

# Bash-скрипты, руководство в 11 частях

Источник: <https://likegeeks.com/>

Перевод: <https://habrahabr.ru/company/ruvds/blog/>

Читать онлайн (много полезных комментариев):

[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: expect и автоматизация интерактивных утилит](#)

<b>Bash-скрипты: начало</b>	<b>5</b>
Как устроены bash-скрипты	5
Установка разрешений для файла сценария	6
Вывод сообщений	6
Использование переменных	7
Переменные среды	7
Пользовательские переменные	8
Подстановка команд	8
Математические операции	9
Управляющая конструкция if-then	9
Управляющая конструкция if-then-else	11
Сравнение чисел	12
Сравнение строк	13
Проверки файлов	16
Итоги	17
<b>Bash-скрипты, часть 2: циклы</b>	<b>18</b>
Циклы for	18
Перебор простых значений	18
Перебор сложных значений	19
Инициализация цикла списком, полученным из результатов работы команды	19
Разделители полей	20
Обход файлов, содержащихся в директории	21
Циклы for в стиле C	22
Цикл while	23
Вложенные циклы	24
Обработка содержимого файла	24
Управление циклами	25
Команда break	26
Команда continue	27
Обработка вывода, выполняемого в цикле	27
Пример: поиск исполняемых файлов	28
Итоги	29
<b>Bash-скрипты, часть 3: параметры и ключи командной строки</b>	<b>30</b>
Чтение параметров командной строки	30
Проверка параметров	31
Подсчёт параметров	32
Захват всех параметров командной строки	33
Команда shift	34
Ключи командной строки	35
Как различать ключи и параметры	36
Обработка ключей со значениями	37
Использование стандартных ключей	38
Получение данных от пользователя	39
Ввод паролей	40
Чтение данных из файла	41

Итоги	42
<b>Bash-скрипты, часть 4: ввод и вывод</b>	<b>43</b>
Стандартные дескрипторы файлов	43
STDIN	43
STDOUT	43
STDERR	44
Перенаправление потока ошибок	44
Перенаправление потоков ошибок и вывода	45
Перенаправление вывода в скриптах	45
Временное перенаправление вывода	46
Постоянное перенаправление вывода	46
Перенаправление ввода в скриптах	47
Создание собственного перенаправления вывода	48
Создание дескрипторов файлов для ввода данных	49
Закрытие дескрипторов файлов	50
Получение сведений об открытых дескрипторах	50
Подавление вывода	52
Итоги	52
<b>Bash-скрипты, часть 5: сигналы, фоновые задачи, управление сценариями</b>	<b>53</b>
Сигналы Linux	53
Отправка сигналов скриптам	54
Завершение работы процесса	54
Временная остановка процесса	54
Перехват сигналов	55
Перехват сигнала выхода из скрипта	56
Модификация перехваченных сигналов и отмена перехвата	57
Выполнение сценариев командной строки в фоновом режиме	59
Выполнение скриптов, не завершающих работу при закрытии терминала	60
Просмотр заданий	60
Перезапуск приостановленных заданий	61
Планирование запуска скриптов	62
Удаление заданий, ожидающих выполнения	63
Запуск скриптов по расписанию	63
Запуск скриптов при входе в систему и при запуске оболочки	65
Итоги	65
<b>Bash-скрипты, часть 6: функции и разработка библиотек</b>	<b>66</b>
Объявление функций	66
Использование функций	66
Использование команды return	68
Запись вывода функции в переменную	69
Аргументы функций	69
Работа с переменными в функциях	72
Глобальные переменные	72
Локальные переменные	73
Передача функциям массивов в качестве аргументов	74
Рекурсивные функции	75

Создание и использование библиотек	76
Вызов <code>bash</code> -функций из командной строки	77
Итоги	77
<b>Bash-скрипты, часть 7: <code>sed</code> и обработка текстов</b>	<b>78</b>
Основы работы с <code>sed</code>	78
Выполнение наборов команд при вызове <code>sed</code>	79
Чтение команд из файла	80
Флаги команды замены	80
Символы-разделители	82
Выбор фрагментов текста для обработки	83
Удаление строк	84
Вставка текста в поток	85
Замена строк	87
Замена символов	87
Вывод номеров строк	88
Чтение данных для вставки из файла	88
Пример	89
Итоги	90
<b>Bash-скрипты, часть 8: язык обработки данных <code>awk</code></b>	<b>91</b>
Особенности вызова <code>awk</code>	91
Чтение <code>awk</code> -скриптов из командной строки	91
Позиционные переменные, хранящие данные полей	92
Использование нескольких команд	93
Чтение скрипта <code>awk</code> из файла	94
Выполнение команд до начала обработки данных	94
Выполнение команд после окончания обработки данных	95
Встроенные переменные: настройка процесса обработки данных	96
Встроенные переменные: сведения о данных и об окружении	98
Пользовательские переменные	101
Условный оператор	101
Цикл <code>while</code>	103
Цикл <code>for</code>	105
Форматированный вывод данных	106
Встроенные математические функции	107
Строковые функции	107
Пользовательские функции	108
Итоги	108
<b>Bash-скрипты, часть 9: регулярные выражения</b>	<b>110</b>
Что такое регулярные выражения	110
Типы регулярных выражений	110
Регулярные выражения POSIX BRE	111
Специальные символы	112
Якорные символы	113
Символ «точка»	114
Классы символов	115
Отрицание классов символов	115

Диапазоны символов	116
Специальные классы символов	117
Символ «звёздочка»	117
Регулярные выражения POSIX ERE	119
Вопросительный знак	119
Символ «плюс»	120
Фигурные скобки	121
Символ логического «или»	122
Группировка фрагментов регулярных выражений	122
Практические примеры	123
Подсчёт количества файлов	123
Проверка адресов электронной почты	124
Итоги	125
<b>Bash-скрипты, часть 10: практические примеры</b>	<b>126</b>
Отправка сообщений в терминал пользователя	126
Команды who и mesg	126
Команда write	127
Создание скрипта для отправки сообщений	128
Проверка возможности записи в терминал пользователя	129
Проверка правильности вызова скрипта	130
Получение сведений о терминале пользователя	130
Отправка длинных сообщений	131
Скрипт для мониторинга дискового пространства	133
Итоги	136
<b>Bash-скрипты, часть 11: expect и автоматизация интерактивных утилит</b>	<b>137</b>
Основы expect	137
Автоматизация bash-скрипта	137
Autoexpect — автоматизированное создание expect-скриптов	139
Работа с переменными и параметрами командной строки	140
Ответы на разные вопросы, которые могут появиться в одном и том же месте	142
Условный оператор	143
Цикл while	144
Цикл for	144
Объявление и использование функций	145
Команда interact	146
Итоги	147

# Bash-скрипты: начало

Сегодня поговорим о bash-скриптах. Это — [сценарии командной строки](#), написанные для оболочки bash. Существуют и другие оболочки, например — zsh, tcsh, ksh, но мы сосредоточимся на bash. Этот материал предназначен для всех желающих, единственное условие — умение работать в [командной строке](#) Linux.

Сценарии командной строки — это наборы тех же самых команд, которые можно вводить с клавиатуры, собранные в файлы и объединённые некоей общей целью. При этом результаты работы команд могут представлять либо самостоятельную ценность, либо служить входными данными для других команд. Сценарии — это мощный способ автоматизации часто выполняемых действий.

Итак, если говорить о командной строке, она позволяет выполнить несколько команд за один раз, введя их через точку с запятой:

```
pwd ; whoami
```

На самом деле, если вы опробовали это в своём терминале, ваш первый bash-скрипт, в котором задействованы две команды, уже написан. Работает он так. Сначала команда `pwd` выводит на экран сведения о текущей рабочей директории, потом команда `whoami` показывает данные о пользователе, под которым вы вошли в систему.

Используя подобный подход, вы можете совмещать сколько угодно команд в одной строке, ограничение — лишь в максимальном количестве аргументов, которое можно передать программе. Определить это ограничение можно с помощью такой команды:

```
getconf ARG_MAX
```

Командная строка — отличный инструмент, но команды в неё приходится вводить каждый раз, когда в них возникает необходимость. Что если записать набор команд в файл и просто вызывать этот файл для их выполнения? Собственно говоря, тот файл, о котором мы говорим, и называется сценарием командной строки.

## Как устроены bash-скрипты

Создайте пустой файл с использованием команды `touch`. В его первой строке нужно указать, какую именно оболочку мы собираемся использовать. Нас интересует bash, поэтому первая строка файла будет такой:

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

В других строках этого файла символ решётки используется для обозначения комментариев, которые оболочка не обрабатывает. Однако, первая строка — это особый случай, здесь решётка, за которой следует восклицательный знак (эту последовательность называют [шебанг](#)) и путь к bash, указывают системе на то, что сценарий создан именно для bash.

Команды оболочки отделяются знаком перевода строки, комментарии выделяют знаком решётки. Вот как это выглядит:

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

```
# This is a comment
```

```
pwd
```

```
whoami
```

Тут, так же, как и в командной строке, можно записывать команды в одной строке, разделяя точкой с запятой. Однако, если писать команды на разных строках, файл легче читать. В любом случае оболочка их обработает.

## ***Установка разрешений для файла сценария***

Сохраните файл, дав ему имя myscript, и работа по созданию bash-скрипта почти закончена. Сейчас осталось лишь сделать этот файл исполняемым, иначе, попытавшись его запустить, вы столкнётесь с ошибкой Permission denied.

*Попытка запуска файла сценария с неправильно настроенными разрешениями*

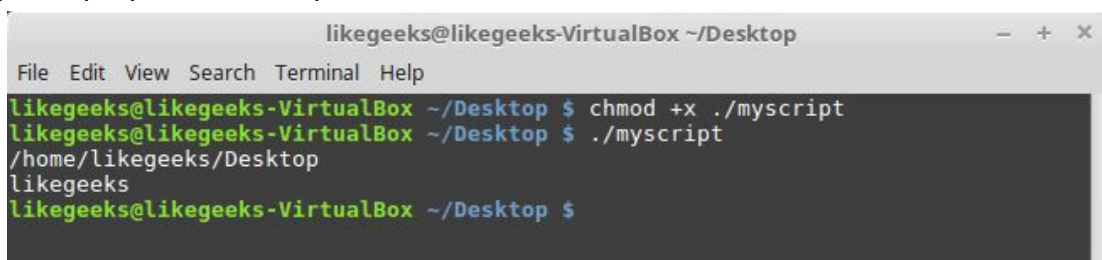
Сделаем файл исполняемым:

```
chmod +x ./myscript
```

Теперь попытаемся его выполнить:

```
./myscript
```

После настройки разрешений всё работает как надо.

A screenshot of a terminal window titled 'likegeeks@likegeeks-VirtualBox ~/Desktop'. The terminal shows the following commands and output: 'likegeeks@likegeeks-VirtualBox ~/Desktop \$ chmod +x ./myscript', 'likegeeks@likegeeks-VirtualBox ~/Desktop \$ ./myscript', followed by the output '/home/likegeeks/Desktop' and 'likegeeks'. The prompt returns to 'likegeeks@likegeeks-VirtualBox ~/Desktop \$'.

