

# Bash Pitfalls

This page is a compilation of common mistakes made by bash users. Each example is flawed in some way.

## Содержание

1. for f in \$(ls \*.mp3)
2. cp \$file \$target
3. Filenames with leading dashes
4. [ \$foo = "bar" ]
5. cd \$(dirname "\$f")
6. [ "\$foo" = bar && "\$bar" = foo ]
7. [[ \$foo > 7 ]]
8. grep foo bar | while read -r; do ((count++)); done
9. if [grep foo myfile]
10. if [bar="\$foo"]; then ...
11. if [ [ a = b ] && [ c = d ] ]; then ...
12. read \$foo
13. cat file | sed s/foo/bar/ > file
14. echo \$foo
15. \$foo=bar
16. foo = bar
17. echo <<EOF
18. su -c 'some command'
19. cd /foo; bar
20. [ bar == "\$foo" ]
21. for i in {1..10}; do ./something &; done
22. cmd1 && cmd2 || cmd3
23. echo "Hello World!"
24. for arg in \$\*
25. function foo()
26. echo "~"
27. local var=\$(cmd)
28. export foo=~ /bar
29. sed 's/\$foo/good bye/'
30. tr [A-Z] [a-z]
31. ps ax | grep gedit
32. printf "\$foo"
33. for i in {1..\$n}
34. if [[ \$foo = \$bar ]] (depending on intent)
35. if [[ \$foo =~ 'some RE' ]]
36. [ -n \$foo ] or [ -z \$foo ]
37. [[ -e "\$broken\_symlink" ]] returns 1 even though \$broken\_symlink exists
38. ed file <<<"g/d\{0,3\}/s//e/g" fails
39. expr sub-string fails for "match"
40. On UTF-8 and Byte-Order Marks (BOM)
41. content=\$(- 42. for file in /\* ; do if [[ \$file != \*.\* ]]
- 43. somecmd 2>&1 >>logfile
- 44. cmd; (( ! \$? )) || die
- 45. y=\$(( array[\$x] ))
- 46. read num; echo \$((num+1))
- 47. IFS=, read -ra fields <<< "\$csv\_line"
- 48. export CDPATH=.:~/myProject
- 49. OIFS="\$IFS"; ...; IFS="\$OIFS"
- 50. hosts=( \$(aws ...) )
- 51. Non-atomic writes with xargs -P
- 52. find . -exec sh -c 'echo {}' \;
- 53. sudo mycmd > /myfile
- 54. sudo ls /foo/\*
- 55. myprogram 2>&-

```

56. Using xargs without -0
57. unset a[0]
58. month=$(date +%m); day=$(date +%d)
59. i=$(( 10#$i ))
60. set -euo pipefail
    1. errexit
    2. pipefail
    3. nounset
61. [[ -v hash[$key] ]]
62. (( hash[$key]++ ))
63. while ... done <<< "$(foo)"
64. cmd > "file$((i++))"

```

## 1. for f in \$(ls \*.mp3)

One of the most common mistakes BASH programmers make is to write a loop like this:

```

for f in $(ls *.mp3); do      # Wrong!
    some command $f          # Wrong!
done

for f in $(ls)                # Wrong!
for f in `ls`                 # Wrong!

for f in $(find . -type f)    # Wrong!
for f in `find . -type f`     # Wrong!

files=$(find . -type f)      # Wrong!
for f in ${files[@]}          # Wrong!

```

Yes, it would be great if you could just treat the output of `ls` or `find` as a list of filenames and iterate over it. But you **cannot**. This entire approach is fatally flawed, and there is no trick that can make it work. You must use an entirely different approach.

There are at least 6 problems with this:

1. If a filename contains whitespace (or any character in the current value of `$IFS`), it undergoes WordSplitting. Assuming we have a file named **01 - Don't Eat the Yellow Snow.mp3** in the current directory, the `for` loop will iterate over each word in the resulting file name: `01`, `-`, `Don't`, `Eat`, etc.
2. If a filename contains `"`). If `ls` produces any output containing a `*` character, the word containing it will become recognized as a pattern and substituted with a list of all filenames that match it.
3. If the command substitution returns multiple filenames, there is no way to tell where the first one ends and the second one begins. Pathnames may contain *any* character except NUL. Yes, this includes newlines.
4. The `ls` utility may mangle filenames. Depending on which platform you're on, which arguments you used (or didn't use), and whether its standard output is pointing to a terminal or not, `ls` may randomly decide to replace certain characters in a filename with `"?"`, or simply not print them at all. Never try to parse the output of `ls`. `ls` is just plain unnecessary. It's an external command whose output is intended specifically to be read by a human, not parsed by a script.
5. The CommandSubstitution strips *all* trailing newline characters from its output. That may seem desirable since `ls` adds a newline, but if the last filename in the list ends with a newline, ``...`` or `$()` will remove *that* one also.
6. In the `ls` examples, if the first filename starts with a hyphen, it may lead to pitfall #3.

You can't simply double-quote the substitution either:

```
for f in "$(ls *.mp3)"; do      # Wrong!
```

This causes the entire output of **ls** to be treated as a single word. Instead of iterating over each file name, the loop will only execute *once*, assigning to **f** a string with all the filenames rammed together.

Nor can you simply change IFS to a newline. Filenames can also contain newlines.

Another variation on this theme is abusing word splitting and a **for** loop to (incorrectly) read lines of a file. For example:

```
IFS=$'\n'
for line in $(cat file); do ...    # Wrong!
```

This doesn't work! Especially if those lines are filenames. Bash (or any other Bourne family shell) just doesn't work this way.

### So, what's the right way to do it?

There are several ways, primarily depending on whether you need a recursive expansion or not.

If you don't need recursion, you can use a simple glob. Instead of **ls**:

```
for file in ./*.mp3; do      # Better! and...
    some command "$file"    # ...always double-quote expansions!
done
```

POSIX shells such as Bash have the globbing feature specifically for this purpose — to allow the shell to expand patterns into a list of matching filenames. There is no need to interpret the results of an external utility. Because globbing is the very last expansion step, each match of the **./\*.mp3** pattern correctly expands to a separate word, and isn't subject to the effects of an unquoted expansion.

*Question:* What happens if there are no **/\*.mp3**-files in the current directory? Then the **for** loop is executed once, with **file="./\*.mp3"**, which is not the expected behavior! The workaround is to test whether there is a matching file:

```
# POSIX
for file in ./*.mp3; do
    [ -e "$file" ] || continue
    some command "$file"
```

Another solution is to use Bash's **shopt -s nullglob** feature, though this should only be done after reading the documentation and carefully considering the effect of this setting on all other globs in the script.

If you need recursion, the standard solution is **find**. When using **find**, be sure you use it properly. For POSIX sh portability, use the **-exec** option:

```
find . -type f -name '*.mp3' -exec some command {} \;

# Or, if the command accepts multiple input filenames:

find . -type f -name '*.mp3' -exec some command {} +
```

If you're using bash, then you have two additional options. One is to use GNU or BSD **find** **-print0** option, together with bash's **read -d ''** option and a ProcessSubstitution:

```
while IFS= read -r -d '' file; do
    some command "$file"
done < <(find . -type f -name '*.mp3' -print0)
```

The advantage here is that "some command" (indeed, the entire **while** loop body) is executed in the current shell. You can set variables and have them persist after the loop ends.

The other option, available in Bash 4.0 and higher, is **globstar**, which permits a glob to be expanded recursively:

```
shopt -s globstar
for file in ./**/*.mp3; do
    some command "$file"
done
```

Note the double quotes around **\$file** in the examples above. This leads to our second pitfall:

## 2. cp \$file \$target

What's wrong with the command shown above? Well, nothing, **if** you happen to know in advance that **\$file** and **\$target** have no white space (and you've not modified **\$IFS**, and you can guarantee the code won't be called in a context where **\$IFS** may have been modified) or wildcards in them. However, the results of the expansions are still subject to WordSplitting and pathname expansion. Always double-quote parameter expansions.

```
cp -- "$file" "$target"
```

Without the double quotes, you'll get a command like

**cp 01 - Don't Eat the Yellow Snow.mp3 /mnt/usb**, which will result in errors like

**cp: cannot stat '01': No such file or directory.** If **\$file** has wildcards in it (**\*** or **?** or **[]**), they will be expanded if there are files that match them. With the double quotes, all's well, unless "**\$file**" happens to start with a **-**, in which case **cp** thinks you're trying to feed it command line options (See pitfall #3 below.)

Even in the somewhat uncommon circumstance that you can guarantee the variable's contents, it is conventional and good practice to quote parameter expansions, especially if they contain file names. Experienced script writers will always use quotes except perhaps for a small number of cases in which it is *absolutely* obvious from the immediate code context that a parameter contains a guaranteed safe value. Experts will most likely consider the **cp** command in the title always wrong. You should too.

## 3. Filenames with leading dashes

Filenames with leading dashes can cause many problems. Globbs like **\*.mp3** are sorted into an expanded list (according to your current locale), and sorts before letters in most locales. The list is then passed to some command, which may incorrectly interpret the **-filename** as an option. There are two major solutions to this.

One solution is to insert `--` between the command (like **cp**) and its arguments. That tells it to stop scanning for options, and all is well:

```
cp -- "$file" "$target"
```

There are potential problems with this approach. You have to be sure to insert `--` for *every* usage of the parameter in a context where it might possibly be interpreted as an option — which is easy to miss and may involve a lot of redundancy.

Most well-written option parsing libraries understand this, and the programs that use them correctly should inherit that feature for free. However, still be aware that it is ultimately up to the application to recognize *end of options*. Some programs that manually parse options, or do it incorrectly, or use poor 3rd-party libraries may not recognize it. Standard utilities *should*, with a few exceptions that are specified by POSIX. **echo** is one example.

