BASH Programming

Full Guide to Bash Arrays

2 years ago • by Sam U

In itself, Linux is merely an operating system kernel; the kernel is a crucial component of the operating system, which facilitates I/O devices communicating with the software used by the user. Besides, it manages memory, CPU, and protects hardware and software from malfunctioning. The interface or software part that the user uses to interact with the hardware is called Command Line Interface (CLI) or a Shell.

Linux shell is a program with an interface that takes commands from the user, interprets them, and sends them to the kernel to perform a specified operation. Command Line Interface (CLI) is the minimalist way to interact with the hardware of the system. There are tons of commands for performing various functionalities, such as making a directory, moving a directory, creating a file, deleting a file, etc.

Shell is a basic command-line interpreter. It yields an interface between the user and the kernel. In Linux, there are many types of shells; a list of commonly used shells are mentioned below:

- Bourne Shell
- Bourne Again Shell [Bash]
- C Shell
- Korn Shell

Different types of shells offer different capabilities. Ken Thompson introduced the first shell for Unix called Thompson Shell. Bourne shell was one of the widely adopted shells developed by Stephen Bourne in 1977 at Bell Laboratories. Bourne Shell has an advanced version called Bourne Again Shell. Bourne Again Shell is also called Bash. Bash was developed by Brian Fox that contained all the features of the Bourne shell but was it much more efficient.

Bash is the default shell from many Linux distributions, and the key features that distinguish **Bash** from **share** are mentioned below:

- The powerful command editing feature
- Unlimited size of event history
- Introduction of aliases
- Unlimited size of arrays

Bash shell has many advanced features, including powerful editing and modification features, making it incredibly user-friendly.

The commands are the fundamental part of Bash; commands tell the shell what operation to perform. In general, the shell takes one command at a time, runs it, and then displays the output, which is also called standard output in the shell. While executing a command, you cannot interact with the shell; the shell completes the operation before making itself available for the next command. However, the execution of any command can be interrupted. The time of execution of command ultimately depends upon the type of function. For instance, if you are downloading a package, it might take longer than listing the current working directory path.

Though the shell is designed to execute one command at a time, if you want to execute multiple commands to perform a specific task, Bash has a solution called Bash scripting.

1 Bash Scripting:

A script is a set of commands that tells the computer what it should do; a Bash script is also a set of commands that tells what Bash should perform. A Shell script is a text file that contains a sequence of commands to perform a particular task. Bash used Bash programming language, which like any other programming language, provides all the tools to perform logical operations such as assigning variables, conditional statements, loop structures, and arrays.

As mentioned above, Bash scripting is like any other programming language. To create a Bash program, you do not need a powerful Integrated Development Environment (IDE) because it can be made on any simple text editor, whether it is **nano**, **vim**, or text editor that comes with the desktop environment.

To create a Bash script, open the text editor and reference the **“/bin/bash”** path using **“#!”** called **hash-bang** or **shebang**. The **“/bin/bash”** is the path of the Bash interpreter. The formatting in Bash scripting is very crucial; even a space can cause errors. And shebang has to be on the top of the script. Type the script and save the file with the **“.sh”** extension. A basic **“hello world”** Bash script is shown below:

```
#! /bin/bash
echo “Hello Linux”
```

```
#! /bin/bash
echo "Hello Linux"
```

```
sam@sam:~$ bash hello.sh
Hello Linux
```

To run the script in the CLI, type **“bash”** and specify the path of the script.

Assigning variables in Bash scripting is simple. It does not need any data type; any character, word, or string can be used as a variable:

```
variable_Name = [Value]

For instance:

#! /bin/bash

var=”Hello Linux”

echo $var
```

```
#! /bin/bash
var="Hello Linux"
echo $var
```

```
sam@sam:~$ bash hello.sh
Hello Linux
```

The **“Hello Linux”** string is assigned to a variable called **“var”** in the above script. As a proper programming language, Bash also supports conditional structures such as ***if-then***, ***nested-if***, and loop structures such as ***for-in*** and ***while-do***.

A single variable can hold one value that can be manipulated in the code. If you want to define more than one variable of the same data type simultaneously, arrays are used. Moreover, arrays are also key elements of the Bash programming language. Arrays are a collection of elements that are identified by the index number. Arrays

are essential when it comes to implementing data structure. Instead of typing multiple variables, arrays save time and are easy on memory.

2 What Are Arrays?

The developers use many aspects of the Bash programming language. There is plenty of data available for other programming structures such as loops and conditional statements, but a structure that is not extensively covered is an array. The Bash array is a crucial structure of any programming language. It is implemented in the data structure.

