

Ловушки для Bash

Эта страница представляет собой сборник распространенных ошибок, допускаемых пользователями bash. В каждом примере есть какие-то недостатки.

Содержание

1. для f в \$(ls * .mp3)
2. cp \$file \$target
3. Имена файлов с начальными тире
4. [\$foo = "bar"]
5. cd \$(имя файла "\$f")
6. ["\$foo" = bar && "\$bar" = foo]
7. [[\$foo > 7]]
8. grep foo bar | во время чтения -r; do ((count++));
готово
9. если [grep foo myfile]
10. если [bar="\$foo"]; тогда ...
11. если [[a = b] && [c = d]]; то ...
12. прочитайте \$foo
13. файл cat | sed s/foo/bar/ > файл
14. echo \$foo
15. \$foo=бар
16. foo = бар
17. эхо <
18. su -с 'некоторая команда'
19. cd /foo; bar
20. [bar == "\$foo"]
21. ибо я в {1..10}; делаю ./что-то &; сделано
22. cmd1 && cmd2 || cmd3
23. эхо "Привет, мир!"
24. для аргументов в \$*
25. функция foo()
26. эхо "~"
27. локальная переменная=\$(cmd)
28. экспорт foo=~ /bar
29. sed's/\$foo/до свидания/'
30. tr [A-Z] [a-z]
31. ps ax | grep gedit
32. printf "\$foo"
33. для i в {1..\$n}
34. если [[\$ foo = \$bar]] (в зависимости от намерения)
35. если [[\$foo =~ 'some RE']]
36. [-n \$foo] или [-z \$foo]
37. [[-e "\$broken_symlink"]] возвращает 1, даже если \$broken_symlink существует
38. ошибка файла редактирования <<<"g/d\{0,3\}/s/e/g"
39. ошибка подстроки exrg для "совпадения"
40. Для UTF-8 и меток порядка байтов (спецификация)
41. содержимое=\$(
42. для файла в .* ; сделать, если [[\$file != *.*]]
43. somescmd 2>&1 >> лог-файл
44. cmd; ((! \$?)) || умри
45. y=\${(массив[\$x])}
46. прочитать число; echo \$((число+1))
47. IFS=, поля для чтения <<< "\$csv_line"
48. экспорт CDPATH=.:~/MyProject
49. OIFS="\$IFS"; ...; IFS= "\$OIFS"
50. хосты=(\$(aws ...))
51. Неатомная запись с помощью xargs -P

```

52. Найти. -exec sh -c 'echo {}' \;
53. sudo mycmd > /myfile
54. sudo ls /foo/*
55. моя программа 2>&-
56. Использование xargs без -0
57. сбросить значение a[0]
58. месяц = $ (дата +% m); день = $ (дата +%d)
59. i=$((10#$i ))
60. набор -euo pipefail
    1. ошибка
    2. обрыв трубы
    3. набор существительных
61. [[ -v хэш[$key] ]]
62. (( хэш [$key]++ ))
63. пока ... сделано <<< "$(foo)"
64. cmd > "файл $((i++))"

```

1. для f в \$(ls *.mp3)

Одна из самых распространенных ошибок, которые совершают программисты BASH, - это писать цикл, подобный этому:

```

для f в $(ls *.mp3); делай # Неправильно!
какая-то команда $f # Неправильная!
сделано

```

```

для f в $(ls) # Неправильно!
для f в `ls` # Неверно!

```

```

для f в $(найти . -введите f) # Неправильно!
для f в `найти . -введите f` # Неправильно!

```

```

файлы=$(найти . -тип f) # Неправильно!
для f в ${files[@]} # Неверно!

```

Да, было бы здорово, если бы вы могли просто обрабатывать выходные данные `ls` или `find` как список имен файлов и выполнять итерации по нему. Но ты не можешь. Весь этот подход фатально ошибочен, и нет никакого трюка, который мог бы заставить его работать. Вы должны использовать совершенно другой подход.

С этим связано как минимум 6 проблем:

1. Если имя файла содержит пробел (или любой символ в текущем значении `$IFS`), оно подвергается разделению на слова. Предполагая, что у нас есть файл с именем `01 - Don't Eat the Yellow Snow.mp3` в текущем каталоге, цикл `for` будет перебирать каждое слово в полученном имени файла: `01`, `-`, `He`, `Есть` и т.д.
2. Если имя файла содержит символы глобуса, оно подвергается расширению имени файла ("глобусу"). Если `ls` выдает какой-либо вывод, содержащий символ `*`, содержащее его слово будет распознано как шаблон и заменено списком всех имен файлов, которые ему соответствуют.
3. Если подстановка команды возвращает несколько имен файлов, невозможно определить, где заканчивается первое и начинается второе. Имена путей могут содержать *любой* символ, кроме NUL. Да, это включает в себя новые строки.
4. Утилита `ls` может изменять имена файлов. В зависимости от того, на какой платформе вы находитесь, какие аргументы вы использовали (или не использовали), и указывает ли его стандартный вывод на терминал или нет, `ls` может случайным образом решить заменить определенные символы в имени

файла на "?" или просто не печатать их вообще. Никогда не пытайтесь анализировать выходные данные `ls`. `ls` просто не нужен. Это внешняя команда, вывод которой предназначен специально для чтения человеком, а не для анализа скриптом.

5. Дополнение `Commands` удаляет все завершающие символы новой строки из своих выходных данных. Это может показаться желательным, поскольку `ls` добавляет новую строку, но если последнее имя файла в списке заканчивается новой строкой, `` . . . `` или `$ ()` также удалят это.
6. В примерах `ls`, если первое имя файла начинается с дефиса, это может привести к ловушке `#3`.

Вы также не можете просто заключить замену в двойные кавычки:

для `f` в `"$(ls *.mp3)"`; делай `#` неправильно!

Это приводит к тому, что весь вывод `ls` обрабатывается как одно слово. Вместо перебора каждого имени файла цикл будет выполняться только один раз, присваивая `f` строку со всеми именами файлов, объединенными вместе.

Вы также не можете просто изменить `IFS` на новую строку. Имена файлов также могут содержать новые строки.

Другой вариант этой темы - злоупотребление разделением слов и циклом `for` для (неправильного) чтения строк файла. Например:

`IFS=$' \n'`
для строки в `$(файл cat)`; `do ... #` неправильно!

Это не работает! Особенно, если эти строки являются именами файлов. `Bash` (или любая другая оболочка семейства `Bourne`) просто так не работает.

Итак, как правильно это сделать?

Есть несколько способов, в первую очередь в зависимости от того, нужно ли вам рекурсивное расширение или нет.

Если вам не нужна рекурсия, вы можете использовать простой глобус. Вместо `ls`:

для файла в формате `./*.mp3`; сделай `#` Лучше! и...
какая-то команда `"$file" # ...` всегда заключает расширения в двойные кавычки!
Выполнено

Оболочки `POSIX`, такие как `Bash`, имеют функцию глобализации специально для этой цели — чтобы позволить оболочке расширять шаблоны в список совпадающих имен файлов. Нет необходимости интерпретировать результаты внешней утилиты.

Поскольку глобализация - это самый последний шаг расширения, каждое совпадение шаблона `./* .mp3` корректно расширяется до отдельного слова и не подвержено эффектам расширения без кавычек.

Вопрос: Что произойдет, если в текущем каталоге нет `*.mp3`-файлов? Затем цикл `for` выполняется один раз с `file="./*.mp3"`, что не является ожидаемым поведением! Обходной путь заключается в проверке наличия соответствующего файла:

```
# POSIX
для файла в ./*.mp3; выполнить
[ -e "$file" ] || продолжить
выполнение некоторой команды "$file"
```

Другим решением является использование функции **shopt -s nullglob** от Bash, хотя это следует делать только после прочтения документации и тщательного рассмотрения влияния этого параметра на все другие глобусы в скрипте.

Если вам нужна рекурсия, стандартное решение - **find**. При использовании **find** убедитесь, что вы используете его правильно. Для переносимости POSIX sh используйте опцию **-exec**:

```
Найти. -введите f -name '*.mp3' -выполните какую-нибудь команду {} \;
# Или, если команда принимает несколько входных имен файлов:
find . -введите f -name '*.mp3' -выполните какую-нибудь команду {} +
```

Если вы используете **bash**, у вас есть два дополнительных варианта. Один из них заключается в использовании опции GNU или BSD **find -print0** вместе с опцией **bash read -d "** и заменой процесса:

```
в то время как IFS= read -r -d " file; выполните
некоторую команду "$file"
< <(find . -введите f -name '*.mp3' -print0)
```

Преимущество здесь в том, что "некоторая команда" (фактически, все тело цикла **while**) выполняется в текущей оболочке. Вы можете установить переменные и сохранить их после завершения цикла.

Другой вариант, доступный в Bash 4.0 и выше, - это **globstar**, который позволяет рекурсивно расширять глобус:

```
shopt -s globstar
для файла в ./**/*.mp3; выполните
какую-нибудь команду "$file"
готово
```

Обратите внимание на двойные кавычки вокруг **\$file** в приведенных выше примерах. Это приводит к нашей второй ловушке:

2. cp \$file \$target

Что не так с командой, показанной выше? Ну, ничего, **если** вы заранее знаете, что в **\$file** и **\$target** нет пробелов (и вы не изменяли **\$IFS**, и вы можете гарантировать, что код не будет вызываться в контексте, где **\$IFS** мог быть изменен) или подстановочных знаков в них. Тем не менее, результаты расширений по-прежнему подлежат разделению слов и расширению пути. Всегда заключайте расширения параметров в двойные кавычки.

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

Без двойных кавычек вы получите команду типа `cp 01 - Не ешьте желтый снег.mp3 /mnt/usb`, что приведет к ошибкам типа `cp: невозможно указать '01': нет такого файла или каталога`. Если в `$file` есть подстановочные знаки (* или ? или []), они будут расширены, если есть файлы, которые им соответствуют. С двойными кавычками все в порядке, если только "`$file`" не начинается с `-`, и в этом случае `cp` думает, что вы пытаетесь передать ему параметры командной строки (см. Ловушку # 3 ниже.)

Даже в тех несколько необычных обстоятельствах, когда вы можете гарантировать содержимое переменной, обычной и хорошей практикой является указывать расширения параметров, особенно если они содержат имена файлов. Опытные авторы сценариев всегда будут использовать кавычки, за исключением, возможно, небольшого числа случаев, когда из непосредственного контекста кода *абсолютно* очевидно, что параметр содержит гарантированное безопасное значение. Эксперты, скорее всего, сочтут, что команда `cp` в названии всегда неверна. Вы тоже должны.

3. Имена файлов с начальными тире

Имена файлов с начальными тире могут вызвать много проблем. Глобусы, такие как `*.mp3`, сортируются в расширенный список (в соответствии с вашим текущим языком) и сортируются перед буквами в большинстве языков. Затем список передается некоторой команде, которая может неправильно интерпретировать `-filename` как параметр. Для этого есть два основных решения.