Another option is to ensure that your filenames always begin with a directory by using relative or absolute pathnames.

```
for i in ./*.mp3; do
    cp "$i" /target
...
done
```

In this case, even if we have a file whose name begins with `-`, the glob will ensure that the variable always contains something like `./-foo.mp3`, which is perfectly safe as far as **cp** is concerned.

Finally, if you can guarantee that all results will have the same prefix, and are only using the variable a few times within a loop body, you can simply concatenate the prefix with the expansion. This gives a theoretical savings in generating and storing a few extra characters for each word.

```
for i in *.mp3; do
    cp ".$i" /target
...
done
```

## 4. [ \$foo = "bar" ]

This is very similar to the issue in pitfall #2, but I repeat it because it's so important. In the example above, the quotes are in the wrong place. You do *not* need to quote a string literal in bash (unless it contains metacharacters or pattern characters). But you *should* quote your variables if you aren't sure whether they could contain white space or wildcards.

This example can break for several reasons:

- If a variable referenced in `[` doesn't exist, or is blank, then the `[` command would end up looking like:

```
[ = "bar" ] # Wrong!
```

...and will throw the error: **unary operator expected**. (The `=` operator is *binary*, not unary, so the `[` command is rather shocked to see it there.)

- If the variable contains internal whitespace, then it gets split into separate words before the `[` command sees it. Thus:

```
[ multiple words here = "bar" ]
```

While that may look OK to you, it's a syntax error as far as `[` is concerned. The correct way to write this is:

```
# POSIX
[ "$foo" = bar ] # Right!
```

This works fine on POSIX-conformant implementations even if `$foo` begins with a `-`, because POSIX `[` determines its action depending on the number of arguments passed to it. Only very ancient shells have a problem with this, and you shouldn't worry about them when writing new code (see the `x"$foo"` workaround below).

In Bash and many other ksh-like shells, there is a superior alternative which uses the `[[` keyword.


```
# Bash / Ksh
[[ $foo == bar ]] # Right!
```

You don't need to quote variable references on the left-hand side of `=` in `[ [ ]]` because they don't undergo word splitting or globbing, and even blank variables will be handled correctly. On the other hand, quoting them won't hurt anything either. Unlike `[` and `test`, you may also use the identical `==`. Do note however that comparisons using `[[` perform pattern matching against the string on the right hand side, not just a plain string comparison. To make the string on the right literal, you must quote it if any characters that have special meaning in pattern matching contexts are used.

```
# Bash / Ksh
match=b*r
[[ $foo == "$match" ]] # Good! Unquoted would also match against the
pattern b*r.
```

You may have seen code like this:

```
# POSIX / Bourne
[ x"$foo" = xbar ] # Ok, but usually unnecessary.
```

The `x"$foo"` hack is required for code that must run on *very* ancient shells which lack `[[` and have a more primitive `[`, which  gets confused if `$foo` begins with a `-` or is `!` or `(`. On said older systems, `[` only requires extra caution for the token on the left-hand side of `=`; it handles the right-hand token correctly.

Note that shells that require this workaround are not POSIX-conforming. Such extreme portability is rarely a requirement and makes your code less readable (and uglier).

## 5. `cd $(dirname "$f")`

This is yet another quoting error. As with a variable expansion, the result of a CommandSubstitution undergoes WordSplitting and pathname expansion. So you should quote it:

```
cd -P -- "$(dirname -- "$f")"
```

What's not obvious here is how the quotes nest. A C programmer reading this would expect the first and second double-quotes to be grouped together; and then the third and fourth. But that's not the case in Bash. Bash treats the double-quotes *inside* the command substitution as one pair, and the double-quotes *outside* the substitution as another pair.

Another way of writing this: the parser treats the command substitution as a "nesting level", and the quotes inside it are separate from the quotes outside it.

## 6. [ "\$foo" = bar && "\$bar" = foo ]

You can't use `&&` inside the old test (or `[]`) command. The Bash parser sees `&&` outside of `[ [ ] ]` or `(( ))` and breaks your command into *two* commands, before and after the `&&`. Use one of these instead:

```
[ bar = "$foo" ] && [ foo = "$bar" ] # Right! (POSIX)
[[ $foo = bar && $bar = foo ]]      # Also right! (Bash / Ksh)
```

(Note that we reversed the constant and the variable inside `[` for the legacy reasons discussed in pitfall #4. We could also have reversed the `[ [` case, but the expansions would require quoting to prevent interpretation as a pattern.) The same thing applies to `| |`. Either use `[ [` instead, or use two `[` commands.

Avoid this:

```
[ bar = "$foo" -a foo = "$bar" ] # Not portable.
```

The binary `-a` and `-o`, and `( / )` (grouping) operators are XSI extensions to the POSIX standard. All are marked as obsolescent in POSIX-2008. They should not be used in new code. One of the practical problems with `[ A = B -a C = D ]` (or `-o`) is that 🌐 POSIX does not specify the results of a `test` or `[` command with more than 4 arguments. It probably works in most shells, but you can't count on it. If you have to write for POSIX shells, then you should use two `test` or `[` commands separated by a `&&` operator instead.

## 7. [[ \$foo > 7 ]]

There are multiple issues here. First, the `[[` command *not* be used solely for evaluating arithmetic expressions. It should be used for test expressions involving one of the supported test operators. Though technically you *can* do math using some of `[ [`'s operators, it only makes sense to do so in conjunction with one of the non-math test operators somewhere in the expression. If you just want to do a numeric comparison (or any other shell arithmetic), it is much better to just use `(( ))` instead:

```
# Bash / Ksh
((foo > 7))      # Right!
[[ foo -gt 7 ]] # Works, but is uncommon. Use ((...)) instead.
```

If you use the `>` operator inside `[ [ ] ]`, it's treated as a string comparison (test for collation order by locale), *not* an integer comparison. This may work sometimes, but it will fail when you

least expect it. If you use `>` inside `[ ]`, it's even worse: it's an output redirection. You'll get a file named `7` in your directory, and the test will succeed as long as `$foo` is not empty.

If strict POSIX-conformance is a requirement, and `((` is not available, then the correct alternative using `[ ]` is

```
# POSIX
[ "$foo" -gt 7 ]          # Also right!
[ "$((foo > 7))" -ne 0 ] # POSIX-compatible equivalent to ((, for more
                        general math operations.
```

If the contents of `$foo` are not sanitised and are out of your control (if for instance they're coming from an external source), then all but `[ "$foo" -gt 7 ]` constitute an arbitrary command injection vulnerability as the contents of `$foo` are interpreted as an arithmetic expression (and for instance, the `a[$(reboot)]` arithmetic expression would run the `reboot` command when evaluated). The `[ ]` builtin requires the operands be *decimal* integers, so it's not affected. But it's critical that `$foo` be quoted, or you'd still get a command injection vulnerability (for instance with values such as `-v a[$(reboot)] -o 8`).

If the input to any arithmetic context (including `((`, `let`, array indices), or `[ [ ... ] ]` test expressions involving numeric comparisons can't be guaranteed then you must *always* validate your input before evaluating the expression.

```
# POSIX
case $foo in
  (" | *[^0123456789]*)
    printf '$foo is not a sequence of decimal digits: "%s"\n' "$foo"
  >&2
    exit 1
  ;;
  *)
    [ "$foo" -gt 7 ]
esac
```

Note that as `[test]`'s arithmetic operator expect *decimal* integers, `010` for instance would be interpreted as the number 10, not 8 expressed in octal. In `bash [ 010 -gt 8 ]` would return true, while `[ [ 010 -gt 8 ] ]` and `(( 010 > 8 ))` would return false.

## 8. `grep foo bar | while read -r; do ((count++)); done`

The code above looks OK at first glance, doesn't it? Sure, it's just a poor implementation of `grep -C`, but it's intended as a simplistic example. Changes to `count` won't propagate outside the `while` loop because each command in a pipeline is executed in a separate SubShell. This surprises almost every Bash beginner at some point.

POSIX doesn't specify whether or not the last element of a pipeline is evaluated in a subshell. Some shells such as ksh93 and Bash `>= 4.2` with `shopt -s lastpipe` enabled will run the `while` loop in this example in the original shell process, allowing any side-effects within to take effect. Therefore, portable scripts must be written in such a way as to not depend upon either behavior.

For workarounds for this and similar issues, please see Bash FAQ #24. It's a bit too long to fit here.



## 9. if [grep foo myfile]

Many beginners have an incorrect intuition about **if** statements brought about by seeing the very common pattern of an **if** keyword followed immediately by a **[** or **[[**. This convinces people that the **[** is somehow part of the **if** statement's syntax, just like parentheses used in C's **if** statement.

This is *not* the case! **if** takes a *command*. **[** is a command, not a syntax marker for the **if** statement. It's equivalent to the **test** command, except that the final argument must be a **]**. For example:

```
# POSIX
if [ false ]; then echo "HELP"; fi
if test false; then echo "HELP"; fi
```

are equivalent — both checking that the argument "false" is non-empty. In both cases HELP will always be printed, to the surprise of programmers from other languages guessing about shell syntax.

The syntax of an **if** statement is:

```
if COMMANDS
then <COMMANDS>
elif <COMMANDS> # optional
then <COMMANDS>
else <COMMANDS> # optional
fi # required
```

Once again, **[** is a command. It takes arguments like any other regular *simple command*. **if** is a *compound command* which contains other commands — and **there is no [** in its syntax!

While bash has a builtin command **[** and thus *knows* about **[** it has nothing special to do with **]**. Bash only passes **]** as argument to the **[** command, which requires **]** to be the last argument only to make scripts look better.

There may be zero or more optional **elif** sections, and one optional **else** section.


The **if** compound command is made up of two or more sections containing *lists* of commands, each delimited by a **then**, **elif**, or **else** keyword, and is terminated by the **fi** keyword. The exit status of the final command of the first section and each subsequent **elif** section determines whether each corresponding **then** section is evaluated. Another **elif** is evaluated until one of the **then** sections is executed. If no **then** section is evaluated, then the **else** branch is taken, or if no **else** is given, the **if** block is complete and the overall **if** command returns 0 (true).

