[Параметры и ключи командной строки](https://habrahabr.ru/company/ruvds/blog/326328/)

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

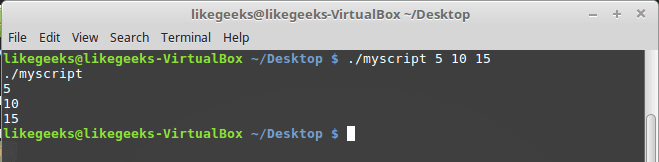
echo $2

echo $3

Запустим сценарий с параметрами:

./myscript 5 10 15

Вот что он выведет в консоль.



*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

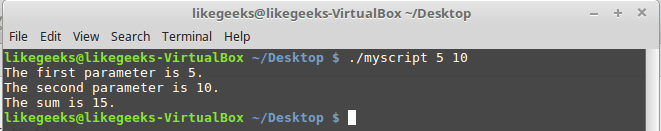
total=$[ $1 + $2 ]

echo The first parameter is $1.

echo The second parameter is $2.

echo The sum is $total.

Запустим скрипт и проверим результат вычислений.



*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:

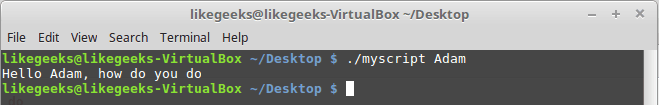
#!/bin/bash

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

Запустим его:

./myscript Adam

Он выведет то, что мы от него ожидаем.



*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**

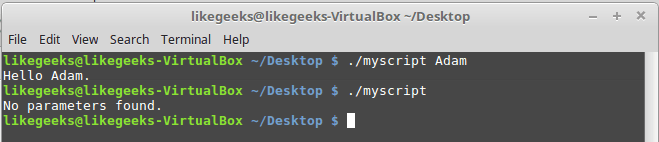
echo Hello $1.

**else**

echo "No parameters found. "

**fi**

Вызовем скрипт сначала с параметром, а потом без параметров.



*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:

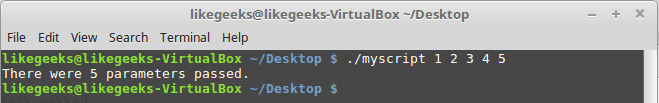
#!/bin/bash

echo There were $# parameters passed.

Вызовем сценарий.

./myscript 1 2 3 4 5

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

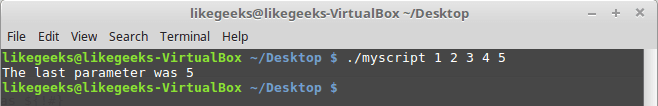


*Подсчёт количества параметров в скрипте*  
  
Эта переменная даёт необычный способ получения последнего из переданных скрипту параметров, не требующий знания их количества. Вот как это выглядит:

#!/bin/bash

echo The last parameter was ${!#}

Вызовем скрипт и посмотрим, что он выведет.



*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:

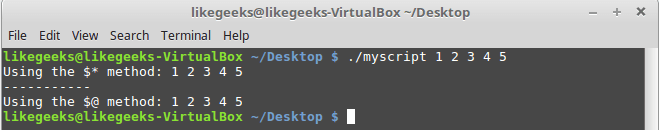
#!/bin/bash

echo "Using the \$\* method: $\*"

echo "-----------"

echo "Using the \$@ method: $@"

Вот вывод скрипта.



*Переменные $\* и $@*  
  
Как видно, при выводе обеих переменных получается одно и то же. Теперь попробуем пройтись 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** "$@"

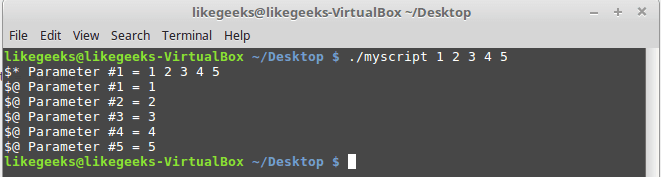
**do**

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

count=$(( $count + 1 ))

**done**

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



*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**

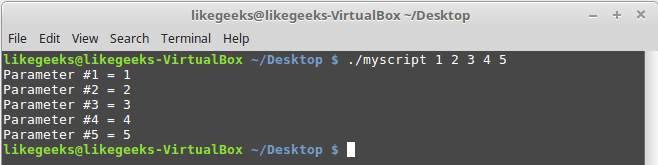
echo "Parameter #$count = $1"

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.



*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**

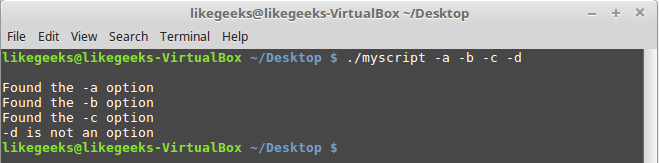
shift

**done**

Запустим скрипт:

$ ./myscript –a –b –c –d

И проанализируем то, что он выведет в терминал.



*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** $@

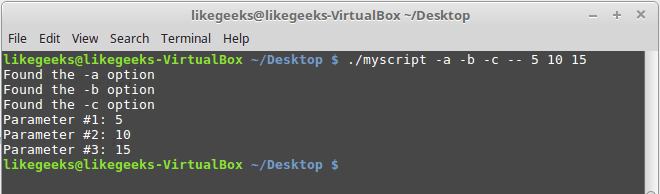
**do**

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.



*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"

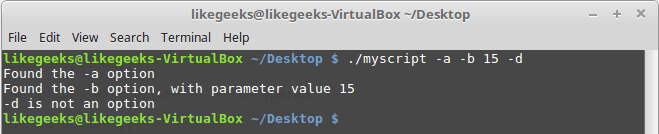
count=$(( $count + 1 ))

**done**

Вызовем этот скрипт в таком виде:

./myscript -a -b test1 -d

Посмотрим на результаты его работы.



*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 прочитать 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. Использовав 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 введённые данные либо со стандартного ввода (с клавиатуры), либо используя другие дескрипторы файлов. После получения данных, эта команда помещает их в переменную:

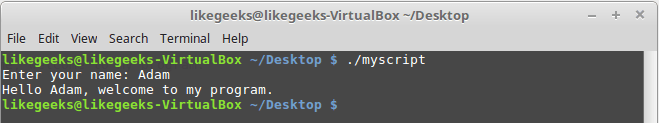
#!/bin/bash

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.



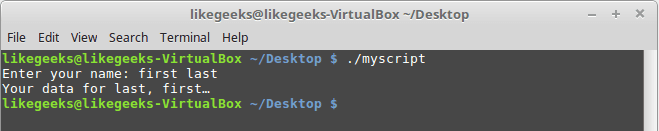
*Обработка пользовательского ввода*  
  
При вызове read можно указывать и несколько переменных:

#!/bin/bash

read -p "Enter your name: " first last

echo "Your data for $last, $first…"

Вот что выведет скрипт после запуска.

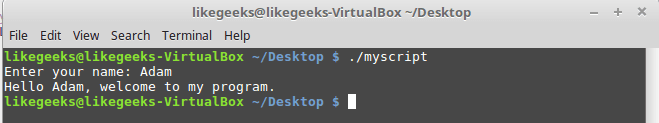


*Несколько переменных в команде read*  
  
Если, вызвав read, не указывать переменную, данные, введённые пользователем, будут помещены в специальную переменную среды REPLY:

#!/bin/bash

read -p "Enter your name: "

echo Hello $REPLY, welcome to my program.



*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.



*Ограничение времени на ввод данных*

Ввод паролей

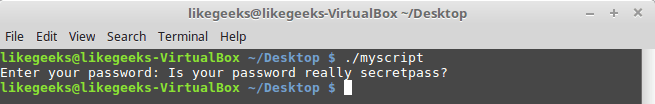
Иногда то, что вводит пользователь в ответ на вопрос скрипта, лучше на экране не показывать. Например, так обычно делают, запрашивая пароли. Ключ -s команды read предотвращает отображение на экране данных, вводимых с клавиатуры. На самом деле, данные выводятся, но команда read делает цвет текста таким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

#!/bin/bash

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

echo "Is your password really $pass? "

Вот как отработает этот скрипт.



