

[Каталог документации](#) / [Раздел "Программирование, языки"](#) / [Оглавление документа](#)

Advanced Bash-Scripting Guide: Искусство программирования на языке сценариев командной оболочки

[Назад](#)

[Вперед](#)

## Глава 3. Служебные символы

Служебные символы, используемые в текстах сценариев.

#

**Комментарии.** Строки, начинающиеся с символа # (за исключением комбинации #!) -- являются комментариями.

```
# Эта строка -- комментарий.
```

Комментарии могут располагаться и в конце строки с исполняемым кодом.

```
echo "Далее следует комментарий." # Это комментарий.
```

Комментариям могут предшествовать [пробелы](#) (пробел, табуляция).

```
# Перед комментарием стоит символ табуляции.
```



Исполняемые команды не могут следовать за комментарием в той же самой строке. Пока что еще не существует способа отделения комментария от "исполняемого кода", следующего за комментарием в той же строке.



Само собой разумеется, экранированный символ # в операторе **echo** не воспринимается как начало комментария. Более того, он может использоваться в [операциях подстановки параметров](#) и в [константных числовых выражениях](#).

```
echo "Символ # не означает начало комментария."  
echo 'Символ # не означает начало комментария.'  
echo Символ \# не означает начало комментария.  
echo А здесь символ # означает начало комментария.
```

```
echo ${PATH#*:}          # Подстановка -- не комментарий.  
echo $(( 2#101011 ))    # База системы счисления -- не  
                        комментарий.
```

```
# Спасибо, S.C.
```

Кавычки " ' и \ экранируют действие символа #.

В операциях **поиска по шаблону** символ # так же не воспринимается как начало комментария.

;

**Разделитель команд.** [Точка-с-запятой] Позволяет записывать две и более команд в одной строке.

```
echo hello; echo there
```

Следует отметить, что символ ";" иногда так же как и # необходимо **экранировать**.

;;

**Ограничитель в операторе выбора case** . [Двойная-точка-с-запятой]

```
case "$variable" in
abc)  echo "$variable = abc" ;;
xyz)  echo "$variable = xyz" ;;
esac
```

.

**команда "точка".** Эквивалент команды **source** (см. **Пример 11-18**). Это **встроенная** команда **bash**.

.

**"точка"** может являться частью имени файла . Если имя файла начинается с точки, то это "скрытый" файл, т.е. команда **ls** при обычных условиях его не отображает.

```
bash$ touch .hidden-file
bash$ ls -l
total 10
-rw-r--r--  1 bozo      4034 Jul 18 22:04 data1.addressbook
-rw-r--r--  1 bozo      4602 May 25 13:58 data1.addressbook.bak
-rw-r--r--  1 bozo       877 Dec 17  2000 employment.addressbook
```

```
bash$ ls -al
total 14
drwxrwxr-x  2 bozo  bozo      1024 Aug 29 20:54 ./
drwx----- 52 bozo  bozo      3072 Aug 29 20:51 ../
-rw-r--r--  1 bozo  bozo      4034 Jul 18 22:04 data1.addressbook
-rw-r--r--  1 bozo  bozo      4602 May 25 13:58 data1.addressbook.bak
-rw-r--r--  1 bozo  bozo       877 Dec 17  2000 employment.addressbook
-rw-rw-r--  1 bozo  bozo         0 Aug 29 20:54 .hidden-file
```

Если подразумевается имя каталога, то *одна точка* означает текущий каталог и *две точки* -- каталог уровнем выше, или родительский каталог.

```
bash$ pwd
/home/bozo/projects
```

```
bash$ cd .
bash$ pwd
/home/bozo/projects
```

```
bash$ cd ..
bash$ pwd
/home/bozo/
```

Символ *точка* довольно часто используется для обозначения каталога назначения в операциях копирования/перемещения файлов.

```
bash$ cp /home/bozo/current_work/junk/* .
```

Символ "точка" в операциях поиска. При выполнении **поиска по шаблону** , в **регулярных выражениях**, символ "точка" обозначает одиночный символ.