If you want to make a decision based on the output of a **grep** command, you do *not* want to enclose it in parentheses, brackets, backticks, or *any other* syntax! Just use **grep** as the **COMMANDS** after the **if**, like this:

```
if grep -q fooregex myfile; then
...
fi
```

If the **grep** matches a line from **myfile**, then the exit code will be 0 (true), and the **then** part will be executed. Otherwise, if there are no matches, **grep** will return non-zero and the overall **if** command will be zero.

**See also:**

- BashGuide/TestsAndConditionals
-  [http://wiki.bash-hackers.org/syntax/ccmd/if\\_clause](http://wiki.bash-hackers.org/syntax/ccmd/if_clause)

## 10. if [bar="\$foo"]; then ...

```
[bar="$foo"]      # Wrong!
[ bar="$foo" ]    # Still wrong!
[bar = "$foo"]    # Also wrong!
[[bar="$foo"]]    # Wrong again!
[[ bar="$foo" ]]  # Guess what? Wrong!
[[bar = "$foo"]]  # Do I really need to say it...
```

As explained in the previous example, **[** is a command (which can be proven with **type -t [** or **whence -v [**). Just like with any other simple command, Bash expects the command to be followed by a space, then the first argument, then another space, etc. You can't just run things all together without putting the spaces in! Here are the correct ways:

```
if [ bar = "$foo" ]; then ...
if [[ bar = "$foo" ]]; then ...
```

In the first form, **[** is the command name, and **bar**, **=**, the expansion of **"\$foo"**, and **]** are separate arguments to it. There must be whitespace between each pair of arguments, so the shell knows where each argument begins and ends. The second form is similar, except that **[[** is a special keyword, which is terminated by the **]]**. For more details on the difference between the two, see Bash FAQ 31.

## 11. if [ [ a = b ] && [ c = d ] ]; then ...

Here we go again. **[** is a *command*. It is not a syntactic marker that sits between **if** and some sort of C-like "condition". Nor is it used for grouping. You cannot take C-like **if** commands and translate them into Bash commands just by replacing parentheses with square brackets!

If you want to express a compound conditional, do this:

```
if [ a = b ] && [ c = d ]; then ...
```

Note that here we have two *commands* after the **if**, joined by an **&&** (logical AND, shortcut evaluation) operator. It's precisely the same as:

```
if test a = b && test c = d; then ...
```

If the first **test** command returns false, the body of the **if** statement is not entered. If it returns true, then the second **test** command is run; and if that also one returns true, then the body of the **if** statement *will* be entered. (C programmers are already familiar with **&&**. Bash

uses the same *short-circuit evaluation*. Likewise `| |` does short-circuit evaluation for the *OR* operation.)

[[ keyword permit the use of `&&`, so it could also be written this way:

```
if [[ a = b && c = d ]]; then ...
```

See pitfall #6 for a pitfall related to *tests* combined with conditional operators.

## 12. read \$foo

You don't use a `$` before the variable name in a `read` command. If you want to put data into the variable named `foo`, you do it like this:

```
read foo
```

Or more safely:

```
IFS= read -r foo
```

`read $foo` would read a line of input and put it in the variable(s) whose name(s) are in `$foo`. This might be useful if you actually intended `foo` to be a reference to some other variable; but in the majority of cases, this is simply a bug.

## 13. cat file | sed s/foo/bar/ > file

You **cannot** read from a file and write to it in the same pipeline. Depending on what your pipeline does, the file may be clobbered (to 0 bytes, or possibly to a number of bytes equal to the size of your operating system's pipeline buffer), or it may grow until it fills the available disk space, or reaches your operating system's file size limitation, or your quota, etc.

If you want to make a change to a file safely, other than appending to the end of it, use a text editor.

```
printf %s\\n ',s/foo/bar/g' w q | ed -s file
```

If you are doing something that cannot be done with a text editor there *must* be a temporary file created at some point(\*). For example, the following is completely portable:

```
sed 's/foo/bar/g' file > tmpfile && mv tmpfile file
```

The following will *only* work on GNU sed 4.x:


```
sed -i 's/foo/bar/g' file(s)
```

Note that this also creates a temporary file, and does the same sort of renaming trickery — it just handles it transparently.

And the following equivalent command requires perl 5.x:

```
perl -pi -e 's/foo/bar/g' file(s)
```

For more details on replacing contents of files, please see Bash FAQ #21.

(\*) **sponge** from  moreutils uses this example in its manual:

```
sed '...' file | grep '...' | sponge file
```

Rather than using a temporary file plus an atomic **mv**, this version "soaks up" (the actual description in the manual!) all the data, before opening and writing to the **file**. This version will cause data loss if the program or system crashes during the write operation, because there's no copy of the original file on disk at that point.

Using a temporary file + **mv** still incurs a slight risk of data loss in case of a system crash / power loss; to be 100% certain that either the old or the new file will survive a power loss, you must use **sync** before the **mv**.

## 14. echo \$foo

This relatively innocent-looking command causes *massive* confusion. Because the **\$foo** isn't quoted, it will not only be subject to WordSplitting, but also file globbing. This misleads Bash programmers into thinking their variables *contain* the wrong values, when in fact the variables are OK — it's just the word splitting or filename expansion that's messing up their view of what's happening.

```
msg="Please enter a file name of the form *.zip"
echo $msg
```

This message is split into words and any globs are expanded, such as the \*.zip. What will your users think when they see this message:

```
Please enter a file name of the form freenfss.zip lw35nfss.zip
```

To demonstrate:

```
var="*.zip"    # var contains an asterisk, a period, and the word "zip"
echo "$var"    # writes *.zip
echo $var      # writes the list of files which end with .zip
```

In fact, the **echo** command cannot be used with absolute safety here. If the variable contains **-n** for example, **echo** will consider that an option, rather than data to be printed. The only absolutely *sure* way to print the value of a variable is using **printf**:

```
printf "%s\n" "$foo"
```

## 15. \$foo=bar

No, you don't assign a variable by putting a **\$** in front of the variable name. This isn't perl.

## 16. foo = bar

No, you can't put spaces around the `=` when assigning to a variable. This isn't C. When you write `foo = bar` the shell splits it into three words. The first word, `foo`, is taken as the command name. The second and third become the arguments to that command.

Likewise, the following are also wrong:

```
foo= bar    # WRONG!
foo =bar    # WRONG!
$foo = bar; # COMPLETELY WRONG!

foo=bar     # Right.
foo="bar"   # More Right.
```

## 17. echo <<EOF

A here document is a useful tool for embedding large blocks of textual data in a script. It causes a redirection of the lines of text in the script to the standard input of a command. Unfortunately, `echo` is not a command which reads from stdin.

```
# This is wrong:
echo <<EOF
Hello world
How's it going?
EOF

# This is what you were trying to do:
cat <<EOF
Hello world
How's it going?
EOF

# Or, use quotes which can span multiple lines (efficient, echo is
built-in):
echo "Hello world
How's it going?"
```

Using quotes like that is fine — it works great, in all shells — but it doesn't let you just drop a block of lines into the script. There's syntactic markup on the first and last line. If you want to have your lines untouched by shell syntax, and don't want to spawn a `cat` command, here's another alternative:

```
# Or use printf (also efficient, printf is built-in):
printf %s "\
Hello world
How's it going?
"
```

In the example, the `\` on the first line prevents an extra newline at the beginning of the text block. There's a literal newline at the end (because the final quote is on a new line). The lack of `\n` in the `printf` format argument prevents `printf` from adding an extra newline at the end. The `\` trick won't work in single quotes. If you need/want single quotes around the block of text, you have two choices, both of which necessitate shell syntax "contaminating" your data:

```
printf %s \
'Hello world
'
```

```
printf %s 'Hello world'
```

## 18. su -c 'some command'

This syntax is *almost* correct. The problem is, on many platforms, **SU** takes a **-C** argument, but it's not the one you want. For example, on OpenBSD:

```
$ su -c 'echo hello'
su: only the superuser may specify a login class
```

You want to pass **-C 'some command'** to a shell, which means you need a username before the **-C**.

```
su root -c 'some command' # Now it's right.
```

**SU** assumes a username of root when you omit one, but this falls on its face when you want to pass a command to the shell afterward. You must supply the username in this case.

## 19. cd /foo; bar

If you don't check for errors from the **cd** command, you might end up executing **bar** in the wrong place. This could be a major disaster, if for example **bar** happens to be **rm -f \***.

You must **always** check for errors from a **cd** command. The simplest way to do that is:

```
cd /foo && bar
```

If there's more than just one command after the **cd**, you might prefer this:

```
cd /foo || exit 1
bar
baz
bat ... # Lots of commands.
```

**cd** will report the failure to change directories, with a stderr message such as "bash: cd: /foo: No such file or directory". If you want to add your own message in stdout, however, you could use command grouping:

```
cd /net || { echo >&2 "Can't read /net. Make sure you've logged in to
the Samba network, and try again."; exit 1; }
do_stuff
more_stuff
```

Note there's a required space between **{** and **echo**, and a required **;** **}**. You could also write a **die** function, if you prefer.

Some people also like to enable **set -e** to make their scripts abort on *any* command that returns non-zero, but this can be rather tricky to use correctly (since many common commands may return a non-zero for a warning condition, which you may not want to treat as fatal).

By the way, if you're changing directories a lot in a Bash script, be sure to read the Bash help on **pushd**, **popd**, and **dirs**. Perhaps all that code you wrote to manage **cd**'s and **pwd**'s is completely unnecessary.

