11. Unix — What is Shell?

A **Shell** provides you with an interface to the Unix system. It gathers input from you and executes programs based on that input. When a program finishes executing, it displays that program's output.

Shell is an environment in which we can run our commands, programs, and shell scripts. There are different flavors of a shell, just as there are different flavors of operating systems. Each flavor of shell has its own set of recognized commands and functions.

Shell Prompt

The prompt, **$**, which is called the **command prompt**, is issued by the shell. While the prompt is displayed, you can type a command.

Shell reads your input after you press **Enter**. It determines the command you want executed by looking at the first word of your input. A word is an unbroken set of characters. Spaces and tabs separate words.

Following is a simple example of the **date** command, which displays the current date and time:

```
$date
Thu Jun 25 08:30:19 MST 2009
```

You can customize your command prompt using the environment variable PS1 explained in the Environment tutorial.

Shell Types

In Unix, there are two major types of shells:

- **Bourne shell** — If you are using a Bourne-type shell, the **$** character is the default prompt.
- **C shell** — If you are using a C-type shell, the **%** character is the default prompt.

The Bourne Shell has the following subcategories –

- Bourne shell (sh)
- Korn shell (ksh)
- Bourne Again shell (bash)
- POSIX shell (sh)

The different C-type shells follow –

- C shell (csh)
- TENEX/TOPS C shell (tcsh)

The original Unix shell was written in the mid-1970s by Stephen R. Bourne while he was at the AT&T Bell Labs in New Jersey.

Bourne shell was the first shell to appear on Unix systems, thus it is referred to as "the shell".

Bourne shell is usually installed as **/bin/sh** on most versions of Unix. For this reason, it is the shell of choice for writing scripts that can be used on different versions of Unix.

In this chapter, we are going to cover most of the Shell concepts that are based on the Borne Shell.

Shell Scripts

The basic concept of a shell script is a list of commands, which are listed in the order of execution. A good shell script will have comments, preceded by **#** sign, describing the steps.

There are conditional tests, such as value A is greater than value B, loops allowing us to go through massive amounts of data, files to read and store data, and variables to read and store data, and the script may include functions.

We are going to write many scripts in the next sections. It would be a simple text file in which we would put all our commands and several other required constructs that tell the shell environment what to do and when to do it.

Shell scripts and functions are both interpreted. This means they are not compiled.

Example Script

Assume we create a **test.sh** script. Note all the scripts would have the **.sh** extension. Before you add anything else to your script, you need to alert the system that a shell script is being started. This is done using the **shebang** construct. For example –

```
#!/bin/sh
```

This tells the system that the commands that follow are to be executed by the Bourne shell. *It's called a shebang because the* **#** *symbol is called a hash, and the* **!** *symbol is called a bang.*

To create a script containing these commands, you put the shebang line first and then add the commands –

```
#!/bin/bash
pwd
ls
```

Shell Comments

You can put your comments in your script as follows –

```
#!/bin/bash


# Author : Zara Ali
# Copyright (c) Tutorialspoint.com
# Script follows here:
pwd
ls
```

Save the above content and make the script executable –

```
$chmod +x test.sh
```

The shell script is now ready to be executed –

```
$./test.sh
```

Upon execution, you will receive the following result –

```
/home/amrood
index.htm  unix-basic_utilities.htm  unix-directories.htm
test.sh    unix-communication.htm    unix-environment.htm
```

Note: To execute a program available in the current directory, use **./program_name**

Extended Shell Scripts

Shell scripts have several required constructs that tell the shell environment what to do and when to do it. Of course, most scripts are more complex than the above one.

The shell is, after all, a real programming language, complete with variables, control structures, and so forth. No matter how complicated a script gets, it is still just a list of commands executed sequentially.

