT"]="Utah")
ind1=0
ind2=CA
echo "1. \${states1[ind1]} = ${states1[ind1]}"
echo "2. \${states2[ind2]} = ${states2[ind2]}"
echo "3. \${states2[\$ind2]} = ${states2[$ind2]}"

$ ./aarray2.sh
```

```
1. ${states1[ind1]} = California
2. ${states2[ind2]} =
3. ${states2[$ind2]} = California
```

65. Mapfile

Using mapfile you can load the content of a file into an array.

```
$ cat states.txt
CA
NV
LA
TX

$ mapfile states < states.txt

$ echo ${states[@]}
CA NV LA TX
```

Using mapfile -n, you can specify the number of lines to be loaded. This example loads only the first two lines.

```
$ mapfile -n 2 states < states.txt

$ echo ${states[@]}
CA NV
```

Using mapfile -O, you can start the index with a number other than 0. The following example starts the index with 5.

```
$ unset states
$ mapfile -O 5 states < states.txt
$ echo ${states[5]}
```

```
CA
```

Using mapfile -s, you can skip lines. The following ignores the first two lines while loading.

```
$ mapfile -s 2 states < states.txt
$ echo ${states[@]}
LA TX
```

Using -C and -c (together), you can display the array element numbers as they are loaded from the file to the array. This is helpful to display a progress bar while loading a big file.

- -C is a callback command (echo in this example).
- -c is the quantum, i.e. the number of lines to be read for each pass
- Every ‘quantum’ lines ‘callback’ will be executed

The –C and –c usage is demonstrated in the following example.

```
$ mapfile -c 1 -C echo  states < states.txt
0
1
2
3
```

Note: Instead of reading it from a file you can also read from a specific file descriptor using "mapfile -u filedescriptor". Also, please note that readarray is a synonym for mapfile.

If you don't specify an array name, bash will use MAPFILE as array name.

```
$ mapfile < states.txt
$ echo ${MAPFILE[@]}
CA NV LA TX
```

Chapter 7. Bash I/O Redirection

66. Input and Output Redirection

The following three file descriptors are opened by bash, by default.

- stdin - For standard input, to get input from user via the keyboard. The file descriptor for stdin is 0.
- stdout - For standard output, to display the output on the screen. The file descriptor for stdout is 1.
- stderr - For standard error, to display the error messages on the screen. The file descriptor for stderr is 2.

Input Redirection Using <

As shown in several previous examples, you can get input from a file instead of the keyboard by using the < symbol. If you include a file descriptor in front of the < symbol, the input for that file descriptor will be taken from the specified file. Since 0 is the file descriptor for the standard input, < and 0< will do exactly the same thing.

The following example uses the cat command, which (when an argument is not specified) takes its input from stdin and writes to stdout.

```
$ cat 0< /etc/passwd
$ cat < /etc/passwd
```

The following is a valid, but useless, redirection. Nothing happens here, as the input redirection is not passed to any command.

```
$ < /etc/passwd
```

Output Redirection Using >

As was also shown previously, > is used for output redirection.

A file descriptor added in front of the > will send the specified file descriptor's output to the specified file. Also, 1> is the same as >.

Create a new file and add some content to it quickly:

```
cat > newfile.txt
```

Note: When using > to redirect, if the file exists, it will be overwritten; if the file doesn't exist, it will be created.

Append Using >>

Since > truncates the file, and creates the content, it is not always appropriate. You may want to use >> which appends to an existing file (without overwriting the existing content).

In the following example, if newfile.txt doesn't exist, it will be created. If it exists, the new content will be appended to the existing content.

```
cat >> newfile.txt
```

67. Error Redirection

As just shown, using output redirection >, we can send the output of any command to a file as shown below.

```
ls -l > /tmp/output.txt
```

However, when the command displays an error, that doesn't go to a file, as shown below.

```
$ ls -l /etc/doesnotexist.conf > /tmp/output.txt
ls: cannot access /etc/doesnotexist.conf: No such file or
directory


$ cat /tmp/output.txt
```

If you want both output and error to be redirected to the same output file, use &> as shown below.

```
$ ls -l /etc/doesnotexist.conf &> /tmp/output.txt


$ cat /tmp/output.txt
ls: cannot access /etc/doesnotexist.conf: No such file or
directory
```

The above can also be written as shown below.

```
$ ls -l /etc/doesnotexist.conf >/tmp/output.txt 2>&1


$ cat /tmp/output.txt
ls: cannot access /etc/doesnotexist.conf: No such file or
directory
```

Please note the following:

- 2>&1 means file descriptor 2 is redirected to file descriptor 1, which means, standard error is also redirected to standard output. So, if standard output was pointed to a file, the standard error will also be pointed to the same file where the standard output is pointing.
- &> is same as >&
- Use of "> file 2>&1" is easier to read than "&> file".

When you use "> file 2>&1" it means:

- > file indicates that it redirects the standard output to a file.
- 2>&1 indicates that the standard error (2>) is redirected to the same file descriptor that is pointed by standard output (&1), which is a file.

Warning: The sequence in which you give the operators is important. The following is not the same as the previous example.

```
$ ls -l /etc/doesnotexist.conf 2>&1 >/tmp/1.txt
```

Wrong: This example has 2>&1 first, and >/tmp/1.txt next. This will not produce the intended results because:

- 2>&1 indicates that standard error (2>) is redirected to the same file descriptor associated with standard output (&1). Since standard output at this stage is the terminal, the error messages go to the terminal, and not to the /tmp/1.txt file that comes next.
- >/tmp/1.txt redirects standard output, so the output would go to the /tmp/1.txt file. However standard error was already redirected, and is not changed.

Append Standard Output and Standard Error

If you want to append both the standard output and the standard error to an existing file do the following.

```
$ ls -l /etc/doesnotexist.conf &>> /tmp/output.txt


$ cat /tmp/output.txt
ls: cannot access /etc/doesnotexist.conf: No such file or
directory
```

The following is equivalent to the last example:

```
$ ls -l /etc/doesnotexist.conf >> /tmp/output.txt 2>&1


$ cat /tmp/output.txt
ls: cannot access /etc/doesnotexist.conf: No such file or
directory
```

Pipe Standard Error to Other Commands

The following example pipes only the standard output, not the error:

```
$ ls -l /etc/doesnotexists.conf | wc -l
```

```
ls: cannot access /etc/doesnotexists.conf: No such file or
directory
0
```

The following pipes both standard output and error:

```
$ ls -l /etc/doesnotexists.conf 2>&1 | wc -l
1
```

The "|&" is an easier way is to do it:

```
$ ls -l /etc/doesnotexists.conf |& wc -l
1
```

Note: |& is the same as 2>&1 |

68. Here Document

Using the HERE document syntax, you can send commands to an interactive program from a shell script.

```
$ program << BLOCK
command1
command2
command3
BLOCK
```

All the commands that are enclosed between the "<< BLOCK" and "BLOCK" are passed to the “program”. Please note that "BLOCK" can be any word that you specify, as long as the beginning and end match.

FTP scripting example using HERE document:

```
$ ftp -in URL << SCRIPTEND
```

```
user USERNAME PASSWORD
binary
cd PATH
mget PATH
SCRIPTEND
```

In the above ftp scripting example:

- ftp : ftp command
- -i : Disable interactive prompting
- -n : Disable autologin
- user USERNAME PASSWORD : login using the supplied user name and password
- binary : Set binary mode for binary file transfer
- cd PATH : change directory in remote machine
- mget PATH : get the specified file
- SCRIPTEND : tag used to delimit HERE block

Another good use of HERE document is to create a quick file from a shell script, using the cat command.

```
$ vi here-cat.sh
cat > cities.txt <<EOF
Los Angeles
     Las Vegas
     San Francisco
Santa Monica
EOF

$ ./here-cat.sh

$ cat cities.txt
Los Angeles
	Las Vegas
	San Francisco
Santa Monica
```

The above example will take everything within the "<<EOF" and "EOF" block and write that to cities.txt file (including the leading tabs in the lines).

If you don't want the leading tabs in the individual lines to be written to the files, use "<<-EOF" (instead of <<EOF) as shown in the following example.

```
$ vi here-cat1.sh
cat > cities.txt <<-EOF
Los Angeles
	Las Vegas
	San Francisco
Santa Monica
EOF

$ ./here-cat1.sh

$ cat cities.txt
Los Angeles
Las Vegas
San Francisco
Santa Monica
```

Note: "<<-EOF" removes only leading tabs and not leading spaces.

You can use HERE doc to display the usage of your shell script as shown below.

```
$ cat here-backup.sh
if [ $# -ne 2 ]; then
cat <<-EOF
     Usage: here-backup.sh source destination
     source - The source directory for backup
     destination - The destination directory to store the
backup
EOF
fi

$ ./here-backup.sh
Usage: here-backup.sh source destination
source - The source directory for backup
destination - The destination directory to store the
backup
```

Another variation of HERE document is <<<, where we don't need to specify the keyword to start the block and end the block. The string that we pass after <<< itself is used as input to the command.

This is helpful when you are creating a file with just one line entry in it, as shown below.

```
$ cat > newfile.txt <<< this-is-good

$ cat newfile.txt
this-is-good

$ cat > newfile.txt <<< "this is good"

$ cat newfile.txt
this is good
```

69. Process List

Process substitution creates a named pipe under /dev/fd, which is used by multiple commands to write the output, and the other command to read from it for its input.

Using process substitution you can either redirect the output of multiple commands, or read input for multiple commands.

Syntax:

```
>(list-of-commands)
<(list-of-commands)
```

- >(list-of-commands) will take the input from the /dev/fd/xx
- <(list-of-commands) will write the output to /dev/fd/xx

The following simple example shows how to use process list. In addition to displaying the output of the "wc -l" command, bash also displays the /dev/fd/xx file that was used by the process list.