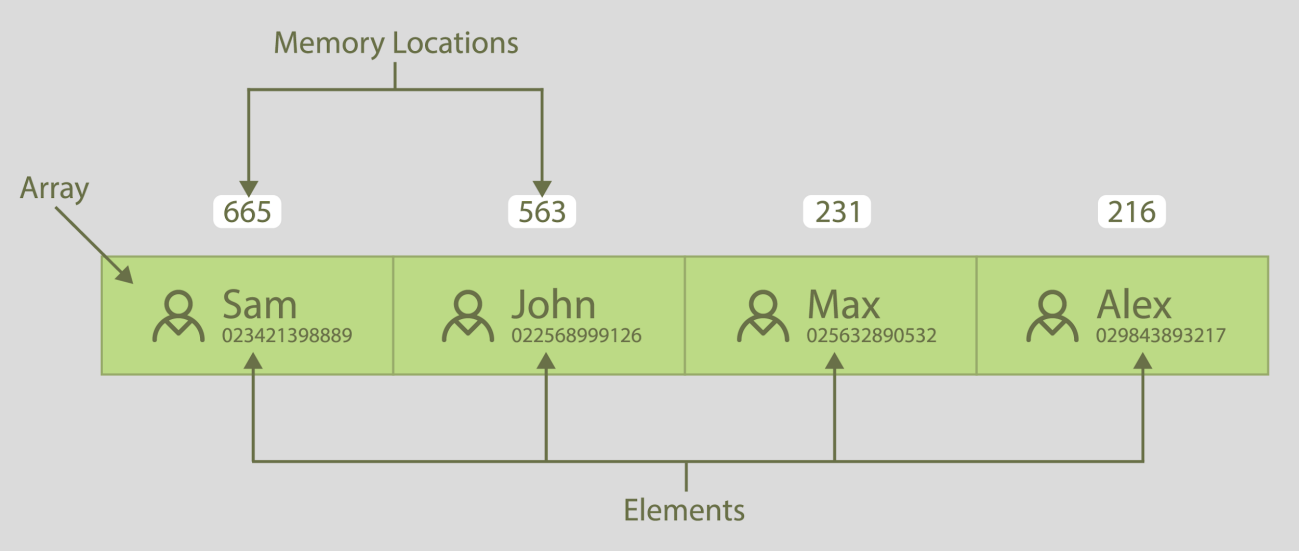
Let’s understand array with a real-life example:

- Post office box
- Pages of a book
- Chessboard
- A carton of eggs

The array is an arrangement of items. Therefore, every item is called an array if arranged in a manner. For example, egg cartons are the perfect example of the arrangement of items in 2D array manner. Eggs in the carton are elements where the box is an array. Similarly, pages in a book are arranged so the book would be called an array where pages would be elements.

Likewise, the contact numbers in our phones, songs, and an arrangement of apps on the home screen are also examples of an array.

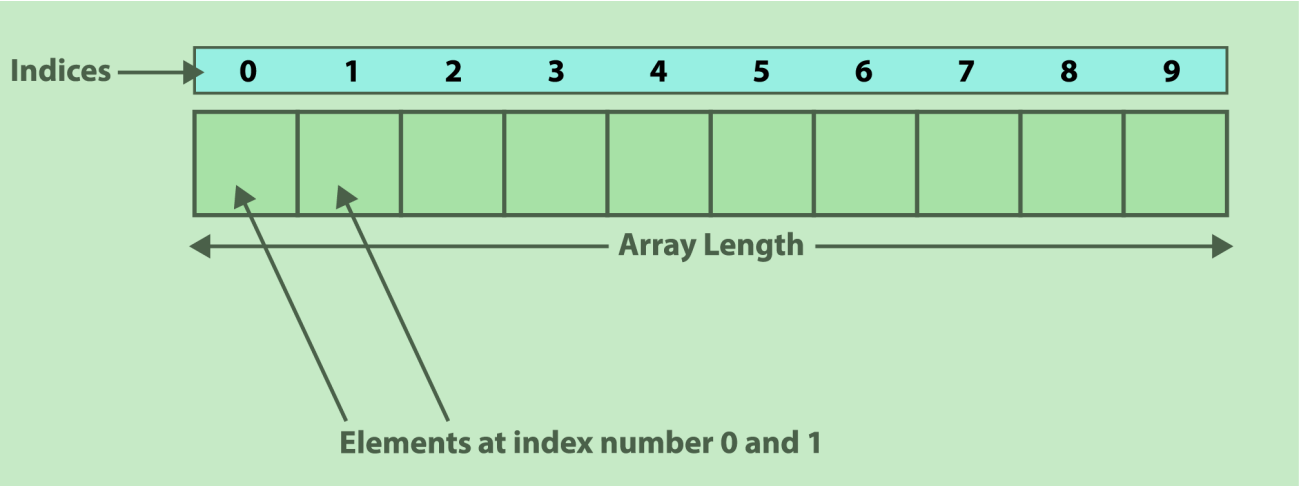
Let’s take an example of contacts in our phone, and the contact book is an example of an array where the contacts are elements of that array. We can manipulate the elements, such as adding a contact number and deleting a contact number.



In the above demonstration, the contact numbers are elements of the array where the numbers above are memory locations.

When you visit an eCommerce site, the items you put in the shopping cart are also an example of an array, since you can add items to the shopping cart and remove them.

A variable that can store multiple variables is called an array. There is no limit when it comes to assigning a number of variables in an array. Array elements are referenced by the index number, which usually starts with zero. The array is mainly used in implementing data structure, which is an approach to organize and manage data effectively. Let’s visualize an array as a container with multiple compartments, as shown in the image below:



There are ten compartments in the above demonstration, so the length of the array would be 10. The first compartment number would be 0 and the last would be 9. The compartments can also be termed as the elements of the array.

Instead of defining multiple variables one by one, arrays help them define them at once; that is an efficient way of assigning variables in programming.

