



SCM/Build & Release Management

Chiranjeevi Patel

Contents

Linux basics:	2
File Handling.....	2
Text Processing	3
Linux File Permissions	4
System Administration.....	4
Process Management	5
Archival	5
Network	6
File Systems.....	6
Advanced Commands	7
Editor commands	7
Process Management	8
Exercise 1:	8
Exercise 2:	9
Shell Scripting:.....	11
Introduction	11
Shell Scripting Basics	12
Variables.....	13
Command Line Arguments	14
Command Substitution	15
Arithmetic Expansion	15
Control Constructs	16
Functions.....	20
Variable Scope.....	21
Exercise:	24
.....	Error! Bookmark not defined.

Linux basics:

Basic Linux Commands

- File Handling
- Text Processing
- System Administration
- Process Management
- Archival
- Network
- File Systems
- Advanced Commands Cat

File Handling (contd...)

- pwd - print name of current working directory

Usage: pwd

- vim – Vi Improved, a programmers text editor

Usage: vim [OPTION] [file]...

eg. vim file1.txtMkdir

- cp – copy files and directories

Usage: cp [OPTION]... SOURCE DEST

eg. cp sample.txt sample_copy.txt

cp sample_copy.txt target_dir

- mv – move (rename) files

Usage: mv [OPTION]... SOURCE DEST

eg. mv source.txt target_dir

mv old.txt new.txt

- rm remove

files or directories

Usage: rm [OPTION]... FILE...

eg. rm file1.txt , rm rf

some_dir

- find – search for files in a directory hierarchy

Usage: find [OPTION] [path] [pattern]

eg. find file1.txt, find name

file1.txt

- history – prints recently used commands

Usage: history

- Pattern

A Pattern is an expression that describes a set of strings which is used to give a concise description of a set, without having to list all elements.

eg. ab*cd matches anything that starts with ab and

ends with cd etc.

ls *.txt – prints all text files

Text Processing

- cat – concatenate files and print on the standard output

Usage: cat [OPTION] [FILE]...

eg. cat file1.txt file2.txt

cat n

file1.txt

- echo – display a line of text

Usage: echo [OPTION] [string] ...

eg. echo I love India

echo \$HOME

- grep print lines matching a pattern

Usage: grep [OPTION] PATTERN [FILE]...

eg. grep i

apple sample.txt

- wc print the number of newlines, words, and bytes in files

Usage: wc [OPTION]... [FILE]...

eg. wc file1.txt

wc L

file1.txt

- sort – sort lines of text files

Usage: sort [OPTION]... [FILE]...

eg. sort file1.txt

sort -r file1.txt

Linux File Permissions

- 3 types of file permissions – read, write, execute
- 10 bit format from 'ls l' command

1 2 3 4 5 6 7 8 9 1

Filetype owner group others

eg. drwxrw- r—means owner has all three permissions, group has read and write, others have only read permission

read permission – 4, write – 2, execute 1

eg. rwxrwr=764

673 = rwrwxwx

System Administration

- chmod – change file access permissions

Usage: chmod [OPTION] [MODE] [FILE]

eg. chmod 744 calculate.sh

- chown – change file owner and group

Usage: chown [OPTION]... OWNER[:[GROUP]] FILE...

eg. chown remo myfile.txt

- su – change user ID or become superuser

Usage: su [OPTION] [LOGIN]

eg. su remo, su

- passwd – update a user's authentication tokens(s)

Usage: passwd [OPTION]

eg. passwd

- who – show who is logged on

Usage: who [OPTION]

eg. who , who -b, who -q

Process Management

- ps – report a snapshot of the current processes

Usage: ps [OPTION]

eg. ps, ps -el

- kill – to kill a process(using signal mechanism)

Usage: kill [OPTION] pid

eg. kill -9 2275

Archival

- tar – to archive a file

Usage: tar [OPTION] DEST SOURCE

eg. tar cvf /home/archive.tar /home/original

tar xvf /home/archive.tar

- zip – package and compress (archive) files

Usage: zip [OPTION] DEST SOURCE

eg. zip original.zip original

- unzip – list, test and extract compressed files in a ZIP archive

Usage: unzip filename

eg. unzip original.zip

Network

- ssh – SSH client (remote login program)