**Двойные кавычки** . В строке *"STRING"*, ограниченной двойными кавычками не выполняется интерпретация большинства служебных символов, которые могут находиться в строке. см. **Глава 5**.

**Одинарные кавычки** . [Одинарные кавычки] *'STRING'* экранирует все служебные символы в строке *STRING*. Это более строгая форма экранирования. Смотрите так же **Глава 5**.

**Запятая** . Оператор **запятая** используется для вычисления серии арифметических выражений. Вычисляются все выражения, но возвращается результат последнего выражения.

```
let "t2 = ((a = 9, 15 / 3))" # Присваивает значение переменной "a" и
вычисляет "t2".
```

**escape.** [обратный слэш] Комбинация `\X` "экранирует" символ `X`. Аналогичный эффект имеет комбинация с "одинарными кавычками", т.е. `'X'`. Символ `\` может использоваться для экранирования кавычек `"` и `'`.

Более детальному рассмотрению темы экранирования посвящена [Глава 5](#).

/

**Разделитель, используемый в указании пути к каталогам и файлам.** [слэш] Отделяет элементы пути к каталогам и файлам (например `/home/bozo/projects/Makefile`).

В **арифметических операциях** `--` это оператор деления.

\

**Подстановка команд.** [обратные кавычки] **Обратные кавычки** могут использоваться для записи в переменную команды `'command'`.

:

**пустая команда.** [двоеточие] Это эквивалент операции "NOP" (*no op*, нет операции). Может рассматриваться как синоним встроенной команды **true**. Команда `:"` так же является встроенной командой Bash, которая всегда **возвращает** `"true"` (0).

```
:
echo $?    # 0
```

Бесконечный цикл:

```
while :
do
    operation-1
    operation-2
    ...
    operation-n
done

# То же самое:
# while true
# do
#     ...
# done
```

Символ-заполнитель в условном операторе `if/then`:

```
if condition
then :    # Никаких действий не производится и управление передается дальше
else
    take-some-action
fi
```

Как символ-заполнитель в операциях, которые предполагают наличие двух операндов, см. [Пример 8-2](#) и [параметры по-умолчанию](#).

```
: ${username=`whoami`}
# ${username=`whoami`} без символа : выдает сообщение об ошибке,
# если "username" не является командой...
```

Как символ-заполнитель для оператора [вложенного документа](#). См. [Пример 17-9](#).

В операциях с [подстановкой параметров](#) (см. [Пример 9-13](#)).

```
: ${HOSTNAME?} ${USER?} ${MAIL?}
# Вывод сообщения об ошибке, если одна или более переменных не определены.
```


В операциях [замены подстроки с подстановкой значений переменных](#).

В комбинации с оператором > ([оператор перенаправления вывода](#)), усекает длину файла до нуля. Если указан несуществующий файл -- то он создается.

```
: > data.xxx # Файл "data.xxx" -- пуст
# Тот же эффект имеет команда cat /dev/null >data.xxx
# Однако в данном случае не производится создание нового процесса,
# поскольку ":" является встроенной командой.
```

См. так же [Пример 12-11](#).

В комбинации с оператором >> -- оператор перенаправления с добавлением в конец файла и обновлением времени последнего доступа (: >> [new\\_file](#)). Если задано имя несуществующего файла, то он создается. Эквивалентно команде [touch](#).

 Вышеизложенное применимо только к обычным файлам и неприменимо к конвейерам, символическим ссылкам и другим специальным файлам.

Символ : может использоваться для создания комментариев, хотя и не рекомендуется. Если строка комментария начинается с символа #, то такая строка не проверяется интерпретатором на наличие ошибок. Однако в случае оператора : это не так.

```
: Это комментарий, который генерирует сообщение об ошибке, ( if [ $x -eq 3 ]
).
```

Символ ":" может использоваться как разделитель полей в /etc/passwd и переменной [\\$PATH](#).

```
bash$ echo $PATH
/usr/local/bin:/bin:/usr/bin:/usr/X11R6/bin:/sbin:/usr/sbin:/usr/games
```

!

