
6.7 Arrays

Bash provides one-dimensional indexed and associative array variables. Any variable may be used as an indexed array; the `declare` builtin will explicitly declare an array. There is no maximum limit on the size of an array, nor any requirement that members be indexed or assigned contiguously. Indexed arrays are referenced using integers (including arithmetic expressions (see Shell Arithmetic)) and are zero-based; associative arrays use arbitrary strings. Unless otherwise noted, indexed array indices must be non-negative integers.

An indexed array is created automatically if any variable is assigned to using the syntax

```
name[subscript]=value
```

The *subscript* is treated as an arithmetic expression that must evaluate to a number. To explicitly declare an array, use

```
declare -a name
```

The syntax

```
declare -a name[subscript]
```

is also accepted; the *subscript* is ignored.

Associative arrays are created using

```
declare -A name
```

Attributes may be specified for an array variable using the `declare` and `readonly` builtins. Each attribute applies to all members of an array.

Arrays are assigned to using compound assignments of the form

```
name=(value1 value2 … )
```

where each *value* may be of the form `[`*subscript*`]=`*string*. Indexed array assignments do not require anything but *string*. When assigning to indexed arrays, if the optional subscript is supplied, that index is assigned to; otherwise the index of the element assigned is the last index assigned to by the statement plus one. Indexing starts at zero.

Each *value* in the list undergoes all the shell expansions described above (see Shell Expansions).

When assigning to an associative array, the words in a compound assignment may be either assignment statements, for which the subscript is required, or a list of words that is interpreted as a sequence of alternating keys and values: *name*=(*key1 value1 key2 value2* …). These are treated identically to *name*=([*key1*]=*value1* [*key2*]=*value2* …). The first word in the list determines how the remaining words are interpreted; all

assignments in a list must be of the same type. When using key/value pairs, the keys may not be missing or empty; a final missing value is treated like the empty string.

This syntax is also accepted by the `declare` builtin. Individual array elements may be assigned to using the *`name`*`[`*`subscript`*`]=`*`value`* syntax introduced above.

When assigning to an indexed array, if *name* is subscripted by a negative number, that number is interpreted as relative to one greater than the maximum index of *name*, so negative indices count back from the end of the array, and an index of -1 references the last element.

The '+=' operator will append to an array variable when assigning using the compound assignment syntax; see Shell Parameters above.

Any element of an array may be referenced using `${`*`name`*`[`*`subscript`*`]}`. The braces are required to avoid conflicts with the shell's filename expansion operators. If the *subscript* is '@' or '*', the word expands to all members of the array *name*. These subscripts differ only when the word appears within double quotes. If the word is double-quoted, `${`*`name`*`[*]}` expands to a single word with the value of each array member separated by the first character of the `IFS` variable, and `${`*`name`*`[@]}` expands each element of *name* to a separate word. When there are no array members, `${`*`name`*`[@]}` expands to nothing. If the double-quoted expansion occurs within a word, the expansion of the first parameter is joined with the beginning part of the original word, and the expansion of the last parameter is joined with the last part of the original word. This is

analogous to the expansion of the special parameters '@' and '*'. `${#name[subscript]}` expands to the length of `${name[subscript]}`. If *subscript* is '@' or '*', the expansion is the number of elements in the array. If the *subscript* used to reference an element of an indexed array evaluates to a number less than zero, it is interpreted as relative to one greater than the maximum index of the array, so negative indices count back from the end of the array, and an index of -1 refers to the last element.

Referencing an array variable without a subscript is equivalent to referencing with a subscript of 0. Any reference to a variable using a valid subscript is legal, and `bash` will create an array if necessary.

An array variable is considered set if a subscript has been assigned a value. The null string is a valid value.

It is possible to obtain the keys (indices) of an array as well as the values. ${!*name*[@]} and ${!*name*[*]} expand to the indices assigned in array variable *name*. The treatment when in double quotes is similar to the expansion of the special parameters '@' and '*' within double quotes.

The `unset` builtin is used to destroy arrays. `unset name[subscript]` destroys the array element at index *subscript*. Negative subscripts to indexed arrays are interpreted as described above. Unsetting the last element of an array variable does not unset the variable. `unset name`, where *name* is an array, removes the entire array. `unset name[subscript]` behaves differently depending on the array type when given a subscript of '*' or '@'. When *name* is an associative array, it removes the

element with key '*' or '@'. If *name* is an indexed array, `unset` removes all of the elements, but does not remove the array itself.

When using a variable name with a subscript as an argument to a command, such as with `unset`, without using the word expansion syntax described above, the argument is subject to the shell's filename expansion. If filename expansion is not desired, the argument should be quoted.

The `declare`, `local`, and `readonly` builtins each accept a `-a` option to specify an indexed array and a `-A` option to specify an associative array. If both options are supplied, `-A` takes precedence. The `read` builtin accepts a `-a` option to assign a list of words read from the standard input to an array, and can read values from the standard input into individual array elements. The `set` and `declare` builtins display array values in a way that allows them to be reused as input.

Next: The Directory Stack, Previous: Aliases, Up: Bash Features [Contents][Index]