The unset builtin command

Synopsis

```
unset [-f|v] [-n] [NAME ...]
```

Description

The `unset` builtin command is used to unset values and attributes of shell variables and functions. Without any option, `unset` tries to unset a variable first, then a function.

Options

Option	Description
`-f`	treats each `NAME` as a function name
`-v`	treats each `NAME` as a variable name
`-n`	treats each `NAME` as a name reference and unsets the variable itself rather than the variable it references

Exit status

Status	Reason
0	no error
!=0	invalid option
!=0	invalid combination of options (`-v` **and** `-f`)
!=0	a given `NAME` is read-only

Examples

```
unset -v EDITOR
```

```
unset -f myfunc1 myfunc2
```

Scope

In bash, unset has some interesting properties due to its unique dynamic scope. If a local variable is both declared and unset (by calling unset on the local) from within the same function scope, then the variable appears unset to that scope and all child scopes until either returning from the function, or another local variable of the same name is declared underneath where the original variable was unset. In other words, the variable looks unset to everything until returning from the function in which the variable was set (and unset), at which point variables of the same name from higher scopes are uncovered and accessible once again.

If however unset is called from a child scope relative to where a local variable has been set, then the variable of the same name in the next-outermost scope becomes visible to its scope and all children - as if the variable that was unset was never set to begin with. This property allows looking upwards through the stack as variable names are unset, so long as unset and the local it unsets aren't together in the same scope level.

Here's a demonstration of this behavior.

```
#!/usr/bin/env bash

FUNCNEST=10

# Direct recursion depth.
# Search up the stack for the first non-FUNCNAME[1] and count how dee
p we are.
callDepth() {
    # Strip "main" off the end of FUNCNAME[@] if current function is
named "main" and
    # Bash added an extra "main" for non-interactive scripts.
    if [[ main == !(!("${FUNCNAME[1]}")|!("${FUNCNAME[-1]}")) && $- !
= *i* ]]; then
        local -a 'fnames=("${FUNCNAME[@]:1:${#FUNCNAME[@]}-2}")'
    else
        local -a 'fnames=("${FUNCNAME[@]:1}")'
    fi

    if (( ! ${#fnames[@]} )); then
        printf 0
        return
    fi

    local n
    while [[ $fnames == ${fnames[++n]} ]]; do
        :
    done

    printf -- $n
}

# This function is the magic stack walker.
unset2() {
    unset -v -- "$@"
}

f() {
    local a
    if (( (a=$(callDepth)) <= 4 )); then
        (( a == 1 )) && unset a
        (( a == 2 )) && declare -g a='global scope yo'
        f
    else
        trap 'declare -p a' DEBUG
        unset2 a   # declare -- a="5"
        unset a a  # declare -- a="4"
        unset a    # declare -- a="2"
        unset a    # ./unset-tests: line 44: declare: a: not found
        :          # declare -- a="global scope yo"
    fi
}

a='global scope'
f
```

```
# vim: set fenc=utf-8 ff=unix ts=4 sts=4 sw=4 ft=sh nowrap et:
```

output:

```
declare -- a="5"
declare -- a="4"
declare -- a="2"
./unset-tests: line 44: declare: a: not found
declare -- a="global scope yo"
```

Some things to observe:

- `unset2` is only really needed once. We remain 5 levels deep in `f` 's for the remaining `unset` calls, which peel away the outer layers of `a` 's.
- Notice that the "a" is unset using an ordinary unset command at recursion depth 1, and subsequently calling unset reveals a again in the global scope, which has since been modified in a lower scope using declare -g.
- Declaring a global with declare -g bypasses all locals and sets or modifies the variable of the global scope (outside of all functions). It has no affect on the visibility of the global.
- This doesn't apply to individual array elements. If two local arrays of the same name appear in different scopes, the entire array of the inner scope needs to be unset before any elements of the outer array become visible. This makes "unset" and "unset2" identical for individual array elements, and for arrays as a whole, unset and unset2 behave as they do for scalar variables.

Args

Like several other Bash builtins that take parameter names, unset expands its arguments.

```
 ~ $ ( a=({a..d}); unset 'a[2]'; declare -p a )
declare -a a='([0]="a" [1]="b" [3]="d")'
```

As usual in such cases, it's important to quote the args to avoid accidental results such as globbing.

```
 ~ $ ( a=({a..d}) b=a c=d d=1; set -x; unset "${b}["{2..3}-c\]; decla
re -p a )
+ unset 'a[2-1]' 'a[3-1]'
+ declare -p a
declare -a a='([0]="a" [3]="d")'
```

Of course hard to follow indirection is still possible whenever arithmetic is involved, also as shown above, even without extra expansions.

In Bash, the `unset` builtin only evaluates array subscripts if the array itself is set.

```
 ~ $ ( unset -v 'a[$(echo a was set >&2)0]' )
 ~ $ ( a=(); unset -v 'a[$(echo a was set >&2)0]' )
a was set
```

Portability considerations

Quoting POSIX:

```
If neither -f nor -v is specified, name refers to a variable; if a va
riable by that name does not exist, it is unspecified whether a funct
ion by that name, if any, shall be unset.
```

Therefore, it is recommended to explicitly specify `-f` or `-v` when using `unset`. Also, I prefer it as a matter of style.

See also

- The declare builtin command
- **The unset builtin command**
- POSIX ''unset'' utility (http://pubs.opengroup.org/onlinepubs/9699919799/utilities/V3_chap02.html#tag_18_

Discussion