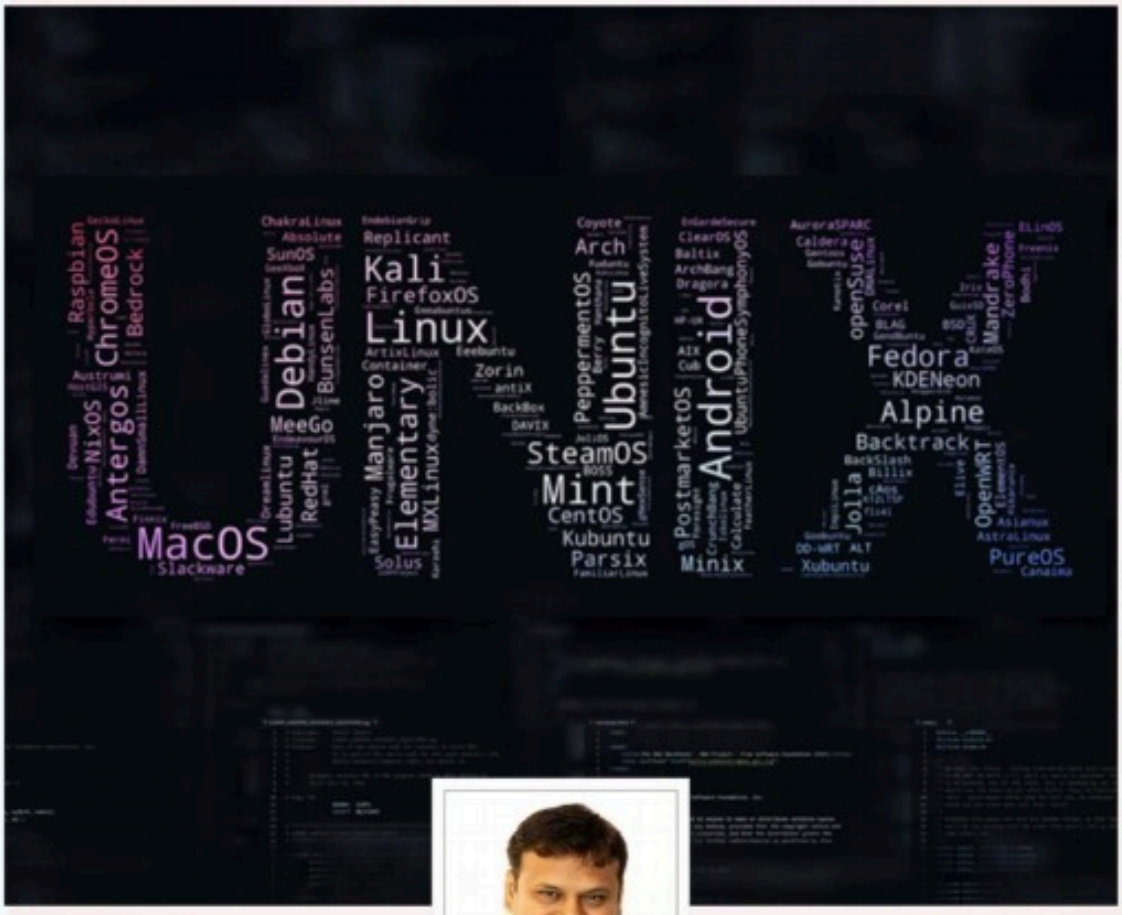
unix shell scripting

Jump2Learn
The Online Learning Place

UNIX
& SHELL PROGRAMMING

CHAPTER 3 - SHELL PROGRAMMING



AUTHOR

Mr. Manish Dolia

M.C.A.,Ph.D.(Pursuing)

DOLAT USHA INSTITUTE OF
APPLIED SCIENCES, VALSAD

3.1 SCREEN EDITOR "VI"

There are many ways to edit files in Unix. Editing files using the screen-oriented text editor vi is one of the best ways. This editor enables you to edit lines in context with other lines in the file.

An improved version of the vi editor which is called the VIM has also been made available now. Here, VIM stands for Vi IMproved.

vi is generally considered the de facto standard in Unix editors because –

- It's usually available on all the flavors of Unix system.
- Its implementations are very similar across the board.
- It requires very few resources.
- It is more user-friendly than other editors such as the ed or the ex.

You can use the vi editor to edit an existing file or to create a new file from scratch. You can also use this editor to just read a text file.

STARTING THE VI EDITOR

The following table lists out the basic commands to use the vi editor –

SR.NO.	COMMAND	DESCRIPTION
1	vi filename	Creates a new file if it already does not exist, otherwise opens an existing file.
2	vi-R filename	Opens an existing file in the read-only mode.
3	view filename	Opens an existing file in the read-only mode.

Following is an example to create a new file testfile if it already does not exist in the current working directory –

$vi testfile

The above command will generate the following output –

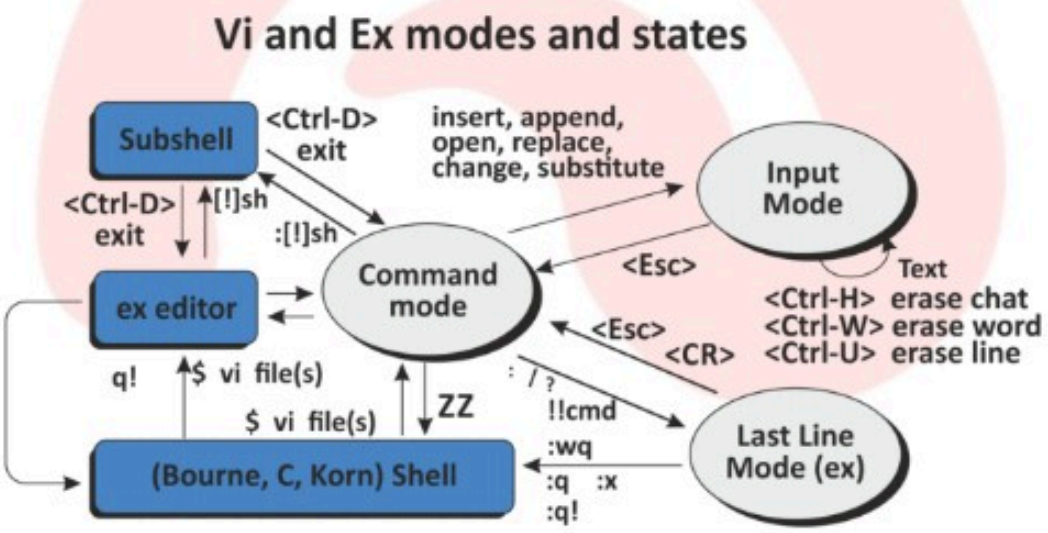
```
|
~
~
~
~
~
~
~
~
~
```

~
~
~
"testfile" [New File]

You will notice a tilde (~) on each line following the cursor. A tilde represents an unused line. If a line does not begin with a tilde and appears to be blank, there is a space, tab, newline, or some other non-viewable character present.

You now have one open file to start working on. Before proceeding further, let us understand a few important concepts.

OPERATION MODES



While working with the vi editor, we usually come across the following two modes –

- Command mode – This mode enables you to perform administrative tasks such as saving the files, executing the commands, moving the cursor, cutting (yanking) and pasting the lines or words, as well as finding and replacing. In this mode, whatever you type is interpreted as a command.
- Insert mode – This mode enables you to insert text into the file. Everything that's typed in this mode is interpreted as input and placed in the file.

vi always starts in the **command mode**. To enter text, you must be in the insert mode for which simply type i. To come out of the insert mode, press the **Esc** key, which will take you back to the command mode.
Hint – If you are not sure which mode you are in, press the Esc key twice; this will take you to the command mode. You open a file using the vi editor. Start by typing some characters and then come to the command mode to understand the difference.

GETTING OUT OF VI

The command to quit out of vi is :**q**. Once in the command mode, type colon, and 'q', followed by return. If your file has been modified in any way, the editor will warn you of this, and not let you quit. To ignore this message, the command to quit out of vi without saving is :**q!.** This lets you exit vi without saving any of the changes.
The command to save the contents of the editor is :w. You can combine the above command with the quit command, or use :wq and return.
The easiest way to **save your changes and exit vi** is with the ZZ command. When you are in the command mode, type **ZZ.** The **ZZ** command works the same way as the :**wq** command.