*Успешный запуск bash-скрипта*

## ***Вывод сообщений***

Для вывода текста в консоль Linux применяется команда echo. Воспользуемся знанием этого факта и отредактируем наш скрипт, добавив пояснения к данным, которые выводят уже имеющиеся в нём команды:

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

```
# our comment is here
```

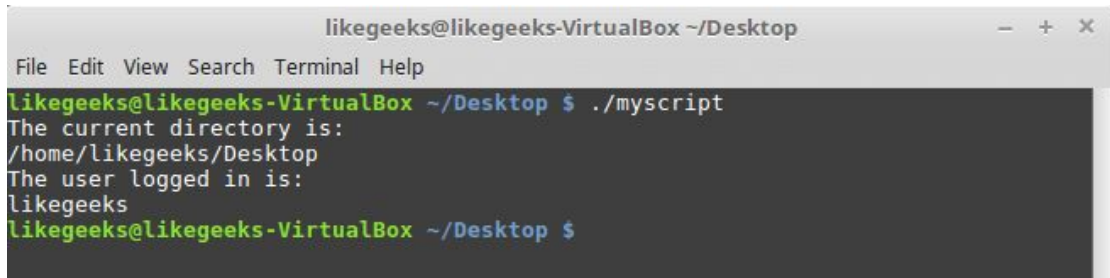
```
echo "The current directory is:"
```

```
pwd
```

```
echo "The user logged in is:"
```

whoami

Вот что получится после запуска обновлённого скрипта.

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: 'The current directory is: /home/likegeeks/Desktop', 'The user logged in is: likegeeks', and the prompt returns to 'likegeeks@likegeeks-VirtualBox ~/Desktop \$'.

*Вывод сообщений из скрипта*

Теперь мы можем выводить поясняющие надписи, используя команду `echo`. Если вы не знаете, как отредактировать файл, пользуясь средствами Linux, или раньше не встречались с командой `echo`, взгляните на [этот](#) материал.

## **Использование переменных**

Переменные позволяют хранить в файле сценария информацию, например — результаты работы команд для использования их другими командами.

Нет ничего плохого в исполнении отдельных команд без хранения результатов их работы, но возможности такого подхода весьма ограничены.

Существуют два типа переменных, которые можно использовать в `bash`-скриптах:

- Переменные среды
- Пользовательские переменные

### **Переменные среды**

Иногда в командах оболочки нужно работать с некоторыми системными данными. Вот, например, как вывести домашнюю директорию текущего пользователя:

```
#!/bin/bash

# display user home

echo "Home for the current user is: $HOME"
```

Обратите внимание на то, что мы можем использовать системную переменную `$HOME` в двойных кавычках, это не мешает системе её распознать. Вот что получится, если выполнить вышеприведённый сценарий.

*Использование переменной среды в сценарии*

А что если надо вывести на экран значок доллара? Попробуем так:



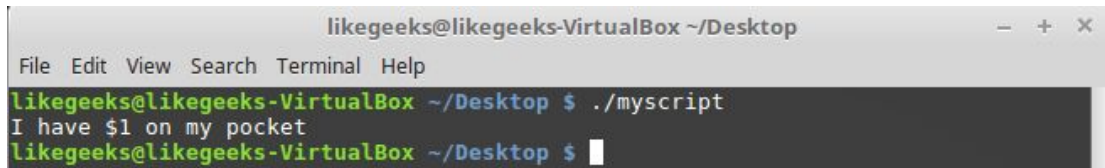
```
echo "I have $1 in my pocket"
```

Система обнаружит знак доллара в строке, ограниченной кавычками, и решит, что мы сослались на переменную. Скрипт попытается вывести на экран значение неопределённой переменной \$1. Это не то, что нам нужно. Что делать?

В подобной ситуации поможет использование управляющего символа, обратной косой черты, перед знаком доллара:

```
echo "I have \$1 in my pocket"
```

Теперь сценарий выведет именно то, что ожидается.

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 a prompt 'likegeeks@likegeeks-VirtualBox ~/Desktop \$' followed by the command './myscript'. The output of the script is 'I have \$1 on my pocket'. The prompt is now 'likegeeks@likegeeks-VirtualBox ~/Desktop \$' with a cursor.

*Использование управляющей последовательности для вывода знака доллара*

## ***Пользовательские переменные***

В дополнение к переменным среды, bash-скрипты позволяют задавать и использовать в сценарии собственные переменные. Подобные переменные хранят значение до тех пор, пока не завершится выполнение сценария.

Как и в случае с системными переменными, к пользовательским переменным можно обращаться, используя знак доллара:

```
#!/bin/bash

# testing variables

grade=5

person="Adam"

echo "$person is a good boy, he is in grade $grade"
```

Вот что получится после запуска такого сценария.

*Пользовательские переменные в сценарии*

## ***Подстановка команд***

Одна из самых полезных возможностей bash-скриптов — это возможность извлекать информацию из вывода команд и назначать её переменным, что позволяет использовать эту информацию где угодно в файле сценария.

Сделать это можно двумя способами.

- С помощью значка обратного апострофа «`»
- С помощью конструкции `$()`

Используя первый подход, проследите за тем, чтобы вместо обратного апострофа не ввести одиночную кавычку. Команду нужно заключить в два таких значка:

```
mydir=`pwd`
```

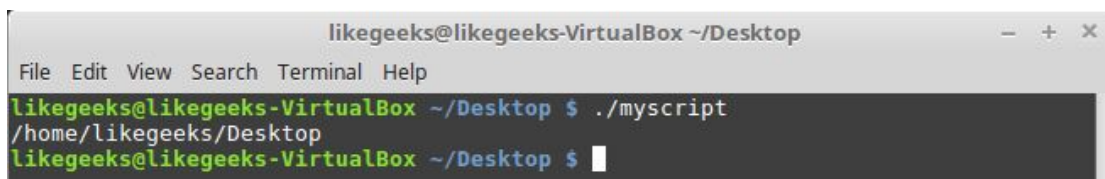
При втором подходе то же самое записывают так:

```
mydir=$(pwd)
```

А скрипт, в итоге, может выглядеть так:

```
#!/bin/bash
mydir=$(pwd)
echo $mydir
```

В ходе его работы вывод команды `pwd` будет сохранён в переменной `mydir`, содержимое которой, с помощью команды `echo`, попадёт в консоль.



The screenshot shows a terminal window titled "likegeeks@likegeeks-VirtualBox ~/Desktop". The terminal has a menu bar with "File", "Edit", "View", "Search", "Terminal", and "Help". The prompt is "likegeeks@likegeeks-VirtualBox ~/Desktop \$". The user enters the command `./myscript`. The output is `/home/likegeeks/Desktop`. The prompt then changes to "likegeeks@likegeeks-VirtualBox ~/Desktop \$".

*Скрипт, сохраняющий результаты работы команды в переменной*

## **Математические операции**

Для выполнения математических операций в файле скрипта можно использовать конструкцию вида `$((a+b))`:

```
#!/bin/bash
var1=$(( 5 + 5 ))
echo $var1
var2=$(( $var1 * 2 ))
echo $var2
```

*Математические операции в сценарии*

## **Управляющая конструкция if-then**

В некоторых сценариях требуется управлять потоком исполнения команд. Например, если некое значение больше пяти, нужно выполнить одно действие, в противном случае — другое. Подобное применимо в очень многих ситуациях, и здесь нам поможет управляющая конструкция if-then. В наиболее простом виде она выглядит так:

```
if команда

then

команды

fi
```

А вот рабочий пример:

```
#!/bin/bash

if pwd

then

echo "It works"

fi
```

В данном случае, если выполнение команды pwd завершится успешно, в консоль будет выведен текст «it works».

Воспользуемся имеющимися у нас знаниями и напишем более сложный сценарий. Скажем, надо найти некоего пользователя в /etc/passwd, и если найти его удалось, сообщить о том, что он существует.

```
#!/bin/bash

user=likegeeks

if grep $user /etc/passwd

then

echo "The user $user Exists"

fi
```

Вот что получается после запуска этого скрипта.

#### *Поиск пользователя*

Здесь мы воспользовались командой grep для поиска пользователя в файле /etc/passwd. Если команда grep вам незнакома, её описание можно найти [здесь](#).

В этом примере, если пользователь найден, скрипт выведет соответствующее сообщение. А если найти пользователя не удалось? В данном случае скрипт просто завершит выполнение, ничего нам не сообщив. Хотелось бы, чтобы он сказал нам и об этом, поэтому усовершенствуем код.

## Управляющая конструкция *if-then-else*

Для того, чтобы программа смогла сообщить и о результатах успешного поиска, и о неудаче, воспользуемся конструкцией *if-then-else*. Вот как она устроена:

```
if команда

then

команды

else

команды

fi
```

Если первая команда возвратит ноль, что означает её успешное выполнение, условие окажется истинным и выполнение не пойдёт по ветке *else*. В противном случае, если будет возвращено что-то, отличное от нуля, что будет означать неудачу, или ложный результат, будут выполнены команды, расположенные после *else*.

Напишем такой скрипт:

```
#!/bin/bash

user=anotherUser

if grep $user /etc/passwd

then

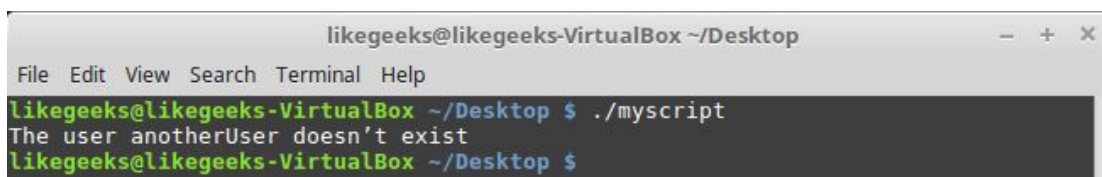
echo "The user $user Exists"

else

echo "The user $user doesn't exist"

fi
```

Его исполнение пошло по ветке *else*.



```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript
The user anotherUser doesn't exist
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

*Запуск скрипта с конструкцией if-then-else*

Ну что же, продолжаем двигаться дальше и зададимся вопросом о более сложных условиях. Что если надо проверить не одно условие, а несколько? Например, если нужный пользователь найден, надо вывести одно сообщение, если выполняется ещё какое-то условие — ещё одно сообщение, и так далее. В подобной ситуации нам помогут вложенные условия. Выглядит это так:

```
if команда1
then
команды
elif команда2
then
команды
fi
```

Если первая команда вернёт ноль, что говорит о её успешном выполнении, выполнятся команды в первом блоке then, иначе, если первое условие окажется ложным, и если вторая команда вернёт ноль, выполнится второй блок кода.

```
#!/bin/bash

user=anotherUser

if grep $user /etc/passwd
then

echo "The user $user Exists"

elif ls /home

then

echo "The user doesn't exist but anyway there is a directory under /home"

fi
```

В подобном скрипте можно, например, создавать нового пользователя с помощью команды useradd, если поиск не дал результатов, или делать ещё что-нибудь полезное.

## ***Сравнение чисел***

В скриптах можно сравнивать числовые значения. Ниже приведён список соответствующих команд.

n1 -eq n2 Возвращает истинное значение, если n1 равно n2.  
n1 -ge n2 Возвращает истинное значение, если n1 больше или равно n2.  
n1 -gt n2 Возвращает истинное значение, если n1 больше n2.  
n1 -le n2 Возвращает истинное значение, если n1 меньше или равно n2.  
n1 -lt n2 Возвращает истинное значение, если n1 меньше n2.  
n1 -ne n2 Возвращает истинное значение, если n1 не равно n2.

В качестве примера опробуем один из операторов сравнения. Обратите внимание на то, что выражение заключено в квадратные скобки.