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 : Zara Ali
# Copyright (c) Tutorialspoint.com
# 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?
Zara Ali
Hello, Zara Ali
$
```

12. Unix — Using Shell Variables

In this chapter, we will learn how to use Shell variables in Unix. 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 valid variable names −

```
_ALI

TOKEN_A

VAR_1

VAR_2
```

Following are the examples of invalid 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="Zara Ali"
```

The above example defines the variable NAME and assigns the value "Zara Ali" 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="Zara Ali"
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="Zara Ali"
echo $NAME
```

The above script will produce the following value –

```
Zara Ali
```

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="Zara Ali"
readonly NAME
NAME="Qadiri"
```

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="Zara Ali"
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.

13. Unix — Special Variables

In this chapter, we will discuss in detail about special variable in Unix. In one of our previous chapters, we understood how to be careful when we use certain non-alphanumeric characters in variable names. This is because those characters are used in the names of special Unix variables. These variables are reserved for specific functions.

For example, the **$** character represents the process ID number, or PID, of the current shell:

```
$echo $$
```

The above command writes the PID of the current shell –

```
29949
```

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

Variable	Description
$0	The filename of the current script.
$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).
$#	The number of arguments supplied to a script.
$*	All the arguments are double quoted. If a script receives two arguments, $* is equivalent to $1 $2.
$@	All the arguments are individually double quoted. If a script receives two arguments, $@ is equivalent to $1 $2.
$?	The exit status of the last command executed.
$$	The process number of the current shell. For shell scripts, this is the process ID under which they are executing.
$!	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 Zara Ali
File Name : ./test.sh
First Parameter : Zara
Second Parameter : Ali
Quoted Values: Zara Ali
Quoted Values: Zara Ali
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 command-line 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 Zara Ali 10 Years Old
Zara
Ali
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 Zara Ali
File Name : ./test.sh
First Parameter : Zara
Second Parameter : Ali
Quoted Values: Zara Ali
Quoted Values: Zara Ali
Total Number of Parameters : 2
$echo $?
0
$
```

14. Unix — Using Shell Arrays

In this chapter, we will discuss how to use shell arrays in Unix. A shell variable is capable enough to hold a single value. These variables are called scalar variables.

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="Zara"
NAME02="Qadir"
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]="Zara"
NAME[1]="Qadir"
NAME[2]="Mahnaz"
NAME[3]="Ayan"
NAME[4]="Daisy"
```

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

```
set -A array_name value1 value2 ... valuen
```

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

```
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]="Zara"
NAME[1]="Qadir"
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: Zara
Second Index: Qadir
```

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]="Zara"
NAME[1]="Qadir"
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: Zara Qadir Mahnaz Ayan Daisy
Second Method: Zara Qadir Mahnaz Ayan Daisy
```

15. Unix — Shell Basic 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 inverted commas.

Arithmetic Operators

The following arithmetic operators are supported by Bourne Shell.

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

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 % $a\` will give 0
=	Assignment - Assigns right operand in left operand	a=$b would assign value of b into a
==	Equality - Compares two numbers, if both are same then returns true.	[$a == $b] would return false.
!=	Not Equality - Compares two numbers, if both are different then returns true.	[$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.

Unix - Shell Arithmetic Operators Example

Here is an example which uses all the arithmetic operators –

```
#!/bin/sh


a=10
b=20
val=`expr $a + $b`


echo "a + b : $val"
```

```

val=`expr $a - $b`
echo "a - b : $val"

val=`expr $a \* $b`
echo "a * b : $val"

val=`expr $b / $a`
echo "b / a : $val"

val=`expr $b % $a`
echo "b % a : $val"

if [ $a == $b ]
then
   echo "a is equal to b"
fi

if [ $a != $b ]
then
   echo "a is not equal to b"
fi
```

The above script will produce the following result –

