UNIX SHELL PROGRAMMING

Unix - What is Shells?

The 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.

A shell is an environment in which we can run our commands, programs, and shell scripts. There are different flavors of shells, 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 command prompt, is issued by the shell. While the prompt is displayed, you can type a command.

The 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 **date** command which displays current date and time:

[um\_ramya@ssh ~]$ date

Tue Jul 24 13:40:15 IST 2012

# Shell Types:

In UNIX there are two major types of shells:

1. The Bourne shell. If you are using a Bourne-type shell, the default prompt is the $ character.
2. The C shell. If you are using a C-type shell, the default prompt is the % character.

There are again various subcategories for Bourne Shell which are listed as follows:

* 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 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 to use on several different versions of UNIX.

# Shell Scripts:

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 a pound 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.

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

## Example Script:

To create a test.sh script. All the scripts would have **.sh** extension. Before adding anything else to the 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, 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 : R.Manjusha  # Script follows here:  pwd  ls |

Now you save the above content and make this script executable as follows:

[r\_manjusha@ssh ~]$ chmod +x test1.sh

Now you have your shell script ready to be executed as follows:

[r\_manjusha@ssh ~]$ ./test1.sh

This would produce following result:

/home/staff/ase/it/r\_manjusha

hi.txt newfile palindrome.sh test1.sh test.sh unixstuff

**Note:** To execute your any program available in current directory you would execute using**./program\_name**

# Extended Shell Scripts:

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. No matter how complicated a script gets, however, it is still just a list of commands executed sequentially.

Following script use 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  echo "What is your name?"  read PERSON  echo "Hello, $PERSON" |

Here is sample run of the script:

|  |
| --- |
| [r\_manjusha@ssh ~]$./test.sh  What is your name?  Patrick  Hello, Patrick  [r\_manjusha@ssh ~]$ |

# Unix - Using Shell 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 would 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="Shalini" |

Above example defines the variable NAME and assigns it the value "Shalini". Variables of this type are called scalar variables. A scalar variable can hold only one value at a time.

The shell enables you to store any value you want in a variable. For example:

|  |
| --- |
| VAR1="Shalini"  VAR2=100 |

Accessing Values:

To access the value stored in a variable, prefix its name with the dollar sign ( $):

For example, following script would access the value of defined variable NAME and would print it on STDOUT:

|  |
| --- |
| #!/bin/sh  NAME="Shalini"  echo $NAME |

This would produce following value:

|  |
| --- |
| Shalini |

Read-only Variables:

The shell provides a way to mark variables as read-only by using the readonly command. After a variable is marked read-only, its value cannot be changed.

For example, following script would give error while trying to change the value of NAME:

|  |
| --- |
| #!/bin/sh  NAME="Shalini"  readonly NAME  NAME="Preethi" |

This would produce following result:

|  |
| --- |
| /bin/sh: NAME: This variable is read only. |

Unsetting Variables:

Unsetting or deleting a variable tells the shell to remove the variable from the list of variables that it tracks. Once you unset a variable, you would not be able to access stored value in the variable.

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

|  |
| --- |
| unset variable\_name |

Above command would unset the value of a defined variable. Here is a simple example:

|  |
| --- |
| #!/bin/sh  NAME="Shalini"  unset NAME  echo $NAME |

Above example would 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 command prompt.
* **Environment Variables:** An environment variable is a variable that 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.

# Unix - Using Shell Arrays

A shell variable is capable enough to hold a single value. This type of variables are called scalar variables.

Shell supports a different type of variable called an array variable that 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.

Say that 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="Ahan"  NAME02="Tuhinaa"  NAME03="Diya"  NAME04="Aradhana"  NAME05="Adhiraa" |

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

|  |
| --- |
| 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]="Ahan"  NAME[1]="Tuhinaa"  NAME[2]="Diya"  NAME[3]="Aradhana"  NAME[4]="Adhiraa" |

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

|  |
| --- |
| set -A array\_name value1 value2 ... valuen |

If you are using **bash** shell then, 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 the simplest example:

|  |
| --- |
| #!/bin/sh  NAME[0]="Ahan"  NAME[1]="Tuhinaa"  NAME[2]="Diya"  NAME[3]="Aradhana"  NAME[4]="Adhiraa"  echo "First Index: ${NAME[0]}"  echo "Second Index: ${NAME[1]}" |

This would produce following result:

|  |
| --- |
| [r\_manjusha@ssh ~]$./test.sh  First Index: Ahan  Second Index: Tuhinaa |

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 is the simplest example:

|  |
| --- |
| #!/bin/sh  NAME[0]="Ahan"  NAME[1]="Tuhinaa"  NAME[2]="Diya"  NAME[3]="Aradhana"  NAME[4]="Adhiraa"  echo "First Method: ${NAME[\*]}"  echo "Second Method: ${NAME[@]}" |

This would produce following result:

|  |
| --- |
| [r\_manjusha@ssh ~]$./test.sh  First Method: AhanTuhinaa Diya AradhanaAdhiraa  Second Method: AhanTuhinaa Diya AradhanaAdhiraa |

# Unix - Shell Basic Operators

These are the following basic shell operators:

* Arithmetic Operators.
* Relational Operators.
* Boolean Operators.
* String Operators.
* File Test Operators.