```
$ wc -l <(ls /etc/; cat /etc/passwd; echo "Hello World")
254 /dev/fd/63
```

This is very effective when you want to pass multiple arguments to a command, and both the arguments are output of some other commands.

Here is an example using diff to compare two directory listings:

```
diff <(ls /etc/) <(ls /backup/etc)
```

Here is an example using diff to compare ls options:

```
diff <(ls) <(ls -a)
```

As we explained above, each <( ) creates a /dev/fd/xx file. If you add an echo command in front of the above diff statement, you can see that it is really doing the diff between the two /dev/fd/xx streams that contain the output of the corresponding ls commands.

```
$ echo diff -w <(ls) <(ls -a)
diff -w /dev/fd/63 /dev/fd/62
```

70. /dev/null

/dev/null is the null device, i.e. it is nothing. Anything that goes into it never comes back. If anything comes out of it, it is just nothing. However /dev/null is very useful, as shown below.

Ignore the standard output.

```
ls -l /etc/passwd /etc/junk > /dev/null
```

Ignore the standard error.

```
ls -l /etc/passwd /etc/junk 2> /dev/null
```

Ignore both standard output and standard error.

```
ls -l /etc/passwd /etc/junk > /dev/null 2> /dev/null
ls -l /etc/passwd /etc/junk > /dev/null 2>&1
ls -l /etc/passwd /etc/junk &> /dev/null
```

When you want to truncate your log files, don't do rm and touch, which might mess-up the file permission. Instead, do the following, which will just empty the file content without changing the existing file permissions.

```
cat /dev/null > /usr/local/apache2/logs/error_log
```

Chapter 8. Additional Bash Features and Commands

71. Use ; to Separate Multiple Commands

The list commands (both && and ||) execute the commands in the list conditionally, depending on the status of the previous command in the list.

However using ; you can simply group multiple commands and execute ALL of them, irrespective of the return status of the previous commands.

In the following example, the pwd1 command fails with a return status of 127 (command not found), but bash still moves on to execute the rest of the commands in the list.

```
$ ls; pwd1; cd; date
anaconda-ks.cfg  bin  install.log  install.log.syslog
rules.sh  sshd_config.txt
-bash: pwd1: command not found
Sat Aug 06 15:00:16 PDT 2011
```

72. Command Grouping with () and {}

Both () and {} are used to group multiple commands together.

Syntax:

```
( command1; command2; .... commandn )
{ command1; command2; .... commandn; }
```

Note: It is very important to note that the semi-colon after the last command is required within the { } grouping.

- () creates a sub shell and executes all the commands that are enclosed in that new sub shell environment. This is helpful when you want to redirect the output of all the commands to an output stream. But, Any variable assignment within ( ) will not be visible outside the grouping.
- {} doesn't create a sub shell. Bash executes all the commands that are enclosed in the current shell environment. Any variable assignment within { } will be visible outside the grouping.

Since () creates a new sub shell, the FILENAME1 variable defined inside the parentheses is not available outside the grouping.

```
$ ( ls /etc/|wc -l; FILENAME1=/etc/passwd; echo $FILENAME1
)
223
/etc/passwd


$ echo $FILENAME1
```

Since {} doesn't create a new sub shell, the FILENAME2 variable defined inside the braces is available outside the grouping.

```
$ { ls /etc/|wc -l; FILENAME2=/etc/passwd; echo
$FILENAME2; }
219
/etc/passwd


$ echo $FILENAME2
/etc/passwd
```

{ } is extremely useful when you combine it with && or ||. For example, when you want to echo an error message in a grouping condition only for one condition, but not for others, you can use the method shown here:

```
$ file=/etc/yp.conf
$ [ -e $file ] && { rm $file || echo "Unable to delete
yp.conf.bak"; }
rm: cannot remove `/etc/yp.conf': Permission denied
Unable to delete yp.conf.bak
```

The following two forms of { } are valid:

1. semicolon before the closing brace:

```
$ { ls; pwd; }
```

2. { and } in their own lines with all the commands in-between:

```
$ {
> ls
> pwd
> }
```

Be careful to avoid the following syntax errors while using { }:

error1: no space after { and/or before }:

```
$ {ls;pwd;}
-bash: syntax error near unexpected token `}'
```

You should always have a space before after { and before }.

```
$ { ls;pwd; }
```

error2: Missing ; before }

```
$ { ls; pwd }
-bash: syntax error near unexpected token `;'
```

73. User Defined Functions

If you are repeatedly executing the same set of commands all the time, consider putting them in a function; any time you want to execute the commands, you just execute the function. Inside a shell script, if you are repeating some set of commands, define a function, and call it appropriately.

Shell functions are executed in the current shell context without having to create a new shell to execute them. A shell function exits with the exit status of the last command in the function.

Syntax:

```
function name () {
command1
command2
}
```

- The keyword function is optional, you can just define the function using only the "name ()"
- When you do use the keyword function, you can leave out the "()" after the function name. i.e "function name" is same as "function name ()"

All of the following are valid function definitions.

```
function name () {
}
name () {
}
function name {
}
```

Note: Preferred method is "name () { }"

To call a function, simply reference the function name, i.e "name" in the command line or in a script will invoke the function.

Examples of Functions

Simple example:

```
$ cat function.sh
function display1 () {
 echo "display1: Hello World!"
}
function display2 {
 echo "display2: Hello World!"
}
display3 () {
 echo "display3: Hello World!"
}
display1
display2
display3
```

Use local builtin to define local variables inside the function:

```
$ cat function1.sh
function getstate () {
 state="CA"
 local city="Los Angeles"
 echo "Inside function: state=[$state] city=[$city]"
}
getstate
echo "Outside function: state=[$state] city=[$city]"

$ ./function1.sh
Inside function: state=[CA] city=[Los Angeles]
```

```
Outside function: state=[CA] city=[]
```

Calling a Function in Bash PS1

Earlier we explained how you can change your command prompt using the PS1 variable. You can also use a user defined function in the PS1 variable.

For example, if you want to display the total number of running httpd processes in your prompt, add the following function to your .bash_profile.

```
$ vi ~/.bash_profile
function httpdcount {
  ps aux | grep httpd | grep -v grep | wc -l
}
export PS1='\u@\h [`httpdcount`]$ '

$ source ~/.bash_profile
```

Passing Arguments to a Function

Just as you can pass arguments to a shell script, you can also pass arguments to a user-defined function.

Like the positional parameters in a shell script, you can refer to the function arguments using $1, $2, $3, etc.

The following example defines a getcapital function that takes the state as an argument. State is processed inside the function by referring to it as "$1"

```
$ cat function2.sh
function getcapital () {
 state="$1"
 case $state in
```

```
    "CA" ) echo "Sacramento";;
    "UT" ) echo "Salt Lake City";;
    "NV" ) echo "Carson City";;
      *  ) echo "State not recognized";;
  esac
}
getcapital "UT"

$ ./function2.sh
Salt Lake City
```

74. Set Builtin

Using the set builtin you can:

- Display shell variables and their values, including the environment variable, functions, etc.
- Manipulate the shell options (view/modify shell options)
- Set positional parameters

View Shell Variables

Without any arguments, the set builtin displays all the variables and functions.

```
$ set
BASH=/bin/bash
BASHOPTS=checkwinsize:cmdhist:expand_aliases:extquote:forc
e_fignore:hostcomplete
BASH_VERSINFO=([0]="4" [1]="1" [2]="2" [3]="1"
[4]="release" [5]="x86_64-unknown-linux-gnu")
BASH_VERSION='4.1.2(1)-release'
COLORS=/etc/DIR_COLORS
CVS_RSH=ssh
..
```

Set or Unset Shell Options

To set an option using option name, do the following.

```
set -o option-name
```

To unset an option using option name, do the following.

```
set +o option-name
```

Each option also has a short option name that can be called directly to set or unset (Without having to specify the full option name).

To set an option using the one character option short form, do the following.

```
set -[one-character-option-short-form]
```

To unset an option using the one character option short form, do the following.

```
set +[one-character-option-short-form]
```

For example, noglob is a shell option that can be turned on using any one of the following method.

```
set -o noglob
set -f
```

- noglob is the option name
- f is the one character option short form

To view all the current shell options that are set, do the following:

```
$ echo $-
fhimBH
```

Note: The output indicates that the options f, h, i, m, B and H are set.

The noglob shell option can be turned off using either of the following methods:

```
set +o noglob
set +f
```

When globbing is disabled, shell will not recognize the globbing characters ?, *, etc. as shown below.