*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**

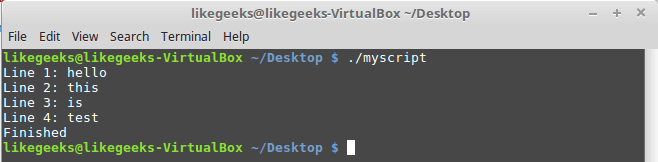
echo "Line $count: $line"

count=$(( $count + 1 ))

**done**

echo "Finished"

Посмотрим на него в деле.



*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.

[Ввод и вывод](https://habrahabr.ru/company/ruvds/blog/326594/)

## Стандартные дескрипторы файлов

Всё в Linux — это файлы, в том числе — ввод и вывод. Операционная система идентифицирует файлы с использованием дескрипторов.  
  
Каждому процессу позволено иметь до девяти открытых дескрипторов файлов. Оболочка bash резервирует первые три дескриптора с идентификаторами 0, 1 и 2. Вот что они означают.

* 0, STDIN — стандартный поток ввода.
* 1, STDOUT — стандартный поток вывода.
* 2, STDERR — стандартный поток ошибок.

Эти три специальных дескриптора обрабатывают ввод и вывод данных в сценарии.  
Вам нужно как следует разобраться в стандартных потоках. Их можно сравнить с фундаментом, на котором строится взаимодействие скриптов с внешним миром. Рассмотрим подробности о них.

## STDIN

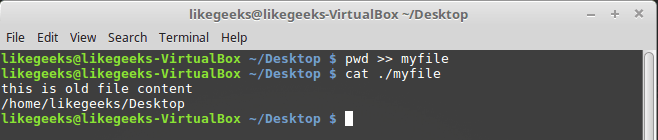
STDIN — это стандартный поток ввода оболочки. Для терминала стандартный ввод — это клавиатура. Когда в сценариях используют символ перенаправления ввода — <, Linux заменяет дескриптор файла стандартного ввода на тот, который указан в команде. Система читает файл и обрабатывает данные так, будто они введены с клавиатуры.  
  
Многие команды bash принимают ввод из STDIN, если в командной строке не указан файл, из которого надо брать данные. Например, это справедливо для команды cat.  
  
Когда вы вводите команду cat в командной строке, не задавая параметров, она принимает ввод из STDIN. После того, как вы вводите очередную строку, cat просто выводит её на экран.

## STDOUT

STDOUT — стандартный поток вывода оболочки. По умолчанию это — экран. Большинство bash-команд выводят данные в STDOUT, что приводит к их появлению в консоли. Данные можно перенаправить в файл, присоединяя их к его содержимому, для этого служит команда >>.  
  
Итак, у нас есть некий файл с данными, к которому мы можем добавить другие данные с помощью этой команды:

pwd >> myfile

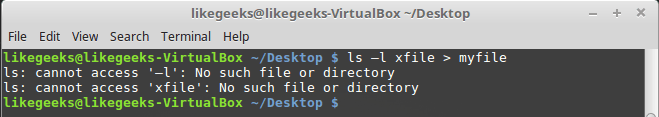
То, что выведет pwd, будет добавлено к файлу myfile, при этом уже имеющиеся в нём данные никуда не денутся.



*Перенаправление вывода команды в файл*  
  
Пока всё хорошо, но что если попытаться выполнить что-то вроде показанного ниже, обратившись к несуществующему файлу xfile, задумывая всё это для того, чтобы в файл myfile попало сообщение об ошибке.

**ls** –l xfile > myfile

После выполнения этой команды мы увидим сообщения об ошибках на экране.



*Попытка обращения к несуществующему файлу*  
  
При попытке обращения к несуществующему файлу генерируется ошибка, но оболочка не перенаправила сообщения об ошибках в файл, выведя их на экран. Но мы-то хотели, чтобы сообщения об ошибках попали в файл. Что делать? Ответ прост — воспользоваться третьим стандартным дескриптором.

## STDERR

STDERR представляет собой стандартный поток ошибок оболочки. По умолчанию этот дескриптор указывает на то же самое, на что указывает STDOUT, именно поэтому при возникновении ошибки мы видим сообщение на экране.  
  
Итак, предположим, что надо перенаправить сообщения об ошибках, скажем, в лог-файл, или куда-нибудь ещё, вместо того, чтобы выводить их на экран.

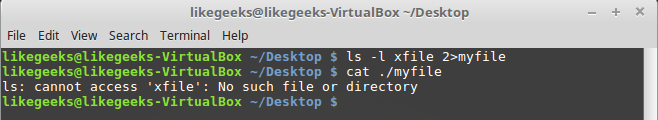
### **▍Перенаправление потока ошибок**

Как вы уже знаете, дескриптор файла STDERR — 2. Мы можем перенаправить ошибки, разместив этот дескриптор перед командой перенаправления:

ls -l xfile 2>myfile

cat ./myfile

Сообщение об ошибке теперь попадёт в файл myfile.

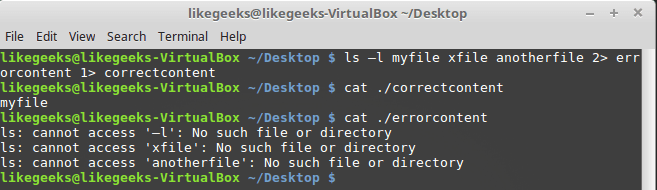


*Перенаправление сообщения об ошибке в файл*

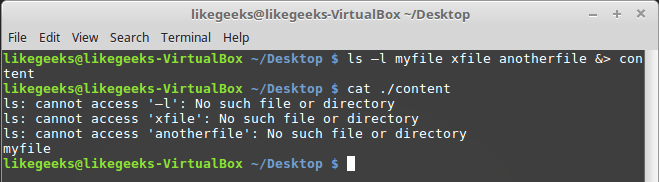
### **▍Перенаправление потоков ошибок и вывода**

При написании сценариев командной строки может возникнуть ситуация, когда нужно организовать и перенаправление сообщений об ошибках, и перенаправление стандартного вывода. Для того, чтобы этого добиться, нужно использовать команды перенаправления для соответствующих дескрипторов с указанием файлов, куда должны попадать ошибки и стандартный вывод:

ls –l myfile xfile anotherfile 2> errorcontent 1> correctcontent



*Перенаправление ошибок и стандартного вывода*  
  
Оболочка перенаправит то, что команда ls обычно отправляет в STDOUT, в файл correctcontent благодаря конструкции 1>. Сообщения об ошибках, которые попали бы в STDERR, оказываются в файле errorcontent из-за команды перенаправления 2>.  
  
Если надо, и STDERR, и STDOUT можно перенаправить в один и тот же файл, воспользовавшись командой &>:



*Перенаправление STDERR и STDOUT в один и тот же файл*  
  
После выполнения команды то, что предназначено для STDERR и STDOUT, оказывается в файле content.

## Перенаправление вывода в скриптах

Существует два метода перенаправления вывода в сценариях командной строки:

* Временное перенаправление, или перенаправление вывода одной строки.
* Постоянное перенаправление, или перенаправление всего вывода в скрипте либо в какой-то его части.

### **▍Временное перенаправление вывода**

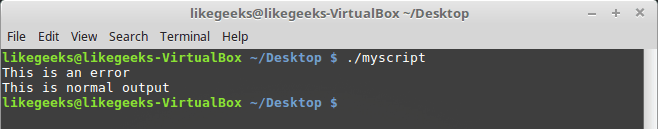
В скрипте можно перенаправить вывод отдельной строки в STDERR. Для того, чтобы это сделать, достаточно использовать команду перенаправления, указав дескриптор STDERR, при этом перед номером дескриптора надо поставить символ амперсанда (&):

#!/bin/bash

echo "This is an error" >&2

echo "This is normal output"

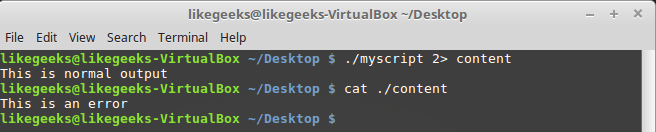
Если запустить скрипт, обе строки попадут на экран, так как, как вы уже знаете, по умолчанию ошибки выводятся туда же, куда и обычные данные.



