

Глава 16. Перенаправление ввода/вывода

В системе по-умолчанию всегда открыты три "файла" --

`stdin` (клавиатура), `stdout` (экран) и `stderr` (вывод сообщений об ошибках на экран). Эти, и любые другие открытые файлы, могут быть перенаправлены. В данном случае, термин "перенаправление" означает получить вывод из файла, команды, программы, сценария или даже отдельного блока в сценарии (см. [Пример 3-1](#) и [Пример 3-2](#)) и передать его на вход в другой файл, команду, программу или сценарий.

С каждым открытым файлом связан дескриптор файла. [1] Дескрипторы файлов `stdin`, `stdout` и `stderr` -- 0, 1 и 2, соответственно. При открытии дополнительных файлов, дескрипторы с 3 по 9 остаются незанятыми. Иногда дополнительные дескрипторы могут сослужить неплохую службу, временно сохраняя в себе ссылку на `stdin`, `stdout` или `stderr`. [2] Это упрощает возврат дескрипторов в нормальное состояние после сложных манипуляций с перенаправлением и перестановками (см. [Пример 16-1](#)).

```
COMMAND_OUTPUT >
# Перенаправление stdout (вывода) в файл.
# Если файл отсутствовал, то он создается, иначе --
перезаписывается.

ls -lR > dir-tree.list
# Создает файл, содержащий список дерева каталогов.

: > filename
# Операция > усекает файл "filename" до нулевой длины.
# Если до выполнения операции файла не существовало,
# то создается новый файл с нулевой длиной (тот же эффект дает
команда 'touch').
# Символ : выступает здесь в роли местозаполнителя, не выводя
ничего.

> filename
# Операция > усекает файл "filename" до нулевой длины.
# Если до выполнения операции файла не существовало,
# то создается новый файл с нулевой длиной (тот же эффект дает
команда 'touch').
# (тот же результат, что и выше -- ": >", но этот вариант
неработоспособен
# в некоторых командных оболочках.)

COMMAND_OUTPUT >>
# Перенаправление stdout (вывода) в файл.
# Создает новый файл, если он отсутствовал, иначе -- дописывает
в конец файла.

# Однострочные команды перенаправления
# (затрагивают только ту строку, в которой они встречаются):
# -----
-----
```

```

1>filename
# Перенаправление вывода (stdout) в файл "filename".
1>>filename
# Перенаправление вывода (stdout) в файл "filename", файл
открывается в режиме добавления.
2>filename
# Перенаправление stderr в файл "filename".
2>>filename
# Перенаправление stderr в файл "filename", файл открывается в
режиме добавления.
&>filename
# Перенаправление stdout и stderr в файл "filename".

#=====
=====
# Перенаправление stdout, только для одной строки.
LOGFILE=script.log

echo "Эта строка будет записана в файл \"$LOGFILE\"." 1>$LOGFILE
echo "Эта строка будет добавлена в конец файла \"$LOGFILE\"."
1>>$LOGFILE
echo "Эта строка тоже будет добавлена в конец файла
\"$LOGFILE\"." 1>>$LOGFILE
echo "Эта строка будет выведена на экран и не попадет в файл
\"$LOGFILE\"."
# После каждой строки, сделанное перенаправление автоматически
"сбрасывается".

# Перенаправление stderr, только для одной строки.
ERRORFILE=script.errors

bad_command1 2>$ERRORFILE          # Сообщение об ошибке запишется
в $ERRORFILE.
bad_command2 2>>$ERRORFILE          # Сообщение об ошибке добавится
в конец $ERRORFILE.
bad_command3                        # Сообщение об ошибке будет
выведено на stderr,
                                     #+ и не попадет в $ERRORFILE.
# После каждой строки, сделанное перенаправление также
автоматически "сбрасывается".

#=====
=====

2>&1
# Перенаправляется stderr на stdout.
# Сообщения об ошибках передаются туда же, куда и стандартный
вывод.

i>&j
# Перенаправляется файл с дескриптором i в j.