```
a + b : 30
a - b : -10
a * b : 200
b / a : 2
b % a : 0
a is not equal to b
```

The following points need to be considered when using the Arithmetic Operators –

- There must be spaces between the operators and the expressions. For example, 2+2 is not correct; it should be written as 2 + 2.
- Complete expression should be enclosed between **‘ ‘**, called the inverted commas.
- You should use **** on the ***** symbol for multiplication.

- **if...then...fi** statement is a decision-making statement which has been explained in the next chapter.

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 –

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.

Unix - Shell Relational Operators Example

Here is an example which uses all the relational operators –

```
#!/bin/sh


a=10
b=20
```

```
if [ $a -eq $b ]
then
   echo "$a -eq $b : a is equal to b"
else
   echo "$a -eq $b: a is not equal to b"
fi

if [ $a -ne $b ]
then
   echo "$a -ne $b: a is not equal to b"
else
   echo "$a -ne $b : a is equal to b"
fi

if [ $a -gt $b ]
then
   echo "$a -gt $b: a is greater than b"
else
   echo "$a -gt $b: a is not greater than b"
fi

if [ $a -lt $b ]
then
   echo "$a -lt $b: a is less than b"
else
   echo "$a -lt $b: a is not less than b"
fi

if [ $a -ge $b ]
then
   echo "$a -ge $b: a is greater or  equal to b"
else
   echo "$a -ge $b: a is not greater or equal to b"
fi

if [ $a -le $b ]
```

```
then
   echo "$a -le $b: a is less or  equal to b"
else
   echo "$a -le $b: a is not less or equal to b"
fi
```

The above script will generate the following result –

```
10 -eq 20: a is not equal to b
10 -ne 20: a is not equal to b
10 -gt 20: a is not greater than b
10 -lt 20: a is less than b
10 -ge 20: a is not greater or equal to b
10 -le 20: a is less or  equal to b
```

The following points need to be considered while working with relational operators –

- There must be spaces between the operators and the expressions. For example, 2+2 is not correct; it should be written as 2 + 2.
- **if...then...else...fi** statement is a decision-making statement which has been explained in the next chapter.

Boolean Operators

The following Boolean operators are supported by the Bourne Shell.

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

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.

Unix - Shell Boolean Operators Example

Here is an example which uses all the Boolean operators −

```
#!/bin/sh

a=10
b=20

if [ $a != $b ]
then
   echo "$a != $b : a is not equal to b"
else
   echo "$a != $b: a is equal to b"
fi

if [ $a -lt 100 -a $b -gt 15 ]
then
   echo "$a -lt 100 -a $b -gt 15 : returns true"

else
   echo "$a -lt 100 -a $b -gt 15 : returns false"
fi

if [ $a -lt 100 -o $b -gt 100 ]
then
   echo "$a -lt 100 -o $b -gt 100 : returns true"
else
   echo "$a -lt 100 -o $b -gt 100 : returns false"
fi

if [ $a -lt 5 -o $b -gt 100 ]
then
   echo "$a -lt 100 -o $b -gt 100 : returns true"
else
   echo "$a -lt 100 -o $b -gt 100 : returns false"
fi
```

The above script will generate the following result –

```
10 != 20 : a is not equal to b
10 -lt 100 -a 20 -gt 15 : returns true
10 -lt 100 -o 20 -gt 100 : returns true
10 -lt 5 -o 20 -gt 100 : returns false
```

The following points need to be considered while using the operators –

- There must be spaces between the operators and the expressions. For example, 2+2 is not correct; it should be written as 2 + 2.
- **if...then...else...fi** statement is a decision-making statement which has been explained in the next chapter.

String Operators

The following string operators are supported by Bourne Shell.

Assume variable **a** holds "abc" and variable **b** holds "efg" then –

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 non-zero 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.

Unix - Shell String Operators Example

Here is an example which uses all the string operators –

```
#!/bin/sh


a="abc"
b="efg"


if [ $a = $b ]
then
   echo "$a = $b : a is equal to b"
else


   echo "$a = $b: a is not equal to b"
fi


if [ $a != $b ]
then
   echo "$a != $b : a is not equal to b"
else
   echo "$a != $b: a is equal to b"
fi


if [ -z $a ]
then
   echo "-z $a : string length is zero"
else
   echo "-z $a : string length is not zero"
fi


if [ -n $a ]
then
   echo "-n $a : string length is not zero"
else
   echo "-n $a : string length is zero"
fi
```