```
#!/bin/bash

val1=6

if [ $val1 -gt 5 ]
```

```
then

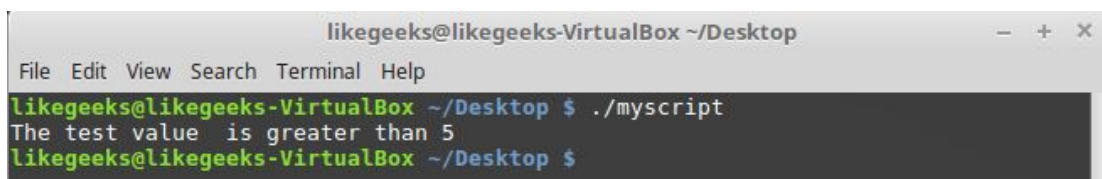
echo "The test value $value1 is greater than 5"

else

echo "The test value $value1 is not greater than 5"

fi
```

Вот что выведет эта команда.

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 a prompt 'likegeeks@likegeeks-VirtualBox ~/Desktop \$' followed by the command './myscript'. The output of the script is 'The test value is greater than 5'. The prompt then changes to 'likegeeks@likegeeks-VirtualBox ~/Desktop \$' again.

```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript
The test value is greater than 5
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

### *Сравнение чисел в скриптах*

Значение переменной val1 больше чем 5, в итоге выполняется ветвь then оператора сравнения и в консоль выводится соответствующее сообщение.

## **Сравнение строк**

В сценариях можно сравнивать и строковые значения. Операторы сравнения выглядят довольно просто, однако у операций сравнения строк есть определённые особенности, которых мы коснёмся ниже. Вот список операторов.

str1 = str2 Проверяет строки на равенство, возвращает истину, если строки идентичны.

str1 != str2 Возвращает истину, если строки не идентичны.

str1 < str2 Возвращает истину, если str1 меньше, чем str2.

str1 > str2 Возвращает истину, если str1 больше, чем str2.

-n str1 Возвращает истину, если длина str1 больше нуля.

-z str1 Возвращает истину, если длина str1 равна нулю.

Вот пример сравнения строк в сценарии:

```
#!/bin/bash

user ="likegeeks"

if [$user = $USER]

then

echo "The user $user is the current logged in user"

fi
```

В результате выполнения скрипта получим следующее.

### Сравнение строк в скриптах

Вот одна особенность сравнения строк, о которой стоит упомянуть. А именно, операторы «>» и «<» необходимо экранировать с помощью обратной косой черты, иначе скрипт будет работать неправильно, хотя сообщений об ошибках и не появится. Скрипт интерпретирует знак «>» как команду перенаправления вывода.

Вот как работа с этими операторами выглядит в коде:

```
#!/bin/bash

val1=text

val2="another text"

if [ $val1 \> $val2 ]

then

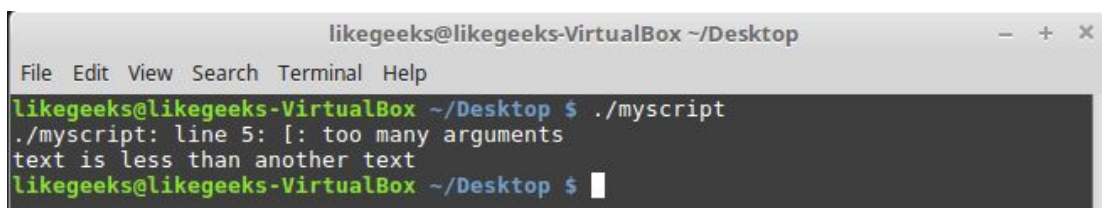
echo "$val1 is greater than $val2"

else

echo "$val1 is less than $val2"

fi
```

Вот результаты работы скрипта.



```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript
./myscript: line 5: [: too many arguments
text is less than another text
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

### Сравнение строк, выведенное предупреждение

Обратите внимание на то, что скрипт, хотя и выполняется, выдаёт предупреждение:

```
./myscript: line 5: [: too many arguments
```

Для того, чтобы избавиться от этого предупреждения, заключим \$val2 в двойные кавычки:

```
#!/bin/bash

val1=text

val2="another text"

if [ $val1 \> "$val2" ]

then

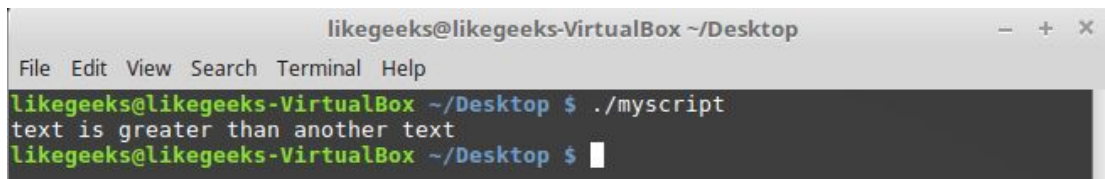
echo "$val1 is greater than $val2"

else

echo "$val1 is less than $val2"

fi
```

Теперь всё работает как надо.



```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript
text is greater than another text
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

#### Сравнение строк

Ещё одна особенность операторов «>» и «<» заключается в том, как они работают с символами в верхнем и нижнем регистрах. Для того, чтобы понять эту особенность, подготовим текстовый файл с таким содержимым:

Likegeeks

likegeeks

Сохраним его, дав имя myfile, после чего выполним в терминале такую команду:

```
sort myfile
```

Она отсортирует строки из файла так:

likegeeks

Likegeeks

Команда sort, по умолчанию, сортирует строки по возрастанию, то есть строчная буква в нашем примере меньше прописной. Теперь подготовим скрипт, который будет сравнивать те же строки:

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

```
val1=Likegeeks
```

```
val2=likegeeks
```

```
if [ $val1 \> $val2 ]
```

```
then
```

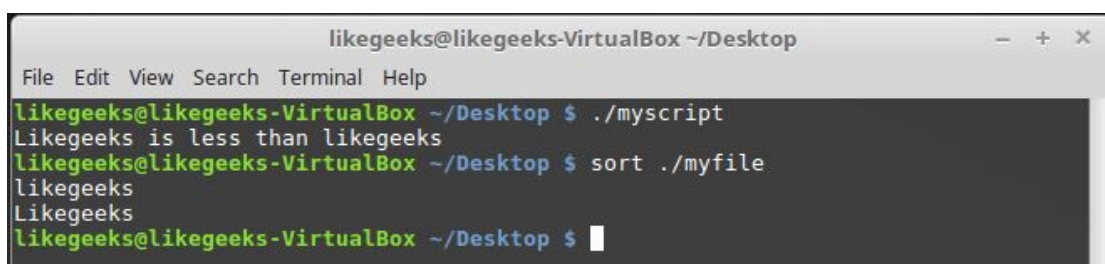
```
echo "$val1 is greater than $val2"
```

```
else
```

```
echo "$val1 is less than $val2"
```

```
fi
```

Если его запустить, окажется, что всё наоборот — строчная буква теперь больше прописной.



```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript
Likegeeks is less than likegeeks
likegeeks@likegeeks-VirtualBox ~/Desktop $ sort ./myfile
likegeeks
Likegeeks
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

Команда sort и сравнение строк в файле сценария



В командах сравнения прописные буквы меньше строчных. Сравнение строк здесь выполняется путём сравнения ASCII-кодов символов, порядок сортировки, таким образом, зависит от кодов символов. Команда `sort`, в свою очередь, использует порядок сортировки, заданный в настройках системного языка.

## Проверки файлов

Пожалуй, нижеприведённые команды используются в `bash`-скриптах чаще всего. Они позволяют проверять различные условия, касающиеся файлов. Вот список этих команд.

- `-d file` Проверяет, существует ли файл, и является ли он директорией.
- `-e file` Проверяет, существует ли файл.
- `-f file` Проверяет, существует ли файл, и является ли он файлом.
- `-r file` Проверяет, существует ли файл, и доступен ли он для чтения.
- `-s file` Проверяет, существует ли файл, и не является ли он пустым.
- `-w file` Проверяет, существует ли файл, и доступен ли он для записи.
- `-x file` Проверяет, существует ли файл, и является ли он исполняемым.
- `file1 -nt file2` Проверяет, новее ли `file1`, чем `file2`.
- `file1 -ot file2` Проверяет, старше ли `file1`, чем `file2`.
- `-O file` Проверяет, существует ли файл, и является ли его владельцем текущий пользователь.
- `-G file` Проверяет, существует ли файл, и соответствует ли его идентификатор группы идентификатору группы текущего пользователя.

Эти команды, как впрочем, и многие другие рассмотренные сегодня, несложно запомнить. Их имена, являясь сокращениями от различных слов, прямо указывают на выполняемые ими проверки. Попробуем одну из команд на практике:

```
#!/bin/bash

mydir=/home/likegeeks

if [ -d $mydir ]

then

echo "The $mydir directory exists"

cd $ mydir

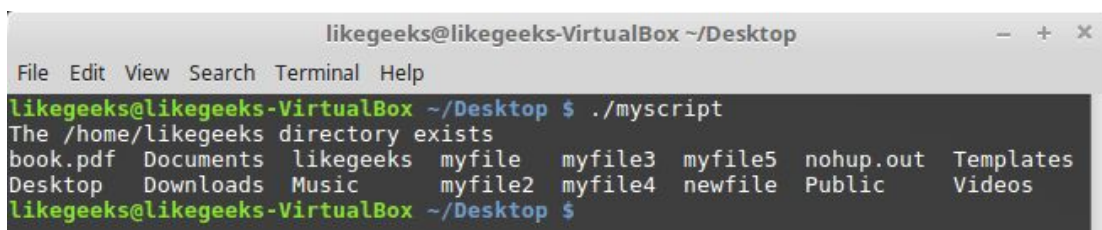
ls

else

echo "The $mydir directory does not exist"

fi
```

Этот скрипт, для существующей директории, выведет её содержимое.



```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript
The /home/likegeeks directory exists
book.pdf Documents likegeeks myfile myfile3 myfile5 nohup.out Templates
Desktop Downloads Music myfile2 myfile4 newfile Public Videos
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

Полагаем, с остальными командами вы сможете поэкспериментировать самостоятельно, все они применяются по тому же принципу.

## Итоги

Сегодня мы рассказали о том, как приступить к написанию bash-скриптов и рассмотрели некоторые базовые вещи. На самом деле, тема bash-программирования огромна. Эта статья является переводом первой части большой серии из 11 материалов. Если вы хотите продолжения прямо сейчас — вот список оригиналов этих материалов. Для удобства сюда включён и тот, перевод которого вы только что прочли.

