Arrays

Purpose

An array is a parameter that holds mappings from keys to values. Arrays are used to store a collection of parameters into a parameter. Arrays (in any programming language) are a useful and common composite data structure, and one of the most important scripting features in Bash and other shells.

Here is an **abstract** representation of an array named `NAMES`. The indexes go from 0 to 3.

```
NAMES
 0: Peter
 1: Anna
 2: Greg
 3: Jan
```

Instead of using 4 separate variables, multiple related variables are grouped grouped together into *elements* of the array, accessible by their *key*. If you want the second name, ask for index 1 of the array `NAMES`.

Indexing

Bash supports two different types of ksh-like one-dimensional arrays. **Multidimensional arrays are not implemented**.

- *Indexed arrays* use positive integer numbers as keys. Indexed arrays are **always sparse**, meaning indexes are not necessarily contiguous. All syntax used for both assigning and dereferencing indexed arrays is an arithmetic evaluation context (see Referencing). As in C and many other languages, the numerical array indexes start at 0 (zero). Indexed arrays are the most common, useful, and portable type. Indexed arrays were first introduced to Bourne-like shells by ksh88. Similar, partially compatible syntax was inherited by many derivatives including Bash. Indexed arrays always carry the `-a` attribute.
- *Associative arrays* (sometimes known as a "hash" or "dict") use arbitrary nonempty strings as keys. In other words, associative arrays allow you to look up a value from a table based upon its corresponding string label. **Associative arrays are always unordered**, they merely *associate* key-value pairs. If you retrieve multiple values from the array at once, you can't count on them coming out in the same order you put them in. Associative arrays always carry the `-A` attribute, and unlike indexed arrays, Bash requires that they always be declared explicitly (as indexed arrays are

the default, see declaration). Associative arrays were first introduced in ksh93, and similar mechanisms were later adopted by Zsh and Bash version 4. These three are currently the only POSIX-compatible shells with any associative array support.

Syntax

Referencing

To accommodate referring to array variables and their individual elements, Bash extends the parameter naming scheme with a subscript suffix. Any valid ordinary scalar parameter name is also a valid array name: `[[:alpha:]_][[:alnum:]_]*`. The parameter name may be followed by an optional subscript enclosed in square brackets to refer to a member of the array.

The overall syntax is `arrname[subscript]` - where for indexed arrays, `subscript` is any valid arithmetic expression, and for associative arrays, any nonempty string. Subscripts are first processed for parameter and arithmetic expansions, and command and process substitutions. When used within parameter expansions or as an argument to the unset builtin, the special subscripts `*` and `@` are also accepted which act upon arrays analogously to the way the `@` and `*` special parameters act upon the positional parameters. In parsing the subscript, bash ignores any text that follows the closing bracket up to the end of the parameter name.

With few exceptions, names of this form may be used anywhere ordinary parameter names are valid, such as within arithmetic expressions, parameter expansions, and as arguments to builtins that accept parameter names. An *array* is a Bash parameter that has been given the `-a` (for indexed) or `-A` (for associative) *attributes*. However, any regular (non-special or positional) parameter may be validly referenced using a subscript, because in most contexts, referring to the zeroth element of an array is synonymous with referring to the array name without a subscript.

```
# "x" is an ordinary non-array parameter.
$ x=hi; printf '%s ' "$x" "${x[0]}"; echo "${_[0]}"
hi hi hi
```

The only exceptions to this rule are in a few cases where the array variable's name refers to the array as a whole. This is the case for the `unset` builtin (see destruction) and when declaring an array without assigning any values (see declaration).

Declaration

The following explicitly give variables array attributes, making them arrays:

Syntax	Description
`ARRAY=()`	Declares an **indexed** array `ARRAY` and initializes it to be empty. This can also be used to empty an existing array.