```
if [ $a ]
then
   echo "$a : string is not empty"
else
   echo "$a : string is empty"
fi
```

The above script will generate the following result –

```
abc = efg: a is not equal to b
abc != efg : a is not equal to b
-z abc : string length is not zero
-n abc : string length is not zero
abc : string is not empty
```

The following points need to be considered while using the operator –

- There must be spaces between the operators and the expressions. For example, 2+2 is not correct. It should be written as 2 + 2.
- **if...then...else...fi** statement is a decision-making statement which has been explained in the next chapter.

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 –

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.

Unix - Shell File Test Operators Example

The following example uses all the **file test** operators –

Assume a variable file holds an existing file name **"/var/www/tutorialspoint/unix/test.sh"** the size of which is 100 bytes and has **read**, **write** and **execute** permission –

```
#!/bin/sh


file="/var/www/tutorialspoint/unix/test.sh"


if [ -r $file ]
then
```

```
   echo "File has read access"
else
   echo "File does not have read access"
fi

if [ -w $file ]
then
   echo "File has write permission"
else
   echo "File does not have write permission"
fi


if [ -x $file ]
then
   echo "File has execute permission"
else
   echo "File does not have execute permission"
fi

if [ -f $file ]
then
   echo "File is an ordinary file"
else
   echo "This is sepcial file"
fi

if [ -d $file ]
then
   echo "File is a directory"
else
   echo "This is not a directory"
fi
if [ -s $file ]
then
   echo "File size is zero"
else
   echo "File size is not zero"
```

```
fi

if [ -e $file ]
then
   echo "File exists"
else
   echo "File does not exist"
fi
```

The above script will produce the following result –

```
File has read access
File has write permission
File has execute permission

File is an ordinary file
This is not a directory
File size is zero
File exists
```

The following points need to be considered while using file test operators –

- There must be spaces between the operators and the expressions. For example, 2+2 is not correct; it should be written as 2 + 2.
- **if...then...else...fi** statement is a decision-making statement which has been explained in the next chapter.

C Shell Operators

Following link will give you a brief idea on C Shell Operators: C Shell Operators

Unix - C Shell Operators

We will now list down all the operators available in C Shell. Here most of the operators are very similar to what we have in C Programming language.

Operators are listed in the order of decreasing precedence –

Arithmetic and Logical Operators

The following table lists out a few Arithmetic and Logical Operators:

Operator	Description

( )	Change precedence
~	1's complement
!	Logical negation
*****	Multiply
/	Divide
%	Modulo
+	Add
-	Subtract
< <	Left shift
> >	Right shift
= =	String comparison for equality
! =	String comparison for non-equality
= ~	Pattern matching
&	Bitwise "and"
^	Bitwise "exclusive or"
|	Bitwise "inclusive or"
& &	Logical "and"
| |	Logical "or"
+ +	Increment

--	Decrement
=	Assignment
*** =**	Multiply left side by right side and update left side
/ =	Divide left side by right side and update left side
+ =	Add left side to right side and update left side
- =	Subtract left side from right side and update left side
^ =	"Exclusive or" left side to right side and update left side
% =	Divide left by right side and update left side with remainder

File Test Operators

The following operators test various properties associated with a Unix file.

Operator	Description
-r file	Checks if file is readable; if yes, then the condition becomes true.
-w file	Checks if file is writable; if yes, then the condition becomes true.
-x file	Checks if file is executable; if yes, then the condition becomes 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.
-z file	Checks if file has size greater than 0; if yes, then the condition becomes true.
-d file	Checks if file is a directory; if yes, then the condition becomes true.
-e file	Checks if file exists; is true even if file is a directory but exists.
-o file	Checks if user owns the file; returns true if the user is the owner of the file.

Korn Shell Operators

Following link helps you understand Korn Shell Operators: Korn Shell Operators

Unix - Korn Shell Operators

We will now discuss all the operators available in Korn Shell. Most of the operators are very similar to what we have in the C Programming language.

Operators are listed in the order of decreasing precedence –

Arithmetic and Logical Operators

Operator	Description
+	Unary plus
-	Unary minus
! ~	Logical negation; binary inversion (one's complement)
*****	Multiply
/	Divide
%	Modulo
+	Add
-	Subtract
< <	Left shift
> >	Right shift
= =	String comparison for equality
! =	String comparison for non-equality
= ~	Pattern matching
&	Bitwise "and"

^	Bitwise "exclusive or"
|	Bitwise "inclusive or"
&&	Logical "and"
||	Logical "or"
++	Increment
--	Decrement
=	Assignment

File Test Operators

Following operators test various properties associated with a Unix file.

Operator	Description
-r file	Checks if file is readable; if yes, then the condition becomes true.
-w file	Checks if file is writable; if yes, then the condition becomes true.
-x file	Checks if file is executable; if yes, then the condition becomes 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.
-s file	Checks if file has size greater than 0; if yes, then the condition becomes true.
-d file	Checks if file is a directory; if yes, then the condition becomes true.
-e file	Checks if file exists; is true even if file is a directory but exists.

16. Unix — Shell Decision Making

In this chapter, we will understand shell decision-making in Unix. While writing a shell script, there may be a situation when you need to adopt one path out of the given two paths. So you need to make use of conditional statements that allow your program to make correct decisions and perform the right actions.

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

Unix Shell - The if...fi statement

The **if...fi** statement is the fundamental control statement that allows Shell to make decisions and execute statements conditionally.

Syntax