```

```

    # Вывод в файл с дескриптором i передается в файл с дескриптором
j.

>&j
    # Перенаправляется файл с дескриптором 1 (stdout) в файл с
дескриптором j.
    # Вывод на stdout передается в файл с дескриптором j.

0< FILENAME
< FILENAME
    # Ввод из файла.
    # Парная команде ">", часто встречается в комбинации с ней.
    #
    # grep search-word <filename

[j]<>filename
    # Файл "filename" открывается на чтение и запись, и связывается
с дескриптором "j".
    # Если "filename" отсутствует, то он создается.
    # Если дескриптор "j" не указан, то, по-умолчанию, берется
дескриптор 0, stdin.
    #
    # Как одно из применений этого -- запись в конкретную позицию в
файле.
echo 1234567890 > File      # Записать строку в файл "File".
exec 3<> File              # Открыть "File" и связать с дескриптором 3.
read -n 4 <&3               # Прочитать 4 символа.
echo -n . >&3               # Записать символ точки.
exec 3>&-                   # Закрыть дескриптор 3.
cat File                   # ==> 1234.67890
    # Произвольный доступ, да и только!

|
    # Конвейер (канал).
    # Универсальное средство для объединения команд в одну цепочку.
    # Похоже на ">", но на самом деле -- более обширная.
    # Используется для объединения команд, сценариев, файлов и
программ в одну цепочку (конвейер).
cat *.txt | sort | uniq > result-file
    # Содержимое всех файлов .txt сортируется, удаляются
повторяющиеся строки,
    # результат сохраняется в файле "result-file".

```

Операции перенаправления и/или конвейеры могут комбинироваться в одной командной строке.

```
command < input-file > output-file
```

```
command1 | command2 | command3 > output-file
```

См. [Пример 12-23](#) и [Пример A-17](#).

Допускается перенаправление нескольких потоков в один файл.

```
ls -yz >> command.log 2>&1
```

```
# Сообщение о неверной опции "yz" в команде "ls" будет записано в файл
"command.log".
# Поскольку stderr перенаправлен в файл.
```

Заккрытие дескрипторов файлов

`n<&-`

Закрывает дескриптор входного файла `n`.

`0<&-`, `<&-`

Закрывает `stdin`.

`n>&-`

Закрывает дескриптор выходного файла `n`.

`1>&-`, `>&-`

Закрывает `stdout`.

Дочерние процессы наследуют дескрипторы открытых файлов. По этой причине и работают конвейеры. Чтобы предотвратить наследование дескрипторов -- закройте их перед запуском дочернего процесса.

```
# В конвейер передается только stderr.
```

```
exec 3>&1                                # Сохранить текущее "состояние"
stdout.
ls -l 2>&1 >&3 3>&- | grep bad 3>&-        # Закрывает дескр. 3 для 'grep'
(но не для 'ls').
#             ^^^^      ^^^^
exec 3>&-                                # Теперь закрыть его для
оставшейся части сценария.

# Спасибо S.C.
```

Дополнительные сведения о перенаправлении ввода/вывода вы найдете в [Приложение D](#).

16.1. С помощью команды `exec`

Команда `exec <filename` перенаправляет ввод со `stdin` на файл. С этого момента весь ввод, вместо `stdin` (обычно это клавиатура), будет производиться из этого файла. Это дает возможность читать содержимое файла, строку за строкой, и анализировать каждую введенную строку с помощью [sed](#) и/или [awk](#).

Пример 16-1. Перенаправление `stdin` с помощью `exec`

```
#!/bin/bash
# Перенаправление stdin с помощью 'exes'.

exes 6<&0          # Связать дескр. #6 со стандартным вводом (stdin).
                  # Сохраняя stdin.

exes < data-file   # stdin заменяется файлом "data-file"

read a1            # Читается первая строка из "data-file".
read a2            # Читается вторая строка из "data-file."

echo
echo "Следующие строки были прочитаны из файла."
echo "-----"
echo $a1
echo $a2

echo; echo; echo

exes 0<&6 6<&-
# Восстанавливается stdin из дескр. #6, где он был предварительно
сохранен,
#+ и дескр. #6 закрывается ( 6<&- ) освобождая его для других
процессов.
#
# <&6 6<&-      дает тот же результат.

echo -n "Введите строку "
read b1 # Теперь функция "read", как и следовало ожидать, принимает
данные с обычного stdin.
echo "Строка, принятая со stdin."
echo "-----"
echo "b1 = $b1"

echo

exit 0
```

Аналогично, конструкция **exes >filename** перенаправляет вывод на `stdout` в заданный файл. После этого, весь вывод от команд, который обычно направляется на `stdout`, теперь выводится в этот файл.