Одним из решений является вставка `--` между командой (например, `cp`) и ее аргументами. Это говорит ему прекратить сканирование параметров, и все хорошо:

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

С этим подходом есть потенциальные проблемы. Вы должны быть уверены, что вставляете — для *каждого* использования параметра в контексте, где он может быть интерпретирован как опция, что легко пропустить и может привести к большой избыточности.

Большинство хорошо написанных библиотек синтаксического анализа опций понимают это, и программы, которые правильно их используют, должны наследовать эту функцию бесплатно. Тем не менее, имейте в виду, что в конечном итоге приложение должно распознавать *end of options*. Некоторые программы, которые вручную анализируют параметры, или делают это неправильно, или используют плохие сторонние библиотеки, могут не распознать его. Стандартные утилиты *должны*, за несколькими исключениями, которые указаны в POSIX. `echo` - один из примеров.

Другой вариант - убедиться, что ваши имена файлов всегда начинаются с каталога, используя относительные или абсолютные пути.

```
для i в ./*.mp3; сделайте
cp "$ i" / target
...
Выполнено
```

В этом случае, даже если у нас есть файл, имя которого начинается с `-`, глобус гарантирует, что переменная всегда будет содержать что-то вроде `./-foo.mp3`, что

совершенно безопасно с точки зрения **ср**.

Наконец, если вы можете гарантировать, что все результаты будут иметь один и тот же префикс, и используете переменную только несколько раз в теле цикла, вы можете просто объединить префикс с расширением. Это дает теоретическую экономию при создании и сохранении нескольких дополнительных символов для каждого слова.

```
для i в *.mp3; сделайте
ср "$i" /target
...
Выполнено
```

4. [\$foo = "bar"]

Это очень похоже на проблему в pitfall # 2, но я повторяю это, потому что это *очень* важно. В приведенном выше примере кавычки стоят не в том месте. Вам *не* нужно заключать строковый литерал в кавычки в bash (если он не содержит метасимволы или символы шаблона). Но вам *следует* заключать свои переменные в кавычки, если вы не уверены, могут ли они содержать пробелы или подстановочные знаки.

Этот пример может сломаться по нескольким причинам:

- Если переменная, на которую ссылается `[`, не существует или является пустой, то команда `[` в конечном итоге будет выглядеть так:

```
[ = "bar" ] # Неправильно!
```

... и выдаст ошибку: **ожидаемый унарный оператор**. (Оператор `=` является *двоичным*, а не унарным, поэтому команда `[` довольно шокирована, увидев его там.)

- Если переменная содержит внутренние пробелы, то она разбивается на отдельные слова до того, как команда `[` увидит ее. Таким образом:

```
[ несколько слов здесь = "бар" ]
```

Хотя это может показаться вам нормальным, это синтаксическая ошибка, насколько это касается `[`. Правильный способ написать это:

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

Это отлично работает в реализациях, совместимых с POSIX, даже если `$foo` начинается с `-`, потому что POSIX `[` определяет свое действие в зависимости от количества переданных ему аргументов. Только у очень древних оболочек есть проблемы с этим, и вы не должны беспокоиться о них при написании нового кода (см. Обходной путь `X "$foo"` ниже).

В Bash и многих других ksh-подобных оболочках существует превосходная альтернатива, которая использует ключевое слово `[[`.


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

Вам не нужно заключать ссылки на переменные в кавычки с левой стороны `=` в `[[]]`, потому что они не подвергаются разделению на слова или глобированию, и даже пустые переменные будут обрабатываться правильно. С другой стороны, цитирование их тоже ничему не повредит. В отличие от `[` и `test`, вы также можете использовать идентичные `==`. Однако обратите внимание, что сравнения с использованием `[[` выполняют сопоставление шаблона со строкой в правой части, а не просто простое сравнение строк. Чтобы сделать строку в правом литерале, вы должны заключить ее в кавычки, если используются какие-либо символы, которые имеют особое значение в контекстах сопоставления с образцом.

```
# Совпадение Bash / Ksh
=b * r
[[ $foo == "$match" ]] # Хорошо! Без кавычек также будет соответствовать
шаблону b * r.
```

Возможно, вы видели подобный код:

```
# POSIX / Борн
[x"$foo" = xbar ] # Хорошо, но обычно не требуется.
```

Хак `"$foo"` требуется для кода, который должен выполняться на *очень* древних оболочках, в которых отсутствует `[` и есть более примитивный `[`, который  запутывается, если `$foo` начинается с `-` или есть `!` или `(`. В указанных старых системах `[` требуется только дополнительная осторожность для токена левая сторона `=`; он правильно обрабатывает правый токен.

Обратите внимание, что оболочки, требующие этого обходного пути, не соответствуют POSIX. Такая чрезвычайная переносимость редко является обязательным требованием и делает ваш код менее читаемым (и более уродливым).

5. `cd $(имя файла "$f")`

Это еще одна ошибка цитирования. Как и в случае расширения переменной, результат замены команды подвергается разделению на слова и расширению пути. Так что вы должны процитировать это:

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

Что здесь не очевидно, так это то, как вложены кавычки. Программист на C, читающий это, ожидал бы, что первая и вторая двойные кавычки будут сгруппированы вместе; а затем третья и четвертая. Но в Bash это не так. Bash обрабатывает двойные кавычки *внутри* подстановки команд как одну пару, а двойные кавычки *вне* подстановки - как другую пару.

Другой способ написать это: анализатор обрабатывает подстановку команды как "уровень вложенности", а кавычки внутри него отделены от кавычек вне его.

6. `["$foo" = bar && "$bar" = foo]`

Вы не можете использовать `&&` внутри старой команды `test` (или `[`). Анализатор Bash видит `&&` за пределами `[[]]` или `(())` и разбивает вашу команду на *две* команды, до и после `&&`. Вместо этого используйте один из них:

```
[bar = "$foo" ] && [foo = "$bar" ] # Правильно! (POSIX)
[[ $foo = bar && $bar = foo ]] # Тоже правильно! (Bash / Ksh)
```

(Обратите внимание, что мы поменяли местами константу и переменную внутри `[` по устаревшим причинам, рассмотренным в `pitfall # 4`. Мы могли бы также изменить регистр `[`, но расширения потребовали бы цитирования, чтобы предотвратить интерпретацию как шаблон.) То же самое относится и к `|`. Либо используйте `[` вместо этого, либо используйте две команды.

Избегайте этого:

```
[bar = "$foo" -a foo = "$bar" ] # Не переносится.
```

Операторы binary `-a` и `-o` и `(/)` (группировка) являются расширениями XSI стандарта POSIX. Все они помечены как устаревшие в POSIX-2008. Они не должны использоваться в новом коде. Одна из практических проблем с `[A = B -a C = D]` (или `-o`) заключается в том, что POSIX не указывает результаты теста или команды с более чем 4 аргументами. Вероятно, это работает в большинстве оболочек, но вы не можете на это рассчитывать. Если вам нужно писать для оболочек POSIX, то вместо этого вам следует использовать две команды `test` или `[`, разделенные оператором `&&`.

7. `[[$foo > 7]]`

Здесь есть несколько проблем. Во-первых, команда `[[` не должна использоваться исключительно для вычисления арифметических выражений. Его следует использовать для тестовых выражений, включающих один из поддерживаемых операторов тестирования. Хотя технически вы можете выполнять математические вычисления, используя некоторые из операторов `[`, имеет смысл делать это только в сочетании с одним из нематематических тестовых операторов где-нибудь в выражении. Если вы просто хотите выполнить числовое сравнение (или любую другую арифметику оболочки), гораздо лучше просто использовать `(())` вместо этого:

```
# Bash / Ksh
((foo > 7)) # Правильно!
[[ foo -gt 7 ]] # Работает, но встречается редко. Вместо этого
используйте ((...)).
```

Если вы используете оператор `>` внутри `[[]]`, он обрабатывается как сравнение строк (проверка порядка сортировки по локали), а не как сравнение целых чисел. Иногда это может сработать, но это не работает, когда вы меньше всего этого ожидаете. Если вы используете `>` внутри `[]`, это еще хуже: это перенаправление вывода. Вы получите файл с именем `7` в своем каталоге, и тест будет успешным, если `$foo` не является пустым.

Если строгое соответствие POSIX является обязательным требованием, а `(())` недоступен, тогда правильная альтернатива с использованием `[` является

```
# POSIX
[ "$foo" -gt 7 ] # Тоже правильно!
```



```
[ "$((foo > 7))" -ne 0 ] # POSIX-совместимый эквивалент ((, для более
общих математических операций.
```

Если содержимое `$foo` не очищено и находится вне вашего контроля (если, например, они поступают из внешнего источника), то все, кроме `["$foo" -gt 7]`, представляют собой уязвимость для произвольного ввода команды, поскольку содержимое `$foo` интерпретируется как арифметическое выражение (например, арифметическое выражение `a[$(reboot)]` будет запускать команду **перезагрузки** при вычислении). `[builtin` требует, чтобы операнды были десятичными целыми числами, поэтому это не влияет. Но очень важно, чтобы `$foo` был заключен в кавычки, иначе вы все равно получите уязвимость при внедрении команды (например, с такими значениями, как `-v a[$(reboot)] -o 8`).

Если ввод в любой арифметический контекст (включая `(`, `let`, индексы массива) или `[[. . .]]` тестовые выражения, включающие числовые сравнения, не могут быть гарантированы, тогда вы *всегда* должны проверять свой ввод перед вычислением выражения.

```
# POSIX
case $foo в
( "" | *[!0123456789]*)
printf '$foo не является последовательностью десятичных цифр: "%s"\n'
"$foo" >&2
выход 1
;;
*)
[ "$foo" -gt 7 ]
esac
```

Обратите внимание, что, поскольку арифметический оператор `/test` ожидает десятичные целые числа, `010`, например, будет интерпретироваться как число 10, а не 8, выраженное в восьмеричном. В `bash [010 -gt 8]` вернет `true`, в то время как `[[010 -gt 8]]` и `((010 > 8))` вернут `false`.

8. `grep foo bar | во время чтения -r; do ((count++));` **ГОТОВО**

Приведенный выше код на первый взгляд выглядит нормально, не так ли? Конечно, это просто плохая реализация `grep -c`, но она задумана как упрощенный пример. Изменения в `count` не будут распространяться за пределы цикла `while`, потому что каждая команда в конвейере выполняется в отдельной подоболочке. В какой-то момент это удивляет почти каждого новичка в Bash.

POSIX не указывает, оценивается ли последний элемент конвейера в подоболочке. Некоторые оболочки, такие как `ksh93` и `Bash >= 4.2` с включенным `shopt -s lastpipe`, будут запускать цикл `while` в этом примере в исходном процессе оболочки, позволяя вступать в силу любым побочным эффектам внутри. Поэтому переносимые скрипты должны быть написаны таким образом, чтобы не зависеть ни от того, ни от другого поведения.

Для решения этой и подобных проблем, пожалуйста, смотрите Bash FAQ # 24. Это слишком длинно, чтобы поместиться здесь.

