

# Parameter expansion

## Introduction

One core functionality of Bash is to manage **parameters**. A parameter is an entity that stores values and is referenced by a **name**, a **number** or a **special symbol**.

- parameters referenced by a name are called **variables** (this also applies to arrays)
- parameters referenced by a number are called **positional parameters** and reflect the arguments given to a shell
- parameters referenced by a **special symbol** are auto-set parameters that have different special meanings and uses

**Parameter expansion** is the procedure to get the value from the referenced entity, like expanding a variable to print its value. On expansion time you can do very nasty things with the parameter or its value. These things are described here.

**If you saw** some parameter expansion syntax somewhere, and need to check what it can be, try the overview section below!

**Arrays** can be special cases for parameter expansion, every applicable description mentions arrays below. Please also see the article about arrays.

For a more technical view what a parameter is and which types exist, see the dictionary entry for "parameter".

## Overview

Looking for a specific syntax you saw, without knowing the name?

- Simple usage
  - `$PARAMETER`
  - `${PARAMETER}`
- Indirection
  - `${!PARAMETER}`
- Case modification
  - `${PARAMETER^}`
  - `${PARAMETER^^}`
  - `${PARAMETER,}`
  - `${PARAMETER,,}`
  - `${PARAMETER~}`
  - `${PARAMETER~~}`
- Variable name expansion

- `${!PREFIX*}`
- `${!PREFIX@}`
- Substring removal (also for **filename manipulation!**)
  - `${PARAMETER#PATTERN}`
  - `${PARAMETER##PATTERN}`
  - `${PARAMETER%PATTERN}`
  - `${PARAMETER%%PATTERN}`
- Search and replace
  - `${PARAMETER/PATTERN/STRING}`
  - `${PARAMETER//PATTERN/STRING}`
  - `${PARAMETER/PATTERN}`
  - `${PARAMETER//PATTERN}`
- String length
  - `${#PARAMETER}`
- Substring expansion
  - `${PARAMETER:OFFSET}`
  - `${PARAMETER:OFFSET:LENGTH}`
- Use a default value
  - `${PARAMETER:-WORD}`
  - `${PARAMETER-WORD}`
- Assign a default value
  - `${PARAMETER:=WORD}`
  - `${PARAMETER=WORD}`
- Use an alternate value
  - `${PARAMETER:+WORD}`
  - `${PARAMETER+WORD}`
- Display error if null or unset
  - `${PARAMETER:?WORD}`
  - `${PARAMETER?WORD}`

## Simple usage

`$PARAMETER`

`${PARAMETER}`

The easiest form is to just use a parameter's name within braces. This is identical to using `$FOO` like you see it everywhere, but has the advantage that it can be immediately followed by characters that would be interpreted as part of the parameter name otherwise. Compare these two expressions ( `WORD="car"` for example), where we want to print a word with a trailing "s":

```
echo "The plural of $WORD is most likely $WORDS"
echo "The plural of $WORD is most likely ${WORD}s"
```

Why does the first one fail? It prints nothing, because a parameter (variable) named " `WORDS` " is undefined and thus printed as "" (*nothing*). Without using braces for parameter expansion, Bash will interpret the sequence of all valid characters from the introducing " `$` "

up to the last valid character as name of the parameter. When using braces you just force Bash to **only interpret the name inside your braces**.

Also, please remember, that **parameter names are** (like nearly everything in UNIX®) **case sensitive!**

The second form with the curly braces is also needed to access positional parameters (arguments to a script) beyond \$9 :

```
echo "Argument 1 is: $1"
echo "Argument 10 is: ${10}"
```

## Simple usage: Arrays

See also the article about general array syntax

For arrays you always need the braces. The arrays are expanded by individual indexes or mass arguments. An individual index behaves like a normal parameter, for the mass expansion, please read the article about arrays linked above.

- \${array[5]}
- \${array[\*]}
- \${array[@]}

## Indirection

`${!PARAMETER}`

In some cases, like for example

```
${PARAMETER}

${PARAMETER:0:3}
```

you can instead use the form

```
${!PARAMETER}
```

to enter a level of indirection. The referenced parameter is not `PARAMETER` itself, but the parameter whose name is stored as the value of `PARAMETER`. If the parameter `PARAMETER` has the value "TEMP", then `${!PARAMETER}` will expand to the value of the parameter named `TEMP` :

```
read -rep 'Which variable do you want to inspect? ' look_var

printf 'The value of "%s" is: "%s"\n' "$look_var" "${!look_var}"
```

Of course the indirection also works with special variables:

```
# set some fake positional parameters
set one two three four

# get the LAST argument ("#" stores the number of arguments, so "!"#
will reference the LAST argument)
echo ${!#}
```

You can think of this mechanism as being roughly equivalent to taking any parameter expansion that begins with the parameter name, and substituting the `!PARAMETER` part with the value of `PARAMETER`.

```
echo "${!var^^}"
# ...is equivalent to
eval 'echo "${!$var}^^"'
```

It was an unfortunate design decision to use the `!` prefix for indirection, as it introduces parsing ambiguity with other parameter expansions that begin with `!`. Indirection is not possible in combination with any parameter expansion whose modifier requires a prefix to the parameter name. Specifically, indirection isn't possible on the `${!var@}`, `${!var*}`, `${!var[@]}`, `${!var[*]}`, and `${#var}` forms. This means the `!` prefix can't be used to retrieve the indices of an array, the length of a string, or number of elements in an array indirectly (see [indirection](#) for workarounds). Additionally, the `!`-prefixed parameter expansion conflicts with ksh-like shells which have the more powerful "name-reference" form of indirection, where the exact same syntax is used to expand to the name of the variable being referenced.

Indirect references to array names are also possible since the Bash 3 series (exact version unknown), but undocumented. See [indirection](#) for details.

Chet has added an initial implementation of the ksh `nameref` declaration command to the `git devel` branch. (`declare -n`, `local -n`, etc, will be supported). This will finally address many issues around passing and returning complex datatypes to/from functions.