```
$ set +f

$ ls -l /etc/???.conf
-rw-r--r--. 1 root root 1780 Oct 15  2009 /etc/cas.conf
-rw-r--r--. 1 root root    0 Apr  4 14:20 /etc/gai.conf
-rw-r--r--. 1 root root 1917 Jul  6  2010 /etc/ntp.conf
-rw-r--r--. 1 root root  186 Jan 29 09:02 /etc/sos.conf
-rw-r--r--. 1 root root  813 Nov 20 13:47 /etc/yum.conf

$ set -f

$ ls -l /etc/???.conf
ls: cannot access /etc/???.conf: No such file or directory
```

The following table lists all the shell options (and their short forms) that can be set or unset using the set command. The optios marked in bold are enabled by default.

Option Name	Option Short Form	Description
allexport	-a	Marks variables for export to the environment of subsequent commands

Option Name	Option Short Form	Description
braceexpand	-B	Perform brace expansion
emacs		For command line editing using emacs like keys
errexit	-e	Exit if pipeline returns non-zero exit status.
errtrace	-E	ERR trap is inherited by child process that are forked from the shell
functrace	-T	DEBUG and RETURN trap are inherited by child process that are forked from the shell
hashall	-h	Enable hash table that remembers the commands (and its count) that are executed
histexpand	-H	History substitution using ! is enabled
history		Enables command history. For interactive shell this is set automatically.
ignoreeof		Interactive shell will not terminate when an EOF is encountered
keyword	-k	Arguments formed as assignment statement are set in the environment for commands.
monitor	-m	Enable job control
noclobber	-C	Output redirection >, >&, <> will not overwrite an existing file.

Option Name	Option Short Form	Description
noexec	-n	Commands are read, but not executed. Use this inside shell script for syntax checking. Cannot be used in interactive shell.
noglob	-f	File name globbing is disabled
notify	-b	Notify the status of a terminated background job immediately.
nounset	-u	During parameter expansion, unset variables will be reported as an error, and non interactive shell will exit.
onecmd	-t	Executes only one command. After that it just exits.
physical	-P	cd will not following symbolic links, instead it will follow the physical directory, when you do cd to symbolic links.
pipefail		Pipeline returns status of last command that exits with non-zero status (zero if all the commands are successful).
posix		Strict superset of posix standard
privileged	-p	Run shell in privileged mode. $BASH_ENV, $ENV are not processed. SHELLOPTS, BASHOPTS, CDPATH, GLOBIGNORE are ignored.
verbose	-v	Verbose mode. Input lines are printed

Positional Parameters

Typically positional parameters are passed to the shell script as arguments. They can be manipulated inside the shell script using $1, $2, etc.

Using the set command, we can assign values to the positional parameters without having to pass the values as an argument to the shell scripts.

Inside a shell script (or in the command line), when you do the following, "one" will be stored in $1, "two" will be stored in $2, "three" will be stored in $3.

```
$ set one two three
$ echo $1
one
```

The following example demonstrates how the set command can be used to set positional parameters.

```
$ cat set.sh
set `uname -a`
echo "$1"
set `ls /etc/*.conf`
echo "$1"
set one two three four five
echo "$1"
var="LA NY CA"
set -- $var
echo "$1"
set --
echo "$1"

$ ./set.sh
```

```
Linux
/etc/asound.conf
one
LA
```

Note: It is important to understand the last set command in the above set.sh script. When you use "set --" bash unsets all the positional parameters that were set earlier. This is why the last "echo $1" statement doesn't print anything.

75. Bash Command Line Options

When you invoke bash from command line, the following is the syntax:

```
bash [long-options] [single-char-options] [arguments]
```

Note: The long options should come before the single character options.

Long options

The following table explains the long options that can be passed to the bash command line.

Long Option	Description
--debugger	Sets debugger profile before starting the shell. This enables extdebug shopt option, and functract set option
--help	Displays bash version and a brief help message about bash command line options.
--init-file --rcfile	Define your own custom .bashrc file. Instead of reading standard ~/.bashrc, it will read the custom file you've specified. (for example: bash --init-file ~/.mybashrc)
--noediting	This will not use the readline to read commands in interactive shell
--noprofile	This will not execute /etc/profile, or

	~/.bash_profile, or ~/.bash_login, or ~/.profile during startup
--norc	This will not execute ~/.bashrc in interactive shell
--posix	Strict superset of posix standard
--restricted	Execute shell in restricted mode
--verbose	Verbose mode
--version	Display bash version information

Single Character Options

Using option -c, any string that is passed after this will be treated as a command and will be executed.

```
$ bash -c "cat /etc/passwd | wc -l"
38
```

Option -i forces bash to run as an interactive shell.

```
$ bash -i
```

Option -l (or option --login) makes the bash shell act as if it was started during the login.

```
$ bash -l
$ bash --login
```

Option -r runs the bash shell in restricted mode.

```
$ bash -r
```

Option -s reads the commands to be executed from the standard input.

```
$ bash -s
```

Options –O and +O let you specify any one of the shopt options to be set, or unset, respectively. For example, you can start a bash shell with the cdspell option set as shown below.

```
$ bash -O cdspell


$ shopt cdspell
cdspell         on
```

Two hyphens (--) provided as argument to the bash command line indicates the end of the bash option list. Anything that comes after -- will be treated as filenames and arguments.

76. Coproc Builtin

coproc is new and was introduced in Bash 4.

Using coproc you can start a background process and interact with its input and output. This is just like starting a job in the background with &, however with the coproc builtin you can establish two-way communication between the background process and the shell that created the process.

Syntax:

```
coproc [NAME] command
```

Note: NAME can be anything that you specify.

Coproc creates the following three environment variables.

- NAME_PID that stores the process id of the background process that was called using coproc

- NAME[0] contains the file descriptor for the standard output of the background process
- NAME[1] contains the file descriptor for the standard input of the background process.

If you don't specify a name for the coproc, the default name is COPROC. In this case, bash creates three variables with the following names:

- COPROC_PID
- COPROC[0]
- COPROC[1]

To demonstrate, the following sample shell script will be called from another shell script using coproc.

```
$ cat coproc-1.sh
for ((i=0; i<=10; i++))
{
 echo "Testing"
 sleep 2;
}
```

The coproc1.sh example shown below creates two co-processes using the coproc command.

- The 1st coproc calls another shell script (coproc-1.sh)
- The 2nd coproc calls a set of commands that are grouped using { }.
- The 1st coproc doesn't have a custom name, so it uses COPROC_PID, COPROC[0], and COPROC[1].
- The 2nd coproc uses CUSTOM as its name, so it uses CUSTOM_PID, CUSTOM[0] and CUSTOM[1].

```
$ cat coproc1.sh
coproc ./coproc-1.sh
echo "COPROC_PID=$COPROC_PID"
echo "COPROC[0]=${COPROC[0]}"
echo "COPROC[1]=${COPROC[1]}"
coproc CUSTOM { ls -l; sleep 2; cat /etc/passwd; sleep
2; }
echo "CUSTOM_PID=$CUSTOM_PID"
echo "CUSTOM[0]=${CUSTOM[0]}"
echo "CUSTOM[1]=${CUSTOM[1]}"
```

77. Command Execution Sequence

When you execute a command from the command line, it can be an alias, a function, a bash builtin, or a shell script. You can define a function with the same name as a bash builtin, or a shell script with the same name as an external program, etc. If you do that, when you execute a command, how does bash decide what needs to be executed. In what order does it look for those?

Following is the sequence in which bash looks for a command when you execute it from the command line:

1. When you execute a command with a full path (i.e. if the command contains a slash), that specific program will be executed. If it is not found, bash will report an error message.

The following command executes the specific program:

```
$ /home/ramesh/bin/backup
```

2. If the command you execute doesn't contain a full path (i.e. there is no slash in the command specification), bash will search as shown in this example:

```
$ mycommand
```

- Bash searches functions to see if there is a match. If a function with the name "mycommand" exists, it will be executed.
- If there is no function by that name, it searches shell builtins to see if there is a match, and executes the builtin if it exists.
- If there is no function or builtin by that name, bash searches for a matching command in the hash table. If a command by name "mycommand" exists in the hash table, it will be executed.
- If there is no matching function, builtin, or command in the hash table, bash searches each and every directory in the $PATH environment variable to locate an executable file with that name. If a program exists in any one of those directories with the name "mycommand", it will be executed.
- If there is no function, builtin, command in the hash table, or program in the $PATH directories, bash searches for a command_not_found_handle function. If it exists, the command is passed as an argument to that function.
- Otherwise bash displays an error message with exit status 127 (command not found).

78. Bash Environment Variables

All the programs that are started by the bash shell have access to the environment variables and their values.

To create an environment variable use the export (or declare -x) command:

```
export PASSWD=/etc/passwd
(or)
declare -x PASSWD=/etc/passwd
```

To view all environment variables use export (or declare -x):

```
$ export
declare -x CVS_RSH="ssh"
declare -x HISTSIZE="1000"
declare -x HOME="/home/ramesh"
declare -x PASSWD="/etc/passwd"
declare -x USER="ramesh
..

$ declare -x
declare -x CVS_RSH="ssh"
declare -x HISTSIZE="1000"
declare -x HOME="/home/ramesh"
declare -x PASSWD="/etc/passwd"
declare -x USER="ramesh"
..
```

Note: The output of the export command will be in the format of 'declare x parameter=value'. This makes it easier to copy/paste the output to another environment to create identical environment variables.

To delete an environment variable, do one of the following:

```
export -n PASSWD
(or)
declare +x PASSWD
```

Exporting an environment variable is a one way operation, i.e, you can only export environment variables from parent to child. You cannot export an environment variable from child to parent.

Please note that the export (and declare -x) command displays both user defined variables and bash shell variables.

79. Bash Shell Variables

The following shell variables are available to users.

Variable Name	Description
CDPATH	Directory list used by cd builtin as search path
HOME	User's home directory
IFS	Character list used to separate fields
MAIL	Notify users when a mail arrives in the mentioned file
MAILPATH	File list used to check for mails
OPTARG	For getopts: Last option's argument value
OPTIND	For getopts: Last option's index value
PATH	Directory list where shell searches for commands
PS1	Primary prompt string. Default value is ‘\s-\v\$ ’
PS2	Secondary prompt string. Default value is ‘>'
BASH	Full pathname to the bash executable
BASHOPTS	List of currently enabled shell options (shopt)
BASHPID	PID of current bash shell
BASH_ALIASES	Associative array contains bash internal aliases

Variable Name	**Description**
BASH_ARGC	Array with number of parameters in each frame of the current bash execution call stack. Bash sets this only when extdebug shopt option is enabled.
BASH_ARGV	Array with all parameters in the current bash execution call stack. Bash sets this only when extdebug shopt option is enabled.
BASH_CMDS	Associative array corresponds to the bash hash table entries.
BASH_COMMAND	The command currently being executed or about to be executed, unless the shell is executing a command as the result of a trap, in which case it is the command executing at the time of the trap.
BASH_ENV	During shell script execution, this file is executed as startup file.
BASH_EXECUTIO N_STRING	When using bash -c, this contains the command arguments
BASH_LINENO	Array with lines numbers in the source file that corresponds to individual functions.
BASH_REMATCH	Array that contains values assigned by the =~ for the [[command.
BASH_SOURCE	Array that contains the filenames of all the functions referred by FUNCNAME environment variable.
BASH_SUBSHELL	Contains total number of time a subshell or subshell environment is spawned. Incremented by one each time a subshell or subshell environment is spawned. The initial value is 0

Variable Name	Description
BASH_VERSINFO	Array that contains bash version number in the order of: major, minor, path, build, release status, MACHTYPE
BASH_VERSION	A string containing the bash version number.
BASH_XTRACEFD	When 'set -x' is executed, trace output are written to this file descriptor.
COLUMNS	Contains current terminal width.
DIRSTACK	Array that contains directory stack contents.
EUID	The userid of logged in user. (same as uid from /etc/passwd for the user)
FCEDIT	Editor used by the fc command
FIGNORE	List of suffixes to ignore for filename completion
FUNCNAME	Array containing all the function names in the execution call stack.
GLOBIGNORE	List of patterns to be ignored for filename expansion
GROUPS	List of groups the current user belongs to
histchars	3 history expansion characters. default is !^#
HISTCMD	Index number for the current command in the history list
HISTCONTROL	List of options that control history list. Possible values: ignorespace, ignoredups, ignoreboth, erasedups, etc.

Variable Name	Description
HISTFILE	History file name. Default is ~/.bash_history
HISTFILESIZE	Maximum number of lines in history file.
HISTIGNORE	List of patterns that decides what commands should be saved in the history
HISTSIZE	Maximum number of commands in the history list. Default is 500.
HISTTIMEFORMAT	Display timestamp for history entries in this format
HOSTFILE	Reads hostname from this file for hostname completion.
HOSTNAME	Current host name.
HOSTTYPE	The host architecture.
IGNOREEOF	Number of consecutive EOF characters required for the shell to exit.
INPUTRC	For readline, use this file instead of the default ~/.inputrc
LANG	Locale category
LINENO	The current line number of the shell script that is getting executed.
LINES	Total number of lines available in the current terminal
MACHTYPE	Current system type in "cpu-company-system" format
MAILCHECK	Mail check interval (in seconds). Default is 60.

Variable Name	Description
OLDPWD	The previous working directory
OPTERR	Display error messages by getopts when OPTERR value is 1
OSTYPE	Operating system type
PIPESTATUS	Array containing exist status from the foreground pipeline processes
POSIXLY_CORRECT	Setting this same as 'set -o posix'
PPID	Parent PID of current shell
PROMPT_COMMAND	The content of this will be executed before displaying PS1
PROMPT_DIRTRIM	Indicates how many directory level should be printed for \w and \W in PS1
PS3	Prompt for select command
PS4	Prints this value before the command is printed for 'set -x'. Default is '+ '
PWD	Contains current working directory
RANDOM	Contains a random number between 0 and 32767
REPLY	Stores the value read from user using read builtin
SECONDS	Number of seconds since the shell was started
SHELL	full pathname of the bash shell

Variable Name	Description
SHELLOPTS	List of shell options that are currently enabled. (same as on values of 'set -o')
SHLVL	Total number of bash shell that were launched in that session.
TIMEFORMAT	Format string for time command output. Default is "\nreal\t%3lR\nuser\t %3lU\nsys\t%3lS"
TMOUT	Timeout value for read builtin. On interactive shell bash exits if user doesn't type anything within this time.
TMPDIR	Indicates the directory where temporary files will be created
UID	Contains the user id of current logged in user (same as uid in /etc/passwd)

80. Exit Command and Exit Status

Every linux command returns an exit status, which indicates whether or not the command was executed successfully.

Exit status of 0 indicates that the command executed successfully. Exit status of non-zero indicates that the command failed. To view the exit status of a command use the $? special parameter.

The following ls command executes successfully, 0 is returned

```
$ ls -l /etc/passwd
-rw-r--r--. 1 root root 1659 Jun  6 08:41 /etc/passwd
$ echo $?
0
```

The following ls command fails, so a non-zero exit status is returned.

```
$ ls -l /etc/junk
ls: cannot access /etc/junk: No such file or directory

$ echo $?
2
```

Using the exit command, you can terminate a shell script and also return an user-defined exit status value to the shell. The exit status value should be in the range 0 - 255.

Normally the exit status of a shell script is the exit status of the last command that got executed in the shell script. Use the exit command if you want to return a custom exit status from a shell script.

The following simple exit.sh example will return an exit status of 200.

```
$ cat exit.sh
echo "Hello World!"
exit 200

$ ./exit.sh
Hello World!

$ echo $?
200
```

Note: When a fatal signal with value n terminates a command, the exit status from bash is 128+n.

When you have multiple commands in a pipe, the exit status is the status of the last command in the pipe. Even when one of the commands in the pipe fail, and if the last command is successful, the exit status will be 0.

```
$ pwdd
$ echo $?
127

$ pwdd | ls
$ echo $?
0
```

The following are some standard exit status codes.

- 0 Success
- 1 General failure message
- 2 Returned by shell builtins when they are misused
- 126 Permission issue with the command
- 127 Command not found
- 128+n Command terminated by fatal signal n
- 130 Ctrl-C termination

81. Bash Restricted Shell

In a bash restricted shell, some commands cannot be executed.

Start a bash restricted shell as shown below (the commands are identical):

```
$ bash -r (or)
$ bash --restricted
```

Verify that you are on restricted shell mode.

```
$ shopt | grep restricted
restricted_shell        on
```

The following are a few things you cannot do in a bash restricted shell.

1. Cannot do CD

```
$ cd /etc
bash: cd: restricted
```

2. Cannot change PATH, SHELL, ENV, BASH_ENV variables

```
$ export PATH=$PATH:/root/bin
bash: PATH: readonly variable
```

3. Cannot give / in the command name. (including ./)

```
$ /usr/bin/find
bash: /usr/bin/find: restricted: cannot specify `/' in
command names
```

4. Cannot perform any output redirection. (including >, >|, <>, &<, >&, >>)

```
$ ls -l > /tmp/1.txt
bash: /tmp/1.txt: restricted: cannot redirect output
```

5. Cannot specify a filename with / to the . command (i.e source command)

```
$ source /home/ramesh/.bashrc
bash: source: /home/ramesh/.bashrc: restricted
```

Note: Inside a shell script, use set -r to start the restricted mode.

82. Debug a Bash Shell Script

When things go wrong in a shell script, you might want to use debug mode to help you understand better what it is doing.

set -x

For example, the following doesn't display anything except the output.

```
$ ./function2.sh
Salt Lake City
```

To debug the shell script, use "sh -x scriptname", which will display the command it is currently executing as it moves through the shell script as shown below.

```
$ sh -x function2.sh
+ getcapital UT
+ state=UT
+ case $state in
+ echo 'Salt Lake City'
Salt Lake City
```

The "+ " in the prefix indicates the command that the shell scripts echos.

As we discussed earlier, you can change the "+ " suffix to anything you like using PS4 variable, as shown below.

```
$ export PS4="---> "

$ sh -x function2.sh
---> getcapital UT
---> state=UT
---> case $state in
```

```
---> echo 'Salt Lake City'
Salt Lake City
```

Instead of doing "sh -x scriptname" from the command line, you can also add "set -x" inside the shellscript, and just execute the shell script as usual.

```
$ cat debug-function2.sh
set -x
PS4="****> "
function getcapital () {
 state="$1"
 case $state in
   "CA" ) echo "Sacramento";;
   "UT" ) echo "Salt Lake City";;
   "NV" ) echo "Carson City";;
     *  ) echo "State not recognized";;
 esac
}
getcapital "UT"

$ ./debug-function2.sh
----> PS4='****> '
*****> getcapital UT
*****> state=UT
*****> case $state in
*****> echo 'Salt Lake City'
Salt Lake City
```

When you use "set -x", bash prints the command after performing the expansion and substitutions, so you can see the result just before execution.

The options "set -x" and "set -o xtrace" are exactly the same. You can use either one of them.

You can also do partial debugging of your script. Add "set -x" to a specific location in the shell script where you want to begin the debugging. Add "set +x" to a specific location in the shell script where you want to end the debugging.

set -n (and set -v)

To check the syntax without executing the shell script, use “set –n”:

```
$ sh -n debug-function2.sh

debug-function2.sh: line 10: syntax error near unexpected
token `newline'

debug-function2.sh: line 10: ` bsac'
```

Combine -n with the verbose option (-v) to display the statements and do the syntax check (without executing the statements).

```
$ sh -nv debug-function2.sh
```

When you use "set -v" (or sh -v, or set -o verbose), bash prints the commands as they are read from the input (before performing expansion and substitutions).

Using the Echo Statement for Debug

For larger shell scripts, when you suspect some issues, make a copy of the script and add several echo statements to it to debug the issue. After identifying the problem, remove the debug parts and copy your revised script over the original.

For example, let us assume that you want to debug the script myshellscript.sh. Take a copy of it first, as shown below.

```
cp myshellscript.sh myshellscript-debug.sh
```

Modify myshellscript-debug.sh, and add several echo statements where you think it is appropriate. Make sure to give a specific prefix in the echo statement, which will make it easier to identify the debug echo statements and remove them later, for example:

```
$ vi myshellscript-debug.sh
echo "MYDEBUG: [your custom message]"
```

After you've fixed the issue, remove all the debug echo statement using the sed command as shown below. This sed command will delete all the lines in the myshellscript-debug.sh that contains the word "MYDEBUG".