9. если [grep foo myfile]

Многие новички имеют неверное представление об операторах `if`, вызванных очень распространенным шаблоном ключевого слова `if`, за которым сразу следует `[` или `[[`. Это убеждает людей в том, что `[` каким-то образом является частью синтаксиса оператора `if`, точно так же, как круглые скобки, используемые в операторе `if` языка C.

Это *не* тот случай! **если** принимает *команду*. `[` является командой, а не синтаксическим маркером для оператора `if`. Это эквивалентно команде `test`, за исключением того, что последним аргументом должен быть `a` `]`. Например:

```
# POSIX
если [ false]; затем echo "HELP"; fi
если test false; затем echo "HELP"; fi
```

эквивалентны — оба проверяют, что аргумент "false" не является пустым. В обоих случаях всегда будет выводиться СПРАВКА, к удивлению программистов с других языков, догадывающихся о синтаксисе оболочки.

Синтаксис оператора `if` является:

```
если КОМАНДЫ
, то <КОМАНДЫ>
если <КОМАНДЫ> # необязательно
, то <КОМАНДЫ>
else <КОМАНДЫ> # необязательный
fi # требуется
```

Еще раз, `[` - это команда. Он принимает аргументы, как и любая другая обычная *простая команда*. `if` — это *составная команда*, которая содержит другие команды - и в ее синтаксисе **нет** `!`

В то время как `bash` имеет встроенную команду `[` и, следовательно, *знает* о `[` это не имеет ничего общего с `]`. `Bash` передает `]` в качестве аргумента только `[` команде, которая требует, чтобы `]` был последним аргументом только для улучшения внешнего вида скриптов.

Может быть ноль или более необязательных разделов `elif` и один необязательный раздел `else`.


Составная команда `if` состоит из двух или более разделов, содержащих *списки команд*, каждая из которых разделена ключевым словом `then`, `elif` или `else`, и завершается ключевым словом `fi`. Статус завершения последней команды первого раздела и каждого последующего раздела `elif` определяет, оценивается ли каждый соответствующий раздел `then`. Другой `elif` вычисляется до тех пор, пока не будет выполнен один из разделов `then`. Если раздел `then` не вычисляется, то берется ветвь `else`, или, если `else` не задано, блок `if` завершен, и общая команда `if` возвращает 0 (true).

Если вы хотите принять решение на основе выходных данных команды **grep**, вы не хотите заключать его в круглые скобки, скобки, обратные знаки или *любой другой синтаксис*! Просто используйте **grep** в качестве **КОМАНД** после **if**, вот так:

```
если grep -q fooregex myfile; тогда
...
fi
```

Если **grep** соответствует строке из **myfile**, то код выхода будет равен 0 (true), и будет выполнена часть **then**. В противном случае, если совпадений нет, **grep** вернет ненулевое значение, а общая команда **if** будет равна нулю.

Смотри также:

- Руководство по башу / Тесты и условия
-  http://wiki.bash-hackers.org/syntax/ccmd/if_clause

10. если [bar="\$foo"]; тогда ...

```
[bar="$foo"] # Неправильно!
[bar="$foo" ] # Все еще неправильно!
[bar = "$foo"] # Тоже неправильно!
[[bar="$foo"]] # Снова неправильно!
[[ bar="$foo" ]] # Угадайте, что? Неправильно!
[[bar = "$foo"]] # Мне действительно нужно это говорить...
```

Как объяснялось в предыдущем примере, **[** - это команда (которую можно проверить с помощью **type -t [** или **откуда -v [**). Как и в случае с любой другой простой командой, Bash ожидает, что за командой будет следовать пробел, затем первый аргумент, затем еще один пробел и т.д. Вы не можете просто запускать все вместе, не вставляя пробелы! Вот правильные способы:

```
если [bar = "$foo" ]; тогда ...

если [[ bar = "$foo" ]]; то ...
```

В первой форме **[** - это имя команды, а **bar**, **=**, расширение **"\$foo"** и **]** - это отдельные аргументы для нее. Между каждой парой аргументов должны быть пробелы, чтобы оболочка знала, где начинается и заканчивается каждый аргумент. Вторая форма аналогична, за исключением того, что **[[** является специальным ключевым словом, которое завершается символом **]]**. Для получения более подробной информации о различии между ними, см. Bash FAQ 31.

11. если [[a = b] && [c = d]]; то ...

Ну вот, опять. **[** это команда. Это не синтаксический маркер, который находится между **if** и каким-то C-подобным "условием". Он также не используется для группировки. Вы не можете взять C-подобные команды **if** и перевести их в команды Bash, просто заменив круглые скобки квадратными скобками!

Если вы хотите выразить составное условное выражение, сделайте это:

```
если [a = b] && [c = d]; то ...
```

Обратите внимание, что здесь у нас есть две *команды* после **if**, к которым присоединяется оператор **&&** (логическое И, вычисление ярлыка). Это точно так же, как:

```
если тест a = b && тест c = d; тогда ...
```

Если первая **ТЕСТОВАЯ** команда возвращает false, тело инструкции **if** не вводится. Если он возвращает true, то выполняется вторая **ТЕСТОВАЯ** команда; и если эта команда также возвращает true, то **будет** введено тело инструкции *if*. (Программисты на C уже знакомы с **&&**. Bash использует ту же самую *оценку короткого замыкания*. Аналогично **||** выполняет оценку короткого замыкания для операции *ИЛИ*.)

Ключевое слово **[[** разрешает использование **&&**, поэтому оно также может быть написано таким образом:

```
если [[ a = b && c = d ]]; то ...
```

Смотрите Ловушку # 6 для ловушки, связанной с *тестами* в сочетании с условными операторами.

12. прочитайте \$foo

Вы не используете **\$** перед именем переменной в команде **ЧТЕНИЯ**. Если вы хотите поместить данные в переменную с именем **foo**, вы делаете это следующим образом:

```
читать foo
```

Или более безопасно:

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

read \$foo прочитает строку ввода и поместит ее в переменную (переменные), имена которых указаны в **\$foo**. Это может быть полезно, если вы на самом деле предполагали, что **foo** будет ссылкой на какую-то другую переменную; но в большинстве случаев это просто ошибка.

13. файл cat | sed s/foo/bar/ > файл

Вы не **можете** читать из файла и записывать в него в одном конвейере. В зависимости от того, что делает ваш конвейер, файл может быть заблокирован (до 0 байт или, возможно, до количества байт, равного размеру буфера конвейера вашей операционной системы), или он может увеличиваться до тех пор, пока не заполнит доступное дисковое пространство, или не достигнет ограничения размера файла вашей операционной системы, или вливания квота и т.д.

Если вы хотите внести изменения в файл безопасно, кроме добавления в конец, используйте текстовый редактор.

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

Если вы делаете что-то, что невозможно сделать с помощью текстового редактора, в какой-то момент *должен* быть создан временный файл (*). Например, следующее полностью переносимо:

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

Следующее будет работать *только* на GNU sed 4.x:

```
файл (ы) sed -i's/foo/bar/g'
```

Обратите внимание, что это также создает временный файл и выполняет тот же обман с переименованием — он просто обрабатывает его прозрачно.

И для следующей эквивалентной команды требуется perl 5.x:

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

Для получения более подробной информации о замене содержимого файлов, пожалуйста, смотрите Bash FAQ # 21.

(*) **sponge** из  moreutils использует этот пример в своем руководстве:

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

Вместо того, чтобы использовать временный файл плюс атомарный **mv**, эта версия "впитывает" (фактическое описание в руководстве!) все данные, перед открытием и записью в **файл**. Эта версия приведет к потере данных, если программа или система выйдет из строя во время операции записи, потому что в этот момент на диске нет копии исходного файла.

Использование временного файла + **mv** по-прежнему сопряжено с небольшим риском потери данных в случае сбоя системы / потери питания; чтобы быть на 100% уверенным, что старый или новый файл выдержит потерю питания, вы должны использовать **синхронизацию** перед **mv**.

14. echo \$foo

Эта относительно невинно выглядящая команда вызывает *массовую* путаницу. Поскольку **\$foo** не заключен в кавычки, он будет подвергаться не только разделению слов, но и глобированию файлов. Это вводит в заблуждение программистов Bash, заставляя их думать, что их переменные *содержат* неправильные значения, когда на самом деле с переменными все в порядке — просто разделение слов или расширение имени файла портят их представление о том, что происходит.

```
Сообщение="Пожалуйста, введите имя файла формы *.zip"
echo $ msg
```

Это сообщение разбивается на слова, и любые глобусы расширяются, например, *.zip . Что подумают ваши пользователи, когда увидят это сообщение:

```
Пожалуйста, введите имя файла формы freenfss.zip lw35nfss.zip
```

Чтобы продемонстрировать:

```
var="*.zip" # var содержит звездочку, точку и слово "zip"
echo "$var" # записывает *.zip
echo $var # записывает список файлов, которые заканчиваются на .zip
```

На самом деле, команда **echo** не может быть использована здесь с абсолютной безопасностью. Например, если переменная содержит **-n**, **echo** будет считать это параметром, а не данными для печати. Единственный абсолютно *надежный* способ вывести значение переменной - это использовать **printf**:

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

15. \$foo=бар

Нет, вы не присваиваете переменную, помещая **\$** перед именем переменной. Это не perl.

16. foo = bar

Нет, вы не можете ставить пробелы вокруг **=** при присвоении переменной. Это не C. Когда вы пишете **foo = bar**, оболочка разбивает его на три слова. В качестве имени команды берется первое слово **foo**. Второй и третий становятся аргументами этой команды.

Аналогично, следующее также неверно:

```
foo= bar # НЕПРАВИЛЬНО!
foo =bar # НЕПРАВИЛЬНО!
$foo = bar; # СОВЕРШЕННО НЕПРАВИЛЬНО!

foo=bar # Вправо.
foo="bar" # Правее.
```

17. эхо <

Документ **here** является полезным инструментом для встраивания больших блоков текстовых данных в скрипт. Это вызывает перенаправление строк текста в скрипте на стандартный ввод команды. К сожалению, **echo** - это не команда, которая считывается из **stdin**.

```
# Это неправильно:
эхо <
Привет, мир
Как дела?
EOF

# Это то, что вы пытались сделать:
cat <
Привет, мир
Как дела?
EOF
```

```
# Или используйте кавычки, которые могут занимать несколько строк
(эффективно, встроено echo):
эхо "Привет, мир
Как дела?"
```

Такое использование кавычек прекрасно — оно отлично работает во всех оболочках, но оно не позволяет вам просто вставить блок строк в скрипт. В первой и последней строке есть синтаксическая разметка. Если вы хотите, чтобы синтаксис оболочки не затрагивал ваши строки, и не хотите создавать команду **cat**, вот еще одна альтернатива:

```
# Или используйте printf (также эффективно, printf встроен):
printf %s "\
Привет, мир
Как дела?
"
```

В примере `printf` символ `\` в первой строке предотвращает дополнительный перевод строки в начале текстового блока. В конце есть буквальный перевод строки (потому что последняя цитата находится на новой строке). Отсутствие `\n` в аргументе формата `printf` не позволяет **printf** добавлять дополнительный перевод строки в конце. `\` `trick` не будет работать в одинарных кавычках. Если вам нужны / нужны одинарные кавычки вокруг блока текста, у вас есть два варианта, оба из которых требуют, чтобы синтаксис оболочки "загрязнял" ваши данные:

```
printf %s \
"Привет, мир
"

printf %s "Привет, мир
"
```

18. `su -c` 'некоторая команда'

Этот синтаксис *почти* правильный. Проблема в том, что на многих платформах **SU** принимает аргумент `-C`, но это не тот, который вам нужен. Например, на OpenBSD:

```
$ su -c 'эхо-привет'
su: только суперпользователь может указать класс входа
```

Вы хотите передать `-c` 'некоторую команду' в оболочку, что означает, что вам нужно имя пользователя перед `-C`.

```
su root -c 'некоторая команда' # Теперь все правильно.
```

SU принимает имя пользователя `root`, когда вы опускаете его, но это становится очевидным, когда вы хотите передать команду в оболочку позже. В этом случае вы должны указать имя пользователя.

