Shell Tips!

A Complete Guide On How To Use Bash Arrays

Last updated: 2020-09-26

On This Page

I. Difference between Bash Indexed Arrays and Associative Arrays

II. How to declare a Bash Array?

- Bash Indexed Array (ordered lists)
- Bash Associative Array (dictionaries, hash table, or key/value pair)
- When to use double quotes with Bash Arrays?

III. Array Operations

- How to iterate over a Bash Array? (loop)

- How to shuffle the elements of an Array in a shell script?
- How to sort the elements of an Array in a shell script?
- How to get a subset of an Array?
- How to check if a Bash Array is empty?
- How to check if a Bash Array contains a value?
- How to store each line of a file into an indexed array?

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* and an *associative bash array*. The detailed examples include how to *sort* and *shuffle* arrays.

> 👉 Many fixes and improvements have been made with Bash version 5, read more details with the post What's New in GNU Bash 5?

in which the keys (indexes) are ordered integers. You can think about it as an ordered list of items. Then, an `associative` array, a.k.a `hash table`, is an array in which the keys are represented by arbitrary strings.

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.

Bash Indexed Array (ordered lists)

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.

```
[me@linux ~]$ myIndexedArray=(one two three)
[me@linux ~]$ echo ${myIndexedArray[*]}
one two three

[me@linux ~]$ myIndexedArray[5]='five'
```

```
[me@linux ~]$ myIndexedArray+=('six')
[me@linux ~]$ echo ${myIndexedArray[*]}
one two three four five six
```

With the `declare` built-in command and the lowercase " `-a` " option, you would simply do the following:

```
[me@linux ~]$ declare -a mySecondIndexedArray
[me@linux ~]$ mySecondIndexedArray[0]='zero'
[me@linux ~]$ echo ${mySecondIndexedArray[*]}
zero
```

Bash Associative Array (dictionaries, hash table, or key/value pair)

You *cannot* create an associative array on the fly in Bash. You can only use the `declare` built-in command with the uppercase " `-A` " option. The `+=` operator allows you to append one or multiple key/value to an associative Bash array.

```
[me@linux ~]$ declare -A myAssociativeArray
[me@linux ~]$ myAssociativeArray[a]=123
[me@linux ~]$ myAssociativeArray[b]=456
[me@linux ~]$ myAssociativeArray+=([c]=789
[d]=012)
[me@linux ~]$ echo ${myAssociativeArray[*]}
012 789 456 123
```

> ⚠ Do not confuse `-a` (lowercase) with `-A` (uppercase). It would silently fail. Indeed, declaring an Indexed array will accept subscript but will ignore it and treat the rest of your

> *assigning associative array*".

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.

```
[me@linux ~]$ myArray=("1st item" "2nd item" "3rd item" "4th item")
[me@linux ~]$ printf 'Word -> %s\n' ${myArray[@]}
# word splitting based on $IFS
Word -> 1st
Word -> item
Word -> 2nd
Word -> item
Word -> 3rd
Word -> item
Word -> 4th
Word -> item
[me@linux ~]$ printf 'Word -> %s\n' "${myArray[*]}" # use the full array as one word
Word -> 1st item 2nd item 3rd item 4th item
[me@linux ~]$ printf 'Word -> %s\n' "${myArray[@]}" # use arrays entries
Word -> 1st item
Word -> 2nd item
Word -> 3rd item
Word -> 4th item
```

Array Operations

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 loop on such a variable.

```
[me@linux ~]$ myDemoArray=(1 2 3 4 5)

# Using '*'
[me@linux ~]$ echo ${myDemoArray[*]}
1 2 3 4 5

# Using '@'
[me@linux ~]$ echo ${myDemoArray[@]}
1 2 3 4 5

# For Loop Exampe with '*', will echo only once
all the values
[me@linux ~]$ myDemoArray=(1 2 3 4 5)
[me@linux ~]$ for value in "${myDemoArray[*]}"; do
echo "$value"; done
1 2 3 4 5

# For Loop Example with '@', will echo
individually each values
[me@linux ~]$ myDemoArray=(1 2 3 4 5)
[me@linux ~]$ for value in "${myDemoArray[@]}"; do
echo "$value"; done
1
2
3
```

When looping over a bash array it's often useful to access the keys of the array separately of the values. This can be done by using the `!` (bang) notation.

```
# Print Indexed Array Keys
[me@linux ~]$ for keys in "${!myDemoArray[@]}"; do
echo "$keys"; done
0
1
2
3
4

# Print Associative Array Values
[me@linux ~]$ myAssociativeArray=([a]=123 [b]=456)
[me@linux ~]$ for value in
"${myAssociativeArray[@]}"; do echo "$value"; done
456
123

# Print Associative Array Keys
[me@linux ~]$ myAssociativeArray=([a]=123 [b]=456)
[me@linux ~]$ for keys in
"${!myAssociativeArray[@]}"; do echo "$keys"; done
b
a

# Iterate over key and value of an Associative
Array
[me@linux ~]$ myAssociativeArray=([a]=123 [b]=456)
[me@linux ~]$ for key in
"${!myAssociativeArray[@]}"
> do
>   echo -n "key  : $key, "
```