“ssh is a program for logging into a remote machine and for executing commands on a remote machine”

Usage: ssh [options] [user]@hostname

eg. ssh X

guest@10.105.11.20

- scp – secure copy (remote file copy program)

“scp copies files between hosts on a network”

Usage: scp [options] [[user]@host1:file1] [[user]@host2:file2]

eg. scp file1.txt guest@10.105.11.20:~/Desktop/

File Systems

- mount – mount a file system

Usage: mount t

type device dir

eg. mount /dev/sda5 /media/target

- umount – unmount file systems

Usage: umount [OPTIONS] dir | device...

eg. umount /media/target

- du – estimate file space usage

Usage: du [OPTION]... [FILE]...

eg. du

- df – report filesystem disk space usage

Usage: df [OPTION]... [FILE]...

eg. df

- quota – display disk usage and limits

Usage: quota [OPTION]

eg. quota v

Advanced Commands

- reboot – reboot the system

Usage: reboot [OPTION]

eg. reboot

- poweroff – power off the system

Usage: poweroff [OPTION]

eg. Poweroff

- sed stream

editor for filtering and transforming text

Usage: sed [OPTION] [inputfile]...

eg. sed 's/love/hate/g' loveletter.txt

- awk pattern

scanning and processing language

eg. awk F:

'{ print \$1 }' sample_awk.txt

Editor commands

- vim – Vi Improved, a programmers text editor

Usage: vim [OPTION] [file]...

eg. vi hello.c

Process Management(contd...)

- bg – make a foreground process to run in background

Usage: type 'ctrl+z' and then 'bg <job id>'

- fg – to make background process as foreground process

Usage: fg [jobid]

- jobs – displays the names and ids of background jobs

Usage: jobs

Exercise 1:

1. After you have logged in, create in your home directory, a practice directory called Exercises. Remember that UNIX is case sensitive!
\$ mkdir exercises
2. Change into that directory. Create several other directories called Question1, Question2, Bearing in mind that leaving a space in a directory name is possible, but undesirable, and also that UNIX is case-sensitive. \$ mkdir question1
\$ mkdir question2

\$ mkdir question3
3. Create a .txt as name-profile.txt file using vi. Write a small essay about yourself in the file regarding your background, hobbies and professional goals etc. Have at least two paragraphs and each paragraph having at least 10 lines. \$ vi name-profile.txt
4. Make a second copy of the file in the directory called Question2
\$ cp name-profile.txt question2/details
5. Change the name of the file in the Question2 directory to 2nd-name-profile.txt
\$ mv name-profile.txt 2nd-name-profile.txt
6. Using the chmod command, change the permissions on the file in the Question1 directory (and the directory itself) so that anyone logged into Ranger can read the file (but not alter it). The permissions for the directory will read: rwxr-xr-x and the file permissions will read rw-r--r-- when you are finished with this. You may also need to change the permissions on files and directories above the Question1 directory to accomplish this. You will almost certainly need to read the man page for chmod in order to do this. \$ chmod 744 quant

Exercise 2:

1. Check which directory you are currently in. If necessary, move to your home directory. (Remember: cd will do this from anywhere).pwd
2. Move to the root directory. ("Move to..." means "change your current working directory to...". It is useful to picture the process as movement around the tree structure.) cd
3. Work your way down one directory at a time to your home directory. cd
4. Experiment with using relative and absolute pathnames; show how the two can produce the same results. \$ Cd test/har/put, (\$ cd test

\$ cd har

\$ cd put)