```
sed -i '/MYDEBUG/ d' myshellscript-debug.sh
```

Now, overwrite your original file with this modified final version.

```
cp myshellscript-debug.sh myshellscript.sh
```

There is another neat trick you can do on a big shell script that you frequently need to debug. Just add a debug routine inside the shell script, that will print debug message only when you set a debug flag.

Add a function called "debug" at the top of your shell script. Inside the shell script, call the "debug" function with the message to be displayed whenever it is appropriate for the debug function as shown below.

```
$ cat debug-getopts.sh
debug () {
[ -n "$DEBUG" ] && [ $DEBUG == "ON" ] && echo "===>DEBUG:
$1"
}
while getopts "cvf:z" Option
do
  debug "[$LINENO] Processing $Option"
  case $Option in
    c ) echo -n "Option c is used. "
```

```
        echo "OPTIND=$OPTIND"
        ;;
    v ) echo -n "Option v is used. "
        echo "OPTIND=$OPTIND"
        ;;
    f ) echo -n "Option f is used. "
        echo "OPTIND=$OPTIND OPTARG=$OPTARG"
        ;;
    z ) echo -n "Option z is used. "
        echo "OPTIND=$OPTIND"
        ;;
  esac
done
shift $(($OPTIND - 1))
debug "[$LINENO] End of the script"
```

When you execute the script normally, it doesn't print your debug message.

```
$ ./debug-getopts.sh -c
Option c is used. OPTIND=2
```

However, when you set the DEBUG variable to ON and execute the shell script, it will display your custom debug message.

```
$ DEBUG=ON ./debug-getopts.sh -c
===>DEBUG: [6] Processing c
Option c is used. OPTIND=2
===>DEBUG: [23] End of the script
```

Note: The $LINENO bash variable used in this example will come in very handy during debugging. Don't make the mistake of adding the $LINENO to the echo statement in the debug function. That would always print the line number of that particular line, and not the line number from where the debug function was called.

83. IFS

“IFS” stands for “Internal Field Splitter” and identifies what characters bash uses to identify word boundaries.

By default (even when IFS is not set), the following characters are used for IFS in word splitting.

- space
- tab
- new line

The following simple example shows how you can define IFS inside your program to change the word splitting behavior.

In the following example, the "states" variable contains values separated by commas. When you loop through $states in a for loop, it will be treated as just one value.

When you set the IFS value to comma, the same for loop will treat the individual values separately, as bash will use the comma (,) for word splitting.

```
$ cat ifs.sh
i=1;
states="CA,UT,NV,NY"
IFS=,
for state in $states
do
  echo "$((i++)). $state"
done

$ ./ifs.sh
1. CA
2. UT
```

```
3. NV
4. NY
```

84. Trapping Signals using trap

Using trap command you can trap signals (For example, Ctrl-C, Ctrl-D).

Syntax:

```
trap command SIGNAL
```

Whenever the specific SIGNAL is received, the specified command will execute.

Display a message anytime you press Ctrl-C:

```
$ cat debug-trap.sh
trap 'echo Processing $i' INT
for (( i=1; i<10; i++))
do
   sleep $i
done

$ ./debug-trap.sh
^CProcessing 6
^CProcessing 7
^CProcessing 8
^CProcessing 9
```

Display a message when an error occurs.

```
$ cat trap-err.sh
```

```
trap 'echo Error occurred $i' ERR
for (( i=1; i<3; i++))
do
   sleepp $i
done

$ ./trap-err.sh
./trap-err.sh: line 4: sleepp: command not found
Error occurred 1
./trap-err.sh: line 4: sleepp: command not found
Error occurred 2
```

Note: if you trap Ctrl-C you will not be able to stop execution of a runaway script.

Trap is very helpful for debugging your shell script. Use the DEBUG signal and echo the variables that you want to watch in the trap command as shown below:

```
$ cat trap-debug.sh
trap 'echo Debugging day=$day i=$i' DEBUG
i=1
for day in Mon Tue Wed Thu Fri
do
 echo "Weekday $((i++)) : $day"
 if [ $i -eq 3 ]; then
   break;
 fi
done

$ ./trap-debug.sh
Debugging day= i=
Debugging day= i=1
Debugging day=Mon i=1
```

```
Weekday 1 : Mon
Debugging day=Mon i=2
Debugging day=Mon i=2
Debugging day=Tue i=2
Weekday 2 : Tue
Debugging day=Tue i=3
Debugging day=Tue i=3
```

85. Automatic Case Conversion using declare

When a variable is defined using the declare command with one of the following specific options, each value stored into the variable is automatically converted to either lower case, or upper case, or capitals:

- declare -l will treat the content of the variable as lower case
- declare -u will treat the content of the variable as upper case
- declare -c will treat the content of the variable as capitalized

In the following example, the variable "small" is defined using "declare -l". Any value that is stored into this variable "small" will be automatically converted to lower case as shown below.

```
$ declare -l small
$ small="THIS IS BIG"

$ echo $small
this is big
```

In the following example, the variable "big" is defined using "declare -u". Any value that is stored into this variable "big" will be automatically converted to upper case as shown below.

```
$ declare -u big
$ big="this is small"
$ echo $big
THIS IS SMALL
```

In the following example, the variable "capital" is defined using "declare -c". Any value that is stored into this variable "capital" will be automatically converted to capitalized text as shown below.

```
$ declare -c capital
$ capital="this is small and BIG"

$ echo $capital
This is small and big
```

After you declare a variable, if you use the set command to view the variable and the content, bash doesn't tell you how the variable was initially declared.

```
$ set | egrep 'small|big'
big='THIS IS SMALL'
capital='This is small and big'
small='this is big'
```

When you use 'declare -p' to print the variables, bash will also show you how the variable was defined initially, i.e. which declare option was used to define it.

```
$ declare -p | egrep 'small|big'
declare -u big="THIS IS SMALL"
declare -c capital="This is small and big"
declare -l small="this is big"
```

86. Exec Builtin

exec built-in is used to replace the current shell with the given command. This is helpful when you are writing a wrapper program for a standard command.

When you use the exec command inside your custom shell script, it should be the last command; commands in the shell script after exec are not executed since the shell running the script is replaced by the one specified the exec command.

```
$ cat mytar.sh
echo "Wrapper for tar command"
echo " Doing some pre processing"
exec "/bin/tar" "$@"
echo " Doing some post processing"
```

The echo statement after the exec command does not get executed:

```
$ ./mytar.sh
Wrapper for tar command
 Doing some pre processing
/bin/tar: You must specify one of the `-A the cdtrux' or
`--test-label'  options
Try `/bin/tar --help' or `/bin/tar --usage' for more
information.
```

Exec is also helpful when your current shell is not bash, and you would like to start a bash login shell. In that case you can switch to bash by doing 'exec bash -l' or 'exec bash --login'.execfail

By default, the shell will not exit when the exec command fails as specified by the execfail shopt option as shown below.

```
$ shopt execfail
execfail        off
```

When you are running bash in restricted mode, you cannot use exec, as it is disabled for bash restricted mode.

You can specify a global redirection for your shell script using exec command. When you use exec command for specifying redirection inside your shell script, commands that follow the exec command will be executed properly.

87. Commenting Bash Code

Bash scripts may be commented as shown in the following example:

```
$ cat comment.sh
# 1. This is a simple single line comment
echo "Hello World!"
echo "Hello World!" # 2. This is partial line comment
: <<COMMENT
-------------------------------------------
3. Multi line comment inside shell script
can be elegantly documented without giving #
in front of every line
-------------------------------------------
COMMENT
# 4. Comment large blocks of code temporarily
: <<TEMP
for i in "${array[@]}"
do
 echo $i
done
TEMP
```

- If a line starts with #, the whole line is treated as a comment line.

- Part of a line can be commented. For example, at the end of a command, you can give # and comment the rest of the line
- Quick multi line documentation can be done inside a shell script without having to # every line. Do this by specifying the lines between ": <<COMMENT" and "COMMENT". The word "COMMENT" can be anything.
- The same multi line commenting concept can be utilized when you want to temporarily comment a large block of code for debugging purpose. This is better than adding # in front of each and every line.

88. Source or Dot Command

When you execute a command or shell-script, bash typically spawns a new process to execute it.

However, if you want to execute a shell-script in the current shell, you can use the source command.

Let us say that you added few environment variables to .bash_profile. If you logout and login again, .bash_profile will be executed and the new environment variables you set will be available for you.

After you modify.bash_profile, if you use the source command to execute it in the current shell, you don't need to logout and login again to use the new environment variables you added to the .bash_profile.

```
$ source ~/.bash_profile
```

The dot command is exactly the same as source command. Both of the following do the same thing.

```
$ source ~/.bash_profile
$ . ~/.bash_profile
```

89. Caller Builtin

The caller command prints the context of the current subroutine call, i.e. it displays the line number and filename from which it was called.

If you use the caller command outside a function, it will display "0 NULL". Use caller inside a function to get meaningful results:

```
$ cat caller.sh
function1 () {
 caller
}
function2 () {
 caller 0
}
function1
function1
function2
function2
```

As shown in the example, you can specify an expression as an argument to caller command. If the value is 0, it indicates the top frame. If you specify 1, it will go back 1 frame before the current frame. If you specify 2, it will go back 2 frames before the current frame, etc. In the example, function1 contains just "caller", and function2 contains "caller 0".

When function1 is called, it will display the line number from which it was called, and the current file name. When function2 is called, it will display the line number from which it was called, the top frame name ('main' in this case, since "caller 0" was used), and the current file name.