If you want to specify/state any particular name for the file, you can do so by specifying it after the **:w**. For example, if you wanted to save the file you were working on as another filename called **filename2**, you would type :**w filename2** and return.

MOVING WITHIN A FILE

To move around within a file without affecting your text, you must be in the command mode (press Esc twice). The following table lists out a few commands you can use to move around one character at a time –

Sr.No.	Command	Description
1	**K**	Moves the cursor up one line.
2	**J**	Moves the cursor down one line.
3	**H**	Moves the cursor to the left one character position.
4	**L**	Moves the cursor to the right one character position.

The following points need to be considered to move within a file –

- vi is case-sensitive. You need to pay attention to capitDodiazation when using the commands.
- Most commands in vi can be prefaced by the number of times you want the action to occur. For example, 2j moves the cursor two lines down the cursor location.

There are many other ways to move within a file in vi. Remember that you must be in the command mode **(press Esc twice).** The following table lists out a few commands to move around the file –

Given below is the list of commands to move around the file.

Sr.No.	Command	Description
1	0 or |	0 or | (Positions the cursor at the beginning of a line)
2	$	$ (Positions the cursor at the end of a line)
3	W	W (Positions the cursor to the next word)
4	B	B (Positions the cursor to the previous word)
5	(	((Positions the cursor to the beginning of the current sentence)
6	)	) (Positions the cursor to the beginning of the next sentence)
7	E	E (Moves to the end of the blank delimited word)
8	{	{ (Moves a paragraph back)
9	}	} (Moves a paragraph forward)
10	[[	[[(Moves a section back)
11	]]	]] (Moves a section forward)
12	n|	n| (Moves to the column n in the current line)
13	1G	1G (Moves to the first line of the file)
14	G	G (Moves to the last line of the file)
15	nG	nG (Moves to the nth line of the file)
16	:n	:n (Moves to the nth line of the file)
17	Fc	Fc (Moves forward to c)
18	Fc	Fc (Moves back to c)
19	H	H (Moves to the top of the screen)
20	nH	nH (Moves to the nth line from the top of the screen)
21	M	M (Moves to the middle of the screen)
22	L	L (Move to the bottom of the screen)
23	Nl	Nl (Moves to the nth line from the bottom of the screen)

24	:x	:x (Colon followed by a number would position the cursor on the line number represented by x

CONTROL COMMANDS

The following commands can be used with the Control Key to performs functions as given in the table below –
Given below is the list of control commands.

Sr.No.	Command	Descruotuib
1	CTRL+d	Moves forward 1/2 screen
2	CTRL+f	Moves forward one full screen
3	CTRL+u	Moves backward 1/2 screen
4	CTRL+b	Moves backward one full screen
5	CTRL+e	Moves the screen up one line
6	CTRL+y	Moves the screen down one line
7	CTRL+u	Moves the screen up 1/2 page
8	CTRL+d	Moves the screen down 1/2 page
9	CTRL+b	Moves the screen up one page
10	CTRL+f	Moves the screen down one page
11	CTRL+I	Redraws the screen

Editing Files

To edit the file, you need to be in the insert mode. There are many ways to enter the insert mode from the command mode –

SR. NO.	COMMAND	DESCRIPTION
1	I	Inserts text before the current cursor location
2	I	Inserts text at the beginning of the current line
3	A	Inserts text after the current cursor location
4	A	Inserts text at the end of the current line
5	O	Creates a new line for text entry below the cursor location
6	O	Creates a new line for text entry above the cursor location

DELETING CHARACTERS

Here is a list of important commands, which can be used to delete characters and lines in an open file –

Sr.No.	Command	Description
1	X	Deletes the character under the cursor location
2	X	Deletes the character before the cursor location
3	Dw	Deletes from the current cursor location to the next word
4	d^	Deletes from the current cursor position to the beginning of the line
5	d$	Deletes from the current cursor position to the end of the line
6	D	Deletes from the cursor position to the end of the current line
7	dd	Deletes the line the cursor is on

As mentioned above, most commands in vi can be prefaced by the number of times you want the action to occur. For example, 2x deletes two characters under the cursor location and 2dd deletes two lines the cursor is on.

It is recommended that the commands are practiced before we proceed further.

CHANGE COMMANDS

You also have the capability to change characters, words, or lines in vi without deleting them. Here are the relevant commands –

Sr.No.	Command	Description
1	Cc	Removes the contents of the line, leaving you in insert mode.
2	Cw	Changes the word the cursor is on from the cursor to the lowercase w end of the word.
3	R	Replaces the character under the cursor. vi returns to the command mode after the replacement is entered.
4	R	Overwrites multiple characters beginning with the character currently under the cursor. You must use Esc to stop the overwriting.
5	s	Replaces the current character with the character you type. Afterward, you are left in the insert mode.

6	S	Deletes the line the cursor is on and replaces it with the new text. After the new text is entered, vi remains in the insert mode.

COPY AND PASTE COMMANDS

You can copy lines or words from one place and then you can paste them at another place using the following commands –

Sr.No.	Command	Description
1	yy	Copies the current line.
2	yw	Copies the current word from the character the lowercase w cursor is on, until the end of the word.
3	p	Puts the copied text after the cursor.
4	P	Puts the yanked text before the cursor.

ADVANCED COMMANDS

There are some advanced commands that simplify day-to-day editing and allow for more efficient use of vi –
Given below is the list advanced commands.

Sr.	Command	Description
1	J	Joins the current line with the next one. A count of j commands join many lines.
2	<<	Shifts the current line to the left by one shift width.
3	>>	Shifts the current line to the right by one shift width.
4	~	Switches the case of the character under the cursor.
5	^G	Press Ctrl and G keys at the same time to show the current filename and the status.
6	U	Restores the current line to the state it was in before the cursor entered the line.
7	u	This helps undo the last change that was done in the file. Typing 'u' again will re-do the change.
8	J	Joins the current line with the next one. A count joins that many lines.
9	:f	Displays the current position in the file in % and the file name, the total number of file.
10	:f	Renames the current file to filename.

	filename	
11	:w filename	Writes to file filename.
12	:e filename	Opens another file with filename.
13	:cd dirname	Changes the current working directory to dirname.
14	:e #	Toggles between two open files.
15	:n	In case you open multiple files using vi, use :n to go to the next file in the series.
16	:p	In case you open multiple files using vi, use :p to go to the previous file in the series.
17	:N	In case you open multiple files using vi, use :N to go to the previous file in the series.
18	:r file	Reads file and inserts it after the current line.
19	:nr file	Reads file and inserts it after the line n.

WORD AND CHARACTER SEARCHING

The vi editor has two kinds of searches: **string** and **character**. For a string search, the **/** and **?** commands are used. When you start these commands, the command just typed will be shown on the last line of the screen, where you type the particular string to look for.
These two commands differ only in the direction where the search takes place –

- The / command searches forwards (downwards) in the file.
- The ? command searches backwards (upwards) in the file.

The **n** and **N** commands repeat the previous search command in the same or the opposite direction, respectively. Some characters have special meanings. These characters must be preceded by a backslash (\) to be included as part of the search expression.

Sr.No.	Character	Description
1	^	Searches at the beginning of the line (Use at the beginning of a search expression).
2	.	Matches a single character.
3	*	Matches zero or more of the previous character.
4	$	End of the line (Use at the end of the search

		expression).
5	[	Starts a set of matching or non-matching expressions.
6	<	This is put in an expression escaped with the backslash to find the ending or the beginning of a word.
7	>	This helps see the '<' character description above.

The character search searches within one line to find a character entered after the command. The f and F commands search for a character on the current line only. f searches forwards and F searches backwards and the cursor moves to the position of the found character.

The t and T commands search for a character on the current line only, but for t, the cursor moves to the position before the character, and T searches the line backwards to the position after the character.

Set Commands

You can change the look and feel of your vi screen using the following :set commands. Once you are in the command mode, type :set followed by any of the following commands.