Пример 16-2. Перенаправление `stdout` с помощью `exes`

```
#!/bin/bash
# reassign-stdout.sh

LOGFILE=logfile.txt

exes 6>&1          # Связать дескр. #6 со stdout.
                  # Сохраняя stdout.

exes > $LOGFILE    # stdout замещается файлом "logfile.txt".

# ----- #
# Весь вывод от команд, в данном блоке, записывается в файл $LOGFILE.
```

```

echo -n "Logfile: "
date
echo "-----"
echo

echo "Вывод команды \"ls -al\""
echo
ls -al
echo; echo
echo "Вывод команды \"df\""
echo
df

# ----- #

exec 1>&6 6>&-      # Восстановить stdout и закрыть дескр. #6.

echo
echo "== stdout восстановлено в значение по-умолчанию =="
echo
ls -al
echo

exit 0

```

Пример 16-3. Одновременное перенаправление устройств, stdin и stdout, с помощью команды exec

```

#!/bin/bash
# upperconv.sh
# Преобразование символов во входном файле в верхний регистр.

E_FILE_ACCESS=70
E_WRONG_ARGS=71

if [ ! -r "$1" ]      # Файл доступен для чтения?
then
    echo "Невозможно прочитать из заданного файла!"
    echo "Порядок использования: $0 input-file output-file"
    exit $E_FILE_ACCESS
fi
                    # В случае, если входной файл ($1) не задан
                    #+ код завершения будет этим же.

if [ -z "$2" ]
then
    echo "Необходимо задать выходной файл."
    echo "Порядок использования: $0 input-file output-file"
    exit $E_WRONG_ARGS
fi

exec 4<&0
exec < $1            # Назначить ввод из входного файла.

exec 7>&1
exec > $2            # Назначить вывод в выходной файл.

```

```

# Предполагается, что выходной файл доступен для
записи

# (добавить проверку?).

# -----
#   cat - | tr a-z A-Z    # Перевод в верхний регистр
#   ^^^^^               # Чтение со stdin.
#   ^^^^^^^^^^^         # Запись в stdout.
# Однако, и stdin и stdout были перенаправлены.
# -----

exec 1>&7 7>&-           # Восстановить stdout.
exec 0<&4 4<&-           # Восстановить stdin.

# После восстановления, следующая строка выводится на stdout, чего и
следовало ожидать.
echo "Символы из \"$1\" преобразованы в верхний регистр, результат
записан в \"$2\"."

exit 0

```

Примечания

- [1] *дескриптор файла* -- это просто число, по которому система идентифицирует открытые файлы. Рассматривайте его как упрощенную версию указателя на файл.
- [2] При использовании *дескриптора с номером 5* могут возникать проблемы. Когда Bash порождает дочерний процесс, например командой [exec](#), то дочерний процесс наследует дескриптор 5 как "открытый" (см. архив почты Чета Рамея (Chet Ramey), [SUBJECT: RE: File descriptor 5 is held open](#)) Поэтому, лучше не использовать этот дескриптор.

[Назад](#)

Арифметические
подстановки

[К началу](#)

[Наверх](#)

[Вперед](#)

Перенаправление для блоков
кода

16.2. Перенаправление для блоков кода

Блоки кода, такие как циклы [while](#), [until](#) и [for](#), условный оператор [if/then](#), так же могут смешиваться с перенаправлением `stdin`. Даже функции могут использовать эту форму перенаправления (см. [Пример 22-7](#)). Оператор перенаправления `<`, в таких случаях, ставится в конце блока.