3 Applications of Arrays:

Arrays are such a powerful utility that they can be used in many scientific calculations. Arrays in any programming language are much more functional than other structures. Some notable implementation of arrays are mentioned below:

- Arrays are used to manage multiple variables with the same name.
- Arrays can be used in vectors, where vectors are typically one-dimensional arrays that are extensively used in machine learning.
- Arrays are also used in implementing stacks, and stacks behave like a real pile of physical objects.
- Arrays are also implemented in queues, deques, and hash tables.

- Matrices, which are a rectangular array of elements, are also implemented using arrays.
- Graphs in many programs are drawn using lists which is also any implementation of array.
- Many algorithms, such as CPU scheduling algorithms and sorting algorithms, are implemented using the array.
- Arrays are also used in in-program dynamic memory allocation.
- Arrays are also used in speech processing.
- Noise removing filters are also using arrays.

The above implementations of arrays clearly show the potential of the arrays data type.

4 Syntax of Arrays in Bash:

Bash comes with the support of both indexed array (one-dimensional array) and associative arrays, which will be discussed in the later section. A typical syntax of assigning array in Bash is mentioned below:

```
name_of_array[subscript]=value
```

Since arrays are collections of objects, the object number in the array is called index number or subscript. Subscripts indicate the position of the object in the array. For instance, to assign or modify the value of **x**th object in the array, the syntax would be:

```
name_of_array[x]=value
```

The **“declare”** keyword can also be used to declare an array:

```
declare -a name_of_array
```

To declare an associative array:

```
declare -A name_of_array
```

The syntax of compound assignment of an array is:

```
name_of_array=(value1 value2 …)
```

Any of the previously mentioned methods can be utilized to state arrays in Bash scripting.

5 Assigning Arrays in Bash:

Arrays in Bash scripting can be assigned in various ways. The simplest way to assign an array in Bash scripting is assigning a set of values with space in round brackets to a variable as demonstrated below:

```
my_array=(1 2 3 4)
```

The Bash arrays can have different types of elements. To assign array with string elements:

```
my_array=(jan feb mar apr)
```

To explicitly assigning an array with indices:

```
my_array=([0]='jan' [1]='feb' [2]='mar' [3]='apr')
```

To assign the array with index, type the name of the array, mention the index in the square brackets, **"[index_number]"** and assign a value to it:

```
my_array[0]='jan'
my_array[1]='feb'
```

The array can also be declared with the **"declare"** keyword. The options **"-a"** and **"-A"** is used to declare indexed and associative arrays, respectively:

```
declare -a my_array
my_array[0]='jan'
my_array[1]='feb'
```

String values are used as the index in associative arrays:

```
declare -A my_array
my_array[first]='jan'
my_array[second]='feb'
```

Or:

```
my_array=([first]='jan' [second]='feb' [third]='mar'
[fourth]='apr')
```

The array can also be created from the output of other commands.

For example, the **"seq"** command is used to create a list of numbers:

```
my_array=('seq 1 6')
```

5.1 Assigning Arrays Through Loop:

Array can also be assigned through loops, for example:

```
#! /bin/bash
while
read
do
my_array[$n]=$REPLY
let n++
done < <(seq 1 6)
echo "Array elements are:" ${my_array[@]}
```

```
#! /bin/bash
while
        read
do
        my_array[$n]=$REPLY
let n++
done < <(seq 1 6)
echo "Array elements are: "${my_array[@]}
```

```
sam@sam:~$ bash loop_as.sh
Array elements are: 1 2 3 4 5 6
sam@sam:~$
```

The **“$REPLY”** is the special variable and equals the current input.

5.2 Assigning Arrays From Strings:

An entire string can also be assigned as an array. For example:

```
my_array_string=”Hello this is Linux”
my_array=(${my_array_string// / })
```

In the above script, the delimiter is a “**space**”. A delimiter is a character that individualizes the text string, such as slashes, commas, colons, pipes, and even spaces. In the next example, the delimiter is dash:

```
my_array_string=”Hello-this-is-Linux”
my_array=(${my_array_string//-/ })
```

Let’s implement it in Bash scripting:

```
#! /bin/bash
my_array_string="Hello this is Linux"
my_array=(${my_array_string// / })
echo ${my_array[3]}
#-------------------------------------
my_array_string2="Hello-this-is-Linux"
my_array=(${my_array_string2//-/ })
```

```
echo ${my_array[@]}
```

```
#! /bin/bash
my_array_string="Hello this is Linux"
my_array=(${my_array_string// / })
echo ${my_array[3]}
#-----------------------------------------------
my_array_string2="Hello-this-is-Linux"
my_array=(${my_array_string2//-/ })
echo ${my_array[@]}
```

```
sam@sam:~$ bash array_strings.sh
Linux
Hello this is Linux
sam@sam:~$
```

6 Types of Array in Bash:

There are many ways and approaches to use an array. In Bash, there are two types of primary arrays:

- Indexed arrays
- Associative arrays

6.1 Indexed Arrays:

Indexed arrays are the primary form of an array that stores elements referenced through an index number starting from 0. An example of an indexed array in Bash scripting is mentioned below:

```
my_array=(a b c d)
```

Or arrays can also be declared using the “**declare**” keyword**:**

```
my_array[0] = “First Item”
my_array[1] = “Second Item”
```

In the above example, **“array”** is a variable **“a, b, c, and d”** are the elements of the array. The array’s length would be 4, and the index number of the **“a”** element would be on the zeroth index and **“d”** on the third index.