Sr.No.	Command	Description
1	:set ic	Ignores the case when searching
2	:set ai	Sets autoindent
3	:set noai	Unsets autoindent
4	:set nu	Displays lines with line numbers on the left side
5	:set sw	Sets the width of a software tabstop. For example, you would set a shift width of 4 with this command — :set sw = 4
6	:set ws	If wrapscan is set, and the word is not found at the bottom of the file, it will try searching for it at the beginning
7	:set wm	If this option has a value greater than zero, the editor will automatically "word wrap". For example, to set the wrap margin to two characters, you would type this: :set wm = 2
8	:set ro	Changes file type to "read only"
9	:set term	Prints terminal type

10	:set bf	Discards control characters from input

RUNNING COMMANDS

The vi has the capability to run commands from within the editor. To run a command, you only need to go to the command mode and type :! command.
For example, if you want to check whether a file exists before you try to save your file with that filename, you can type :! ls and you will see the output of ls on the screen.
You can press any key (or the command's escape sequence) to return to your vi session.

REPLACING TEXT

The substitution command **(:s/)** enables you to quickly replace words or groups of words within your files. Following is the syntax to replace text –
:s/search/replace/g
The g stands for globally. The result of this command is that all occurrences on the cursor's line are changed.

IMPORTANT POINTS TO NOTE

The following points will add to your success with vi –

- You must be in command mode to use the commands. (Press Esc twice at any time to ensure that you are in command mode.)
- You must be careful with the commands. These are case-sensitive.
- You must be in insert mode to enter text.

3.2. ENVIRONMENTAL & USER DEFINED VARIABLES

A variable is a character string to which we assign a value. The value assigned could be a number, text, filename, device, or any other type of data.
A variable is nothing more than a pointer to the actual data. The shell enables you to create, assign, and delete variables.

VARIABLE NAMES

The name of a variable can contain only letters (a to z or A to Z), numbers (0 to 9) or the underscore character (_).
By convention, Unix shell variables will have their names in UPPERCASE.
The following examples are vDodiad variable names –
_DODIA
TOKEN_A

VAR_1
VAR_2
Following are the examples of invDodiad variable names –
2_VAR
-VARIABLE
VAR1-VAR2
VAR_A!
The reason you cannot use other characters such as !, *, or - is that these characters have a special meaning for the shell.

DEFINING VARIABLES
Variables are defined as follows –
variable_name=variable_value
For example –
NAME="Manish Dodia"
The above example defines the variable NAME and assigns the value "Manish Dodia" to it. Variables of this type are called scalar variables. A scalar variable can hold only one value at a time.
Shell enables you to store any value you want in a variable. For example –
VAR1="Manish Dodia"
VAR2=100

ACCESSING VALUES
To access the value stored in a variable, prefix its name with the dollar sign ($) –
For example, the following script will access the value of defined variable NAME and print it on STDOUT –

```
#!/bin/sh
NAME="Manish Dodia"
echo $NAME
```


The above script will produce the following value –
Manish Dodia

READ-ONLY VARIABLES
Shell provides a way to mark variables as read-only by using the read-only command. After a variable is marked read-only, its value cannot be changed.
For example, the following script generates an error while trying to change the value of NAME –

```
#!/bin/sh
NAME="Manish Dodia"
readonly NAME
NAME="Akashi"
```

The above script will generate the following result –
/bin/sh: NAME: This variable is read only.

UNSETTING VARIABLES

Unsetting or deleting a variable directs the shell to remove the variable from the list of variables that it tracks. Once you unset a variable, you cannot access the stored value in the variable.

Following is the syntax to unset a defined variable using the **unset** command –

```
unset variable_name
```

The above command unsets the value of a defined variable. Here is a simple example that demonstrates how the command works –

```
#!/bin/sh
NAME="Manish Dodia"
unset NAME
echo $NAME
```

The above example does not print anything. You cannot use the unset command to unset variables that are marked readonly.

VARIABLE TYPES

When a shell is running, three main types of variables are present –

- Local Variables – A local variable is a variable that is present within the current instance of the shell. It is not available to programs that are started by the shell. They are set at the command prompt.
- Environment Variables – An environment variable is available to any child process of the shell. Some programs need environment variables in order to function correctly. Usually, a shell script defines only those environment variables that are needed by the programs that it runs.

- Shell Variables – A shell variable is a special variable that is set by the shell and is required by the shell in order to function correctly. Some of these variables are environment variables whereas others are local variables.

- The following table shows a number of special variables that you can use in your shell scripts –

SR.NO.	VARIABLE	DESCRIPTION
1	$0	The filename of the current script.
2	$n	These variables correspond to the arguments with which a script was invoked. Here n is a positive decimal number corresponding to the position of an argument (the first argument is $1, the second argument is $2, and so on).
3	$#	The number of arguments supplied to a script.
4	$*	All the arguments are double quoted. If a script receives two arguments, $* is equivalent to $1 $2.
5	$@	All the arguments are individually double quoted. If a script receives two arguments, $@ is equivalent to $1 $2.
6	$?	The exit status of the last command executed.
7	$$	The process number of the current shell. For shell scripts, this is the process ID under which they are executing.
8	$!	The process number of the last background command.

COMMAND-LINE ARGUMENTS

- The command-line arguments $1, $2, $3, ...$9 are positional parameters, with $0 pointing to the actual command, program, shell script, or function and $1, $2, $3, ...$9 as the arguments to the command.
- Following script uses various special variables related to the command line –


```
#!/bin/sh
echo "File Name: $0"
echo "First Parameter : $1"
echo "Second Parameter : $2"
echo "Quoted Values: $@"
echo "Quoted Values: $*"
echo "Total Number of Parameters : $#"
```

Here is a sample run for the above script –
$./test.sh Manish Dodia
File Name : ./test.sh
First Parameter : Manish
Second Parameter : Dodia
Quoted Values: Manish Dodia
Quoted Values: Manish Dodia
Total Number of Parameters : 2

SPECIAL PARAMETERS $* AND $@

- There are special parameters that allow accessing all the command-line arguments at once. $* and $@ both will act the same unless they are enclosed in double quotes, "".
- Both the parameters specify the command-line arguments. However, the "$*" special parameter takes the entire list as one argument with spaces between and the "$@" special parameter takes the entire list and separates it into separate arguments.
- We can write the shell script as shown below to process an unknown number of commandline arguments with either the $* or $@ special parameters –


```
#!/bin/sh
for TOKEN in $*
do
  echo $TOKEN
done
```


Here is a sample run for the above script –
$./test.sh Manish Dodia 10 Years Old
Manish
Dodia
10
Years
Old
Note – Here do...done is a kind of loop that will be covered in a subsequent tutorial.

EXIT STATUS

The **$?** variable represents the exit status of the previous command.

Exit status is a numerical value returned by every command upon its completion. As a rule, most commands return an exit status of 0 if they were successful, and 1 if they were unsuccessful.

Some commands return additional exit statuses for particular reasons. For example, some commands differentiate between kinds of errors and will return various exit values depending on the specific type of failure.

Following is the example of successful command –

```
$./test.sh Manish Dodia
File Name : ./test.sh
First Parameter : Manish
Second Parameter : Dodia
Quoted Values: Manish Dodia
Quoted Values: Manish Dodia
Total Number of Parameters : 2
$echo $?
0
$
```


ARRAY VARIABLE

Shell supports a different type of variable called an array variable. This can hold multiple values at the same time. Arrays provide a method of grouping a set of variables. Instead of creating a new name for each variable that is required, you can use a single array variable that stores all the other variables.

All the naming rules discussed for Shell Variables would be applicable while naming arrays.

DEFINING ARRAY VALUES

The difference between an array variable and a scalar variable can be explained as follows.

Suppose you are trying to represent the names of various students as a set of variables. Each of the individual variables is a scalar variable as follows –

```
NAME01="Manish"
NAME02="Akash"
NAME03="Mahnaz"
NAME04="Ayan"
```

NAME05="Daisy"

We can use a single array to store all the above mentioned names. Following is the simplest method of creating an array variable. This helps assign a value to one of its indices.

array_name[index]=value

Here array_name is the name of the array, index is the index of the item in the array that you want to set, and value is the value you want to set for that item.