Пример 16-4. Перенаправление в цикл *while*

```

#!/bin/bash

if [ -z "$1" ]
then

```

```

    Filename=names.data      # По-умолчанию, если имя файла не задано.
else
    Filename=$1
fi
# Конструкцию проверки выше, можно заменить следующей строкой
(подстановка параметров):
#+ Filename=${1:-names.data}

count=0

echo

while [ "$name" != Smith ] # Почему переменная $name взята в кавычки?
do
    read name              # Чтение из $Filename, не со stdin.
    echo $name
    let "count += 1"
done <"$Filename"         # Перенаправление на ввод из файла
$Filename.
#      ^^^^^^^^^^^^^^

echo; echo "Имен прочитано: $count"; echo

# Обратите внимание: в некоторых старых командных интерпретаторах,
#+ перенаправление в циклы приводит к запуску цикла в субоболочке
(subshell).
# Таким образом, переменная $count, по окончании цикла, будет
содержать 0,
# значение, записанное в нее до входа в цикл.
# Bash и ksh стремятся избежать запуска субоболочки (subshell), если
это возможно,
#+ так что этот сценарий, в этих оболочках, работает корректно.
#
# Спасибо Heiner Steven за это примечание.

exit 0

```

Пример 16-5. Альтернативная форма перенаправления в цикле *while*

```

#!/bin/bash

# Это альтернативный вариант предыдущего сценария.

# Предложил: by Heiner Steven
#+ для случаев, когда циклы с перенаправлением
#+ запускаются в субоболочке, из-за чего переменные, устанавливаемые в
цикле,
#+ не сохраняют свои значения по завершении цикла.

if [ -z "$1" ]
then
    Filename=names.data      # По-умолчанию, если имя файла не задано.
else
    Filename=$1
fi

```



```

exec 3<&0                # Сохранить stdin в дескр. 3.
exec 0<"$Filename"      # Перенаправить stdin.

count=0
echo

while [ "$name" != Smith ]
do
    read name            # Прочитать с перенаправленного stdin
    ($Filename).
    echo $name
    let "count += 1"
done <"$Filename"        # Цикл читает из файла $Filename.
#      ^^^^^^^^^^^^^^^

exec 0<&3                # Восстановить stdin.
exec 3<&-                 # Закрыть временный дескриптор 3.

echo; echo "Имен прочитано: $count"; echo

exit 0

```

Пример 16-6. Перенаправление в цикл *until*

```

#!/bin/bash
# То же самое, что и в предыдущем примере, только для цикла "until".

if [ -z "$1" ]
then
    Filename=names.data    # По-умолчанию, если файл не задан.
else
    Filename=$1
fi

# while [ "$name" != Smith ]
until [ "$name" = Smith ]  # Проверка != изменена на =.
do
    read name              # Чтение из $Filename, не со stdin.
    echo $name
done <"$Filename"          # Перенаправление на ввод из файла
#      ^^^^^^^^^^^^^^^   $Filename.

# Результаты получаются теми же, что и в случае с циклом "while", в
# предыдущем примере.

exit 0

```

Пример 16-7. Перенаправление в цикл *for*

```

#!/bin/bash

if [ -z "$1" ]
then
    Filename=names.data    # По-умолчанию, если файл не задан.

```

```

else
    Filename=$1
fi

line_count=`wc $Filename | awk '{ print $1 }'`
#           Число строк в файле.
#
# Слишком запутано, тем не менее показывает
#+ возможность перенаправления stdin внутри цикла "for"...
#+ если вы достаточно умны.
#
# Более короткий вариант      line_count=$(wc < "$Filename")

for name in `seq $line_count` # "seq" выводит последовательность
чисел.
# while [ "$name" != Smith ] -- более запутанно, чем в случае с
циклом "while" --
do
    read name                # Чтение из файла $Filename, не со
stdin.
    echo $name
    if [ "$name" = Smith ]
    then
        break
    fi
done <"$Filename"           # Перенаправление на ввод из файла
$Filename.
#      ^^^^^^^^^^^^^
exit 0

```

Предыдущий пример можно модифицировать так, чтобы перенаправить вывод из цикла.