**инверсия (или логическое отрицание)** используется в условных операторах. Оператор **!** инвертирует **код завершения** команды, к которой он применен. (см. [Пример 6-2](#)). Так же используется для логического отрицания в операциях сравнения, например, операция сравнения "равно" ( **=** ), при использовании оператора отрицания, преобразуется в операцию сравнения -- "не равно" ( **!=** ). Символ **!** является зарезервированным ключевым словом BASH.

В некоторых случаях символ **!** используется для **косвенного обращения к переменным**.

Кроме того, из *командной строки* оператор **!** запускает *механизм историй* Bash (см. [Приложение F](#)). Примечательно, что этот механизм недоступен из сценариев (т.е. исключительно из командной строки).

\*

**символ-шаблон.** [звездочка] Символ **\*** служит "шаблоном" для **подстановки** в имена файлов. Одиночный символ **\*** означает любое имя файла в заданном каталоге.

```
bash$ echo *
abs-book.shtml add-drive.sh agram.sh alias.sh
```

В **регулярных выражениях** токен **\*** представляет любое количество (в том числе и 0) символов.

\*

**арифметический оператор.** В арифметических выражениях символ **\*** обозначает операцию умножения.

Двойная звездочка (два символа звездочки, следующих подряд друг за другом -- **\*\***), обозначает операцию **возведения в степень**.

?

**Оператор проверки условия.** В некоторых выражениях символ **?** служит для проверки выполнения условия.

В **конструкциях с двойными скобками**, символ **?** подобен трехместному оператору языка C. См. [Пример 9-28](#).

В выражениях с **подстановкой параметра**, символ **?** проверяет -- установлена ли **переменная**.

?

**символ-шаблон.** Символ `?` обозначает одиночный символ при **подстановке** в имена файлов. В **регулярных выражениях** служит для обозначения **одиночного символа**.

`$`

### **Подстановка переменной.**

```
var1=5
var2=23skidoo

echo $var1      # 5
echo $var2      # 23skidoo
```

Символ `$`, предшествующий имени переменной, указывает на то, что будет получено *значение* переменной.

`$`

**end-of-line (конец строки).** В **регулярных выражениях**, символ `"$"` обозначает конец строки.

`${}`

### **Подстановка параметра.**

`$*`, `$@`

**параметры командной строки.**

`$?`

**код завершения.** Переменная `$?` хранит **код завершения** последней выполненной команды, **функции** или сценария.

`$$`

**id процесса.** Переменная `$$` хранит *id процесса* сценария.

`()`

**группа команд.**

```
(a=hello; echo $a)
```



Команды, заключенные в *круглые скобки* исполняются в дочернем процессе -- **subshell-e**.

Переменные, создаваемые в дочернем процессе не видны в "родительском" сценарии. Родительский процесс-сценарий, **не может обращаться к переменным, создаваемым в дочернем процессе**.

```
a=123
( a=321; )
```

```
echo "a = $a"    # a = 123
# переменная "a" в скобках подобна локальной переменной.
```

### Инициализация массивов.

```
Array=(element1 element2 element3)
```

```
{xxx,yyy,zzz,...}
```

### Фигурные скобки.

```
grep Linux file*.{txt,htm*}
# Поиск всех вхождений слова "Linux"
# в файлах "fileA.txt", "file2.txt", "fileR.html", "file-87.htm", и пр.
```

Команда интерпретируется как список команд, разделенных точкой с запятой, с вариациями, представленными в *фигурных скобках*. [1] При интерпретации имен файлов (*подстановка*) используются параметры, заключенные в фигурные скобки.



Использование *неэкранированных или неокавыченных* пробелов внутри фигурных скобок недопустимо.

```
echo {file1,file2}\ :{\ A," B",' C'}
```

```
file1 : A file1 : B file1 : C file2 : A file2 : B file2 : C
```

```
{}
```

**Блок кода.** [фигурные скобки] Известен так же как "вложенный блок", эта конструкция, фактически, создает анонимную функцию. Однако, в отличие от обычных *функций*, переменные, создаваемые во вложенных блоках кода, доступны объемлющему сценарию.

```
bash$ { local a; a=123; }
bash: local: can only be used in a function
```

```
a=123
{ a=321; }
echo "a = $a"    # a = 321    (значение, присвоенное во вложенном блоке кода)

# Спасибо, S.C.
```

Код, заключенный в фигурные скобки, может выполнять *перенаправление ввода-вывода*.



