2. Shell Scripts:

As the name indicates, a shell script is a text file that guides and coaxes the shell into performing a sequence of actions. A script can hold any series of commands (both internal shell commands and external Unix commands, with or without arguments), programs, or even other previously written scripts. Scripts can use redirections and pipes, allowing you to write your own filters.

There are, alas, small but nagging differences between the Bourne and C shells and these affect the shell script syntaxes and available features. First, the Bourne shell, /bin/sh, is the father of all the Unix shells and is available on most, if not all, Unix installations. Further, Bourne shell scripts will "run" under the C shell, /bin/csh. What happens, in fact, is that the C Shell can recognize a Bourne shell script and invoke the Bourne shell to execute it. Bourne shell scripts will also run under the Korn shell (/bin/ksh).

Suppose you regularly wanted to show the date, list the current logged-in users, and check their status. You could, of course, enter the commands date and who (with suitable arguments) each time, but why not make the shell do the work with a script file?

Simply create a file called dw (using vi or ed) containing the five following lines (the first three of which are optional, cosmetic, but highly recommended):

```
:
# @(#) dw - show date and users – SRC 01/01/2003
#
date
who -u
```

The colon is treated as a NOP (no operation) by the Bourne Shell-in nonjargon terms, the colon is successfully ignored by the Bourne Shell.

The C Shell does make use of the first character found in a script. An initial # (also known as the comment character) tells the C shell that the following text is intended as a C shell script. An initial symbol other than # will cause the C shell to invoke the Bourne shell to execute
(or try to execute) your script as a Bourne shell script.

The @(#) string is a useful trick used by the what command to extract the title and purpose of a script. what scans an argument file and displays information from any commented section containing the sequence @(#):

```
$ what dw

 dw:
 dw -- show date and users -- RSP 10/07/1996

$ _
```

For this reason, @(#) is often known as the what string.

The third line, a single #, is just for improved legibility. A blank line here would be equally acceptable. The fourth and fifth lines do the real work by invoking date and who.

Important : The two worst things you can do is to start a Bourne shell script with a # and omit the # at the start of a C shell script! For maximum portability, always start your Bourne scripts with a colon. C shells must start with a #.

There are several ways of exploiting the dw file. Since sh, the Bourne shell, is a program that accepts input from its standard input (usually your keyboard), you can simply redirect input as follows:

```
(1) $sh <dw
(2) $sh dw
```

Although these two examples give the same result, there is a technical difference. In the sh dw example, the shell retains your keyboard as standard input; with sh<dw, the standard input has been redirected to

the file dw. dw is not yet a proper shell script. A shell script, by definition, is an executable text file that can be run like a normal command. Before you (or anyone else) can run your new dw script as a regular command you must make the file executable. You'll recall that the usual default permissions for the files you create are simply read/write for you, the owner. The easiest way to add execution permission for you as user-owner is

```
$chmod u+x dw
```

You may wish to extend execution rights to your group by changing the first chmod argument from u+x to ug+x. Or you can be generous and give everyone execution rights (whether they want them or not) by using a+x.

dw is a new command that you've added to the hundreds that came from your Unix vendor. Since dw acts like a normal command, you can use all the tricks you've learned that are appropriate for a command that sends data to the standard output. for ex. $ dw>datewho.dat

You could even generate a background process with

```
$dw >datewho.dat &
44
```

The display of a process id (44 in my example) indicates that the & as worked in the usual way.

You can also filter the results of dw as follows :

```
$ dw | grep 'rakesh'
rakesh tty06 Nov 13 12:12 0:03 170
$_
```

Here, we use grep to search the output of dw for occurrences of the string "rakesh". grep means global-find regular expressions and print.

```
$ dw | grep 'Nov 13'
```

Yet another way of invoking dw exists using the Bourne shell dot command:

```
$ . dw
```

Note : The dot command is available in the Bourne and Korn shells, but not in the C shell.

Here the login shell itselt reads and executes the commands found in the file dw - there is no subshell involved. Once again, the displayed results are the same as for sh <dw, sh dw and dw. Briefly, when a subshell runs a command, any changes that the command makes to your environment (variable assignments, changes of directory and so on) affect only that subshell. When you return to the parent shell, the original enviornments is restored. Since the dot command executes its argument command in the current shell, any changes made are retained when the command exists.