5. Explore your systems filestore. Try to get into the home directory of someone else yo know! (You may not be able to view their files.) cd .. ls -l
6. Display your current working directory using the pwd command.pwd
7. Make a directory called exercises. Mkdir exercises
8. Change your directory to the directory exercises. Display the current working directory. Cd exercises
9. Return to your home directory. Cd
10. List the contents of your directory. Use the -l, -a and -F options and compare the output. Ls -l, Ls -a Ls -F
11. Change your directory to the directory exercises. Create a file called example1 using the cat command containing the following text:
12. water, water everywhere
13. and all the boards did shrink;
14. water, water everywhere,
15. Nor drop to drink \$ mkdir exercises \$ cd >example1
16. List the contents of your directory. Use the -l option to obtain a long listing. Ls -l
17. Display a list of logged on users. who
18. Obtain further information for a particular user using the finger command.

\$ finger username

19. Use the man command to obtain further information on the finger command.

\$ man finger

20. Use the man -k command to find what manual entries there are related to passwords. \$ man -k passwd
21. Use the grep command to search the file example1 for occurrences of the string 'water'. \$ grep 'water' example1
22. Use man and the keyword option to find out more information on communications and e-mail in Unix. \$ man mail
23. Print out a file on paper. \$ lpr example1

24. Send a message to another user on your Unix system, and get them to reply.

```
$ talk username
```

25. Create a small text file and send it to another user.

26. When you receive a message, save it to a file other than your mailbox. (Remember you can always send yourself a message if you don't have one.)

27. Send a message to a user on a different computer system. \$ write username

28. Send a note to your course tutor telling him that you can use mail now.

29. Try to move to the home directory of someone else in your group. There are several ways to do this, and you may find that you are not permitted to enter certain directories. See what files they have, and what the file permissions are. (Remember that you can protect your own files from prying eyes, or from interference.)

30. Try to copy a file from another user's directory to your own.

31. Set permissions on all of your files and directories to those that you want. You may want to give read permission on some of your files and directories to members of your group. \$ chmod g+r example1

32. Put a listing of the files in your directory into a file called filelist. (Then delete it!) \$ ls | cat >filelist

33. Create a text file containing a short story, then use the spell program to check the spelling of the words in the file. \$ spell filename

34. Redirect the output of the spell program to a file called errors.

```
$ spell filename | cat >errors
```

35. Type the command ls -l and examine the format of the output. Pipe the output of the command ls -l to the word count program wc to obtain a count of the number of files in your directory. 3.

Use cut to strip away the reference material and leave just the text field. \$ ls -l | wc -l \$ ls -l | cut -d ' ' -f 9

36. Use tr to strip away any tags that are actually in the text (e.g. attached to the words), so that you are left with just the words. \$ tr -d '%' \$ tr -d '@' < filename >> filename

37. Create a file containing some lines that you think would match the regular expression: (^[0-9]{1,5}[a-zA-z]+\$)|none and some lines that you think would not match. Use egrep to see if your intuition is correct. \$ egrep '!' filename

```
$ grep -E '!' filename
```

38. Type umask 000 and then create a file called world.txt containing the words "hello world". Look at the permissions on the file. What's happened? Now type umask 022 and create a file called world2.txt. When might this feature be useful?

```
$ umask 000 $ vi world.txt $ ls -l $ umask 022 $ ls -l
```

39. Create a file called "hello.txt" in your home directory using the command cat -u > hello.txt. Ask your partner to change into your home directory and run tail -f hello.txt. Now type several lines into hello.txt. What appears on your partner's screen? \$ cat -u > hello.txt

40. Use find to display the names of all files in the /home subdirectory tree. Can you do this without displaying errors for files you can't read? `$ find exercises`
41. What does the cut command do? Can you use it together with w to produce a list of login names and CPU times corresponding to each active process? `$ w | cut -d " "`
42. Use ps, w and top to show all processes that are executing. `$ ps $ w $top`
43. Type mount (with no parameters) and try to interpret the output. `$ mount`
44. Startup a number of sleep 60 processes in the background, and terminate them all at the same time using the pkill command. `$ ps -f`

`sleep number & $ pkill -P processid`

Use `ps -aeH` to display the process hierarchy. Look for the init process. See if you can identify important system daemons. Can you also identify your shell and its subprocesses? `$ ps -aeH`

Shell Scripting:

Introduction

Unix uses shells to accept commands given by the user, there are quite a few different shells available. The most commonly used shells are SH(Bourne SHell) CSH(C SHell) and KSH(Korn SHell), most of the other shells you encounter will be variants of these shells and will share the same syntax, KSH is based on SH and so is BASH(Bourne again shell). TCSH(Extended C SHell) is based on CSH.

