UNIT 5 Tutorial

1. Bash shell arrays

In Bash an array is an indexed list of strings to conveniently store multiple strings without needing a delimiter which is often error prone since word splitting breaks a string wherever there is whitespace. The array feature provides a safer way to represent multiple string elements where an array simply maps integers to strings and such a string element can contain any character, including a whitespace. The use of arrays is strongly encouraged in Bash. Associative arrays are also supported in Bash.

Array element numbering is zero-based, that means the first element is indexed with the number 0. Here is an example to show how an explicit declaration of an array is done using the **declare** command:

```
declare  -a  my_array
```

However, array variables are more often created using compound assignments:

```
my_array=(element1 element2 …. elementN)
```

We can see how to dereference the elements in an array with the following example:

```
$ my_array=( black brown red )

$ echo ${my_array[@]}
black brown red
```

Notice the syntax used to expand the array is using **${my_array[@]},** telling Bash to replace this syntax with each element in the array.

It is usually safer to use quotes when assigning string variables. Here is another example:

```
$ my_array=( “black” “ brown” “ red” “sea blue”  “ sky blue” )
```

We can specify explicit indexes as follows:

```
$ my_array=([0]=“black” [1]=“brown” [2]=“red” [6]=“sea blue”)
echo ${my_array[@]}
black brown red sea blue
```

In the above example there is a gap in the indices sequence between 2 and 6 so this is a sparse array.
The following example illustrates the use of a **for** loop to iterate over array elements.

```
#! /bin/bash
my_array=("black" "brown" "red" "sea blue")
for colour in "${my_array[@]}"; do
   echo  "$colour"
done
exit 0
```

The output of the above example will be, assume the script program is named **array_example**:
$ ./array_example
black
brown
red
sea blue

Note in the example the use of quotes in **"${my_array[@]}"** to achieve the intended expansion. Without the quotes Bash will word split the array elements where there are white spaces.

The example above expanded the array using a **for** loop statement. The array can be expanded easily to access its elements as arguments. Here is an example using the **printf** command.

Assume our sample array has been created as follows:

$ my_array=("black" "brown" "red" "sea blue")

The **printf** command will display an output as follows:

$ printf "%s\n" "${my_array[@]}"
black
brown
red
sea blue

Using * to expand array elements
Another form of expanding array elements, in place of ${**my_array[@]},** is to use **${my_array[*]}**. However this form converts an array into a simple single string, which is useful for display purposes but loses the proper separation of elements. To illustrate this, the **printf** command is now run again but this time using the **${my_array[*]}** syntax, as follows:

$ printf "%s\n" "${my_array[*]}"
black brown red sea blue

You will see in this output that a simple single string (one line) is displayed.

Number of elements in an array

It is often useful to know the number of element in an array variable. The **${#my_array[@]}** syntax can be used as follows:

Assume again our sample array has been created as follows:

$ my_array=("black" "brown" "red" "sea blue")

We can count the number of array elements as:

$ echo The number of elements is: ${#my_array[@]}
The number of elements is: 4

The length of a single element in an array can be found as follows:

$ echo The length of element 3 is ${#my_array[3]} characters
The length of element 3 is 8 characters.

Copying an array

To make a copy of an array you can expand the array elements and store them in the new array as follows:

$ your_array=("${my_array[@]}")

Associative arrays

Bash supports associative arrays, which are arrays that go beyond the classical array that maps a number to a string. The associative array can map one string to another, so as to realise key/value pairs. An associative array is declared using the 'declare –A' command. We will introduce the concept with the aid of a small example which will use an array to list teachers that are assigned to specific subjects.