Another useful aspect of manipulating Bash Arrays is to be able to get the total count of all the elements in an array. You can get the length (i.e. size) of an Array variable with the `#` (hashtag) notation.

```
[me@linux ~]$ myArray=(a b c d)
[me@linux ~]$ echo "myArray contain ${#myArray[*]} elements"
myArray contain 4 elements

[me@linux ~]$ myAssociativeArray=([a]=123 [b]=456)
[me@linux ~]$ echo "myAssociativeArray contain ${#myAssociativeArray[*]} elements"
myAssociativeArray contain 2 elements
```

How to remove a key from a Bash Array or delete the full array? (delete)

The `unset` bash builtin command is used to *unset* (delete or remove) any values and attributes from a shell variable or function. This means that you can simply use it to *delete* a Bash array in full or only *remove* part of it by specifying the key. `unset` take the variable name as an argument, so don't forget to remove the `$` (dollar) sign in front of the variable name of your array. See the complete example below.

```
[me@linux ~]$ declare -A myArray=([one]=un [two]=deux [three]=trois)
[me@linux ~]$ echo ${myArray[*]}
deux trois un
[me@linux ~]$ echo ${myArray[one]}
un
[me@linux ~]$ unset myArray[one]
```

How to shuffle the elements of an Array in a shell script?

There are two reasonable options to shuffle the elements of a bash array in a shell script. First, you can either use the external command-line tool `shuf` that comes with the GNU coreutils, or `sort -R` in older coreutils versions. Second, you can use a native bash implementation with only shell builtin and a randomization function. Both methods presented below assume the use of an indexed array, it will not work with associative arrays.

The `shuf` command line generates random permutations from a file or the standard input. By using the `-e` option, `shuf` would treat each argument as a separate input line. ***Do not forget to use the double-quote otherwise elements with whitespaces will be split***. Once the array is shuffled we can reassign its new value to the same variable.

```
[me@linux ~]$ myArray=("1st item" "2nd item" "3rd
item")
[me@linux ~]$ echo ${myArray[@]}
1st item 2nd item 3rd item

# CORRECT: Uses double-quote
[me@linux ~]$ shuf -e "${myArray[@]}"
2nd item
1st item
3rd item

# WRONG: Missing double-quote
[me@linux ~]$ shuf -e ${myArray[@]}
2nd
```

```
)
[me@linux ~]$ echo ${myArray[@]}
2nd item 1st item 3rd item
```

The second option to shuffle the elements of a bash array is to implement an unbiased algorithm like the Fisher-Yates shuffle. The challenge to implement such a solution is that you may need to few bash tricks to work around some limitations. For example, a bash function return value is limited to number values between 0 and 255, nor can you *safely* use indirection in bash. This example will implement a `rand` function and a `shuffle` function. Both functions use local and global variables to pass values around. If you need to shuffle an array larger than 32768 entries or your array is not a dense indexed array, then use the first method above using `shuf`.

We want to ensure that every permutation is equally likely when shuffling the array. The first function `rand` is used to generates a random number instead of using `$((RANDOM % i))` with a modulo operator in bash arithmetic which would produce a biased number. We compensate this by using a range of values that is a multiple of the $RANDOM modulus. The $RANDOM number range between 0 and 32767. Since we still rely on this number, it will limit our max random number generator to 32768 values. We use a bash while loop with the bash null command to iterate over a series of $RANDOM numbers until we get one below the max value. We use a bash if statement to ensure that we don't end up in an infinite loop in cases where max value is zero which would happen if we provide a number larger than 32768 to `rand`.

The `shuffle` function uses a bash for loop to permute they entries in the array based on the unbiased random number we

```
    while (( (_RANDOM=$RANDOM) >= max )); do :;
done
    _RANDOM=$(( _RANDOM % $1 ))
  else
    return 1
  fi
}

# shuffle an array using the rand function
# GLOBALS: _array, _RANDOM
shuffle() {
  local i tmp size
  size=${#_array[*]}
  for ((i=size-1; i>0; i--)); do
    if ! rand $((i+1)); then exit 1; fi
    tmp=${_array[i]} _array[i]=${_array[$_RANDOM]}
_array[$_RANDOM]=$tmp
  done
}

[me@linux ~]$ myArray=("1st item" "2nd item" "3rd
item")
[me@linux ~]$ echo ${myArray[@]}
1st item 2nd item 3rd item
[me@linux ~]$ _array=("${myArray[@]}"); shuffle ;
myArray=("${_array[@]}")
[me@linux ~]$ echo ${myArray[@]}
2nd item 1st item 3rd item
```

How to sort the elements of an Array in a shell script?

sorted.