**Пример 3–1. Вложенные блоки и перенаправление ввода-вывода**

```
#!/bin/bash
# Чтение строк из файла /etc/fstab.

File=/etc/fstab

{
read line1
read line2
} < $File

echo "Первая строка в $File :"
echo "$line1"
echo
echo "Вторая строка в $File :"
echo "$line2"

exit 0
```

**Пример 3–2. Сохранение результата исполнения вложенного блока в файл**

```
#!/bin/bash
# rpm-check.sh

# Запрашивает описание rpm-архива, список файлов, и проверяется возможность
# установки.
# Результат сохраняется в файле.
#
# Этот сценарий иллюстрирует порядок работы со вложенными блоками кода.

SUCCESS=0
E_NOARGS=65

if [ -z "$1" ]
then
    echo "Порядок использования: `basename $0` rpm-file"
    exit $E_NOARGS
fi

{
    echo
    echo "Описание архива:"
    rpm -qpi $1          # Запрос описания.
    echo
    echo "Список файлов:"
    rpm -qpl $1          # Запрос списка.
    echo
    rpm -i --test $1     # Проверка возможности установки.
    if [ "$?" -eq $SUCCESS ]
    then
        echo "$1 может быть установлен."
    else
        echo "$1 -- установка невозможна!"
    fi
    echo
} > "$1.test"          # Перенаправление вывода в файл.

echo "Результаты проверки rpm-архива находятся в файле $1.test"

# За дополнительной информацией по ключам команды rpm см. man rpm.
```

```
exit 0
```



В отличие от групп команд в (круглых скобках), описанных выше, вложенные блоки кода, заключенные в {фигурные скобки} исполняются в пределах того же процесса, что и сам скрипт (т.е. не вызывают запуск дочернего процесса -- `subshell`). [2]

```
{ } \;
```

**pathname** -- полное имя файла (т.е. путь к файлу и его имя). Чаще всего используется совместно с командой **find**.



Обратите внимание на то, что символ ";", которым завершается ключ `-exec` команды **find**, экранируется обратным слэшем. Это необходимо, чтобы предотвратить его интерпретацию.

```
[ ]
```

**test.**

**Проверка истинности** выражения, заключенного в квадратные скобки `[ ]`. Примечательно, что `[` является частью встроенной команды **test** (и ее синонимом), и *не* имеет никакого отношения к "внешней" утилите `/usr/bin/test`.

```
[[ ]]
```

**test.**

Проверка истинности выражения, заключенного между `[[ ]]` (**зарезервированное слово** интерпретатора).

См. описание конструкции `[[ ... ]]` ниже.

```
[ ]
```

**элемент массива.**

При работе с **массивами** в квадратных скобках указывается порядковый номер того элемента массива, к которому производится обращение.

```
Array[1]=slot_1
echo ${Array[1]}
```

```
[ ]
```

**диапазон символов.**

В **регулярных выражениях**, в квадратных скобках задается **диапазон искомых символов**.

```
(( ))
```

**двойные круглые скобки.**

Вычисляется целочисленное выражение, заключенное между двойными круглыми скобками `(( ))`.

См. обсуждение, посвященное **конструкции (( ... ))** .

> &> >& >> <

### перенаправление.

Конструкция **scriptname >filename** перенаправляет вывод scriptname в файл filename. Если файл filename уже существовал, то его прежнее содержимое будет утеряно.

Конструкция **command &>filename** перенаправляет вывод команды command, как со stdout, так и с stderr, в файл filename.

Конструкция **command >&2** перенаправляет вывод со stdout на stderr.

Конструкция **scriptname >>filename** добавляет вывод scriptname к файлу filename. Если задано имя несуществующего файла, то он создается.

### подстановка процесса.

**(command)>**

**<(command)**

В **операциях сравнения**, символы "<" и ">" обозначают операции **сравнения строк** .