The Bourne shell didn't originally have any mechanism to perform simple arithmetic but it uses external programs, either **awk** or the must simpler program **expr**.

Here is simple example to add two numbers:

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

This would produce following result:

|  |
| --- |
| Total value : 4 |

**NOTE:**

* There must be spaces between operators and expressions for example 2+2 is not correct, where as it should be written as 2 + 2.
* Complete expression should be enclosed between **``**, called inverted commas.

Arithmetic Operators:

Following arithmeticoperators 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 % $a` will give 0 |
| = | Assignment - Assign right operand in left operand | a=$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 note here that all the conditional expressions would be put inside square braces with one spaces around them, for example [ $a == $b ] is correct where as [$a==$b] is incorrect.

All the arithmetical calculations are done using long integers.

Relational Operators:

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

For example, following operators would 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 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 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 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 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 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 condition becomes true. | [ $a -le $b ] is true. |

It is very important to note here that all the conditional expressions would be put inside square braces with one spaces around them, for example [ $a <= $b ] is correct where as [$a <= $b] is incorrect.

Boolean Operators:

Following boolean operators are supported by 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 condition would be true. | [ $a -lt 20 -o $b -gt 100 ] is true. |
| -a | This is logical AND. If both the operands are true then condition would be true otherwise it would be false. | [ $a -lt 20 -a $b -gt 100 ] is false. |

String Operators:

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 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 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. | [ -z $a ] is not false. |
| str | Check if str is not the empty string. If it is empty then it returns false. | [ $a ] is not false. |

File Test Operators:

There are following operators to test various properties associated with a Unix file.

Assume a variable **file** holds an existing file name "test" whose size is 100 bytes and has read, write and execute permission on:

[Show Examples](http://www.tutorialspoint.com/unix/unix-file-operators.htm)

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description** | **Example** |
| -b file | Checks if file is a block special file if yes then condition becomes true. | [ -b $file ] is false. |
| -c file | Checks if file is a character special file if yes then condition becomes true. | [ -b $file ] is false. |
| -d file | Check if file is a directory if yes then condition becomes true. | [ -d $file ] is not true. |
| -f file | Check if file is an ordinary file as opposed to a directory or special file if yes then condition becomes true. | [ -f $file ] is true. |
| -g file | Checks if file has its set group ID (SGID) bit set if yes then condition becomes true. | [ -g $file ] is false. |
| -k file | Checks if file has its sticky bit set if yes then condition becomes true. | [ -k $file ] is false. |
| -p file | Checks if file is a named pipe if yes then condition becomes true. | [ -p $file ] is false. |
| -t file | Checks if file descriptor is open and associated with a terminal if yes then condition becomes true. | [ -t $file ] is false. |
| -u file | Checks if file has its set user id (SUID) bit set if yes then condition becomes true. | [ -u $file ] is false. |
| -r file | Checks if file is readable if yes then condition becomes true. | [ -r $file ] is true. |
| -w file | Check if file is writable if yes then condition becomes true. | [ -w $file ] is true. |
| -x file | Check if file is execute if yes then condition becomes true. | [ -x $file ] is true. |
| -s file | Check if file has size greater than 0 if yes then condition becomes true. | [ -s $file ] is true. |
| -e file | Check if file exists. Is true even if file is a directory but exists. | [ -e $file ] is true. |

# Unix - Shell Decision Making

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 right actions.

Unix Shell supports conditional statements which are used to perform different actions based on different conditions.

* The **if...else** statements
* 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:

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 |

Here Shell *expression* is evaluated. If the resulting value is *true*, given *statement(s)* are executed. If *expression* is *false* then no statement would be not executed. Most of the times you will use comparison operators while making decisions.

Give you attention on the spaces between braces and expression. This space is mandatory otherwise you would get syntax error.

If **expression** is a shell command then it would be assumed true if it return 0 after its 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 |

This will produce following result:

|  |
| --- |
| a is not equal to b |

# The if...else...fi statement

The **if...else...fi** statement is the next form of control statement that allows Shell to execute statements in more controlled way and making decision between two choices.

## 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 |

Here Shell *expression* is evaluated. If the resulting value is *true*, given *statement(s)* are executed. If *expression* is *false* then no statement would be not executed.

## Example:

If we take above example then it can be written in better way using *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 |

This will produce following result:

|  |
| --- |
| a is not equal to b |

# 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 |

There is nothing special about this code. It 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 non 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 |

This will produce following result:

|  |
| --- |
| a is less than b |

# 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.

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 support **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 several different statements to execute 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 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 |

This will produce following result:

|  |
| --- |
| New Zealand is famous for kiwi. |

# Shell Substitutions

# What is Substitution?

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

## Example:

Following is the example, while printing value of the variable its substitued by its value. Same time "\n" is substituted by a new line:

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

This would produce following result. Here **-e** option enables interpretation of backslash escapes.

|  |
| --- |
| Value of a is 10 |

Here is the result without -e option:

|  |
| --- |
| Value of a is 10\n |

Here are following escape sequences which can be used in 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 **-E** option to disable interpretation of backslash escapes (default).