```
if [ expression ]
then
   Statement(s) to be executed if expression is true
fi
```

The Shell *expression* is evaluated in the above syntax. If the resulting value is *true*, given *statement(s)* are executed. If the *expression* is *false* then no statement would be executed. Most of the times, comparison operators are used for making decisions.

It is recommended to be careful with the spaces between braces and expression. No space produces a syntax error.

If **expression** is a shell command, then it will be assumed true if it returns **0** after execution. If it is a Boolean expression, then it would be true if it returns true.

Example

```
#!/bin/sh

a=10
b=20

if [ $a == $b ]
then
   echo "a is equal to b"
fi

if [ $a != $b ]
then
   echo "a is not equal to b"
fi
```

The above script will generate the following result –

```
a is not equal to b
```

Unix Shell - The if...else...fi statement

The **if...else...fi** statement is the next form of control statement that allows Shell to execute statements in a controlled way and make the right choice.

Syntax

```
if [ expression ]
then
   Statement(s) to be executed if expression is true
else
   Statement(s) to be executed if expression is not true
fi
```

The Shell *expression* is evaluated in the above syntax. If the resulting value is *true*, given *statement(s)* are executed. If the *expression* is *false*, then no statement will be executed.

Example

The above example can also be written using the *if...else* statement as follows –

```
#!/bin/sh

a=10
b=20

if [ $a == $b ]
then
   echo "a is equal to b"
else
   echo "a is not equal to b"
fi
```

Upon execution, you will receive the following result –

```
a is not equal to b
```

Unix Shell - The if...elif...fi statement

The **if...elif...fi** statement is the one level advance form of control statement that allows Shell to make correct decision out of several conditions.

Syntax

```
if [ expression 1 ]
then
   Statement(s) to be executed if expression 1 is true
elif [ expression 2 ]
then
   Statement(s) to be executed if expression 2 is true
elif [ expression 3 ]
then
   Statement(s) to be executed if expression 3 is true
else
   Statement(s) to be executed if no expression is true
fi
```

This code is just a series of *if* statements, where each *if* is part of the *else* clause of the previous statement. Here statement(s) are executed based on the true condition, if none of the condition is true then *else* block is executed.

Example