А так же -- операции **сравнения целых чисел**. См. так же **Пример 12-6**.

<<

**перенаправление ввода на встроенный документ.**

<, >

### Посимвольное ASCII-сравнение.

```
veg1=carrots
veg2=tomatoes

if [[ "$veg1" < "$veg2" ]]
then
    echo "Не смотря на то, что в словаре слово $veg1 предшествует слову $veg2,"
    echo "это никак не отражает мои кулинарные предпочтения."
else
    echo "Интересно. Каким словарем вы пользуетесь?"
fi
```

\<, \>

**границы отдельных слов в регулярных выражениях.**

```
bash$ grep '\<the\>' textfile
```

|

**конвейер.** Передает вывод предыдущей команды на ввод следующей или на вход командного интерпретатора shell. Этот метод часто используется для связывания последовательности команд в единую цепочку.

```
echo ls -l | sh
# Передает вывод "echo ls -l" командному интерпретатору shell,
#+ тот же результат дает простая команда "ls -l".
```

```
cat *.lst | sort | uniq
# Объединяет все файлы ".lst", сортирует содержимое и удаляет повторяющиеся строки.
```

Конвейеры (еще их называют каналами) -- это классический способ взаимодействия процессов, с помощью которого stdout одного процесса перенаправляется на stdin другого. Обычно используется совместно с командами вывода, такими как **cat** или **echo**, от которых поток данных поступает в "фильтр" (команда, которая на входе получает данные, преобразует их и обрабатывает).

```
cat $filename | grep $search_word
```

В конвейер могут объединяться и сценарии на языке командной оболочки.


```
#!/bin/bash
# uppercase.sh : Преобразование вводимых символов в верхний регистр.

tr 'a-z' 'A-Z'
# Диапазоны символов должны быть заключены в кавычки
#+ чтобы предотвратить порождение имен файлов от однобуквенных имен файлов.

exit 0
```

А теперь попробуем объединить в конвейер команду **ls -l** с этим сценарием.

```
bash$ ls -l | ./uppercase.sh
-RW-RW-R-- 1 BOZO BOZO 109 APR 7 19:49 1.TXT
-RW-RW-R-- 1 BOZO BOZO 109 APR 14 16:48 2.TXT
-RW-R--R-- 1 BOZO BOZO 725 APR 20 20:56 DATA-FILE
```

 Выход stdout каждого процесса в конвейере должен читаться на входе stdin последующим, в конвейере, процессом. Если этого не делается, то поток данных *блокируется*, в результате конвейер будет работать не так как ожидается.

```
cat file1 file2 | ls -l | sort
# Вывод команды "cat file1 file2" будет утерян.
```

Конвейер выполняется в **дочернем процессе**, а посему -- не имеет доступа к переменным сценария.

```
variable="initial_value"
echo "new_value" | read variable
echo "variable = $variable"      # variable = initial_value
```

Если одна из команд в конвейере завершается аварийно, то это приводит к аварийному завершению работы всего конвейера.

>|

принудительное перенаправление, даже если установлен ключ **noclobber option**.

||

**логическая операция OR (логическое ИЛИ)**. В **опрециях проверки условий**, оператор || возвращает 0 (success), если один из операндов имеет значение true (ИСТИНА).

&

**Выполнение задачи в фоне**. Команда, за которой стоит &, будет исполняться в фоновом режиме.

```
bash$ sleep 10 &
[1] 850
[1]+  Done                  sleep 10
```

В сценариях команды, и даже **циклы** могут запускаться в фоновом режиме.