19. `cd /foo; bar`

Если вы не проверите наличие ошибок в команде **cd**, вы можете в конечном итоге выполнить **bar** в неправильном месте. Это может стать серьезной катастрофой, если, например, **bar** окажется **rm -f ***.

Вы **всегда** должны проверять наличие ошибок в команде **cd**. Самый простой способ сделать это:

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

Если после **cd** есть более одной команды, вы можете предпочесть это:

```
cd /foo || выход из 1
бара
baz
bat ... # Много команд.
```

cd сообщит о невозможности изменения каталогов с помощью сообщения stderr, такого как "bash: cd: /foo: нет такого файла или каталога". Однако, если вы хотите добавить свое собственное сообщение в стандартный вывод, вы можете использовать группировку команд:

```
cd /net || { echo >&2 "Не удастся прочитать /net. Убедитесь, что вы
вошли в сеть Samba, и повторите попытку."; выход 1; }
делай_стафф
мо_стафф
```

Обратите внимание, что между **{** и **echo** есть обязательный пробел, а перед закрытием обязательно **;** **}**. Вы также можете написать функцию **die**, если хотите.

Некоторым людям также нравится разрешать **set -e** прерывать свои скрипты по *любой* команде, которая возвращает ненулевое значение, но это может быть довольно сложно использовать правильно (поскольку многие распространенные команды могут возвращать ненулевое значение для условия предупреждения, которое вы, возможно, не захотите рассматривать как фатальное).

Кстати, если вы часто меняете каталоги в скрипте Bash, обязательно ознакомьтесь с справкой Bash по **pushd**, **popd** и **dirs**. Возможно, весь тот код, который вы написали для управления **cd** и **pwd**, совершенно не нужен.

Говоря об этом, сравните это:

```
найдите ... -введите d -print0 | в то время как IFS= read -r -d "
subdir; сделайте
здесь = $ PWD
cd "$subdir" && что угодно && ...
cd "$here"
готово
```

С помощью этого:

```
найдите ... -введите d -print0 | в то время как IFS= read -r -d "
subdir; do
(cd "$subdir" || exit; что угодно; ...)
Выполнено
```


Принудительное использование подболочки здесь приводит к тому, что **cd** встречается только в подболочке; для следующей итерации цикла мы возвращаемся к нашему обычному местоположению, независимо от того, был ли **cd** успешным или неудачным. Нам не нужно вносить изменения вручную, и мы не застряли в бесконечной цепочке `... && ...` логики, препятствующей использованию других условных выражений. Версия с подболочками проще и чище (хотя и немного медленнее).

Другой подход заключается в том, чтобы **безоговорочно** перемещаться туда, где мы должны быть, в начале каждой итерации цикла:

```
здесь = $PWD
найти ... -введите d -print0 | в то время как IFS= read -r -d " subdir;
сделать
cd "$here" || продолжить
cd "$subdir" || продолжить
что угодно
...
ГОТОВО
```

По крайней мере, таким образом, мы можем **перейти** к следующей итерации цикла и не должны составлять бесконечную последовательность `&&` вместе, чтобы гарантировать, что мы достигнем **cd** в конце тела цикла.

20. `[bar == "$foo"]`

Оператор `==` недопустим для команды POSIX `[`. Вместо этого используйте `=` или ключевое слово `[[`.

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

В Bash `["$X" == y]` принимается как расширение, что часто заставляет программистов Bash думать, что это правильный синтаксис. Это не так; это багизм. Если вы собираетесь использовать Bashisms, вы могли бы просто использовать `[[` вместо этого.

21. **ибо я в {1..10}; делаю ./что-то &; сделано**

Вы не можете поставить `;` сразу после `&`. Просто удалите посторонний `;` полностью.

```
для i в {1..10}; do ./что-то и сделано
```

Или:

```
для i в {1..10}; do
./что-то и
сделано
```

`&` уже функционирует как командный терминатор, точно так же, как `;` делает. И вы не можете смешивать два.

В общем, символ `;` может быть заменен новой строкой, но не все новые строки могут быть заменены на `;`.

22. `cmd1 && cmd2 || cmd3`

Некоторые люди пытаются использовать `&&` и `||` в качестве сокращенного синтаксиса для `if ... then ... else ... fi`, возможно, потому, что они думают, что они умны. Например,

```
# НЕПРАВИЛЬНО!
[[ -s $errorlog ]] && echo "0, были некоторые ошибки". || echo "Успешно".
```

Однако эта конструкция *не* полностью эквивалентна `if ... fi` в общем случае. Команда, которая идет после `&&`, также генерирует статус выхода, и если этот статус выхода не "true" (0), то **также** будет вызвана команда, которая идет после `||`. Например:

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

Что здесь произошло? Похоже, что **Я** должен быть 1, но в итоге получается 0. Почему? Потому что оба `i ++` и `i --` были выполнены. Команда `((i ++))` имеет статус выхода, и этот статус выхода выводится из C-подобной оценки выражения внутри круглых скобок. Значение этого выражения равно 0 (начальное значение `i`), а в C выражение с целочисленным значением 0 считается *ложным*. Итак, `((i ++))` (когда `i` равно 0) имеет статус выхода 1 (false), и поэтому команда `((i - -))` также выполняется.

Другой умный человек думает, что мы можем исправить это, используя оператор предварительного увеличения, поскольку статус выхода из `++i` (с `i` изначально 0) имеет значение true:

```
i=0
true && (( ++i )) || (( --i )) # ВСЕ ЕЩЕ НЕПРАВИЛЬНО!
echo "$i" # Выводит 1 по глупой случайности
```

Но это упускает суть примера. Это просто случайно сработало, и вы не *можете* полагаться на `X && Y || Z`, если у `Y` есть какие-**либо** шансы на сбой! (Этот пример по-прежнему завершается неудачей, если мы инициализируем `i` значением -1 вместо 0.)

Если вам нужна безопасность, или если вы просто не уверены, как это работает, или если что-то в предыдущих параграфах было не совсем понятно, пожалуйста, просто используйте простой синтаксис `if ... fi` в своих программах.

```
i=0
, если true; тогда
((i++))
ещё
((i--))
fi
echo "$i" # Выводит 1
```

Этот раздел также относится к Bourne shell, вот код, который его иллюстрирует:

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

Вывод - это две строки "true" и "false", вместо одной строки "true".

23. эхо "Привет, мир!"

Проблема здесь в том, что в интерактивной оболочке Bash (в версиях до 4.3) вы увидите ошибку типа:

```
bash: !": событие не найдено
```

Это связано с тем, что в настройках интерактивной оболочки по умолчанию Bash выполняет расширение истории в стиле csh с использованием восклицательного знака. Это **не** проблема в сценариях оболочки; только в интерактивных оболочках.

К сожалению, очевидная попытка "исправить" это не сработает:

```
$ echo "привет \!"  
привет!
```

Самое простое решение - отключить параметр histexpand: это можно сделать с помощью **set +H** или **set +o histexpand**

Вопрос: Почему игра с **histexpand** более уместна, чем одинарные кавычки?

Я лично столкнулся с этой проблемой, когда манипулировал файлами песен, используя такие команды, как

```
mp3info -t "Не показывай этого" ...  
mp3info -t "Ах! Лия!" ...
```

Использование одинарных кавычек крайне неудобно из-за того, что все песни имеют апострофы в названиях. Использование двойных кавычек столкнулось с проблемой расширения истории. (И представьте файл, в названии которого есть оба. Цитирование было бы чудовищным.)

Поскольку я никогда не использую расширение истории, моим личным предпочтением было отключить его в ~/.bashrc. — GreyCat

Эти решения будут работать:

```
эхо "Привет, мир!"
```

или

```
эхо "Привет, мир"!
```

или

```
set +H  
повторяет "Привет, мир!"
```

или

```
histchars=
```

Многие люди просто предпочитают указывать **set +H** или **set +o histexpand** в своем `~/ .bashrc`, чтобы навсегда отключить расширение истории. Однако это личное предпочтение, и вы должны выбрать то, что подходит вам лучше всего.

Другое решение:

```
exmark='!'
эхо "Привет, мир $exmark"
```

В Bash 4.3 и новее, двойная кавычка ниже **!** больше не запускает расширение истории, но расширение истории по-прежнему выполняется в двойных кавычках, поэтому, хотя **echo "Hello World!"** в порядке, это все равно будет проблемой:

```
эхо "Привет, мир!(и остальной Вселенной)"
эхо "foo!'bar'"
```

24. для аргументов в \$*

Bash (как и все оболочки Bourne) имеет специальный синтаксис для обращения к списку позиционных параметров по одному за раз, и **\$ *** не так ли. Ни один из них не является **\$@**. Оба они расширяются до списка слов в параметрах вашего скрипта, а не до каждого параметра как отдельного слова.

Правильный синтаксис:

```
для аргументов в "$@"

# Или просто:
для аргументов
```

Поскольку циклическое изменение позиционных параметров является обычным делом в скриптах, **для arg** по умолчанию используется значение **for arg** в **"\$ @"**. Двойные кавычки **"\$ @"** - это особая магия, которая заставляет использовать каждый параметр как одно слово (или одну итерацию цикла). Это то, что вы должны использовать не менее 99% времени.

Вот пример:

```
# Неверная версия
для x в $*; сделать
эхо "параметр: '$x'"
Выполнено

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

Это должно было быть написано:

```
# Правильная версия
для x в "$@"; параметр "выполнить
эхо": '$ x'"
Выполнено
# или лучше:
для x сделайте
echo "параметр: '$ x'"
Выполнено

$ ./myscript 'arg 1' arg2 arg3
параметр: 'аргумент 1'
параметр: 'arg2'
параметр: 'arg3'
```

25. функция foo()

Это работает в некоторых оболочках, но не в других. Вы *никогда* не должны комбинировать ключевое слово **function** со скобками **()** при определении функции.

Bash (по крайней мере, некоторые версии) позволит вам смешивать два. Большинство оболочек не примут это (например, zsh 4.x и, возможно, выше). Некоторые оболочки будут принимать **функцию foo**, но для максимальной переносимости вы всегда должны использовать:

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

26. эхо "~"

Расширение Тильды применяется только тогда, когда '~' не заключен в кавычки. В этом примере echo записывает '~' в стандартный вывод, а не путь к домашнему каталогу пользователя.

Цитирование параметров пути, которые выражаются относительно домашнего каталога пользователя, должно выполняться с помощью \$HOME, а не '~'. Например, рассмотрим ситуацию, когда \$HOME - это "/ home / мои фотографии".