As an illustration, your .profile file is (almost) a normal shell script (it has one nonstandard quirk: it can be run without execution permission) that is executed whenever you log in. You can run .profile with the dot command, you can immediately reinitialize the environment variables in your current shell without having to log out and in again:

```
$ . .profile
```

Shell Script Summary :

Let's pause to summarize the six basic shell-scripting steps :

a. Plan the command sequences and test them on the keyboard.
b. Choose a good, nonclashing name for your script.
c. Create the script file with your favourite editor.
d. Start the file with a colon and add pithy comments.
e. Make the file executable with chmod.
f. If necessary, mv the script to its proper directory.

3. Script Debugging Tips

(a) If you get a "command not found" or similar message, use pwd and ls to check that you are still in your home directory and that dw was created there.

(b) If you get an "executable permission denied" message, perhaps you forgot to chmod.

(c) Less common, but worth keeping in mind, is that another command of the same name exists in a directory listed ahead of your current directory in your PATH variable. Check your PATH with

echo $PATH.

(d) For script testing, it is useful to have both the current and home directories in your PATH variable. Remeber that date and who are in the /bin directory, so /bin must be in your PATH. Finally, make sure that dw contains the correct text. Recall that even if your PATH does not contain the directory holding a particular command (or script), you can still run it (permissions permitting!) by using the full path name /usr/stan/dw. The general rule is that when you invoke a command with a prefixed path starting with /, the shell does not bother to check the paths set in the PATH variable.

(e) For more advanced debugging, you can include the command "set -v" in your script. This turns on the verbose mode in which each line of the script is echoed on the screen as it is read by the shell. The feature is turned off with "set +v"

The verbose mode can also be switched on with "$ sh -v dw" which saves your editing the dw file. You can also "set -xv". This reduces the verbosity by displaying on the executable lines-comments are not displayed.

4. Shell Variables

Shell variables (sometimes called shell parameters) play a vital role in shell programming. Shell variables are rather like the variables used in conventional programming languages such as C. Variable names with capital letters are usually reserved for standard variables with predetermined Unix meanings.

Your own personal variables should start with a letter or an underscore. Following this can be any sequence of letters, digits, or underscores. Unix is case-sensitive. Upper-and lowercase letters are considered to be different. And naturally, you should make your names usefully mnemonic and avoid incorporating any of the predefined environment variable names.

Examples :

```
$ CH="Chapter "
$ ch=1
$ echo $CH$ch

 Chapter 1

$ echo $CH{ch}

 Chapter {ch}

$ echo ${CH}ch

 Chapter ch

$ echo ${CH}${ch}

 Chapter 1

$ CH1=$CH$ch
$ echo $CH1

 Chapter 1

$ echo CH1

 CH1

$ echo "$CH"ch

 Chapter ch
```

4.1 Setting and Unsetting Variables : The act of naming a variable in an assignment statement such as RSPLIB=/usr/rak/lib serves both to define (or set or declare) it and give it a value. There are occasions when you want to remove a variable name from the list. The unset special command can do this. Alternatively, you can simply assigns a null value to a variable without actually removing it from the shell's list. Note that the environment variables PATH, PS1, PS2, MAILCHECK, and IFS cannot be unset. for ex. unset RSPLIB.

4.2 Exporting Variables: Any variable "known" to a shell can be used in a shell script running under that shell. Unless you take specific steps, the names and values of variables are local (confined) to the shell in which they are declared. If a parent shell names a variable, that variable will not be known to any child processes (such as subshells) unless you export it using the export command.

```
$ GREET=Hello
$ echo $GREET
Hello
$ sh <now running subshell>

$ echo $GREET
$ _ <newline response:GREET unknown in subshell>
  <Ctrl-D back to parent shell>

$ export GREET
$ sh <run a subshell again>
$ echo $GREET
Hello <GREET is now known to subshell>
$ GREET=Saludos <reassign GREET>
$ echo GREET
Saludos
$ _ <Ctrl-D back to parent shell>
$ echo $GREET
Hello <Parent shell retains original value of GREET>
```

The set command can be used to display all the variables known to the current shell, together with their values. The rule that a subshell cannot alter the values of variables set in its parent shell is of fundamental importance. It is related to the concept of local and global variables found in most programming languages.

4.3 How Shell Scripts Work?

What really happens when you enter dw at the prompt? The full story is quite complex, but some of the subtleties need to be appreciated in order to master the art of shell scripting. Here's brief walkthrough to give you a feel for the sequence and to establish some essential shell jargon.

The first hurdle for your parent sh shell is parsing your command line. Parsing means that the command line is broken into tokens (or words) to determine the name(s) of the command(s) and the nature of the options and arguments, if any. During this parsing, certain special symbols (known as metacharacters) may trigger a variety of transformations to the arguments (such as command substitutions, parameter substitutions, blank interpretation, and file-name generation, etc.).