Syntax	Description
`ARRAY[0]=`	Generally sets the first element of an **indexed** array. If no array `ARRAY` existed before, it is created.
`declare -a ARRAY`	Declares an **indexed** array `ARRAY`. An existing array is not initialized.
`declare -A ARRAY`	Declares an **associative** array `ARRAY`. This is the one and only way to create associative arrays.

As an example, and for use below, let's declare our `NAMES` array as described above:

```
declare -a NAMES=('Peter' 'Anna' 'Greg' 'Jan')
```

Storing values

Storing values in arrays is quite as simple as storing values in normal variables.

Syntax	Description
`ARRAY[N]=VALUE`	Sets the element `N` of the **indexed** array `ARRAY` to `VALUE`. **`N` can be any valid arithmetic expression**.
`ARRAY[STRING]=VALUE`	Sets the element indexed by `STRING` of the **associative array** `ARRAY`.
`ARRAY=VALUE`	As above. If no index is given, as a default the zeroth element is set to `VALUE`. Careful, this is even true of associative arrays - there is no error if no key is specified, and the value is assigned to string index "0".
`ARRAY=(E1 E2 …)`	Compound array assignment - sets the whole array `ARRAY` to the given list of elements indexed sequentially starting at zero. The array is unset before assignment unless the += operator is used. When the list is empty (`ARRAY=()`), the array will be set to an empty array. This method obviously does not use explicit indexes. An **associative array** can **not** be set like that! Clearing an associative array using `ARRAY=()` works.
`ARRAY=([X]=E1 [Y]=E2 …)`	Compound assignment for indexed arrays with index-value pairs declared individually (here for example `X` and `Y`). X and Y are arithmetic expressions. This syntax can be combined with the above - elements declared without an explicitly specified index are assigned sequentially starting at either the last element with an explicit index, or zero.
`ARRAY=([S1]=E1 [S2]=E2 …)`	Individual mass-setting for **associative arrays**. The named indexes (here: `S1` and `S2`) are strings.

Syntax	Description
`ARRAY+=(E1 E2 …)`	Append to ARRAY.
`ARRAY=("${ANOTHER_ARRAY[@]}")`	Copy ANOTHER_ARRAY to ARRAY, copying each element.

As of now, arrays can't be exported.

Getting values

For completeness and details on several parameter expansion variants, see the article about parameter expansion and check the notes about arrays.

Syntax	Description
`${ARRAY[N]}`	Expands to the value of the index `N` in the **indexed** array `ARRAY`. If `N` is a negative number, it's treated as the offset from the maximum assigned index (can't be used for assignment) - 1
`${ARRAY[S]}`	Expands to the value of the index `S` in the **associative** array `ARRAY`.
`"${ARRAY[@]}"` `${ARRAY[@]}` `"${ARRAY[*]}"` `${ARRAY[*]}`	Similar to mass-expanding positional parameters, this expands to all elements. If unquoted, both subscripts `*` and `@` expand to the same result, if quoted, `@` expands to all elements individually quoted, `*` expands to all elements quoted as a whole.
`"${ARRAY[@]:N:M}"` `${ARRAY[@]:N:M}` `"${ARRAY[*]:N:M}"` `${ARRAY[*]:N:M}`	Similar to what this syntax does for the characters of a single string when doing substring expansion, this expands to `M` elements starting with element `N`. This way you can mass-expand individual indexes. The rules for quoting and the subscripts `*` and `@` are the same as above for the other mass-expansions.

For clarification: When you use the subscripts `@` or `*` for mass-expanding, then the behaviour is exactly what it is for `$@` and `$*` when mass-expanding the positional parameters. You should read this article to understand what's going on.

Metadata

Syntax	Description
`${#ARRAY[N]}`	Expands to the **length** of an individual array member at index `N` (**stringlength**)
`${#ARRAY[STRING]}`	Expands to the **length** of an individual associative array member at index `STRING` (**stringlength**)
`${#ARRAY[@]}` `${#ARRAY[*]}`	Expands to the **number of elements** in `ARRAY`

Syntax	Description
`${!ARRAY[@]}` `${!ARRAY[*]}`	Expands to the **indexes** in `ARRAY` since BASH 3.0

Destruction

The unset builtin command is used to destroy (unset) arrays or individual elements of arrays.