```
"~/ dir с пробелами" # расширяется до "~/ dir с пробелами"
~ "/ dir с пробелами" # расширяется до "~/ dir с пробелами"
~/ "каталог с пробелами" # расширяется до "/ home / мои фотографии /
каталог с пробелами"
"$HOME / каталог с пробелами" # расширяется до "/home/ мои фотографии /
каталог с пробелами"
```

27. локальная переменная=\$(cmd)

При объявлении локальной переменной в функции **локальная** переменная действует как самостоятельная команда. Иногда это может странно взаимодействовать с остальной частью строки — например, если вы хотите получить статус выхода (\$?) из CommandSubstitution, вы не можете этого сделать. статус выхода **local** маскирует его.

Еще одна проблема с этим синтаксисом заключается в том, что в некоторых оболочках (например, bash) **локальный** `var=$(cmd)` обрабатывается как *присваивание*, то есть правая часть обрабатывается особым образом, точно так же, как `var= $(cmd)`; в то время как в других оболочках **локальный** `var= $(cmd)` **не** рассматривается как присвоение, и правая часть будет разделена на слова (потому что она не заключена в кавычки).

Цитирование правой части поможет обойти проблему разделения слов, но не проблему маскировки статуса выхода. По обеим причинам для этого лучше использовать отдельные команды:

Переключить отображение номеров строк

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

Обе проблемы также актуальны для **экспорта** и только для **чтения**.

Следующая ошибка описывает другую проблему с этим синтаксисом:

28. экспортировать `foo=~ /bar`

Расширение тильды (с именем пользователя или без него) гарантированно происходит только тогда, когда тильда появляется в начале слова, либо сама по себе, либо после косой черты. Это также гарантированно происходит, когда тильда появляется сразу после `=` в присваивании.

Однако команды `export` и `local` **не** обязательно представляют собой назначение. В некоторых оболочках (например, bash) `export foo=~ /bar` будет подвергаться расширению тильды; в других этого не произойдет.

```
foo=~ /bar; экспортировать foo # Вправо!
экспортировать foo="$HOME /bar" # Правильно!
```

Использование расширения параметров вместо расширения тильды позволяет использовать дальнейшее улучшение, чтобы:

```
экспорт foo="${HOME%}/bar" # Лучше!
```

Который выдает `/bar` вместо `//bar`, когда `$HOME` равен `/` (что раньше было обычным для пользователя `root`, по крайней мере), что лучше, поскольку в некоторых системах пути, начинающиеся с двойной косой черты, имеют особое значение.

29. `sed 's/$foo/до свидания/'`

В одинарных кавычках расширения параметров bash, такие как `$foo`, не расширяются. Это цель одинарных кавычек, чтобы защитить символы, такие как `$`, от оболочек.

Измените кавычки на двойные кавычки:

```
foo="привет"; sed "s/$foo/до свидания /"
```

Но имейте в виду, что если вы используете двойные кавычки, вам может потребоваться использовать больше экранирований. Смотрите страницу цитат.

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

Здесь есть (как минимум) три ошибки. Первая проблема заключается в том, что `[Az]` и `[az]` рассматриваются оболочкой как глобусы. Если у вас нет одинарных имен файлов в вашем текущем каталоге, вам будет казаться, что команда верна; но если вы это сделаете, все пойдет не так. Вероятно, в 0300 часов в выходные.

Вторая проблема заключается в том, что это не совсем правильное обозначение для `tr`. Что это на самом деле делает, так это переводит '[' в '['; все, что находится в диапазоне от `Az` до `az`; и `]` в `]`. Таким образом, вам даже не нужны эти скобки, и первая проблема исчезнет.

Третья проблема заключается в том, что в зависимости от локали, `A-Z` или `a-z` могут не дать вам 26 символов ASCII, которые вы ожидали. На самом деле, в некоторых языках `z` находится в середине алфавита! Решение этой проблемы зависит от того, что вы хотите, чтобы произошло:

```
# Используйте это, если вы хотите изменить регистр 26 латинских букв
LC_COLLATE=C tr A-Z a-z

# Используйте это, если вы хотите, чтобы преобразование регистра зависело
от языкового стандарта, который может быть больше похож на то, что
пользователь ожидает
от '[:upper:]' '[:lower:]'
```

Кавычки требуются для второй команды, чтобы избежать дублирования.

31. `ps ax | grep gedit`

Основная проблема здесь заключается в том, что имя запущенного процесса по своей сути ненадежно. Может быть более одного легитимного процесса `gedit`. Может быть что-то еще, маскирующееся под `gedit` (изменение сообщаемого имени выполняемой команды является тривиальным). *Реальные* ответы на этот вопрос см. в разделе *Управление процессами*.

Ниже приведены быстрые и грязные вещи.

Поиск PID (например) `gedit`, многие люди начинают с

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

что, в зависимости от условий гонки, часто приводит к получению самого `grep`. Чтобы отфильтровать `grep`:

```
ps ax | grep -v grep | grep gedit # будет работать, но некрасиво
```

Альтернативой этому является использование:

```
ps ax | grep '[g]редактировать' # цитировать, чтобы избежать глобуса
оболочки
```

Это будет игнорировать сам `grep` в таблице процессов, поскольку это `[g]edit`, и `grep` ищет `gedit` после оценки.

В GNU / Linux параметр `-C` может использоваться вместо этого для фильтрации по имени команды:

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

Но зачем беспокоиться, если вместо этого можно просто использовать `pgrep`?

```
$ pgrep gedit
10530
```

Теперь на втором шаге PID часто извлекается с помощью `awk` или `cut`:

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

но даже это может быть обработано некоторыми из триллионов параметров для `ps`:

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

Если вы застряли в 1992 году и не используете `pgrep`, вместо этого вы можете использовать древний, устаревший, устаревший `pidof` (только для GNU / Linux):

```
$ pidof gedit
10530
```

и если вам нужен PID для завершения процесса, `pskill` может быть вам интересен. Обратите внимание, однако, что, например, `pgrep / pskill ssh` также найдет процессы с именем `sshd`, и вы не захотите их уничтожить.

К сожалению, некоторые программы не запускаются с их именем, например, `firefox` часто запускается как `firefox-bin`, что вам нужно будет выяснить с помощью — well — `ps ax | grep firefox`. 😊 Или вы можете придерживаться `pgrep`, добавив некоторые параметры:

```
$ pgrep -fl firefox
3128 /usr/библиотека/firefox/firefox
7120 /usr/lib/firefox/плагин-контейнер /usr/lib/flashplugin-
установщик/libflashplayer.so -greomni /usr/lib/firefox/omni.ja 3128
истинный плагин
```

Пожалуйста, ознакомьтесь с `ProcessManagement`. Серьезно.

32. printf "\$foo"

Это неправильно не из-за кавычек, а из-за использования строки формата. Если `$foo` не находится строго под вашим контролем, то любые символы `\` или `%` в переменной

могут вызвать нежелательное поведение.

Всегда указывайте свою собственную строку формата:

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

33. для *i* в {1..\$n}

BashParser выполняет расширение *скобок перед* любыми другими расширениями или заменами. Таким образом, код расширения фигурных скобок видит литерал `$ n`, который не является числовым, и поэтому он не расширяет фигурные скобки в список чисел. Это делает практически невозможным использование расширения фигурных скобок для создания списков, размер которых известен только во время выполнения.

Сделайте это вместо:

```
для ((i=1; i<=n; i++)); делать  
...  
Выполнено
```

В случае простой итерации по целым числам, для начала почти всегда следует предпочесть арифметический цикл `for` расширению в фигурных скобках, потому что расширение в фигурных скобках предварительно расширяет каждый аргумент, который может быть медленнее и излишне потребляет память.

34. если `[[$foo = $bar]]` (в зависимости от намерения)

Когда правая часть оператора `=` внутри `[[` не заключена в кавычки, `bash` выполняет сопоставление с шаблоном вместо того, чтобы рассматривать его как строку. Итак, в приведенном выше коде, если `bar` содержит `*`, результат *всегда* будет `true`. Если вы хотите проверить равенство строк, правая часть должна быть заключена в кавычки:

```
если [[ $foo = "$bar" ]]
```

Если вы хотите выполнить сопоставление с образцом, было бы разумно выбрать имена переменных, которые указывают, что правая часть содержит шаблон. Или используйте комментарии.

Также стоит отметить, что если вы заключаете в кавычки правую часть `=~`, это *также* приводит к простому сравнению строк, а не к сопоставлению регулярных выражений. Это приводит нас к:

35. если `[[$foo =~ 'some RE']]`

Кавычки вокруг правой части оператора `=~` превращают его в строку, а не в обычное выражение. Если вы хотите использовать длинное или сложное регулярное выражение и избежать большого количества экранирования обратной косой черты, поместите его в переменную:

```
re='некоторый RE'  
, если [[ $foo =~ $re ]]
```

Это также работает с учетом разницы в том, как `=~` работает в разных версиях `bash`. Использование переменной позволяет избежать некоторых неприятных и тонких проблем.

Та же проблема возникает с сопоставлением шаблонов внутри `[[:`:

```
[[ $foo = "*.glob" ]] # Неправильно! *.glob обрабатывается как
литеральная строка.
[[ $foo = *.глоб ]] # Правильно. *.глоб рассматривается как шаблон в
стиле глобуса.
```

36. `[-n $foo]` или `[-z $foo]`

При использовании команды `[` вы **должны** заключать в кавычки каждую замену, которую вы ей задаете. В противном случае `$foo` может расшириться до 0 слов, или 42 слов, или любого количества слов, отличного от 1, что нарушает синтаксис.