6.2 Associative Arrays:

Associative arrays are the arrays that use string as index. In other words, the array index in associative arrays is in named form. Associative arrays are declared in Bash using the **“declare”** keyword.

```
declare -A my_array
my_array[one] = “First Item”
my_array[two] = “Second Item”
```

Associative arrays are not part of Bash before they are included in version 4. To identify which version are you using, use the command given below:

```
$bash --version
```

```
sam@sam:~$ bash --version
GNU bash, version 5.0.17(1)-release (x86_64-pc-linux-gnu)
Copyright (C) 2019 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>

This is free software; you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
```

If the version is four or above, then you can use associative arrays. To declare associative array **“-A”** option is used explicitly:

```
declare -A my_array
```

Elements can also be initialized one by one:

```
my_array[month1]=”jan”
my_array[month2]=”feb”
```

Any string or set of characters are used to declare an associative array:

```
my_array["this is a string"]="Hello Linux"
```

It is important to note that the string in the array indexes, as mentioned above, is containing space. Another way of initialization of associative arrays are given below:

```
my_array=([month1]=jan [month2]=feb [month3]=mar)
```

Currently, Bash does not support multidimensional arrays. However, different methods can emulate multidimensional arrays, which can be found in the examples section.

7 Accessing an Array in Bash:

Like all other programming languages, arrays in Bash are also accessed through index numbers. Let's understand it through an example:

```
my_array=(jan feb mar apr)
echo ${my_array[1]}
```

The **"echo"** is a Bash command that prints the standard output in the command-line interface (CLI). In the above example, the **"echo"** command is printing the item on the first index of the array **"my_array"**. The **"feb"** will be printed on the standard output since the index number of **"feb"** is **1**.

7.1 Displaying All Elements of an Array:

To display all the elements of the array quoted separately, follow:

```
echo ${my_array[@]}
```

To display all the elements as a single quote string, use:

```
echo ${my_array[*]}
```

7.2 Displaying Specific Element of an Array:

To display any element of the array, use:

```
echo ${my_array[x]}
```

Replace the **“x”** with the index number of the element you want to display. For example, to print the third element of the array, use:

```
echo ${my_array[2]}
```

Print last element of an array through the subscript expansion method:

```
echo ${my_array[@]: -1}
```

To print the last element via subscript syntax, use:

```
echo ${my_array[-1]}
```

To print a range of elements, use the syntax mentioned below:

```
echo ${my_array[@]:x:y}
```

Where **“x”** is the first index number, and the **“y”** would be the last index number. For instance, to display elements from index **“0”** to **“2”**, use:

```
echo ${my_array[@]:1:3}
```

The above command will print three elements from index 0 to 2. All the operations for accessing arrays are demonstrated in the following image:

```
#! /bin/bash

my_array=(jan feb mar apr)

echo "All the elements of the array:"${my_array[@]}

echo "The second element of the array:"${my_array[1]} #index
starts from 0

echo "Last element of the array through substring
expansion:"${my_array[@]: -1}

echo "Last element of the array through
subscript:"${my_array[-1]}
```

```
echo "Elements from index 1 to 3:"${my_array[@]:1:3}
```

```
#! /bin/bash
my_array=(jan feb mar apr)
echo "All the elements of the array:"${my_array[@]}
echo "The second element of the array:"${my_array[1]} #index starts from 0
echo "Last element of the array through substring expansion:"${my_array[@]: -1}
echo "Last element of the array through subscript:"${my_array[-1]}
echo "Elements from index 1 to 3:"${my_array[@]:1:3}
```

```
sam@sam:~$ bash accessing_arrays.sh
All the elements of the array:jan feb mar apr
The second element of the array:feb
Last element of the array through substring expansion:apr
Last element of the array through subscript:apr
Elements from index 1 to 3:feb mar apr
```

7.3 Accessing the Initialized Indexes of Array:

The index of an array is the key element while programming. To get the index number, use:

```
#! /bin/bash
my_array[3]=”jan”
my_array[5]=”feb”
my_array[9]=”mar”
my_array[12]=”mar”
echo “The list of indexes:”${!my_array[@]}
```

```
#! /bin/bash
my_array[3]="jan"
my_array[5]="feb"
my_array[9]="mar"
my_array[12]="apr"
echo "The list of indexes:"${!my_array[@]}
```

```
sam@sam:~$ bash indexes.sh
The list of indexes:3 5 9 12
sam@sam:~$
```

8 Modification of Arrays in Bash:

One of the benefits of using arrays is that any array element can easily be accessed and modified. Arrays in Bash have various ways to change; all the methods are mentioned below:

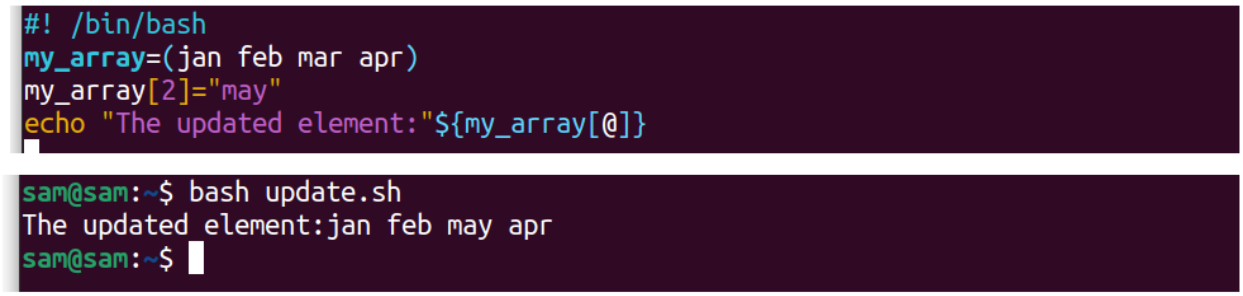
8.1 Updating Elements:

To update a particular element in an array, follow the following syntax:

```
my_array[<index_number>]=value
```

For example:

```
#! /bin/bash
my_array=(jan feb mar apr)
my_array[2]=”may”
echo “The updated element:”${my_array[@]}
```



In the above example, the element on the second index, which is **“mar”** will be replaced by **“may”**.

8.2 Adding Elements:

To add elements to the end of an array:

```
my_array+=(jun Jul)
```

To add an element at the starting of an array:

```
my_array=(‘dec’ ${my_array[@]})
```

Let’s implement it in a Bash script:

```
#! /bin/bash
my_array=(jan feb mar apr)
my_array+=(jun jul)
```

```
echo "Array after adding elements:"${my_array[@]}
my_array=("dec" ${my_array[@]})
echo "Adding element at the end of the array:"${my_array[@]}
```

```
#! /bin/bash
my_array=(jan feb mar apr)
my_array+=(jun jul)
echo "Array after adding elements:"${my_array[@]}
my_array=("dec" ${my_array[@]})
echo "Adding element at the end of the array:"${my_array[@]}
```

```
sam@sam:~$ bash adding.sh
Array after adding elements:jan feb mar apr jun jul
Adding element at the end of the array:dec jan feb mar apr jun jul
sam@sam:~$
```

8.3 Inserting Elements:

To insert an element at a specific index, follow:

```
my_array(jan feb mar apr)
i=2
my_array=("${my_array[@]:0:$i}" "aug" "${my_array[@]:$i}")
```

The above example is inserting the element **“aug”** on the second index of the array**(my_array)** and shifting the following elements to the next indexes. The elements **“mar”** and **“apr”** will be shifted to index 3 and 4 respectively:

```
#! /bin/bash
my_array=(jan feb mar apr)
i=2
my_array=("${my_array[@]:0:$i}" "aug" "${my_array[@]:$i}")
echo "Array after inserting an element:"${my_array[@]}
```

```
#! /bin/bash
my_array=(jan feb mar apr)
i=2
my_array=("${my_array[@]:0:$i}" "aug" "${my_array[@]:$i}")
echo "Array after inserting an element:"${my_array[@]}
```

```
sam@sam:~$ bash inserting.sh
Array after inserting an element:jan feb aug mar apr
sam@sam:~$
```

8.4 Deleting Elements:

In Bash arrays, elements can be deleted using the "**unset**" command. For example, to remove all the elements of an array, use:

```
my_array=(jan feb mar apr)

unset my_array
```

The **"unset"** is the built-in command to delete the declared variables. To unset a specific element in an array, use:

```
#! /bin/bash

my_array=(jan feb mar apr)

unset my_array[2]

echo "Array after deletion of element on third
index:"${my_array[@]}
```

Elements can also be removed using the "**pattern**" command:

```
my_pattern(${my_array[@]/ju*/})
```

The elements that start with **"ju"** will be removed from the array, as shown in the output of the following script:

```
#! /bin/bash
my_array=(jan feb mar apr may jun jul)
my_pattern(${my_array[@]/ju*/})
echo “Array after elements deletion by pattern:”${my_pattern[@]}
```

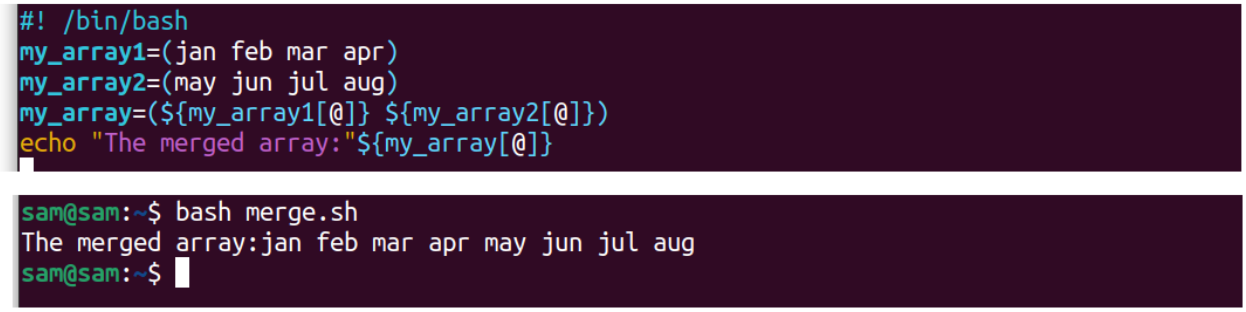
8.5 Merging Arrays:

To merge two arrays use:

```
my_array=(${my_array1[@]} ${my_array2[@]})
```

Let’s merge two arrays in Bash:

```
#! /bin/bash
my_array1=(jan feb mar apr)
my_array2=(may jun jul aug)
my_array=(${my_array1[@]} ${my_array2[@]})
echo "The merged array:"${my_array[@]}
```



8.6 Removing Gaps in Array Elements:

To remove the unintended gaps in the array and re-indexing array use:

```
#! /bin/bash
my_array=(jan feb mar apr)
my_array2=(${my_array[@]})
echo “Array after removing gaps:”${my_array2[@]}
```

```
#! /bin/bash
my_array=(jan feb   mar  apr)
my_array2=(${my_array[@]})
echo "Array after removing gaps:"${my_array2[@]}
```

```
sam@sam:~$ bash gap_removing.sh
Array after removing gaps:jan feb mar apr
sam@sam:~$
```

In the above demonstration, elements of **“my_array”** have gaps in them.

9 Iterating Through the Array with Loops in Bash:

There are various ways to access an array; either you can access them explicitly by typing every element, or you can loop through the elements of the array. Let’s understand it through an example:

```
my_array=(e1 e2 e3 e4 e5 e6)
```

First, use the ***“for…in”*** loop:

```
for i in ${my_array[@]}
do
echo $i
done
```

C is a widely used programming language. Luckily in Bash, you can also use the C language style “for” loop, which is also termed as the classic loop:

```
for((i=0;i<${#my_array[@]};i++));
do
echo ${my_array[i]}
```

```
done
```

Arrays can also be accessed through ***while*** loop:

```
i=0
while[ $i -lt ${#my_array[@]} ];
do
echo my_array[$i]
i=$((i+1))
done
```

Instead of **“-lt”**, the less than sign **“<”** can also be used**,** the above loop can also be written as:

```
i=0
while (( $i < ${#my_array[@]} ));
do
echo my_array[$i]
((i++))
done
```

The ***until*** loop can also be used to iterate through the arrays:

```
i=0
until [ $i -ge ${#my_array[@]} ];
do
echo ${my_array[i]}
i=$((i+1))
done
```

In numerical format:

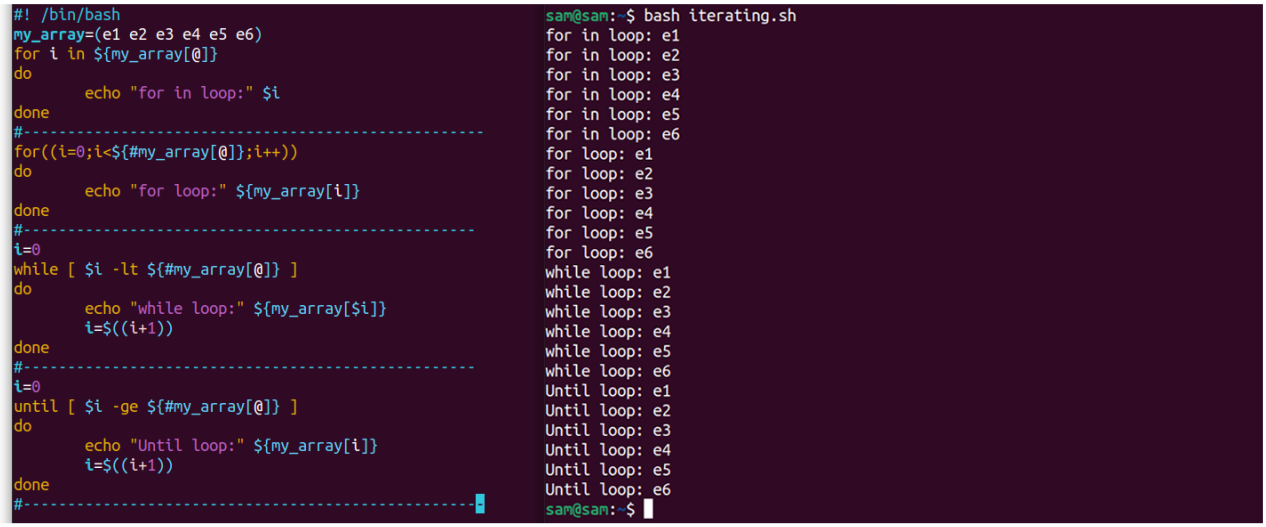
```
i=0
until (( $i < ${#my_array[@]} ));
do
echo ${my_array[i]}
```

```
i=$((i+1))

done
```

The script of implementation of all the loop structures in Bash is mentioned below:

```
#! /bin/bash
my_array=(e1 e2 e3 e4 e5 e6)
for i in ${my_array[@]}
do
        echo "for in loop:" $i
done
#-----------------------------------------------------
for((i=0;i<${#my_array[@]};i++))
do
        echo "for loop:" ${my_array[i]}
done
#-----------------------------------------------------
i=0
while [ $i -lt ${#my_array[@]} ]
do
        echo "while loop:" ${my_array[$i]}
i=$((i+1))
done
#-----------------------------------------------------
i=0
until [ $i -ge ${#my_array[@]} ]
do
        echo "Until loop:" ${my_array[i]}
i=$((i+1))
done
#-----------------------------------------------------
```



10 Length of an Array in Bash:

Knowing the length of the array is very important when working with arrays. To identify the length of an array, use:

```
my_array=(jan feb mar apr)
echo ${#my_array[@]}
```

The character **“#”** is used before the array name.