Syntax	Description

`unset -v ARRAY`
`unset -v ARRAY[@]`
`unset -v ARRAY[*]`
	Destroys a complete array

`unset -v ARRAY[N]`	Destroys the array element at index `N`
`unset -v ARRAY[STRING]`	Destroys the array element of the associative array at index `STRING`

It is best to explicitly specify -v when unsetting variables with unset.

Specifying unquoted array elements as arguments to any command, such as with the syntax above **may cause pathname expansion to occur** due to the presence of glob characters.
Example: You are in a directory with a file named `x1`, and you want to destroy an array element `x[1]`, with

```
unset x[1]
```

then pathname expansion will expand to the filename `x1` and break your processing!

Even worse, if `nullglob` is set, your array/index will disappear.

To avoid this, **always quote** the array name and index:

```
unset -v 'x[1]'
```

This applies generally to all commands which take variable names as arguments. Single quotes preferred.

Usage

Numerical Index

Numerical indexed arrays are easy to understand and easy to use. The Purpose and Indexing chapters above more or less explain all the needed background theory.

Now, some examples and comments for you.

Let's say we have an array `sentence` which is initialized as follows:

```
sentence=(Be liberal in what you accept, and conservative in what you
send)
```

Since no special code is there to prevent word splitting (no quotes), every word there will be assigned to an individual array element. When you count the words you see, you should get 12. Now let's see if Bash has the same opinion:

```
$ echo ${#sentence[@]}
12
```

Yes, 12. Fine. You can take this number to walk through the array. Just **subtract 1 from the number of elements, and start your walk at 0 (zero)**:

```
((n_elements=${#sentence[@]}, max_index=n_elements - 1))

for ((i = 0; i <= max_index; i++)); do
  echo "Element $i: '${sentence[i]}'"
done
```

You always have to remember that, it seems newbies have problems sometimes. Please understand that **numerical array indexing begins at 0 (zero)**!

The method above, walking through an array by just knowing its number of elements, only works for arrays where all elements are set, of course. If one element in the middle is removed, then the calculation is nonsense, because the number of elements doesn't correspond to the highest used index anymore (we call them "*sparse arrays*").

Now, suppose that you want to replace your array `sentence` with the values in the previously-declared array `NAMES`. You might think you could just do

```
$ unset sentence ; declare -a sentence=NAMES
$ echo ${#sentence[@]}
1
# omit calculating max_index as above, and iterate as one-liner
$ for ((i = 0; i < ${#sentence[@]}; i++)); do  echo "Element $i: '${s
entence[i]}'" ; done
Element 0: 'NAMES'
```

Obviously that's wrong. What about

```
$ unset sentence ; declare -a sentence=${NAMES}
```

? Again, wrong:

```
$ echo ${#sentence[*]}
1
$ for ((i = 0; i < ${#sentence[@]}; i++)); do  echo "Element $i: '${s
entence[i]}'" ; done
Element 0: 'Peter'
```

So what's the **right** way? The (slightly ugly) answer is, reuse the enumeration syntax:

```
$ unset sentence ; declare -a sentence=("${NAMES[@]}")
$ echo ${#sentence[@]}
4
$ for ((i = 0; i < ${#sentence[@]}; i++)); do  echo "Element $i: '${s
entence[i]}'" ; done
Element 0: 'Peter'
Element 1: 'Anna'
Element 2: 'Greg'
Element 3: 'Jan'
```

Associative (Bash 4)

Associative arrays (or *hash tables*) are not much more complicated than numerical indexed arrays. The numerical index value (in Bash a number starting at zero) just is replaced with an arbitrary string:

```
# declare -A, introduced with Bash 4 to declare an associative array
declare -A sentence

sentence[Begin]='Be liberal in what'
sentence[Middle]='you accept, and conservative'
sentence[End]='in what you send'
sentence['Very end']=...
```

