

# Bash-скрипты, часть 3: параметры и ключи командной строки

<https://likegeeks.com/linux-bash-scripting-awesome-guide-part3/>

- Блог компании RUVDS.com,
- Настройка Linux,
- Серверное администрирование
- [Перевод](#)

[Bash-скрипты: начало](#)

[Bash-скрипты, часть 2: циклы](#)

[Bash-скрипты, часть 3: параметры и ключи командной строки](#)

[Bash-скрипты, часть 4: ввод и вывод](#)

[Bash-скрипты, часть 5: сигналы, фоновые задачи, управление сценариями](#)

[Bash-скрипты, часть 6: функции и разработка библиотек](#)

[Bash-скрипты, часть 7: sed и обработка текстов](#)

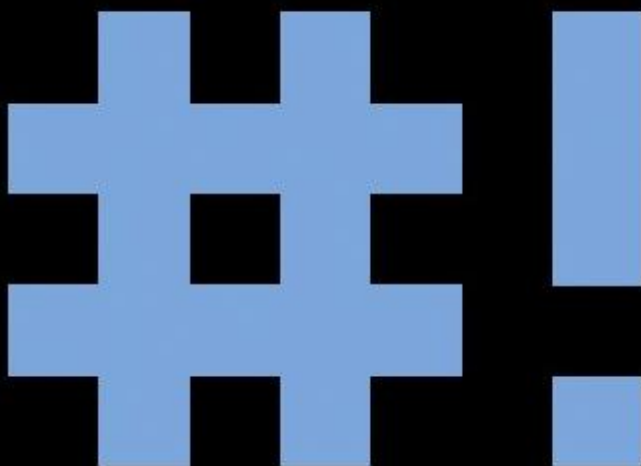
[Bash-скрипты, часть 8: язык обработки данных awk](#)

[Bash-скрипты, часть 9: регулярные выражения](#)

[Bash-скрипты, часть 10: практические примеры](#)

[Bash-скрипты, часть 11: expec и автоматизация интерактивных утилит](#)

Освоив предыдущие части этой серии материалов, вы узнали о том, что такое bash-скрипты, как их писать, как управлять потоком выполнения программы, как работать с файлами. Сегодня мы поговорим о том, как добавить скриптам интерактивности, оснастив их возможностями по получению данных от пользователя и по обработке этих данных.



Habrahabr10

Промо-код для скидки в 10% на наши виртуалы

Наиболее распространённый способ передачи данных сценариям заключается в использовании параметров командной строки. Вызвав сценарий с параметрами, мы передаём ему некую информацию, с которой он может работать. Выглядит это так:

```
$ ./myscript 10 20
```

В данном примере сценарию передано два параметра — «10» и «20». Всё это хорошо, но как прочесть данные в скрипте?

## Чтение параметров командной строки

Оболочка `bash` назначает специальным переменным, называемым позиционными параметрами, введённые при вызове скрипта параметры командной строки:

- `$0` — имя скрипта.
- `$1` — первый параметр.
- `$2` — второй параметр — и так далее, вплоть до переменной `$9`, в которую попадает девятый параметр.

Вот как можно использовать параметры командной строки в скрипте с помощью этих переменных:

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

```
echo $0
```

```
echo $1
```

