Bash: Guide to Bash Arrays

The **Bash array variables** come in two flavors, the **one-dimensional indexed arrays**, and the **associative arrays**. The indexed arrays are sometimes called **lists** and the associative arrays are sometimes called **dictionaries or hash tables**. The support for Bash Arrays simplifies heavily how you can write your shell scripts to support more complex logic or to safely preserve field separation.

This guide covers the standard bash array operations and how to **declare** (**set**), **append**, **iterate over** (**loop**), **check** (**test**), **access** (**get**), and **delete** (**unset**) a value in an **indexed bash array**.

How to declare a Bash Array?

Arrays in Bash are one-dimensional array variables. The declare shell builtin is used to declare array variables and give them attributes using the **-a** and **-A** options.

Note that there is **no upper limit (maximum) on the size (length)** of a Bash array and the values in an Indexed Array and an Associative Array can be any strings or numbers, with the null string being a valid value.

Remember that the **null string** is a zero-length string, which is an empty string. This is not to be confused with the **bash null command** which has a completely different meaning and purpose.

You can create an Indexed Array on the fly in Bash using compound assignment or by using the builtin command **declare**. The **+=** operator allows you to append a value to an indexed Bash array.

```
#!/bin/bash

array=(one two three)
echo ${array[*]}
# output:
one two three

array[5]='five'
```

```
echo ${array[*]}
# output:
one two three five

aray[4]='four'
echo ${array[*]}
# output:
one two three four five

array+=('six')
echo ${array[*]}
# output:
one two three four five six

array=("one" "two" "three" "four items")
echo ${array[*]}
# output:
one two three four items
```

```
#!/bin/bash

array=(1 2 3)
echo ${array[*]}
# output:
1 2 3

array+=('a')
array+=('b')
array+=('c')
array+=('d' 'e')
echo ${array[*]}
# output:
1 2 3 a b c d e
```

When to use double quotes with Bash Arrays?

A great benefit of using Bash Arrays is to preserve field separation. Though, to keep that behavior, you must use double quotes as necessary. In absence of quoting, Bash will split the input into a **list of words** based on the $IFS value which by default contain spaces and tabs.

Default IFS variable

The default value for **$IFS** is `<space><tab><newline>`. You can print it with the following command: `cat -etv <<<"$IFS"`

```
#!/bin/bash

myArray=("1st item" "2nd item" "3rd item" "4th item")
printf 'Word -> %s\n' ${myArray[@]} # word splitting based on first char in $IFS , because array is not quoted
# output:
Word -> 1st
Word -> item
Word -> 2nd
Word -> item
Word -> 3rd
Word -> item
Word -> 4th
Word -> item

myArray=("1st item" "2nd item" "3rd item" "4th item")
printf 'Word -> %s\n' "${myArray[*]}" # word splitting based on first char in $IFS, use the full array as one wor
# output:
Word -> 1st item 2nd item 3rd item 4th item
```

```
myArray=("1st item" "2nd item" "3rd item" "4th item")
printf 'Word -> %s\n' "${myArray[@]}" # use arrays entries, because array variable use quotation marks
# output:
Word -> 1st item
Word -> 2nd item
Word -> 3rd item
Word -> 4th item
```

When performing **WordSplitting** on an **unquoted** expansion, **IFS** is used to split the value of the expansion into multiple words.

When performing the `"$*"` or `"${array[*]}"` expansion (`*` not `@`, and **quoted**), the first character of **IFS** is placed between the elements in order to construct the final output string.

Likewise, when doing `"${!prefix*}"`, the first character of **IFS** is placed between the variable names to make the output string.

example with another IFS variable:

```
#!/bin/bash


# set IFS to '+'
IFS='+'

myArray=("1st item" "2nd item" "3rd item" "4th item")
printf 'Word -> %s\n' ${myArray[@]} # output is array items because first char in $IFS not contain space
# output:
Word -> 1st item
Word -> 2nd item
Word -> 3rd item
Word -> 4th item

myArray=("1st item" "2nd item" "3rd item" "4th item")
printf 'Word -> %s\n' "${myArray[*]}" # use the full array as one word with first char in $IFS as delimeter
# output:
Word -> 1st item+2nd item+3rd item+4th item

myArray=("1st item" "2nd item" "3rd item" "4th item")
printf 'Word -> %s\n' "${myArray[@]}" # use arrays entries, because array variable is quoted
# output:
Word -> 1st item
Word -> 2nd item
Word -> 3rd item
Word -> 4th item

# set IFS to 'e' char
IFS='e'

myArray=("1st item" "2nd item" "3rd item" "4th item")
printf 'Word -> %s\n' ${myArray[@]} # output is array items with 'e' char as delimeter
# output:
Word -> 1st it
Word -> m
Word -> 2nd it
Word -> m
Word -> 3rd it
Word -> m
Word -> 4th it
Word -> m

# set IFS to default value
unset IFS
```

When you need array items, always use **quotation marks** like `"${myArray[@]}"`

Array Operations

How to iterate over a Bash Array? (loop)

As discussed above, you can access all the values of a Bash array using the `*` (asterisk) notation. Though, to iterate through all the array values you should use the `@` (at) notation instead.