*Временное перенаправление*  
  
Запустим скрипт так, чтобы вывод STDERR попадал в файл.

./myscript 2> myfile

Как видно, теперь обычный вывод делается в консоль, а сообщения об ошибках попадают в файл.



*Сообщения об ошибках записываются в файл*

### **▍Постоянное перенаправление вывода**

Если в скрипте нужно перенаправлять много выводимых на экран данных, добавлять соответствующую команду к каждому вызову echo неудобно. Вместо этого можно задать перенаправление вывода в определённый дескриптор на время выполнения скрипта, воспользовавшись командой exec:

#!/bin/bash

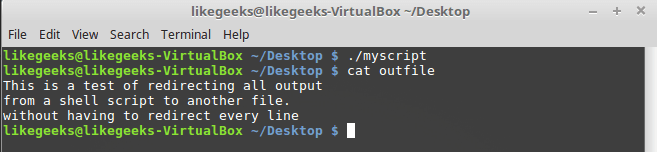
exec 1>outfile

echo "This is a test of redirecting all output"

echo "from a shell script to another file."

echo "without having to redirect every line"

Запустим скрипт.



*Перенаправление всего вывода в файл*  
  
Если просмотреть файл, указанный в команде перенаправления вывода, окажется, что всё, что выводилось командами echo, попало в этот файл.  
  
Команду exec можно использовать не только в начале скрипта, но и в других местах:

#!/bin/bash

exec 2>myerror

echo "This is the start of the script"

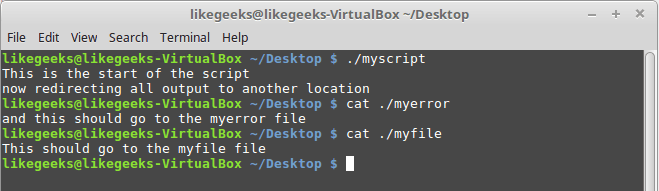
echo "now redirecting all output to another location"

exec 1>myfile

echo "This should go to the myfile file"

echo "and this should go to the myerror file" >&2

Вот что получится после запуска скрипта и просмотра файлов, в которые мы перенаправляли вывод.



*Перенаправление вывода в разные файлы*  
  
Сначала команда exec задаёт перенаправление вывода из STDERR в файл myerror. Затем вывод нескольких команд echo отправляется в STDOUT и выводится на экран. После этого команда exec задаёт отправку того, что попадает в STDOUT, в файл myfile, и, наконец, мы пользуемся командой перенаправления в STDERR в команде echo, что приводит к записи соответствующей строки в файл myerror.  
  
Освоив это, вы сможете перенаправлять вывод туда, куда нужно. Теперь поговорим о перенаправлении ввода.

## Перенаправление ввода в скриптах

Для перенаправления ввода можно воспользоваться той же методикой, которую мы применяли для перенаправления вывода. Например, команда exec позволяет сделать источником данных для STDIN какой-нибудь файл:

exec 0< myfile

Эта команда указывает оболочке на то, что источником вводимых данных должен стать файл myfile, а не обычный STDIN. Посмотрим на перенаправление ввода в действии:

#!/bin/bash

exec 0< testfile

count=1

**while** read line

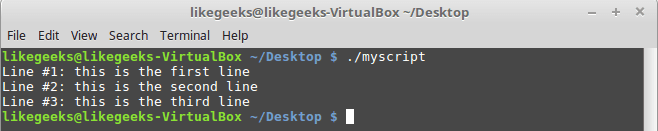
**do**

echo "Line #$count: $line"

count=$(( $count + 1 ))

**done**

Вот что появится на экране после запуска скрипта.



*Перенаправление ввода*  
  
В одном из предыдущих материалов вы узнали о том, как использовать команду read для чтения данных, вводимых пользователем с клавиатуры. Если перенаправить ввод, сделав источником данных файл, то команда read, при попытке прочитать данные из STDIN, будет читать их из файла, а не с клавиатуры.  
  
Некоторые администраторы Linux используют этот подход для чтения и последующей обработки лог-файлов.

## Создание собственного перенаправления вывода

Перенаправляя ввод и вывод в сценариях, вы не ограничены тремя стандартными дескрипторами файлов. Как уже говорилось, можно иметь до девяти открытых дескрипторов. Остальные шесть, с номерами от 3 до 8, можно использовать для перенаправления ввода или вывода. Любой из них можно назначить файлу и использовать в коде скрипта.  
  
Назначить дескриптор для вывода данных можно, используя команду exec:

#!/bin/bash

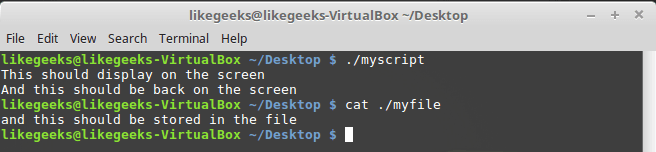
exec 3>myfile

echo "This should display on the screen"

echo "and this should be stored in the file" >&3

echo "And this should be back on the screen"

После запуска скрипта часть вывода попадёт на экран, часть — в файл с дескриптором 3.



*Перенаправление вывода, используя собственный дескриптор*

## Создание дескрипторов файлов для ввода данных

Перенаправить ввод в скрипте можно точно так же, как и вывод. Сохраните STDIN в другом дескрипторе, прежде чем перенаправлять ввод данных.  
  
После окончания чтения файла можно восстановить STDIN и пользоваться им как обычно:

#!/bin/bash

exec 6<&0

exec 0< myfile

count=1

**while** read line

**do**

echo "Line #$count: $line"

count=$(( $count + 1 ))

**done**

exec 0<&6

read -p "Are you done now? " answer

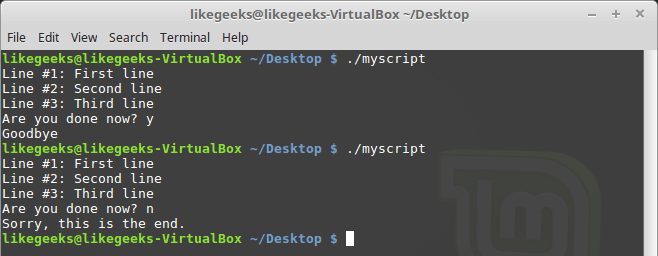
**case** $answer **in**

y) echo "Goodbye";;

n) echo "Sorry, this is the end.";;

**esac**

Испытаем сценарий.



*Перенаправление ввода*  
  
В этом примере дескриптор файла 6 использовался для хранения ссылки на STDIN. Затем было сделано перенаправление ввода, источником данных для STDIN стал файл. После этого входные данные для команды read поступали из перенаправленного STDIN, то есть из файла.  
  
После чтения файла мы возвращаем STDIN в исходное состояние, перенаправляя его в дескриптор 6. Теперь, для того, чтобы проверить, что всё работает правильно, скрипт задаёт пользователю вопрос, ожидает ввода с клавиатуры и обрабатывает то, что введено.

## Закрытие дескрипторов файлов

Оболочка автоматически закрывает дескрипторы файлов после завершения работы скрипта. Однако, в некоторых случаях нужно закрывать дескрипторы вручную, до того, как скрипт закончит работу. Для того, чтобы закрыть дескриптор, его нужно перенаправить в &-. Выглядит это так:

#!/bin/bash

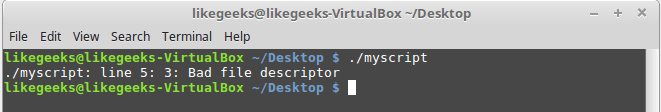
exec 3> myfile

echo "This is a test line of data" >&3

exec 3>&-

echo "This won't work" >&3

После исполнения скрипта мы получим сообщение об ошибке.