As an example, the following commands –

NAME[0]="Manish"
NAME[1]="Akash"
NAME[2]="Mahnaz"
NAME[3]="Ayan"
NAME[4]="Daisy"

If you are using the ksh shell, here is the syntax of array initiDodiazation –

set -A array_name value1 value2 ... valuen

If you are using the bash shell, here is the syntax of array initiDodiazation –

array_name=(value1 ... valuen)

ACCESSING ARRAY VALUES

After you have set any array variable, you access it as follows –

${array_name[index]}

Here array_name is the name of the array, and index is the index of the value to be accessed. Following is an example to understand the concept –

```
#!/bin/sh
NAME[0]="Manish"
NAME[1]="Akash"
NAME[2]="Mahnaz"
NAME[3]="Ayan"
NAME[4]="Daisy"
echo "First Index: ${NAME[0]}"
echo "Second Index: ${NAME[1]}"
```


The above example will generate the following result –

```
$./test.sh
First Index: Manish
Second Index: Akash
```

You can access all the items in an array in one of the following ways –

${array_name[*]}
${array_name[@]}

Here array_name is the name of the array you are interested in. Following example will help you understand the concept –

```
#!/bin/sh
NAME[0]="Manish"
NAME[1]="Akash"
NAME[2]="Mahnaz"
NAME[3]="Ayan"
NAME[4]="Daisy"
echo "First Method: ${NAME[*]}"
echo "Second Method: ${NAME[@]}"
```

The above example will generate the following result –

```
$./test.sh
First Method: Manish Akash Mahnaz Ayan Daisy
Second Method: Manish Akash Mahnaz Ayan Daisy
```

3.3 ARGUMENT PROCESSING & 3.4 SHELL'S INTERPRETATION AT PROMPT

A shell script is a computer program designed to be run by the Unix/Linux shell which could be one of the following:

- The Bourne Shell
- The C Shell
- The Korn Shell
- The GNU Bourne-Again Shell

A shell is a command-line interpreter and typical operations performed by shell scripts include file manipulation, program execution, and printing text.

Shell scripts have several required constructs that tell the shell environment what to do and when to do it.

The shell is, after all, a real programming language, complete with variables, control structures, and so forth.

The following script uses the **read** command which takes the input from the keyboard and assigns it as the value of the variable PERSON and finally prints it on STDOUT.

```
#!/bin/sh
# Author : Manish Dodia
# Script follows here:
echo "What is your name?"
read PERSON
echo "Hello, $PERSON"
```

Here is a sample run of the script –

```
$./test.sh
What is your name?
Manish Dodia
Hello, Manish Dodia
$
```

3.5 ARITHMETIC EXPRESSION EVALUATION

OPERATORS

There are various operators supported by each shell. We will discuss in detail about Bourne shell (default shell) in this chapter.
We will now discuss the following operators –

- Arithmetic Operators
- Relational Operators
- Boolean Operators
- String Operators
- File Test Operators

Bourne shell didn't originally have any mechanism to perform simple arithmetic operations but it uses external programs, either **awk** or **expr.**
The following example shows how to add two numbers –

```
#!/bin/sh
val=`expr 2 + 2`
echo "Total value : $val"
```

The above script will generate the following result –

```
Total value : 4
```

The following points need to be considered while adding –

- There must be spaces between operators and expressions. For example, 2+2 is not correct; it should be written as 2 + 2.
- The complete expression should be enclosed between ‘ ‘, called the backtick.

ARITHMETIC OPERATORS

The following arithmetic operators are supported by Bourne Shell.
Assume variable a holds 10 and variable b holds 20 then –
Show Examples

OPERATOR	DESCRIPTION	EXAMPLE
+ (Addition)	Adds values on either side of the operator	`expr $a + $b` will give 30
- (Subtraction)	Subtracts right hand operand from left hand operand	`expr $a - $b` will give -10
* (Multiplication)	Multiplies values on either side of the operator	`expr $a \* $b` will give 200
/ (Division)	Divides left hand operand by right hand operand	`expr $b / $a` will give 2
% (Modulus)	Divides left hand operand by right hand operand and returns remainder	`expr $b % $a` will give 0
#NAME?	Assigns right operand in left operand	a = $b would assign value of b into a
== (EquDodiaty)	Compares two numbers, if both are same then returns true.	[$a == $b] would return false.
!= (Not EquDodiaty)	Compares two numbers, if both are different then returns true.	[$a != $b] would return true.

It is very important to understand that all the conditional expressions should be inside square braces with spaces around them, for example [$a == $b] is correct whereas, [$a==$b] is incorrect.

All the arithmetical calculations are done using long integers.

RELATIONAL OPERATORS

Bourne Shell supports the following relational operators that are specific to numeric values. These operators do not work for string values unless their value is numeric.

For example, following operators will work to check a relation between 10 and 20 as well as in between "10" and "20" but not in between "ten" and "twenty".

Assume variable a holds 10 and variable b holds 20 then –

Show Examples

OPERATOR	DESCRIPTION	EXAMPLE
-eq	Checks if the value of two operands are equal or not; if yes, then the condition becomes true.	[$a -eq $b] is not true.
-ne	Checks if the value of two operands are equal or not; if values are not equal, then the condition becomes true.	[$a -ne $b] is true.
-gt	Checks if the value of left operand is greater than the value of right operand; if yes, then the condition becomes true.	[$a -gt $b] is not true.
-lt	Checks if the value of left operand is less than the value of right operand; if yes, then the condition becomes true.	[$a -lt $b] is true.
-ge	Checks if the value of left operand is greater than or equal to the value of right operand; if yes, then the condition becomes true.	[$a -ge $b] is not true.
-le	Checks if the value of left operand is less than or equal to the value of right operand; if yes, then the condition becomes true.	[$a -le $b] is true.

It is very important to understand that all the conditional expressions should be placed inside square braces with spaces around them. For example, **[$a <= $b]** is correct whereas,
[$a <= $b] is incorrect.

BOOLEAN OPERATORS

The following Boolean operators are supported by the Bourne Shell.
Assume variable **a** holds 10 and variable **b** holds 20 then –

Show Examples

OPERATOR	DESCRIPTION	EXAMPLE
!	This is logical negation. This inverts a true condition into false and vice versa.	[! false] is true.
-o	This is logical OR. If one of the operands is true, then the condition becomes true.	[$a -lt 20 -o $b -gt 100] is true.
-a	This is logical AND. If both the operands are true, then the condition becomes true otherwise false.	[$a -lt 20 -a $b -gt 100] is false.

STRING OPERATORS

The following string operators are supported by Bourne Shell. Assume variable a holds "abc" and variable b holds "efg" then –

Show Examples

Operator	Description	Example
=	Checks if the value of two operands are equal or not; if yes, then the condition becomes true.	[$a = $b] is not true.
!=	Checks if the value of two operands are equal or not; if values are not equal then the condition becomes true.	[$a != $b] is true.
-z	Checks if the given string operand size is zero; if it is zero length, then it returns true.	[-z $a] is not true.
-n	Checks if the given string operand size is non-zero; if it is nonzero length, then it returns true.	[-n $a] is not false.
str	Checks if str is not the empty string; if it is empty, then it returns false.	[$a] is not false.

22

FILE TEST OPERATORS

We have a few operators that can be used to test various properties associated with a Unix file.

Assume a variable file holds an existing file name "test" the size of which is 100 bytes and has read, write and execute permission on –

Show Examples

OPERATOR	DESCRIPTION	EXAMPLE
-b file	Checks if file is a block special file; if yes, then the condition becomes true.	[-b $file] is false.
-c file	Checks if file is a character special file; if yes, then the condition becomes true.	[-c $file] is false.
-d file	Checks if file is a directory; if yes, then the condition becomes true.	[-d $file] is not true.
-f file	Checks if file is an ordinary file as opposed to a directory or special file; if yes, then the condition becomes true.	[-f $file] is true.
-g file	Checks if file has its set group ID (SGID) bit set; if yes, then the condition becomes true.	[-g $file] is false.
-k file	Checks if file has its sticky bit set; if yes, then the condition becomes true.	[-k $file] is false.
-p file	Checks if file is a named pipe; if yes, then the condition becomes true.	[-p $file] is false.
-t file	Checks if file descriptor is open and associated with a terminal; if yes, then the condition becomes true.	[-t $file] is false.
-u file	Checks if file has its Set User ID (SUID) bit set; if yes, then the condition becomes true.	[-u $file] is false.
-r file	Checks if file is readable; if yes, then the condition becomes true.	[-r $file] is true.
-w file	Checks if file is writable; if yes, then the condition becomes true.	[-w $file] is true.
-x file	Checks if file is executable; if yes, then the condition becomes true.	[-x $file] is true.
-s file	Checks if file has size greater than 0; if yes, then condition becomes true.	[-s $file] is true.
-e file	Checks if file exists; is true even if file is a directory but exists.	[-e $file] is true.