```
$ ./caller.sh
8 ./caller.sh
9 ./caller.sh
11 main ./caller.sh
```

```
12 main ./caller.sh
```

90. Calling Awk and Sed in a Bash Shell Script

The following examples call an awk program inside a bash shell script.

Call a single line awk statement inside a bash shell.

```
$ cat call-awk.sh
echo "1. Simple usage"
INPUT=/etc/passwd
awk -F: '/root/ {print $1}' $INPUT
echo "2. Using bash variables in awk"
INPUT=/etc/passwd
search="root"
awk -F: '/'"$search"'/ {print $1}' $INPUT
```

- In the first awk example above, awk searches for "root" in /etc/passwd and prints it. $INPUT is a bash variable that contains the name of the input file that needs to be passed to the awk.
- The second awk example above is functionally same as the first, but, the awk search string is a bash variable. This shows how to use a bash variable in a awk script. Please make sure you always double-quote the bash variable.

Call a multi-line awk command inside a bash shell.

```
$ cat call-awk1.sh
echo "Calling multi line awk script (with bash variable
printed)"
INPUT="/etc/passwd"
myheader="Header from Bash Var"
awk '
```

```
BEGIN {
 print "***Awk: '"$myheader"'***"
 FS=":"
}
/root/ {
 print "  Awk Body: " $1
}
END {
 print "***Awk Footer***"
}
' $INPUT
```

If you want to call a bash function inside an awk script, you should first export the bash function, then call it using the system() command inside the awk script:

```
$ cat call-awk2.sh
echo "Calling bash function inside awk"
header() {
 echo "***Bash Header***"
}
export -f header
awk '
BEGIN {
 print "**Awk Header**"
 system("header")
}
```

Use the same concept to call sed inside a bash shell script:

```
$ cat call-sed.sh
INPUT=/etc/passwd
echo "1. Simple sed search"
```

```
sed -n '/root/ p' $INPUT
echo "2. Simple sed substitute"
sed -n 's/root/superuser/ p' $INPUT
echo "3. Sed substitute using bash variables"
original=root
replacement=superuser
sed -n "s/$original/$replacement/ p" $INPUT
```

Chapter 9. Bash Shell Expansion

91. Brace Expansion

Brace expansion is used to generate arbitrary strings. One typical use is to create multiple modified command line arguments out of a single argument.

The specified strings are used to generate all possible combinations with the optional surrounding preambles and postscripts. The preamble is prefixed to each string contained within the braces, and the postscript is then appended to each resulting string, expanding left to right.

The echo statement below shows how the brace expansion works. In this, "last" is the preamble, {mce,boot,xorg} are the brace expansions, and ".log" is the postscript.

```
$ echo last{mce,boot,xorg}.log
lastmce.log lastboot.log lastxorg.log
```

Using this concept, you can open three files (lastmce.log, lastboot.log, and lastxorg.log) in vim editor as shown below.

```
$ vi last{mce,boot,xorg}.log
```

Use brace expansion to supply the path to a list of files:

You can use this concept, anytime you are doing something common a group of files located in a common directory.

The following are same:

```
$ ls -l /etc/passwd /etc/group
$ ls -l /etc/{passwd,group}
```

The following are same:

```
vi /etc/passwd /etc/group
vi /etc/{passwd,group}
```

Backup example using brace expansion

```
$ cat brace-backup.sh
set -x # expand the commands
da=`date +%F`
cp $da.log{,.bak}

$ ./brace-backup.sh
++ date +%F
+ da=2011-08-06
+ cp 2011-08-06.log 2011-08-06.log.bak
```

Note: In this example, "cp $da.log{,.bak}" is expanded to "cp $da.log $da.log.bak"

The above backup script copies the current date log file with the extension .bak. The empty first element in the braces produces only the preamble as a result.

Use the same concept to take a quick backup of your files. For example, before modifying a configuration file you might take a backup of it with the extension .bak as shown below.

```
cp httpd.conf httpd.conf.bak
```

Instead of the above, you can use brace expansion:

```
cp httpd.conf{,.bak}
```

Restore example using brace expansion

```
$ cat brace-restore.sh
set -x # expand the commands
da=`date +%F`
cp $da.log{.bak,}

$ ./brace-restore.sh
++ date +%F
+ da=2011-08-06
+ cp 2011-08-06.log.bak 2011-08-06.log
```

Note: In the above example, "cp $da.log{.bak,}" is expanded to "cp $da.log.bak $da.log"

In the above restore script, in cp $da.log{.bak,} the first element in the parameter is .bak, and the second element is empty.

Brace Expansion without preamble and postscript

If there is no preamble and postscript, bash just expands the elements given in the braces.

```
$ cat brace-expand.sh
echo {oct,hex,dec,bin}

$ ./brace-expand.sh
oct hex dec bin
```

The result is just a space separated list of the given strings.

Brace expansion for Integer and character sequences

You can also expand integer or character sequences:

```
$ cat brace-sequence.sh
```

```
cat /var/log/messages.{1..3}
echo {a..f}{1..9}.txt


$ ./brace-sequence.sh
Aug 6 01:18:29 x3 ntpd[2413]: time reset -0.132703 s
Aug 6 01:22:38 x3 ntpd[2413]: synchronized to LOCAL(0),
stratum 10
Aug 6 01:23:44 x3 ntpd[2413]: synchronized to
Aug 6 01:47:48 x3 dhclient: DHCPREQUEST on eth0
Aug 6 01:47:48 x3 dhclient: DHCPACK from 23.42.38.201
..
..
a1.txt a2.txt a3.txt a4.txt b1.txt b2.txt b3.txt b4.txt
c1.txt c2.txt c3.txt c4.txt
```

The first cat command expands messages.1,messages.2 and messages.3 and displays the content. In the echo statement, character and integer sequences are combined and printed.

Sequences with increment value (new in bash 4)

You can specify the increment value used to generate the sequences.

Syntax:

```
<start>..<end>..<incr>
```

incr is numeric. You can use a negative integer, but the correct sign is deduced from the order of start and end.

Example:

```
$ echo /var/log/messages.{1..7..2}
/var/log/messages.1 /var/log/messages.3
/var/log/messages.5 /var/log/messages.7
```

Variables are not expanded by Brace Expansion

Caution: Brace expansion does not expand bash variables, because the brace expansion is the very first step bash performs. Variables are expanded later.

In the following example, the first for statement doesn't use variables in the brace expansion, and prints the values correctly. The second for statement uses variables in the brace expansion, and doesn't print the values as desired.

```
$ cat brace-var-sequence.sh
# Print 1 to 4 using sequences.
for i in {1..4}
do
  echo $i
done
start=1
end=4
# Print 1 to 4 using through variables
echo "Sequences expressed using variables"
for i in {$start..$end}
do
  echo $i
done

$ ./brace-var-sequence.sh
1
2
3
4
Sequences expressed using variables
{1..4}
```

Zero padding (new in bash 4)

The following expands the sequence without any leading zeroes.

```
$ echo {1..5}
1 2 3 4 5
```

The following expands the sequence with a single leading zero.

```
$ echo {01..5}
01 02 03 04 05
```

The following expands the sequence with double leading zeroes.

```
$ echo {001..5}
001 002 003 004 005
```

The following expands it with double leading zeroes and an increment of 2.

```
$ echo {001..5..2}
001 003 005
```

92. Tilde Expansion

The Bash shell provides some variables that are prefixed with ~ (named tilde) that are expanded to standard directory names.

Home Directories

Tilde(~) as a separate word expands to $HOME if it is defined. If $HOME is not defined, then it expands to the home directory of the current user.

In the example below, the value of the HOME variable is /home/oracle. So, cd ~ changes the current directory to /home/oracle.

```
$ pwd
/tmp
$ echo $HOME
/home/oracle

$ cd ~
$ pwd
/home/oracle
```

For tilde expansion, the HOME environment variable overrides the real home directory, as shown in the example below.

```
$ export HOME=/sbin

$ cd ~
$ pwd
/sbin

$ unset HOME

$ cd ~
$ pwd
/home/oracle
```

Home directory of the given user

~user refers to the home directory of a specific user. For example, ~oracle refers to the home directory of the user oracle.

If the given user name does not exist then it doesn't expand. In the following example, ~oracle is a valid user, and it expanded to /home/oracle. ~ora is not a valid user, and it didn't expand.

```
$ echo ~oracle
```

```
/home/oracle


$ echo ~ora
~ora
```

Working Directories

- ~+ expands to the value of the PWD variable which holds the current working directory.
- ~- expands to the value of OLDPWD variable, which holds the previous working directory. If OLDPWD is unset, ~- is not expanded.

The following example compares copies of a file:

```
$ cat tilde-compare.sh
set -x
cd /var/opt/gg
if [ -f gg.c ]
then
echo "File1 exists"
fi
cd /var/opt1/gg
if [ -f gg.c ]
then
echo "File2 exists"
cmp ~+/gg.c ~-/gg.c
fi


$ ./tilde-compare.sh
+ cd /var/opt/gg
+ '[' -f gg.c ']'
+ echo 'File1 exists'
```

```
File1 exists
+ cd /var/opt1/gg
+ '[' -f gg.c ']'
+ echo 'File2 exists'
File2 exists
+ cmp /var/opt1/gg/gg.c /var/opt/gg/gg.c
cmp: EOF on /var/opt1/gg/gg.c
```

- ~+/gg.c expands to /var/opt1/gg/gg.c
- ~-/gg.c expands to /var/opt/gg/gg.c

Expansion for Directories in Stack

Earlier we discussed directory stacks and the use of the dirs, pushd, and popd commands. Tilde expansion can also be used with the directory stack.

- ~+N expands the Nth directory in the directory stack (counting from the left of the list printed by dirs when invoked without options, starting with zero).
- ~-N Expands the Nth directory in the directory stack (counting from the end of the list printed by dirs when invoked without options, starting with zero).

In the following example, the directory stack contains 4 directories. ~+2 gives you the directory path for the second position from the left starting with zero.