*Попытка обращения к закрытому дескриптору файла*  
  
Всё дело в том, что мы попытались обратиться к несуществующему дескриптору.  
  
Будьте внимательны, закрывая дескрипторы файлов в сценариях. Если вы отправляли данные в файл, потом закрыли дескриптор, потом — открыли снова, оболочка заменит существующий файл новым. То есть всё то, что было записано в этот файл ранее, будет утеряно.

## Сигналы Linux

В Linux существует более трёх десятков сигналов, которые генерирует система или приложения. Вот список наиболее часто используемых, которые наверняка пригодятся при разработке сценариев командной строки.

|  |  |  |
| --- | --- | --- |
| **Код сигнала** | **Название** | **Описание** |
| 1 | SIGHUP | Закрытие терминала |
| 2 | SIGINT | Сигнал остановки процесса пользователем с терминала (CTRL + C) |
| 3 | SIGQUIT | Сигнал остановки процесса пользователем с терминала (CTRL + \) с дампом памяти |
| 9 | SIGKILL | Безусловное завершение процесса |
| 15 | SIGTERM | Сигнал запроса завершения процесса |
| 17 | SIGSTOP | Принудительная приостановка выполнения процесса, но не завершение его работы |
| 18 | SIGTSTP | Приостановка процесса с терминала (CTRL + Z), но не завершение работы |
| 19 | SIGCONT | Продолжение выполнения ранее остановленного процесса |

Если оболочка bash получает сигнал SIGHUP когда вы закрываете терминал, она завершает работу. Перед выходом она отправляет сигнал SIGHUP всем запущенным в ней процессам, включая выполняющиеся скрипты.  
  
Сигнал SIGINT приводит к временной остановке работы. Ядро Linux перестаёт выделять оболочке процессорное время. Когда это происходит, оболочка уведомляет процессы, отправляя им сигнал SIGINT.  
  
Bash-скрипты не контролируют эти сигналы, но они могут распознавать их и выполнять некие команды для подготовки скрипта к последствиям, вызываемым сигналами.

## Отправка сигналов скриптам

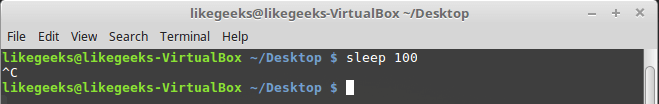
Оболочка bash позволяет вам отправлять скриптам сигналы, пользуясь комбинациями клавиш на клавиатуре. Это оказывается очень кстати если нужно временно остановить выполняющийся скрипт или завершить его работу.

### **Завершение работы процесса**

Комбинация клавиш CTRL + C генерирует сигнал SIGINT и отправляет его всем процессам, выполняющимся в оболочке, что приводит к завершению их работы.  
  
Выполним в оболочке такую команду:

$ **sleep** 100

После этого завершим её работу комбинацией клавиш CTRL + C.



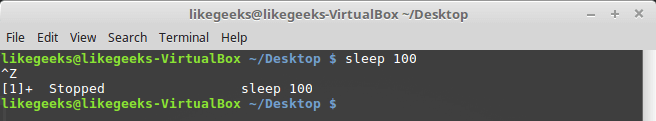
*Завершение работы процесса с клавиатуры*

### **Временная остановка процесса**

Комбинация клавиш CTRL + Z позволяет сгенерировать сигнал SIGTSTP, который приостанавливает работу процесса, но не завершает его выполнение. Такой процесс остаётся в памяти, его работу можно возобновить. Выполним в оболочке команду:

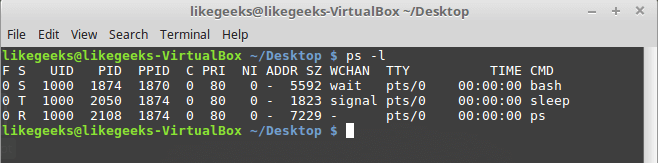
$ **sleep** 100

И временно остановим её комбинацией клавиш CTRL + Z.



*Приостановка процесса*  
  
Число в квадратных скобках — это номер задания, который оболочка назначает процессу. Оболочка рассматривает процессы, выполняющиеся в ней, как задания с уникальными номерами. Первому процессу назначается номер 1, второму — 2, и так далее.  
  
Если вы приостановите задание, привязанное к оболочке, и попытаетесь выйти из неё, bash выдаст предупреждение.  
  
Просмотреть приостановленные задания можно такой командой:

**ps**  –l



*Список заданий*  
  
В колонке S, выводящей состояние процесса, для приостановленных процессов выводится T. Это указывает на то, что команда либо приостановлена, либо находится в состоянии трассировки.  
  
Если нужно завершить работу приостановленного процесса, можно воспользоваться командой kill.   
  
Выглядит её вызов так:

kill processID

## Перехват сигналов

Для того, чтобы включить в скрипте отслеживание сигналов Linux, используется команда trap. Если скрипт получает сигнал, указанный при вызове этой команды, он обрабатывает его самостоятельно, при этом оболочка такой сигнал обрабатывать не будет.  
  
Команда trap позволяет скрипту реагировать на сигналы, в противном случае их обработка выполняется оболочкой без его участия.  
  
Рассмотрим пример, в котором показано, как при вызове команды trap задаётся код, который надо выполнить, и список сигналов, разделённых пробелами, которые мы хотим перехватить. В данном случае это всего один сигнал:

#!/bin/bash

trap "echo ' Trapped Ctrl-C'" SIGINT

echo This is a test script

count=1

**while** [ $count -le 10 ]

**do**

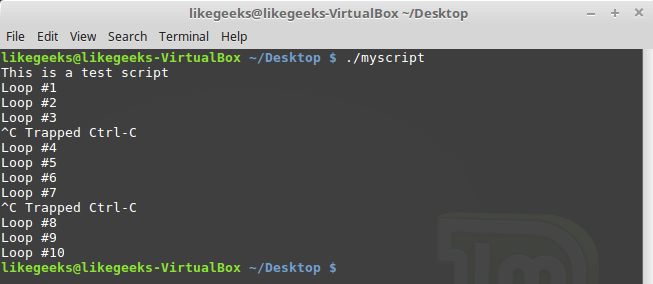
echo "Loop #$count"

sleep 1

count=$(( $count + 1 ))

**done**

Команда trap, использованная в этом примере, выводит текстовое сообщение всякий раз, когда она обнаруживает сигнал SIGINT, который можно сгенерировать, нажав Ctrl + C на клавиатуре.



*Перехват сигналов*  
  
Каждый раз, когда вы нажимаете клавиши CTRL + C, скрипт выполняет команду echo, указанную при вызове trace вместо того, чтобы позволить оболочке завершит его работу.

## Перехват сигнала выхода из скрипта

Перехватить сигнал выхода из скрипта можно, использовав при вызове команды trap имя сигнала EXIT:

#!/bin/bash

trap "echo Goodbye..." EXIT

count=1

**while** [ $count -le 5 ]

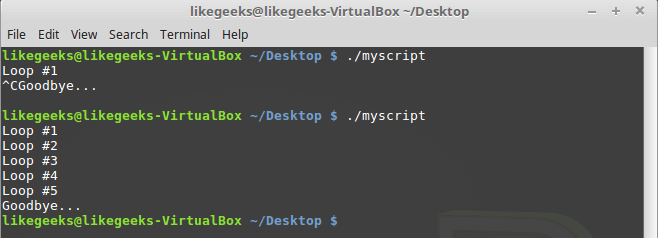
**do**

echo "Loop #$count"

sleep 1

count=$(( $count + 1 ))

**done**



*Перехват сигнала выхода из скрипта*  
  
При выходе из скрипта, будь то нормальное завершение его работы или завершение, вызванное сигналом SIGINT, сработает перехват и оболочка исполнит команду echo.