The difference between the two will arise when you try to loop over such an array using quotes:

The `*` notation will return all the elements of the array as a **single word** result while the `@` notation will return a value for each element of the Bash array as a **separate word**. This becomes clear when performing a for loopicon mdi-link-variant on such a variable.

The difference is subtle; `$*` creates **one argument**, while `$@` will expand into **separate arguments**

example:

```
#!/bin/bash

array=(a b c d e)

# Using '*'
echo ${array[*]}
# output:
a b c d e

# Using '@'
echo ${array[@]}
# output:
a b c d e

# For Loop Exampe with '*', will echo only once all the values
for value in "${array[*]}"; do echo "$value"; done
# output:
a b c d e

# For Loop Example with '@', will echo individually each values
for value in "${array[@]}"; do echo "$value"; done
# output:
a
b
c
d
e
```

How to iterate over a Bash Array? (loop) using index

```
#!/bin/bash

array=(a b c d e)
for index in "${!array[@]}"; do
    echo "${array[$index]}"
done

#output:
a
b
```

```
c
d
e
```

How to get a subset of an Array?

The shell parameter expansions works on arrays which means that you can use the substring Expansion: `${string:<start>:<count>}` notation to get a subset of an array in bash. Example: `${myArray[@]:2:3}`.

The notation can be use with optional `<start>` and `<count>` parameters. The `${myArray[@]}` notation is equivalent to `${myArray[@]:0}`.

Syntax	Desctiption
${myArray[@]:}	Get the subset of entries from to the end of the array
${myArray[@]::}	Get the subset of entries starting from entry
${myArray[@]::}	Get the subset of entries from the beginning of the array

```
#!/bin/bash

array=(1 2 3 4 5)

echo ${array[@]:2:3}
# output:
3 4 5

echo ${array[@]:1}
# output:
2 3 4 5

echo ${array[@]::2}
# output:
1 2

new_array=( ${array[@]::2} )  # create new array from old array
echo ${new_array[@]::2}
# output:
1 2
```

How to check if a Bash Array is empty?

You can check if an array is empty by checking the length (or size) of the array with the `${#array[@]}` syntax and use a bash **if statementicon** as necessary.

```
#!/bin/bash

array=();

if ! (( ${#array[@]} > 0 )); then
    echo "array is empty";
fi
# output:
array is empty

if [[ ${#array[@]} -eq 0 ]]; then
    echo "array is empty";
fi
# output:
array is empty

array=("" 1);
```

```
if [[ ${#array[@]} -ne 0 ]]; then
    echo "array is not empty";
fi
# output:
array is not empty
```

How to check if a Bash Array contains a value

There is no **in array** operator in bash to check if an array contains a value. Instead, to check if a bash array contains a value you will need to test the values in the array by using a bash conditional expression with the binary operator `=~`. The string to the right of the operator is considered a POSIX extended regular expression and matched accordingly.

Be careful, this will not look for an exact match as it uses a shell regex.

```
#!/bin/bash

# False positive match:
# Return True even for partial match

array=(a1 b1 c1 d1 ee)

[[ ${array[*]} =~ 'a' ]] && echo 'yes' || echo 'no'
# output:
yes

[[ ${array[*]} =~ 'a1' ]] && echo 'yes' || echo 'no'
# output:
yes

[[ ${array[*]} =~ 'e' ]] && echo 'yes' || echo 'no'
# output:
yes

[[ ${array[*]} =~ 'ee' ]] && echo 'yes' || echo 'no'
# output:
yes
```

How to check if a Bash Array contains a value - exact match

In order to look for an exact match, your **regex pattern** needs to add extra space before and after the value like `(^|[[:space:]])"VALUE"($|[[:space:]])`

```
#!/bin/bash

# Exact match

array=(aa1 bc1 ac1 ed1 aee)

if [[ ${array[*]} =~ (^|[[:space:]])"a"($|[[:space:]]) ]]; then
    echo "Yes";
else
    echo "No";
fi
# output:
No

if [[ ${array[*]} =~ (^|[[:space:]])"ac1"($|[[:space:]]) ]]; then
    echo "Yes";
else
```

```
    echo "No";
fi
# output:
Yes

find="ac1"
if [[ ${array[*]} =~ (^|[[:space:]])"$find"($|[[:space:]]) ]]; then
    echo "Yes";
else
    echo "No";
fi
# output:
Yes
```