The various shells all have built in functions which allow for the creation of shell scripts, that is, the stringing together of shell commands and constructs to automate what can be automated in order to make life easier for the user.

With all these different shells available, what shell should we script in? That is debatable. For the purpose of this tutorial we will be using SH because it is practically guaranteed to be available on most Unix systems you will encounter or be supported by the SH based shells. Your default shell may not be SH. Fortunately we do not have to be using a specific shell in order to exploit its features because we can specify the shell we want to interpret our shell script within the script itself by including the following in the first line.

```
#!/path/to/shell
```

Usually anything following (#) is interpreted as a comment and ignored but if it occurs on the first line with a (!) following it is treated as being special and the filename following the (!) is considered to point to the location of the shell that should interpret the script.

When a script is "executed" it is being interpreted by an invocation of the shell that is running it. Hence the shell is said to be running non-interactively, when the shell is used "normally" it is said to be running interactively.

Note

There are many variations on the basic commands and extra information which is too specific to be mentioned in this short tutorial, you should read the man page for your shell to get a more comprehensive idea of the options available to you. This tutorial will concentrate on highlighting the most often used and useful commands and constructs.

Shell Scripting Basics

Command Redirection and Pipelines

By default a normal command accepts input from standard input, which we abbreviate to stdin, standard input is the command line in the form of arguments passed to the command. By default a normal command directs its output to standard output, which we abbreviate to stdout, standard output is usually the console display. For some commands this may be the desired action but other times we may wish to get our input for a command from somewhere other than stdin and direct our output to somewhere other than stdout. This is done by redirection:

- We use > to redirect stdout to a file, for instance, if we wanted to redirect a directory listing generated by the ls we could do the following:

```
ls > file
```

- We use < to specify that we want the command immediately before the redirection symbol to get its input from the source specified immediately after the symbol, for instance, we could redirect the input to grep (which searches for strings within files) so that it comes from a file like this:

```
grep searchterm < file
```

- We use >> to append stdout to a file, for instance, if we wanted to append the date to the end of a file we could redirect the output from date like so:

```
date >> file
```

- One can redirect standard error (stderr) to a file by using 2>, if we wanted to redirect the standard error from commandA to a file we would use:

```
commandA 2>
```

Pipelines are another form of redirection that are used to chain commands so that powerful composite commands can be constructed, the pipe symbol '|' takes the stdout from the command preceding it and redirects it to the command following it:

```
ls -l | grep searchword | sort -r
```

The example above first requests a long (-l) directory listing of the current directory using the ls command, the output from this is then piped to grep which filters out all the listings containing the searchword and then finally pipes this through to sort which then sorts the output in reverse (-r, sort then passes the output on normally to stdout.

Variables

When a script starts all environment variables are turned into shell variables. New variables can be instantiated like this:

```
name=value
```

You must do it exactly like that, with no spaces either side of the equals sign, the name must only be made up of alphabetic characters, numeric characters and underscores, it cannot begin with a numeric character. You should avoid using keywords like *for* or anything like that, the interpreter will let you use them but doing so can lead to obfuscated code ;)

Variables are referenced like this: \$name, here is an example:

```
#!/bin/sh
msg1=Hello
msg2=There!
echo $msg1 $msg2
```

This would echo "Hello There!" to the console display, if you want to assign a string to a variable and the string contains spaces you should enclose the string in double quotes ("), the double quotes tell the shell to take the contents literally and ignore keywords, however, a few keywords are still processed. You can still use \$ within a (") quoted string to include variables:

```
#!/bin/sh
msg1="one"
msg2="$msg1 two"
msg3="$msg2 three"
echo $msg3
```

Would echo "one two three" to the screen. The escape character can also be used within a double quoted section to output special characters, the escape character is "\", it outputs the character

immediately following it literally so `\\` would output `\`. A special case is when the escape character is followed by a newline, the shell ignores the newline character which allows the spreading of long commands that must be executed on a single line in reality over multiple lines within the script. The escape character can be used anywhere else too. Except within single quotes.