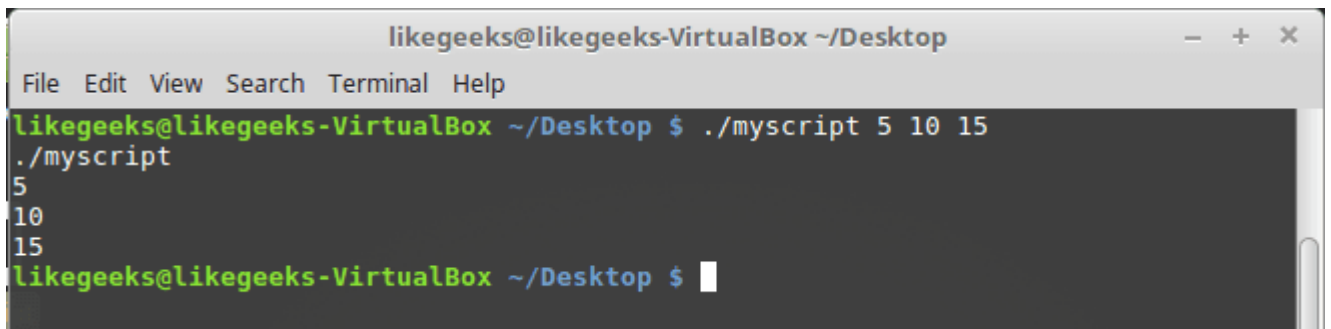
```
echo $2
```

```
echo $3
```

Запустим сценарий с параметрами:

```
./myscript 5 10 15
```

Вот что он выведет в консоль.



```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript 5 10 15
./myscript
5
10
15
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

*Вывод параметров, с которыми запущен скрипт*

Обратите внимание на то, что параметры командной строки разделяются пробелами.

Взглянем на ещё один пример использования параметров. Тут мы найдём сумму чисел, переданных сценарию:

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

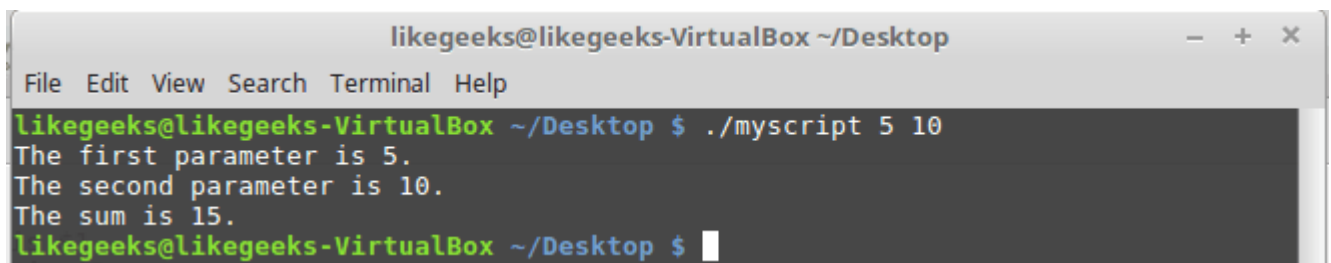
```
total=$(( $1 + $2 ))
```

```
echo The first parameter is $1.
```

```
echo The second parameter is $2.
```

```
echo The sum is $total.
```

Запустим скрипт и проверим результат вычислений.



```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript 5 10
The first parameter is 5.
The second parameter is 10.
The sum is 15.
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

*Сценарий, который находит сумму переданных ему чисел*

Параметры командной строки не обязательно должны быть числами. Сценариям

можно передавать и строки. Например, вот скрипт, работающий со строкой:

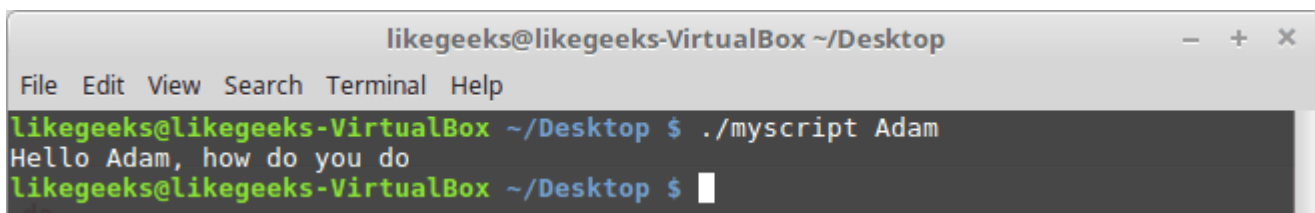
```
#!/bin/bash
```

```
echo Hello $1, how do you do
```

Запустим его:

```
./myscript Adam
```

Он выведет то, что мы от него ожидаем.

A screenshot of a terminal window titled 'likegeeks@likegeeks-VirtualBox ~/Desktop'. The window has a menu bar with 'File', 'Edit', 'View', 'Search', 'Terminal', and 'Help'. The terminal shows the command './myscript Adam' being executed, followed by the output 'Hello Adam, how do you do'. The prompt is 'likegeeks@likegeeks-VirtualBox ~/Desktop \$'.

*Сценарий, работающий со строковым параметром*

Что если параметр содержит пробелы, а нам надо обрабатывать его как самостоятельный фрагмент данных? Полагаем, если вы освоили предыдущие части этого руководства, ответ вы уже знаете. Заключается он в использовании кавычек.

Если скрипту надо больше девяти параметров, при обращении к ним номер в имени переменной надо заключать в фигурные скобки, например так:

```
${10}
```

## Проверка параметров

Если скрипт вызван без параметров, но для нормальной работы кода предполагается их наличие, возникнет ошибка. Поэтому рекомендуется всегда проверять наличие параметров, переданных сценарию при вызове. Например, это можно организовать так:

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

```
if [ -n "$1" ]
```

```
then
```