Speaking of which, compare this:

```
find ... -type d -print0 | while IFS= read -r -d '' subdir; do
    here=$PWD
    cd "$subdir" && whatever && ...
    cd "$here"
done
```

With this:

```
find ... -type d -print0 | while IFS= read -r -d '' subdir; do
    (cd "$subdir" || exit; whatever; ...)
done
```

Forcing a SubShell here causes the **cd** to occur only in the subshell; for the next iteration of the loop, we're back to our normal location, regardless of whether the **cd** succeeded or failed. We don't have to change back manually, and we aren't stuck in a neverending string of ... **&&** ... logic preventing the use of other conditionals. The subshell version is simpler and cleaner (albeit a tiny bit slower).

Another approach is to **cd** unconditionally to where we're supposed to be, at the start of each loop iteration:

```
here=$PWD
find ... -type d -print0 | while IFS= read -r -d '' subdir; do
    cd "$here" || continue
    cd "$subdir" || continue
    whatever
    ...
done
```

At least this way, we can **continue** to the next loop iteration and don't have to string an indefinite series of **&&** together to ensure that we reach the **cd** at the end of the loop body.

## 20. [ bar == "\$foo" ]

The **==** operator is not valid for the POSIX **[** command. Use **=** or the **[[** keyword instead.

```
[ bar = "$foo" ] && echo yes
[[ bar == $foo ]] && echo yes
```

In Bash, **[ "\$X" == y ]** is accepted as an extension, which often leads Bash programmers to think it's the correct syntax. It's not; it's a Bashism. If you're going to use Bashisms, you might as well just use **[[** instead.

## 21. for i in {1..10}; do ./something &; done

You *cannot* put a **;** immediately after an **&**. Just remove the extraneous **;** entirely.

```
for i in {1..10}; do ./something & done
```

Or:

```
for i in {1..10}; do
  ./something &
done
```

& already functions as a command terminator, just like ; does. And you cannot mix the two.

In general, a ; can be replaced by a newline, but not all newlines can be replaced by ;.

## 22. cmd1 && cmd2 || cmd3

Some people try to use && and || as a shortcut syntax for

**if ... then ... else ... fi**, perhaps because they think they are being clever. For instance,

```
# WRONG!
[[ -s $errorlog ]] && echo "Uh oh, there were some errors." || echo
"Successful."
```

However, this construct is *not* completely equivalent to **if ... fi** in the general case. The command that comes after the && also generates an exit status, and if that exit status isn't "true" (0), then the command that comes after the || will *also* be invoked. For example:

```
i=0
true && ((i++)) || ((i--)) # WRONG!
echo "$i"                # Prints 0
```

What happened here? It looks like **i** should be 1, but it ends up 0. Why? Because both the **i++** and **i--** were executed. The **((i++))** command has an exit status, and that exit status is derived from a C-like evaluation of the expression inside the parentheses. That expression's value happens to be 0 (the initial value of **i**), and in C, an expression with an integer value of 0 is considered *false*. So **((i++))** (when **i** is 0) has an exit status of 1 (false), and therefore the **((i--))** command is executed as well.

Another clever person thinks that we can fix it by using the pre-increment operator, since the exit status from **++i** (with **i** initially 0) is true:

```
i=0
true && (( ++i )) || (( --i )) # STILL WRONG!
echo "$i"                   # Prints 1 by dumb luck
```

But that's missing the point of the example. It just to work by , and you *cannot* rely on **x && y || z** if **y** has **any** chance of failure! (This example still fails if we initialize **i** to -1 instead of 0.)

If you need safety, or if you simply aren't sure how this works, or if *anything* in the preceding paragraphs wasn't completely clear, please just use the simple **if ... fi** syntax in your programs.



```
i=0
if true; then
  ((i++))
else
  ((i--))
fi
echo "$i"      # Prints 1
```

This section also applies to Bourne shell, here is the code that illustrates it:

```
# WRONG!
true && { echo true; false; } || { echo false; true; }
```

Output is two lines "true" and "false", instead the single line "true".

## 23. echo "Hello World!"

The problem here is that, in an interactive Bash shell (in versions prior to 4.3), you'll see an error like:

```
bash: !": event not found
```

This is because, in the default settings for an interactive shell, Bash performs csh-style history expansion using the exclamation point. This is **not** a problem in shell scripts; only in interactive shells.

Unfortunately, the obvious attempt to "fix" this won't work:

```
$ echo "hi\!"
hi\!
```

The easiest solution is unsetting the option: this can be done with **set +H** or **set +o histexpand**

Question: Why is playing with **histexpand** more appropriate than single quotes?

*I personally ran into this issue when I was manipulating song files, using commands like*

```
mp3info -t "Don't Let It Show" ...
mp3info -t "Ah! Leah!" ...
```

*Using single quotes is extremely inconvenient because of all the songs with apostrophes in their titles. Using double quotes ran into the history expansion issue. (And imagine a file that has both in its name. The quoting would be atrocious.) Since I never actually use history expansion, my personal preference was to turn it off in ~/.bashrc. — GreyCat*

These solutions will work:

```
echo 'Hello World!'
```

or

```
echo "Hello World!"
```

or

```
set +H
echo "Hello World!"
```

or

```
histchars=
```

Many people simply choose to put **set +H** or **set +o histexpand** in their `~/.bashrc` to deactivate history expansion permanently. This is a personal preference, though, and you should choose whatever works best for you.

Another solution is:

```
exmark='!'
echo "Hello, world$exmark"
```

In Bash 4.3 and newer, a double quote following **!** no longer triggers history expansion, but history expansion is still performed within double quotes, so while **echo "Hello World!"** is OK, these will still be a problem:

```
echo "Hello, World!(and the rest of the Universe)"
echo "foo!'bar'"
```

## 24. for arg in \$\*

Bash (like all Bourne shells) has a special syntax for referring to the list of positional parameters one at a time, and **\$\*** isn't it. Neither is **\$@**. Both of those expand to the list of words in your script's parameters, not to each parameter as a separate word.

The correct syntax is:

```
for arg in "$@"

# Or simply:
for arg
```

Since looping over the positional parameters is such a common thing to do in scripts, **for arg** defaults to **for arg in "\$@"**. The double-quoted **"\$@"** is special magic that causes each parameter to be used as a single word (or a single loop iteration). It's what you should be using at least 99% of the time.

Here's an example:

```
# Incorrect version
for x in $*; do
    echo "parameter: '$x'"
done

$ ./myscript 'arg 1' arg2 arg3
parameter: 'arg'
parameter: '1'
parameter: 'arg2'
parameter: 'arg3'
```

It should have been written:

```
# Correct version
for x in "$@"; do
    echo "parameter: '$x'"
done
# or better:
for x do
    echo "parameter: '$x'"
done

$ ./myscript 'arg 1' arg2 arg3
parameter: 'arg 1'
parameter: 'arg2'
parameter: 'arg3'
```

## 25. function foo()

This works in some shells, but not in others. You should *never* combine the keyword **function** with the parentheses ( ) when defining a function.

Bash (at least some versions) will allow you to mix the two. Most of the shells won't accept that (zsh 4.x and perhaps above will - for example). Some shells will accept **function foo**, but for maximum portability, you should always use:

```
foo() {
...
}
```

## 26. echo "~"

Tilde expansion only applies when '~' is unquoted. In this example echo writes '~' to stdout, rather than the path of the user's home directory.

Quoting path parameters that are expressed relative to a user's home directory should be done using \$HOME rather than '~'. For instance consider the situation where \$HOME is "/home/my photos".

```
"~/dir with spaces" # expands to "~/dir with spaces"
~/dir with spaces" # expands to "~/dir with spaces"
~/dir with spaces" # expands to "/home/my photos/dir with spaces"
"$HOME/dir with spaces" # expands to "/home/my photos/dir with spaces"
```

## 27. local var=\$(cmd)

When declaring a local variable in a function, the **local** acts as a command in its own right. This can sometimes interact oddly with the rest of the line — for example, if you wanted to capture the exit status (\$?) of the CommandSubstitution, you can't do it. **local**'s exit status masks it.

Another problem with this syntax is that in some shells (like bash), **local var=\$(cmd)** is treated as an *assignment*, meaning the right hand side is given special treatment, just like **var=\$(cmd)**; while in other shells, **local var=\$(cmd)** **not** treated as an assignment, and the right hand side will undergo word splitting (because it isn't quoted).

Quoting the right hand side will work around the word splitting issue, but not the exit status masking issue. For both reasons, it's best to use separate commands for this:

Переключить отображение номеров строк

```
1 local var
2 var=$(cmd)
3 rc=$?
```

Both issues are also true of **export** and **readonly**.

The next pitfall describes another issue with this syntax:

## 28. export foo=~ /bar

Tilde expansion (with or without a username) is only guaranteed to occur when the tilde appears at the beginning of a word, either by itself or followed by a slash. It is also guaranteed to occur when the tilde appears immediately after the **=** in an assignment.

However, the **export** and **local** **not** necessarily constitute an assignment. In some shells (like bash), **export foo=~ /bar** will undergo tilde expansion; in others, it will not.

```
foo=~ /bar; export foo    # Right!
export foo="$HOME /bar"  # Right!
```

Using parameter expansion instead of tilde expansion lets use further improve it to:

```
export foo="${HOME%}/ /bar" # Better!
```

Which yields **/bar** instead of **//bar** when **\$HOME** is **/** (which used to be common for the **root** user at least) which is better as on some systems, paths that start with a double-slash have a special meaning.

## 29. sed 's/\$foo/good bye/'

In single quotes, bash parameter expansions like **\$foo** do not get expanded. That is the purpose of single quotes, to protect characters like **\$** from the shell.

Change the quotes to double quotes:

```
foo="hello"; sed "s/$foo/good bye/"
```

But keep in mind, if you use double quotes you might need to use more escapes. See the [Quotes](#) page.