Surrounding anything within single quotes causes it to be treated as literal text that is it will be passed on exactly as intended, this can be useful for sending command sequences to other files in order to create new scripts because the text between the single quotes will remain untouched. For example:

```
#!/bin/sh
echo 'msg="Hello World!'" > hello
echo 'echo $msg' >> hello
chmod 700 hello
./hello
```

This would cause `"msg="Hello World!"` to be echoed and redirected to the file `hello`, `"echo $msg"` would then be echoed and redirected to the file `hello` but this time **appended to the end**. The `chmod` line changes the file permissions of `hello` so that we can execute it. The final line executes `hello` causing it output `"Hello World"`. If we had not used literal quotes we never would have had to use escape characters to ensure that `($)` and `(")` were echoed to the file, this makes the code a little clearer.

A variable may be referenced like so `${VARIABLENAME}`, this allows one to place characters immediately preceding the variable like `${VARIABLENAME}aaa` without the shell interpreting `aaa` as being part of the variable name.

Command Line Arguments

Command line arguments are treated as special variables within the script, the reason I am calling them variables is because they can be changed with the shift command. The command line arguments are enumerated in the following manner `$0`, `$1`, `$2`, `$3`, `$4`, `$5`, `$6`, `$7`, `$8` and `$9`. `$0` is special in that it corresponds to the name of the script itself. `$1` is the first argument, `$2` is the second argument and so on. To reference after the ninth argument you must enclose the number in brackets like this `${nn}`. You can use the shift command to shift the arguments 1 variable to the left so that `$2` becomes `$1`, `$1` becomes `$0` and so on, `$0` gets scrapped because it has nowhere to go, this can be useful to process all the arguments using a loop, using one variable to reference the first argument and shifting until you have exhausted the arguments list.

As well as the commandline arguments there are some special builtin variables:

- `$#` represents the parameter count. Useful for controlling loop constructs that need to process each parameter.
- `$@` expands to all the parameters separated by spaces. Useful for passing all the parameters to some other function or program.
- `$-` expands to the flags(options) the shell was invoked with. Useful for controlling program flow based on the flags set.

- \$\$ expands to the process id of the shell innovated to run the script. Useful for creating unique temporary filenames relative to this instantiation of the script.

Note

The commandline arguments will be referred to as parameters from now on, this is because SH also allows the definition of functions which can take parameters and when called the $$n$ family will be redefined, hence these variables are always parameters, its just that in the case of the parent script the parameters are passed via the command line. One exception is $$0$ which is always set to the name of the parent script regardless of whether it is inside a function or not.

Command Substitution

In the words of the SH manual “Command substitution allows the output of a command to be substituted in place of the command name itself”. There are two ways this can be done. The first is to enclose the command like this:

```
$(command)
```

The second is to enclose the command in back quotes like this:

```
`command`
```

The command will be executed in a sub-shell environment and the standard output of the shell will replace the command substitution when the command completes.

Arithmetic Expansion

Arithmetic expansion is also allowed and comes in the form:

```
$((expression))
```

The value of the expression will replace the substitution. Eg:

```
#!/bin/sh  
echo $((1 + 3 + 4))
```

Will echo "8" to stdout

Control Constructs

The flow of control within SH scripts is done via four main constructs; if...then...elif..else, do...while, for and case.