## Модификация перехваченных сигналов и отмена перехвата

Для модификации перехваченных скриптом сигналов можно выполнить команду trap с новыми параметрами:

#!/bin/bash

trap "echo 'Ctrl-C is trapped.'" SIGINT

count=1

**while** [ $count -le 5 ]

**do**

echo "Loop #$count"

sleep 1

count=$(( $count + 1 ))

**done**

trap "echo ' I modified the trap!'" SIGINT

count=1

**while** [ $count -le 5 ]

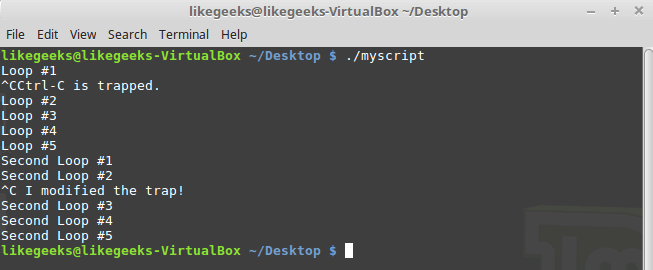
**do**

echo "Second Loop #$count"

sleep 1

count=$(( $count + 1 ))

**done**



*Модификация перехвата сигналов*  
  
После модификации сигналы будут обрабатываться по-новому.  
  
Перехват сигналов можно и отменить, для этого достаточно выполнить команду trap, передав ей двойное тире и имя сигнала:

#!/bin/bash

trap "echo 'Ctrl-C is trapped.'" SIGINT

count=1

**while** [ $count -le 5 ]

**do**

echo "Loop #$count"

sleep 1

count=$(( $count + 1 ))

**done**

trap -- SIGINT

echo "I just removed the trap"

count=1

**while** [ $count -le 5 ]

**do**

echo "Second Loop #$count"

sleep 1

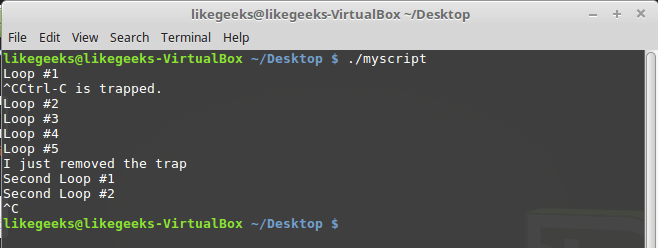
count=$(( $count + 1 ))

**done**

Если скрипт получит сигнал до отмены перехвата, он обработает его так, как задано в действующей команде trap. Запустим скрипт:

$ ./myscript

И нажмём CTRL + C на клавиатуре.



*Сигнал, перехваченный до отмены перехвата*  
  
Первое нажатие CTRL + C пришлось на момент исполнения скрипта, когда перехват сигнала был в силе, поэтому скрипт исполнил назначенную сигналу команду echo. После того, как исполнение дошло до команды отмены перехвата, команда CTRL + C сработала обычным образом, завершив работу скрипта.

## Выполнение сценариев командной строки в фоновом режиме

Иногда bash-скриптам требуется немало времени для выполнения некоей задачи. При этом вам может понадобиться возможность нормально работать в командной строке, не дожидаясь завершения скрипта. Реализовать это не так уж и сложно.  
  
Если вы видели список процессов, выводимый командой ps, вы могли заметить процессы, которые выполняются в фоне и не привязаны к терминалу.  
Напишем такой скрипт:

#!/bin/bash

count=1

**while** [ $count -le 10 ]

**do**

sleep 1

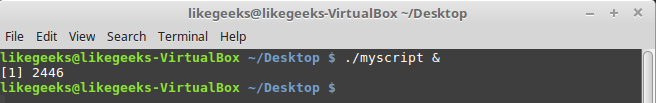
count=$(( $count + 1 ))

**done**

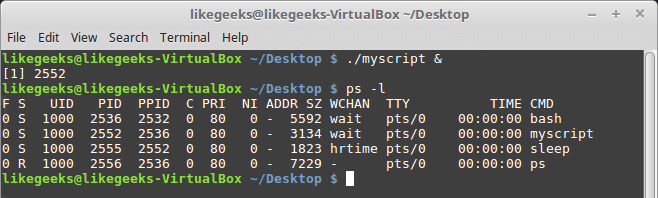
Запустим его, указав после имени символ амперсанда (&):

$ ./myscipt &

Это приведёт к тому, что он будет запущен как фоновый процесс.



*Запуск скрипта в фоновом режиме*  
  
Скрипт будет запущен в фоновом процессе, в терминал выведется его идентификатор, а когда его выполнение завершится, вы увидите сообщение об этом.  
  
Обратите внимание на то, что хотя скрипт выполняется в фоне, он продолжает использовать терминал для вывода сообщений в STDOUT и STDERR, то есть, выводимый им текст или сообщения об ошибках можно будет увидеть в терминале.



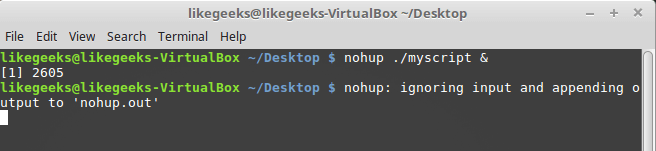
*Список процессов*  
  
При таком подходе, если выйти из терминала, скрипт, выполняющийся в фоне, так же завершит работу.  
  
Что если нужно, чтобы скрипт продолжал работать и после закрытия терминала?

## Выполнение скриптов, не завершающих работу при закрытии терминала

Скрипты можно выполнять в фоновых процессах даже после выхода из терминальной сессии. Для этого можно воспользоваться командой nohup. Эта команда позволяет запустить программу, блокируя сигналы SIGHUP, отправляемые процессу. В результате процесс будет исполняться даже при выходе из терминала, в котором он был запущен.  
  
Применим эту методику при запуске нашего скрипта:

nohup ./myscript &

Вот что будет выведено в терминал.



*Команда nohup*  
  
Команда nohup отвязывает процесс от терминала. Это означает, что процесс потеряет ссылки на STDOUT и STDERR. Для того, чтобы не потерять данные, выводимые скриптом, nohup автоматически перенаправляет сообщения, поступающие в STDOUT и в STDERR, в файл nohup.out.  
  
Обратите внимание на то, что при запуске нескольких скриптов из одной и той же директории то, что они выводят, попадёт в один файл nohup.out.

## Просмотр заданий

Команда jobs позволяет просматривать текущие задания, которые выполняются в оболочке. Напишем такой скрипт:

#!/bin/bash

count=1

**while** [ $count -le 10 ]

**do**

echo "Loop #$count"

sleep 10

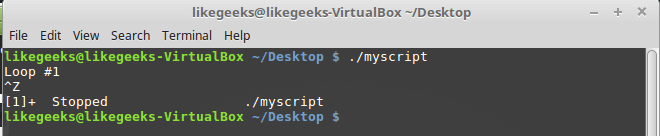
count=$(( $count + 1 ))

**done**

Запустим его:

$ ./myscript

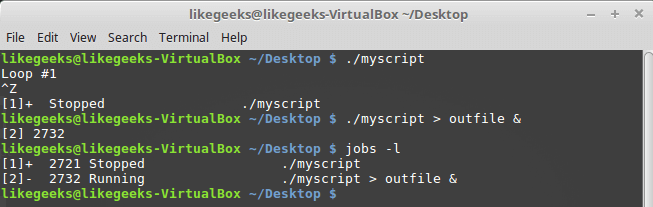
И временно остановим комбинацией клавиш CTRL + Z.



*Запуск и приостановка скрипта*  
  
Запустим тот же скрипт в фоновом режиме, при этом перенаправим вывод скрипта в файл так, чтобы он ничего не выводил на экране:

$ ./myscript > outfile &

Выполнив теперь команду jobs, мы увидим сведения как о приостановленном скрипте, так и о том, который работает в фоне.



*Получение сведений о скриптах*  
  
Ключ -l при вызове команды jobs указывает на то, что нам нужны сведения об ID процессов.