1. [Bash Script Step By Step](#) — здесь речь идёт о том, как начать создание bash-скриптов, рассмотрено использование переменных, описаны условные конструкции, вычисления, сравнения чисел, строк, выяснение сведений о файлах.
  2. [Bash Scripting Part 2, Bash the awesome](#) — тут раскрываются особенности работы с циклами for и while.
  3. [Bash Scripting Part 3, Parameters & options](#) — этот материал посвящён параметрам командной строки и ключам, которые можно передавать скриптам, работе с данными, которые вводит пользователь, и которые можно читать из файлов.
  4. [Bash Scripting Part 4, Input & Output](#) — здесь речь идёт о дескрипторах файлов и о работе с ними, о потоках ввода, вывода, ошибок, о перенаправлении вывода.
  5. [Bash Scripting Part 5, Signals & Jobs](#) — этот материал посвящён сигналам Linux, их обработке в скриптах, запуску сценариев по расписанию.
  6. [Bash Scripting Part 6, Functions](#) — тут можно узнать о создании и использовании функций в скриптах, о разработке библиотек.
  7. [Bash Scripting Part 7, Using sed](#) — эта статья посвящена работе с потоковым текстовым редактором sed.
  8. [Bash Scripting Part 8, Using awk](#) — данный материал посвящён программированию на языке обработки данных awk.
  9. [Bash Scripting Part 9, Regular Expressions](#) — тут можно почитать об использовании регулярных выражений в bash-скриптах.
  10. [Bash Scripting Part 10, Practical Examples](#) — здесь приведены приёмы работы с сообщениями, которые можно отправлять пользователям, а так же методика мониторинга диска.
  11. [Bash Scripting Part 11, Expect Command](#) — этот материал посвящён средству Expect, с помощью которого можно автоматизировать взаимодействие с интерактивными утилитами. В частности, здесь идёт речь об expect-скриптах и об их взаимодействии с bash-скриптами и другими программами.
- Полагаем, одно из ценных свойств этой серии статей заключается в том, что она, начинаясь с самого простого, подходящего для пользователей любого уровня, постепенно ведёт к довольно серьёзным темам, давая шанс всем желающим продвинуться в деле создания сценариев командной строки Linux.

## Bash-скрипты, часть 2: циклы

В [прошлый раз](#) мы рассказали об основах программирования для bash. Даже то небольшое, что уже разобрано, позволяет всем желающим приступить к автоматизации работы в Linux. В этом материале продолжим рассказ о bash-скриптах, поговорим об управляющих конструкциях, которые позволяют выполнять повторяющиеся действия. Речь идёт о циклах `for` и `while`, о методах работы с ними и о практических примерах их применения.

### *Циклы `for`*

Оболочка bash поддерживает циклы `for`, которые позволяют организовывать перебор последовательностей значений. Вот какова базовая структура таких циклов:

```
for var in list
do
команды
done
```

В каждой итерации цикла в переменную `var` будет записываться следующее значение из списка `list`. В первом проходе цикла, таким образом, будет задействовано первое значение из списка. Во втором — второе, и так далее — до тех пор, пока цикл не дойдёт до последнего элемента.

### *Перебор простых значений*

Пожалуй, самый простой пример цикла `for` в bash-скриптах — это перебор списка простых значений:

```
#!/bin/bash

for var in first second third fourth fifth
do
echo The $var item
done
```

Ниже показаны результаты работы этого скрипта. Хорошо видно, что в переменную `$var` последовательно попадают элементы из списка. Происходит так до тех пор, пока цикл не дойдёт до последнего из них.

#### *Простой цикл `for`*

Обратите внимание на то, что переменная `$var` сохраняет значение при выходе из цикла, её содержимое можно менять, в целом, работать с ней можно как с любой другой переменной.

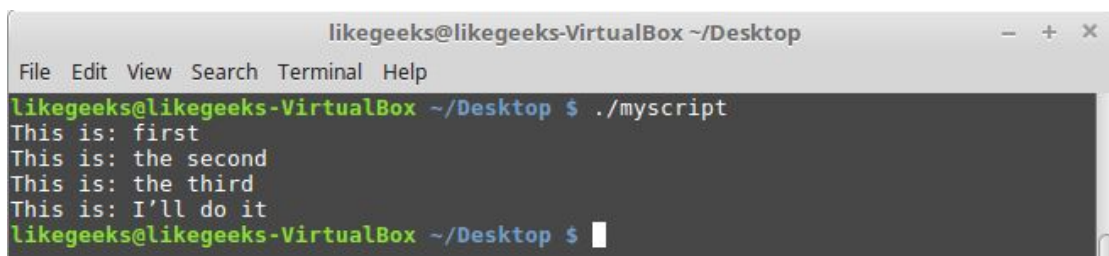
## Перебор сложных значений

В списке, использованном при инициализации цикла `for`, могут содержаться не только простые строки, состоящие из одного слова, но и целые фразы, в которые входят несколько слов и знаков препинания. Например, всё это может выглядеть так:

```
#!/bin/bash

for var in first "the second" "the third" "I'll do it"
do
echo "This is: $var"
done
```

Вот что получится после того, как этот цикл пройдёт по списку. Как видите, результат вполне ожидаем.



*Перебор сложных значений*

## Инициализация цикла списком, полученным из результатов работы команды

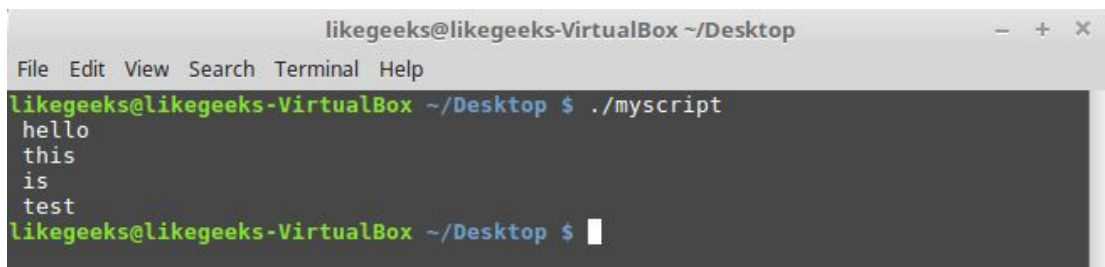
Ещё один способ инициализации цикла `for` заключается в передаче ему списка, который является результатом работы некоей команды. Тут используется подстановка команд для их исполнения и получения результатов их работы.

```
#!/bin/bash

file="myfile"

for var in $(cat $file)
do
echo " $var"
done
```

В этом примере задействована команда `cat`, которая читает содержимое файла. Полученный список значений передаётся в цикл и выводится на экран. Обратите внимание на то, что в файле, к которому мы обращаемся, содержится список слов, разделённых знаками перевода строки, пробелы при этом не используются.



```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript
hello
this
is
test
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

*Цикл, который перебирает содержимое файла*

Тут надо учесть, что подобный подход, если ожидается построчная обработка данных, не сработает для файла более сложной структуры, в строках которого может содержаться по несколько слов, разделённых пробелами. Цикл будет обрабатывать отдельные слова, а не строки. Что, если это совсем не то, что нужно?

## Разделители полей

Причина вышеописанной особенности заключается в специальной переменной окружения, которая называется IFS (Internal Field Separator) и позволяет указывать разделители полей. По умолчанию оболочка bash считает разделителями полей следующие символы:

- Пробел
- Знак табуляции
- Знак перевода строки

Если bash встречает в данных любой из этих символов, он считает, что перед ним — следующее самостоятельное значение списка.

Для того, чтобы решить проблему, можно временно изменить переменную среды IFS. Вот как это сделать в bash-скрипте, если исходить из предположения, что в качестве разделителя полей нужен только перевод строки:

```
IFS=$'\n'
```

После добавления этой команды в bash-скрипт, он будет работать как надо, игнорируя пробелы и знаки табуляции, считая разделителями полей лишь символы перевода строки.

```
#!/bin/bash

file="/etc/passwd"

IFS=$'\n'

for var in $(cat $file)
do
echo " $var"
done
```

Если этот скрипт запустить, он выведет он именно то, что от него требуется, давая, в каждой итерации цикла, доступ к очередной строке, записанной в файл.

*Построчный обход содержимого файла в цикле for*

Разделителями могут быть и другие символы. Например, выше мы выводили на экран содержимое файла `/etc/passwd`. Данные о пользователях в строках разделены с помощью двоеточий. Если в цикле нужно обрабатывать подобные строки, IFS можно настроить так:

```
IFS=:
```

## ***Обход файлов, содержащихся в директории***

Один из самых распространённых вариантов использования циклов `for` в `bash`-скриптах заключается в обходе файлов, находящихся в некоей директории, и в обработке этих файлов.

Например, вот как можно вывести список файлов и папок:

```
#!/bin/bash

for file in /home/likegeeks/*
do
    if [ -d "$file" ]
    then
        echo "$file is a directory"
    elif [ -f "$file" ]
    then
        echo "$file is a file"
    fi
done
```

Если вы разобрались с [предыдущим материалом](#) из этой серии статей, вам должно быть понятно устройство конструкции `if-then`, а так же то, как отличить файл от папки. Если вам сложно понять вышеприведённый код, перечитайте этот материал.

Вот что выведет скрипт.

### *Вывод содержимого папки*

Обратите внимание на то, как мы инициализируем цикл, а именно — на подстановочный знак «\*» в конце адреса папки. Этот символ можно воспринимать как шаблон, означающий: «все файлы с любыми именами». он позволяет организовать автоматическую подстановку имён файлов, которые соответствуют шаблону.

При проверке условия в операторе if, мы заключаем имя переменной в кавычки. Сделано это потому что имя файла или папки может содержать пробелы.

## **Циклы *for* в стиле C**

Если вы знакомы с языком программирования C, синтаксис описания bash-циклов for может показаться вам странным, так как привыкли вы, очевидно, к такому описанию циклов:

```
for (i = 0; i < 10; i++)  
{  
    printf("number is %d\n", i);  
}
```

В bash-скриптах можно использовать циклы for, описание которых выглядит очень похожим на циклы в стиле C, правда, без некоторых отличий тут не обошлось. Схема цикла при подобном подходе выглядит так:

```
for (( начальное значение переменной ; условие окончания цикла; изменение переменной ))
```

На bash это можно написать так:

```
for (( a = 1; a < 10; a++ ))
```

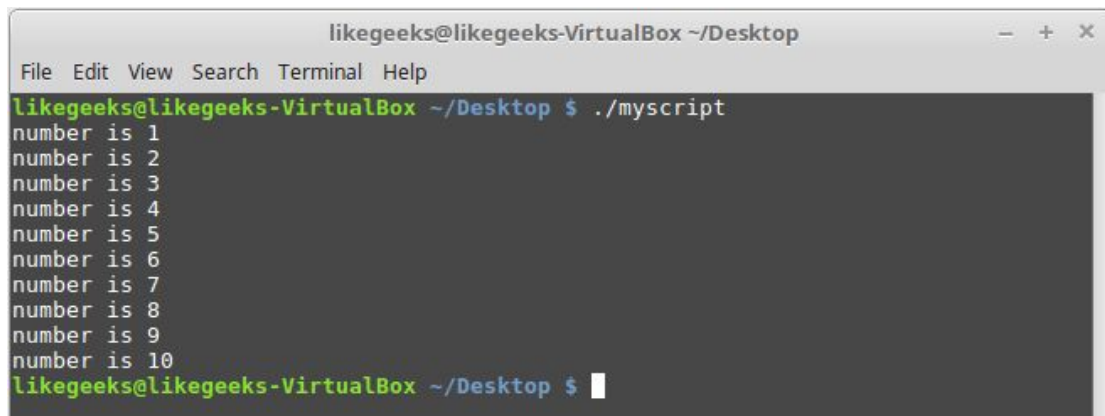
А вот рабочий пример:

```
#!/bin/bash  
  
for (( i=1; i <= 10; i++ ))  
  
do
```

```
echo "number is $i"
```

```
done
```

Этот код выведет список чисел от 1 до 10.

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 prompt is 'likegeeks@likegeeks-VirtualBox ~/Desktop \$'. The user has entered './myscript'. The output of the script is a list of numbers from 1 to 10, each preceded by 'number is '. The prompt is now 'likegeeks@likegeeks-VirtualBox ~/Desktop \$' with a cursor.

*Работа цикла в стиле C*

## Цикл **while**

Конструкция **for** — не единственный способ организации циклов в **bash**-скриптах. Здесь можно пользоваться и циклами **while**. В таком цикле можно задать команду проверки некоего условия и выполнять тело цикла до тех пор, пока проверяемое условие возвращает ноль, или сигнал успешного завершения некоей операции. Когда условие цикла вернёт ненулевое значение, что означает ошибку, цикл остановится.

Вот схема организации циклов **while**

```
while команда проверки условия
```

```
do
```

```
другие команды
```

```
done
```

Взглянем на пример скрипта с таким циклом:

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

```
var1=5
```

```
while [ $var1 -gt 0 ]
```

```
do
```

```
echo $var1
```

```
var1=$(( $var1 - 1 ))
```

```
done
```

На входе в цикл проверяется, больше ли нуля переменная **\$var1**. Если это так, выполняется тело цикла, в котором из значения переменной вычитается единица. Так происходит в каждой итерации, при этом мы выводим в консоль значение переменной до его модификации. Как только **\$var1** примет значение 0, цикл прекращается.



### Результат работы цикла *while*

Если не модифицировать переменную `$var1`, это приведёт к попаданию скрипта в бесконечный цикл.

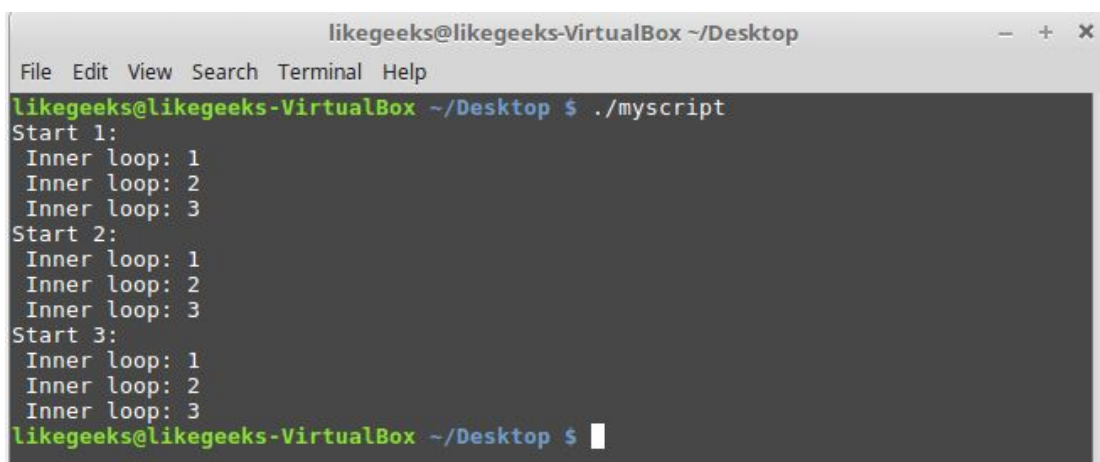
## Вложенные циклы

В теле цикла можно использовать любые команды, в том числе — запускать другие циклы. Такие конструкции называют вложенными циклами:

```
#!/bin/bash

for (( a = 1; a <= 3; a++ ))
do
    echo "Start $a:"
    for (( b = 1; b <= 3; b++ ))
    do
        echo " Inner loop: $b"
    done
done
```

Ниже показано то, что выведет этот скрипт. Как видно, сначала выполняется первая итерация внешнего цикла, потом — три итерации внутреннего, после его завершения снова в дело вступает внешний цикл, потом опять — внутренний.

A screenshot of a terminal window titled 'likegeeks@likegeeks-VirtualBox ~/Desktop'. The terminal shows the execution of a script named 'myscript'. The output consists of three main sections, each starting with 'Start 1:', 'Start 2:', and 'Start 3:' respectively. Each section contains three lines of 'Inner loop: 1', 'Inner loop: 2', and 'Inner loop: 3'. The prompt at the bottom shows the user is still in the terminal, ready to enter more commands.

```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript
Start 1:
Inner loop: 1
Inner loop: 2
Inner loop: 3
Start 2:
Inner loop: 1
Inner loop: 2
Inner loop: 3
Start 3:
Inner loop: 1
Inner loop: 2
Inner loop: 3
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

### Вложенные циклы

## Обработка содержимого файла

Чаще всего вложенные циклы используют для обработки файлов. Так, внешний цикл занимается перебором строк файла, а внутренний уже работает с каждой строкой. Вот, например, как выглядит обработка файла /etc/passwd:

```
#!/bin/bash

IFS=$'\n'

for entry in $(cat /etc/passwd)
do
    echo "Values in $entry -"

    IFS=:

    for value in $entry
    do
        echo " $value"
    done
done
```

В этом скрипте два цикла. Первый проходится по строкам, используя в качестве разделителя знак перевода строки. Внутренний занят разбором строк, поля которых разделены двоеточиями.

### *Обработка данных файла*

Такой подход можно использовать при обработке файлов формата CSV, или любых подобных файлов, записывая, по мере надобности, в переменную окружения IFS символ-разделитель.

## **Управление циклами**

Возможно, после входа в цикл, нужно будет остановить его при достижении переменной цикла определённого значения, которое не соответствует изначально заданному условию окончания цикла.

Надо ли будет в такой ситуации дожидаться нормального завершения цикла? Нет конечно, и в подобных случаях пригодятся следующие две команды:

- break
- continue

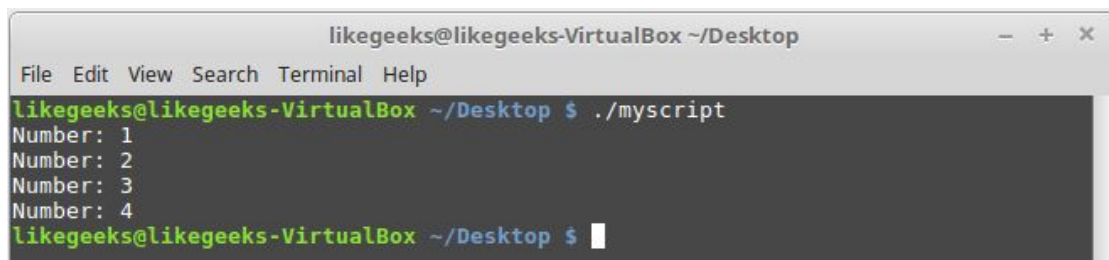
## ***Команда break***

Эта команда позволяет прервать выполнение цикла. Её можно использовать и для циклов for, и для циклов while:

```
#!/bin/bash

for var1 in 1 2 3 4 5 6 7 8 9 10
do
    if [ $var1 -eq 5 ]
    then
        break
    fi
    echo "Number: $var1"
done
```

Такой цикл, в обычных условиях, пройдёт по всему списку значений из списка. Однако, в нашем случае, его выполнение будет прервано, когда переменная \$var1 будет равна 5.

A screenshot of a terminal window titled 'likegeeks@likegeeks-VirtualBox ~/Desktop'. The terminal shows the execution of a script named 'myscript'. The output of the script is 'Number: 1', 'Number: 2', 'Number: 3', and 'Number: 4'. The prompt then shows the user at the shell, indicating the script has finished execution.

```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript
Number: 1
Number: 2
Number: 3
Number: 4
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

### *Досрочный выход из цикла for*

Вот — то же самое, но уже для цикла while:

```
#!/bin/bash

var1=1

while [ $var1 -lt 10 ]
do
    if [ $var1 -eq 5 ]
    then
        break
    fi
```

```
echo "Iteration: $var1"

var1=$(( $var1 + 1 ))

done
```

Команда `break`, исполненная, когда значение `$var1` станет равно 5, прерывает цикл. В консоль выведется то же самое, что и в предыдущем примере.

## ***Команда `continue`***

Когда в теле цикла встречается эта команда, текущая итерация завершается досрочно и начинается следующая, при этом выхода из цикла не происходит. Посмотрим на команду `continue` в цикле `for`:

```
#!/bin/bash

for (( var1 = 1; var1 < 15; var1++ ))
do

if [ $var1 -gt 5 ] && [ $var1 -lt 10 ]

then

continue

fi

echo "Iteration number: $var1"

done
```

Когда условие внутри цикла выполняется, то есть, когда `$var1` больше 5 и меньше 10, оболочка исполняет команду `continue`. Это приводит к пропуску оставшихся в теле цикла команд и переходу к следующей итерации.

*Команда `continue` в цикле `for`*

## ***Обработка вывода, выполняемого в цикле***

Данные, выводимые в цикле, можно обработать, либо перенаправив вывод, либо передав их в конвейер. Делается это с помощью добавления команд обработки вывода после инструкции `done`. Например, вместо того, чтобы показывать на экране то, что выводится в цикле, можно записать всё это в файл или передать ещё куда-нибудь:

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

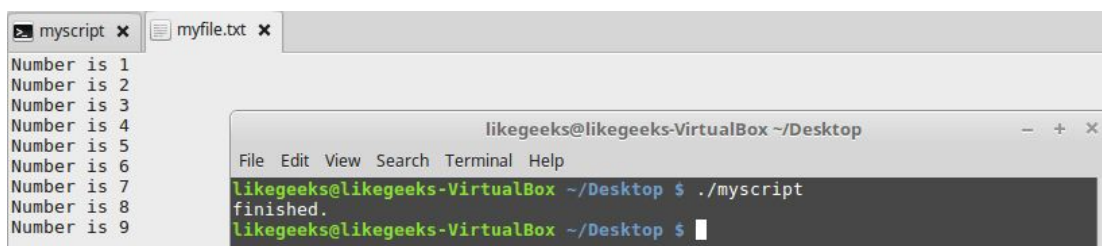
```
for (( a = 1; a < 10; a++ ))
do

echo "Number is $a"

done > myfile.txt

echo "finished."
```

Оболочка создаст файл myfile.txt и перенаправит в этот файл вывод конструкции for. Откроем файл и удостоверимся в том, что он содержит именно то, что ожидается.



*Перенаправление вывода цикла в файл*

## **Пример: поиск исполняемых файлов**

Давайте воспользуемся тем, что мы уже разобрали, и напишем что-нибудь полезное. Например, если надо выяснить, какие именно исполняемые файлы доступны в системе, можно просканировать все папки, записанные в переменную окружения PATH. Весь арсенал средств, который для этого нужен, у нас уже есть, надо лишь собрать всё это воедино:

```
#!/bin/bash

IFS=:

for folder in $PATH
do

echo "$folder:"

for file in $folder/*
do

if [ -x $file ]

then

echo " $file"

fi

done

done
```

Такой вот скрипт, небольшой и несложный, позволил получить список исполняемых файлов, хранящихся в папках из PATH.

*Поиск исполняемых файлов в папках из переменной PATH*

## **Итоги**

Сегодня мы поговорили о циклах `for` и `while` в `bash`-скриптах, о том, как их запускать, как ими управлять. Теперь вы умеете обрабатывать в циклах строки с разными разделителями, знаете, как перенаправлять данные, выведенные в циклах, в файлы, как просматривать и анализировать содержимое директорий.

Если предположить, что вы — разработчик `bash`-скриптов, который знает о них только то, что изложено в [первой части](#) этого цикла статей, и в этой, второй, то вы уже вполне можете написать кое-что полезное. Впереди — третья часть, разобравшись с которой, вы узнаете, как передавать `bash`-скриптам параметры и ключи командной строки, и что с этим всем делать.

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

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.

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

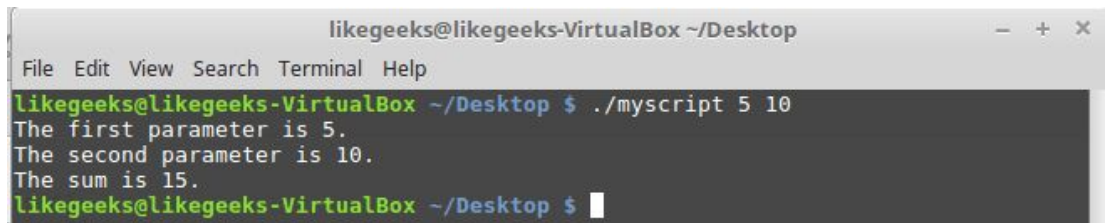
total=$(( $1 + $2 ))

echo The first parameter is $1.

echo The second parameter is $2.

echo The sum is $total.
```

Запустим скрипт и проверим результат вычислений.

A screenshot of a terminal window titled 'likegeeks@likegeeks-VirtualBox ~/Desktop'. The terminal shows the execution of a script named 'myscript' with arguments '5' and '10'. The output of the script is: 'The first parameter is 5.', 'The second parameter is 10.', and 'The sum is 15.'.

```
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:

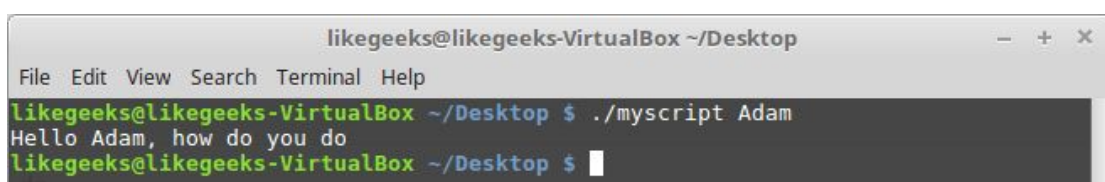
```
#!/bin/bash

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 terminal shows the execution of a script named 'myscript' with the argument 'Adam'. The output of the script is: 'Hello Adam, how do you do'.

```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript Adam
Hello Adam, how do you do
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.

*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 `$*` содержит все параметры, введенные в командной строке, в виде единого «слова». В переменной `$@` параметры разбиты на отдельные «слова». Эти параметры можно перебирать в циклах.

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

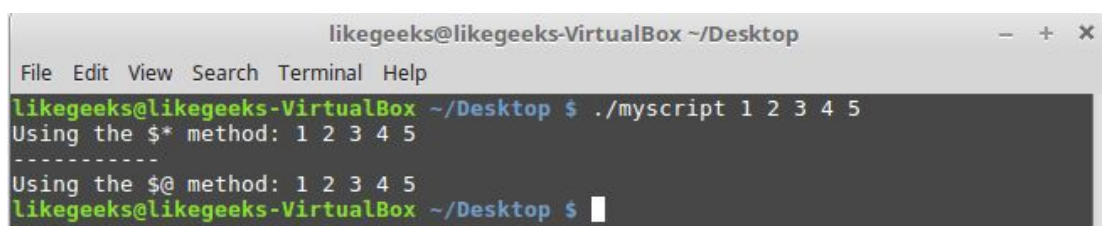
```
#!/bin/bash

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.

### *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

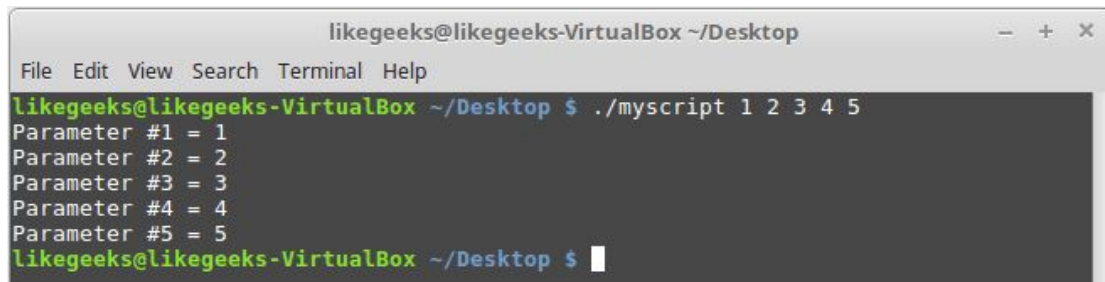
echo "Parameter # $count = $1"

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 terminal shows the command `./myscript 1 2 3 4 5` being executed. The output is:  
Parameter #1 = 1  
Parameter #2 = 2  
Parameter #3 = 3  
Parameter #4 = 4  
Parameter #5 = 5  
The prompt is `likegeeks@likegeeks-VirtualBox ~/Desktop $` with a cursor.

*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.

### *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 Обработать 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:

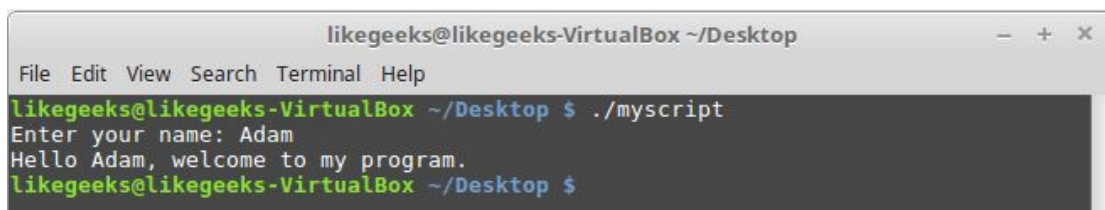
```
#!/bin/bash

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.



```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript
Enter your name: Adam
Hello Adam, welcome to my program.
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

### *Обработка пользовательского ввода*

При вызове `read` можно указывать и несколько переменных:

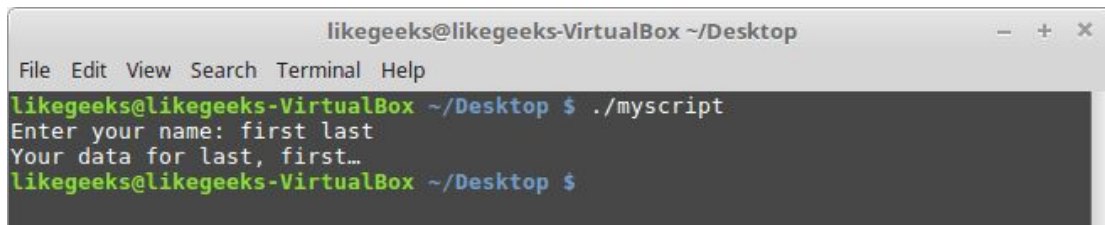
```
#!/bin/bash

read -p "Enter your name: " first last

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

Вот что выведет скрипт после запуска.





```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript
Enter your name: first last
Your data for last, first...
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.
```

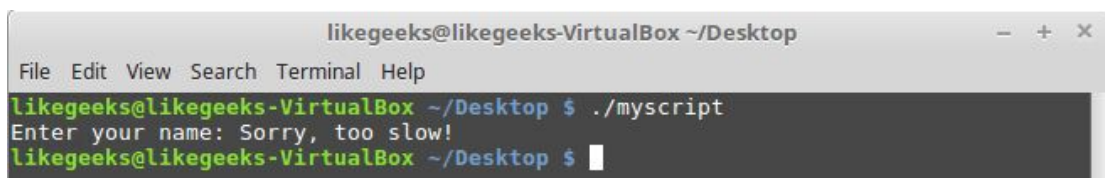
### *Использование переменной среды REPLY*

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

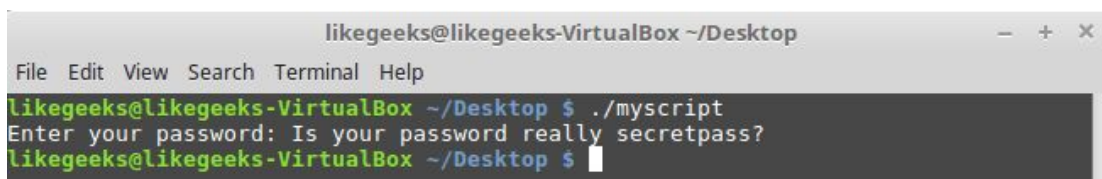
на экране данных, вводимых с клавиатуры. На самом деле, данные выводятся, но команда `read` делает цвет текста таким же, как цвет фона.

```
#!/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

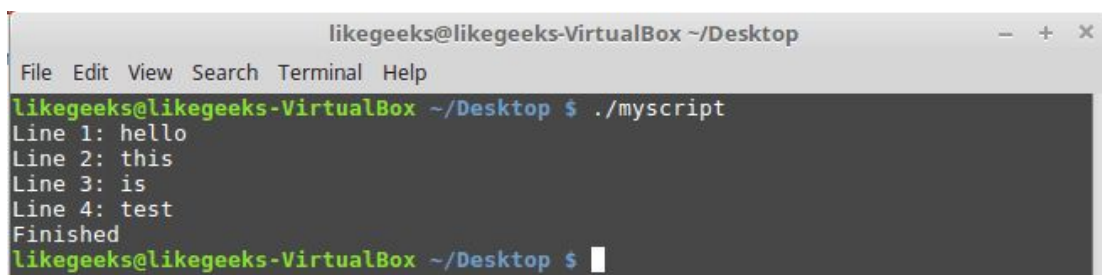
echo "Line $count: $line"

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.

## Bash-скрипты, часть 4: ввод и вывод

В прошлый раз, в третьей части этой серии материалов по bash-скриптам, мы говорили о параметрах командной строки и ключах. Наша сегодняшняя тема — ввод, вывод, и всё, что с этим связано.

Вы уже знакомы с двумя методами работы с тем, что выводят сценарии командной строки:

- Отображение выводимых данных на экране.
- Перенаправление вывода в файл.

Иногда что-то надо показать на экране, а что-то — записать в файл, поэтому нужно разобраться с тем, как в Linux обрабатывается ввод и вывод, а значит — научиться отправлять результаты работы сценариев туда, куда нужно. Начнём с разговора о стандартных дескрипторах файлов.

