SHELL PROGRAMMING PART 6

Command Substitution

Command substitution is the mechanism by which the shell performs a given set of commands and then substitutes their output in the place of the commands.

Syntax

The command substitution is performed when a command is given as –

```
`command`
```

When performing the command substitution make sure that you use the backquote, not the single quote character.

Example

Command substitution is generally used to assign the output of a command to a variable. Each of the following examples demonstrates the command substitution –

```
#!/bin/sh


DATE=`date`

echo "Date is $DATE"


USERS=`who | wc -l`

echo "Logged in user are $USERS"
```

```
UP=`date ; uptime`

echo "Uptime is $UP"
```

Upon execution, you will receive the following result –

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

Variable Substitution

Variable substitution enables the shell programmer to manipulate the value of a variable based on its state.

Here is the following table for all the possible substitutions –

Sr.No.	Form & Description
1	**${var}** Substitute the value of *var*.
2	**${var:-word}** If *var* is null or unset, *word* is substituted for **var**. The value of *var*does not change.

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

Example

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

```
#!/bin/sh


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

echo "1 - Value of var is ${var}"


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

echo "2 - Value of var is ${var}"
```

```
unset var

echo ${var:+"This is default value"}

echo "3 - Value of var is $var"



var="Prefix"

echo ${var:+"This is default value"}

echo "4 - Value of var is $var"



echo ${var:?"Print this message"}

echo "5 - Value of var is ${var}"
```

Upon execution, you will receive the following result –

```
Variable is not set
1 - Value of var is
Variable is not set
2 - Value of var is Variable is not set

3 - Value of var is
This is default value
4 - Value of var is Prefix
Prefix
5 - Value of var is Prefix
```

Creating Functions

To declare a function, simply use the following syntax −

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

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

Example

Following example shows the use of function −

```
#!/bin/sh


# Define your function here

Hello () {

   echo "Hello World"

}


# Invoke your function

Hello
```

Upon execution, you will receive the following output −

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

```
Hello World
```

Pass Parameters to a Function

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

Following is an example where we pass two parameters *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
```

Upon execution, you will receive the following result –

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

Returning Values from Functions

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

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

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

```
return code
```

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

Example

Following function returns a value 10 –

```
#!/bin/sh


# Define your function here
Hello () {
   echo "Hello World $1 $2"
   return 10
}
```

```
# Invoke your function

Hello Zara Ali


# Capture value returnd by last command

ret=$?


echo "Return value is $ret"
```

Upon execution, you will receive the following result −

```
$./test.sh
Hello World Zara Ali
Return value is 10
```

Nested Functions

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

Following example demonstrates nesting of two functions −

```
#!/bin/sh


# Calling one function from another
```

```
number_one () {

   echo "This is the first function speaking..."

   number_two

}


number_two () {

   echo "This is now the second function speaking..."

}


# Calling function one.

number_one
```

Upon execution, you will receive the following result –

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

Function Call from Prompt

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

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

```
$. test.sh
```

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

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

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

```
$ unset -f function_name
```