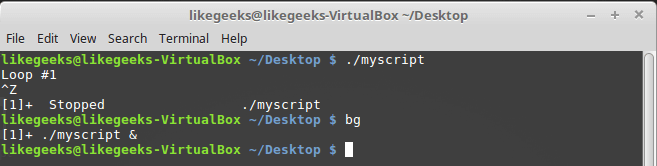
## Перезапуск приостановленных заданий

Для того, чтобы перезапустить скрипт в фоновом режиме, можно воспользоваться командой bg.  
  
Запустим скрипт:

$ ./myscript

Нажмём CTRL + Z, что временно остановит его выполнение. Выполним следующую команду:

$ bg



*Команда bg*  
  
Теперь скрипт выполняется в фоновом режиме.  
  
Если у вас имеется несколько приостановленных заданий, для перезапуска конкретного задания команде bg можно передать его номер.  
  
Для перезапуска задания в обычном режиме воспользуйтесь командой fg:

$ fg 1

## Планирование запуска скриптов

Linux предоставляет пару способов запуска bash-скриптов в заданное время. Это команда at и планировщик заданий cron.  
  
Вызов команды at выглядит так:

at [-f filename] time

Эта команда распознаёт множество форматов указания времени.

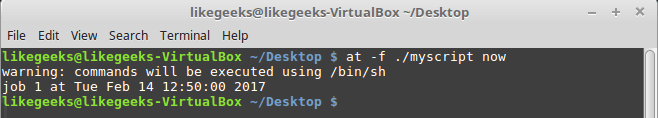
* Стандартный, с указанием часов и минут, например — 10:15.
* С использованием индикаторов AM/PM, до или после полудня, например — 10:15PM.
* С использованием специальных имён, таких, как now, noon, midnight.

В дополнение к возможности указания времени запуска задания, команде at можно передать и дату, используя один из поддерживаемых ей форматов.

* Стандартный формат указания даты, при котором дата записывается по шаблонам MMDDYY, MM/DD/YY, или DD.MM.YY.
* Текстовое представление даты, например, Jul 4 или Dec 25, при этом год можно указать, а можно обойтись и без него.
* Запись вида now + 25 minutes.
* Запись вида 10:15PM tomorrow.
* Запись вида 10:15 + 7 days.

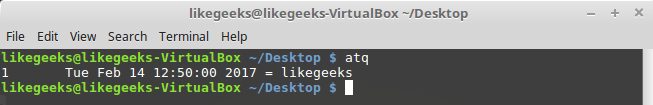
Не будем углубляться в эту тему, рассмотрим простой вариант использования команды:

$ at -f ./myscript now



*Планирование заданий с использованием команды at*  
  
Ключ -M при вызове at используется для отправки того, что выведет скрипт, по электронной почте, если система соответствующим образом настроена. Если отправка электронного письма невозможна, этот ключ просто подавит вывод.  
  
Для того чтобы посмотреть список заданий, ожидающих выполнения, можно воспользоваться командой atq:

$ atq

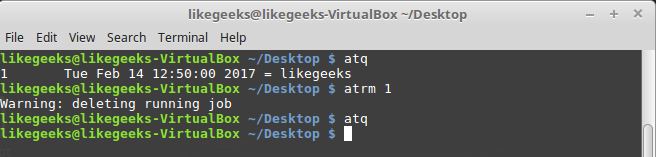


*Список заданий, ожидающих выполнения*

## Удаление заданий, ожидающих выполнения

Удалить задание, ожидающее выполнения, позволяет команда atrm. При её вызове указывают номер задания:

$ atrm 18



*Удаление задания*

## Запуск скриптов по расписанию

Планирование однократного запуска скриптов с использованием команды at способно облегчить жизнь во многих ситуациях. Но как быть, если нужно, чтобы скрипт выполнялся в одно и то же время ежедневно, или раз в неделю, или раз в месяц?  
  
В Linux имеется утилита crontab, позволяющая планировать запуск скриптов, которые нужно выполнять регулярно.  
  
Crontab выполняется в фоне и, основываясь на данных в так называемых cron-таблицах, запускает задания по расписанию.  
  
Для того, чтобы просмотреть существующую таблицу заданий cron, воспользуйтесь такой командой:

$ crontab –l

При планировании запуска скрипта по расписанию crontab принимает данные о том, когда нужно выполнить задание, в таком формате:

минута, час, день месяца, месяц, день недели.

Например, если надо, чтобы некий скрипт с именем command выполнялся ежедневно в 10:30, этому будет соответствовать такая запись в таблице заданий:

30 10 \* \* \* command

Здесь универсальный символ «\*», использованный для полей, задающих день месяца, месяц и день недели, указывает на то, что cron должен выполнять команду каждый день каждого месяца в 10:30.  
  
Если, например, надо, чтобы скрипт запускался в 4:30PM каждый понедельник, понадобится создать в таблице заданий такую запись:

30 16 \* \* 1 command

Нумерация дней недели начинается с 0, 0 означает воскресенье, 6 — субботу. Вот ещё один пример. Здесь команда будет выполняться в 12 часов дня в первый день каждого месяца.

00 12 1 \* \* command

Нумерация месяцев начинается с 1.  
Для того чтобы добавить запись в таблицу, нужно вызвать crontab с ключом -e:

**crontab** –e

Затем можно вводить команды формирования расписания:

30 10 \* \* \* /home/likegeeks/Desktop/myscript

Благодаря этой команде скрипт будет вызываться ежедневно в 10:30. Если вы столкнётесь с ошибкой «Resource temporarily unavailable», выполните нижеприведённую команду с правами root-пользователя:

$ rm -f /var/run/crond.pid

Организовать периодический запуск скриптов с использованием cron можно ещё проще, воспользовавшись несколькими специальными директориями:

/etc/cron.hourly

/etc/cron.daily

/etc/cron.weekly

/etc/cron.monthly

Если поместить файл скрипта в одну из них, это приведёт, соответственно, к его ежечасному, ежедневному, еженедельному или ежемесячному запуску.

## Запуск скриптов при входе в систему и при запуске оболочки

Автоматизировать запуск скриптов можно, опираясь на различные события, такие, как вход пользователя в систему или запуск оболочки. [Тут](https://likegeeks.com/linux-environment-variables/) можно почитать о файлах, которые обрабатываются в подобных ситуациях. Например, это следующие файлы:

$HOME/.bash\_profile

$HOME/.bash\_login

$HOME/.profile

Для того, чтобы запускать скрипт при входе в систему, поместите его вызов в файл .bash\_profile.  
  
А как насчёт запуска скриптов при открытии терминала? Организовать это поможет файл .bashrc.

## Объявление функций

Функцию можно объявить так:

**functionName** {

}

Или так:

**functionName**() {

}

Функцию можно вызвать без аргументов и с аргументами.

## Использование функций

Напишем скрипт, содержащий объявление функции и использующий её:

#!/bin/bash

**function** myfunc {

echo "This is an example of using a function"

}

count=1

**while** [ $count -le 3 ]

**do**

myfunc

count=$(( $count + 1 ))

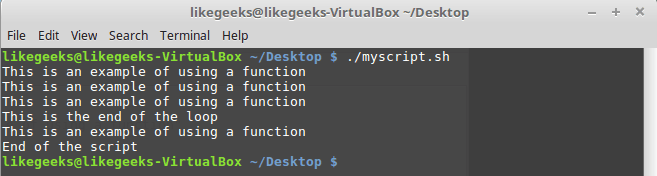
**done**

echo "This is the end of the loop"

myfunc

echo "End of the script"

Здесь создана функция с именем myfunc. Для вызова функции достаточно указать её имя.



*Результаты вызова функции*  
  
Функцию можно вызывать столько раз, сколько нужно. Обратите внимание на то, что попытавшись использовать функцию до её объявления, вы столкнётесь с ошибкой. Напишем демонстрирующий это скрипт:

#!/bin/bash

count=1

**while** [ $count -le 3 ]

**do**

myfunc

count=$(( $count + 1 ))

**done**

echo "This is the end of the loop"

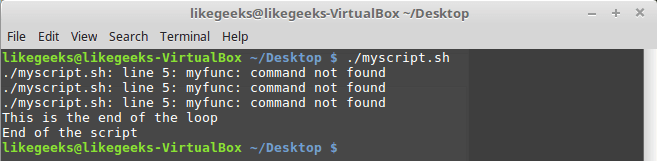
**function** myfunc {

echo "This is an example of using a function"

}

echo "End of the script"

Как и ожидается, ничего хорошего после его запуска не произошло.