CONDITIONAL STATEMENTS

Unix Shell supports conditional statements which are used to perform different actions based on different conditions. We will now understand two decision-making statements here –

- The if...else statement
- The case...esac statement

THE IF...ELSE STATEMENTS

If else statements are useful decision-making statements which can be used to select an option from a given set of options.

Unix Shell supports following forms of if...else statement –

- if...fi statement
- if...else...fi statement
- if...elif...else...fi statement

Most of the if statements check relations using relational operators discussed in the previous chapter.

The case...esac Statement

You can use multiple if...elif statements to perform a multiway branch. However, this is not always the best solution, especially when all of the branches depend on the value of a single variable.

Unix Shell supports case...esac statement which handles exactly this situation, and it does so more efficiently than repeated if...elif statements.

There is only one form of case...esac statement which has been described in detail here –

- case...esac statement

The case...esac statement in the Unix shell is very similar to the switch...case statement we have in other programming languages like C or C++ .

LOOPS

loops available to shell programmers –

- The while loop
- The for loop
- The until loop
- The select loop

You will use different loops based on the situation. For example, the while loop executes the given commands until the given condition remains true; the until loop executes until a given condition becomes true.

Once you have good programming practice you will gain the expertise and thereby, start using appropriate loop based on the situation. Here, while and for loops are available in most of the other programming languages like C, C++ and PERL, etc.

NESTING LOOPS

All the loops support nesting concept which means you can put one loop inside another similar one or different loops. This nesting can go up to unlimited number of times based on your requirement.

Here is an example of nesting while loop. The other loops can be nested based on the programming requirement in a similar way –

NESTING WHILE LOOPS

It is possible to use a while loop as part of the body of another while loop.

Syntax

```
while command1 ; # this is loop1, the outer loop
do
  Statement(s) to be executed if command1 is true

  while command2 ; # this is loop2, the inner loop
  do
    Statement(s) to be executed if command2 is true
  done

  Statement(s) to be executed if command1 is true
done
```

Example

Here is a simple example of loop nesting. Let's add another countdown loop inside the loop that you used to count to nine –

```
#!/bin/sh

a=0
while [ "$a" -lt 10 ]    # this is loop1
do
  b="$a"
  while [ "$b" -ge 0 ]  # this is loop2
  do
    echo -n "$b "
    b=`expr $b - 1`
```

```
  done
  echo
  a=`expr $a + 1`
done
```

This will produce the following result. It is important to note how echo -n works here. Here -n option lets echo avoid printing a new line character.

```
0
1 0
2 1 0
3 2 1 0
4 3 2 1 0
5 4 3 2 1 0
6 5 4 3 2 1 0
7 6 5 4 3 2 1 0
8 7 6 5 4 3 2 1 0
9 8 7 6 5 4 3 2 1 0
```

following two statements that are used to control shell loops–

- The break statement
- The continue statement

THE INFINITE LOOP

All the loops have a limited life and they come out once the condition is false or true depending on the loop.

A loop may continue forever if the required condition is not met. A loop that executes forever without terminating executes for an infinite number of times. For this reason, such loops are called infinite loops.

Example

Here is a simple example that uses the while loop to display the numbers zero to nine –

```
#!/bin/sh

a=10

until [ $a -lt 10 ]
do
  echo $a
  a=`expr $a + 1`
done
```

This loop continues forever because a is always greater than or equal to 10 and it is never less than 10.

THE BREAK STATEMENT

The break statement is used to terminate the execution of the entire loop, after completing the execution of all of the lines of code up to the break statement. It then steps down to the code following the end of the loop.

Syntax

The following break statement is used to come out of a loop –

```
break
```

The break command can also be used to exit from a nested loop using this format –

```
break n
```

Here n specifies the nth enclosing loop to the exit from.

Example

Here is a simple example which shows that loop terminates as soon as a becomes 5 –

```
#!/bin/sh

a=0
while [ $a -lt 10 ]
do
  echo $a
  if [ $a -eq 5 ]
  then
    break
  fi
  a=`expr $a + 1`
done
```

Upon execution, you will receive the following result –

```
0
1
2
3
4
5
```

Here is a simple example of nested for loop. This script breaks out of both loops if var1 equals 2 and var2 equals 0 –

```
#!/bin/sh
for var1 in 1 2 3
do
```

```
  for var2 in 0 5
  do
    if [ $var1 -eq 2 -a $var2 -eq 0 ]
    then
      break 2
    else
      echo "$var1 $var2"
    fi
  done
done
```

Upon execution, you will receive the following result. In the inner loop, you have a break command with the argument 2. This indicates that if a condition is met you should break out of outer loop and ultimately from the inner loop as well.

```
1 0
1 5
```

THE CONTINUE STATEMENT

The continue statement is similar to the break command, except that it causes the current iteration of the loop to exit, rather than the entire loop.

This statement is useful when an error has occurred but you want to try to execute the next iteration of the loop.

Syntax

```
continue
```

Like with the break statement, an integer argument can be given to the continue command to skip commands from nested loops.

```
continue n
```

Here n specifies the nth enclosing loop to continue from.

Example

The following loop makes use of the continue statement which returns from the continue statement and starts processing the next statement –

```
#!/bin/sh
NUMS="1 2 3 4 5 6 7"
for NUM in $NUMS
do
  Q=`expr $NUM % 2`
  if [ $Q -eq 0 ]
  then
    echo "Number is an even number!!"
    continue
```

```
  fi
  echo "Found odd number"
done
```

Upon execution, you will receive the following result –

```
Found odd number
Number is an even number!!
Found odd number
Number is an even number!!
Found odd number
Number is an even number!!
Found odd number
```

WHAT IS SUBSTITUTION?

The shell performs substitution when it encounters an expression that contains one or more special characters.

Example

Here, the printing value of the variable is substituted by its value. Same time, "\n" is substituted by a new line –

```
#!/bin/sh
a=10
echo -e "Value of a is $a \n"
```

You will receive the following result. Here the -e option enables the interpretation of backslash escapes.

```
Value of a is 10
```

Following is the result without -e option –

```
Value of a is 10\n
```

The following escape sequences which can be used in echo command –

SR.NO.	ESCAPE	& DESCRIPTION
1	\\	backslash
2	\a	alert (BEL)
3	\b	backspace
4	\c	suppress trailing newline
5	\f	form feed
6	\n	new line
7	\r	carriage return
8	\t	horizontal tab
9	\v	vertical tab

You can use the -E option to disable the interpretation of the backslash escapes (default).
You can use the -n option to disable the insertion of a new line.

COMMAND SUBSTITUTION

Command substitution is the mechanism by which the shell performs a given set of commands and then substitutes their output in the place of the commands.
Syntax
The command substitution is performed when a command is given as –
`command`

When performing the command substitution make sure that you use the backquote, not the single quote character.
Example
Command substitution is generally used to assign the output of a command to a variable. Each of the following examples demonstrates the command substitution –

```
#!/bin/sh
DATE=`date`
echo "Date is $DATE"

USERS=`who | wc -l`
echo "Logged in user are $USERS"

UP=`date ; uptime`
echo "Uptime is $UP"
```

Upon execution, you will receive the following result –

```
Date is Thu Jul  2 03:59:57 MST 2009
Logged in user are 1
Uptime is Thu Jul  2 03:59:57 MST 2009
03:59:57 up 20 days, 14:03,  1 user,  load avg: 0.13, 0.07, 0.15
```

VARIABLE SUBSTITUTION

Variable substitution enables the shell programmer to manipulate the value of a variable based on its state.
Here is the following table for all the possible substitutions –

SR.NO.	FORM	FORM & DESCRIPTION
1	${var}	Substitute the value of var.
2	${var:-word}	If var is null or unset, word is substituted for var. The value of var does not change.
3	${var:=word}	If var is null or unset, var is set to the value of word.
4	${var:?message}	If var is null or unset, message is printed to standard error. This checks that variables are set correctly.
5	${var:+word}	If var is set, word is substituted for var. The value of var does not change.