## Case modification

```
${PARAMETER^}

${PARAMETER^^}

${PARAMETER,}

${PARAMETER,,}

${PARAMETER~}

${PARAMETER~~}
```

These expansion operators modify the case of the letters in the expanded text.

The `^` operator modifies the first character to uppercase, the `,` operator to lowercase. When using the double-form (`^^` and `,,`), all characters are converted.

The (**currently undocumented**) operators `~` and `~~` reverse the case of the given text (in `PARAMETER`). `~` reverses the case of first letter of words in the variable while `~~` reverses case for all. Thanks to Bushmills and geirha on the Freenode [IRC\(\)](#) channel for this finding.

### Example: Rename all \*.txt filenames to lowercase

```
for file in *.txt; do
    mv "$file" "${file,,}"
done
```

**Note:** Case modification is a handy feature you can apply to a name or a title. Or is it? Case modification was an important aspect of the Bash 4 release. Bash version 4, RC1 would perform word splitting, and then case modification, resulting in title case (where every word is capitalized). It was decided to apply case modification to values, not words, for the Bash 4 release. Thanks Chet.

## Case modification: Arrays

Case modification can be used to create the proper capitalization for names or titles. Just assign it to an array:

```
declare -a title=(my hello world john smith)
```

For array expansion, the case modification applies to **every expanded element, no matter if you expand an individual index or mass-expand** the whole array using `@` or `*` subscripts. Some examples:

Assume: `array=(This is some Text)`

- `echo "${array[@],}"`
  - `⇒ this is some text`
- `echo "${array[@],,}"`
  - `⇒ this is some text`
- `echo "${array[@]^}"`
  - `⇒ This Is Some Text`
- `echo "${array[@]^^}"`
  - `⇒ THIS IS SOME TEXT`
- `echo "${array[2]^^}"`
  - `⇒ SOME`

## Variable name expansion

```
${!PREFIX*}
```

```
${!PREFIX@}
```

This expands to a list of all set **variable names** beginning with the string `PREFIX` . The elements of the list are separated by the first character in the `IFS` -variable (<space> by default).

This will show all defined variable names (not values!) beginning with "BASH":

```
$ echo ${!BASH*}
BASH BASH_ARGC BASH_ARGV BASH_COMMAND BASH_LINENO BASH_SOURCE BASH_SU
BSHELL BASH_VERSINFO BASH_VERSION
```

This list will also include array names.

## Substring removal

`${PARAMETER#PATTERN}`

`${PARAMETER##PATTERN}`

`${PARAMETER%PATTERN}`

`${PARAMETER%%PATTERN}`

This one can **expand only a part** of a parameter's value, **given a pattern to describe what to remove** from the string. The pattern is interpreted just like a pattern to describe a filename to match (globbing). See Pattern matching for more.

Example string (*just a quote from a big man*):

```
MYSTRING="Be liberal in what you accept, and conservative in what you
send"
```

## From the beginning

`${PARAMETER#PATTERN}` and `${PARAMETER##PATTERN}`

This form is to remove the described pattern trying to **match it from the beginning of the string**. The operator `#` will try to remove the shortest text matching the pattern, while `##` tries to do it with the longest text matching. Look at the following examples to get the idea (matched text ~~marked struck~~, remember it will be removed!):

Syntax	Result
<code>\${MYSTRING#*in}</code>	<del>Be liberal in</del> what you accept, and conservative in what you send
<code>\${MYSTRING##*in}</code>	<del>Be liberal in what you accept, and conservative in</del> what you send

## From the end

`${PARAMETER%PATTERN}` and `${PARAMETER%%PATTERN}`

In the second form everything will be the same, except that Bash now tries to match the pattern from the end of the string:

Syntax	Result
<code>\${MYSTRING%in*}</code>	Be liberal in what you accept, and conservative <del>in what you send</del>
<code>\${MYSTRING%%in*}</code>	Be liberal <del>in what you accept, and conservative in what you send</del>

The second form nullifies variables that begin with `in` , by working from the end.

## Common use

### How the heck does that help to make my life easier?

Well, maybe the most common use for it is to **extract parts of a filename**. Just look at the following list with examples:

- **Get name without extension**
  - `${FILENAME%. *}`
  - $\Rightarrow$  `bash_hackers.txt`
- **Get extension**
  - `${FILENAME##*.*}`
  - $\Rightarrow$  `bash_hackers.txt`
- **Get directory name**
  - `${PATHNAME%/*}`
  - $\Rightarrow$  `/home/bash/bash_hackers.txt`
- **Get filename**
  - `${PATHNAME##*/}`
  - $\Rightarrow$  `/home/bash/bash_hackers.txt`

These are the syntaxes for filenames with a single extension. Depending on your needs, you might need to adjust `shortest/longest` match.

## Substring removal: Arrays

As for most parameter expansion features, working on arrays **will handle each expanded element**, for individual expansion and also for mass expansion.

Simple example, removing a trailing `is` from all array elements (on expansion):

Assume: `array=(This is a text)`

- `echo "${array[@]%is}"`
  - $\Rightarrow$  `Th a text`
  - (it was: `This is is a text` )

All other variants of this expansion behave the same.

## Search and replace

```
${PARAMETER/PATTERN/STRING}
```

```
${PARAMETER//PATTERN/STRING}
```

```
${PARAMETER/PATTERN}
```

```
${PARAMETER//PATTERN}
```

This one can substitute (*replace*) a substring matched by a pattern, on expansion time. The matched substring will be entirely removed and the given string will be inserted. Again some example string for the tests:

```
MYSTRING="Be liberal in what you accept, and conservative in what you send"
```

The two main forms only differ in **the number of slashes** after the parameter name:

```
${PARAMETER/PATTERN/STRING} and ${PARAMETER//PATTERN/STRING}
```

The first one (*one slash*) is to only substitute **the first occurrence** of the given pattern, the second one (*two slashes*) is to substitute **all occurrences** of the pattern.