Пример 16-8. Перенаправление устройств (stdin и stdout) в цикле *for*

```

#!/bin/bash

if [ -z "$1" ]
then
    Filename=names.data      # По-умолчанию, если файл не задан.
else
    Filename=$1
fi

Savefile=$Filename.new      # Имя файла, в котором сохраняются
результаты.
FinalName=Jonah             # Имя, на котором завершается чтение.

line_count=`wc $Filename | awk '{ print $1 }'` # Число строк в
заданном файле.

for name in `seq $line_count`
do
    read name

```

```

    echo "$name"
    if [ "$name" = "$FinalName" ]
    then
        break
    fi
done < "$Filename" > "$Savefile"      # Перенаправление на ввод из
файла $Filename,
#      ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^      и сохранение результатов в
файле.

exit 0

```

Пример 16-9. Перенаправление в конструкции *if/then*

```

#!/bin/bash

if [ -z "$1" ]
then
    Filename=names.data      # По-умолчанию, если файл не задан.
else
    Filename=$1
fi

TRUE=1

if [ "$TRUE" ]              # конструкции "if true" и "if :" тоже вполне
допустимы.
then
    read name
    echo $name
fi <"$Filename"
#  ^^^^^^^^^^^^^

# Читает только первую строку из файла.

exit 0

```

Пример 16-10. Файл с именами "names.data", для примеров выше

```

Aristotle
Belisarius
Capablanca
Euler
Goethe
Hamurabi
Jonah
Laplace
Maroczy
Purcell
Schmidt
Simmelweiss
Smith
Turing
Venn
Wilson
Znosko-Borowski

```

```
# Это файл с именами для примеров
#+ "redir2.sh", "redir3.sh", "redir4.sh", "redir4a.sh", "redir5.sh".
```

Перенаправление `stdout` для блока кода, может использоваться для сохранения результатов работы этого блока в файл. См. [Пример 3-2](#).

[Встроенный документ](#) -- это особая форма перенаправления для блоков кода.

[Назад](#)

[К началу](#)

[Вперед](#)

Перенаправление
ввода/вывода

[Наверх](#)

Область применения

Глава 15. Арифметические подстановки

Арифметические подстановки -- это мощный инструмент, предназначенный для выполнения арифметических операций в сценариях. Перевод строки в числовое выражение производится с помощью [обратных одиночных кавычек](#), [двойных круглых скобок](#) или предложения [let](#).

Вариации

Арифметические подстановки в обратных одиночных кавычках (часто используются совместно с командой [expr](#))

```
z=`expr $z + 3`           # Команда 'expr' вычисляет значение
                           # выражения.
```

Арифметические подстановки в двойных круглых скобках, и предложение **let**

В арифметических подстановках, обратные одиночные кавычки могут быть заменены на двойные круглые скобки `$((...))` или очень удобной конструкцией, с применением предложения **let**.

```
z=$(( $z + 3 ))
# $((EXPRESSION)) -- это подстановка арифметического выражения.
# Не путайте с

#+ подстановкой команд.

let z=z+3
let "z += 3"  # Кавычки позволяют вставлять пробелы и специальные
              # операторы.
# Оператор 'let' вычисляет арифметическое выражение,
#+ это не подстановка арифметического выражения.
```

Все вышеприведенные примеры эквивалентны. Вы можете использовать любую из этих форм записи "по своему вкусу".

Примеры арифметических подстановок в сценариях:

1. [Пример 12-6](#)
2. [Пример 10-14](#)
3. [Пример 25-1](#)
4. [Пример 25-6](#)
5. [Пример A-19](#)

[Назад](#)

[К началу](#)

[Вперед](#)

Подстановка команд

[Наверх](#)

Перенаправление
ввода/вывода

Глава 14. Подстановка команд

Подстановка команд -- это подстановка результатов выполнения команды [1] или даже серии команд; буквально, эта операция позволяет вызвать команду в другом окружении.

Классический пример подстановки команд -- использование обратных одиночных кавычек (``...``). Команды внутри этих кавычек представляют собой текст командной строки.

```
script_name=`basename $0`  
echo "Имя этого файла-сценария: $script_name."
```