How Delete item from bash array

```
#!/bin/bash

array=("a" "b" "c" "a" "aa")
del="a"

for index in "${!array[@]}"; do
    if [[ ${array[$index]} == $del ]]; then
        unset 'array[index]'
    fi
done
echo ${array[@]}

# output:
b c aa
```

How to delete elements of one array from another array in bash

```
array=("a" "b" "c" "a" "aa")
del_a=("a" "g" "b")

for index in "${!array[@]}"; do
    for del in "${del_a[@]}"; do
        if [[ ${array[$index]} == $del ]]; then
            unset 'array[index]'
        fi
    done
done
echo ${array[@]}

#output:
c aa
```

Merging arrays in bash

```
#!/bin/bash

array_one=("a" "b" "c" "d")
array_two=("1" "2" "3" "4")
array=("${array_one[@]}" "${array_two[@]}")
echo ${array[@]}
echo ${#array[@]}

# output:
a b c d 1 2 3 4
8
```

```
array_one=("a" "b" "c" "d")
array_two=("1" "2" "3" "4")

# Add new element at the end of the array
array_one=("${array_one[@]}" "another_value")
echo ${array_one[@]}
# output:
a b c d another_value

# Add array elements from another array
array_one=("${array_one[@]}" "${array_two[@]}")
echo ${array_one[@]}
# output
a b c d another_value 1 2 3 4
```

How to store each line of a file into an indexed array?

The easiest and safest way to read a file into a bash array is to use the **mapfile** builtin which read lines from the standard input. When no array variable name is provided to the **mapfile** command, the input will be stored into the **$MAPFILE** variable. Note that the **mapfile** command will split by default on newlines character but will preserve it in the array values, you can remove the trailing delimiter using the **-t** option and change the delimiter using the **-d** option.

```
~]$ mapfile MYFILE < example.txt
~]$ printf '%s' "${MYFILE[@]}"
line 1
line 2
line 3

~]$ mapfile < example.txt # Use default MAPFILE array
~]$ printf '%s' "${MAPFILE[@]}"
line 1
line 2
line 3
```

Bash split string into array

Method 1: Bash split string into array using parenthesis

```
#!/bin/bash

myvar="string1 string2 string3"

# Redefine myvar to myarray using parenthesis
myarray=($myvar)

echo "Number of elements in the array: ${#myarray[@]}"

# output
Number of elements in the array: 3
```

Method 2: Bash split string into array using read

```
#!/bin/bash

myvar="string1 string2 string3"

# Storing as array into myarray
read -a myarray <<< $myvar

echo "Number of elements in the array: ${#myarray[@]}"
```

```
# output
Number of elements in the array: 3
```

Bash split string into array using delimiter

We can combine read with IFS (Internal Field Separator) to define a delimiter.

Assuming your variable contains strings separated by comma character instead of white space as we used in above examples.

We can provide the delimiter value using IFS and create array from string with spaces

```
#!/bin/bash

myvar="string1,string2,string3"

# Here comma is our delimiter value
IFS="," read -a myarray <<< $myvar

echo "Number of elements in the array: ${#myarray[@]}"

# output
Number of elements in the array: 3
```

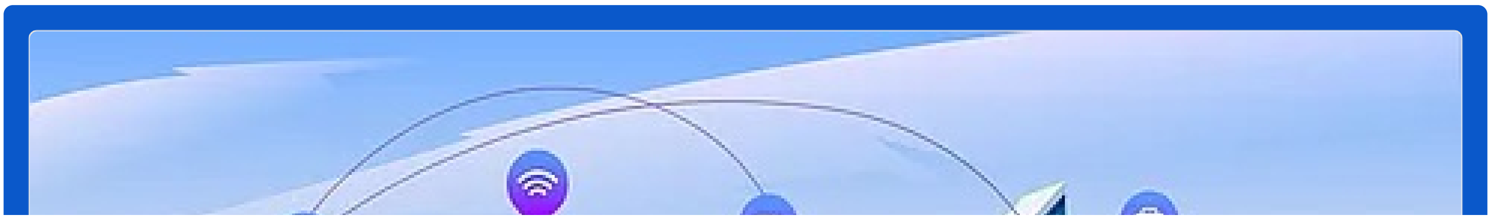
Summary

Syntax	Desctiption
arr=()	Create an empty array
arr=(1 2 3)	Initialize array
${arr[2]}	Retrieve third element
${arr[@]}	Retrieve all elements
${!arr[@]}	Retrieve array indices
${#arr[@]}	Calculate array size
arr[0]=3	Overwrite 1st element
arr+=(4)	Append value(s)
str=$(ls)	Save ls output as a string
arr=($(ls))	Save ls output as an array of files
${arr[@]:s:n}	Retrieve n elements starting at index s

bash

« Previous article

Next article »



Architecture

SUBSCRIBE FOR NEW ARTICLES

@ Email Address

SUBSCRIBE

comments powered by Disqus