LINUX SHELL SCRIPTING - 3

Using Shell Arrays

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

If you are using **bash** shell the 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/bash

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]}"
```

This would produce 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 is the simplest example:

```
#!/bin/bash

NAME[0]="Zara"
NAME[1]="Qadir"
NAME[2]="Mahnaz"
NAME[3]="Ayan"
NAME[4]="Daisy"
echo "First Method: ${NAME[*]}"
echo "Second Method: ${NAME[@]}"
```

This would produce following result:

```
$./test.sh
First Method: Zara Qadir Mahnaz Ayan Daisy
Second Method: Zara Qadir Mahnaz Ayan Daisy
```

Linux - 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 backtick.

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.

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.
- The complete expression should be enclosed between ` ` called the backtick.
- You should use **** on the ***** symbol for multiplication.

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.

Example

Here is an example which uses all the relational operators

```
#!/bin/sh

a=10
b=20

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

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

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

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

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

if [ $a -le $b ]
then
   echo "a is less or  equal to b"
else
   echo "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
```

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.

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 −

String Operators

The following string operators are supported by Bourne Shell.

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

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.

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 consi

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.

Operator	Description	Example
-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.
-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.

Example

The following example uses all the **file test** operators. Assume a variable file holds an existing file name **"/tmp/test.txt"** the size of which is 100 bytes and has **read**, **write** and **execute** permission.

```
#!/bin/sh

file="/tmp/test.txt"

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 not a 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 not zero"
else
   echo "File size is 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 does not have write permission
File does not have execute permission
This is sepcial file
This is not a directory
File size is not zero
File does not exist
```