### Пример 3-3. Запуск цикла в фоновом режиме

```
#!/bin/bash
# background-loop.sh

for i in 1 2 3 4 5 6 7 8 9 10      # Первый цикл.
do
    echo -n "$i "
done & # Запуск цикла в фоне.
      # Иногда возможны случаи выполнения этого цикла после второго цикла.

echo  # Этот 'echo' иногда не отображается на экране.

for i in 11 12 13 14 15 16 17 18 19 20  # Второй цикл.
do
    echo -n "$i "
done

echo  # Этот 'echo' иногда не отображается на экране.

# =====

# Ожидается, что данный сценарий выведет следующую последовательность:
# 1 2 3 4 5 6 7 8 9 10
# 11 12 13 14 15 16 17 18 19 20
```

```
# Иногда возможен такой вариант:
# 11 12 13 14 15 16 17 18 19 20
# 1 2 3 4 5 6 7 8 9 10 bozo $
# (Второй 'echo' не был выполнен. Почему?)

# Изредка возможен такой вариант:
# 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
# (Первый 'echo' не был выполнен. Почему?)

# Крайне редко встречается и такое:
# 11 12 13 1 2 3 4 5 6 7 8 9 10 14 15 16 17 18 19 20
# Второй цикл начал исполняться раньше первого.

exit 0
```



Команда, исполняемая в пределах сценария в фоне, может подвесить сценарий, ожидая нажатия клавиши. К счастью, это легко "лечится".

&&

**Логическая операция AND (логическое И).** В операциях проверки условий, оператор && возвращает 0 (success) тогда, и только тогда, когда *оба* операнда имеют значение true (ИСТИНА).

—

**префикс ключа.** С этого символа начинаются опциональные ключи команд.

**COMMAND** **-[Option1][Option2][...]**

**ls -al**

**sort -dfu \$filename**

**set -- \$variable**

```
if [ $file1 -ot $file2 ]
then
    echo "Файл $file1 был создан раньше чем $file2."
fi

if [ "$a" -eq "$b" ]
then
    echo "$a равно $b."
fi

if [ "$c" -eq 24 -a "$d" -eq 47 ]
then
    echo "$c равно 24, а $d равно 47."
fi
```

—

**перенаправление из/в stdin или stdout.** [дефис]

```
(cd /source/directory && tar cf - . ) | (cd /dest/directory && tar xpvf -)
# Перемещение полного дерева файлов и подкаталогов из одной директории в
другую
```

```
# [спасибо Алану Коксу (Alan Cox) <a.cox@swansea.ac.uk>, за небольшие
поправки]

# 1) cd /source/directory      Переход в исходный каталог, содержимое
которого будет перемещено
# 2) &&                        "И-список": благодаря этому все последующие
команды будут выполнены
#                               только тогда, когда 'cd' завершится успешно
# 3) tar cf - .                ключом 'c' архиватор 'tar' создает новый
архив,                               ключом 'f' (file) и последующим '-' задается
#                               в архив помещается текущий каталог ('.') с
#                               вложенными подкаталогами.
# 4) |                          конвейер с ...
# 5) ( ... )                  subshell-ом (дочерним экземпляром командной
оболочки)
# 6) cd /dest/directory        Переход в каталог назначения.
# 7) &&                        "И-список", см. выше
# 8) tar xpvf -                Разархивирование ('x'), с сохранением
атрибутов "владельца" и прав доступа ('p') к файлам,
#                               с выдачей более подробных сообщений на stdout
#                               ('v'),
#                               файл архива -- stdin ('f' с последующим '-').
#                               Примечательно, что 'x' -- это команда, а 'p',
#                               'v' и 'f' -- ключи
# Во как!
```

```
# Более элегантный вариант:
#   cd source-directory
#   tar cf - . | (cd ../target-directory; tar xzf -)
#
# cp -a /source/directory /dest      имеет тот же эффект.
```

```
bunzip2 linux-2.4.3.tar.bz2 | tar xvf -
# --разархивирование tar-файла-- | --затем файл передается утилите
"tar"--
# Если у вас утилита "tar" не поддерживает работу с "bunzip2",
# тогда придется выполнять работу в два этапа, с использованием конвейера.
# Целью данного примера является разархивирование тарбола (tar.bz2) с
исходными текстами ядра.
```

Обратите внимание, что в этом контексте "--" – не самостоятельный оператор Bash, а скорее опция, распознаваемая некоторыми утилитами UNIX (такими как **tar**, **cat** и т.п.), которые выводят результаты своей работы в stdout.

```
bash$ echo "whatever" | cat -
whatever
```

В случае, когда ожидается имя файла, тогда "-" перенаправляет вывод на stdout (вспомните пример с **tar cf**) или принимает ввод с stdin.

```
bash$ file
Usage: file [-bciknvzL] [-f namefile] [-m magicfiles] file...
```

Сама по себе команда **file** без параметров завершается с сообщением об ошибке.