First, let's try to say "happy" instead of "conservative" in our example string:

```
${MYSTRING//conservative/happy}
```

⇒ Be liberal in what you accept, and ~~conservative~~happy in what you send

Since there is only one "conservative" in that example, it really doesn't matter which of the two forms we use.

Let's play with the word "in", I don't know if it makes any sense, but let's substitute it with "by".

### First form: Substitute first occurrence

```
${MYSTRING/in/by}
```

⇒ Be liberal ~~in~~by what you accept, and conservative in what you send

### Second form: Substitute all occurrences

```
${MYSTRING//in/by}
```

⇒ Be liberal ~~in~~by what you accept, and conservative ~~in~~by what you send

**Anchoring** Additionally you can "anchor" an expression: A # (hashmark) will indicate that your expression is matched against the beginning portion of the string, a % (percent-sign) will do it for the end portion.

```
MYSTRING=xxxxxxxxxx
echo ${MYSTRING/#x/y} # RESULT: yxxxxxxxxxx
echo ${MYSTRING/%x/y} # RESULT: xxxxxxxxxxxy
```

If the replacement part is completely omitted, the matches are replaced by the nullstring, i.e., they are removed. This is equivalent to specifying an empty replacement:



```
echo ${MYSTRING//conservative/}
# is equivalent to
echo ${MYSTRING//conservative}
```

## Search and replace: Arrays

This parameter expansion type applied to arrays **applies to all expanded elements**, no matter if an individual element is expanded, or all elements using the mass expansion syntaxes.

A simple example, changing the (lowercase) letter `t` to `d`:

Assume: `array=(This is a text)`

- `echo "${array[@]/t/d}"`
  - `⇒ This is a dext`
- `echo "${array[@]//t/d}"`
  - `⇒ This is a dexd`

## String length

`${#PARAMETER}`

When you use this form, the length of the parameter's value is expanded. Again, a quote from a big man, to have a test text:

```
MYSTRING="Be liberal in what you accept, and conservative in what you send"
```

Using `echo ${#MYSTRING} ...`

`⇒ 64`

The length is reported in characters, not in bytes. Depending on your environment this may not always be the same (multibyte-characters, like in UTF8 encoding).

There's not much to say about it, mh?

## (String) length: Arrays

For arrays, this expansion type has two meanings:

- For **individual** elements, it reports the string length of the element (as for every "normal" parameter)
- For the **mass subscripts** `@` and `*` it reports the number of set elements in the array

Example:

Assume: `array=(This is a text)`

- `echo ${#array[1]}`

- ⇒ 2 (the word "is" has a length of 2)
- `echo ${#array[@]}`
  - ⇒ 4 (the array contains 4 elements)

**Attention:** The number of used elements does not need to conform to the highest index. Sparse arrays are possible in Bash, that means you can have 4 elements, but with indexes 1, 7, 20, 31. **You can't loop through such an array with a counter loop based on the number of elements!**

## Substring expansion

`${PARAMETER:OFFSET}`

`${PARAMETER:OFFSET:LENGTH}`

This one can expand only a **part** of a parameter's value, given a **position to start** and maybe a **length**. If `LENGTH` is omitted, the parameter will be expanded up to the end of the string. If `LENGTH` is negative, it's taken as a second offset into the string, counting from the end of the string.

`OFFSET` and `LENGTH` can be **any** arithmetic expression. **Take care:** The `OFFSET` starts at 0, not at 1!

Example string (a quote from a big man): `MYSTRING="Be liberal in what you accept, and conservative in what you send"`

## Using only Offset

In the first form, the expansion is used without a length value, note that the offset 0 is the first character:

```
echo ${MYSTRING:35}
```

⇒ ~~Be liberal in what you accept, and~~ conservative in what you send

## Using Offset and Length

In the second form we also give a length value:

```
echo ${MYSTRING:35:12}
```

⇒ ~~Be liberal in what you accept, and conservative in what you send~~

## Negative Offset Value

If the given offset is negative, it's counted from the end of the string, i.e. an offset of -1 is the last character. In that case, the length still counts forward, of course. One special thing is to do when using a negative offset: You need to separate the (negative) number from

the colon:

```
${MYSTRING: -10:5}
${MYSTRING:(-10):5}
```

Why? Because it's interpreted as the parameter expansion syntax to **use a default value**.

## Negative Length Value

If the `LENGTH` value is negative, it's used as offset from the end of the string. The expansion happens from the first to the second offset then:

```
echo "${MYSTRING:11:-17}"
```

⇒ ~~Be liberal~~ in what you accept, and conservative ~~in what you send~~

This works since Bash 4.2-alpha, see also Bash changes.

## Substring/Element expansion: Arrays

For arrays, this expansion type has again 2 meanings:

- For **individual** elements, it expands to the specified substring (as for every “normal” parameter)
- For the **mass subscripts** `@` and `*` it mass-expands individual array elements denoted by the 2 numbers given (*starting element, number of elements*)

Example:

Assume: `array=(This is a text)`

- `echo ${array[0]:2:2}`
  - ⇒ `is` (the “is” in “This”, array element 0)
- `echo ${array[@]:1:2}`
  - ⇒ `is a` (from element 1 inclusive, 2 elements are expanded, i.e. element 1 and 2)

## Use a default value

```
${PARAMETER:-WORD}
```

```
${PARAMETER-WORD}
```

If the parameter `PARAMETER` is unset (never was defined) or null (empty), this one expands to `WORD`, otherwise it expands to the value of `PARAMETER`, as if it just was `${PARAMETER}`. If you omit the `:` (colon), like shown in the second form, the default value is only used when the parameter was **unset**, not when it was empty.

```
echo "Your home directory is: ${HOME:-/home/$USER}."
echo "${HOME:-/home/$USER} will be used to store your personal data."
```

If `HOME` is unset or empty, everytime you want to print something useful, you need to put that parameter syntax in.

```
#!/bin/bash

read -p "Enter your gender (just press ENTER to not tell us): " GENDER
echo "Your gender is ${GENDER:-a secret}."
```

It will print "Your gender is a secret." when you don't enter the gender. Note that the default value is **used on expansion time**, it is **not assigned to the parameter**.