If the elements of an array are in a string format, then to know the length of a string element in an array, use:

```
my_array=(january february march april)
echo ${#my_array[1]}
```

The above commands will output the length of the second element of the array, which is **8,** since **“february”** is 8 characters long.

```
#! /bin/bash
my_array=(jan feb mar apr)
echo "The length of the array:"${#my_array[@]}
my_array=(january february march april)
echo "The length of the string element:"${#my_array[1]}
```

11 Accessing Associative Arrays in Bash:

Accessing the associative arrays are similar to accessing the indexed arrays. The only difference is that in associative arrays the index is string:

```
declare -A my_array=([month1]=jan [month2]=feb [month3]=mar)
echo ${my_array[month1]}
```

To list the indices of associative arrays, use:

```
echo ${!my_array[@]}
```

To display the values of the array, use:

```
echo ${my_array[@]}
```

Iterate through the associative arrays:

```
my_array=([month1]=jan [month2]=feb [month3]=mar [month5]=apr)
for i in ${!my_array[@]} ;
do
echo my_array[$i]
done
```

To count the elements of the associative arrays, use:

```
my_array=([month1]=jan [month2]=feb [month3]=mar [month5]=apr)
echo {#my_array[@]}
```

All the previously mentioned structures are implemented in the script given below:

```
#! /bin/bash
declare -A my_array=([month1]="jan" [month2]="feb"
[month3]="mar" [month4]="apr")
echo "The first element:" ${my_array[month1]}
echo "Indexes of associative arrays:" ${!my_array[@]}
echo "Number of elements of associative array:" ${#my_array[@]}
echo "Elements of associative arrays:" ${my_array[@]}
#-------------Iterating the associative array-------------------

for i in ${!my_array[@]}
do
        echo ${my_array[$i]}
done
```

```
#! /bin/bash
declare -A my_array=([month1]="jan" [month2]="feb" [month3]="mar" [month4]="apr")
echo "The first element:" ${my_array[month1]}
echo "Indexes of associative arrays:" ${!my_array[@]}
echo "Number of elements of associative array:" ${#my_array[@]}
echo "Elements of associative arrays:" ${my_array[@]}
#-------------Iterating the associative array-------------------

for i in ${!my_array[@]}
do
        echo ${my_array[$i]}
done
```

```
sam@sam:~$ bash a_array.sh
The first element: jan
Indexes of associative arrays: month3 month2 month1 month4
Number of elements of associative array: 4
Elements of associative arrays: mar feb jan apr
mar
feb
jan
apr
```

	Action
***echo* $array[@]**	To print all elements of an array
***echo* $*!*array[@]**	To print the all indexes of an array
***echo* $#array[@]**	To print the length of an array
***echo* $array[*x*]**	To print a specific element of an array by index “x”
array[*x*]=value	To insert/replace an element to a specific index of an array
unset array[x]	To remove an element at a specific index

12 Bash Array Examples:

Bash arrays are the data structure and are very useful for handling the collection of variables. Arrays have various uses in programming. Let's further elaborate on the uses of arrays through examples:

12.1 Example 1: Reading a File Through Array:

To read a file, we need to create a file first. There are various ways to create a file in Linux, for instance, using a redirection operator, cat, or touch command. The created file can be edited in **nano** or **vim** editor.

I have created a file in **"nano"** and saved it with the name of **"my_file.txt"**. To read file, use:

```
$cat my_file
#! /bin/bash
echo "Enter the name of the file"
read file
file=( `cat "$file"`)
for l in ${file[@]}
do
echo $l
done
```

```
#! /bin/bash
echo "Enter the name of the file:"
read file
file=(`cat "$file"`)
for l in ${file[@]}
do
        echo $l
done
```

```
sam@sam:~$ bash reading_file.sh
Enter the name of the file:
textfile.txt
The content of the file: Hello-this-is-Linux
sam@sam:~$
```

12.2 Example 2: Bubble Sorting in Bash:

Sorting is used to manage the data, and it is one of the well-known techniques in programming to make the algorithm functionality more efficient such as search algorithm. Bubble sorting, which is also known as sinking sorting, is one of the easy-to-understand sorting approaches. Bubble sort steps through the provided array list, compare the array elements, swap the element in the temporary variables and repeat the task until the array is in order. An example of bubble sorting in bash is given below:

```
#! /bin/bash
my_array=(2 3 1 5 4)
echo "Unsorted array:" ${my_array[*]}
for ((x=0; x<5; x++))
do

for ((y=0; y<5-i-1; y++))

do
if [ ${my_array[y]} -gt ${my_array[$((y+1))]} ]
then
temp=${my_array[y]}

my_array[$y]=${my_array[$((y+1))]}

my_array[$((y+1))]=$temp
fi

done
done
echo “Sorted array:” ${my_array[*]}
```

```
#! /bin/bash

my_array=(2 3 1 5 4)
echo "Unsorted array:"${my_array[*]}

for ((x=0; x<5; x++))
do
        for ((y=0; y<5-i-1; y++))
        do
                if [ ${my_array[y]} -gt ${my_array[$((y+1))]} ]
                then
                        temp=${my_array[y]}
                        my_array[$y]=${my_array[$((y+1))]}
                        my_array[$((y+1))]=$temp

                fi
        done
done
echo "Sorted array:" ${my_array[*]}
```

```
sam@sam:~$ bash sort.sh
Unsorted array:2 3 1 5 4
Sorted array: 1 2 3 4 5
sam@sam:~$
```