## 30. tr [A-Z] [a-z]

There are (at least) three things wrong here. The first problem is that **[A-Z]** and **[a-z]** are seen as globs by the shell. If you don't have any single-lettered filenames in your current directory, it'll seem like the command is correct; but if you do, things will go wrong. Probably at 0300 hours on a weekend.

The second problem is that this is not really the correct notation for `tr`. What this actually does is translate '[' into '['; anything in the range A-Z into a-z; and ']' into ']'. So you don't even need those brackets, and the first problem goes away.

The third problem is that depending on the locale, A-Z or a-z may not give you the 26 ASCII characters you were expecting. In fact, in some locales z is in the middle of the alphabet! The solution to this depends on what you want to happen:

```
# Use this if you want to change the case of the 26 latin letters
LC_COLLATE=C tr A-Z a-z

# Use this if you want the case conversion to depend upon the locale,
which might be more like what a user is expecting
tr '[:upper:]' '[:lower:]'
```

The quotes are required on the second command, to avoid globbing.

## 31. ps ax | grep gedit

The fundamental problem here is that the name of a running process is inherently unreliable. There could be more than one legitimate gedit process. There could be something else disguising itself as gedit (changing the reported name of an executed command is trivial). For *real* answers to this, see ProcessManagement.

The following is the quick and dirty stuff.

Searching for the PID of (for example) gedit, many people start with

```
$ ps ax | grep gedit
10530 ?      S        6:23 gedit
32118 pts/0  R+       0:00 grep gedit
```

which, depending on a RaceCondition, often yields grep itself as a result. To filter grep out:

```
ps ax | grep -v grep | grep gedit    # will work, but ugly
```

An alternative to this is to use:

```
ps ax | grep '[g]edit'                # quote to avoid shell GLOB
```

This will ignore the grep itself in the process table as that is [g]edit and grep is looking for gedit once evaluated.

On GNU/Linux, the parameter -C can be used instead to filter by commandname:

```
$ ps -C gedit
  PID TTY          TIME CMD
10530 ?            00:06:23 gedit
```

But why bother when you could just use pgrep instead?

```
$ pgrep gedit
10530
```

Now in a second step the PID is often extracted by awk or cut:

```
$ ps -C gedit | awk '{print $1}' | tail -n1
```

but even that can be handled by some of the trillions of parameters for ps:

```
$ ps -C gedit -opid=
10530
```

If you're stuck in 1992 and aren't using pgrep, you could use the ancient, obsolete, deprecated pidof (GNU/Linux only) instead:

```
$ pidof gedit
10530
```

and if you need the PID to kill the process, *pkill* might be interesting for you. Note however that, for example, **pgrep/pkill ssh** would also find processes named sshd, and you wouldn't want to kill those.

Unfortunately some programs aren't started with their name, for example firefox is often started as firefox-bin, which you would need to find out with — well — **ps ax | grep firefox**. 😊 Or, you can stick with pgrep by adding some parameters:

```
$ pgrep -fl firefox
3128 /usr/lib/firefox/firefox
7120 /usr/lib/firefox/plugin-container /usr/lib/flashplugin-
installer/libflashplayer.so -greomni /usr/lib/firefox/omni.ja 3128 true
plugin
```

Please read ProcessManagement. Seriously.

## 32. printf "\$foo"

This isn't wrong because of quotes, but because of a *format string exploit*. If **\$foo** is not strictly under your control, then any `\` or `%` characters in the variable may cause undesired behavior.

Always supply your own format string:

```
printf %s "$foo"
printf '%s\n' "$foo"
```

## 33. for i in {1..\$n}

BashParser performs BraceExpansion *before* any other expansions or substitutions. So the brace expansion code sees the literal **\$n**, which is not numeric, and therefore it doesn't expand the curly braces into a list of numbers. This makes it nearly impossible to use brace expansion to create lists whose size is only known at run-time.

Do this instead:

```
for ((i=1; i<=n; i++)); do
...
done
```

In the case of simple iteration over integers, an arithmetic **for** loop should almost always be preferred over brace expansion to begin with, because brace expansion pre-expands every argument which can be slower and unnecessarily consumes memory.

## 34. if [[ \$foo = \$bar ]] (depending on intent)

When the right-hand side of an = operator inside [[ is not quoted, bash does pattern matching against it, instead of treating it as a string. So, in the code above, if **bar** contains **\***, the result will *always* be true. If you want to check for equality of strings, the right-hand side should be quoted:

```
if [[ $foo = "$bar" ]]
```

If you want to do pattern matching, it might be wise to choose variable names that indicate the right-hand side contains a pattern. Or use comments.

It's also worth pointing out that if you quote the right-hand side of =~ it *also* forces a simple string comparison, rather than a regular expression matching. This leads us to:

## 35. if [[ \$foo =~ 'some RE' ]]

The quotes around the right-hand side of the =~ operator cause it to become a string, rather than a RegularExpression. If you want to use a long or complicated regular expression and avoid lots of backslash escaping, put it in a variable:

```
re='some RE'
if [[ $foo =~ $re ]]
```

This also works around the difference in how =~ works across different versions of bash. Using a variable avoids some nasty and subtle problems.

The same problem occurs with pattern matching inside [[:

```
[[ $foo = "*.glob" ]]      # Wrong! *.glob is treated as a literal
string.
[[ $foo = *.glob ]]        # Correct. *.glob is treated as a glob-style
pattern.
```

## 36. [ -n \$foo ] or [ -z \$foo ]

When using the [ command, you **must** quote each substitution that you give it. Otherwise, **\$foo** could expand to 0 words, or 42 words, or any number of words that isn't 1, which breaks the syntax.

```
[ -n "$foo" ]
[ -z "$foo" ]
[ -n "$(some command with a "$file" in it)" ]

# [[ doesn't perform word-splitting or glob expansion, so you could also
use:
[[ -n $foo ]]
[[ -z $foo ]]
```

### 37. `[[ -e "$broken_symlink" ]]` returns 1 even though `$broken_symlink` exists

Test follows symlinks, therefore if a symlink is broken, i.e. it points to a file that doesn't exist or is in a directory you don't have access to, test -e returns 1 for it even though it exists.

In order to work around it (and prepare against it) you should use:

```
# bash/ksh/zsh
[[ -e "$broken_symlink" || -L "$broken_symlink" ]]


# POSIX sh+test
[ -e "$broken_symlink" ] || [ -L "$broken_symlink" ]
```

### 38. `ed file <<<"g/d\{0,3\}/s//e/g"` fails

The problem caused because `ed` doesn't accept 0 for `\{0,3\}`.

You can check that the following do work:

```
ed file <<<"g/d\{1,3\}/s//e/g"
```

Note that this happens even though POSIX states that BRE (which is the Regular Expression flavor used by `ed`)  should accept 0 as the minimum number of occurrences (see section 5).

### 39. `expr` sub-string fails for "match"

This works reasonably well most of the time

```
word=abcde
expr "$word" : ".\(.*\)"
bcde
```

But WILL fail for the word "match"

```
word=match
expr "$word" : ".\(.*\)"
```

The problem is "match" is a keyword. Solution (GNU only) is prefix with a '+'

```
word=match
expr + "$word" : ".\(.*\)"
atch
```

Or, y'know, stop using `expr`. You can do everything `expr` does by using Parameter Expansion. What's that thing up there trying to do? Remove the first letter of a word? That can be done in POSIX shells using Parameter Expansion or Substring Expansion:

```
$ word=match
$ echo "${word#?}"      # Parameter Expansion
atch
$ echo "${word:1}"      # Substring Expansion
atch
```



Seriously, there's no excuse for using **expr** unless you're on Solaris with its non-POSIX-conforming **/bin/sh**. It's an external process, so it's much slower than in-process string manipulation. And since nobody uses it, nobody understands what it's doing, so your code is obfuscated and hard to maintain.

## 40. On UTF-8 and Byte-Order Marks (BOM)

**In general:** Unix UTF-8 text does not use BOM. The encoding of plain text is determined by the locale or by mime types or other metadata. While the presence of a BOM would not normally damage a UTF-8 document meant only for reading by humans, it is problematic (often syntactically illegal) in any text file meant to be interpreted by automated processes such as scripts, source code, configuration files, and so on. Files starting with BOM should be considered equally foreign as those with MS-DOS linebreaks.

**In shell scripting:** 'Where UTF-8 is used transparently in 8-bit environments, the use of a BOM will interfere with any protocol or file format that expects specific ASCII characters at the beginning, such as the use of "#!" of at the beginning of Unix shell scripts.' 🌐  
[http://unicode.org/faq/utf\\_bom.html#bom5](http://unicode.org/faq/utf_bom.html#bom5)

## 41. content=\$(**<file**)

There isn't anything wrong with this expression, but you should be aware that command substitutions (all forms: **(ksh)**) remove any trailing newlines. This is often inconsequential or even desirable, but if you must preserve the literal output including any possible trailing newlines, it gets tricky because you have no way of knowing whether the output had them or how many. One ugly but usable workaround is to add a postfix inside the command substitution and remove it on the outside:

```
absolute_dir_path_x=$(readlink -fn -- "$dir_path"; printf x)
absolute_dir_path=${absolute_dir_path_x%x}
```

A less portable but arguably prettier solution is to use **read** with an empty delimiter.

```
# Ksh (or bash 4.2+ with lastpipe enabled)
readlink -fn -- "$dir_path" | IFS= read -rd '' absolute_dir_path
```

The downside to this method is that the **read** will always return false unless the command outputs a NUL byte causing only part of the stream to be read. The only way to get the exit status of the command is through **PIPESTATUS**. You could also intentionally output a NUL byte to force **read** to return true, and use **pipefail**.

```
set -o pipefail
{ readlink -fn -- "$dir_path" && printf '\0'; } | IFS= read -rd ''
absolute_dir_path
set +o pipefail
```

This is somewhat of a portability mess, as Bash supports both **pipefail** and **PIPESTATUS**, ksh93 supports only, and only recent versions of mksh support **pipefail**, while earlier versions supported only. Additionally, a bleeding-edge ksh93 version is required in order for **read** to stop at the NUL byte.