## Use a default value: Arrays

For arrays, the behaviour is very similar. Again, you have to make a difference between expanding an individual element by a given index and mass-expanding the array using the `@` and `*` subscripts.

- For individual elements, it's the very same: If the expanded element is `NULL` or unset (watch the `: -` and `-` variants), the default text is expanded
- For mass-expansion syntax, the default text is expanded if the array
  - contains no element or is unset (the `: -` and `-` variants mean the **same** here)
  - contains only elements that are the nullstring (the `: -` variant)

In other words: The basic meaning of this expansion type is applied as consistent as possible to arrays.

Example code (please try the example cases yourself):

```
####
# Example cases for unset/empty arrays and nullstring elements
####

### CASE 1: Unset array (no array)

# make sure we have no array at all
unset array

echo ${array[@]:-This array is NULL or unset}
echo ${array[@]-This array is NULL or unset}

### CASE 2: Set but empty array (no elements)

# declare an empty array
array=()

echo ${array[@]:-This array is NULL or unset}
echo ${array[@]-This array is NULL or unset}

### CASE 3: An array with only one element, a nullstring
array=("")

echo ${array[@]:-This array is NULL or unset}
echo ${array[@]-This array is NULL or unset}

### CASE 4: An array with only two elements, a nullstring and a normal word
array=("" word)

echo ${array[@]:-This array is NULL or unset}
echo ${array[@]-This array is NULL or unset}
```

## Assign a default value

```
${PARAMETER:=WORD}
```

```
${PARAMETER=WORD}
```

This one works like the **using default values**, but the default text you give is not only expanded, but also **assigned** to the parameter, if it was unset or null. Equivalent to using a default value, when you omit the `:` (colon), as shown in the second form, the default value will only be assigned when the parameter was **unset**.

```
echo "Your home directory is: ${HOME:=/home/$USER}."
echo "$HOME will be used to store your personal data."
```

After the first expansion here ( `${HOME:=/home/$USER}` ), `HOME` is set and usable.

Let's change our code example from above:

```
#!/bin/bash

read -p "Enter your gender (just press ENTER to not tell us): " GENDER
echo "Your gender is ${GENDER:=a secret}."
echo "Ah, in case you forgot, your gender is really: $GENDER"
```

## Assign a default value: Arrays

For arrays this expansion type is limited. For an individual index, it behaves like for a "normal" parameter, the default value is assigned to this one element. The mass-expansion subscripts `@` and `*` **can not be used here** because it's not possible to assign to them!

## Use an alternate value

```
${PARAMETER:+WORD}
```

```
${PARAMETER+WORD}
```

This form expands to nothing if the parameter is unset or empty. If it is set, it does not expand to the parameter's value, **but to some text you can specify**:

```
echo "The Java application was installed and can be started.${JAVAPATH:+ NOTE: JAVAPATH seems to be set}"
```

The above code will simply add a warning if `JAVAPATH` is set (because it could influence the startup behaviour of that imaginary application).

Some more unrealistic example... Ask for some flags (for whatever reason), and then, if they were set, print a warning and also print the flags:

```
#!/bin/bash

read -p "If you want to use special flags, enter them now: " SPECIAL_FLAGS
echo "The installation of the application is finished${SPECIAL_FLAGS:+ (NOTE: there are special flags set: $SPECIAL_FLAGS)}."
```

If you omit the colon, as shown in the second form ( `${PARAMETER+WORD}` ), the alternate value will be used if the parameter is set (and it can be empty)! You can use it, for example, to complain if variables you need (and that can be empty) are undefined:

```
# test that with the three stages:

# unset foo
# foo=""
# foo="something"

if [[ ${foo+isset} = isset ]]; then
    echo "foo is set..."
else
    echo "foo is not set..."
fi
```

## Use an alternate value: Arrays

Similar to the cases for arrays to expand to a default value, this expansion behaves like for a "normal" parameter when using individual array elements by index, but reacts differently when using the mass-expansion subscripts `@` and `*`:

- For individual elements, it's the very same: If the expanded element is **not** NULL or unset (watch the `:+` and `+` variants), the alternate text is expanded
- For mass-expansion syntax, the alternate text is expanded if the array
  - contains elements where min. one element is **not** a nullstring (the `:+` and `+` variants mean the same here)
  - contains **only** elements that are **not** the nullstring (the `:+` variant)

For some cases to play with, please see the code examples in the description for using a default value.

## Display error if null or unset

```
${PARAMETER:?WORD}
```

```
${PARAMETER?WORD}
```

If the parameter `PARAMETER` is set/non-null, this form will simply expand it. Otherwise, the expansion of `WORD` will be used as appendix for an error message:

```
$ echo "The unset parameter is: ${p_unset?not set}"
bash: p_unset: not set
```

After printing this message,

- an interactive shell has `$?` to a non-zero value
- a non-interactive shell exits with a non-zero exit code

The meaning of the colon ( `:` ) is the same as for the other parameter expansion syntaxes: It specifies if

- only unset or
- unset and empty parameters

are taken into account.