Example
Following is the example to show various states of the above substitution −

```
#!/bin/sh
echo ${var:-"Variable is not set"}
echo "1 - Value of var is ${var}"

echo ${var:="Variable is not set"}

echo "2 - Value of var is ${var}"
unset var
echo ${var:+"This is default value"}
echo "3 - Value of var is $var"
var="Prefix"
echo ${var:+"This is default value"}
echo "4 - Value of var is $var"
echo ${var:?"Print this message"}
echo "5 - Value of var is ${var}"
```

Upon execution, you will receive the following result −

```
Variable is not set
1 - Value of var is
Variable is not set
2 - Value of var is Variable is not set
3 - Value of var is
This is default value
```

4 - Value of var is Prefix
Prefix
5 - Value of var is Prefix

THE METACHARACTERS

Unix Shell provides various metacharacters which have special meaning while using them in any Shell Script and causes termination of a word unless quoted. For example, ? matches with a single character while listing files in a directory and an * matches more than one character. Here is a list of most of the shell special characters (also called metacharacters) –

* ? [ ] ' " \ $; & ( ) | ^ < > new-line space tab
A character may be quoted (i.e., made to stand for itself) by preceding it with a \.
Example
Following example shows how to print a * or a ? –

```
#!/bin/sh
echo Hello; Word
```

Upon execution, you will receive the following result –

```
Hello
./test.sh: line 2: Word: command not found
shell returned 127
```

Let us now try using a quoted character –

```
#!/bin/sh
echo Hello\; Word
```

Upon execution, you will receive the following result –

```
Hello; Word
```

The $ sign is one of the metacharacters, so it must be quoted to avoid special handling by the shell –

```
#!/bin/sh
echo "I have \$1200"
```

Upon execution, you will receive the following result –

```
I have $1200
```

The following table lists the four forms of quoting –

SR.NO.	QUOTING	DESCRIPTION
1	Single quote	All special characters between these quotes lose their special meaning.
2	Double quote	Most special characters between these quotes lose their special meaning with these exceptions –
	$	
	`	
	\$	
	\'	
	\"	
	\\	
3	Backslash	Any character immediately following the backslash loses its special meaning.
4	Back quote	Anything in between back quotes would be treated as a command and would be executed.

THE SINGLE QUOTES

Consider an echo command that contains many special shell characters –

```
echo <-$1500.**>; (update?) [y|n]
```

Putting a backslash in front of each special character is tedious and makes the line difficult to read –

```
echo \<-\$1500.\*\*\>\; \(update\?\) \[y\|n\]
```

There is an easy way to quote a large group of characters. Put a single quote (') at the beginning and at the end of the string –

```
echo '<-$1500.**>; (update?) [y|n]'
```

Characters within single quotes are quoted just as if a backslash is in front of each character. With this, the echo command displays in a proper way.

If a single quote appears within a string to be output, you should not put the whole string within single quotes instead you should precede that using a backslash (\) as follows –

```
echo 'It\'s Shell Programming
```

THE DOUBLE QUOTES

Try to execute the following shell script. This shell script makes use of single quote –

```
VAR=MANISH
echo '$VAR owes <-$1500.**>; [ as of (`date +%m/%d`) ]'
```

Upon execution, you will receive the following result –

```
$VAR owes <-$1500.**>; [ as of (`date +%m/%d`) ]
```

This is not what had to be displayed. It is obvious that single quotes prevent variable substitution. If you want to substitute variable values and to make inverted commas work as expected, then you would need to put your commands in double quotes as follows –

```
VAR=MANISH
echo "$VAR owes <-\$1500.**>; [ as of (`date +%m/%d`) ]"
```

Upon execution, you will receive the following result –

```
MANISH owes <-$1500.**>; [ as of (07/02) ]
```

Double quotes take away the special meaning of all characters except the following –

- $ for parameter substitution
- Backquotes for command substitution
- \$ to enable literal dollar signs
- \` to enable literal backquotes
- \" to enable embedded double quotes
- \\ to enable embedded backslashes
- All other \ characters are literal (not special)

Characters within single quotes are quoted just as if a backslash is in front of each character. This helps the echo command display properly.

If a single quote appears within a string to be output, you should not put the whole string within single quotes instead you should precede that using a backslash (\) as follows –

```
echo 'It\'s Shell Programming'
```

THE BACKQUOTES

Putting any Shell command in between backquotes executes the command.

Syntax

Here is the simple syntax to put any Shell command in between backquotes –

```
var=`command`
```

Example

The date command is executed in the following example and the produced result is stored in DATA variable.

```
DATE=`date`
echo "Current Date: $DATE"
```

Upon execution, you will receive the following result –

Current Date: Thu Jul 2 05:28:45 MST 2009

3.7 REDIRECTION

OUTPUT REDIRECTION

The output from a command normally intended for standard output can be easily diverted to a file instead. This capability is known as output redirection.

If the notation > file is appended to any command that normally writes its output to standard output, the output of that command will be written to file instead of your terminal.

Check the following who command which redirects the complete output of the command in the users file.

```
$ who > users
```

Notice that no output appears at the terminal. This is because the output has been redirected from the default standard output device (the terminal) into the specified file. You can check the users file for the complete content –

```
$ cat users
oko       tty01   Sep 12 07:30
ai        tty15   Sep 12 13:32
ruth      tty21   Sep 12 10:10
pat       tty24   Sep 12 13:07
steve     tty25   Sep 12 13:03
$
```

If a command has its output redirected to a file and the file already contains some data, that data will be lost. Consider the following example –

```
$ echo line 1 > users
$ cat users
line 1
$
```

You can use >> operator to append the output in an existing file as follows –

```
$ echo line 2 >> users
$ cat users
line 1
line 2
$
```

INPUT REDIRECTION

Just as the output of a command can be redirected to a file, so can the input of a command be redirected from a file. As the greater-than character > is used for output redirection, the less-than character < is used to redirect the input of a command.

The commands that normally take their input from the standard input can have their input redirected from a file in this manner. For example, to count the number of lines in the file users generated above, you can execute the command as follows –

```
$ wc -l users
2 users
$
```

Upon execution, you will receive the following output. You can count the number of lines in the file by redirecting the standard input of the wc command from the file users –

```
$ wc -l < users
2
$
```

Note that there is a difference in the output produced by the two forms of the wc command. In the first case, the name of the file users is listed with the line count; in the second case, it is not.

In the first case, wc knows that it is reading its input from the file users. In the second case, it only knows that it is reading its input from standard input so it does not display file name.

HERE DOCUMENT

A here document is used to redirect input into an interactive shell script or program.

We can run an interactive program within a shell script without user action by supplying the required input for the interactive program, or interactive shell script.

The general form for a here document is –

```
command << delimiter
document
delimiter
```

Here the shell interprets the << operator as an instruction to read input until it finds a line containing the specified delimiter. All the input lines up to the line containing the delimiter are then fed into the standard input of the command.

The delimiter tells the shell that the here document has completed. Without it, the shell continues to read the input forever. The delimiter must be a single word that does not contain spaces or tabs.

Following is the input to the command wc -l to count the total number of lines –

```
$wc -l << EOF
  This is a simple lookup program
```

```
        for good (and bad) restaurants
        in Cape Town.
EOF
3
$
```

You can use the here document to print multiple lines using your script as follows –

```
#!/bin/sh
cat << EOF
This is a simple lookup program
for good (and bad) restaurants
in Cape Town.
EOF
```

Upon execution, you will receive the following result –

```
This is a simple lookup program
for good (and bad) restaurants
in Cape Town.
```

The following script runs a session with the vi text editor and saves the input in the file test.txt.

```
#!/bin/sh
filename=test.txt
vi $filename <<EndOfCommands
i
This file was created automatically from
a shell script
^[
ZZ
EndOfCommands
```

If you run this script with vim acting as vi, then you will likely see output like the following –

```
$ sh test.sh
Vim: Warning: Input is not from a terminal
$
```

After running the script, you should see the following added to the file test.txt –

```
$ cat test.txt
This file was created automatically from
a shell script
$
```

DISCARD THE OUTPUT

Sometimes you will need to execute a command, but you don't want the output displayed on the screen. In such cases, you can discard the output by redirecting it to the file /dev/null –

$ command > /dev/null

Here command is the name of the command you want to execute. The file /dev/null is a special file that automatically discards all its input.