## 42. for file in ./\* ; do if [[ \$file != \*.\* ]]

One way to prevent programs from interpreting filenames passed to them as options is to use pathnames (see pitfall #3 above). For files under the current directory, names may be prefixed with a relative pathname `./`.

In the case of a pattern like `*.*` however, problems can arise because it matches a string of the form `./filename`. In a simple case, you can just use the glob directly to generate the desired matches. If however a separate pattern-matching step is required (e.g. the results have been preprocessed and stored in an array, and need to be filtered), it could be solved by taking the prefix into account in the pattern: `[[ $file != *.* ]]`, or by stripping the pattern from the match.

```
# Bash
shopt -s nullglob
for path in ./*; do
    [[ ${path##*/} != *.* ]] && rm "$path"
done


# Or even better
for file in *; do
    [[ $file != *.* ]] && rm "./$file"
done

# Or better still
for file in *.*; do
    rm "./$file"
done
```

Another possibility is to signal the *end of options* with a `--` argument. (Again, covered in pitfall #3).

```
shopt -s nullglob
for file in *; do
    [[ $file != *.* ]] && rm -- "$file"
done
```

## 43. `somecmd 2>&1 >>logfile`

This is by far the most common mistake involving redirections. Typically, someone wanting to direct both stdout and stderr to a file or pipe will try this and not understand why stderr is *still* showing up on their terminal. If you're perplexed by this, you probably don't understand how  redirections or possibly file descriptors work to begin with. Redirections are evaluated left-to-right before the command is executed. This semantically incorrect code essentially means: "first redirect standard error to where standard out is currently pointing (the tty), then redirect standard out to logfile". This is backwards. Standard error is already going to the tty. Use the following instead:

```
somecmd >>logfile 2>&1
```

See a more in-depth explanation,  [Copy descriptor explained](#), and [BashGuide - redirection](#).

## 44. `cmd; (( ! $? )) || die`

`$?` is only required if you need to retrieve the exact status of the previous command. If you only need to test for success or failure (any non-zero status), just test the command directly. e.g.:

```
if cmd; then
    ...
fi
```

Checking an exit status against a list of alternatives might follow a pattern like this:

```
status=$?
case $status in
    0)
        echo success >&2
        ;;
    1)
        echo 'Must supply a parameter, exiting.' >&2
        exit 1
        ;;
    *)
        echo "Unknown error $status, exiting." >&2
        exit "$status"
esac
```

## 45. `y=$(( array[$x] ))`

This pitfall occurs in any context in which an array's name and index are subject to word expansions before being evaluated as a name literal. Pitfall 61 and 62 are the consequence of the exact same mechanisms.

The code given to an arithmetic expansion or compound command undergoes an initial pass of expansions and substitutions to generate the text to be parsed and evaluated as an arithmetic expression. *This must be handled carefully.*

For example, this expression is stitched together by expanding one code fragment into another.

```
$ x='$(date >&2)'          # redirection is just so we can see everything
                           happen
$ y=$((array["$x"]))      # Quotes don't help. The array doesn't even have
                           to exist
Mon Jun  2 10:49:08 EDT 2014
```

Next, the expanded string is passed to the arithmetic processor, which will need to get a reference to the array variable in the shell's internal symbol table using a lookup function to resolve the variable's "name". This name resolver takes a string -

**`array[$(date >&2)]`** - consisting of the name, including the index and all literal code within brackets, just like e.g. **`read`** or **`printf -v`** do with variable names passed as arguments. The variable resolver performs expansions, *including command substitution*, to resolve the index.

(For numeric indexed arrays, the lookup function next evaluates the expanded text of the index as an arithmetic expression. Consequently, mutually recursive variable lookups and arithmetic expansions can occur to any depth (up to Bash's defined limit), any of which can produce unintended side-effects.)

Most of the time, there is no need to use any kind of expansion within an arithmetic expansion. Use variable names directly in the expression (no `$`) wherever possible (i.e. except for positional parameters and POSIX "special variables"). Validate variables before using them and assure no expansion generates anything but a numeric literal - most issues are automatically avoided.

Escape any expansions to pass them into the expression without expanding them first:

Переключить отображение номеров строк

```
1 # Typical task reading some columns into an associative array.
2 typeset -A arr
3 printf -v offset '%(s)T' -1
4
5 while IFS=' ' read -r x y; do
6     # validate input (see next pitfall)
7     [[ "$x $y" == +([0123456789])' ' +([0123456789]) ]] || continue
8     # Escaped substitution passes the entire expression literally.
9     (( arr[\$(date -d "@$x" +%F)] = y - offset ))
10 done
```

Another option is to use `let` with single-quoted arguments. `((expr))` is equivalent to `let "expr"` (double-quoted args).

## 46. read num; echo \$((num+1))

Always validate your input (see BashFAQ/054) before using `num` in an arithmetic context as it allows code injection.

```
$ echo 'a[$(echo injection >&2)]' | bash -c 'read num; echo $((num+1))'
injection
1
```

## 47. IFS=, read -ra fields <<< "\$csv\_line"

Unbelievable as it may seem, POSIX requires the treatment of IFS as a field *terminator*, rather than a field *separator*. What this means in our example is that if there's an empty field at the end of the input line, it will be discarded:

```
$ IFS=, read -ra fields <<< "a,b,"
$ declare -p fields
declare -a fields='([0]="a" [1]="b")'
```

Where did the empty field go? It was eaten for historical reasons ("because it's always been that way"). This behavior is not unique to bash; all conformant shells do it. A non-empty field is properly scanned:

```
$ IFS=, read -ra fields <<< "a,b,c"
$ declare -p fields
declare -a fields='([0]="a" [1]="b" [2]="c")'
```

So, how do we work around this nonsense? As it turns out, appending an IFS character to the end of the input string will force the scanning to work. If there was a trailing empty field, the extra IFS character "terminates" it so that it gets scanned. If there was a trailing non-empty field, the IFS character creates a new, empty field that gets dropped.

```
$ input="a,b,"
$ IFS=, read -ra fields <<< "$input,"
$ declare -p fields
declare -a fields='([0]="a" [1]="b" [2]="")'
```

The same issue may appear when we use `read` with a fixed number of field variables, and we wish to get the rest of the line, with no alterations, into the final variable. The shell thwarts us.

```
$ echo 'foo:bar:' | { IFS=: read -r f1 rest; declare -p f1 rest; }
declare -- f1="foo"
declare -- rest="bar"
```

The trailing `:` has been stripped off. This only happens when there is precisely *one* extra separator in the rest of the line, and it's the final character. In all other cases, the pitfall remains untriggered.

```
$ echo 'foo:bar:f' | { IFS=: read -r f1 rest; declare -p rest; }
declare -- rest="bar:f"
$ echo 'foo:bar::' | { IFS=: read -r f1 rest; declare -p rest; }
declare -- rest="bar::"
```

This allows the problem to go undetected for *ages* before finally striking.

To work around this behavior, we can add an extra separator character to the end of the line, and then remove it afterward.

```
$ input='foo:bar:'
$ echo "$input:" | { IFS=: read -r f1 rest; rest=${rest%:}; declare -p
rest; }
declare -- rest="bar:"
```

## 48. export CDPATH=.:~/myProject

Do not export CDPATH.

Setting CDPATH in `.bashrc` is not an issue, but exporting it will cause any bash or sh script you run, that happen to use `cd`, to potentially change behaviour.

There are two problems. A script that does the following:

```
cd some/dir || exit
cmd to be run in some/dir
```

may change directory to `~/myProject/some/dir` instead of `./some/dir`, depending on what directories exist at the time. So the `cd` may succeed and take the script to the wrong directory, with potentially harmful effects of the following commands which now run in a different directory than intended.

The second problem is when `cd` is run in a context where the output is captured:

```
output=$(cd some/dir && some command)
```

As a side-effect when CDPATH is set, `cd` will output something like `/home/user/some/dir` to stdout to indicate that it found a directory through CDPATH, which in turn will end up in the output variable along with the intended output of `some command`.

A script can make itself immune to a CDPATH inherited from the environment by always prepending `./` to relative paths, or run `unset CDPATH` at the start of the script, but don't assume every scripter has taken this pitfall into account, so don't export CDPATH.

## 49. OIFS="\$IFS"; ...; IFS="\$OIFS"

Directly assigning a variable's value to a temporary variable isn't alone enough to restore its state. The assignment will always result in a *set* but *empty* temporary variable even if the initial variable was unset. This is a particular problem for IFS because an *empty* IFS has a completely different meaning from an *unset* IFS, and setting IFS to a temporary value for a command or two is a common requirement.

An easy workaround is to designate a prefix to distinguish set from unset vars, then strip it when finished.

```
oIFS=${IFS+_${IFS}}
IFS=/; echo "${array[*]}"
${oIFS:+'false'} unset -v IFS || IFS=${oIFS#_}
```

A local variable is usually preferable when possible.

```
f() {
    local IFS
    IFS=/; echo "${array[*]}"
}
f
```

Subshells are another possibility.

```
( IFS=/; echo "${array[*]}" )
```

## 50. hosts=( \$(aws ...) )

It is not safe to populate an array with a raw `$(...)` CommandSubstitution. The output of the command undergoes word splitting (on *all* whitespace, even ones that are inside quotes) and then globbing. If there's a word like `*` or `eh?` or `[abc]` in the result, it will be expanded based on filenames in the current working directory.