# Code examples

## Substring removal

Removing the first 6 characters from a text string:

```
STRING="Hello world"

# only print 'Hello'
echo "${STRING%?????}"

# only print 'world'
echo "${STRING#?????}"

# store it into the same variable
STRING=${STRING#?????}
```

## Bugs and Portability considerations

- **Fixed in 4.2.36** ( patch (<ftp://ftp.cwru.edu/pub/bash/bash-4.2-patches/bash42-036>)). Bash doesn't follow either POSIX or its own documentation when expanding either a quoted "\$@" or "\${arr[@]}" with an adjacent expansion. "\$@\$x" expands in the same way as "\$\*\$x" - i.e. all parameters plus the adjacent expansion are concatenated into a single argument. As a workaround, each expansion needs to be quoted separately. Unfortunately, this bug took a very long time to notice.

```
~ $ set -- a b c; x=foo; printf '<%s> ' "$@$x" "$*"$x" "$@"$x"
<a b cfoo> <a b cfoo> <a> <b> <cfoo>
```

- Almost all shells disagree about the treatment of an unquoted \$@ , \${arr[@]} , \$\* , and \${arr[\*]} when IFS (<http://mywiki.woledge.org/IFS>) is set to null. POSIX is unclear about the expected behavior. A null IFS causes both word splitting and pathname expansion to behave randomly. Since there are few good reasons to leave IFS set to null for more than the duration of a command or two, and even fewer to expand \$@ and \$\* unquoted, this should be a rare issue. **Always quote them!**



```
touch x 'y z'
for sh in bb {{d,b}a,{m,}k,z}sh; do
    echo "$sh"
    "$sh" -s a 'b c' d \* </dev/fd/0
done <<\EOF
${ZSH_VERSION+:} false && emulate sh
IFS=
printf '<%s> ' $*
echo
printf "<%s> " $@
echo
EOF
```

```
bb
<ab cd*>
<ab cd*>
dash
<ab cd*>
<ab cd*>
bash
<a> <b c> <d> <x> <y z>
<a> <b c> <d> <x> <y z>
mksh
<a b c d *>
<a b c d *>
ksh
<a> <b c> <d> <x> <y z>
<a> <b c> <d> <x> <y z>
zsh
<a> <b c> <d> <x> <y z>
<a> <b c> <d> <x> <y z>
```

When `IFS` is set to a non-null value, or unset, all shells behave the same - first expanding into separate args, then applying pathname expansion and word-splitting to the results, except for `zsh`, which doesn't do pathname expansion in its default mode.

- Additionally, shells disagree about various wordsplitting behaviors, the behavior of inserting delimiter characters from `IFS` in `$*`, and the way adjacent arguments are concatenated, when `IFS` is modified in the middle of expansion through side-effects.

```
for sh in bb {{d,b}a,po,{m,}k,z}sh; do
    printf '%-4s: ' "$sh"
    "$sh" </dev/fd/0
done <<\EOF
${ZSH_VERSION+:} false && emulate sh
set -f -- a b c
unset -v IFS
printf '<%s> ' ${*}${IFS=${*}}${IFS:= -}"${*}"
echo
EOF
```

```

bb   : <a b cab< <a-b-c>
dash: <a b cab< <a-b-c>
bash: <a> <b> <ca> <b> <c-a b c>
posh: <a> <b> <ca b c> <a-b-c>
mksh: <a> <b> <ca b c> <a-b-c>
ksh  : <a> <b> <ca> <b> <c> <a b c>
zsh  : <a> <b> <ca> <b> <c> <a-b-c>

```

ksh93 and mksh can additionally achieve this side effect (and others) via the `${cmds;}` expansion. I haven't yet tested every possible side-effect that can affect expansion halfway through expansion that way.

- As previously mentioned, the Bash form of indirection by prefixing a parameter expansion with a `!` conflicts with the same syntax used by mksh, zsh, and ksh93 for a different purpose. Bash will "slightly" modify this expansion in the next version with the addition of namerefs.
- Bash (and most other shells) don't allow `.`'s in identifiers. In ksh93, dots in variable names are used to reference methods (i.e. "Discipline Functions"), attributes, special shell variables, and to define the "real value" of an instance of a class.
- In ksh93, the `_` parameter has even more uses. It is used in the same way as `self` in some object-oriented languages; as a placeholder for some data local to a class; and also as the mechanism for class inheritance. In most other contexts, `_` is compatible with Bash.
- Bash only evaluates the subscripts of the slice expansion (`${x:y:z}`) if the parameter is set (for both nested expansions and arithmetic). For ranges, Bash evaluates as little as possible, i.e., if the first part is out of range, the second won't be evaluated. ksh93 and mksh always evaluate the subscript parts even if the parameter is unset.

```

$ bash -c 'n="y[\$(printf yo >&2)1]" m="y[\$(printf jo >&2)1]";
x=(); echo "${x[@]:n,6:m}"' # No output
$ bash -c 'n="y[\$(printf yo >&2)1]" m="y[\$(printf jo >&2)1]";
x=[5]=hi); echo "${x[@]:n,6:m}"'
yo
$ bash -c 'n="y[\$(printf yo >&2)1]" m="y[\$(printf jo >&2)1]";
x=[6]=hi); echo "${x[@]:n,6:m}"'
yojo
$ bash -c 'n="y[\$(printf yo >&2)1]" m="y[\$(printf jo >&2)1]";
x=12345; echo "${x:n,5:m}"'
yojo
$ bash -c 'n="y[\$(printf yo >&2)1]" m="y[\$(printf jo >&2)1]";
x=12345; echo "${x:n,6:m}"'
yo

```

## Quote Nesting

- In most shells, when dealing with an "alternate" parameter expansion that expands to multiple words, and nesting such expansions, not all combinations of nested quoting are possible.

```
# Bash
$ typeset -a a=(meh bleh blerg) b
$ IFS=e
$ printf "<%s> " "${b[@]}-${a[@]}" "${a[@]}"; echo # The entire PE
is quoted so Bash considers the inner quotes redundant.
<meh> <bleh> <blerg meh> <bleh> <blerg>
$ printf "<%s> " "${b[@]}-${a[@]} ${a[@]}"; echo # The outer quotes
cause the inner expansions to be considered quoted.
<meh> <bleh> <blerg meh> <bleh> <blerg>
$ b=(meep beep)
$ printf "<%s> " "${b[@]}-${a[@]}" "${a[@]}" "${b[@]}-${a[@]} ${a
[@]}"; echo # Again no surprises. Outer quotes quote everything recu
rsively.
<meep> <beep> <meep> <beep>
```