To do all this, the shell must scan the environment to pick up the values assigned to environment variables. The shell also sets the special variables with useful values such as $#, the number of arguments, and the positional parameters ($0,$1,and so on), encountered during the parse. Next, the shell will set up any explicit I/O redirections indicated by the > and < symbols.

The shell needs to locate the file, say dw (using the directory search sequence in PATH) and check the directory and file permissions. If the file is not found, the command is aborted. We talk about an exit from a process back to its parent. The shell always passes to the parent process a parameter $? known as the exit status (also called the condition code or return code). By convention, a zero value indicates a successful command execution; a nonzero value indicates failure and its value tells you the reason for failure. You can perform commands conditionally as follows : comm1 && comm2 comm3 || comm4. If comm1 executes successfully, returning an exit status of 0, then comm2 will be executed; if comm1 fails, returning a nonzero exit status, comm2 will be ignored. The second line reverses the logic: comm4 will only execute if comm3 is unsuccessful.

Having located dw and checked permissions, the shell must analyze the type of the dw file. If it is a compiled, binary program, a dw process is spawned to execute it. If dw is an executable text file, a subshell is spawned that will read the commands from the file. During this process, further parsing and substitutions will occur depending on the text encountered. For ex., occurences of positional parameters will be replaced by the appropriate values.

As each line of the script "unfolds," further processes and subshells may be spawned to any number of levels down the process chain, depending on the type of commands encountered. Any special commands such as echo and cd will be executed "within" the current shell. Eventually, child processes and subshells succeed and die, the script's end-of-file is reached, and control reverts back to the original login shell. Yet, it is possible to overload the system with too many processes and subshells. Both your login shell and the system as a whole have limits set by the size of the process table in the kernel. The error message Fork failed-too many processes or similar is Unix telling you to cool down (or buy a larger system!).

4.4 The Positional Parameters : Study the following scripts

```
:
# @(#) ssc.sh, RSP 08/07/2002
# This daft program illustrates the use of positional parameters
# and the shift command.
# usage : enter ssc followed by at least one argument
echo 'Command='
echo $0
echo 'Number of Arguments='
echo $#
echo 'The Original Argument List='
echo $*
shift
echo 'The Argument List After A Shift='
echo $*
```

Create and chmod this file as usual; then test it with various arguments:

Example 1.

```
 $ ssc -a b cd efg

Command=
ssc
```

```
Number Of Arguments=
4

The Original Argument List=
-a b cd efg

The Argument List After A Shift=
b cd efg

$ _
```

Example 2

```
 $ ssc "-a b" cd efg
```

The number of arguments in above is three, not four, because the space between -a and b has been quoted, that is to say hidden, from the shell.

Example 3

Try ssc with no arguments. The first $* correctly reports and empty list of arguments, but you'll get an error on the shift command-there is nothing to shift!

Example 4

An example that may puzzle you is

```
$ ssc x y x >misc.tmp
```

What will happen?

The redirection expression >misc.tmp is not considered a command argument.

4.5 echo AND THE SHELL META CHARACTERS:

The echo command lets you display texts and variables so that you can annotate and tabulate the output of commands embedded in your scripts. The arguments of echo are groups of characters separated by one or more spaces or tabs. It is vital to remember that these arguments are handled by the shell before being "echoed", so many strange transformations are possible: you do not always get back exactly the string of characters shown in the arguments.

After echoing the final (possibly transformed) argument, echo gives a concluding newline. You can suppress this newline in two ways: use the -n switch or the special '/c' escape sequence.

```
 $ echo Enter Your Name :
 Enter Your Name :
 $ _
```

But,

```
 $ echo -n Enter Your Name :
 Enter Your Name :$ _
```

and

```
 $ echo Enter Your Name :'\c'
 Enter Your Name :$_
```

 The last variant can also be written as

```
 $ echo 'Hear This :\c'
 Hear This :$ _
```

 or as

```
 $ echo "Hear This :\c"
 Hear This :$ _
```

Warning
Don't confuse a single forward quote(') with a back quote(`).
Don't confuse two forward quotes('') with a double quote(").
Don't confuse two backquotes(``) with a double quote(").

echo also allows the following useful escape sequences (your terminal may not recognize all of them, though) :

'\b'	Backspace
'\f'	Formfeed
'\n'	Newline
'\r'	Carriage Return
'\t'	Tab
'\v'	Vertical Tab
'\oct'	The ASCII code represented by the octal number oct. For ex, echo '\07' is equivalent to sending BELL (or Ctrl-G) to the terminal. Armed with a list of octal codes for the special graphics symbols, you can give your scripts jazzy windows and boxes. Octal numbers must start with a zero.