First we will declare an associative array called teachers:

$ declare -A teachers

Next we will assign some element to the associative array:

$ teachers=(["Science"]="I. Newton" ["Maths"]="A. Einstein" ["English"]="W. Shakespeare")

Next we will look at a single element:

$ echo ${teachers[Maths]}
A. Einstein

Now we will look at all the elements:

$ echo ${teachers[@]}
A. Einstein W. Shakespeare I. Newton

We will now write a small script to use a **for** loop to iterate over the 'teachers' associative array, as follows:

```
#! /bin/bash
declare -A teachers
teachers=( ["Science"]="I. Newton" ["Maths"]="A. Einstein" ["English"]="W. Shakespeare" )

for subj in "${!teachers[@]}"; do
  echo "For the subject $subj, the teacher is: ${teachers[$subj]}."
done
```

exit 0

When we run the above the output will be as follows:

$ ./assoc_array_example
For the subject Maths, the teacher is: A. Einstein.
For the subject English, the teacher is: W. Shakespeare.
For the subject Science, the teacher is: I. Newton.

There are some points to note from the above example.

First to clarify what is a key and what is a value. In the example array element below "English" is the key and "W.Shakespeare" is the value:

["English"]="W. Shakespeare"

The keys are accessed using an exclamation mark: **${!array[@]}**, the values are accessed using **${array[@]}**.

We can iterate over the key/value pairs like this:

```
for i in "${!array[@]}"
do
  echo "key  : $i"
  echo "value: ${array[$i]}"
done
```

So in the **for** statement the **${!teachers[@]}** the keys are accessed using an exclamation mark, and the values are accessed using **${array[@]}**.

Note, the order of the keys returned from the associative array using the **${!teachers[@]}** syntax is not predictable.

2. Shell arithmetic

The Bash shell's built-in arithmetic features are based on integer mathematics only. Floating point arithmetic can be achieved using external programs as will be briefly introduced.

2.1 Simple Arithmetic

We will look at the well-known UNIX shell's **let** command for doing simple arithmetic operators. However, be aware that the **let** command is now quite antiquated and later we will see how the shell provides a more convenient way of assigning arithmetic values.

Consider the following examples:

Example 1:
```
x=10
y=$x+5
echo $y
```

Here, the **echo** command outputs the value of y to the screen as follows:

$ echo $y
10+5

Clearly, no arithmetic operation has been performed.

However, the **let** command allows us to perform simple arithmetic operations. The **let** command will assign an actual arithmetic value to a variable, as follows:

Example 2:
```
x=10
let y=$x+5
echo $y
```

This time the **echo $y** command outputs the expected answer, **15**, as follows:

$ echo $y
15

The **let** command above could be written a bit neater as follows, allowing spaces to be used within the parentheses:

let y=($x + 5)

The **let** command works on simple integer arithmetic only and includes the following operators:

-x	negate x
x*y	multiply
x/y	divide
x%y	remainder
x+y	addition
a-y	subtraction

2.2 Arithmetic using the ((..)) syntax

A more intuitive syntax, which is recommended to be used instead of the **let** command, is to wrap the arithmetic statement in double parentheses. This form of arithmetic expression is a more familiar style for programmers and is more forgiving about the use of spaces. Variables inside (( )) parentheses do not require the $ symbol (as string literals are not supported).

The evaluation of the arithmetic expressions is based on fixed-width integers without overflow checking, however, the divide-by-zero is error is recognised.

The operators (not a complete list) are similar to that of the C programming language as listed below in order of precedence:

VAR++ and VAR--	post-increment and post-decrement
++VAR and --VAR	pre-increment and pre-decrement
- and +	unary minus and plus

! and ~	logical and bitwise negation
**	Exponentiation
*, / and %	multiplication, division, remainder
+ and -	addition, subtraction
<< and >>	left and right bitwise shifts
<=, >=, < and >	comparison operators
== and !=	equality and inequality
&	bitwise AND
^	bitwise exclusive OR
|	bitwise OR
&&	logical AND
||	logical OR
expr ? expr : expr	conditional evaluation
=, *=, /=, %=, +=, -=, <<=, >>=, &=, ^= and |=	Assignments
,	separator between expressions

Here is an example using the arithmetic ((..)) syntax

x=77
((y = 3 * x)) # $x can be represented as simply x
echo $y

Of course, y will have a value of 231following the above.