Добавим символ "-" и получим более полезный результат. Это заставит командный интерпретатор ожидать ввода от пользователя.

```
bash$ file -
abc
standard input:          ASCII text

bash$ file -
#!/bin/bash
standard input:          Bourne-Again shell script text executable
```

Теперь команда принимает ввод пользователя со stdin и анализирует его.

Используя передачу stdout по конвейеру другим командам, можно выполнять довольно эффектные трюки, например **вставка строк в начало файла**.

С помощью команды **diff --** находить различия между одним файлом и *частью* другого:

```
grep Linux file1 | diff file2 -
```

И наконец пример использования служебного символа "-" с командой **tar**.

**Пример 3-4. Резервное архивирование всех файлов, которые были изменены в течение последних суток**

```
#!/bin/bash

# Резервное архивирование (backup) всех файлов в текущем каталоге,
# которые были изменены в течение последних 24 часов
#+ в тарболл (tarball) (.tar.gz - файл).

BACKUPFILE=backup
archive=${1:-$BACKUPFILE}
# На случай, если имя архива в командной строке не задано,
#+ т.е. по-умолчанию имя архива -- "backup.tar.gz"

tar cvf - `find . -mtime -1 -type f -print` > $archive.tar
gzip $archive.tar
echo "Каталог $PWD заархивирован в файл \"$archive.tar.gz\"."

# Stephane Chazelas заметил, что вышеприведенный код будет "падать"
#+ если будет найдено слишком много файлов
#+ или если имена файлов будут содержать символы пробела.

# Им предложен альтернативный код:
# -----
```



```
# find . -mtime -1 -type f -print0 | xargs -0 tar rvf "$archive.tar"
#      используется версия GNU утилиты "find".

# find . -mtime -1 -type f -exec tar rvf "$archive.tar" '{}' \;
#      более универсальный вариант, хотя и более медленный,
#      зато может использоваться в других версиях UNIX.
# -----
```

```
exit 0
```



Могут возникнуть конфликтные ситуации между оператором перенаправления "-" и именами файлов, начинающимися с символа "-". Поэтому сценарий должен проверять имена файлов и предавать их префиксом пути, например, ./-FILENAME, \$PWD/-FILENAME или \$PATHNAME/-FILENAME.

Если значение переменной начинается с символа "-", то это тоже может быть причиной появления ошибок.

```
var="-n"
echo $var
# В данном случае команда приобретет вид "echo -n" и ничего не
выведет.
```

-

**предыдущий рабочий каталог.** [дефис] Команда **cd** – выполнит переход в предыдущий рабочий каталог, путь к которому хранится в [переменной окружения \\$OLDPWD](#) .



Не путайте оператор "-" (предыдущего рабочего каталога) с оператором "-" (переназначения). Еще раз напомню, что интерпретация символа "-" зависит от контекста, в котором он употребляется.

-

**Минус.** Знак минус в [арифметических операциях](#).

=

**Символ "равно".** Оператор [присваивания](#)

```
a=28
echo $a    # 28
```

В зависимости от [контекста применения](#), символ "=" может выступать в качестве оператора [сравнения](#).

+

**Плюс.** Оператор сложения в [арифметических операциях](#).

В зависимости от [контекста применения](#), символ + может выступать как оператор [регулярного выражения](#).

+

**Ключ (опция).** Дополнительный флаг для ключей (опций) команд.

Отдельные внешние и **встроенные** команды используют символ "+" для разрешения некоторой опции, а символ "-" -- для запрещения.