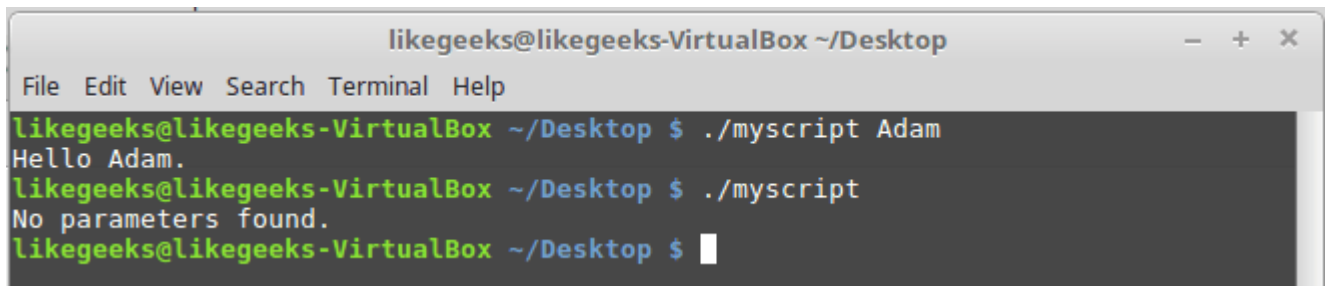
```
echo Hello $1.
```

```
else
```

```
echo "No parameters found. "
```

```
fi
```

Вызовем скрипт сначала с параметром, а потом без параметров.



```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript Adam
Hello Adam.
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript
No parameters found.
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

*Вызов скрипта, проверяющего наличие параметров командной строки*

## Подсчёт параметров

В скрипте можно подсчитать количество переданных ему параметров. оболочка `bash` предоставляет для этого специальную переменную. А именно, переменная `$#` содержит количество параметров, переданных сценарию при вызове.

Опробуем её:

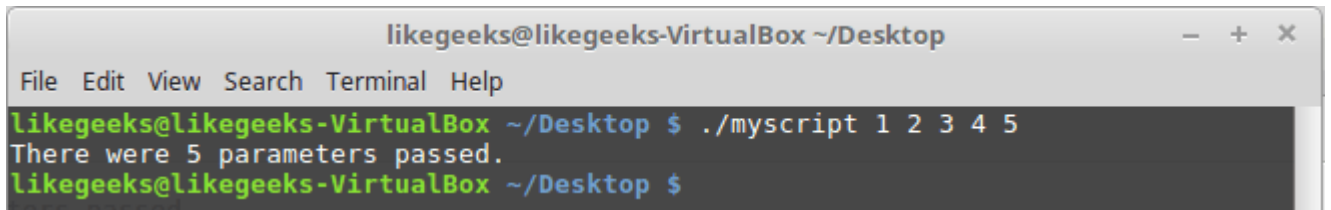
```
#!/bin/bash
```

```
echo There were $# parameters passed.
```

Вызовем сценарий.

```
./myscript 1 2 3 4 5
```

В результате скрипт сообщит о том, что ему передано 5 параметров.



```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript 1 2 3 4 5
There were 5 parameters passed.
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

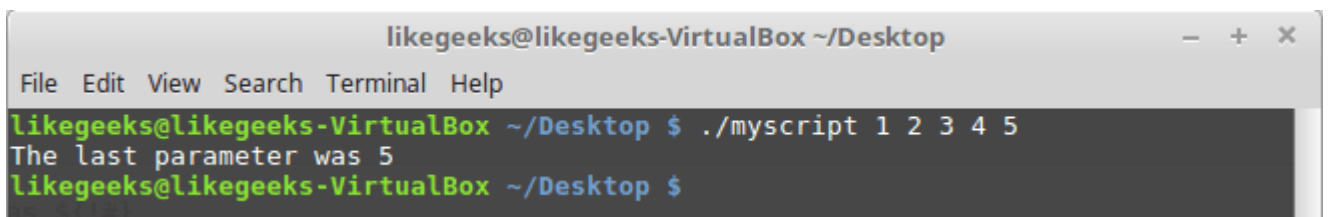
#### *Подсчёт количества параметров в скрипте*

Эта переменная даёт необычный способ получения последнего из переданных скрипту параметров, не требующий знания их количества. Вот как это выглядит:

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

```
echo The last parameter was ${!#}
```

Вызовем скрипт и посмотрим, что он выведет.



```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript 1 2 3 4 5
The last parameter was 5
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

#### *Обращение к последнему параметру*

## Захват всех параметров командной строки

В некоторых случаях нужно захватить все параметры, переданные скрипту. Для этого можно воспользоваться переменными `$*` и `$@`. Обе они содержат все параметры командной строки, что делает возможным доступ к тому, что передано сценарию, без использования позиционных параметров.

Переменная `$*` содержит все параметры, введённые в командной строке, в виде единого «слова».

В переменной `$@` параметры разбиты на отдельные «слова». Эти параметры можно перебирать в циклах.

Рассмотрим разницу между этими переменными на примерах. Сначала взглянем на их содержимое:

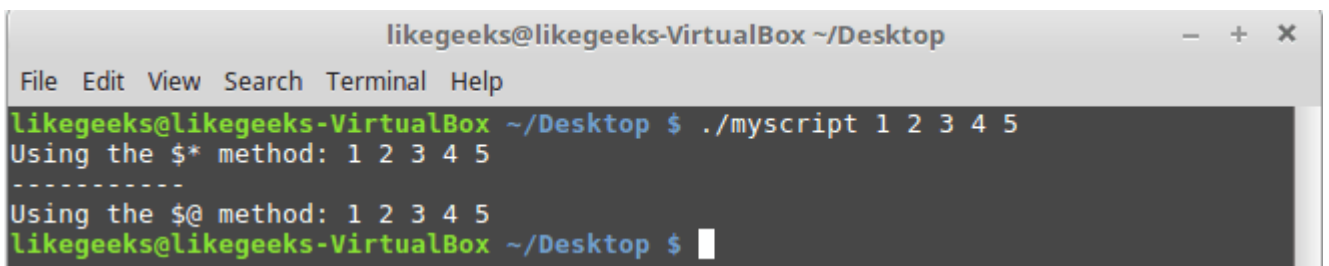
```
#!/bin/bash
```