Beware: don't rely on the fact that the elements are ordered in memory like they were declared, it could look like this:

```
# output from 'set' command
sentence=([End]="in what you send" [Middle]="you accept, and conserva
tive " [Begin]="Be liberal in what " ["Very end"]="...")
```

This effectively means, you can get the data back with `"${sentence[@]}"`, of course (just like with numerical indexing), but you can't rely on a specific order. If you want to store ordered data, or re-order data, go with numerical indexes. For associative arrays, you usually query known index values:

```
for element in Begin Middle End "Very end"; do
    printf "%s" "${sentence[$element]}"
done
printf "\n"
```

A nice code example: Checking for duplicate files using an associative array indexed with the SHA sum of the files:

```
# Thanks to Tramp in #bash for the idea and the code

unset flist; declare -A flist;
while read -r sum fname; do
    if [[ ${flist[$sum]} ]]; then
        printf 'rm -- "%s" # Same as >%s<\n' "$fname" "${flist[$su
m]}"
    else
        flist[$sum]="$fname"
    fi
done <  <(find . -type f -exec sha256sum {} +)  >rmdups
```

Integer arrays

Any type attributes applied to an array apply to all elements of the array. If the integer attribute is set for either indexed or associative arrays, then values are considered as arithmetic for both compound and ordinary assignment, and the += operator is modified in the same way as for ordinary integer variables.

```
 ~ $ ( declare -ia 'a=(2+4 [2]=2+2 [a[2]]="a[2]")' 'a+=(42 [a[4]]+=
3)'; declare -p a )
declare -ai a='([0]="6" [2]="4" [4]="7" [5]="42")'
```

`a[0]` is assigned to the result of `2+4`. `a[2]` gets the result of `2+2`. The last index in the first assignment is the result of `a[2]`, which has already been assigned as `4`, and its value is also given `a[2]`.

This shows that even though any existing arrays named `a` in the current scope have already been unset by using `=` instead of `+=` to the compound assignment, arithmetic variables within keys can self-reference any elements already assigned within the same compound-assignment. With integer arrays this also applies to expressions to the right of the `=`. (See evaluation order, the right side of an arithmetic assignment is typically evaluated first in Bash.)

The second compound assignment argument to declare uses `+=`, so it appends after the last element of the existing array rather than deleting it and creating a new array, so `a[5]` gets `42`.

Lastly, the element whose index is the value of `a[4]` (`4`), gets `3` added to its existing value, making `a[4] == 7`. Note that having the integer attribute set this time causes += to add, rather than append a string, as it would for a non-integer array.

The single quotes force the assignments to be evaluated in the environment of `declare`. This is important because attributes are only applied to the assignment after assignment arguments are processed. Without them the `+=` compound assignment would have been invalid, and strings would have been inserted into the integer array without evaluating the arithmetic. A special-case of this is shown in the next section.

Bash declaration commands are really keywords in disguise. They magically parse arguments to determine whether they are in the form of a valid assignment. If so, they are evaluated as assignments. If not, they are undergo normal argument expansion before

being passed to the builtin which evaluates the resulting string as an assignment (somewhat like `eval`, but there are differences.) `'Todo:'` Discuss this in detail.

Indirection

Arrays can be expanded indirectly using the indirect parameter expansion syntax. Parameters whose values are of the form: `name[index]`, `name[@]`, or `name[*]` when expanded indirectly produce the expected results. This is mainly useful for passing arrays (especially multiple arrays) by name to a function.