Now lets see what can happen if we leave the outside unquoted.

```
# Bash
$ typeset -a a=(meh bleh blerg) b
$ IFS=e
$ printf "<%s> " "${b[@]}-${a[@]}" "${a[@]}"; echo # Inner quotes ma
ke inner expansions quoted.
<meh> <bleh> <blerg meh> <bleh> <blerg>
$ printf "<%s> " "${b[@]}-${a[@]} ${a[@]}"; echo' # No quotes at all w
ordsplits / globs, like you'd expect.
<m> <h> <bl> <h> <bl> <rg m> <h> <bl> <h> <bl> <rg>
```

This all might be intuitive, and is the most common implementation, but this design sucks for a number of reasons. For one, it means Bash makes it absolutely impossible to expand any part of the inner region *unquoted* while leaving the outer region quoted. Quoting the outer forces quoting of the inner regions recursively (except nested command substitutions of course). Word-splitting is necessary to split words of the inner region, which cannot be done together with outer quoting. Consider the following (only slightly far-fetched) code:

```
# Bash (non-working example)

unset -v IFS # make sure we have a default IFS

if some crap; then
    typeset -a someCmd=(myCmd arg1 'arg2 yay!' 'third*arg*' 4)
fi

someOtherCmd=mycommand
typeset -a otherArgs=(arg3 arg4)

# What do you think the programmer expected to happen here?
# What do you think will actually happen...

"${someCmd[@]}-${someOtherCmd} arg2 "${otherArgs[@]}" arg5
```

This final line is perhaps not the most obvious, but I've run into cases where this type of logic can be desirable and realistic. We can deduce what was intended:

- If `someCmd` is set, then the resulting expansion should run the command: `"myCmd" "arg1" "arg2 yay!" "third*arg*" "4" "arg5"`
- Otherwise, if `someCmd` is not set, expand `$someOtherCmd` and the inner args, to run a different command: `"mycommand" "arg2" "arg3" "arg4" "arg5"`.

Unfortunately, it is impossible to get the intended result in Bash (and most other shells) without taking a considerably different approach. The only way to split the literal inner parts is through word-splitting, which requires that the PE be unquoted. But, the only way to expand the outer expansion correctly without word-splitting or globbing is to quote it. Bash will actually expand the command as one of these:

```
# The quoted PE produces a correct result here...
$ bash -c 'typeset -a someCmd=(myCmd arg1 "arg2 yay!" "third*arg*" 4); printf "<%s> " "${someCmd[@]}-"$someOtherCmd" arg2 "${otherArgs[@]}"' arg5; echo'
<myCmd> <arg1> <arg2 yay!> <third*arg*> <4> <arg5>

# ...but in the opposite case the first 3 arguments are glued together. There are no workarounds.
$ bash -c 'typeset -a otherArgs=(arg3 arg4); someOtherCmd=mycommand; printf "<%s> " "${someCmd[@]}-"$someOtherCmd" arg2 "${otherArgs[@]}"' arg5; echo'
<mycommand arg2 arg3> <arg4> <arg5>

# UNLESS! we unquote the outer expansion allowing the inner quotes to affect the necessary parts while allowing word-splitting to split the literals:
$ bash -c 'typeset -a otherArgs=(arg3 arg4); someOtherCmd=mycommand; printf "<%s> " "${someCmd[@]}-"$someOtherCmd" arg2 "${otherArgs[@]}"' arg5; echo'
<mycommand> <arg2> <arg3> <arg4> <arg5>

# Success!!!
$ bash -c 'typeset -a someCmd=(myCmd arg1 "arg2 yay!" "third*arg*" 4); printf "<%s> " "${someCmd[@]}-"$someOtherCmd" arg2 "${otherArgs[@]}"' arg5; echo'
<myCmd> <arg1> <arg2> <yay!> <third*arg*> <4> <arg5>

# ...Ah f^^k. (again, no workaround possible.)
```

## The ksh93 exception

To the best of my knowledge, ksh93 is the only shell that acts differently. Rather than forcing nested expansions into quoting, a quote at the beginning and end of the nested region will cause the quote state to reverse itself within the nested part. I have no idea whether it's an intentional or documented effect, but it does solve the problem and consequently adds a lot of potential power to these expansions.

All we need to do is add two extra double-quotes:

```
# ksh93 passing the two failed tests from above:

$ ksh -c 'otherArgs=(arg3 arg4); someOtherCmd="mycommand"; printf "
<%s> " "${someCmd[@]}-""$someOtherCmd" arg2 "${otherArgs[@]}"'"'"'" arg5;
echo '
<mycommand> <arg2> <arg3> <arg4> <arg5>

$ ksh -c 'typeset -a someCmd=(myCmd arg1 "arg2 yay!" "third*arg*"
4); printf "<%s> " "${someCmd[@]}-""$someOtherCmd" arg2 "${otherArgs
[@]}"'"'"'" arg5; echo '
<myCmd> <arg1> <arg2 yay!> <third*arg*> <4> <arg5>
```

This can be used to control the quote state of any part of any expansion to an arbitrary depth. Sadly, it is the only shell that does this and the difference may introduce a possible compatibility problem.

## See also

- Internal: Introduction to expansion and substitution
- Internal: Arrays
- Dictionary, internal: Parameter

## Discussion

Stan R., [2010/06/19 23:48\(\)](#)

Found an error under the "Search and replace" sub-heading, near the beginning of the "Anchoring" section, where there is an example snippet:

```
MYSTRING=xxxxxxxxxx
echo ${MYSTRING//#x/y} # RESULT: yxxxxxxxxx
echo ${MYSTRING//%x/y} # RESULT: xxxxxxxxxy
```

This should be:

```
MYSTRING=xxxxxxxxxx
echo ${MYSTRING/#x/y} # RESULT: yxxxxxxxxx
echo ${MYSTRING/%x/y} # RESULT: xxxxxxxxxy
```

The difference is MYSTRING/ instead of MYSTRING//, since you cannot have both anchoring and .

*Jan Schampera, [2010/06/20 12:56\(\)](#)*

Thanks for this finding. A late night typo, maybe :)

Stan R., [2010/06/21 01:11 \(\)](#)

Glad I could help.