```
echo "Using the \${*} method: ${*}"
```

```
echo "-----"
```

```
echo "Using the \${@} method: ${@}"
```

Вот вывод скрипта.



```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript 1 2 3 4 5
Using the ${*} method: 1 2 3 4 5
-----
Using the ${@} method: 1 2 3 4 5
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

*Переменные `${*}` и `${@}`*

Как видно, при выводе обеих переменных получается одно и то же. Теперь попробуем пройти по содержимому этих переменных в циклах для того, чтобы увидеть разницу между ними:

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

```
count=1
```

```
for param in "${*}"
```

```
do
```

```
echo "\${*} Parameter #${count} = $param"
```

```
count=$(( ${count} + 1 ))
```

```
done
```

```
count=1
```

```
for param in "$@"
```

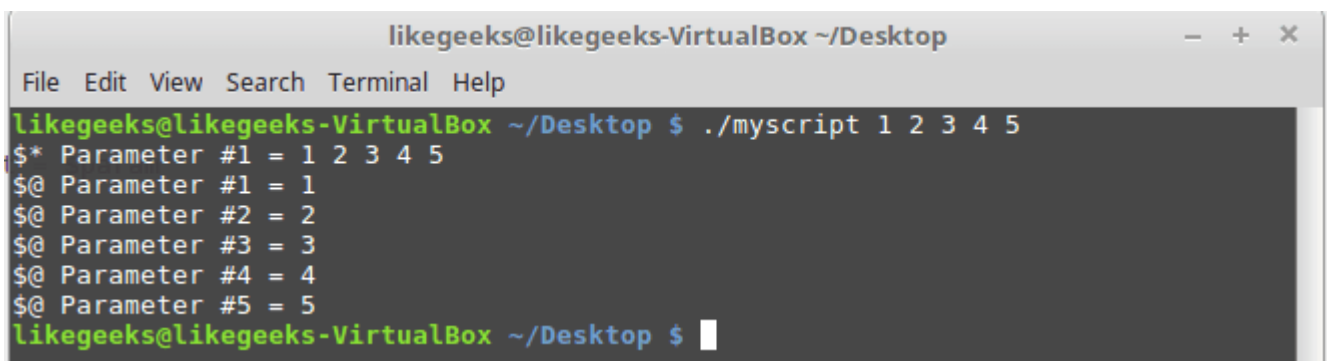
```
do
```

```
echo "\$@ Parameter #$count = $param"
```

```
count=$(( $count + 1 ))
```

```
done
```

Взгляните на то, что скрипт вывел в консоль. Разница между переменными вполне очевидна.



```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript 1 2 3 4 5
* Parameter #1 = 1 2 3 4 5
@ Parameter #1 = 1
@ Parameter #2 = 2
@ Parameter #3 = 3
@ Parameter #4 = 4
@ Parameter #5 = 5
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

*Разбор переменных `$*` и `$@` в цикле*

Переменная `$*` содержит все переданные скрипту параметры как единый фрагмент данных, в то время как в переменной `$@` они представлены самостоятельными значениями. Какой именно переменной воспользоваться — зависит от того, что именно нужно в конкретном сценарии.

## Команда `shift`

Использовать команду `shift` в `bash`-скриптах следует с осторожностью, так как она, в прямом смысле слова, сдвигает значения позиционных параметров.

Когда вы используете эту команду, она, по умолчанию, сдвигает значения позиционных параметров влево. Например, значение переменной `$3` становится значением переменной `$2`, значение `$2` переходит в `$1`, а то, что было до этого в `$1`, теряется. Обратите внимание на то, что при этом значение переменной `$0`, содержащей имя скрипта, не меняется.

Воспользовавшись командой `shift`, рассмотрим ещё один способ перебора



переданных скрипту параметров:

```
#!/bin/bash

count=1

while [ -n "$1" ]

do

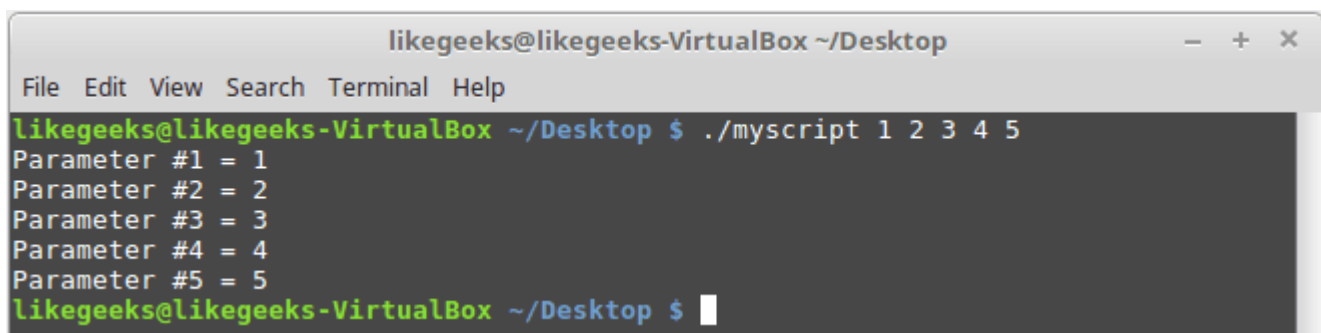
echo "Parameter #$count = $1"

count=$(( $count + 1 ))

shift

done
```

Скрипт задействует цикл `while`, проверяя длину значения первого параметра. Когда длина станет равна нулю, происходит выход из цикла. После проверки первого параметра и вывода его на экран, вызывается команда `shift`, которая сдвигает значения параметров на одну позицию.

A screenshot of a terminal window titled 'likegeeks@likegeeks-VirtualBox ~/Desktop'. The window shows the execution of a script named 'myscript' with five arguments: '1 2 3 4 5'. The output of the script is displayed line by line: 'Parameter #1 = 1', 'Parameter #2 = 2', 'Parameter #3 = 3', 'Parameter #4 = 4', and 'Parameter #5 = 5'. The prompt returns to the shell after the script finishes.

```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript 1 2 3 4 5
Parameter #1 = 1
Parameter #2 = 2
Parameter #3 = 3
Parameter #4 = 4
Parameter #5 = 5
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

*Использование команды `shift` для перебора параметров*

Используя команду `shift`, помните о том, что при каждом её вызове значение переменной `$1` безвозвратно теряется.

## Ключи командной строки

Ключи командной строки обычно выглядят как буквы, перед которыми ставится тире. Они служат для управления сценариями. Рассмотрим такой пример:

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

```
echo
```

```
while [ -n "$1" ]
```

```
do
```

```
case "$1" in
```

```
-a) echo "Found the -a option" ;;
```

```
-b) echo "Found the -b option" ;;
```

```
-c) echo "Found the -c option" ;;
```

```
*) echo "$1 is not an option" ;;
```

```
esac
```

```
shift
```

```
done
```

Запустим скрипт:

```
$ ./myscript -a -b -c -d
```

И проанализируем то, что он выведет в терминал.

```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript -a -b -c -d
Found the -a option
Found the -b option
Found the -c option
-d is not an option
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

### Обработка ключей в скрипте

В этом коде использована конструкция `case`, которая сверяет переданный ей ключ со списком обрабатываемых скриптом ключей. Если переданное значение нашлось в этом списке, выполняется соответствующая ветвь кода. Если при вызове скрипта будет использован любой ключ, обработка которого не предусмотрена, будет исполнена ветвь «\*».

## Как различать ключи и параметры

Часто при написании `bash`-скриптов возникает ситуация, когда надо использовать и параметры командной строки, и ключи. Стандартный способ это сделать заключается в применении специальной последовательности символов, которая сообщает скрипту о том, когда заканчиваются ключи и начинаются обычные параметры.

Эта последовательность — двойное тире (`--`). оболочка использует её для указания позиции, на которой заканчивается список ключей. После того, как скрипт обнаружит признак окончания ключей, то, что осталось, можно, не опасаясь ошибок, обрабатывать как параметры, а не как ключи. Рассмотрим пример:

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

```
while [ -n "$1" ]
```

```
do
```

```
case "$1" in
```

```
-a) echo "Found the -a option" ;;
```

```
-b) echo "Found the -b option";;
```

```
-c) echo "Found the -c option" ;;
```

```
--) shift
```

```
break ;;
```

```
*) echo "$1 is not an option";;
```

```
esac
```

```
shift
```

```
done
```

```
count=1
```

```
for param in $@
```

```
do
```

```
echo "Parameter #${count}: $param"
```

```
count=$(( $count + 1 ))
```

```
done
```

Этот сценарий использует команду `break` для прерывания цикла `while` при обнаружении в строке двойного тире.

Вот что получится после его вызова.

```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript -a -b -c -- 5 10 15
Found the -a option
Found the -b option
Found the -c option
Parameter #1: 5
Parameter #2: 10
Parameter #3: 15
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

### Обработка ключей и параметров командной строки

Как видно, когда скрипт, разбирая переданные ему данные, находит двойное тире, он завершает обработку ключей и считает всё, что ещё не обработано, параметрами.