### **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.

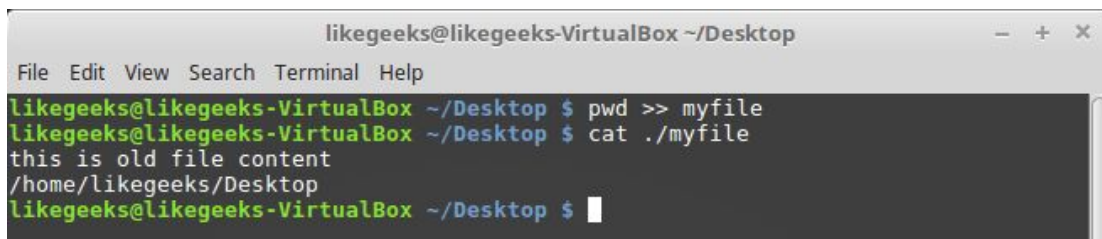
### **STDOUT**

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.



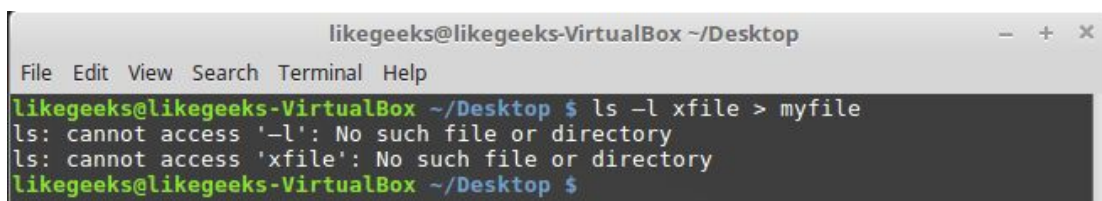
```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ pwd >> myfile
likegeeks@likegeeks-VirtualBox ~/Desktop $ cat ./myfile
this is old file content
/home/likegeeks/Desktop
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

*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.



```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ ls -l xfile > myfile
ls: cannot access '-l': No such file or directory
ls: cannot access 'xfile': No such file or directory
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

*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.

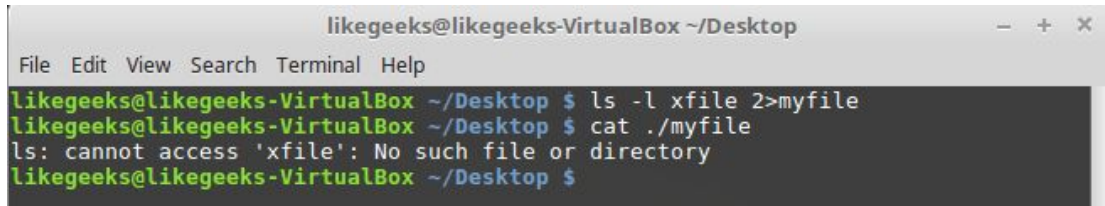
## Перенаправление потока ошибок

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.



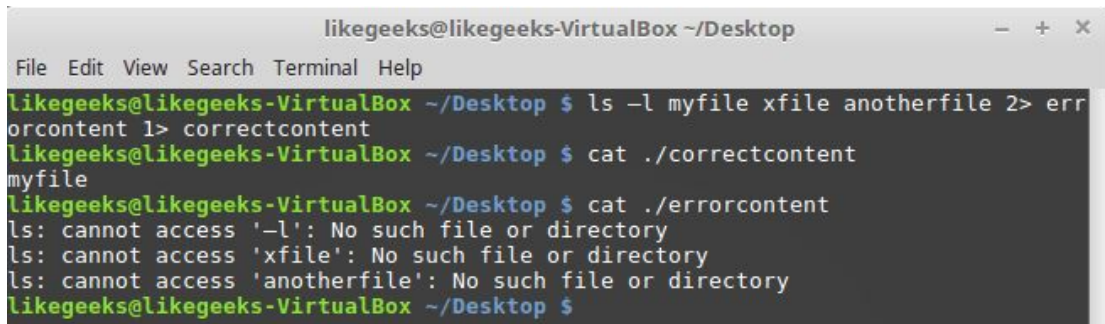
```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ ls -l xfile 2>myfile
likegeeks@likegeeks-VirtualBox ~/Desktop $ cat ./myfile
ls: cannot access 'xfile': No such file or directory
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

*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
```

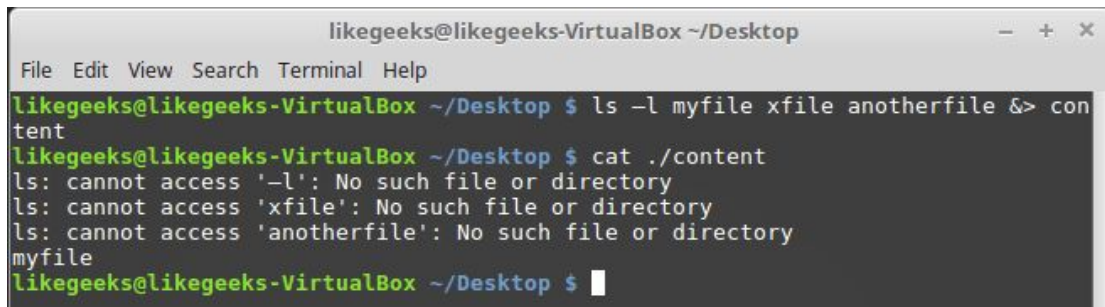


```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ ls -l myfile xfile anotherfile 2> errorcontent 1> correctcontent
likegeeks@likegeeks-VirtualBox ~/Desktop $ cat ./correctcontent
myfile
likegeeks@likegeeks-VirtualBox ~/Desktop $ cat ./errorcontent
ls: cannot access '-l': No such file or directory
ls: cannot access 'xfile': No such file or directory
ls: cannot access 'anotherfile': No such file or directory
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

*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 &>:



```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ ls -l myfile xfile anotherfile &> content
likegeeks@likegeeks-VirtualBox ~/Desktop $ cat ./content
ls: cannot access '-l': No such file or directory
ls: cannot access 'xfile': No such file or directory
ls: cannot access 'anotherfile': No such file or directory
myfile
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

*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

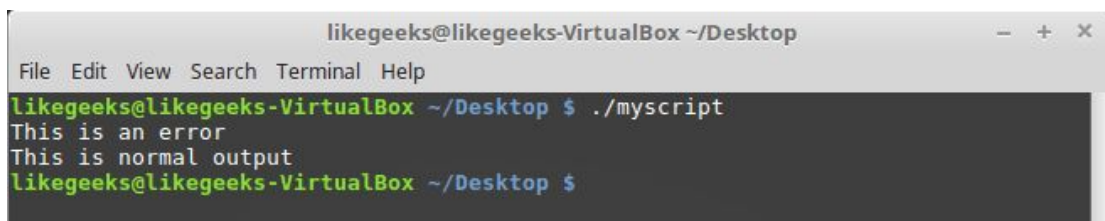
В скрипте можно перенаправить вывод отдельной строки в STDERR. Для того, чтобы это сделать, достаточно использовать команду перенаправления, указав дескриптор STDERR, при этом перед номером дескриптора надо поставить символ амперсанда (&):

```
#!/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.

A screenshot of a terminal window titled 'likegeeks@likegeeks-VirtualBox ~/Desktop'. The terminal shows the execution of a script named './myscript'. The output of the script is displayed on two lines: 'This is an error' and 'This is normal output'. The prompt 'likegeeks@likegeeks-VirtualBox ~/Desktop \$' is visible at the bottom of the terminal window.

```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript
This is an error
This is normal output
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

### *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 посмотреть 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"
```

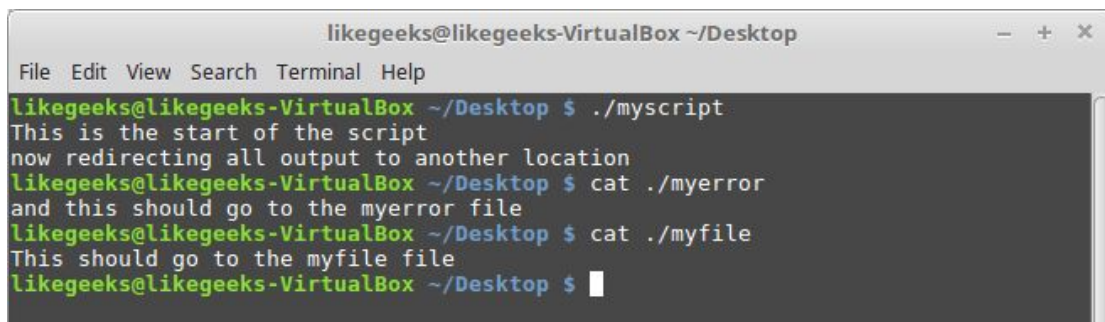
```
echo "now redirecting all output to another location"
```

```
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.



```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript
This is the start of the script
now redirecting all output to another location
likegeeks@likegeeks-VirtualBox ~/Desktop $ cat ./myerror
and this should go to the myerror file
likegeeks@likegeeks-VirtualBox ~/Desktop $ cat ./myfile
This should go to the myfile file
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

### *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:

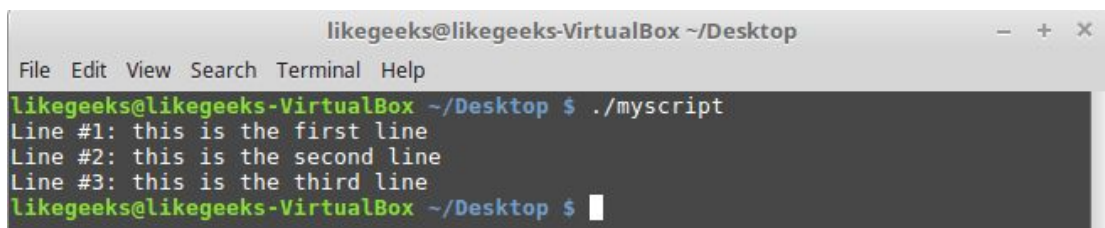
```
#!/bin/bash

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

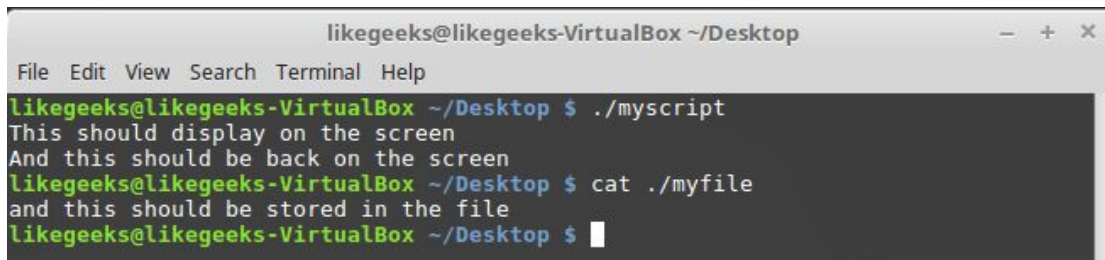
exec 3>myfile
```

```
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.

A screenshot of a terminal window titled 'likegeeks@likegeeks-VirtualBox ~/Desktop'. The terminal shows the execution of a script './myscript' which outputs 'This should display on the screen' and 'And this should be back on the screen'. Then, the command 'cat ./myfile' is executed, showing the output 'and this should be stored in the file'. The prompt is 'likegeeks@likegeeks-VirtualBox ~/Desktop \$'.