This example is an "isSubset"-like predicate which returns true if all key-value pairs of the array given as the first argument to isSubset correspond to a key-value of the array given as the second argument. It demonstrates both indirect array expansion and indirect key-passing without eval using the aforementioned special compound assignment expansion.

```
isSubset() {
    local -a 'xkeys=("${!'"$1"'[@]}")' 'ykeys=("${!'"$2"'[@]}")'
    set -- "${@/%/[key]}"

    (( ${#xkeys[@]} <= ${#ykeys[@]} )) || return 1

    local key
    for key in "${xkeys[@]}"; do
        [[ ${!2+_} && ${!1} == ${!2} ]] || return 1
    done
}

main() {
    # "a" is a subset of "b"
    local -a 'a=({0..5})' 'b=({0..10})'
    isSubset a b
    echo $? # true

    # "a" contains a key not in "b"
    local -a 'a=([5]=5 {6..11})' 'b=({0..10})'
    isSubset a b
    echo $? # false

    # "a" contains an element whose value != the corresponding member
of "b"
    local -a 'a=([5]=5 6 8 9 10)' 'b=({0..10})'
    isSubset a b
    echo $? # false
}

main
```

This script is one way of implementing a crude multidimensional associative array by storing array definitions in an array and referencing them through indirection. The script takes two keys and dynamically calls a function whose name is resolved from the array.

```
callFuncs() {
    # Set up indirect references as positional parameters to minimize
local name collisions.
    set -- "${@:1:3}" ${2+'a["$1"]' "$1"'["$2"]'}

    # The only way to test for set but null parameters is unfortunate
ly to test each individually.
    local x
    for x; do
        [[ $x ]] || return 0
    done

    local -A a=(
        [foo]='([r]=f [s]=g [t]=h)'
        [bar]='([u]=i [v]=j [w]=k)'
        [baz]='([x]=l [y]=m [z]=n)'
        ) ${4+${a["$1"]+"${1}=${!3}"}} # For example, if "$1" is "ba
r" then define a new array: bar=([u]=i [v]=j [w]=k)

    ${4+${a["$1"]+"${!4-:}"}} # Now just lookup the new array. for in
puts: "bar" "v", the function named "j" will be called, which prints
"j" to stdout.
}

main() {
    # Define functions named {f..n} which just print their own names.
    local fun='() { echo "$FUNCNAME"; }' x

    for x in {f..n}; do
        eval "${x}${fun}"
    done

    callFuncs "$@"
}

main "$@"
```

Bugs and Portability Considerations

- Arrays are not specified by POSIX. One-dimensional indexed arrays are supported using similar syntax and semantics by most Korn-like shells.
- Associative arrays are supported via `typeset -A` in Bash 4, Zsh, and Ksh93.
- In Ksh93, arrays whose types are not given explicitly are not necessarily indexed. Arrays defined using compound assignments which specify subscripts are associative by default. In Bash, associative arrays can *only* be created by explicitly declaring them as associative, otherwise they are always indexed. In addition, ksh93 has several other compound structures whose types can be determined by the compound assignment syntax used to create them.
- In Ksh93, using the `=` compound assignment operator unsets the array, including any attributes that have been set on the array prior to assignment. In order to preserve attributes, you must use the `+=` operator. However, declaring an

associative array, then attempting an `a=(…)` style compound assignment without specifying indexes is an error. I can't explain this inconsistency.

```
 $ ksh -c 'function f { typeset -a a; a=([0]=foo [1]=bar); types
et -p a; }; f' # Attribute is lost, and since subscripts are giv
en, we default to associative.
typeset -A a=([0]=foo [1]=bar)
 $ ksh -c 'function f { typeset -a a; a+=([0]=foo [1]=bar); type
set -p a; }; f' # Now using += gives us the expected results.
typeset -a a=(foo bar)
 $ ksh -c 'function f { typeset -A a; a=(foo bar); typeset -p a;
}; f' # On top of that, the reverse does NOT unset the attribut
e. No idea why.
 ksh: f: line 1: cannot append index array to associative array
a
```