## Обработка ключей со значениями

По мере усложнения ваших скриптов, вы столкнётесь с ситуациями, когда обычных ключей уже недостаточно, а значит, нужно будет использовать ключи с некими значениями. Например, вызов сценария в котором используется подобная возможность, выглядит так:

```
./myscript -a test1 -b -c test2
```

Скрипт должен уметь определять, когда вместе с ключами командной строки используются дополнительные параметры:

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

```
while [ -n "$1" ]
```

```
do
```

```
case "$1" in
```

```
-a) echo "Found the -a option";;
```

```
-b) param="$2"
```

```
echo "Found the -b option, with parameter value $param"
```

```
shift ;;
```

```
-c) echo "Found the -c option";;
```

```
--) shift
```

```
break ;;
```

```
*) echo "$1 is not an option";;
```

```
esac
```

```
shift
```

```
done
```

```
count=1
```

```
for param in "$@"
```

```
do
```

```
echo "Parameter #$count: $param"
```

```
count=$(( $count + 1 ))
```

```
done
```

Вызовем этот скрипт в таком виде:

```
./myscript -a -b test1 -d
```

Посмотрим на результаты его работы.

```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript -a -b 15 -d
Found the -a option
Found the -b option, with parameter value 15
-d is not an option
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

### Обработка параметров ключей

В данном примере в конструкции `case` обрабатываются три ключа. Ключ `-b` требует наличия дополнительного параметра. Так как обрабатываемый ключ находится в переменной `$1`, соответствующий ему параметр будет находиться в `$2` (тут используется команда `shift`, поэтому, по мере обработки, всё, что передано сценарию, сдвигается влево). Когда с этим мы разобрались, осталось лишь извлечь значение переменной `$2` и у нас будет параметр нужного ключа. Конечно, тут понадобится ещё одна команда `shift` для того, чтобы следующий ключ попал в `$1`.

## Использование стандартных ключей

При написании `bash`-скриптов вы можете выбирать любые буквы для ключей командной строки и произвольно задавать реакцию скрипта на эти ключи. Однако, в мире Linux значения некоторых ключей стали чем-то вроде стандарта, которого полезно придерживаться. Вот список этих ключей:

- a Вывести все объекты.
- c Произвести подсчёт.
- d Указать директорию.
- e Развернуть объект.
- f Указать файл, из которого нужно прочитывать данные.
- h Вывести справку по команде.
- i Игнорировать регистр символов.
- l Выполнить полноформатный вывод данных.
- n Использовать неинтерактивный (пакетный) режим.
- o Позволяет указать файл, в который нужно перенаправить вывод.
- q Выполнить скрипт в `quiet`-режиме.
- r Обрабатывать папки и файлы рекурсивно.
- s Выполнить скрипт в `silent`-режиме.
- v Выполнить многословный вывод.
- x Исключить объект.
- y Ответить «yes» на все вопросы.

Если вы работаете в Linux, вам, скорее всего, знакомы многие из этих ключей. Используя их в общепринятом значении в своих скриптах, вы поможете пользователям взаимодействовать с ними, не беспокоясь о чтении документации.

## Получение данных от пользователя

Ключи и параметры командной строки — это отличный способ получить данные от того, кто пользуется скриптом, однако в некоторых случаях нужно больше интерактивности.

Иногда сценарии нуждаются в данных, которые пользователь должен ввести во время выполнения программы. Именно для этой цели в оболочке `bash` имеется команда `read`.

Эта команда позволяет принимать введенные данные либо со стандартного ввода (с клавиатуры), либо используя другие дескрипторы файлов. После получения данных, эта команда помещает их в переменную:

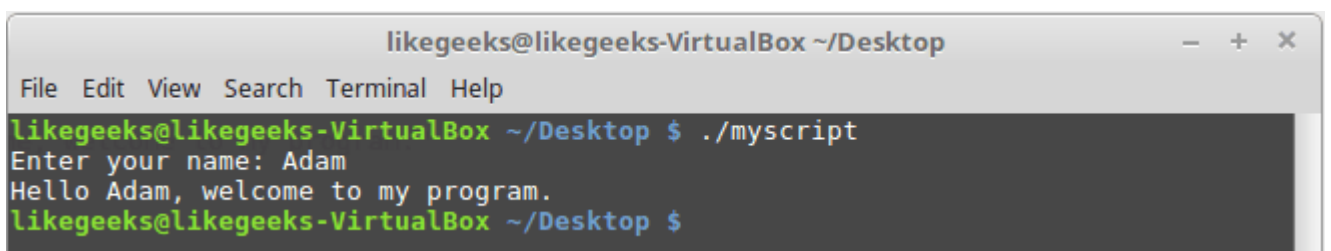
```
#!/bin/bash
```