To select a replacement, you need to know whether the command writes its output on a single line, or multiple lines. If it's a single line:

```
read -ra hosts < <(aws ...)
```

If it's multiple lines (and you're targeting bash 4.0 or later):

```
readarray -t hosts < <(aws ...)
```

If it's multiple lines (and you want compatibility with bash 3.x, *or* want your command's exit status to be reflected in success or failure of the **read** operation without depending on behavior only available in bash 4.4 and newer):

```
IFS=$'\n' read -r -d '' -a hosts < <(aws ... && printf '\0')
```

This will prevent globbing. It still won't help you if you needed to avoid splitting on quoted whitespace, but unfortunately *nothing* bash can do handles that case. For generalized CSV (comma-separated value) file handling, you really need to switch to a language that has a dedicated CSV input library.

## 51. Non-atomic writes with xargs -P

GNU **xargs** supports running multiple jobs in parallel. **-P n** where **n** is the number of jobs to run in parallel.

```
seq 100 | xargs -n1 -P10 echo "$a" | grep 5
seq 100 | xargs -n1 -P10 echo "$a" > myoutput.txt
```

This will work fine for many situations but has a deceptive flaw: If **\$a** contains more than 8192 characters (the limit depends on platform and version), the **echo** may not be atomic (it may be split into multiple **write()** calls), and there is a risk that two lines will be mixed.

```
$ perl -e 'print "a"x10000, "\n" ' > foo
$ strace -e write bash -c 'read -r foo < foo; echo "$foo"' >/dev/null
write(1, "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa...", 8192) = 8192
write(1, "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa...", 1809) = 1809
+++ exited with 0 +++
```

Obviously the same issue arises if there are multiple calls to **echo** or **printf**:

```
slowprint() {
    printf 'Start-%s ' "$1"
    sleep "$1"
    printf '%s-End\n' "$1"
}
export -f slowprint
seq 10 | xargs -n1 -I {} -P4 bash -c "slowprint {}"
# Compare to no parallelization
seq 10 | xargs -n1 -I {} bash -c "slowprint {}"
# Be sure to see the warnings in the next Pitfall!
```

Outputs from the parallel jobs are mixed together, because each job consists of two (or more) separate **write()** calls.


If you need the outputs unmixed, it is therefore recommended to use a tool that guarantees output will be serialized (such as GNU Parallel).

For further details see  a demonstration of the mixing problem.

## 52. find . -exec sh -c 'echo {}' \;

This command contains a vulnerability. The filename that is found by **find** is injected into a shell command and parsed by **sh**. If the filename contains shell metacharacters like **;** or **\$( ... )** then the filename may be *executed as code* by **sh**.

The "slowprint" example in the previous Pitfall would have been a bug if the input weren't guaranteed to be integers.

To be more precise,  POSIX **find** does not specify whether an argument which contains *more than* just **{}** is expanded. GNU **find** allows this CodeInjection to occur. Other implementations choose a safer path:

```
# uname -a
HP-UX imadev B.10.20 A 9000/785 2008897791 two-user license
```

```
# find /dev/null -exec sh -c 'echo {}' \;  
{}
```

The correct approach is to *separate* the filename argument from the script argument:

```
find . -exec sh -c 'echo "$1"' x {} \;
```

## 53. `sudo mycmd > /myfile`

Redirection is done *before* the command is executed. Usually that doesn't matter, but with **sudo** we have a command being executed as a different user than the redirection.

If the redirection must be executed with **sudo**-granted privileges, then you need a wrapper:

```
sudo sh -c 'mycmd > /myfile'
```

Instead of a wrapper you can use **tee**:

```
mycmd | sudo tee /myfile >/dev/null
```

This may be easier to write if **mycmd** has a lot of quoting.

## 54. `sudo ls /foo/*`

This is very similar to the previous pitfall. Globbing is also done *before* the command is executed. If the directory isn't readable by your normal user privileges, then you may need the globbing to be done in a shell that has the **sudo**-granted privileges:

```
sudo sh -c 'ls /foo/*'
```

## 55. `myprogram 2>&-`

**Do not** close stdin, stdout or stderr as a "shorthand" for redirecting to `/dev/null`. Write it out correctly.

```
myprogram 2>/dev/null
```

Why? Consider what happens when your program tries to write an error message to stderr. If stderr has been redirected to `/dev/null`, the write succeeds, and your program is free to carry on, secure in the knowledge that it has diligently reported the error condition.

But if stderr has been *closed*, then the write will fail. At that point, your program may do something unpredictable. It may carry on and ignore the failure, or it may immediately exit, considering the execution environment so broken that it cannot safely continue. Or whatever else the programmer decided the program should do when its world has become a dystopian hell.

All programs are assured that stdin, stdout and stderr will *exist* and will be readable/writable in an appropriate and reasonable manner. By closing one of them, you have violated your promise to this program. This is not acceptable.



Of course, an even better solution would be to actually log the errors somewhere, so you can go back and read them and figure out what's wrong.

## 56. Using `xargs` without `-0`

`xargs` splits on whitespace. This is unfortunate because whitespace is allowed in filenames and commonly used by GUI users. `xargs` also treats `'` and `"` specially, which can also lead to problems:

```
touch Dad\'s\ \"famous\" 1\'\' pizza.txt
touch Dad\'s\ 12\'\' records.txt
touch 2\'\"x1\'\' wood.txt
touch 2\'\"x4\'\' wood.txt
```

Here `xargs` warns:

```
# Do not do this
$ find . -type f | xargs wc
xargs: unmatched single quote; by default quotes are special to xargs
unless you use the -0 option
```

Here `xargs` does not warn at all:

```
# Do not do this
echo * | xargs wc
find *famous* -type f | xargs wc
find *4* -type f | xargs wc
```

Instead use `xargs -0`:

```
# Do this instead
printf '%s\0' * | xargs -0 wc
find . -type f -name '*famous*' -print0 | xargs -0 wc
find . -type f -name '*4*' -exec wc {} +
```

If using `-0` is not simple, an alternative is to use GNU Parallel, which splits on `\n`. And while `\n` is also allowed in filenames they never occur unless your users are malicious. In any case: **If** you use `xargs` without `-0` put a comment in your code explaining why that is safe in your particular situation.

## 57. `unset a[0]`

When passing an indexed array element to `unset`, it needs to be quoted. Otherwise, it may be treated as a glob, and expanded against the files in the current directory. If there happens to be a file named `a0` then the glob is expanded to `a0` and you end up executing `unset a0`.

It is also generally preferable to specify the `-V` option when unsetting a variable. Otherwise, if the variable specified by the argument to `unset` does not exist, and there happens to be a function with that name, the `unset` command will unset the function instead of just doing nothing. This does not seem to happen with bash's `unset` if the argument contains `[ . . . ]` (you need to use the `-f` option explicitly to unset functions that have `[ . . . ]` in their name), but that is undocumented, so it is best to use `-V` anyway even when unsetting an array element.

```
unset -v 'a[0]'      # Always quote indexed array elements when unsetting.
```

## 58. month=\$(date +%m); day=\$(date +%d)

Calling **date** multiple times is a bad idea. Imagine what happens if the first call occurs a millisecond before midnight on April 30, and the second call occurs a millisecond after midnight on May 1. You would end up with month=04 and day=01.

It's better to call *date* *one* time, retrieving all of the fields you want in a single invocation.

A common idiom for that:

```
eval "$(date +'month=%m day=%d year=%Y dayname="%A" monthname="%B"' )"
```

Or with bash's (4.2 or above) **printf** builtin:

```
printf -v d '%(month=%m day=%d year=%Y dayname="%A" monthname="%B")T'
eval "$d"
```

Remember things like month or day names are locale-dependent, hence the quotes around **%A** or **%B** to avoid problems in locales where day or month names contain spaces or other special characters for the shell.

Or, you may retrieve a timestamp in epoch format (seconds since the start of 1970), and then use that to generate human-readable date/time fields as needed.

```
# Requires bash 4.2 or above
printf -v now '%(%s)T' -1      # Or now=$EPOCHSECONDS in bash 5.0
                                # -1 may be omitted in 4.3 or above
printf -v month '%(%m)T' "$now"
printf -v day '%(%d)T' "$now"
```

If your system's **strftime()** doesn't support **%S**, you can get the epoch time with:

```
now=$(awk 'BEGIN{srand(); print srand()}')
```

## 59. i=\$(( 10#\$i ))

Forced base 10 interpretation only works with signless numbers. As long as **\$i** contains a string of digits with no leading **-** or **+**, everything is fine. But if **\$i** might be negative, this conversion could fail, either noisily (with an error message), or even worse, silently (simply yielding the wrong result).

If there's any chance **\$i** could be negative, use this instead:

```
i=$(( ${i%[!+-]*}10#${i#[!+-]} ))
```

For explanations, please see [ArithmeticExpression](#).

## 60. set -euo pipefail

There are many pitfalls to enabling these options at the start of a script.