Shmerl, [2010/10/20 16:37 \(\)](#)

I found an error. You write:

`${PARAMETER:-WORD} ${PARAMETER-WORD}` If the parameter `PARAMETER` is unset (never was defined) or null (empty), this one expands to `WORD`, otherwise it expands to the value of `PARAMETER`, as if it just was `${PARAMETER}`. **If you omit the `:` (colon), like shown in the second form, the default value is only used when the parameter was unset, not when it was empty.**

Running a simple test

```
str=""; echo ${str-test_result}
```

Produces:

**test\_result**

Now running:

```
unset str; echo ${str-test_result}
```

Produces nothing. So the above is just the opposite. It should be: If you omit the `:` (colon), like shown in the second form, the default value is only used when **the parameter was empty, not when it was unset.**

Jan Schampera, [2010/10/20 21:13 \(\)](#)

It works (and always worked) for me:

```
bonsai@core:~$ str=""; echo ${str-test_result}
```

```
bonsai@core:~$ unset str; echo ${str-test_result}
test_result
bonsai@core:~$
```

I'm not sure what could be the reason for your tests to do the exact opposite. Is this really a Bash? Though, this behaviour is specified by `POSIX(R)`, too: **"In the parameter expansions [shown previously], use of the `<colon>` in the format shall result in a test for a parameter that is unset or null; omission of the `<colon>` shall result in a test for a parameter that is only unset."**

Do you have any way to find out where your (or my) problem is here?

Shmerl, [2010/10/21 15:59 \(\)](#)

*Sorry, I really meant the case with **plus**.*

*Run this test:*

```
str=""; echo ${str+'test_result'}
```

*produces: **test\_result***

```
unset str; echo ${str+'test_result'}
```

*produces: (nothing)*

*I was wrong about other cases, this difference only applies to + (alternative value) case.*

Shmerl, [2010/10/21 16:03 \(\)](#)

*Make sure there is a plus there - the preview ate it.*

Jan Schampera, [2010/10/23 09:36 \(\)](#), [2010/10/23 09:37 \(\)](#)

*It does exactly what it should do. + produces an alternate value, means it expands to something **instead** of the real value of the parameter, if the parameter is "set" ( + ) or "set or null" ( :+ ).*

Shmerl, [2010/10/24 04:20 \(\)](#)

*Yes, but your text in the wiki above is misleading. It writes:*

*If you omit the colon, as shown in the second form  
(\${PARAMETER+WORD}), the alternate value will not be used when the  
parameter is empty, only if it is **unset**!*

*While it should say:*

*If you omit the colon, as shown in the second form  
(\${PARAMETER+WORD}), the alternate value will not be used when the  
parameter is **unset**, only if it is **empty**!*

Jan Schampera, [2010/10/24 06:09 \(\)](#)

*Oh, yes! Sorry, it took a while to see what you mean :(*

*I rephrased it a bit. If you're not okay with it, feel free to change it.  
Thanks for pointing it out.*

Shmerl, [2010/10/20 16:38 \(\)](#)

*Actually the previous note relates to all similar examples with :-, :=, etc.*

my, [2011/01/25 10:55 \(\)](#), [2011/02/01 05:31 \(\)](#)

*the `${ ^^ }` and friends have a pattern parameter. But it seems anything beyond `[ ]` does not work.*

```
TEST="abcABCxyz"
```

```
echo ${TEST^[abc]} # change first char if it is a||b||c
```

```
echo ${TEST^[!abc]} # change first char if it is NOT a||b||c
```

```
TEST="hello world"
```

```
echo ${TEST^^[aeiou]} # change ANY char that is in aeiou
```

```
AbcABCxyz
```

```
abcABCxyz
```

```
hEllO wOrld
```

Jan Champera, [2011/02/01 05:30 \(\)](#)

*I don't see what doesn't work here. Did you mean*

```
${TEST^[!abc]}
```

```
# (instead of ${TEST^[abc]})
```

*here?*

Frederick Grose, [2011/02/11 22:43 \(\)](#)

*In GNU bash, 4.1.7(1)-release (x86\_64-redhat-linux-gnu),  
the 'Negative Length Value' syntax,*

```
echo "${MYSTRING:11:-17}"
```

*results in the following:*

```
bash: -17: substring expression < 0
```

*LENGTH seems to behave like the array element COUNT.*

Jan Champera, [2011/02/12 08:44 \(\)](#)



*Hello Frederick,*

*you're absolutely right, this is a Bash 4.2 feature. I added a note.*

*Altair IV, [2012/01/12 13:02 \(\)](#)*

*Hmm? The `[b]${!array[*]}/[b]`/`[b]${!array[@]}/[b]` patterns aren't listed here? They print out a list of all existing array indexes, and can be used for looping through sparse and associative arrays.*

*Jan Schampera, [2012/06/20 21:24 \(\)](#)*

*This is listed in the arrays page, but might belong here, too, yes. It's a mixed subject, since all array-related expansion is parameter expansion.*

*David C, [2012/04/27 00:01 \(\)](#)*

*Under your Code Example of Substring Removal, you only remove 1 character per "?". Therefore the example needs to be updated to echo "\${STRING#???????}" (6 question marks). Another example for context would be echo "\${STRING%???????}" which will give you "Hello".*

*Jan Schampera, [2012/06/20 21:18 \(\)](#)*

*Late but fixed. Thank you.*

*R.W. Emerson II, [2012/12/17 09:58 \(\)](#), [2014/10/06 04:34 \(\)](#)*

*How do we determine whether one string contains another? How do we find the position of the embedded substring?*

*Substring removal offers a solution. If the substring is present, substring removal will change the length of the enclosing string. The `$tFind` position can be derived from the length difference.*

```

declare    tText="ABCDEFGG"
declare    tFind="CD"
declare    tTemp=${tText/"$tFind"/}  # $tFind present in $tText?
declare -i tFPos                      # Calculate $tFind position

if test "$tText" != "$tTemp" ; then
    echo "Found $tFind in $tText"
fi

tTemp=${tText#"$tFind"}              # "EFG" (Out of original
                                     # 7 characters, 3 remain so 7-3=4 were removed)
tFPos=${#tText}-${#tTemp}-${#tFind}  # 7-3-2=2 (Out of 4 removed,
                                     # $tFind accounts for 2, so 2 precede $tFind)
echo "Found $tFind at position $tFPos"  # 2