```
echo -n "Enter your name: "
```

```
read name
```

```
echo "Hello $name, welcome to my program."
```

Обратите внимание на то, что команда `echo`, которая выводит приглашение, вызывается с ключом `-n`. Это приводит к тому, что в конце приглашения не выводится знак перевода строки, что позволяет пользователю скрипта вводить данные там же, где расположено приглашение, а не на следующей строке.

A screenshot of a terminal window titled "likegeeks@likegeeks-VirtualBox ~/Desktop". The window has a menu bar with "File", "Edit", "View", "Search", "Terminal", and "Help". The terminal shows the command `./myscript` being executed. The output is "Enter your name: Adam" followed by "Hello Adam, welcome to my program." on the same line. The prompt then returns to `likegeeks@likegeeks-VirtualBox ~/Desktop $`.

### Обработка пользовательского ввода

При вызове `read` можно указывать и несколько переменных:

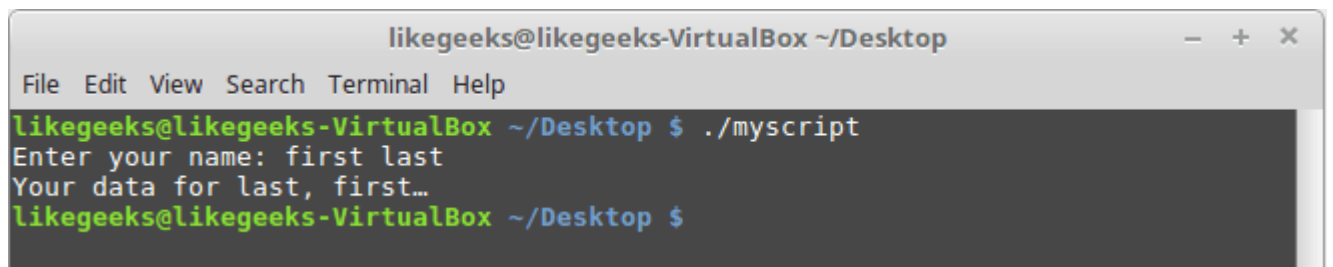
```
#!/bin/bash
```

```
read -p "Enter your name: " first last
```



```
echo "Your data for $last, $first..."
```

Вот что выведет скрипт после запуска.

A terminal window titled 'likegeeks@likegeeks-VirtualBox ~/Desktop'. It shows the execution of a script './myscript'. The prompt is 'likegeeks@likegeeks-VirtualBox ~/Desktop \$'. The script outputs 'Enter your name: first last' and 'Your data for last, first...'. The prompt returns to 'likegeeks@likegeeks-VirtualBox ~/Desktop \$'.

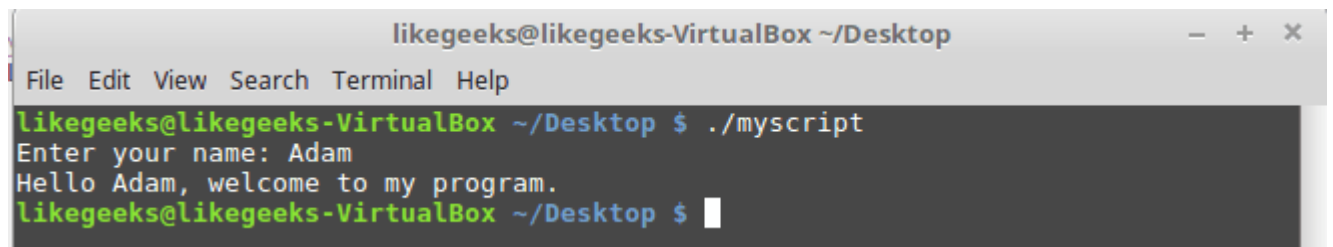
*Несколько переменных в команде read*

Если, вызвав `read`, не указывать переменную, данные, введённые пользователем, будут помещены в специальную переменную среды `REPLY`:

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

```
read -p "Enter your name: "
```

```
echo Hello $REPLY, welcome to my program.
```

A terminal window titled 'likegeeks@likegeeks-VirtualBox ~/Desktop'. It shows the execution of a script './myscript'. The prompt is 'likegeeks@likegeeks-VirtualBox ~/Desktop \$'. The script outputs 'Enter your name: Adam' and 'Hello Adam, welcome to my program.'. The prompt returns to 'likegeeks@likegeeks-VirtualBox ~/Desktop \$'.

*Использование переменной среды REPLY*

Если скрипт должен продолжать выполнение независимо от того, введёт пользователь какие-то данные или нет, вызывая команду `read` можно воспользоваться ключом `-t`. А именно, параметр ключа задаёт время ожидания ввода в секундах:

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