Examples

(1)
```
 $ echo Today is:  $ echo 'Today is:'
 Today is:  Today is:
 $ _  $ _
```

Both will produce same results, but for slightly different reasons.
In first version echo has two arguments, which are echoed one after the other. In second example, echo sees only one argument.

(2)

```
 $ echo 'What does "quote" mean?'
 What does "quote" mean?

 $ echo "What does 'quote' mean?"
 What does 'quote' mean?
```

```
(3) $ echo Today is `date`
 Today is Sun Nov 14 15:33:50 RSP 1993

 $ echo 'Today is `date`'
 Today is `date`

 $ echo "Today is `date`"
 Today is Sun Nov 14 15:33:50 RSP 1993
```

```
(4) $ pwd
 /usr/trainee/ggg/misc

 $ HERE=`pwd`
 $ echo $HERE
 /usr/trainee/ggg/misc

 $ cd <do things in /usr/trainee>

 $ cd $HERE <return to /usr/trainee/ggg/misc>
```

(5) Escaping with \

```
 $ echo \*
 *

 $ echo \\\?
 \?

 $ echo `date`
 Sun Nov 14 15:33:50 RSP 1993

 $ echo \`date\`
 `date`

 $ echo "\`date\`"
 `date`
```

(6) The Comment Character #

```
 $ echo Enter your Name: # added in version 2.0
 Enter your Name:

 $ echo Enter your Name '#':
```

```
 Enter your Name #:

 $ echo 'Enter your Name #:'
 Enter your name #:

 $ echo abc#def
 abc#def

 $ echo abc #def
 abc

 $ echo abc \#def
 abc #def

 $ echo abc \\#def
 abc \

 $ echo abc \\\#def
 abc \#def

 Note again the double \\ needed to display a single \.
```

(7) Double Quotes

```
 $ echo abc "#"def
 abc #def

 $ echo "abc #def"
 abc #def

 $ echo $HOME
 /usr/trainee

 $ echo '$HOME'
 $HOME

 $ echo "$HOME"
 /usr/trainee

 $ echo "\$HOME"
 $HOME

 $ echo "`date`"
 Sun Nov 14 15:33:50 RSP 1993

 $ echo "\""
 "

 $ echo "\\"
 \


 $ echo '\\'
 \\
```

(8) Metacharacter sequences

```
 $ echo The '*' in your eye
 The * in your eye

 $ echo 'The * in your eye'
 The * in your eye

 $ echo Have you read "War and Peace?"
 Have you read War and Peace?

 $ echo 'Have you read "War and Peace?"'
 Have you read "War and Peace?"

 $ echo "Have you read 'War and Peace?'"
 Have you read 'War and Peace?'

 $ echo [a-c]??[x-z]

 $ echo [d-e]*[!a-z]

 $ echo ?
 ?

 $ echo \?
 ?

 $ echo '?'
 ?

 $ echo "?"
 ?

 $ echo /bin/??
 /bin/ed /bin/dw /bin/vi

 $ echo ???

 $ echo \*
 *

 $ echo "*"
 *

 $ echo '*'
 *

 $ echo *
 <lists all filenames except thos with leading dot>

 $ echo 'What's that ?'
 > OK
 > OK'
```

5 Metacharacter Summary

SYMBOL MEANING
> command > file redirects stdout to file.

>> command >> file appends stdout to file.

< command < file redirects stdin from file.

| comm1 | comm2 pipeline; stdout from comm1
 directed to stdin of comm2 asynchronously.

<<delim Takes data (known as a here document) embedded in a shell script and redirects it to a command's stdin. The data is redirected until the string specified in delim is reached.

* Wildcard matches all strings including null but excluding dot file names.

? Wildcard matches all single characters except initial dot of a dot filename.

[char_list] Matches any single character in char_list.

[!char_list] Matches any single character not in char_list.

[r1-r2] Matches any single character in range r1 to r2.

[!r1-r2] Matches any single character not in range r1 to r2.

& comm1 and comm2 executes comm1 in background. comm2 does not wait for comm1 to finish asynchronous execution).

; comm1;comm2 executes comm1 thencomm2(sequential execution).

`commlist` The command(s) in commlist are executed and the output of the final command replaces the backquoted expression (command substitution).

(commlist) The command(s) in commlist are run in a subshell.

{commlist} The command(s) in commlist are run in the current shell.

$n The parameter $0 is set to the command name; the parameters $1-$9 are set to the first nine arguments.

$* Holds all the positional parameters starting with $1.

$@ Same as $* except when quoted as "$@". "$@" is "$1" "$2"...but "$*" is "$1 $2...".