- Only Bash and mksh support compound assignment with mixed explicit subscripts and automatically incrementing subscripts. In ksh93, in order to specify individual subscripts within a compound assignment, all subscripts must be given (or none). Zsh doesn't support specifying individual subscripts at all.
- Appending to a compound assignment is a fairly portable way to append elements after the last index of an array. In Bash, this also sets append mode for all individual assignments within the compound assignment, such that if a lower subscript is specified, subsequent elements will be appended to previous values. In ksh93, it causes subscripts to be ignored, forcing appending everything after the last element. (Appending has different meaning due to support for multi-dimensional arrays and nested compound datastructures.)

```
 $ ksh -c 'function f { typeset -a a; a+=(foo bar baz); a+=([3]=
blah [0]=bork [1]=blarg [2]=zooj); typeset -p a; }; f' # ksh93 f
orces appending to the array, disregarding subscripts
typeset -a a=(foo bar baz '[3]=blah' '[0]=bork' '[1]=blarg' '[2]
=zooj')
 $ bash -c 'function f { typeset -a a; a+=(foo bar baz); a+=(bla
h [0]=bork blarg zooj); typeset -p a; }; f' # Bash applies += to
every individual subscript.
declare -a a='([0]="foobork" [1]="barblarg" [2]="bazzooj" [3]="b
lah")'
 $ mksh -c 'function f { typeset -a a; a+=(foo bar baz); a+=(bla
h [0]=bork blarg zooj); typeset -p a; }; f' # Mksh does like Bas
h, but clobbers previous values rather than appending.
set -A a
typeset a[0]=bork
typeset a[1]=blarg
typeset a[2]=zooj
typeset a[3]=blah
```

- In Bash and Zsh, the alternate value assignment parameter expansion (`${arr[idx]:=foo}`) evaluates the subscript twice, first to determine whether to expand the alternate, and second to determine the index to assign the alternate to. See evaluation order.

```
 $ : ${_[$(echo $RANDOM >&2)1]:=$(echo hi >&2)}
13574
hi
14485
```

- In Zsh, arrays are indexed starting at 1 in its default mode. Emulation modes are required in order to get any kind of portability.
- Zsh and mksh do not support compound assignment arguments to `typeset`.
- Ksh88 didn't support modern compound array assignment syntax. The original (and most portable) way to assign multiple elements is to use the `set -A name arg1 arg2 …` syntax. This is supported by almost all shells that support ksh-like arrays except for Bash. Additionally, these shells usually support an optional `-s` argument to `set` which performs lexicographic sorting on either array elements or the positional parameters. Bash has no built-in sorting ability other than the usual comparison operators.

```
 $ ksh -c 'set -A arr -- foo bar bork baz; typeset -p arr' # Cla
ssic array assignment syntax
typeset -a arr=(foo bar bork baz)
 $ ksh -c 'set -sA arr -- foo bar bork baz; typeset -p arr' # Na
tive sorting!
typeset -a arr=(bar baz bork foo)
 $ mksh -c 'set -sA arr -- foo "[3]=bar" "[2]=baz" "[7]=bork"; t
ypeset -p arr' # Probably a bug. I think the maintainer is aware
of it.
set -A arr
typeset arr[2]=baz
typeset arr[3]=bar
typeset arr[7]=bork
typeset arr[8]=foo
```

- Evaluation order for assignments involving arrays varies significantly depending on context. Notably, the order of evaluating the subscript or the value first can change in almost every shell for both expansions and arithmetic variables. See evaluation order for details.
- Bash 4.1.* and below cannot use negative subscripts to address array indexes relative to the highest-numbered index. You must use the subscript expansion, i.e. `"${arr[@]:(-n):1}"`, to expand the nth-last element (or the next-highest indexed after `n` if `arr[n]` is unset). In Bash 4.2, you may expand (but not assign to) a negative index. In Bash 4.3, ksh93, and zsh, you may both assign and expand negative offsets.
- ksh93 also has an additional slice notation: `"${arr[n..m]}"` where `n` and `m` are arithmetic expressions. These are needed for use with multi-dimensional arrays.
- Assigning or referencing negative indexes in mksh causes wrap-around. The max index appears to be `UINT_MAX`, which would be addressed by `arr[-1]`.
- So far, Bash's `-v var` test doesn't support individual array subscripts. You may supply an array name to test whether an array is defined, but can't check an element. ksh93's `-v` supports both. Other shells lack a `-v` test.