Note, spaces can be omitted if preferred as follows:

((y=3*$x)) # spaces can be left out if preferred

The following three operations are equivalent:

((x=$x+9))

((x = x + 9))

((x += 9))

Here is an example command to make a random number:

((x = RANDOM % 10 + 1)) # a random number from 1 to 10

2.3 Arithmetic Expansion

In arithmetic expansion the arithmetic expression is evaluated and the result is substituted. The format for arithmetic expansion is:

$((arithmetic expression))

The arithmetic expression is evaluated to expand to the result. The result will be a one word digit in Bash.

There is an alternative syntax for arithmetic expansion in Bash as follows:

$[arithmetic expression]

Do not use (**DO NOT USE!**) this style as it is now deprecated, giving preference to the $((...)) form.

At the command prompt, try the following simple examples:

x=2; y=3; echo $((x + y))

var=$((x * 3)); echo $var

var=$((++x)); echo $var

A constant with a leading 0 (zero) is interpreted as an octal number, a leading "0x" or "0X" indicates hexadecimal. Otherwise, numbers take the form [BASE#]N, where BASE represents the arithmetic base and N is the number in that base, as in the following examples:

$ echo $((10#34   ))
34

$ echo $((8#34   ))
28

$ echo $((16#39   ))
57

$ echo $((16#3A))
58

$ echo $((13#34   ))
43

$ echo $((2#10101110))
174

By default the base 10 is assumed.

As seen in some of the examples, variables in arithmetic expansion can be used with or without variable expansion, for example:

```
x=7
echo $((x))       #  Normal use. Good.
echo $(($x))      # Works fine, but not recommended, better to use variables directly.
```

2.4 Floating point arithmetic

Bash does not natively support floating point arithmetic. There are command line tools that can be used to perform floating point calculations. Probably the best-known tool (utility) is called **bc**, which is defined as an arbitrary precision calculator language. The **bc** has four special variables: scale, ibase, obase, and last. The default base for input and output is 10. The variable 'scale' is use to set the precision for results. We will look at some easy examples to show some features of **bc** with simple arithmetic.

Type **bc** on the command line and you will enter the **bc**. Type 'quit' to exit.

```
$ bc
bc 1.07.1
Copyright 1991-1994, 1997, 1998, 2000, 2004, 2006, 2008, 2012-2017 Free Software
Foundation, Inc.
This is free software with ABSOLUTELY NO WARRANTY.
For details type `warranty'.
```

You are now in the bc utility. Type the following example where we set the scale value to 3 and enter a small equation. We see the answer is 1.618 (which happens to be the The Golden Ratio: Phi, 1.618) to three decimal places as defined by the scale value:

```
scale = 3
( 1 + sqrt(5) ) / 2
1.618
```

Next we see how to pipe our expression directly into the **bc** utility:

```
$ echo  "scale=3;  (1 + sqrt(5))/2"  |  bc
1.618
```

For the above we could have used a heredoc or a herestring (heredocs will be described later on) as follows:

```
$ bc  <<<  "scale=3; (1 + sqrt(5))/2"
1.618
```

UsingAWK for arithmetic

Floating point arithmetic can also be achieved using **awk**. Here are small examples:

```
$ awk 'BEGIN { print 1000/9 }'
111.111

$ awk 'BEGIN { print 2 ^ 8 }'
256
```

The **printf** statement can be used to format the output including the precision of results:

```
$ awk "BEGIN { printf \"%.2f\n\",  (1 + sqrt(5))/2 }"
1.62
```

Here we calculate the value of phi and assign it to a variable:

```
$ phi= $( awk "BEGIN { printf \"%.2f\n\", (1 + sqrt(5))/2 }" )
$ echo phi
1.62
```

As we can now imagine, the full power of **awk's** arithmetic operations is available to the shell programmer.