12.3 Example 3: Multidimensional Arrays in Bash:

Multidimensional arrays are not the official part of the Bash programming language. But Bash supports the major programming structures, most importantly loops. Multidimensional arrays can easily be simulated using ***"for"*** loops:

```
#! /bin/bash
declare -a my_array
echo "Enter the number of rows"
read rows
echo "Enter the number of columns"
read cols
for ((x=0; x<rows; x++))
do
for ((y=0; y<cols; y++))
do
my_array[${x},${y}]=$RANDOM #Assigning a random number
```

```
done
done
for ((i=0; i<rows; i++))
do
for ((y=0; y<cols; y++))
do
echo -ne "${my_array[${x},${y}]}\t"
done
echo
done
```

The above code takes rows and columns as input from the user then generates a pseudo-random number from ***0-32767***.

```
#! /bin/bash

declare -a my_array
echo "Enter the number of rows"
read rows
echo "Enter the number of columns"
read cols
for ((x=0; x<rows; x++))
do
        for ((y=0; y<cols; y++))
        do
                my_array[${x},${y}]=$RANDOM
        done
done
for ((i=0; i<rows; i++))
do
        for ((y=0; y<cols; y++))
        do
                echo -ne "${my_array[${x},${y}]}\t"
        done
        echo
done
```

```
sam@sam:~$ bash md_array.sh
Enter the number of rows
3
Enter the number of columns
3
13055   15642   5356
13055   15642   5356
13055   15642   5356
```

12.4 Example 4: Formatting a Poem in Bash:

The following example is another implementation of the array. The script is taking stanza lines as input from the user, format them, and print the entire stanza in the standard output:

```
#! /bin/bash
echo "Enter First line of stanza"
read line[1]
echo "Enter Second line of stanza"
read line[2]
echo "Enter third line of stanza"
read line[3]
echo "Enter fourth line of stanza"
read line[4]
echo "Enter the name of the author"
read line[5]
for i in 1 2 3 4 #Getting four lines of the stanza
do
echo -e " \e[3m${line[i]}\e[10m" #Making the text italic
done
echo -e " \e[4m${line[5]}\e[10m" #Underlining the text
```

```
#! /bin/bash
echo "Enter Firts line of stanza"
read line[1]
echo "Enter Second line of stanza"
read line[2]
echo "Enter the third line of stanza"
read line[3]
echo "Enter fourth line of stanza"
read line[4]
echo "Enter the name of the author"
read line[5]
echo "------------------------------------------------"
for i in 1 2 3 4
do
        #printf "     %s\n" "${line[i]}"
        echo -e "     \e[3m${line[i]}\e[10m"
done
        echo -e "     \e[4m${line[5]}\e[10m"


echo "------------------------------------------------"
~
```

```
sam@sam:~$ bash poem.sh
Enter Firts line of stanza
Friends will sometimes go away
Enter Second line of stanza
Some may disappoint or others betray
Enter the third line of stanza
But new ones will come to stay
Enter fourth line of stanza
In good time.
Enter the name of the author
Abimbola T. Alabi
------------------------------------------------
     Friends will sometimes go away
     Some may disappoint or others betray
     But new ones will come to stay
     In good time.
     Abimbola T. Alabi
------------------------------------------------
```

Conclusion:

The array is one of the critical structures in any programming language. It allows to store different elements of the same data type in a single variable, and those elements can be accessed through index position. Arrays are used in data structure, hash tables, linked lists, or search trees.

Linux is growing, though it has a very small desktop computer market. The primary source to interact with the Linux kernel is the shell. Shell is an interface that assists a user to communicate with the Linux system's kernel. There are various types of shells, but the widely adopted shell is the Bourne Again Shell, also known as Bash. Bash takes command as input from the user and interprets it for the kernel to perform a task.

Similarly, to execute multiple commands or perform a specific task, Bash scripting is used. Bash scripting is also called shell scripting and uses Bash programming language, which is no less than any other scripting language. Like any other programming language, Bash includes everything such as variable defining, conditional statements, and loops. The array is an important data structure that is used to manage the data.

The function of arrays in Bash scripting is the same as other programming languages. But still, arrays are not as advanced in Bash as other scripting or programming languages.

Bash offers two types of arrays, indexed array and associative arrays. Associative arrays were introduced in the fourth version of bash. In the indexed array, the indexes are numeric, whereas, in associative arrays, indexes can be strings. The indexes of associative arrays are also called keys.

Bash provides various array modification options such as inserting an element, deleting an element, replacing an element, and accessing an element at a specific index. Bash arrays can have multiple uses, playlists in music players, and contacts in your

contact list are examples of usage of an array. Moreover, arrays can be used as data management, stacks, queues, heaps, etc.

In Bash, arrays are not as powerful as in other programming languages. There are multiple reasons: Bash is not an object-oriented programming language, the syntax is hard to learn, slow execution time, and vulnerable to errors. Additionally, it does not support multidimensional arrays.

Despite that, arrays can be useful in performing various tasks such as parameter sweep, log alerting while performing cronjobs, and many other programming logic.