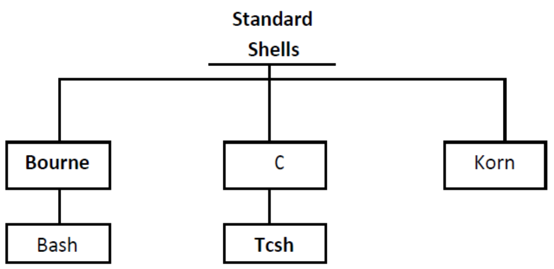
Shell Script:

The shell is the part of the UNIX that is most visible to the user. It receives and interprets the commands entered by the user. In many respects, this makes it the most important component of the UNIX structure.
To do anything in the system, we should give the shell a command. If the command requires a utility, the shell requests that the kernel execute the utility. If the command requires an application program, the shell requests that it be run. The standard shells are of different types as



There are two major parts to a shell. The first is the interpreter. The interpreter reads your commands and works with the kernel to execute them. The second part of the shell is a programming capability that allows you to write a shell (command) script.
A **shell script** is a file that contains shell commands that perform a useful function. It is also known as shell program.
Three additional shells are used in UNIX today. The Bourne shell, developed by Steve Bourne at the AT&T labs, is the oldest. Because it is the oldest and most primitive, it is not used on many systems today. An enhanced version of Bourne shell, called Bash (Bourne again shell), is used in Linux.
The C shell, developed in Berkeley by Bill Joy, received its name from the fact that its commands were supposed to look like C statements. A compatible version of C shell, called **tcsh** is used in Linux.
The Korn shell, developed by David Korn also of the AT&T labs, is the newest and most powerful. Because it was developed at AT&T labs, it is compatible with the Borne shell.

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.

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
```

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**.

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

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
= (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.

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.

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.

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 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.

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.

Instruction (Sequence Control Instruction, Selection Control Instruction, Repetition or Loop Instruction):

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

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

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

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
```

The echo Command

echo is a built-in command in the *bash* and *C* shells that writes its arguments to standard output.

A shell is a program that provides the command line (i.e., the all-text display user interface) on Linux and other Unix-like operating systems. It also executes (i.e., runs) commands that are typed into it and displays the results. bash is the default shell on Linux.

A command is an instruction telling a computer to do something. An argument is input data for a command. Standard output is the display screen by default, but it can be redirected to a file, printer, etc.

The syntax for echo is

```
echo [option(s)] [string(s)]
```

The items in square brackets are optional. A *string* is any finite sequence of *characters* (i.e., letters, numerals, symbols and punctuation marks).

When used without any options or strings, echo returns a blank line on the display screen followed by the command prompt on the subsequent line. This is because pressing the ENTER key is a signal to the system to start a new line, and thus echo repeats this signal.

When one or more strings are provided as arguments, echo by default repeats those stings on the screen. Thus, for example, typing in the following and pressing the ENTER key would cause echo to repeat the phrase *This is a pen.* on the screen:

```
echo This is a pen.
```

It is not necessary to surround the strings with quotes, as it does not affect what is written on the screen. If quotes (either single or double) are used, they are not repeated on the screen.

Fortunately, echo can do more than merely repeat verbatim what follows it. That is, it can also show the value of a particular variable if the name of the variable is preceded directly (i.e., with no intervening spaces) by the dollar character ($), which tells the shell to substitute the value of the variable for its name.

For example, a variable named *x* can be created and its value set to 5 with the following command:

```
x=5
```

The value of x can subsequently be recalled by the following:

```
echo The number is $x.
```

Echo is particularly useful for showing the values of *environmental variables*, which tell the shell how to behave as a user works at the command line or in *scripts* (short programs).

For example, to see the value of HOME, the environmental value that shows the current user's home directory, the following would be used:

```
echo $HOME
```

Read Command:

To get input from the keyboard, you use the **read** command. The **read** command takes input from the keyboard and assigns it to a variable. Here is an example:

```
#!/bin/bash
```

```
echo -n "Enter some text > "
read text
echo "You entered: $text"
```