If..Then..Elif..Else

This construct takes the following generic form, The parts enclosed within ([]) and (()) are optional:

```
if list
then list
[elif list
then list] ...
[else list]
fi
```

When a Unix command exits it exits with what is known as an *exit status*, this indicates to anyone who wants to know the degree of success the command had in performing whatever task it was supposed to do, usually when a command executes without error it terminates with an exit status of zero. An exit status of some other value would indicate that some error had occurred, the details of which would be specific to the command. The commands' manual pages detail the exit status messages that they produce.

A list is defined in the SH as "a sequence of zero or more commands separated by newlines, semicolons, or ampersands, and optionally terminated by one of these three characters.", hence in the generic definition of the *if* above the list will determine which of the execution paths the script takes. For example, there is a command called test on Unix which evaluates an expression and if it evaluates to true will return zero and will return one otherwise, this is how we can test conditions in the *list* part(s) of the *if* construct because test is a command.

We do not actually have to type the test command directly into the *list* to use it, it can be implied by encasing the test case within ([]) and (()) characters, as illustrated by the following (silly) example:

```
#!/bin/sh
if [ "$1" = "1" ]
then
    echo "The first choice is nice"
elif [ "$1" = "2" ]
then
    echo "The second choice is just as nice"
elif [ "$1" = "3" ]
then
    echo "The third choice is excellent"
else
    echo "I see you were wise enough not to choose"
```

```
echo "You win"
fi
```

What this example does is compare the first parameter (command line argument in this case) with the strings "1", "2" and "3" using tests' (=) test which compares two strings for equality, if any of them match it prints out the corresponding message. If none of them match it prints out the final case. OK the example is silly and actually flawed (the user still wins even if they type in (4) or something) but it illustrates how the *if* statement works.

Notice that there are spaces between (if) and ([), ([) and the test and the test and (]), these spaces must be present otherwise the shell will complain. There must also be spaces between the operator and operands of the test otherwise it will not work properly. Notice how it starts with (if) and ends with (fi), also, notice how (then) is on a separate line to the test above it and that (else) does not require a (then) statement. You must construct this construct exactly like this for it to work properly.

It is also possible to integrate logical AND and OR into the testing, by using two tests separated by either "&&" or "||" respectively. For example we could replace the third test case in the example above with:

```
elif [ "$1" = "3" ] || [ "$1" = "4" ]
then echo "The third choi..."
```

The script would print out "The third choice is excellent" if the first parameter was either "3" OR "4". To illustrate the use of "&&" we could replace the third test case with:

```
elif [ "$1" = "3" ] || [ "$2" = "4" ]
then echo "The third choi..."
```

The script would print out "The third choice is excellent" if and only if the first parameter was "3" AND the second parameter was "4".

"&&" and "||" are both lazily evaluating which means that in the case of "&&", if the first test fails it won't bother evaluating the second because the list will only be true if they *BOTH* pass and since one has already failed there is no point wasting time evaluating the second. In the case of "||" if the first test passes it won't bother evaluating the second test because we only need *ONE* of the tests to pass for the whole list to pass. See the test manual page for the list of tests possible (other than the string equality test mentioned here).

Do...While

The *Do...While* takes the following generic form:

```
while list
```

```
do list
done
```

In the words of the SH manual "The two lists are executed repeatedly while the exit status of the first list is zero." there is a variation on this that uses *until* in place of *while* which executes *until* the exit status of the first list is zero. Here is an example use of the *while* statement:

```
#!/bin/sh
count=$1          # Initialise count to first parameter
while [ $count -gt 0 ]    # while count is greater than 0 do
do
    echo $count seconds till supper time!
    count=$(expr $count -1)    # decrement count by 1
    sleep 1          # sleep for a second using the Unix sleep command
done
echo Supper time!!, YEAH!!    # were finished
```

If called from the commandline with an argument of 4 this script will output

```
4 seconds till supper time!
3 seconds till supper time!
2 seconds till supper time!
1 seconds till supper time!
Supper time!!, YEAH!!
```

You can see that this time we have used the `-gt` of the test command implicitly called via '[' and ']', which stands for greater than. Pay careful attention to the formatting and spacing.