*Попытка воспользоваться функцией до её объявления*  
  
Придумывая имена для функций, учитывайте то, что они должны быть уникальными, иначе проблем не избежать. Если вы переопределите ранее объявленную функцию, новая функция будет вызываться вместо старой без каких-либо уведомлений или сообщений об ошибках. Продемонстрируем это на примере:

#!/bin/bash

**function** myfunc {

echo "The first function definition"

}

myfunc

**function** myfunc {

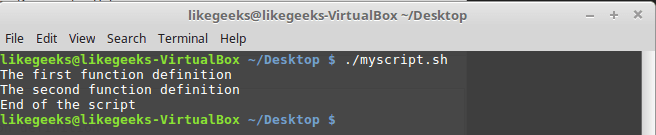
echo "The second function definition"

}

myfunc

echo "End of the script"

Как видно, новая функция преспокойно затёрла старую.



*Переопределение функции*

## Использование команды return

Команда return позволяет задавать возвращаемый функцией целочисленный код завершения. Есть два способа работы с тем, что является результатом вызова функции. Вот первый:

#!/bin/bash

**function** myfunc {

read -p "Enter a value: " value

echo "adding value"

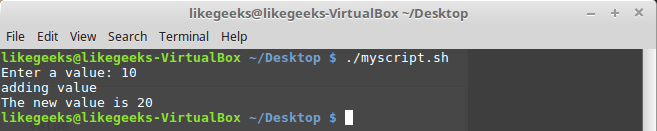
return $(( $value + 10 ))

}

myfunc

echo "The new value is $?"

Команда echo вывела сумму введённого числа и числа 10.



*Вывод значения, возвращаемого функцией*  
  
Функция myfunc добавляет 10 к числу, которое содержится в переменной $value, значение которой задаёт пользователь во время работы сценария. Затем она возвращает результат, используя команду return. То, что возвратила функция, выводится командой echo с использованием переменной $?.  
Если вы выполните любую другую команду до извлечения из переменной $? значения, возвращённого функцией, это значение будет утеряно. Дело в том, что данная переменная хранит код возврата последней выполненной команды.  
  
Учтите, что максимальное число, которое может вернуть команда return — 255. Если функция должна возвращать большее число или строку, понадобится другой подход.

## Запись вывода функции в переменную

Ещё один способ возврата результатов работы функции заключается в записи данных, выводимых функцией, в переменную. Такой подход позволяет обойти ограничения команды return и возвращать из функции любые данные. Рассмотрим пример:

#!/bin/bash

**function** myfunc {

read -p "Enter a value: " value

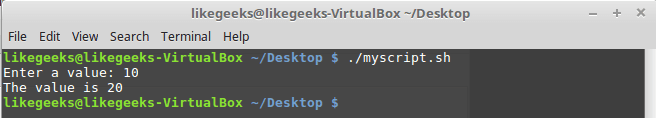
echo $(( $value + 10 ))

}

result=$( myfunc)

echo "The value is $result"

Вот что получится после вызова данного скрипта.



*Запись результатов работы функции в переменную*

## Аргументы функций

Функции bash можно воспринимать как небольшие фрагменты кода, которые позволяют экономить время и место, избавляя нас от необходимости постоянно вводить с клавиатуры или копировать одни и те же наборы команд. Однако, возможности функций гораздо шире. В частности, речь идёт о передаче им аргументов.  
  
Функции могут использовать стандартные позиционные параметры, в которые записывается то, что передаётся им при вызове. Например, имя функции хранится в параметре $0, первый переданный ей аргумент — в $1, второй — в $2, и так далее. Количество переданных функции аргументов можно узнать, обратившись к переменной $#. Если вы знакомы с [третьей частью](https://habrahabr.ru/company/ruvds/blog/326328/) этого цикла материалов, вы не можете не заметить, что всё это очень похоже на то, как скрипты обрабатывают переданные им параметры командной строки.  
  
Аргументы передают функции, записывая их после её имени:

**myfunc** $val1 10 20

Вот пример, в котором функция вызывается с аргументами и занимается их обработкой:

#!/bin/bash

**function** addnum {

**if** [ $# -eq 0 ] || [ $# -gt 2 ]

**then**

echo -1

**elif** [ $# -eq 1 ]

**then**

echo $(( $1 + $1 ))

**else**

echo $(( $1 + $2 ))

**fi**

}

echo -n "Adding 10 and 15: "

value=$(addnum 10 15)

echo $value

echo -n "Adding one number: "

value=$(addnum 10)

echo $value

echo -n "Adding no numbers: "

value=$(addnum)

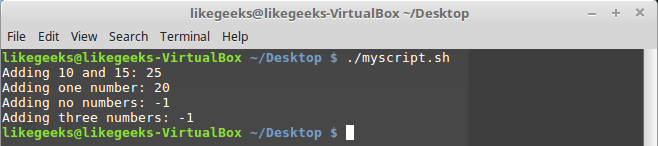
echo $value

echo -n "Adding three numbers: "

value=$(addnum 10 15 20)

echo $value

Запустим скрипт.



*Вызов функции с аргументами*  
  
Функция addnum проверяет число переданных ей при вызове из скрипта аргументов. Если их нет, или их больше двух, функция возвращает значение -1. Если параметр всего один, она прибавляет его к нему самому и возвращает результат. Если параметров два, функция складывает их.  
  
Обратите внимание на то, что функция не может напрямую работать с параметрами, которые переданы скрипту при его запуске из командной строки. Например, напишем такой сценарий:

#!/bin/bash

**function** myfunc {

echo $(( $1 + $2 ))

}

**if** [ $# -eq 2 ]

**then**

value=$( myfunc)

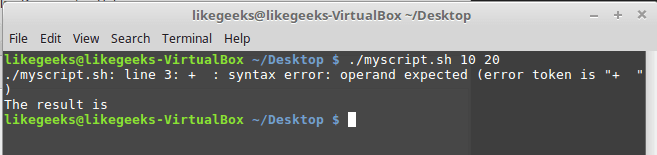
echo "The result is $value"

**else**

echo "Usage: myfunc  a b"

**fi**

При его запуске, а точнее, при вызове объявленной в нём функции, будет выведено сообщение об ошибке.



*Функция не может напрямую использовать параметры, переданные сценарию*  
  
Вместо этого, если в функции планируется использовать параметры, переданные скрипту при вызове из командной строки, надо передать их ей при вызове:

#!/bin/bash

**function** myfunc {

echo $(( $1 + $2 ))

}

**if** [ $# -eq 2 ]

**then**

value=$(myfunc $1 $2)

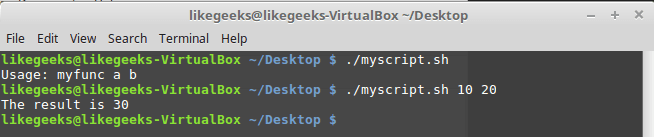
echo "The result is $value"

**else**

echo "Usage: myfunc a b"

**fi**

Теперь всё работает правильно.



*Передача функции параметров, с которыми запущен скрипт*

## Работа с переменными в функциях

Переменные, которыми мы пользуемся в сценариях, характеризуются областью видимости. Это — те места кода, из которых можно работать с этими переменными. Переменные, объявленные внутри функций, ведут себя не так, как те переменные, с которыми мы уже сталкивались. Они могут быть скрыты от других частей скриптов.  
  
Существуют два вида переменных:

* Глобальные переменные.
* Локальные переменные.

### **▍Глобальные переменные**

Глобальные переменные — это переменные, которые видны из любого места bash-скрипта. Если вы объявили глобальную переменную в основном коде скрипта, к такой переменной можно обратиться из функции.  
  