```
#!/bin/sh


a=10

b=20


if [ $a == $b ]

then

   echo "a is equal to b"

elif [ $a -gt $b ]

then

   echo "a is greater than b"

elif [ $a -lt $b ]

then

   echo "a is less than b"

else

   echo "None of the condition met"

fi
```

Upon execution, you will receive the following result –

```
a is less than b
```

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

Unix Shell - 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.

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

Syntax

The basic syntax of the **case...esac** statement is to give an expression to evaluate and to execute several different statements based on the value of the expression.

The interpreter checks each case against the value of the expression until a match is found. If nothing matches, a default condition will be used.

```
case word in
  pattern1)
     Statement(s) to be executed if pattern1 matches
     ;;
  pattern2)
     Statement(s) to be executed if pattern2 matches
     ;;
  pattern3)
     Statement(s) to be executed if pattern3 matches
     ;;
esac
```

Here the string word is compared against every pattern until a match is found. The statement(s) following the matching pattern executes. If no matches are found, the case statement exits without performing any action.

There is no maximum number of patterns, but the minimum is one.

When statement(s) part executes, the command **;;** indicates that the program flow should jump to the end of the entire case statement. This is similar to break in the C programming language.

Example

```
#!/bin/sh

FRUIT="kiwi"

case "$FRUIT" in
   "apple") echo "Apple pie is quite tasty."
   ;;
   "banana") echo "I like banana nut bread."
   ;;
   "kiwi") echo "New Zealand is famous for kiwi."
   ;;
esac
```

Upon execution, you will receive the following result –

```
New Zealand is famous for kiwi.
```

A good use for a case statement is the evaluation of command line arguments as follows –

```
#!/bin/sh

option="${1}"
case ${option} in
   -f) FILE="${2}"
      echo "File name is $FILE"
      ;;
   -d) DIR="${2}"
      echo "Dir name is $DIR"
      ;;
   *)
      echo "`basename ${0}`:usage: [-f file] | [-d directory]"
      exit 1 # Command to come out of the program with status 1
      ;;
esac
```

Here is a sample run of the above program –

```
$./test.sh
test.sh: usage: [ -f filename ] | [ -d directory ]
$ ./test.sh -f index.htm
$ vi test.sh

$ ./test.sh -f index.htm
File name is index.htm
$ ./test.sh -d unix
Dir name is unix
$
```

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++** and **PERL**, etc.

17. Unix — Shell Loop Types

In this chapter, we will discuss shell loops in Unix. A loop is a powerful programming tool that enables you to execute a set of commands repeatedly. In this chapter, we will examine the following types of loops available to shell programmers –

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

Unix Shell - The while Loop

The **while** loop enables you to execute a set of commands repeatedly until some condition occurs. It is usually used when you need to manipulate the value of a variable repeatedly.

Syntax

```
while command
do
   Statement(s) to be executed if command is true
done
```

Here the Shell *command* is evaluated. If the resulting value is *true*, given *statement(s)* are executed. If *command* is *false* then no statement will be executed and the program will jump to the next line after the done statement.

Example

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

```
#!/bin/sh

a=0

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

Upon execution, you will receive the following result −

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

Each time this loop executes, the variable **a** is checked to see whether it has a value that is less than 10. If the value of **a** is less than 10, this test condition has an exit status of 0. In this case, the current value of **a** is displayed and later **a** is incremented by 1.

Unix Shell - The for Loop

The **for** loop operates on lists of items. It repeats a set of commands for every item in a list.

Syntax

```
for var in word1 word2 ... wordN
do
   Statement(s) to be executed for every word.
done
```

Here *var* is the name of a variable and word1 to wordN are sequences of characters separated by spaces (words). Each time the for loop executes, the value of the variable var is set to the next word in the list of words, word1 to wordN.