You can use **-n** option to disable insertion of 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 command substitution make sure that you are using 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 demonstrate 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" |

This will produce 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.

|  |  |
| --- | --- |
| **Form** | **Description** |
| **${var}** | Substitue the value of *var*. |

# Shell Quoting Mechanisms

# 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 charater while listing files in a directory and an **\*** would match more than one characters. 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 is the example which show how to print a **\*** or a **?**:

|  |
| --- |
| #!/bin/sh  echo Hello; Word |

This would produce following result.

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

Now let us try using a quoted character:

|  |
| --- |
| #!/bin/sh  echo Hello\; Word |

This would produce 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" |

This would produce following result:

|  |
| --- |
| I have $1200 |

There are following four forms of quotings:

|  |  |
| --- | --- |
| **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] |

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]' |

Any characters within single quotes are quoted just as if a backslash is in front of each character. So now this echo command displays properly.

If a single quote appears within a string to be output, you should not put the whole string within single quotes instead you whouldpreceed 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=ZARA  echo '$VAR owes <-$1500.\*\*>; [ as of (`date +%m/%d`) ]' |

This would produce following result:

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

So this is not what you wanted to display. It is obvious that single quotes prevent variable substitution. If you want to substitute variable values and to make invert commas work as expected then you would need to put your commands in double quotes as follows:

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

Now this would produce following result:

|  |
| --- |
| ZARA 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).

Any characters within single quotes are quoted just as if a backslash is in front of each character. So now this echo command displays properly.

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

|  |
| --- |
| echo 'It\'s Shell Programming' |

# The Back Quotes:

Putting any Shell command in between back quotes would execute the command

## Syntax:

Here is the simple syntax to put any Shell **command** in between back quotes:

## Example:

|  |
| --- |
| var=`command` |

## Example:

Following would execute **date** command and produced result would be stored in DATA variable.

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

This would produce following result:

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

# Shell Functions

Functions enable you to break down the overall functionality of a script into smaller, logical subsections, which can then be called upon to perform their individual task when it is needed.

Using functions to perform repetitive tasks is an excellent way to create code reuse. Code reuse 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, which are followed by a list of commands enclosed within braces.

## Example:

Following is the simple example of using function:

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

When you would execute above script it would produce following result:

|  |
| --- |
| [um\_ramya@ssh ~]$./test.sh  Hello World  [um\_ramya@ssh ~]$ |

# Pass Parameters to a Function:

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

Following is an example where we pass two parameters *Zara* and *Ali* 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 Zara Ali |

This would produce following result:

|  |
| --- |
| [um\_ramya@ssh ~]$./test.sh  Hello World Zara Ali  [um\_ramya@ssh ~]$ |

# 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 1:

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

This would produce following result:

|  |
| --- |
| [um\_ramya@ssh ~]$./test.sh  Hello World Zara Ali  Return value is 10  [um\_ramya@ssh ~]$ |

# Nested Functions:

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

Following simple example demonstrates a 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 |

This would produce 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* so that they'll be available whenever you log in and you can use them at 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:

|  |
| --- |
| [um\_ramya@ssh ~]$. test.sh |

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

|  |
| --- |
| [um\_ramya@ssh ~]$ number\_one  This is the first function speaking...  This is now the second function speaking...  [um\_ramya@ssh ~]$ |

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

|  |
| --- |
| [um\_ramya@ssh ~]$unset .f function\_name |

**Looping statements:**

**Looping Statements in Shell Scripting:** There are total 2 looping statements which can be used in bash programming

1. while statement
2. for statement

To alter the flow of loop statements, two commands are used they are,

1. break
2. continue

**While loop: example:**

***Syntax***

while command

do

Statement to be executed

done

#!/bin/sh

INPUT\_STRING=hello

while [ "$INPUT\_STRING" != "bye" ]

do

echo "Please type something in (bye to quit)"

read INPUT\_STRING

echo "You typed: $INPUT\_STRING"

done

**For Loop: Example**

***Syntax***

forvar in word1 word2 ...wordn

do

Statement to be executed

done

#!/bin/sh

fori in 1 2 3 4 5

do

echo "Looping ... number $i"

done

**Continue Example:**

fora in 1 2 3 4 5 6 7 8 9 10

do

    # ifa = 5 then continuethe loop and

    # don't move to line 8

    if[ $a== 5 ]

    then

        continue

    fi

    echo"Iteration no $a"

done

**Break Example:**

a=0

# -lt is less than operator

#Iterate the loop until a less than 10

while[ $a-lt 10 ]

do

    # Printthe values

    echo$a

    # increment the value

    a=`expr $a+ 1`

done