%

**модуль.** Модуль (остаток от деления) -- **арифметическая операция.**

В зависимости от **контекста применения**, символ % может выступать в качестве **шаблона.**

~

**домашний каталог.** [тильда] Соответствует содержимому внутренней переменной **\$HOME**.  
**~bozo** -- домашний каталог пользователя bozo, а команда **ls ~bozo** выведет содержимое его домашнего каталога. **~/** -- это домашний каталог текущего пользователя, а команда **ls ~/** выведет содержимое домашнего каталога текущего пользователя.

```
bash$ echo ~bozo
/home/bozo
```

```
bash$ echo ~
/home/bozo
```

```
bash$ echo ~/
/home/bozo/
```

```
bash$ echo ~:
/home/bozo:
```

```
bash$ echo ~nonexistent-user
~nonexistent-user
```

~+

**текущий рабочий каталог.** Соответствует содержимому внутренней переменной **\$PWD**.

~-

**предыдущий рабочий каталог.** Соответствует содержимому внутренней переменной **\$OLDPWD**.

^

**начало-строки.** В **регулярных выражениях** символ "^" задает начало строки текста.

Управляющий символ

**изменяет поведение терминала или управляет выводом текста.** Управляющий символ набирается с клавиатуры как комбинация **CONTROL + <клавиша>**.

- **Ctl-C**

Завершение выполнения процесса.

- **Ctl-D**

Выход из командного интерпретатора (log out) (аналог команды `exit`).

"EOF" (признак конца файла). Этот символ может выступать в качестве завершающего при вводе с `stdin`.

- **Ctl-G**

"BEL" (звуковой сигнал -- "звонок").

- **Ctl-H**

Backspace -- удаление предыдущего символа.

```
#!/bin/bash
```

```
# Вставка символа Ctl-H в строку.
```

```
a="^H^H" # Два символа Ctl-H (backspace).
```

```
echo "abcdef" # abcdef
```

```
echo -n "abcdef$a " # abcd f
```

```
# Пробел в конце ^ ^ двойной шаг назад.
```

```
echo -n "abcdef$a" # abcdef
```

```
# Пробела в конце нет backspace не работает (почему?).
```

```
# Результаты могут получиться совсем не те, что вы ожидаете.
```

```
echo; echo
```

- **Ctl-J**

Возврат каретки.

- **Ctl-L**

Перевод формата (очистка экрана (окна) терминала). Аналогична команде `clear`.

- **Ctl-M**

Перевод строки.

- **Ctl-U**

Стирание строки ввода.

- **Ctl-Z**

Приостановка процесса.

## Пробельный символ

используется как **разделитель команд или переменных**. В качестве пробельного символа могут выступать -- собственно пробел (`space`), символ табуляции, символ перевода строки, символ возврата каретки или комбинация из вышеперечисленных символов. В некоторых случаях, таких как **присваивание значений переменным**, использование пробельных символов недопустимо.

Пустые строки никак не обрабатываются командным интерпретатором и могут свободно использоваться для визуального выделения отдельных блоков сценария.

**\$IFS** -- переменная специального назначения. Содержит символы-разделители полей, используемые некоторыми командами. По-умолчанию -- пробельные символы.

## Примечания

- [1] Интерпретатор, встретив фигурные скобки, раскрывает их и возвращает полученный список команд, которые затем и исполняет.
- [2] Исключение: блок кода, являющийся частью конвейера, *может* быть запущен в дочернем процессе (**subshell-e**).

```
ls | { read firstline; read secondline; }  
# Ошибка! Вложенный блок будет запущен в дочернем процессе,  
# таким образом, вывод команды "ls" не может быть записан в переменные  
# находящиеся внутри блока.  
echo "Первая строка: $firstline; вторая строка: $secondline" # Не  
работает!  
  
# Спасибо S.C.
```

[Назад](#)  
Основы

[К началу](#)  
Наверх

[Вперед](#)  
Переменные и параметры.  
Введение.

Спонсоры:



При поддержке  
**inferno solutions\***

Хостинг:



**Hoster.ru**  
хостинг провайдер

[Закладки на сайте](#)  
[Проследить за страницей](#)

Created 1996–2022 by [Maxim Chirkov](#)  
[Добавить](#), [Поддержать](#), [Вебмастеру](#)