For

The syntax of the *for* command is:

```
for variable in word ...
do list
done
```

The SH manual states "The words are expanded, and then the list is executed repeatedly with the variable set to each word in turn.". A word is essentially some other variable that contains a list of values of some sort, the *for* construct assigns each of the values in the word to variable and then variable can be used within the body of the construct, upon completion of the body variable will be assigned the next value in word until there are no more values in word. An example should make this clearer:

```
#!/bin/sh
fruitlist="Apple Pear Tomato Peach Grape"
for fruit in $fruitlist
do
  if [ "$fruit" = "Tomato" ] || [ "$fruit" = "Peach" ]
  then
    echo "I like ${fruit}es"
  else
    echo "I like ${fruit}s"
  fi
done
```

In this example, *fruitlist* is word, *fruit* is variable and the body of the statement outputs how much this person loves various fruits but includes an *if...then..else* statement to deal with the correct addition of letters to describe the plural version of the fruit, notice that the variable *fruit* was expressed like *\${fruit}* because otherwise the shell would have interpreted the preceding letter(s) as being part of the variable and echoed nothing because we have not defined the variables *fruits* and *fruites* When executed this script will output:

```
I like Apples
I like Pears
I like Tomatoes
I like Peaches
I like Grapes
```

Within the *for* construct, *do* and *done* may be replaced by '{' and '}'. This is not allowed for *while*.

Case

The case construct has the following syntax:

```
case word in
pattern) list ;;
...
esac
```

An example of this should make things clearer:

```
#!/bin/sh
case $1
in
```

```
1) echo 'First Choice';;
2) echo 'Second Choice';;
*) echo 'Other Choice';;
esac
```

"1", "2" and "*" are patterns, word is compared to each pattern and if a match is found the body of the corresponding pattern is executed, we have used "*" to represent everything, since this is checked last we will still catch "1" and "2" because they are checked first. In our example word is "\$1", the first parameter, hence if the script is ran with the argument "1" it will output "First Choice", "2" "Second Choice" and anything else "Other Choice". In this example we compared against numbers (essentially still a string comparison however) but the pattern can be more complex, see the SH man page for more information.

Functions

The syntax of an SH function is defined as follows:

```
name ( ) command
```

It is usually laid out like this:

```
name() {
  commands
}
```

A function will return with a default exit status of zero, one can return different exit status' by using the notation *return exit status*. Variables can be defined locally within a function using *local name=value*. The example below shows the use of a user defined increment function:

Increment Function Example

```
#!/bin/sh

inc() {
    echo (($1 + $2))      # The increment is defined first so we can use it
                        # We echo the result of the first parameter plus the second
    parameter
}

# We check to see that all the command line arguments are present
if [ "$1" "" ] || [ "$2" = "" ] || [ "$3" = "" ]
then
    echo USAGE:
```

```

echo " counter startvalue incrementvalue endvalue"
else
count=$1          # Rename are variables with clearer names
value=$2
end=$3
while [ $count -lt $end ]  # Loop while count is less than end
do
    echo $count

    count=$(inc $count $value)  # Call increment with count and value as parameters
done                          # so that count is incremented by value
fi

```

```

inc() {
    echo $(( $1 + $2 ))
}

```

The function is defined and opened with `inc() {`, the line `echo $(($1 + $2))` uses the notation for arithmetic expression substitution which is `$((expression))` to enclose the expression, `$1 + $2` which adds the first and second parameters passed to the function together, the echo bit at the start echoes them to standard output, we can catch this value by assigning the function call to a variable, as is illustrated by the function call.

```
count=$(inc $count $value)
```

We use command substitution which substitutes the value of a command to substitute the value of the function call whereupon it is assigned to the `count` variable. The command within the command substitution block is `inc $count $value`, the last two values being its parameters. Which are then referenced from within the function using `$1` and `$2`. We could have used the other command substitution notation to call the function if we had wanted:

```
count=`inc $count $value`
```