The example below is a shell script implementation of a *bubble sort* algorithm on a list of dates. It uses the `date` command to do date comparison in bash and sort the dates in descending order.

```
#!/usr/bin/bash
# Example script using bubble sort algorithm in
bash
swapDates() {
  local tmp=${dates[$1]}
  dates[$1]=${dates[$2]}
  dates[$2]=$tmp
}

bubblesort() {
  local size=${#dates[@]} # Size of sortable items

  echo -e "\nArray size: $size"

  n=$size
  until (( n <= 0 )); do
    newn=0
    echo -e "\nIteration: $((size-n))"
    for ((i=0; i < n; i++)); do
      if ((
          ( ${#dates[i+1]} > 0 ) && \
          ( $(date -d "${dates[i+1]}" +%s) >
$(date -d "${dates[i]}" +%s) )
      )); then
        echo swap "${dates[i+1]}" with
"${dates[i]}"
        swapDates $((i+1)) $i
        newn=$i
```

```
}

dates=('Feb 13' 'Jan 17' 'Apr 12' 'Mar 24' 'Apr 6'
'Jan 20')
echo "Unsorted Array: ${dates[@]}"
bubblesort
echo "Sorted Array: ${dates[@]}"

# Example Output
[me@linux ~]$ ./dates-bubble-sort
Unsorted Array: Feb 13 Jan 17 Apr 12 Mar 24 Apr 6
Jan 20

Array size: 6

Iteration: 0
swap Apr 12 with Jan 17
swap Mar 24 with Jan 17
swap Apr 6 with Jan 17
swap Jan 20 with Jan 17

Iteration: 2
swap Apr 12 with Feb 13
swap Mar 24 with Feb 13
swap Apr 6 with Feb 13

Iteration: 4
swap Apr 6 with Mar 24

Iteration: 5
Array sorted

Sorted Array: Apr 12 Apr 6 Mar 24 Feb 13 Jan 20
Jan 17
```

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

`${myArray[@]:<start>}`	Get the subset of entries from `<start>` to the end of the array
`${myArray[@]:<start>:<count>}`	Get the subset of `<count>` entries starting from `<start>` entry
`${myArray[@]::<count>}`	Get the subset of `<count>` entries from the beginning of the array

```
[me@linux ~]$ myArray=("1st item" "2nd item" "3rd item" "4th item")
[me@linux ~]$ echo ${myArray[@]:2:3}
3rd item 4th item
[me@linux ~]$ echo ${myArray[@]:1}
2nd item 3rd item 4th item
[me@linux ~]$ echo ${myArray[@]::2}
1st item 2nd item
```

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 statement as necessary.

```
[me@linux ~]$ myArray=();
[me@linux ~]$ if ! (( ${#myArray[@]} > 0 )); then echo "myArray is empty"; fi
myArray is 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.

```
[me@linux ~]$ myArray=(a b c d)
[me@linux ~]$ [[ ${myArray[*]} =~ 'a' ]] && echo
'yes' || echo 'no'
yes
[me@linux ~]$ [[ ${myArray[*]} =~ 'e' ]] && echo
'yes' || echo 'no'
no

# Return True even for partial match
[me@linux ~]$ myArray=(a1 b1 c1 d1 ee)
[me@linux ~]$ [[ ${myArray[*]} =~ 'a' ]] && echo
'yes' || echo 'no'
yes
[me@linux ~]$ [[ ${myArray[*]} =~ 'a1' ]] && echo
'yes' || echo 'no'
yes
[me@linux ~]$ [[ ${myArray[*]} =~ 'e' ]] && echo
'yes' || echo 'no'
yes
[me@linux ~]$ [[ ${myArray[*]} =~ 'ee' ]] && echo
'yes' || echo 'no'
```

```
]] && echo 'yes' || echo 'no'
yes
[me@linux ~]$ [[ ${myAssociativeArray[*]} =~ 1234
]] && echo 'yes' || echo 'no'
no
```

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:]])`.

```
[me@linux ~]$ [[ ${myAssociativeArray[*]} =~ (^|
[[:space:]])"12"($|[[:space:]]) ]] && echo 'yes'
|| echo 'no'
no
[me@linux ~]$ [[ ${myAssociativeArray[*]} =~ (^|
[[:space:]])"123"($|[[:space:]]) ]] && echo 'yes'
|| echo 'no'
yes
```

With the Bash Associative Arrays, you can extend the solution to test values with `[[ -z "${myArray[$value]}" ]]`.

```
# delete previously set declaration of myArray and
# prevent the error `bash: myArray: cannot convert
indexed to associative array`
[me@linux ~]$ unset myArray

[me@linux ~]$ declare -A myArray=([one]=un
[two]=deux [three]=trois)
[me@linux ~]$ echo ${myArray[*]}
deux trois un

[me@linux ~]$ for value in one two three four
```

```
two is a member of ( two three one )
three is a member of ( two three one )
four is *not* a member of ( two three one )
```

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.

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

The Complete How To Guide of Bash Functions

How To Create Simple Menu with the Shell Select Loop?

What is the Right Way to do Bash Loops?

5 Mistakes To Avoid For Writing High-Quality Bash Comments

How To Script Error Free Bash If Statement?

How To Format Date and Time in Linux, macOS, and Bash?

DISCLAIMERS & CONTACTS