To discard both output of a command and its error output, use standard redirection to redirect STDERR to STDOUT –

$ command > /dev/null 2>&1

Here 2 represents STDERR and 1 represents STDOUT. You can display a message on to STDERR by redirecting STDOUT into STDERR as follows –

$ echo message 1>&2

Redirection Commands

Following is a complete list of commands which you can use for redirection –

SR.NO.	COMMAND	DESCRIPTION
1	pgm > file	Output of pgm is redirected to file
2	pgm < file	Program pgm reads its input from file
3	pgm >> file	Output of pgm is appended to file
4	n > file	Output from stream with descriptor n redirected to file
5	n >> file	Output from stream with descriptor n appended to file
6	n >& m	Merges output from stream n with stream m
7	n <& m	Merges input from stream n with stream m
8	<< tag	Standard input comes from here through next tag at the start of line
9	|	Takes output from one program, or process, and sends it to another

Note that the file descriptor 0 is normally standard input (STDIN), 1 is standard output (STDOUT), and 2 is standard error output (STDERR).

USER DEFINED FUNCTION

Using functions to perform repetitive tasks is an excellent way to create code reuse. This is an important part of modern object-oriented programming principles.

Shell functions are similar to subroutines, procedures, and functions in other programming languages.

Creating Functions

To declare a function, simply use the following syntax –

```
function_name () {
  list of commands
}
```

The name of your function is function_name, and that's what you will use to call it from elsewhere in your scripts. The function name must be followed by parentheses, followed by a list of commands enclosed within braces.

Example

Following example shows the use of function –

```
#!/bin/sh
# Define your function here
Hello () {
  echo "Hello World"
}

# Invoke your function
Hello
```

Upon execution, you will receive the following output –

```
$./test.sh
Hello World
```

PASS PARAMETERS TO A FUNCTION

You can define a function that will accept parameters while calling the function. These parameters would be represented by $1, $2 and so on.

Following is an example where we pass two parameters Manish and Dodia and then we capture and print these parameters in the function.

```
#!/bin/sh
# Define your function here
Hello () {
  echo "Hello World $1 $2"
}

# Invoke your function
Hello Manish Dodia
```

Upon execution, you will receive the following result –

```
$./test.sh
Hello World Manish Dodia
```

RETURNING VALUES FROM FUNCTIONS

If you execute an exit command from inside a function, its effect is not only to terminate execution of the function but also of the shell program that called the function.

If you instead want to just terminate execution of the function, then there is way to come out of a defined function.

Based on the situation you can return any value from your function using the return command whose syntax is as follows –

```
return code
```

Here code can be anything you choose here, but obviously you should choose something that is meaningful or useful in the context of your script as a whole.

Example

Following function returns a value 10 –

```
#!/bin/sh
# Define your function here
Hello () {
  echo "Hello World $1 $2"
  return 10
}
# Invoke your function
Hello Manish Dodia
# Capture value returnd by last command
ret=$?
echo "Return value is $ret"
```

Upon execution, you will receive the following result –

```
$./test.sh
Hello World Manish Dodia
Return value is 10
```


NESTED FUNCTIONS

One of the more interesting features of functions is that they can call themselves and also other functions. A function that calls itself is known as a recursive function.

Following example demonstrates nesting of two functions –

```
#!/bin/sh
# Calling one function from another
number_one () {
```

```
  echo "This is the first function speaking..."
  number_two
}
number_two () {
  echo "This is now the second function speaking..."
}
# Calling function one.
number_one
```

Upon execution, you will receive the following result −

```
This is the first function speaking...
This is now the second function speaking...
```

FUNCTION CALL FROM PROMPT

You can put definitions for commonly used functions inside your .profile. These definitions will be available whenever you log in and you can use them at the command prompt.

Alternatively, you can group the definitions in a file, say test.sh, and then execute the file in the current shell by typing −

```
$. test.sh
```

This has the effect of causing functions defined inside test.sh to be read and defined to the current shell as follows −

```
$ number_one
This is the first function speaking...
This is now the second function speaking...
$
```

To remove the definition of a function from the shell, use the unset command with the .f option. This command is also used to remove the definition of a variable to the shell.

```
$ unset -f function_name
```


UNIX HELP

All the Unix commands come with a number of optional and mandatory options. It is very common to forget the complete syntax of these commands.

Because no one can possibly remember every Unix command and all its options, we have online help available to mitigate this right from when Unix was at its development stage.

Unix's version of Help files are called man pages. If there is a command name and you are not sure how to use it, then Man Pages help you out with every step.

Syntax

Here is the simple command that helps you get the detail of any Unix command while working with the system –

$man command

Example

Suppose there is a command that requires you to get help; assume that you want to know about pwd then you simply need to use the following command –

$man pwd

The above command helps you with the complete information about the pwd command. Try it yourself at your command prompt to get more detail. You can get complete detail on man command itself using the following command –

$man man

Man Page Sections

Man pages are generally divided into sections, which generally vary by the man page author's preference. Following table lists some common sections –

SR.NO.	COMMAND	DESCRIPTION
1	pgm > file	Output of pgm is redirected to file
2	pgm < file	Program pgm reads its input from file
3	pgm >> file	Output of pgm is appended to file
4	n > file	Output from stream with descriptor n redirected to file
5	n >> file	Output from stream with descriptor n appended to file
6	n >& m	Merges output from stream n with stream m
7	n <& m	Merges input from stream n with stream m
8	<< tag	Standard input comes from here through next tag at the start of line
9	|	Takes output from one program, or process, and sends it to another

To sum it up, man pages are a vital resource and the first avenue of research when you need information about commands or files in a Unix system.

3.8 BACKGROUND PROCESS & PRIORITIES OF PROCESS

In Unix, all the instructions outside the kernel are executed in the context of a process.
A process is a sequence of instructions and each process has a block of controlled data associated with it. Processes can be manipulated in a way similar to how files can be manipulated.

UNIX PROCESS INFORMATION

The process table contains information of all the processes that are running currently. The main purpose of this table is to manage all the running processes effectively.
Note: As a multitasking and multi-user OS, Unix will have many processes running at the same time.

THE TABLE CONTAINS INFORMATION LIKE:

- Process Id
- Parent Process ID
- State of the processes
- CPU usage

Processes in Unix go through various states depending on various circumstances. The state changes may be triggered by whether a process needs to wait for a read or write operation to complete, or when a different more urgent process needs to be given a chance to run.
When you start a process (run a command), there are two ways you can run it –

- Foreground Processes
- Background Processes

FOREGROUND PROCESSES

By default, every process that you start runs in the foreground. It gets its input from the keyboard and sends its output to the screen.
You can see this happen with the ls command. If you wish to list all the files in your current directory, you can use the following command –

```
$ls ch*.doc
```

This would display all the files, the names of which start with ch and end with .doc –

```
ch01-1.doc  ch010.doc  ch02.doc  ch03-2.doc
ch04-1.doc  ch040.doc  ch05.doc  ch06-2.doc
```

ch01-2.doc ch02-1.doc

The process runs in the foreground, the output is directed to my screen, and if the ls command wants any input (which it does not), it waits for it from the keyboard.

While a program is running in the foreground and is time-consuming, no other commands can be run (start any other processes) because the prompt would not be available until the program finishes processing and comes out.

BACKGROUND PROCESSES

A background process runs without being connected to your keyboard. If the background process requires any keyboard input, it waits.

The advantage of running a process in the background is that you can run other commands; you do not have to wait until it completes to start another!

The simplest way to start a background process is to add an ampersand (&) at the end of the command.

$ls ch*.doc &

This displays all those files the names of which start with ch and end with .doc –

ch01-1.doc ch010.doc ch02.doc ch03-2.doc
ch04-1.doc ch040.doc ch05.doc ch06-2.doc
ch01-2.doc ch02-1.doc

Here, if the ls command wants any input (which it does not), it goes into a stop state until we move it into the foreground and give it the data from the keyboard.

That first line contains information about the background process - the job number and the process ID. You need to know the job number to manipulate it between the background and the foreground.

Press the Enter key and you will see the following –

[1] + Done ls ch*.doc &
$

The first line tells you that the ls command background process finishes successfully. The second is a prompt for another command.