We will show another quick example to illustrate the scope of variables:

Variable Scope, Example

```

#!/bin/sh
inc() {

    local value=4
    echo "value is $value within the function\n"
}

```

```
echo "\\b\\$1 is $1 within the function"
}
```

```
value=5
echo value is $value before the function
echo "\\$1 is $1 before the function"
echo
```

```
echo -e $(inc $value)
echo
echo value is $value after the function
echo "\\$1 is $1 after the function"
```

```
inc() {
    local value=4
    echo "value is $value within the function\\n"
    echo "\\b\\$1 is $1 within the function"
}
```

We assign a local value to the variable *value* of 4. The next three lines construct the the output we would like, remember that this is being echoed to some buffer and will be replace the function call with all the stuff that was passed to stdout within the function when the function exits. So the calling code will be replaced with whatever we direct to standard output within the function. The function is called like this:

```
echo -e $(inc $value)
```

We have passed the option `-e` to the `echo` command which causes it to process C-style backslash escape characters, so we can process any backslash escape characters which the string generated by the function call contains.

If we just echo the lines we want to be returned by the function it will not pass the newline character onto the buffer even if we explicitly include it with an escape character reference so what we do is actually include the sequence of characters that will produce a new line within the string so that when it is echoed by the calling code with the `-e` the escape characters will be processed and the newlines will be placed where we want them.

```
echo "value is $value within the function\\n"
```

Notice how the newline has been inserted with `\\n`, the first two backslashes indicate that we want to echo a backslash because within double quotes a backslash indicates to

process the next character literally, we have to do this because we are only between double quotes and not the literal-text single quotes. If we had used single quotes we would have had to echo the bit with the newline in separately from the bit that contains \$value otherwise \$value would not be expanded.

```
echo "\\b$1 is $1 within the function"
```

This is our second line, and is contained within double quotes so that the variable \$1 will be expanded, \\b is included so that \b will be placed in the echoed line and our calling code will process this as a backspace character. We have to do this because for some reason the shell prefixes a space to the second line if we do not, the backspace removes this space.

The output from this script called with 2 as the first argument is:

```
value is 5 before the function
$1 is 2 before the function

value is 4 within the function
$1 is 5 within the function

value is 5 after the function
$1 is 2 after the function
```

Tip

You can use ". DIRECTORY/common.sh" to import functions from a script called common.sh in DIRECTORY, a quick example is shown below, first is test.sh:

```
#!/bin/sh
. ./common.sh
if [ "$1" = "" ]; then
    echo USAGE:
    echo "sh test.sh type"
    exit
fi

if `validtype $1`; then
    echo Valid type
else
    echo Invalid type
fi
```


Here is common.sh:

```
#!/bin/sh
validtype() {
  if [ "$1" = "TYPEA" ] ||
    [ "$1" = "TYPEB" ] ||
    [ "$1" = "TYPEC" ] ||
    [ "$1" = "TYPED" ] ||
    [ "$1" = "TYPEE" ];
  then
    exit 0
  else
    exit 1
  fi
}
```

If

Exercise:

1. Write a shell script to find the sum of two numbers
2. Write a shell script to find the factorial of a given number.
3. Write a shell script to find whether two given strings are equal or not.
4. Write a shell script to accept a string which is less than 10 characters. If it is more than 10 reaccept the string.
5. Write a shell script to remove blank lines from a file.
6. Write a Shell Script and accept a file name, find out whether the accepted entry is file or directory. If it is a file display the contents. If it is a directory, use case structure to long listing files, creating and changing the permissions of a specified file.
7. Display all the users' information in a desired format. Accept a login name and display login name, UID, GID, Current shell, Home directory.
8. Accept book information like book id, author name and date of publication. Write a shell script to remove a particular info from database.
9. Write a shell script to find out the files accessed in last ten minutes and file modified in last ten minutes.
10. Write a shell script to locate all lines longer than 15 characters in a file.

DO NOT COPY (Chiranjeevi.aec@gmail.com)