Почти то же самое справедливо и для глобальных переменных, объявленных в функциях. Обращаться к ним можно и в основном коде скрипта после вызова функций.  
  
По умолчанию все объявленные в скриптах переменные глобальны. Так, к переменным, объявленным за пределами функций, можно без проблем обращаться из функций:

#!/bin/bash

**function** myfunc {

value=$(( $value + 10 ))

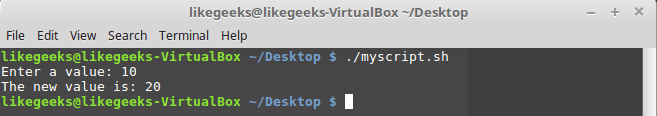
}

read -p "Enter a value: " value

myfunc

echo "The new value is: $value"

Вот что выведет этот сценарий.



*Обращение к глобальной переменной из функции*  
  
Когда переменной присваивается новое значение в функции, это новое значение не теряется когда скрипт обращается к ней после завершения работы функции. Именно это можно видеть в предыдущем примере.  
  
Что если такое поведение нас не устраивает? Ответ прост — надо использовать локальные переменные.

### **▍Локальные переменные**

Переменные, которые объявляют и используют внутри функции, могут быть объявлены локальными. Для того, чтобы это сделать, используется ключевое слово local перед именем переменной:

local temp=$(( $value + 5 ))

Если за пределами функции есть переменная с таким же именем, это на неё не повлияет. Ключевое слово local позволяет отделить переменные, используемые внутри функции, от остальных переменных. Рассмотрим пример:

#!/bin/bash

**function** myfunc {

local temp=$[ $value + 5 ]

echo "The Temp from inside function is $temp"

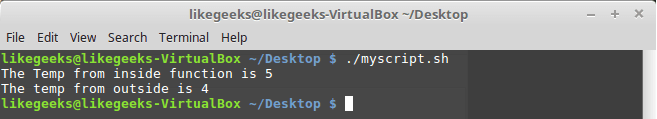
}

temp=4

myfunc

echo "The temp from outside is $temp"

Запустим скрипт.



*Локальная переменная в функции*  
  
Здесь, когда мы работаем с переменной $temp внутри функции, это не влияет на значение, назначенное переменной с таким же именем за её пределами.

## Передача функциям массивов в качестве аргументов

Попробуем передать функции в качестве аргумента массив. Сразу хочется сказать, что работать такая конструкция будет неправильно:

#!/bin/bash

**function** myfunc {

echo "The parameters are: $@"

arr=$1

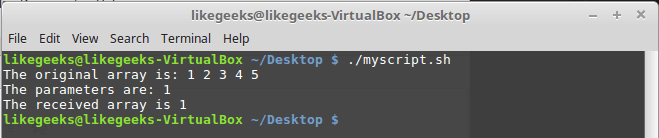
echo "The received array is ${arr[\*]}"

}

myarray=(1 2 3 4 5)

echo "The original array is: ${myarray[\*]}"

myfunc $myarray



*Неправильный подход к передаче функциям массивов*  
  
Как видно из примера, при передаче функции массива, она получит доступ лишь к его первому элементу.  
  
Для того, чтобы эту проблему решить, из массива надо извлечь имеющиеся в нём данные и передать их функции как самостоятельные аргументы. Если надо, внутри функции полученные ей аргументы можно снова собрать в массив:

#!/bin/bash

**function** myfunc {

local newarray

newarray=("$@")

echo "The new array value is: ${newarray[\*]}"

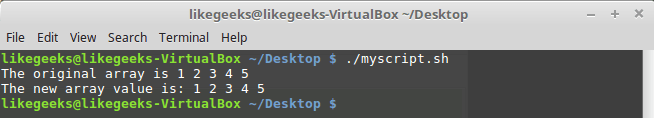
}

myarray=(1 2 3 4 5)

echo "The original array is ${myarray[\*]}"

myfunc ${myarray[\*]}

Запустим сценарий.



*Сборка массива внутри функции*  
  
Как видно из примера, функция собрала массив из переданных ей аргументов.

## Рекурсивные функции

Рекурсия — это когда функция сама себя вызывает. Классический пример рекурсии — функция для вычисления факториала. Факториал числа — это произведение всех натуральных чисел от 1 до этого числа. Например, факториал 5 можно найти так:

5! = 1 \* 2 \* 3 \* 4 \* 5

Если формулу вычисления факториала написать в рекурсивном виде, получится следующее:

x! = x \* (x-1)!

Этой формулой можно воспользоваться для того, чтобы написать рекурсивную функцию:

#!/bin/bash

**function** factorial {

**if** [ $1 -eq 1 ]

**then**

echo 1

**else**

local temp=$(( $1 - 1 ))

local result=$(factorial $temp)

echo $(( $result \* $1 ))

**fi**

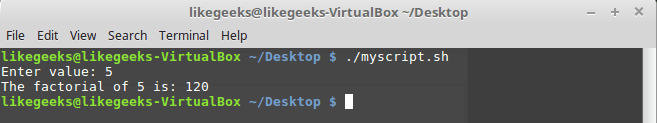
}

read -p "Enter value: " value

result=$(factorial $value)

echo "The factorial of $value is: $result"

Проверим, верно ли работает этот скрипт.



*Вычисление факториала*  
  
Как видите, всё работает как надо.

## Создание и использование библиотек

Итак, теперь вы знаете, как писать функции и как вызывать их в том же скрипте, где они объявлены. Что если надо использовать функцию, тот блок кода, который она собой представляет, в другом скрипте, не используя копирование и вставку?  
  
Оболочка bash позволяет создавать так называемые библиотеки — файлы, содержащие функции, а затем использовать эти библиотеки в любых скриптах, где они нужны.  
  
Ключ к использованию библиотек — в команде source. Эта команда используется для подключения библиотек к скриптам. В результате функции, объявленные в библиотеке, становятся доступными в скрипте, в противном же случае функции из библиотек не будут доступны в области видимости других скриптов.  
  
У команды source есть псевдоним — оператор «точка». Для того, чтобы подключить файл в скрипте, в скрипт надо добавить конструкцию такого вида:

. ./myscript

Предположим, что у нас имеется файл myfuncs, который содержит следующее:

**function** addnum {

echo $(( $1 + $2 ))

}

Это — библиотека. Воспользуемся ей в сценарии:

#!/bin/bash

. ./myfuncs

result=$(addnum 10 20)

echo "The result is: $result"

Вызовем его.



*Использование библиотек*  
  
Только что мы использовали библиотечную функцию внутри скрипта. Всё это замечательно, но что если мы хотим вызвать функцию, объявленную в библиотеке, из командной строки?

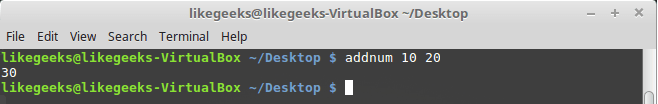
## Вызов bash-функций из командной строки

Если вы освоили [предыдущую часть](https://habrahabr.ru/company/ruvds/blog/326826/) из этой серии, вы, вероятно, уже догадываетесь, что функцию из библиотеки можно подключить в файле .bashrc, используя команду source. Как результат, вызывать функцию можно будет прямо из командной строки.  
  
Отредактируйте .bashrc, добавив в него такую строку (путь к файлу библиотеки в вашей системе, естественно, будет другим):

. /home/likegeeks/Desktop/myfuncs

Теперь функцию можно вызывать прямо из командной строки:

$ addnum 10 20



*Вызов функции из командной строки*  
  
Ещё приятнее то, что такая вот библиотека оказывается доступной всем дочерним процессам оболочки, то есть — ей можно пользоваться в bash-скриптах, не заботясь о подключении к ним этой библиотеки.  
  
Тут стоит отметить, что для того, чтобы вышеприведённый пример заработал, может понадобиться выйти из системы, а потом войти снова. Кроме того, обратите внимание на то, что если имя функции из библиотеки совпадёт с именем какой-нибудь стандартной команды, вместо этой команды будет вызываться функция. Поэтому внимательно относитесь к именам функций.