LISTING RUNNING PROCESSES

It is easy to see your own processes by running the ps (process status) command as follows –

$ps

PID	TTY	TIME	CMD
18358	ttyp3	00:00:00	sh
18361	ttyp3	00:01:31	abiword

18789 ttyp3 00:00:00 ps

One of the most commonly used flags for ps is the -f (f for full) option, which provides more information as shown in the following example –

```
$ps -f
UID   PID  PPID C STIME   TTY   TIME CMD
amrood  6738 3662 0 10:23:03 pts/6 0:00 first_one
amrood  6739 3662 0 10:22:54 pts/6 0:00 second_one
amrood  3662 3657 0 08:10:53 pts/6 0:00 -ksh
amrood  6892 3662 4 10:51:50 pts/6 0:00 ps -f
```

Here is the description of all the fields displayed by ps -f command –

SR.NO.	COLUMN	DESCRIPTION
1	UID	User ID that this process belongs to (the person running it)
2	PID	Process ID
3	PPID	Parent process ID (the ID of the process that started it)
4	C	CPU utilization of process
5	STIME	Process start time
6	TTY	Terminal type associated with the process
7	TIME	CPU time taken by the process
8	CMD	The command that started this process

There are other options which can be used along with ps command –

SR.NO.	OPTION	DESCRIPTION
1	-a	Shows information about all users
2	-x	Shows information about processes without terminals
3	-u	Shows additional information like -f option
4	-e	Displays extended information

STOPPING PROCESSES

Ending a process can be done in several different ways. Often, from a console-based command, sending a CTRL + C keystroke (the default interrupt character) will exit the command. This works when the process is running in the foreground mode.

If a process is running in the background, you should get its Job ID using the ps command. After that, you can use the kill command to kill the process as follows –

```
$ps -f
UID    PID  PPID C STIME   TTY   TIME CMD
amrood  6738 3662 0 10:23:03 pts/6 0:00 first_one
amrood  6739 3662 0 10:22:54 pts/6 0:00 second_one
amrood  3662 3657 0 08:10:53 pts/6 0:00 -ksh
amrood  6892 3662 4 10:51:50 pts/6 0:00 ps -f
$kill 6738
Terminated
```

Here, the kill command terminates the first_one process. If a process ignores a regular kill command, you can use kill -9 followed by the process ID as follows –

```
$kill -9 6738
Terminated
```

PARENT AND CHILD PROCESSES

Each unix process has two ID numbers assigned to it: The Process ID (pid) and the Parent process ID (ppid). Each user process in the system has a parent process.

Most of the commands that you run have the shell as their parent. Check the ps -f example where this command listed both the process ID and the parent process ID.

ZOMBIE AND ORPHAN PROCESSES

Normally, when a child process is killed, the parent process is updated via a SIGCHLD signal. Then the parent can do some other task or restart a new child as needed. However, sometimes the parent process is killed before its child is killed. In this case, the "parent of all processes," the init process, becomes the new PPID (parent process ID). In some cases, these processes are called orphan processes.

When a process is killed, a ps listing may still show the process with a Z state. This is a zombie or defunct process. The process is dead and not being used. These processes are different from the orphan processes. They have completed execution but still find an entry in the process table.

DAEMON PROCESSES

Daemons are system-related background processes that often run with the permissions of root and services requests from other processes.

A daemon has no controlling terminal. It cannot open /dev/tty. If you do a "ps -ef" and look at the tty field, all daemons will have a ? for the tty.

To be precise, a daemon is a process that runs in the background, usually waiting for something to happen that it is capable of working with. For example, a printer daemon waiting for print commands.

If you have a program that calls for lengthy processing, then it's worth to make it a daemon and run it in the background.

THE TOP COMMAND

The top command is a very useful tool for quickly showing processes sorted by various criteria.

It is an interactive diagnostic tool that updates frequently and shows information about physical and virtual memory, CPU usage, load averages, and your busy processes.

Here is the simple syntax to run top command and to see the statistics of CPU utilization by different processes –

EXAMPLE:

$ grep "how" file1 > file2 &

This command will search for the lines containing the pattern "how" and will store the output in file2. If file1 is reasonably large, we may want to proceed with other actions, and run this command in the background. With the '&' at the end, this command will run as a process in the background.

It is possible to suspend a foreground process by using the '^Z' command. This command moves the current foreground process to a suspended state in the background. From there, the process can either be pushed to resume in the background using the 'bg' command, or it can be brought back to the foreground using the 'fg' command.

PROCESS PRIORITIES

Each process is also associated with a priority. This is used to ensure that the OS is able to fairly allocate time to various processing tasks. The 'nice' command can be used to reduced the priority of a process and thus be 'nice' to the other processes, i.e.

$ nice <command>

This line will run the specified command at a lower priority – by default, the priority will be reduced by 10. The command also takes a parameter that can be used to use a different level of 'niceness'.

Example:

$ nice -20 ls

This command runs 'ls' with the priority reduced by 20.

It is also possible to increase the priority with a negative 'niceness'. However, this needs superuser permission.

When a terminal or login session is closed, it sends the SIGHUP signal to thr child processes. By default, this signal will cause the child processes to terminate. The 'nohup' command can be used to allow commands to continue running even when the login session is terminated.

Example:

$ nohup soft file1 > file2

With this command, sorting of file1 and saving in file2 process will continue even if we have logged out of the system.

The 'kill' command can be used to terminate any of the processes depending on permissions.

Example:

$ kill [options] <pid>

This command will terminate a process with the process id <pid>. The PID of a process can be obtained using the 'ps' command.

This 'at' command is used to execute commands at a particular date and time in the future.

Example:

$ at 8pm

sort file1>file2

UNIX DEBUGGING

Unix provides a number of mechanisms to help find bugs in your command scripts. These mechanisms can be used to view a trace of what is being executed i.e. the sequence in which commands are executed. The trace can be used to understand and verify the logic and control flow of the script.

=> set -v

verbose mode: Setting this option before running a command will ensure that the command that will be executed is printed to stdout before it is actually executed.

=> set -x

execution trace mode: Setting this option will display each command as it is executed along with its arguments.

=> set -n

no-exec mode: Setting this option shows any errors without actually running any commands.

A process is a context in which a program executes. Every time when a command or program is run, a new process is created. The process is active for as long as the program is in an active state.

For Example, if we are executing the cat command then a process named "cat" is generated.

Each time a new process is created, the Kernel assigns a unique identification number called the PID i.e. process identification number) which lies in between 0 to 32,767. Other properties of processes include their PPID (Parent PID), TTY (the controlling terminal from where they were launched), UID (the user id that owns this process) and GID (the group that is associated with the process).

3.9 CONDITIONAL EXECUTION

The test command is used to check file types and compare values. You can also use [as test command. It is used for:

- File attributes comparisons
- Perform string comparisons.
- Arithmetic comparisons.

Syntax

```
[ condition ]
OR
[ ! condition ]
OR
[ condition ] && true-command
OR
[ condition ] || false-command
OR
[ condition ] && true-command || false-command
```

Examples

```
[ 5 == 5 ] && echo "Yes" || echo "No"
[ 5 == 15 ] && echo "Yes" || echo "No"
[ 5 != 10 ] && echo "Yes" || echo "No"
[ -f /etc/resolv.conf ] && echo "File /etc/resolv.conf found." || echo "File /etc/resolv.conf not found."
[ -f /etc/resolv1.conf ] && echo "File /etc/resolv.conf found." || echo "File /etc/resolv.conf not found."
```

PRACTICAL SHELL SCRIPT

1.

```
#Initializing two variables
a=10
b=20
 #Check whether they are equal
if [ $a == $b ]
then
   echo "a is equal to b"
fi
 #Check whether they are not equal
if [ $a != $b ]
then
   echo "a is not equal to b"
fi
```

RUNNING

```
$bash -f main.sh
a is not equal to b
```

2.

```
#Initializing two variables
a=20
b=20
 if [ $a == $b ]
then
   #If they are equal then print this
   echo "a is equal to b"
else
   #else print this
   echo "a is not equal to b"
fi
```

RUNNING

```
$bash -f main.sh
a is equal to b
```

3.

```
CARS="bmw"
 #Pass the variable in string
case "$CARS" in
   #case 1
```

```
   "mercedes") echo "Headquarters - Affalterbach, Germany" ;;

   #case 2
   "audi") echo "Headquarters - Ingolstadt, Germany" ;;

   #case 3
   "bmw") echo "Headquarters - Chennai, Tamil Nadu, India" ;;
esac
```

RUNNING

```
$bash -f main.sh
Headquarters - Chennai, Tamil Nadu, India.
```