Bugs

- **Fixed in 4.3** Bash 4.2.* and earlier considers each chunk of a compound assignment, including the subscript for globbing. The subscript part is considered quoted, but any unquoted glob characters on the right-hand side of the `[…]=` will be clumped with the subscript and counted as a glob. Therefore, you must quote anything on the right of the `=` sign. This is fixed in 4.3, so that each subscript assignment statement is expanded following the same rules as an ordinary assignment. This also works correctly in ksh93.

```
$ touch '[1]=a'; bash -c 'a=([1]=*); echo "${a[@]}"'
[1]=a
```

  mksh has a similar but even worse problem in that the entire subscript is considered a glob.

```
$ touch 1=a; mksh -c 'a=([123]=*); print -r -- "${a[@]}"'
1=a
```

- **Fixed in 4.3** In addition to the above globbing issue, assignments preceding "declare" have an additional effect on brace and pathname expansion.

```
$ set -x; foo=bar declare arr=( {1..10} )
+ foo=bar
+ declare 'arr=(1)' 'arr=(2)' 'arr=(3)' 'arr=(4)' 'arr=(5)' 'arr
=(6)' 'arr=(7)' 'arr=(8)' 'arr=(9)' 'arr=(10)'

$ touch xy=foo
+ touch xy=foo
$ declare x[y]=*
+ declare 'x[y]=*'
$ foo=bar declare x[y]=*
+ foo=bar
+ declare xy=foo
```

  Each word (the entire assignment) is subject to globbing and brace expansion. This appears to trigger the same strange expansion mode as `let`, `eval`, other declaration commands, and maybe more.
- **Fixed in 4.3** Indirection combined with another modifier expands arrays to a single word.

```
$ a=({a..c}) b=a[@]; printf '<%s> ' "${!b}"; echo; printf '<%s>
' "${!b/%/foo}"; echo
<a> <b> <c>
<a b cfoo>
```

- **Fixed in 4.3** Process substitutions are evaluated within array indexes. Zsh and ksh don't do this in any arithmetic context.

```
# print "moo"
dev=fd=1 _[1<(echo moo >&2)]=

# Fork bomb
${dev[${dev='dev[1>(${dev[dev]})]'}]}
```

Evaluation order

Here are some of the nasty details of array assignment evaluation order. You can use this testcase code (https://gist.github.com/ormaaj/4942297) to generate these results.

```
Each testcase prints evaluation order for indexed array assignment
contexts. Each context is tested for expansions (represented by digit
s) and
arithmetic (letters), ordered from left to right within the expressio
n. The
output corresponds to the way evaluation is re-ordered for each shel
l:

a[ $1 a ]=${b[ $2 b ]:=${c[ $3 c ]}}              No attributes
a[ $1 a ]=${b[ $2 b ]:=c[ $3 c ]}                 typeset -ia a
a[ $1 a ]=${b[ $2 b ]:=c[ $3 c ]}                 typeset -ia b
a[ $1 a ]=${b[ $2 b ]:=c[ $3 c ]}                 typeset -ia a b
(( a[ $1 a ] = b[ $2 b ] ${c[ $3 c ]} ))          No attributes
(( a[ $1 a ] = ${b[ $2 b ]:=c[ $3 c ]} ))         typeset -ia b
a+=( [ $1 a ]=${b[ $2 b ]:=${c[ $3 c ]}} [ $4 d ]=$(( $5 e )) ) types
et -a a
a+=( [ $1 a ]=${b[ $2 b ]:=c[ $3 c ]} [ $4 d ]=${5}e ) typeset -ia a

bash: 4.2.42(1)-release
2 b 3 c 2 b 1 a
2 b 3 2 b 1 a c
2 b 3 2 b c 1 a
2 b 3 2 b c 1 a c
1 2 3 c b a
1 2 b 3 2 b c c a
1 2 b 3 c 2 b 4 5 e a d
1 2 b 3 2 b 4 5 a c d e

ksh93: Version AJM 93v- 2013-02-22
1 2 b b a
1 2 b b a
1 2 b b a
1 2 b b a
1 2 3 c b a
1 2 b b a
1 2 b b a 4 5 e d
1 2 b b a 4 5 d e

mksh: @(#)MIRBSD KSH R44 2013/02/24
2 b 3 c 1 a
2 b 3 1 a c
2 b 3 c 1 a
2 b 3 c 1 a
1 2 3 c a b
1 2 b 3 c a
1 2 b 3 c 4 5 e a d
1 2 b 3 4 5 a c d e

zsh: 5.0.2
2 b 3 c 2 b 1 a
2 b 3 2 b 1 a c
2 b 1 a
2 b 1 a
1 2 3 c b a
1 2 b a
```

```
1 2 b 3 c 2 b 4 5 e
1 2 b 3 2 b 4 5
```