```
$ dirs -v
 0  /sbin
 1  /var/opt/midas
 2  /var/opt/GG/bin
 3  /root

$ cd ~+2
$ pwd
```

```
/var/opt/GG/bin
```

The top of the stack (zero position) will always contain the current directory. After the above execution, the following is the content of the directory stack.

```
$  dirs -v
 0  /var/opt/GG/bin
 1  /var/opt/midas
 2  /var/opt/GG/bin
 3  /root
```

Display Nth directory from right using ~-

Changing “cd ~+2” to “cd ~-2” in the above example would cd to the second directory from the bottom (/var/opt/midas) instead of the second from the top (/var/opt/GG/bin).

93. Globbing (Pattern Matching)

Regular expressions are used by various utilities, for example grep, sed and awk. Bash doesn't use regular expressions for filename expansion. It uses globbing.

Filename globbing recognizes *, ?, and characters enclosed between [ ]

- ***** matches one or more characters. This also matches the null string. Please note that * doesn't match the hidden files that starts with a . (period)
- **?** matches a single character
- **[abcd]** matches any of the characters enclosed between [ ]. i.e a, or b, or c, or d.
- **[a-z]** Hyphen in the [ ] matches a range. For example, [a-z] matches lower case alphabets, a through z. [0-9] matches numbers, 0 through 9.

The following are few examples of filename globbing.

List all the files under /etc/ that end with .conf

```
ls -l /etc/*.conf
```

List all files under /etc/ that have exactly three characters followed by .conf

```
ls -l /etc/???.conf
```

List all files under /etc/ that start with a, b, c, or d and end with .conf

```
ls -l /etc/[abcd]*.conf
```

You can specify multiple ranges inside [ ]. List all files under /etc/ that start with a, b, c, d, x, y, or z and end with .conf

```
ls -l /etc/[a-dx-z]*.conf
```

If you want to match a dash in the filename, you must specify it as either the first or last character of the glob.

For example, suppose you want to view files that contain a, b, -, x, or y. The following doesn't work as expected, i.e. - is treated as a range specifier:

```
ls -l /etc/[ab-xy]*.conf
```

To get the result you intended, put the - in the front or back of the [ ].

```
ls -l /etc/[-abxy]*.conf
```

Negate the globbing with ^ or !. The following example lists all *.conf files under /etc that do not start with one of the characters a, or b, or c, or d.

```
ls -l /etc/[^abcd]*.conf


ls -l /etc/[!abcd]*.conf
```

The following example shows how you can use globbing in a bash for loop.

```
$ cat globbing.sh
for i in /etc/[abcd]*.conf
do
 echo $i
done
```

Character class matching

Match a POSIX standard character class using:

```
[[:class:]]
```

class could be any one of the following:

- alnum
- alpha
- ascii
- blank
- cntrl
- digit
- graph
- lower
- print
- punct
- space
- upper
- word
- xdigit

Example: List all the files and directories that start with a number

```
$ ls -ld [[:digit:]]*
```

List all the files and directories that start with a lower case alpha character

```
ls -ld [[:lower:]]*
```

94. Extended Globbing

In addition to the standard pattern matching provided by globbing, extended globbing allows the following pattern match options.

- !(pattern): Matches everything except the pattern
- *(pattern): Zero or many occurrences of the pattern is matched
- +(pattern): One or many occurrences of the pattern is matched
- ?(pattern): Zero or exactly one occurrence of the pattern is matched
- @(pattern): Matches any one of the pattern

Extended globbing option is controlled by the extglob shell option.

By default extglob is disabled as shown below.

```
$ shopt extglob
extglob         off
```

Enable the extglob shell option as shown below.

```
$ shopt -s extglob

$ shopt extglob
extglob         on
```

Display all files except *.sh

```
$ ls !(*.sh)
newfile1.txt  newfile2.txt  newfile.txt  states.txt
```

95. Regular Expressions in Bash

As we mentioned, regular expressions cannot be used on filename expansion (or) filename matching. Bash uses only globbing for filename matching.

However regular expressions can be used on strings using the =~ operator inside the [[ ]] brackets.

Syntax:

```
if [[ string =~ pattern ]]
then
 echo "String matched the pattern"
fi
```

Note: The string is compared with the given reg-ex pattern.

The following are some of the regular expression operators:

- ^ beginning of the string
- $ end of the string
- . matches a single character
- * matches zero or more occurrences of the previous character
- + matches one or more occurrence of the previous character
- ? matches zero or one occurrences of the previous character
- {m} A Regular expression followed by {m} matches exactly m occurrences of the preceding expression

- {m,n} A regular expression followed by {m,n} indicates that the preceding item must match at least m times, but not more than n times.
- \b matches a word boundary. \b matches any character(s) at the beginning (\bxx) and/or end (xx\b) of a word

The following simple example demonstrates the use of ~= to match regular expressions.

```
$ cat regex.sh
cities=("los angeles" "los altos" "las vegas" "san
francisco")
for city in "${cities[@]}"
do
  if [[ $city =~ ^los ]]; then
   echo "$city starts with los"
  fi
done

$ ./regex.sh
los angeles starts with los
los altos starts with los
```

96. Get the output of a Linux command

You can execute a linux command and get its output as part of a variable expansion using one of the following two methods:

```
$(unix-command)

`unix-command`
```

The following example shows how to use this in a shell script.

```
$ cat linux-commands.sh
datestring=$(date +"%d-%m-%Y")
echo $datestring
datestring=`date +"%Y%m%d"`
echo $datestring

$ ./linux-commands.sh
06-08-2011
20110806
```

When the output of a command is stored in a variable, trailing newlines are removed. However newlines embedded in the output will not be removed.

You can also nest multiple linux commands as shown in the next example.

```
$ cat linux-commands1.sh
datestring=$(echo day:$(date +"%d") month:$(date +"%m")
year:$(date +"%y"))
echo $datestring
datestring=`echo year:\`date +"%Y"\``
echo $datestring

$ ./linux-commands1.sh
day:06 month:08 year:11
year:2011
```

Note: When using the backtick method to nest multiple linux commands, you should use backslash for the inner backticks.

97. Parameter Expansion Basics :- := :+

${variable}

${variable} is the same as $variable.

When you refer to positional parameters above 9, you should use parenthesis after the $.

- $1 is same as ${1}
- ${10} to refer 10th positional parameter is correct
- $10 to refer 10th positional parameter is wrong.

Also, when you have some characters that follow immediately after the variable name, that are not part of the variable name, use {} to separate the variable name from the rest of the characters.

For example, ${shell}v4 will substitute the value of the $shell variable. If you remove the braces { } in this example, i.e. you use $shellv4, bash treats the entire string shellv4 as a variable name.

In the following example, #1 and #2 are the same, but #3 and #4 are different.

```
$ cat parameter-simple.sh
shell="bash"
echo "1. Shell is : $shell"
echo "2. Shell is : ${shell}"
echo "3. Shell is : $shellv4"
echo "4. Shell is : ${shell}v4"

$ ./parameter-simple.sh
Shell is : bash
Shell is : bash
Shell is :
```

```
Shell is : bashv4
```

${variable:-defaultvalue}

To test whether a parameter is unset (or null) use the default value option.

When the variable is not defined (i.e unset), or is defined but is null, this syntax returns the defaultvalue. Please note that this doesn't change the value of the variable. It just returns the defaultvalue, when the variable is not defined (or null).

When the variable fname has a value, ${fname:-Jason} returns the actual value of the fname variable, as shown in parameter-dash.sh below.

```
$ cat parameter-dash.sh
fname="John"
name=${fname:-Jason}
echo $name

$ ./parameter-dash.sh
John
```

When the variable fname is not defined, is defined but contains null, Jason is supplied instead. This is explained in parameter-dash1.sh below.

```
$ cat parameter-dash1.sh
name=${fname:-Jason}
echo $name
lname=
name=${lname:-Smith}
echo $name

$ ./parameter-dash1.sh
```

```
Jason
Smith
```

${variable-defaultvalue}

The difference between :- and - is the operation performed when the value of the variable is null.

- If the variable is null, - will not return the default value.
- If the variable is null, :- will return the default value.

Remember that :- and - do not change the value of the variable. They simply return the current value of the variable, or the default value if it is not set (or is null, for :-).

These default value constructs are typically used when you want to assign the value of a variable to another variable.

${variable:=defaultvalue}

This expression does change the value of the variable. When the parameter is unset (or not defined), or when the variable is defined but has a null value, this sets the defaultvalue to the variable.

```
$ cat parameter-equal.sh
echo ${fname:=Jason}
lname=Bourne
echo ${lname:=Smith}

$ ./parameter-equal.sh
Jason
Bourne
```

${variable=defaultvalue}

The difference between := and = is that if the value of the variable is null, = without the colon will not assign the default value to the variable.

The := method will assign the default value to the variable whenever it is either undefined or null. Most of the times this is what you need, so it is safe to use :- all the time.

${variable:+newvalue}

When the parameter has a value already, the newvalue is assigned to the variable; if not nothing happens.

:+ is exact opposite of :-

When the variable contains a value, this returns the newvalue.

Please note that this doesn't change the value of the variable, it just returns the newvalue, if the variable was defined earlier with some other value. When the variable doesn't contain a value (or was not defined earlier), this will return null.

In the example below, the first echo will return null. Since fname was not set previously, it will not use the alternatevalue. The second echo will return the newvalue, as lname was defined with a value before.