```
[ -n "$foo" ]
[ -z "$foo" ]
[ -n "$(некоторая команда с "$file" в ней)" ]

# [[ не выполняет разделение слов или расширение глобуса, поэтому вы
также можете использовать:
[[ -n $foo ]]
[[ -z $foo ]]
```

37. `[[-e "$broken_symlink"]]` возвращает 1, даже если `$broken_symlink` существует

Тест следует символическим ссылкам, поэтому, если символическая ссылка повреждена, т. Е. Указывает на файл, который не существует или находится в каталоге, к которому у вас нет доступа, `test -e` возвращает для него 1, даже если он существует.

Чтобы обойти это (и подготовиться к этому), вы должны использовать:

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


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

38. `ed` файл `<<<"g/d\{0,3\}/s//e/g"` завершается ошибкой

Проблема возникла из-за того, что `ed` не принимает 0 для `\{0,3\}`.

Вы можете проверить, работают ли следующие:

```
файл редактирования <<<"g/d\{1,3\}/s//e/g"
```

Обратите внимание, что это происходит, даже если POSIX утверждает, что BRE (который является разновидностью регулярного выражения, используемого `ed`)  должен принимать 0 в качестве минимального числа вхождений (см. Раздел 5).

39. ошибка подстроки expr для "совпадения"

В большинстве случаев это работает достаточно хорошо

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

Но не получится для слова "совпадение"

```
слово = сопоставить
выражение "$word" : "\(.*\)"
```

Проблема в том, что "совпадение" - это ключевое слово. Решение (только для GNU) имеет префикс с '+'

```
слово = сопоставить
выражение + "$word" : "\(.*\)"
атч
```


Или, знаете, перестаньте использовать **expr**. Вы можете делать все, что делает **expr**, используя расширение параметров. Что эта штука там пытается сделать? Удалить первую букву слова? Это можно сделать в оболочках POSIX, используя расширение параметров или расширение подстроки:

```
$ word=совпадение
$ echo "${word#?}" # Расширение параметров
при запуске
$ echo "${word:1}" # Расширение подстроки
в начале
```

Серьезно, нет никакого оправдания для использования **expr**, если вы не используете Solaris с его **/bin /sh**, не соответствующим POSIX. Это внешний процесс, поэтому он намного медленнее, чем обработка строк в процессе. И поскольку никто не использует его, никто не понимает, что он делает, поэтому ваш код запутан и его сложно поддерживать.

40. Для UTF-8 и меток порядка байтов (спецификация)

В общем: текст Unix UTF-8 не использует спецификацию. Кодировка обычного текста определяется локализацией, типами `mime` или другими метаданными. Хотя наличие спецификации обычно не повредит документу UTF-8, предназначенному только для чтения людьми, это проблематично (часто синтаксически незаконно) в любом текстовом файле, предназначенном для интерпретации автоматизированными процессами, такими как скрипты, исходный код, файлы конфигурации и так далее. Файлы, начинающиеся со спецификации, следует считать одинаково чужеродными, как и файлы с разрывами строк MS-DOS.

В сценариях оболочки: "Там, где UTF-8 используется прозрачно в 8-разрядных средах, использование спецификации будет мешать любому протоколу или файловому формату, который ожидает определенные символы ASCII в начале, например, использование `"#!"` в начале сценариев оболочки Unix". 
http://unicode.org/faq/utf_bom.html#bom5

41. содержимое=\$(

В этом выражении нет ничего плохого, но вы должны знать, что замены команд (все формы: `...`, \$(...), \$(Это часто несущественно или даже желательно, но если вы должны сохранить буквальный вывод, включая любые возможные завершающие новые строки, это становится сложным, потому что у вас нет способа узнать, были ли они в выводе или сколько. Один уродливый, но полезный обходной путь - добавить постфикс внутри подстановки команды и удалить его снаружи:

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

Менее переносимым, но, возможно, более привлекательным решением является использование **read** с пустым разделителем.

```
# Ksh (или bash 4.2+ с включенным lastpipe)
ссылка для чтения -fn -- "$dir_path" | IFS= read -rd " абсолютный путь к
папке
```

Недостатком этого метода является то, что **ЧТЕНИЕ** всегда будет возвращать false, если команда не выводит нулевой байт, вызывающий чтение только части потока.

Единственный способ получить статус завершения команды - через **PIPESTATUS**.

Вы также можете намеренно вывести нулевой байт, чтобы принудительное **ЧТЕНИЕ** возвращало true, и использовать **pipefail**.

```
установить -o pipefail
{ readlink -fn -- "$dir_path" && printf '\0'; } | IFS= read -rd
"абсолютный_дир_путь
установлен +o файл конвейера
```

Это своего рода проблема с переносимостью, поскольку Bash поддерживает как **pipefail**, так и **PIPESTATUS**, ksh93 поддерживает только **pipefail**, и только последние версии mksh поддерживают **pipefail**, в то время как более ранние версии поддерживали только **PIPESTATUS**. Кроме того, требуется расширенная версия ksh93, чтобы **ЧТЕНИЕ** остановилось на нулевом байте.

42. для файла в ./* ; сделать, если [[\$file != *.*]]

Один из способов помешать программам интерпретировать имена файлов, переданные им в качестве опций, - это использовать имена путей (см. Ловушку № 3 выше). Для файлов в текущем каталоге имена могут иметь префикс относительного пути **./**.

Однако в случае шаблона, подобного ***.***, могут возникнуть проблемы, поскольку он соответствует строке вида **./filename**. В простом случае вы можете просто использовать глобус напрямую для генерации желаемых совпадений. Однако, если требуется отдельный этап сопоставления с образцом (например, результаты были предварительно обработаны и сохранены в массиве, и их необходимо отфильтровать), проблему можно решить, приняв во внимание префикс в шаблоне: **[[\$file != ./*.*]]**, или с помощью удаления шаблона из матча.

```
# Bash
shopt -s nullglob
```

```

для пути в ./*; сделать
[[ ${путь##*/} != *.* ]] && rm "$path"
готово

# Или даже лучше
для файла в *; do
[[ $file != *.* ]] && rm "$file"
готово

# Или еще лучше
для файла в *.*; do
rm "$file"
готово

```


Другая возможность - сигнализировать об *окончании опций* с помощью аргумента `--`. (Опять же, описано в pitfall #3).

```

shopt -s nullglob
для файла в *; делать
[[ $file != *.* ]] && rm -- "$file"
готово

```

43. `somcmd 2>&1 >>лог-файл`

Это, безусловно, самая распространенная ошибка, связанная с перенаправлениями. Как правило, кто-то, желающий направить как `stdout`, так и `stderr` в файл или канал, попытается это и не поймет, почему `stderr` все *еще* отображается на их терминале. Если вы озадачены этим, вы, вероятно, не понимаете, как работают  перенаправления или, возможно, файловые дескрипторы для начала. Перенаправления оцениваются слева направо перед выполнением команды. Этот семантически некорректный код по сути означает: "сначала перенаправьте стандартную ошибку туда, куда в данный момент указывает `standard out` (tty), затем перенаправьте `standard out` в `logfile`". Это наоборот. Стандартная ошибка уже передается в `tty`. Вместо этого используйте следующее:

```
somcmd >> лог-файл 2>&1
```

Посмотреть более подробное объяснение,  Описание копирования объясненои BashGuide - перенаправление.

44. `cmd; ((! $?)) || умри`

`$?` требуется только в том случае, если вам нужно получить точный статус предыдущей команды. Если вам нужно проверить только на успех или неудачу (любой ненулевой статус), просто протестируйте команду напрямую. например:

```

если cmd; то
...
фи

```

Проверка статуса выхода по списку альтернатив может проходить по следующей схеме:

```

статус cmd=$?
case $status в
0)

```

```

echo success >&2
;;
1)
echo 'Должен указать параметр при выходе.' >&2
выход 1
;;
*)
эхо "Неизвестная ошибка $status, выход". > &2
выйдите из "$status"
esac

```

45. `y=$((массив[$x]))`

Эта ошибка возникает в любом контексте, в котором имя и индекс массива подвергаются расширению до слов, прежде чем они будут оценены как литеральное имя. Ловушки 61 и 62 являются следствием одних и тех же механизмов.

Код, заданный для арифметического расширения или составной команды, проходит начальный проход расширений и замен, чтобы сгенерировать текст, который будет проанализирован и оценен как арифметическое выражение. С *этим нужно обращаться осторожно*.

Например, это выражение объединяется путем расширения одного фрагмента кода в другой.

```

$ x = '$ (date > & 2)' # перенаправление предназначено только для того,
чтобы мы могли видеть, как все происходит
$ y=$((array["$ x"])) # Кавычки не помогают. Массив даже не должен
существовать
Пн, 2 июня, 10:49:08 EDT 2014

```

Затем расширенная строка передается арифметическому процессору, который должен будет получить ссылку на переменную массива во внутренней таблице символов оболочки, используя функцию поиска, чтобы разрешить "имя" переменной. Этот распознаватель имен принимает строку - `массив[$(date >&2)$]`, состоящий из имени, включая индекс и весь буквальный код в скобках, точно так же, как, например, `read` или `printf -v` делают с именами переменных, передаваемыми в качестве аргументов. Распознаватель переменных выполняет расширения, *включая подстановку команд*, для разрешения индекса.

(Для числовых индексированных массивов функция поиска затем вычисляет расширенный текст индекса как арифметическое выражение. Следовательно, взаимно рекурсивный поиск переменных и арифметические расширения могут происходить на любую глубину (вплоть до определенного предела Bash), любой из которых может привести к непреднамеренным побочным эффектам.)

В большинстве случаев нет необходимости использовать какое-либо расширение в рамках арифметического расширения. Используйте имена переменных непосредственно в выражении (без `$`) везде, где это возможно (т. Е. За исключением позиционных параметров и "специальных переменных" POSIX). Проверяйте переменные перед их использованием и убедитесь, что расширение не генерирует ничего, кроме числового литерала - большинство проблем автоматически устраняются.

Экранируйте любые расширения, чтобы передать их в выражение, не расширяя их сначала:

Переключить отображение номеров строк

```

1 # Типичная задача, считывающая некоторые столбцы в ассоциативный массив.

```

```

набираем 2 -A arr
printf 3 -v смещение '%(% s) T' -1
4
ЕСЛИ , в то время как 5= ' ' read -r x y; делаем
6 # проверка ввода (см. Следующую ошибку)
7 [[ "$ x $y" == +([0123456789])' ' +([0123456789]) ]] || продолжить
8 # Экранированная подстановка передает все выражение буквально.
9 (( arr[\$(date -d "@$x" +%F)] = y - смещение ))
10 сделано

```

Другой вариант - использовать **let** с аргументами, заключенными в одинарные кавычки. **((выражение))** эквивалентно **let "выражение"** (аргументы в двойных кавычках).

46. прочитайте num; echo \$((num+1))

Всегда проверяйте свои входные данные (см. BashFAQ / 054) перед использованием **num** в арифметическом контексте, поскольку это позволяет вводить код.

```

$ echo 'a[$(эхо-инъекция >&2)]' | bash -c 'считывает число; echo $((num
+1))'
инъекция
1

```

47. IFS=, поля для чтения <<< "\$csv_line"

Каким бы невероятным это ни казалось, POSIX требует обработки IFS как *ограничителя* поля, а не *разделителя* полей. В нашем примере это означает, что если в конце строки ввода есть пустое поле, оно будет удалено:

```

$ IFS=, поля для чтения <<< "a,b,"
$ объявить -р полей
объявить -а полей='([0]="a" [1]="b")'

```

Куда делось пустое поле? Его ели по историческим причинам ("потому что так было всегда"). Это поведение не уникально для **bash**; все соответствующие оболочки делают это. Непустое поле сканируется правильно:

```

$ IFS=, поля для чтения <<< "a,b,c"
$ объявить -р полей
объявить -а полей='([0]="a" [1]="b" [2]="c")'

```

Итак, как нам обойти эту чушь? Как оказалось, добавление символа IFS в конец входной строки заставит сканирование работать. Если в конце было пустое поле, дополнительный символ IFS "завершает" его, чтобы оно сканировалось. Если в конце было непустое поле, символ IFS создает новое пустое поле, которое удаляется.