Example

Here is a simple example that uses the **for** loop to span through the given list of numbers –

```
#!/bin/sh
for var in 0 1 2 3 4 5 6 7 8 9
do
   echo $var
done
```

Upon execution, you will receive the following result –

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

Following is the example to display all the files starting with **.bash** and available in your home. We will execute this script from my root –

```
#!/bin/sh

for FILE in $HOME/.bash*
do
   echo $FILE
done
```

The above script will produce the following result –

```
/root/.bash_history
/root/.bash_logout
/root/.bash_profile
/root/.bashrc
```

Unix Shell - The until Loop

The while loop is perfect for a situation where you need to execute a set of commands while some condition is true. Sometimes you need to execute a set of commands until a condition is true.

Syntax

```
until command
do
```

```
   Statement(s) to be executed until command is true
done
```

Here the Shell *command* is evaluated. If the resulting value is *false*, given *statement(s)* are executed. If the *command* is *true* then no statement will be executed and the program jumps to the next line after the done statement.

Example

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

```
#!/bin/sh

a=0

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

Upon execution, you will receive the following result –

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

Unix Shell - The select Loop

The **select** loop provides an easy way to create a numbered menu from which users can select options. It is useful when you need to ask the user to choose one or more items from a list of choices.

Syntax

```
select var in word1 word2 ... wordN
do
   Statement(s) to be executed for every word.
done
```

Here *var* is the name of a variable and **word1** to **wordN** are sequences of characters separated by spaces (words). Each time the **for** loop executes, the value of the variable var is set to the next word in the list of words, **word1** to **wordN**.

For every selection, a set of commands will be executed within the loop. This loop was introduced in **ksh** and has been adapted into bash. It is not available in **sh**.

Example

Here is a simple example to let the user select a drink of choice –

```
#!/bin/ksh

select DRINK in tea cofee water juice appe all none
do
   case $DRINK in


      tea|cofee|water|all)
         echo "Go to canteen"
         ;;
      juice|appe)
         echo "Available at home"
      ;;
      none)
         break
      ;;
      *) echo "ERROR: Invalid selection"
      ;;
   esac
done
```

The menu presented by the select loop looks like the following –

```
$./test.sh
1) tea
2) cofee
3) water
4) juice
5) appe
6) all
7) none
#? juice
Available at home
#? none
$
```

You can change the prompt displayed by the select loop by altering the variable PS3 as follows –

```
$PS3="Please make a selection => " ; export PS3
$./test.sh
1) tea
2) cofee
3) water
4) juice
5) appe
6) all
7) none
Please make a selection => juice
Available at home
Please make a selection => none
$
```

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
```

18. Unix — Shell Loop Control

In this chapter, we will discuss shell loop control in Unix. So far you have looked at creating loops and working with loops to accomplish different tasks. Sometimes you need to stop a loop or skip iterations of the loop.

In this chapter, we will learn 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 **n**th 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 **n**th 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
```

19. Unix — Shell Substitution

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 can be used in the echo command –

Escape	Description
\	backslash
\a	alert (BEL)
\b	backspace
\c	suppress trailing newline
\f	form feed

\n	new line
\r	carriage return
\t	horizontal tab
\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 −

Form	Description
${var}	Substitute the value of *var*.
${var:-word}	If *var* is null or unset, *word* is substituted for **var**. The value of *var* does not change.
${var:=word}	If *var* is null or unset, *var* is set to the value of **word**.
${var:?message}	If *var* is null or unset, *message* is printed to standard error. This checks that variables are set correctly.
${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
```

20. Unix — Shell Quoting Mechanisms

In this chapter, we will discuss in detail about the Shell quoting mechanisms. We will start by discussing the metacharacters.

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 –

Quoting	Description
Single quote	All special characters between these quotes lose their special meaning.
Double quote	Most special characters between these quotes lose their special meaning with these exceptions: • $ • ` • \$ • \' • \" • \\
Backslash	Any character immediately following the backslash loses its special meaning.
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]
```