$# The total number of positional parameters excluding $0.

Symbol Meaning

$? The exit status value (decimal). A zero value usually
 indicates a successful command. A nonzero value usually
 indicates failure.

$$ The process id of current shell.

$! The process id of the last background command.

$VAR The value of the variable VAR. Empty if VAR not defined.

${VAR}str The value of VAR is prepended to literal string str.

${VAR:-str} $VAR if VAR is defined and non-null; otherwise str.
 $VAR remains unchanged.

${VAR:=str} $VAR if VAR is defined and non-null; otherwise str.
 If VAR is undefined or null, str is assigned to $VAR.

${VAR:?mstr} $VAR if VAR is defined and non-null; otherwise displays the message mstr and exits the shell.
Default message if mstr empty is
VAR: parameter null or not set.

${VAR:+str} str if VAR is defined and non-null; otherwise makes no substitution. Without the :, the expressions work the same, but shell only checks whether VAR is defined (null or not).

\char Takes char literally (escape).

'str' Takes string str literally (single quote).

"str" Takes string str literally after interpreting $,
 `commlist`, and \.

# Comment character; ignores words starting with # and ignores all subsequent words up to next newline. (The Bourne Shell also treats any line starting with semicolon/space as a comment).

VAR=str Assigns str as new value of VAR.

VAR= Assigns null value to VAR. Constrats with unset VAR, which
 undefines VAR.

commlist1 Executes commands in commlist2 only if the commands in &&
commlist1 have executed successfully (i.e., have returned commlist2 an exit status of zero).

commlist1 Executes commands in commlist2 only if the commands in
|| commlist1 have executed unsuccessfully (i.e., have commlist2
returned a nonzero exit status).

---

6. Advanced Shell Scripts

```
$ cat mymenu
:
# @(#)simple 3-choice menu -- RSP 1996
#
quit=n
```

```
clear

while test "$quit" = "n"
do
 echo " Menu"
 echo "---------------------"
 echo
 echo "1. List Users"
 echo "2. Show Date "
 echo "3. Quit"
 echo
 echo -n "Enter choice : .\b"
 read choice
 case $choice in
 1) who;;
 2) date;;
 3) quit=y;;
 *) echo "Invalid choice ! \07\07"
 sleep 5;;
 esac
done
$chmod +x mymenu
$what menu.1
mymenu:
 simple 3-choice menu -- RSP 1996
$ _
```

NOTES

(1) The term blanks refers to the characters designated in the string $IFS (internal field separators). These are space, tab, and newline by default. The term white space (also written whitespace) refers to any combination of space, tab, and newline characters. Unless you alter the value of IFS, therefore, the terms blank(s) and white space are essentially synonymous.

(2) Scripts can always be executed using sh script_name, but providing execution permission is usually more convenient. Executable scripts can be invoked directly by name just like any other Unix program. Also, you have better control over who gets to use your scripts.

6.1 THE while LOOP

The script consists of a while loop that will execute relentlessly as long as the quit variable contains the value "n". The while construct is
one of several conditional or flow control mechanisms that play a key

role in all programming languages.

The shell while follows the following format :

```
while w_list
do

 d_list
done
```

w_list is called the while list, and consists of one or more commands, separated by semicolons or newlines. These commands (which can contain arguments, redirections, pipes, other scripts, and all the other command
mechanisms) are executed in the usual way. What happens next depends on the exit status returned by the last command executed in w_list.

Each command returns an exit status (often known simply as the return value) to the current shell, which is a number used to indicate how the command performed. You can access this number via the special parameter $?. A zero exit status means true (the command was successful); a nonzero exit status means false (the commands failed for some reason).

the while command takes note of the exit status of the final command in w_list: if true, the d_list of commands between do and done (known as do
list) is performed, and control then returns to the while command. The w_list is executed again, and so on. If the final exit status from w_list
is false, the d_list commands are skipped and execution resumes with the
command following done.

NOTE

If the do list ends with done &, the whole loop will be executed as a background process.

6.2 break and continue

There are two commands you can place in the do loop itself that alter the above scenario.

The break command causes an exit from the while loop, sending control directly to the command following done. If you have nested while loops, you can use break n, where n is a number defaulting to 1, to "break" out of n loops.

A continue command in the do loop stops execution of the do loop and returns control to the enclosing while loop. Again, if you have nested while loops, the continue n sends control to the nth enclosing while command. After a break, the while loop is over; after a continue, the while loop continues.

Break is useful when a condition arises in the loop that prevents further action.
continue is used when a particular item in the loop cannot be processed, but you want the loop to carry on processing further items.

[illegible]
[illegible]