```

$ input="a,b,"
$ IFS=, поля для чтения <<< "$input,"
$ объявить -р полей
объявить -а полей='([0]="a" [1]="b" [2]="")'

```

Та же проблема может возникнуть, когда мы используем **read** с фиксированным количеством переменных поля, и мы хотим получить остальную часть строки без изменений в конечную переменную. Снаряд мешает нам.

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

Завершающий `:` был удален. Это происходит только тогда, когда в остальной части строки есть ровно *один* дополнительный разделитель, и это последний символ. Во всех остальных случаях ловушка остается неиспользованной.

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

Это позволяет проблеме оставаться незамеченной целую *вечность*, прежде чем, наконец, нанести удар.

Чтобы обойти это поведение, мы можем добавить дополнительный символ-разделитель в конец строки, а затем удалить его позже.

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

48. экспортировать CDPATH=.:~/MyProject

Не экспортируйте CDPATH.

Установка CDPATH в `.bashrc` не является проблемой, но ее экспорт приведет к тому, что любой запускаемый вами скрипт `bash` или `sh`, который использует `cd`, потенциально изменит поведение.

Есть две проблемы. Скрипт, который выполняет следующее:

```
cd some/dir || выход
из cmd для запуска в some/dir
```

может изменить каталог на `~/MyProject/some/dir` вместо `./some/dir`, в зависимости от того, какие каталоги существуют в данный момент. Таким образом, **КОМПАКТ**-диск может завершиться успешно и перенести скрипт в неправильный каталог с потенциально вредными последствиями следующих команд, которые теперь выполняются в другом каталоге, чем предполагалось.

Вторая проблема заключается в том, что `cd` запускается в контексте, в котором записывается вывод:

```
вывод=$(cd some/dir && некоторая команда)
```

В качестве побочного эффекта, когда установлен CDPATH, `cd` выведет что-то вроде `/home/user/some/dir` в стандартный вывод, чтобы указать, что он нашел каталог через CDPATH, который, в свою очередь, окажется в выходной переменной вместе с предполагаемым выводом **некоторой команды**.

Скрипт может сделать себя невосприимчивым к CDPATH, унаследованному из среды, всегда добавляя `./` к относительным путям, или выполнить `unset CDPATH` в начале скрипта, но не думайте, что каждый сценарист учел эту ошибку, поэтому не экспортируйте CDPATH.

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

Прямого присвоения значения переменной временной переменной недостаточно для восстановления ее состояния. Присваивание всегда будет приводить к *установленной*, но *пустой* временной переменной, даже если начальная переменная не была установлена. Это особая проблема для IFS, потому что *пустой* IFS имеет совершенно другое значение, чем *неустановленный* IFS, и установка IFS на временное значение для одной или двух команд является общим требованием.

Простой обходной путь - назначить префикс, чтобы отличать set от unset vars, а затем удалить его, когда закончите.

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

Локальная переменная обычно предпочтительнее, когда это возможно.

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

Подоболочки - еще одна возможность.

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

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

Небезопасно заполнять массив необработанной заменой `$(...)`

CommandSubstitution. Вывод команды подвергается разделению слов (на *все* пробелы, даже те, которые находятся внутри кавычек), а затем глобализируется. Если есть такое слово, как `*` или `eh?` или `[abc]` в результате он будет расширен на основе имен файлов в текущем рабочем каталоге.

Чтобы выбрать замену, вам нужно знать, записывает ли команда свои выходные данные в одну строку или в несколько строк. Если это одна строка:

```
прочитайте -ra хосты < <(aws ...)
```


Если это несколько строк (и вы ориентируетесь на bash 4.0 или более позднюю версию):

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

Если это несколько строк (и вы хотите совместимости с bash 3.x *или* хотите, чтобы статус завершения вашей команды отражался в успешном или неудачном выполнении операции **ЧТЕНИЯ**, независимо от поведения, доступного только в bash 4.4 и новее):

Эта команда содержит уязвимость CodeInjection. Имя файла, найденное с помощью **find**, вводится в команду оболочки и анализируется **sh**. Если имя файла содержит метасимволы оболочки, такие как **;** или **\$(...)**, то имя файла может быть выполнено как код с помощью **'sh'**.

Пример "slowprint" в предыдущей ловушке был бы ошибкой CodeInjection, если бы входные данные не были гарантированно целыми числами.

Чтобы быть более точным,  POSIX **find** не указывает, расширяется ли аргумент, содержащий *больше, чем* просто **{}**. GNU **find** допускает это кодовое внедрение. Другие реализации выбирают более безопасный путь:

```
# uname -a
HP-UX imadev B.10.20 A 9000/785 2008897791 лицензия для двух
пользователей
# найти /dev/null -exec sh -c 'echo {}' \;
{}
```

Правильный подход заключается в том, чтобы *отделить* аргумент filename от аргумента script:

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

53. sudo mycmd > /myfile

Перенаправление выполняется *до* выполнения команды. Обычно это не имеет значения, но с **sudo** у нас есть команда, выполняемая от имени другого пользователя, а не перенаправление.

Если перенаправление должно выполняться с правами, предоставленными **sudo**, тогда вам нужна оболочка:

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

Вместо обертки вы можете использовать **tee**:

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

Это может быть проще написать, если в **mycmd** много цитат.

54. sudo ls /foo/*

Это очень похоже на предыдущую ловушку. Глобализация также выполняется *перед* выполнением команды. Если каталог недоступен для чтения с вашими обычными правами пользователя, вам может потребоваться выполнить глобализацию в оболочке с правами, предоставленными **sudo**:

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

55. моя программа 2>&-

Не закрывайте `stdin`, `stdout` или `stderr` как "сокращение" для перенаправления в `/dev/null`. Напишите это правильно.

```
моя программа 2>/dev/null
```

Почему? Рассмотрим, что происходит, когда ваша программа пытается записать сообщение об ошибке в `stderr`. Если `stderr` был перенаправлен в `/dev/null`, запись завершается успешно, и ваша программа может продолжать работу, будучи уверенной в том, что она добросовестно сообщила об ошибке. условие.

Но если `stderr` был *закрыт*, то запись завершится ошибкой. В этот момент ваша программа может сделать что-то непредсказуемое. Он может продолжать и игнорировать сбой, или он может немедленно завершиться, учитывая, что среда выполнения настолько нарушена, что она не может безопасно продолжаться. Или что еще, по мнению программиста, должна делать программа, когда ее мир превратился в мрачный ад.

Все программы уверены, что `stdin`, `stdout` и `stderr` будут *существовать* и будут доступны для чтения / записи надлежащим и разумным образом. Закрыв один из них, вы нарушили свое обещание этой программе. Это неприемлемо.

Конечно, еще лучшим решением было бы на самом деле регистрировать ошибки где-нибудь, чтобы вы могли вернуться, прочитать их и выяснить, что не так.

56. Использование `xargs` без `-0`

`xargs` разбивается на пробелы. Это прискорбно, потому что пробелы разрешены в именах файлов и обычно используются пользователями GUI. `xargs` также обрабатывает `"` и `"` специально, что также может привести к проблемам:

```
прикоснись к папиному "знаменитому" 1" pizza.txt
прикоснись к папе 12" records.txt
сенсорный 2 "x1" wood.txt
сенсорный 2 "x4" wood.txt
```

Здесь `xargs` предупреждает:

```
# Не делайте этого
$ find . -введите f | xargs wc
xargs: непревзойденные одинарные кавычки; по умолчанию кавычки являются
специальными для xargs, если вы не используете параметр -0
```

Здесь `xargs` вообще не предупреждает:

```
# Не делайте этого
echo * | xargs wc
find *famous* -введите f | xargs wc
find *4* -введите f | xargs wc
```

Вместо этого используйте `xargs -0`:

```
# Сделайте это вместо
printf '%s\0' * | xargs -0 wc
find . -введите f -name '*famous*' -print0 | xargs -0 wc
find . -введите f -name '*4*' -exec wc {} +
```

Если использовать `-0` непросто, альтернативой является использование GNU Parallel, которое разбивается на `\n`. И хотя `\n` также разрешено в именах файлов, они никогда не встречаются, если только ваши пользователи не являются злонамеренными. В любом случае: **если** вы используете `xargs` без `-0`, добавьте комментарий в свой код, объясняющий, почему это безопасно в вашей конкретной ситуации.

57. сбросить значение `a[0]`

При передаче элемента индексированного массива в `unset` его необходимо заключить в кавычки. В противном случае он может рассматриваться как глобус и расширяться до файлов в текущем каталоге. Если случайно существует файл с именем `a0`, то глобус расширится до `a0`, и вы в конечном итоге выполняете `unset a0`.

Также обычно предпочтительнее указывать параметр `-V` при сбросе переменной. В противном случае, если переменная, указанная в аргументе `unset`, не существует, и, оказывается, существует функция с таким именем, команда `unset` отменит настройку функции вместо того, чтобы просто ничего не делать. Похоже, этого не происходит с `bash unset`, если аргумент содержит `[. . .]` (вам нужно явно использовать параметр `-f` для отмены функций, в имени которых есть `[. . .]`), но это недокументировано, поэтому лучше всего использовать `-V` в любом случае, даже при сбросе элемента массива.

```
unset -v 'a[0]' # Всегда заключать в кавычки элементы индексированного массива при сбросе.
```

58. `месяц = $ (дата +% m); день = $ (дата +%d)`

Называть **дату** несколько раз - плохая идея. Представьте, что произойдет, если первый вызов произойдет за миллисекунду до полуночи 30 апреля, а второй вызов произойдет через миллисекунду после полуночи 1 мая. В итоге вы получите `month = 04` и `day = 01`.

Лучше вызвать `date` *один* раз, извлекая все нужные поля за один вызов.

Обычная идиома для этого:

```
вычисление "$(дата +'месяц=%m день=%d год=%Y имя_дня="%A"
имя_месяца="%B"' )"
```

Или с помощью встроенного `bash` (4.2 или выше) `printf`:

```
printf -v d'$(месяц=%m день=%d год=%Y имя_дня="%A" имя_месяца="%B")Т
' вычислить "$d"
```

Помните, что такие вещи, как названия месяцев или дней, зависят от локали, поэтому кавычки вокруг `%A` или `%B`, чтобы избежать проблем в языках, где названия дней или месяцев содержат пробелы или другие специальные символы для оболочки.

Или вы можете получить временную метку в формате эпохи (секунды с начала 1970 года), а затем использовать ее для создания удобочитаемых полей даты / времени по мере необходимости.