Вывод от команд может использоваться: как аргумент другой команды, для установки значения переменной и даже для генерации списка аргументов цикла [for](#).


```
rm `cat filename`    # здесь "filename" содержит список удаляемых  
файлов.  
#  
# S. C. предупреждает, что в данном случае может возникнуть ошибка  
"arg list too long".  
# Такой вариант будет лучше:    xargs rm -- < filename  
# ( -- подходит для случая, когда "filename" начинается с символа "--"  
# )  
  
textfile_listing=`ls *.txt`  
# Переменная содержит имена всех файлов *.txt в текущем каталоге.  
echo $textfile_listing  
  
textfile_listing2=$(ls *.txt)    # Альтернативный вариант.
```

```

echo $textfile_listing2
# Результат будет тем же самым.

# Проблема записи списка файлов в строковую переменную состоит в том,
# что символы перевода строки заменяются на пробел.
#
# Как вариант решения проблемы -- записывать список файлов в массив.
#   shopt -s nullglob      # При несоответствии, имя файла
#   textfile_listing=( *.txt )
#   ignore
#
# Спасибо S.C.

```

 Подставляемая команда может получиться разбитой на отдельные слова.

```

COMMAND `echo a b`      # 2 аргумента: a и b

COMMAND "`echo a b`"    # 1 аргумент: "a b"

COMMAND `echo`          # без аргументов

COMMAND "`echo`"        # один пустой аргумент

# Спасибо S.C.

```

Даже когда не происходит разбиения на слова, операция подстановки команд может удалять завершающие символы перевода строки.

```

# cd "`pwd`" # Должна выполняться всегда.
# Однако...

mkdir 'dir with trailing newline
'

cd 'dir with trailing newline
'

cd "`pwd`" # Ошибка:
# bash: cd: /tmp/dir with trailing newline: No such file or
# directory

cd "$PWD" # Выполняется без ошибки.


old_tty_setting=$(stty -g) # Сохранить настройки терминала.
echo "Нажмите клавишу "
stty -icanon -echo        # Запретить "канонический" режим
терминала.
# Также запрещает эхо-вывод.
key=$(dd bs=1 count=1 2> /dev/null) # Поймать нажатие на клавишу.
stty "$old_tty_setting"      # Восстановить настройки терминала.


```

```

echo "Количество нажатых клавиш = ${#key}." # ${#variable} =
количество символов в переменной $variable
#
# Нажмите любую клавишу, кроме RETURN, на экране появится
"Количество нажатых клавиш = 1."
# Нажмите RETURN, и получите: "Количество нажатых клавиш = 0."
# Символ перевода строки будет "съеден" операцией подстановки
команды.

```

Спасибо S.C.

 При выводе значений переменных, полученных в результате подстановки команд, командой **echo**, без кавычек, символы перевода строки будут удалены. Это может оказаться неприятным сюрпризом.

```

dir_listing=`ls -l`
echo $dir_listing      # без кавычек

# Вы наверно ожидали увидеть удобочитаемый список каталогов.

# Однако, вы получите:
# total 3 -rw-rw-r-- 1 bozo bozo 30 May 13 17:15 1.txt -rw-rw-r-- 1
bozo
# bozo 51 May 15 20:57 t2.sh -rwxr-xr-x 1 bozo bozo 217 Mar 5 21:13
wi.sh

# Символы перевода строки были заменены пробелами.

echo "$dir_listing"    # в кавычках
# -rw-rw-r--      1 bozo      30 May 13 17:15 1.txt
# -rw-rw-r--      1 bozo      51 May 15 20:57 t2.sh
# -rwxr-xr-x      1 bozo      217 Mar  5 21:13 wi.sh

```

Подстановка команд позволяет даже записывать в переменные содержимое целых файлов, с помощью [перенаправления](#) или команды [cat](#).

```

variable1=`<file1`      # Записать в переменную "variable1"
содержимое файла "file1".
variable2=`cat file2`    # Записать в переменную "variable2" содержимое
файла "file2".

# Замечание 1:
# Удаляются символы перевода строки.
#
# Замечание 2:
# В переменные можно записать даже управляющие символы.

# Выдержки из системного файла /etc/rc.d/rc.sysinit
#+ (Red Hat Linux)

```

```

if [ -f /fsckoptions ]; then
    fsckoptions=`cat /fsckoptions`
...
fi
#
#
if [ -e "/proc/ide/${disk[$device]}/media" ] ; then
    hdmedia=`cat /proc/ide/${disk[$device]}/media`
...
fi
#
#
if [ ! -n "`uname -r | grep -- "-"``" ]; then
    ktag="`cat /proc/version`"
...
fi
#
#
if [ $usb = "1" ]; then
    sleep 5
    mouseoutput=`cat /proc/bus/usb/devices 2>/dev/null|grep -E
"^I.*Cls=03.*Prot=02"`
    kbdoutput=`cat /proc/bus/usb/devices 2>/dev/null|grep -E
"^I.*Cls=03.*Prot=01"`
...
fi

```



Не используйте переменные для хранения содержимого текстовых файлов *большого* объема, без веских на то оснований. Не записывайте в переменные содержимое *бинарных* файлов, даже шутки ради.