See also

- Parameter expansion (contains sections for arrays)
- The classic for-loop (contains some examples to iterate over arrays)
- The declare builtin command
- BashFAQ 005 - How can I use array variables? (http://mywiki.wooledge.org/BashFAQ/005) - A very detailed discussion on arrays with many examples.
- BashSheet - Arrays (http://mywiki.wooledge.org/BashSheet#Arrays) - Bashsheet quick-reference on Greycat's wiki.

Discussion

captainmish, 2011/05/26 10:16 ()

Handy for "slicing" - with

```
${ARRAY[*]:N:M}
```

the M is implied if omitted, so eg

```
${ARRAY[*]:5}
```

will expand to elements 6 to {end of array}

zik, 2011/06/13 17:24 ()

It isn't necessary to subtract 1 from the number of elements. Just change the termination condition to '<'.

```
N_ELEMENTS=${#SENTENCE[@]}

for ((i = 0; i < N_ELEMENTS; i++)); do    # the increment is i pl
us plus but preview isn't showing that way.
  echo "Element $i: '${SENTENCE[i]}'"
done
```

Altair IV, 2012/01/12 13:25 ()

You can safely loop through sparse and associative arrays using the ${!array[@]} expansion pattern.

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

Bash 4.2+ also has negative array indexing. You can expand the second-to-last element with ${array[-2]}, for example, and even apply a parameter expansion to it at the same time!

Yclept Nemo, 2012/11/29 03:43 ()

The isSubset function in the Indirection section could really use some explanation, particularly:

[1]

```
local -a 'xkeys=("${!'"$1"'[@]}")' 'ykeys=("${!'"$2"'[@]}")'
```

I couldn't find any examples of similar usage of local/declare anywhere else. The BashFAQ[1] states, "the right hand side of the assignment is not parsed by the shell". Yet clearly - and also in the following example - some level of parsing is performed.

```
local -a 'a=({0..5})'
```

While I tested myself and for the right-hand side of declare/eval could not produce any unwanted side effects or arbitrary code execution, I can't guarantee that in this form either function is safe. Furthermore the behavior that allows these functions to dereference indirect variable references seems undocumented, and only to work with the array option. The following won't cause any expansion:

```
local 'xkeys="${!'"$1"'[@]}"'
```

[2]

```
set -- "${@/%/[key]}"
```

Sets the positional parameters to the value of the parameter expansion.

[3]

```
[[ ${!2+_} && ${!1} == ${!2} ]] || return 1
```

In reference to ${name+_}, quoting the bash manual: "Omitting the colon results in a test only for a parameter that is unset. Put another way, if the colon is included, the operator tests for both parameter’s existence and that its value is not null; if the colon is omitted, the operator tests only for existence. "

[1] http://mywiki.wooledge.org/BashFAQ/006#Indirection (http://mywiki.wooledge.org/BashFAQ/006#Indirection)