```
# Требуется версия bash 4.2 или выше
printf -v now '%(%s)T' -1 # Или now= $ EPOCHSECONDS в версии bash 5.0
# -1 может быть опущен в 4.3 или выше
printf -v месяц '%(%m)T' "$now"
printf -v день '%(%d)T' "$сейчас"
```

Если `strftime()` вашей системы не поддерживает `%S`, вы можете получить время эпохи с помощью:

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

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

Принудительная интерпретация базы 10 работает только с числами без знаков. Пока `$i` содержит строку цифр без ведущих `-` или `+`, все в порядке. Но если значение `$i` может быть отрицательным, это преобразование может завершиться с ошибкой, либо с шумом (с сообщением об ошибке), либо, что еще хуже, беззвучно (просто давая неправильный результат).

Если есть какой-либо шанс, что `$i` может быть отрицательным, используйте это вместо:

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

Пояснения см. в разделе `ArithmeticExpression`.

60. набор `-euo pipefail`

Есть много подводных камней при включении этих параметров в начале скрипта.

60.1. ошибка

`errexit (set -e)` пытается прервать скрипт при возникновении ошибки, что на первый взгляд кажется хорошей идеей, но у него очень сложные правила относительно того, когда прерывать работу по ошибке или нет. Некоторые из основных проблем с `errexit`

- На самом деле оболочка не может обнаруживать ошибки. Все, что для этого нужно, - это статус завершения команды. Когда команды терпят неудачу, они обычно возвращают ненулевой статус выхода, но многие команды также используют статус выхода для передачи значения `true / false`. Примерами таких команд являются `test`, `,`, `[[...]]`, `((...))`, и `grep`.
- Когда команда, которую вы тестируете с помощью `if` или `&&` или `||`, является функцией, `set -e` игнорирует ненулевые статусы завершения команд в этой функции. Рассмотрим функцию, подобную

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

Если эта команда `cd` завершается с ошибкой, вы определенно не хотите, чтобы эта команда `rm` выполнялась, и при простом использовании функции, когда

включен `errexit`, это так и есть:

```
set -e
cleanup() {
  cd "$1"
  printf 'Ой!\n'
}
очистки / нет / больше / там
# scriptname: cd: /no/longer/there: нет такого файла или каталога
```

Но позже вы решите добавить пользовательское сообщение об ошибке

```
очистки /больше /нет /там || {
  printf >&2 'Ошибка очистки \n'
  выход 1
}
# scriptname: cd: /no/longer/there: нет такого файла или каталога
# Ой!
```

`cd` запускается в контексте, в котором он тестируется, даже если это совсем не очевидно в определении функции, где находится строка. Правильный способ, с `errexit` или без него, - это явно проверить, произошел сбой `cd` или нет

```
# Правильная
очистка() {
  cd "$ 1" || возвращает
  rm -f ./*
}
```

- Замена команды отключает `errexit`. Рассмотрим следующий пример функции, которая имитирует команду `realpath`:

```
set -e
realdir() {
  cd "$1"
  pwd -P
}
realdir /нет / такого / реж
# имя_скрипта: cd: /no/such/dir
```

Он прерван при сбое команды `cd`, но если вы попытаетесь захватить ее вывод, сбой `cd` будет проигнорирован


```
dir=$(realdir /no/such/dir)
#имя_скрипта: cd: /no/such/dir
printf 'dir - это <%s>\n' "$dir"
# dir - это </home/user>
```

Правильный способ - снова явно проверить, произошел сбой `cd` или нет, независимо от того, используете ли вы `errexit`

```
# Right
realdir() {
  cd "$1" || возвращает
  pwd -P
}
```

Посмотреть

- Почему `set -e` (или `set -o errexit`, или `trap ERR`) не делает то, что я ожидал ?.

-  Проблема с полупредикатами

60.2. ошибка в трубе

Обычно статус завершения конвейера - это статус завершения последней команды этого конвейера. С помощью `pipefail`, если какая-либо часть конвейера возвращает ненулевое значение, весь конвейер возвращает ненулевое значение. Это может показаться хорошей идеей, но на самом деле это нормально, когда команды ранее в конвейере выходят ненулевыми, и это не является ошибкой. Рассмотрим следующий пример

```
если some_command | grep -q foo; тогда
printf 'foo найдено \n'
, иначе
printf 'foo не найдено\n'
fi
```


Если включен `pipefail`, вышеприведенный файл иногда будет указывать "foo не найден", даже если `foo` был найден.

`grep -q foo` возвращает 0, чтобы указать, что хотя бы одна строка соответствует шаблону, иначе он возвращает 1, чтобы указать, что ни одна строка не соответствует шаблону. Поскольку функция `-q` (quiet) приводит к тому, что `grep` не выводит соответствующие строки, на самом деле ему не нужно читать весь поток; достаточно найти хотя бы одну строку, соответствующую шаблону. Таким образом, типичная реализация `grep` завершится со статусом 0, как только будет найдена первая совпадающая строка.

`some_command` на другом конце конвейера не будет знать этого. Он видит только свой конец трубы, он не видит, что находится на другом конце. Когда он пытается записать следующий фрагмент данных в канал, а канал закрыт на другом конце, система отправит `some_command` сигнал `SIGPIPE`, чтобы сообщить ему, что он больше не может выполнять запись в канал. Действие по умолчанию, которое необходимо предпринять при получении `SIGPIPE`, - это выйти с ненулевым статусом. Это означает, что весь конвейер возвращает ненулевое значение, и выполняется ветвь `else`, даже если она успешно нашла соответствующую строку.

Если вы знаете, что каждая часть конвейера, кроме первой, будет потреблять весь их ввод, `pipefail` безопасен и часто является хорошим выбором, но его не следует включать глобально, включать только для конвейеров, где это имеет смысл, и отключать впоследствии.

60.3. набор существительных

набор существительных (`set -u`) является наименее плохим из трех вариантов, но также имеет свою долю ошибок. Если цель состоит в том, чтобы перехватить имена переменных с опечатками,  `shellcheck` лучше справляется с их обнаружением.

Посмотрите, каковы преимущества и недостатки использования `set -u` (или `set -o nounset`)?

61. [[-v хэш[\$key]]]

Здесь **ХЭШ** - это ассоциативный массив. Эта конструкция завершается с ошибкой, потому что `$key` расширяется перед вычислением индекса массива, а затем весь

массив плюс расширенный индекс вычисляется во втором проходе. Кажется, что это работает для простых ключей, но это не сработает для ключей, содержащих кавычки, закрывающие квадратные скобки и т. Д. Что еще хуже, он вводит CodeInjection, если `$key` содержит синтаксис CommandSubstitution .

Та же проблема относится к командам `test` и `[` , а также к любому использованию ассоциативного элемента массива в арифметическом контексте.

В более новых версиях `bash` (5.0 и выше) есть опция `assoc_expand_once`, которая подавляет множественные оценки. Другой вариант - заключить весь аргумент в одинарные кавычки:


```
[[ -v 'хэш[$key]' ]]
```

Это имеет то преимущество, что работает как в более старых версиях `bash`, так и в более новых версиях.

62. ((хэш[\$key]++))

Сюрприз! Ассоциативные массивы еще *более* разбиты в математических контекстах. Трюка с одинарными кавычками из предыдущей ловушки также недостаточно, чтобы исправить это.

```
$ declare -x хэш
$ key='\['
$ hash[$key]=17
$ (( хэш [$key]++ ))
bash: ((: hash[']++: неверный индекс массива (токен ошибки - "hash[']++"
)
$ (( 'хэш[$key]++' ))
bash: ((: 'hash[']++' : синтаксическая ошибка: ожидаемый операнд (токен
ошибки - "'hash['] ++'")
```

 По словам Чета Рами, одинарные кавычки здесь не работают, потому что содержимое математического контекста обрабатывается так, как будто оно заключено в двойные кавычки, и поэтому одинарные кавычки не имеют особого значения. До выпуска `bash` 5.2 обратная косая черта все еще сохраняла здесь свое особое значение и могла использоваться для цитирования индекса:

```
(( хэш[\$key]++ )) # Безопасность перед ударом 5.2.
```

Начиная с версии `bash` 5.2, **единственным** безопасным и работающим способом изменения элемента ассоциативного массива при вычислении является создание временной копии значения в обычной (строковой) переменной.

```
tmp=${хэш[$ключ]}
((tmp++)) # Безопасно.
хэш[$key]=$tmp

# Или:
tmp=${hash[$key]}
хэш[$key]=$((tmp+1)) # Безопасно.
```

Я настоятельно рекомендую использовать временную переменную, даже если вы "уверены", что ваш скрипт никогда не будет запущен в версии `bash` 5.2 или новее. Кто-то может неожиданно установить `bash` 5.2. Нет причин использовать старые, более рискованные обходные пути.

Перенаправления для простых команд могут выполняться в подоболочке, поэтому значение \$i в следующем примере может не измениться в основной оболочке (оно не изменится в bash и ksh, но оно будет в dash):

Переключить отображение номеров строк

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

Запуск перенаправлений в подоболочке (т.е. После форка) выполняется в качестве оптимизации; это избавляет оболочку от необходимости восстанавливать перенаправленный файловый дескриптор после завершения команды.

Вы можете использовать следующие альтернативы, чтобы предотвратить эту оптимизацию:

Переключить отображение номеров строк

```
1  #!/bin/bash --
shopt 2 -s nullglob
3
  я 4=0
  объявляю 5 -p files=( * ) я
6 # объявляю -a files=()
7 # объявляю -- i="0"
8
9 # Используйте временную переменную, чтобы убедиться, что файл $((i
++ )) расширяется в
10 # правильной оболочке.
файл 11=file$((i++ ))
12 cmd arg > "$file"
  объявить 13 -p files=( * ) i
14 # объявить -a files=[0]="file0"
15 # объявить -- i="1"
16
17 # Или применить перенаправление на групповую составную команду, в
  которой только
18 # содержит простую команду.
19 {cmd arg ;} > "file$((i++ ))"
  объявить 20 -p i files=( * )
21 # объявить -a files=[0]="file0" [1]="file1"
22 # объявить -- i="2"
```

Bash также применяет эту оптимизацию к составным командам подоболочек:

Переключить отображение номеров строк

```
1  #!/bin/bash --
shopt 2 -s nullglob
3
  i 4=0
  объявить 5 -p файлов=( * ) i
6 # объявить -a файлов=()
7 # объявить -- i="0"
8
```

```

9 (cmd1 arg1; cmd2 arg2) > "файл $((i ++ ))"
объявить 10 -p файлов =( * ) i
11 # объявить -a files=[0]="file0"
12 # объявить -- i="0"
13
14 # Опять же, вы можете применить перенаправление к групповой
составной команде, которая
содержит 15 #составная команда подоболочки для предотвращения
оптимизации.
16 { (cmd1 arg1; cmd2 arg2) ;} > "xfile $((i++ ))"
объявить 17 -p файлов=( * ) i
18 # объявить файлы =[0]="file0" [1]="xfile0"
19 # объявить -- i="1"

```

Та же проблема может возникнуть при использовании перенаправлений типа > "\$BASHPID", < "\${foo=bar}" или (cmd1 & cmd2 & wait) > > (...).

Также смотрите  <https://www.vidarholen.net/contents/blog/?p=865> .

Категория Shell CategoryBashguide

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