Пример 14-1. Глупая выходка

```

#!/bin/bash
# stupid-script-tricks.sh: Люди! Будьте благоразумны!
# Из "Глупые выходки", том I.

dangerous_variable=`cat /boot/vmlinuz` # Сжатое ядро Linux.

echo "длина строки \$dangerous_variable = ${#dangerous_variable}"
# длина строки $dangerous_variable = 794151
# ('wc -c /boot/vmlinuz' даст другой результат.)

# echo "$dangerous_variable"
# Даже не пробуйте раскомментировать эту строку! Это приведет к
зависанию сценария.

# Автор этого документа не знает, где можно было бы использовать
#+ запись содержимого двоичных файлов в переменные.

exit 0

```

Обратите внимание: в данной ситуации не возникает ошибки *переполнения буфера*.

Этот пример показывает превосходство защищенности интерпретирующих языков, таких как Bash, от ошибок программиста, над компилирующими языками программирования.

Подстановка команд, позволяет записать в переменную результаты выполнения [цикла](#). Ключевым моментом здесь является команда [echo](#), в теле цикла.

Пример 14-2. Запись результатов выполнения цикла в переменную

```
#!/bin/bash
# csubloop.sh: Запись результатов выполнения цикла в переменную

variable1=`for i in 1 2 3 4 5
do
    echo -n "$i"                # Здесь 'echo' -- это ключевой момент
done`

echo "variable1 = $variable1"  # variable1 = 12345

i=0
variable2=`while [ "$i" -lt 10 ]
do
    echo -n "$i"                # Опять же, команда 'echo' просто
    необходима.
    let "i += 1"                # Увеличение на 1.
done`

echo "variable2 = $variable2"  # variable2 = 0123456789

exit 0
```

Подстановка команд позволяет существенно расширить набор инструментальных средств, которыми располагает Bash. Суть состоит в том, чтобы написать программу или сценарий, которая выводит результаты своей работы на stdout (как это делает подавляющее большинство утилит в UNIX) и записать вывод от программы в переменную.

```
#include <stdio.h>

/* Программа на C "Hello, world." */

int main()
{
    printf( "Hello, world." );
    return (0);
}
bash$ gcc -o hello hello.c

#!/bin/bash
# hello.sh

greeting=`./hello`
```

```
echo $greeting
bash$ sh hello.sh
Hello, world.
```



Альтернативой обратным одиночным кавычкам, используемым для подстановки команд, можно считать такую форму записи: **\$(COMMAND)**.

```
output=$(sed -n /"$1"/p $file)    # К примеру из "grp.sh".

# Запись в переменную содержимого текстового файла.
File_contents1=$(cat $file1)
File_contents2=$(<$file2)        # Bash допускает и такую запись.
```

Примеры подстановки команд в сценариях:

1. [Пример 10-7](#)
2. [Пример 10-26](#)
3. [Пример 9-26](#)
4. [Пример 12-2](#)
5. [Пример 12-15](#)
6. [Пример 12-12](#)
7. [Пример 12-39](#)
8. [Пример 10-13](#)
9. [Пример 10-10](#)
10. [Пример 12-24](#)
11. [Пример 16-7](#)
12. [Пример A-19](#)
13. [Пример 27-1](#)
14. [Пример 12-32](#)
15. [Пример 12-33](#)
16. [Пример 12-34](#)

Примечания

- [1] *Замещающая команда* может быть внешней системной командой, внутренней (встроенной) командой или даже функцией в сценарии.

[Назад](#)

Команды системного
администрирования

[К началу](#)

[Наверх](#)

[Вперед](#)

Арифметические
подстановки