## 60.1. errexit

errexit (set -e) tries to abort the script when an error occurs, which sounds like a good idea at first, but it has very intricate rules regarding when to abort on error or not. Some of the main problems with errexit are

- It's not actually possible for the shell to detect errors. All it has to go on is a command's exit status. When commands fail, they normally return a non-zero exit status, but many commands also use the exit status to convey a true/false value. Examples of such commands are **test**, **[**, **[[ ... ]]**, **((...))**, and **grep**.
- When a command you test with **if** or **&&** or **||** is a function, set -e ignores non-zero exit statuses of commands in that function. Consider a function like

```
# WRONG
cleanup() {
    cd "$1"
    rm -f ./.*
}
```

If that **cd** command fails, you definitely don't want that **rm** command to run, and with a simple use of the function when errexit is enabled, that happens to be the case:

```
set -e
cleanup() {
    cd "$1"
    printf 'Oops!\n'
}
cleanup /no/longer/there
# scriptname: cd: /no/longer/there: No such file or directory
```

But then later you decide to add a custom error message

```
cleanup /no/longer/there || {
    printf ">&2 'Cleanup failed\n'"
    exit 1
}
# scriptname: cd: /no/longer/there: No such file or directory
# Oops!
```

**cd** is being run in a context where it's tested, even though it's not at all apparent inside the function definition where the line is located. The correct way, with or without errexit, is to explicitly check if **cd** failed or not

```
# Right
cleanup() {
    cd "$1" || return
    rm -f ./.*
}
```

- Command substitution disables errexit. Consider the following example function that mimics a **realpath** command:

```
set -e
realdir() {
    cd "$1"
    pwd -P
}
```

```
readdir /no/such/dir
# scriptname: cd: /no/such/dir
```

It aborted on the failing **cd** command, but if you try to capture its output, then **cd**'s failure is ignored

```
dir=$(readdir /no/such/dir)
# scriptname: cd: /no/such/dir
printf 'dir is <%s>\n' "$dir"
# dir is </home/user>
```

The correct way is again to explicitly check if **cd** failed or not, regardless of whether you use **errexit**

```
# Right
readdir() {
    cd "$1" || return
    pwd -P
}
```

See

- Why doesn't set -e (or set -o errexit, or trap ERR) do what I expected?.
- 🌐 Semipredicate problem

## 60.2. pipefail

Normally, the exit status of a pipeline is the exit status of the last command of that pipeline. With **pipefail**, if any part of the pipeline returns non-zero, the whole pipeline returns non-zero. This may sound like a good idea, but it's actually normal for commands earlier in a pipeline to exit non-zero without it being an error. Consider the following example

```
if some_command | grep -q foo; then
    printf 'foo found\n'
else
    printf 'foo not found\n'
fi
```

If **pipefail** is enabled, the above will sometimes claim "foo not found" even when foo was found.

**grep -q foo** returns 0 to indicate that at least one line matches the pattern, else it returns 1 to indicate that no lines matched the pattern. Since **-q** (quiet) causes **grep** to not output the matching lines, it doesn't actually need to read the whole stream; it's enough to find at least one line that matches the pattern. A typical **grep** implementation will thus exit with status 0 as soon as it found the first matching line.

on the other end of the pipeline will not know that. It only sees its end of the pipe, it does not see what is on the other end. When it tries to write the next chunk of data to the pipe, and the pipe is closed in the other end, the system will send **some\_command** a **SIGPIPE** signal to tell it that it can no longer write to the pipe. The default action to take when **SIGPIPE** is received is to exit with a non-zero status. This means that the whole pipeline returns non-zero, and the **else** branch is executed, even though it successfully found a matching line.

If you know that each part of a pipeline beyond the first will consume all their input, **pipefail** is safe and often a good choice, but it should not be enabled globally, only enabled for pipelines

where it makes sense, and disabled afterwards.

### 60.3. nounset

nounset (set -u) is the least bad of the three options, but has its fair share of gotchas too. If the goal is to catch typoed variable names,  shellcheck does a better job at detecting those.

See What are the advantages and disadvantages of using set -u (or set -o nounset)?

## 61. [[ -v hash[\$key] ]]

Here, **hash** is an associative array. This construct fails because **\$key** is expanded before the array subscript evaluation, and then the whole array plus expanded index is evaluated in a second pass. It will appear to work for simple keys, but it will fail for keys containing quotes, closing square brackets, etc. Even worse, it introduces a CodeInjection if the **\$key** contains syntax.

The same problem applies to the **test** and **[** commands, as well as any use of an associative array element in an arithmetic context.

Newer versions of bash (5.0 and higher) have a option which will suppress the multiple evaluations. Another choice is to single-quote the entire argument:


```
[[ -v 'hash[$key]' ]]
```

This has the advantage of working in older versions of bash as well as newer versions.

## 62. (( hash[\$key]++ ))

Surprise! Associative arrays are even *more* broken in math contexts. The single-quoting trick from the previous pitfall isn't enough to fix this, either.

```
$ declare -A hash
$ key='\['
$ hash[$key]=17
$ (( hash[$key]++ ))
bash: ((: hash['']++ : bad array subscript (error token is "hash['']++ ")
$ (( 'hash[$key]++' ))
bash: ((: 'hash['']++' : syntax error: operand expected (error token is "'hash['']++' ")
```

 According to Chet Ramey, the single quotes don't work here because the contents of a math context are treated as though they are double-quoted, and therefore the single quotes have no special meaning. Prior to the release of bash 5.2, backslash still retained its special meaning here, and could be used to quote the index:

```
(( hash[\$key]++ ))    # Safe before bash 5.2.
```

As of bash 5.2, the **only** safe, working way to modify an element of an associative array in a calculation is to make a temporary copy of the value in a regular (string) variable.

```
tmp=${hash[$key]}
((tmp++))          # Safe.
hash[$key]=$tmp
```

```
# Or:
tmp=${hash[$key]}
hash[$key]=$((tmp+1)) # Safe.
```

I strongly recommend using the temporary variable, even if you're "certain" that your script will never be run in bash 5.2 or newer. Someone might install bash 5.2 by surprise. There's no reason to use the older, riskier workarounds.

I don't know what the deal is with 5.2. If that's broken then pf45's workaround is also broken until bash gets this sorted. It appears bash has made the parser more complex by selectively removing the pre-expansion step for certain parts of expressions. That's bad because the rules for when that happens are undocumented, and you would need to know how bash decides that in order to know where to escape things. **let** looks unaffected, as expected, because its arguments are passed directly to **arith.c** without the parser mangling the expression unpredictably.

```
(ins)gentoo@ormaa-j-laptop (27228) ~ $ ( shopt -u assoc_expand_once;
key='\'; typeset -A arr; arr[$key]=0; let 'arr[$key]++'; typeset -p arr
BASH_VERSINFO )
declare -A arr=(["]"]="1" )
declare -ar BASH_VERSINFO=([0]="5" [1]="2" [2]="0" [3]="1" [4]="release"
[5]="x86_64-pc-linux-gnu")
```

Same with or without **assoc\_expand\_once**. But looks like **(( ))** isn't safe for now until we have more information. Hopefully that gets fixed or reverted.

## 63. while ... done <<< "\$(**foo**)"

Let's start with the correct answer first, and then explain why the wrong answer is wrong. Here's what you should be doing instead:

```
while ... done < <(foo)
```

Or, if you don't actually need the **while** loop to run in the current shell process, you can use an ordinary pipeline:

```
foo | while ... done
```

When using the **< <(**foo**)** (ProcessSubstitution) version, what happens is pretty simple. Depending on your operating system, either a named pipe is created, or a **/dev/fd/\*** entry is used to similar effect. The **while** loop's standard input is redirected from that source. Then, a background subshell is launched, in which the writer command **foo** is executed. The output of **foo** goes into the pipe (or pipe-analogue), and the **while** loop reads from it. It's all very simple. There's nothing to worry about.

With the **<<< "\$(**foo**)"** variant, there are a few additional steps involved, and some of them are undesirable. For starters, the entire output of **foo** has to be collected first. There is no simultaneous execution, as there is with the pipeline or the process substitution. Second, due to the way **\$(...)** works, there are *three* modifications to the input stream. All NUL bytes are discarded, either silently, or with a warning, depending on the version of bash. All trailing newlines are removed. Finally, a newline character is added to the end of the input. All of the

modified input is written into a temporary file, and then the temporary file is opened for reading by the `while` loop.

## 64. `cmd > "file$((i++))"`

Redirections for simple commands may be run in a subshell, so the value of `$i` in the following example may not change in the main shell (it will not in `bash` and `ksh`, but it will in `dash`):

Переключить отображение номеров строк

```
1 #!/bin/bash --
2 i=0
3
4 declare -p files=( * ) i
5 # declare -a files=()
6 # declare -- i="0"
7
8 cmd arg > "file$(( i++ ))"
9
10 declare -p files=( * ) i
11 # declare -a files=[0]="file0"
12 # declare -- i="0"
```

Running the redirections in a subshell (i.e. after the fork) is done as an optimisation; it saves the shell the need to restore the redirected file descriptor after the command has terminated.

You can use the following alternatives to prevent this optimisation:

Переключить отображение номеров строк

```
1 #!/bin/bash --
2 shopt -s nullglob
3
4 i=0
5 declare -p files=( * ) i
6 # declare -a files=()
7 # declare -- i="0"
8
9 # Use a temporary variable to ensure that file$(( i++ )) is
expanded in
10 # the correct shell.
11 file=file$(( i++ ))
12 cmd arg > "$file"
13 declare -p files=( * ) i
14 # declare -a files=[0]="file0"
15 # declare -- i="1"
16
17 # Or apply the redirections to a group compound command that only
18 # contains the simple command.
19 { cmd arg ; } > "file$(( i++ ))"
20 declare -p i files=( * )
21 # declare -a files=[0]="file0" [1]="file1"
22 # declare -- i="2"
```

Bash also applies this optimisation to subshell compound commands:

Переключить отображение номеров строк

```
1 #!/bin/bash --
2 shopt -s nullglob
3
4 i=0
5 declare -p files=( * ) i
```

```

6 # declare -a files=()
7 # declare -- i="0"
8
9 (cmd1 arg1; cmd2 arg2) > "file$(( i++ ))"
10 declare -p files=( * ) i
11 # declare -a files=[0]="file0"
12 # declare -- i="0"
13
14 # Again, you can apply the redirection to a group compound command
that
15 # contains the subshell compound command to prevent the
optimisation.
16 { (cmd1 arg1; cmd2 arg2) ;} > "xfile$(( i++ ))"
17 declare -p files=( * ) i
18 # declare -a files=[0]="file0" [1]="xfile0"
19 # declare -- i="1"

```

The same problem can be encountered when using redirections like `> "$BASHPID"`, `< "${foo=bar}"`, or `(cmd1 & cmd2 & wait) > >(...)`.

Also see  <https://www.vidarholen.net/contents/blog/?p=865>.

CategoryShell CategoryBashguide

BashPitfalls (последним исправлял пользователь emanuele6 2023-01-05 19:04:41)