```
$ cat parameter-plus.sh
echo ${fname:+Jason}
lname=Bourne
echo ${lname:+Smith}
echo $lname

$ ./parameter-plus.sh
Smith
Bourne
```

${variable:?errormessage}

When the parameter is null or not set, the errormessage will be displayed and the shell script will terminate with the exit status 1.

```
$ cat parameter-question.sh
lname=Bourne
name=${lname:?"Variable lname is not defined"}
echo $lname
name=${fname:?"Variable fname is not defined"}
echo "finished"

$ ./parameter-question.sh
Bourne
./parameter-question.sh: line 5: fname: Variable fname is
not defined
```

In the above example, lname is displayed properly, because lname was defined with an value earlier. However fname displays the error message that we specify, because fname was not defined anywhere in the script.

98. Parameter References

! in front of a variable name that is enclosed in ${ } is used to expand the names of the variables that match that prefix pattern.

For example, if you have city1, city2, and city3 variables defined, ${!city*} expands to all the three variables city1 city2 city3. When the IFS variable is set, the expanded variables are separated by the first character in the IFS.

You can use this to define a pointer to a variable name:

```
$ cat parameter-reference.sh
city_in_california="Los Angeles"
echo "Value of city_in_california is: $city_in_california"
pointer=${!city*}
echo "Value of pointer is: $pointer"
var=${!pointer}
```

```
echo "Value of the variable that is referred by the
pointer is: $var"


$ ./parameter-reference.sh
Value of city_in_california is: Los Angeles
Value of pointer is: city_in_california
Value of the variable that is referred by the pointer is:
Los Angeles
```

The next example demonstrates the following:

- ${!city*} expands to city1 city2 city3 (when there is no IFS defined)
- ${!city*} expands to city1<city2<city3 (when the IFS is defined to "<^", it uses the first character of the IFS as separator)
- You can also loop through the ${!city*} in a for loop as shown here.
- When an array is used, it expands to the indexes of the array elements. ${!cities[*]} expands to 0<1<2 (with the IFS first character as separator).

```
$ cat parameter-reference1.sh
city1="Los Angeles"
city2="San Francisco"
city3="New York"
echo "1. \${!city*}: ${!city*}"
IFS="<^"
echo "2. \${!city*} [with IFS]: ${!city*}"
echo "3. Loop through the references"
for varname in ${!city*}
do
 echo $varname
done
```

```
cities=("Los Angeles" "San Francisco" "New York")
echo "4. Array Reference: ${!cities[*]}"
```

Execute the above program to view the output as shown below.

```
$ ./parameter-reference1.sh
1. ${!city*}: city1 city2 city3
2. ${!city*} [with IFS]: city1<city2<city3
3. Loop through the references
city1
city2
city3
4. Array Reference: 0<1<2
```

99. Case Conversion using Parameter Expansion

The following characters are used to control case conversion inside { }

Character	Description
^^	Convert all characters to uppercase
,,	Convert all characters to lowercase
~~	Invert case for all characters (i.e lowercase becomes uppercase, and uppercase becomes lowercase)
^	Convert only first character to upper case
,	Convert only first character to lower case
~	Invert case for first character in every word

The following example demonstrates the usage of case conversion options:

```
$ cat parameter-case.sh
city="los angeles"
echo "city: $city"
echo "1. All Upper Case \${city^^}: ${city^^}"
echo "2. Invert Case for all characters \${city~~}: $
{city~~}"
echo "3. Only 1st character to Upper Case \${city^}: $
{city^}"
echo "4. Invert case for 1st character in every word \$
{city~}: ${city~}"
city="LOS ANGELES"
echo ""
echo "city: $city"
echo "5. All Lower Case \${city,,}: ${city,,}"
echo "6. Invert Case for all characters \${city~~}: $
{city~~}"
echo "7. Only 1st character to Lower Case \${city,}: $
{city,}"
echo "8. Invert case for 1st character in every word \$
{city~}: ${city~}"
```

Execute the above program to view the output as shown below.

```
$ ./parameter-case.sh
city: los angeles
1. All Upper Case ${city^^}: LOS ANGELES
2. Invert Case for all characters ${city~~}: LOS ANGELES
3. Only 1st character to Upper Case ${city^}: Los angeles
4. Invert case for 1st character in every word ${city~}:
Los Angeles
city: LOS ANGELES
5. All Lower Case ${city,,}: los angeles
```

```
6. Invert Case for all characters ${city~~}: los angeles
7. Only 1st character to Lower Case ${city,}: lOS ANGELES
8. Invert case for 1st character in every word ${city~}:
lOS aNGELES
```

100. Substring Using Parameter Expansion ${variable:start:length}

Substring syntax:

```
{variable:start:length}
```

- variable: The variable containing the substring.
- start: The starting position from which characters should be extracted.
- length: This is optional. The total number of characters to be extracted.

For example:

```
city="Los Angeles"
echo "${city:4}"
```

The above example will display "Angeles", as is starts substring extraction from position 4, and extracts the rest of the string.

Please note that the position starts counting from 0 (and not from 1), i.e. the first character "L" is represented by starting position 0.

```
city="Los Angeles"
echo "${city:0:3}"
```

The above example will display "Los", as it starts the substring extraction from position 0 (the first character), and extracts 3 characters from there.

101. Find and Replace using Parameter Expansion

Find and Replace Syntax:

```
${variable/originalstring/replacementstring}
```

- variable: The value of this variable will be used as the input for for find and replace.
- originalstring: The original string (the pattern) to be found in the string.
- replacementstring: The string to be used to replace the original string.

The following example uses find/replace to replace "Angeles" with "Altos" and display "Los Altos"

```
$ city="Los Angeles"
$ echo "${city/Angeles/Altos}"
Los Altos
```

Note: In sed (and inside vi), you use s/originalstring/replacementstring/ pattern, using the front slash to separate the original string and the replacement string. The find and replace syntax used in this parameter substitution is similar.

Global Find and Replace

Syntax:

```
${variable//originalstring/replacementstring}
```

Just use two / after the variable to do a global find and replace.

In following example, all the occurrences of lower case “e” are replaced with upper case “E”. This will display "Los AngElEs".

```
$ echo "${city//e/E}"
```

```
Los AngElEs
```

Delete Pattern

Syntax:

```
${variable/deletepattern/}
```

To delete a pattern from a string, use the same form as find and replace, but don't give any replacementstring. In this case, the / following the originalstring is optional as shown in the next example.

Remove"Los " from the string "Los Angeles" and display only "Angeles":

```
$ echo "${city/Los /}"
Angeles


$ echo "${city/Los }"
Angeles
```

Prefix find and replace

Syntax:

```
${variable/#pattern/replacement}
```

This syntax will search for the given "Pattern" in the prefix of the string. If the Pattern is found (in the prefix), it will be replaced with the replacement string.

In the following example, the variable “cities” has two occurrences of "Los" (one in the prefix and one in the middle). Only the "Los" in the prefix is replaced.

```
$ cities="Los Angeles CA,Los Altos CA"
$ echo "${cities/#Los/City of Los}"
City of Los Angeles CA,Los Altos CA
```

Suffix find and replace

Syntax:

```
${variable/%pattern/replacement}
```

This syntax will search for the given "Pattern" in the suffix of the string. If the Pattern is found (in the suffix), it will be replaced with the replacement string.

In the following example, the variable “cities” has two occurrences of "CA" (one in the suffix and one in the middle). Only the "CA" in the suffix is replaced.

```
$ echo "${cities/%CA/California}"
Los Angeles CA,Los Altos California
```

You can use suffix find and replacement to remove filename extensions:

```
$ file="mydocument.txt"


$ echo ${file%.txt}
mydocument
```

You can use suffix find and replacement to show a different filename extension:

```
$ echo ${file%.txt}.doc
mydocument.doc
```

The following script browses the *.txt files in the current directory and removes the extensions:

```
$ cat parameter-remove-ext.sh
for file in *.txt
do
 mv $file ${file%.txt}
done
```

The following script renames the extensions to .doc, i.e. a filename called mydocument.txt would be renamed it to mydocument.doc by this script.

```
$ cat parameter-rename-ext.sh
for file in *.txt
do
 mv $file ${file%.txt}.doc
done
```

All of the above find and replace examples are combined in this shell script.

```
$ cat parameter-find-replace.sh
city="Los Angeles"
echo "1. Find and Replace - Basic"
echo "${city/Angeles/Altos}"
echo "2. Find and Replace - Global"
echo "${city//e/E}"
echo "3. Find and Replace - Delete Pattern"
echo "${city/Los /}"
echo "${city/Los }"
echo "4. Find and Replace - Prefix Only"
cities="Los Angeles CA,Los Altos CA"
echo "${cities/#Los/City of Los}"
echo "5. Find and Replace - Suffix Only"
echo "${cities/%CA/California}"
```

Execute the above shell script to view the output as shown below.

```
$ ./parameter-find-replace.sh
1. Find and Replace - Basic
Los Altos
2. Find and Replace - Global
Los AngElEs
3. Find and Replace - Delete Pattern
Angeles
Angeles
4. Find and Replace - Prefix Only
City of Los Angeles CA,Los Altos CA
5. Find and Replace - Suffix Only
Los Angeles CA,Los Altos California
```

Thank You

I hope you found the ***Bash 101 Hacks*** eBook helpful.

I sincerely appreciate all the support given by you and other regular readers of my thegeekstuff.com blog.

You have encouraged me in more ways than you know.

Please use the contact form thegeekstuff.com/contact/ to send me your suggestions, or feedback, or questions on this eBook.

Ramesh Natarajan

ramesh@thegeekstuff.com