```
if read -t 5 -p "Enter your name: " name
```

```
then
```

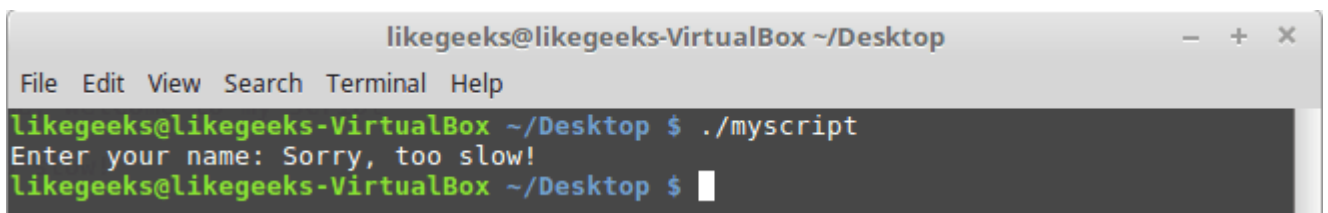
```
echo "Hello $name, welcome to my script"
```

```
else
```

```
echo "Sorry, too slow! "
```

```
fi
```

Если данные не будут введены в течение 5 секунд, скрипт выполнит ветвь условного оператора `else`, выведя извинения.



```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript
Enter your name: Sorry, too slow!
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

*Ограничение времени на ввод данных*

## Ввод паролей

Иногда то, что вводит пользователь в ответ на вопрос скрипта, лучше на экране не показывать. Например, так обычно делают, запрашивая пароли. Ключ - `s` команды `read` предотвращает отображение на экране данных, вводимых с клавиатуры. На самом деле, данные выводятся, но команда `read` делает цвет текста таким же, как цвет фона.

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

```
read -s -p "Enter your password: " pass
```

```
echo "Is your password really $pass? "
```

Вот как отработает этот скрипт.

```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript
Enter your password: Is your password really secretpass?
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

*Ввод конфиденциальных данных*

## Чтение данных из файла

Команда `read` может, при каждом вызове, читать одну строку текста из файла. Когда в файле больше не останется непрочитанных строк, она просто остановится. Если нужно получить в скрипте всё содержимое файла, можно, с помощью конвейера, передать результаты вызова команды `cat` для файла, конструкции `while`, которая содержит команду `read` (конечно, использование команды `cat` выглядит примитивно, но наша цель — показать всё максимально просто, ориентируясь на новичков; опытные пользователи, уверены, это поймут).

Напишем скрипт, в котором используется только что описанный подход к чтению файлов.

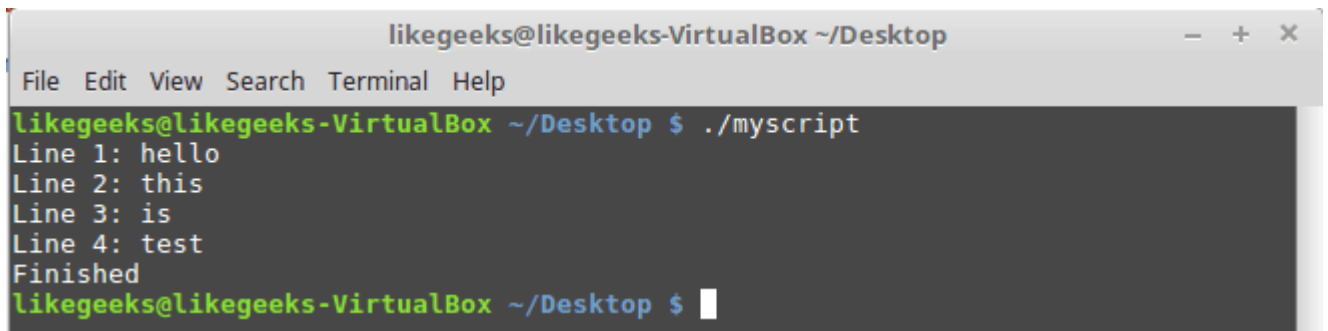
```
#!/bin/bash

count=1

cat myfile | while read line
do
    echo "Line $count: $line"
    count=$(( $count + 1 ))
done

echo "Finished"
```

Посмотрим на него в деле.



```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript
Line 1: hello
Line 2: this
Line 3: is
Line 4: test
Finished
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

### *Чтение данных из файла*

Тут мы передали в цикл `while` содержимое файла и перебрали все строки этого файла, выводя номер и содержимое каждой из них.

## Итоги

Сегодня мы разобрали работу с ключами и параметрами командной строки. Без этих средств диапазон использования скриптов оказывается чрезвычайно узким. Даже если скрипт написан, что называется, «для себя». Тут же мы рассмотрели подходы к получению данных от пользователя во время выполнения программы — это делает сценарии интерактивными.

В следующий раз поговорим об операциях ввода и вывода.

Уважаемые читатели! Спасибо вам за то, что делитесь опытом в комментариях к предыдущим частям этого цикла материалов. Если вам есть что сказать об обработке всего того, что можно передать в скрипт при запуске или во время его работы, уверены, многим будет интересно об этом почитать.