```

*It is necessary to quote \$tFind in the pattern field, to prevent extraneous effects when \$tFind contains brackets or other special pattern characters.*

*liungkejin, [2013/05/30 06:47 \(\)](#), [2014/10/06 04:36 \(\)](#)*

*only "\${!PREFIX\*}" is separated by the first character in the IFS -variable  
 \${!PREFIX\*} , \${!PREFIX@} and "\${!PREFIX@}" are separated by space*

```

testone=; testtwo=; testthree=;
IFS=":";
echo "${!test*}"
    => testone:testtwo:testthree
echo "${!test*}"
    => testone testtwo testthree
echo "${!test@}"
    => testone testtwo testthree
echo "${!test@}"
    => testone testtwo testthree

```

*Dick Guertin (<https://sites.google.com/site/dickguertin/home>), [2014/06/07 03:40 \(\)](#)*

*Under "String length", I'd add that the Parameter can be a system variable, like the positional variables, but with # replacing \$. Thus, something like this is valid: \${#1} for the length of the \$1 variable.*

*Roman S., [2015/02/23 19:17 \(\)](#)*

*Hello there,*

*@1st thanks for sharing your ideas and experiences 😊*

*does one know if this: `mkdir /Linux/Debian7/DVD{1..10}`*

*is equal to this: `for i in {1..10}; do mkdir /Linux/Debian7/DVD$i; done`*  
*or get it expanded to: `mkdir /Linux/Debian7/DVD1 /Linux/Debian7/DVD2 ... /Linux/Debian7/DVD10 ?`*

*Jan Schampera, [2015/07/09 04:32 \(\)](#)*

*It is expanded to multiple strings (but has the same result, at the end of the day)*

*Kip Sanders, [2015/05/07 01:17 \(\)](#)*

*This is the best site I have ever found in terms of showing useful examples of native bash parameter manipulations. However, I still have not found a native bash variable substitution/extraction (without resorting to sed/awk/grep/expr) to extract (rather than stripping) a pattern in the middle of a string parameter, e.g., extracting "123" from the parameter(s) below:*

*`my_var="foo_123_bar"`*

*`my_var2="foo.123:bar"`*

*Of course this can be done in two steps:*

*`$ x=${my_var2/*:}`*

*`$ echo ${x/*.}`*

*123*

*But is there a way to achieve this in one step?*

*Jan Schampera, [2015/07/09 04:35 \(\)](#)*

*You're right, this can only be done in multiple steps. It may be possible to write a generic function to do that (with some backdraws) to wrap the needed steps into one single high-level operation, though.*

*Dan Douglas, [2015/07/20 04:36 \(\)](#)*

*Tricks along these lines work in limited cases.*

```
$ bash -0 extglob -c 'myvar=foo.123:bar; echo "${myvar//@(*.
|:*)}"'
123
```

*Of course, that's quite fragile. Ksh does a better job with pattern substitution and non-greedy quantifiers for regex and shell patterns.*

```
$ ksh -c 'myvar=foo.123:bar; echo "${myvar/#*-[!.]}.*-([^\:]):*/\2}'
123
$ ksh -c 'myvar=foo.123:bar; echo "${myvar/#~(P)[^\.]*?\.[^\:]*?):.*/\1}'
123
```

My preference for this is usually `BASH_REMATCH`.

```
$ { bash /dev/fd/3 ; ksh /dev/fd/3; } 3<<\EOF
${KSH_VERSION+'false'} || typeset -n BASH_REMATCH=.sh.match
myvar=foo.123:bar
[[ $myvar =~ ^[^\.]*\.[^\:]+: ]] && echo "${BASH_REMATCH[1]}"
EOF

123
123
```

Boris Batinkov, [2016/07/25 23:09 \(\)](#)

*In one step, you could simply extract only the numbers:*

```
$ my_var2="foo.123:bar"
$ echo ${my_var2[!0-9]/} 123
```

Christopher LaFave, [2016/03/18 01:57 \(\)](#)

*Thank you so much for this. I badly needed examples in order to become better acquainted with Parameter Expansion and this is a treasure trove of examples. :)*

Thomas, [2016/08/03 08:55 \(\)](#)

*In the section on substring removal you could add the "strip trailing slashes" example `${WORKDIR%/}` to the common uses. Note that because there is no wildcard, it only strips a slash if it's actually the last character.*

graz, [2016/12/01 05:27 \(\)](#)

*Probably the best page I've ever seen explaining BASH specifics. Great examples and simple to read and understand. Great job !*

Valentin Hilbig, [2017/01/15 22:16 \(\)](#)

BTW(): Thanks for compiling this all. Very helpful! But please correct:

*contains only elements that are not the nullstring (the :+ variant)*

*should get two parts erased*

*contains only elements that are ~~not~~ the nullstring (the ~~+~~ variant)*

*such that it reads*

*contains only elements that are the nullstring (the + variant)*

*or shorter contains elements (the + variant)*

*Proof:*

```
a=(); echo ${a[@]}+this is not output};
```


```
a=(); echo ${a[@]:+}this is not output};
```

```
a=""; echo ${a[@]}+this is output};
```

```
a=""; echo ${a[@]:+}this is not output};
```

```
a=.; echo ${a[@]}+this is output};
```

```
a=.; echo ${a[@]:+}this is output};
```

 syntax/pe.txt  Last modified: 2021/12/10 08:12 by ajrou

---

*This site is supported by Performing Databases - your experts for database administration*

---

*Bash Hackers Wiki*

---



Except where otherwise noted, content on this wiki is licensed under the following license:  
GNU Free Documentation License 1.3