*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

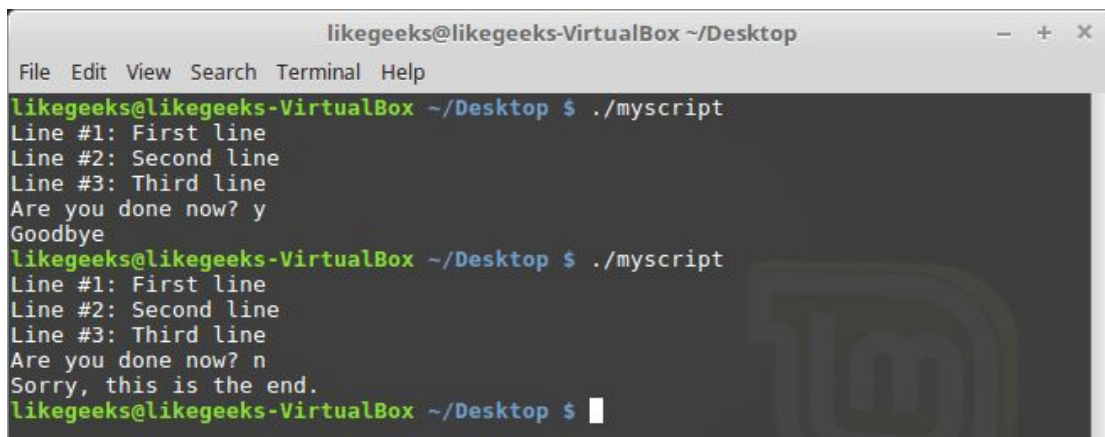
case $answer in

y) echo "Goodbye";;

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

esac
```

Испытаем сценарий.



```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript
Line #1: First line
Line #2: Second line
Line #3: Third line
Are you done now? y
Goodbye
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript
Line #1: First line
Line #2: Second line
Line #3: Third line
Are you done now? n
Sorry, this is the end.
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

### *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

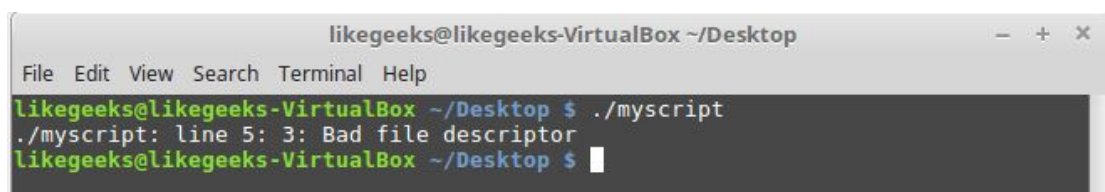
exec 3> myfile

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.



```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript
./myscript: line 5: 3: Bad file descriptor
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

### *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.

## **Получение сведений об открытых дескрипторах**

Для того, чтобы получить список всех открытых в Linux дескрипторов, можно воспользоваться командой `lsof`. Во многих дистрибутивах, вроде Fedora, утилита `lsof` находится в `/usr/sbin`. Эта команда весьма полезна, так как она выводит сведения о каждом дескрипторе, открытом в системе. Сюда входит и то, что открыли процессы, выполняемые в фоне, и то, что открыто пользователями, вошедшими в систему.

У этой команды есть множество ключей, рассмотрим самые важные.

- `-p` Позволяет указать ID процесса.
- `-d` Позволяет указать номер дескриптора, о котором надо получить сведения.

Для того, чтобы узнать PID текущего процесса, можно использовать специальную переменную окружения `$$`, в которую оболочка записывает текущий PID.

Ключ `-a` используется для выполнения операции логического И над результатами, возвращёнными благодаря использованию двух других ключей:

```
lsof -a -p $$ -d 0,1,2
```

#### *Вывод сведений об открытых дескрипторах*

Тип файлов, связанных с `STDIN`, `STDOUT` и `STDERR` — `CHR` (character mode, символьный режим). Так как все они указывают на терминал, имя файла соответствует имени устройства, назначенного терминалу. Все три стандартных файла доступны и для чтения, и для записи.

Посмотрим на вызов команды `lsof` из скрипта, в котором открыты, в дополнение к стандартным, другие дескрипторы:

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

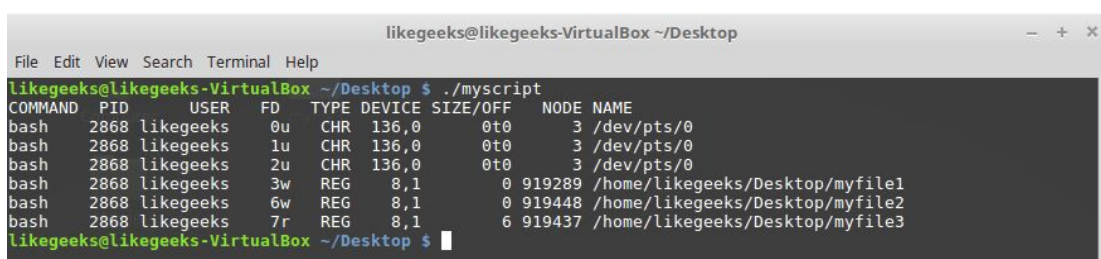
```
exec 3> myfile1
```

```
exec 6> myfile2
```

```
exec 7< myfile3
```

```
lsof -a -p $$ -d 0,1,2,3,6,7
```

Вот что получится, если этот скрипт запустить.



```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript
COMMAND  PID    USER   FD    TYPE  DEVICE  SIZE/OFF  NODE NAME
bash     2868  likegeeks 0u    CHR  136,0    0t0      3 /dev/pts/0
bash     2868  likegeeks 1u    CHR  136,0    0t0      3 /dev/pts/0
bash     2868  likegeeks 2u    CHR  136,0    0t0      3 /dev/pts/0
bash     2868  likegeeks 3w    REG    8,1      0 919289 /home/likegeeks/Desktop/myfile1
bash     2868  likegeeks 6w    REG    8,1      0 919448 /home/likegeeks/Desktop/myfile2
bash     2868  likegeeks 7r    REG    8,1      6 919437 /home/likegeeks/Desktop/myfile3
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

Скрипт открыл два дескриптора для вывода (3 и 6) и один — для ввода (7). Тут же показаны и пути к файлам, использованных для настройки дескрипторов.

## **Подавление вывода**

Иногда надо сделать так, чтобы команды в скрипте, который, например, может исполняться как фоновый процесс, ничего не выводили на экран. Для этого можно перенаправить вывод в `/dev/null`. Это — что-то вроде «чёрной дыры».

Вот, например, как подавить вывод сообщений об ошибках:

```
ls -al badfile anotherfile 2> /dev/null
```

Тот же подход используется, если, например, надо очистить файл, не удаляя его:

```
cat /dev/null > myfile
```

## **Итоги**

Сегодня вы узнали о том, как в сценариях командной строки работают ввод и вывод. Теперь вы умеете обращаться с дескрипторами файлов, создавать, просматривать и закрывать их, знаете о перенаправлении потоков ввода, вывода и ошибок. Всё это очень важно в деле разработки bash-скриптов.

В следующий раз поговорим о сигналах Linux, о том, как обрабатывать их в сценариях, о запуске заданий по расписанию и о фоновых задачах.

## Bash-скрипты, часть 5: сигналы, фоновые задачи, управление сценариями

В прошлый раз мы говорили о работе с потоками ввода, вывода и ошибок в bash-скриптах, о дескрипторах файлов и о перенаправлении потоков. Сейчас вы знаете уже достаточно много для того, чтобы писать что-то своё. На данном этапе освоения bash у вас вполне могут возникнуть вопросы о том, как управлять работающими скриптами, как автоматизировать их запуск.

До сих пор мы вводили имена скриптов в командную строку и нажимали Enter, что приводило к немедленному запуску программ, но это — не единственный способ вызова сценариев. Сегодня мы поговорим о том как скрипт может работать с сигналами Linux, о различных подходах к запуску скриптов и к управлению ими во время работы.

### 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.

К о д с и г н а л а	Н а з в а н и е	О п и с а н и е
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*

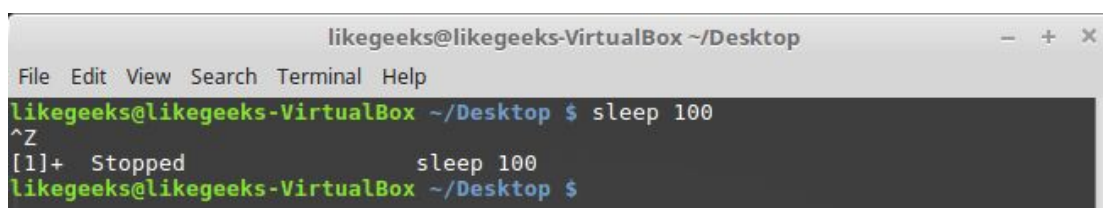
## Временная остановка процесса

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.



```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ sleep 100
^Z
[1]+  Stopped                  sleep 100
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

*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
```

```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ ps -l
F S  UID  PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000  1874  1870  0  80   0 -  5592 wait  pts/0      00:00:00 bash
0 T  1000  2050  1874  0  80   0 -  1823 signal pts/0      00:00:00 sleep
0 R  1000  2108  1874  0  80   0 -  7229 -    pts/0      00:00:00 ps
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

#### 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. Подробности о ней можно почитать [здесь](#). 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 `trap` вместо того, чтобы позволить оболочке завершить его работу.

### ***Перехват сигнала выхода из скрипта***

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

```
#!/bin/bash

trap "echo Goodbye..." EXIT

count=1

while [ $count -le 5 ]

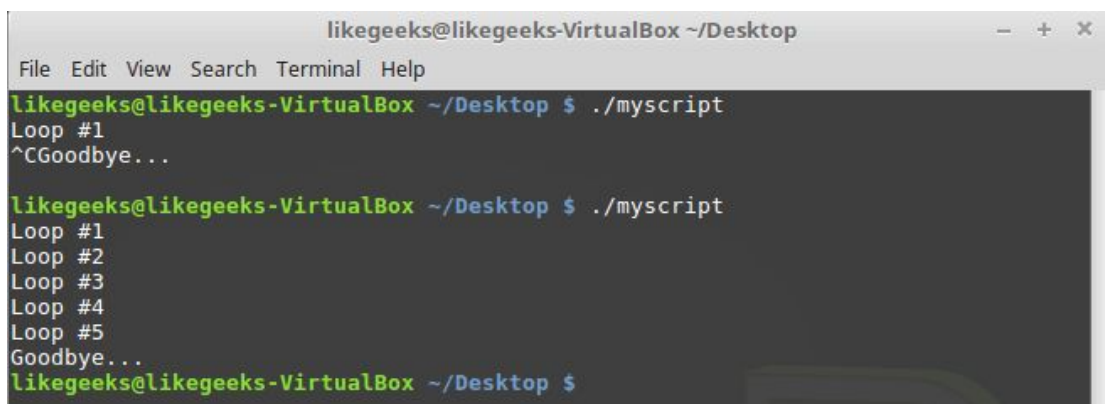
do

echo "Loop #$count"

sleep 1

count=$(( $count + 1 ))

done
```



```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript
Loop #1
^C Goodbye...

likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript
Loop #1
Loop #2
Loop #3
Loop #4
Loop #5
Goodbye...
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

### *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

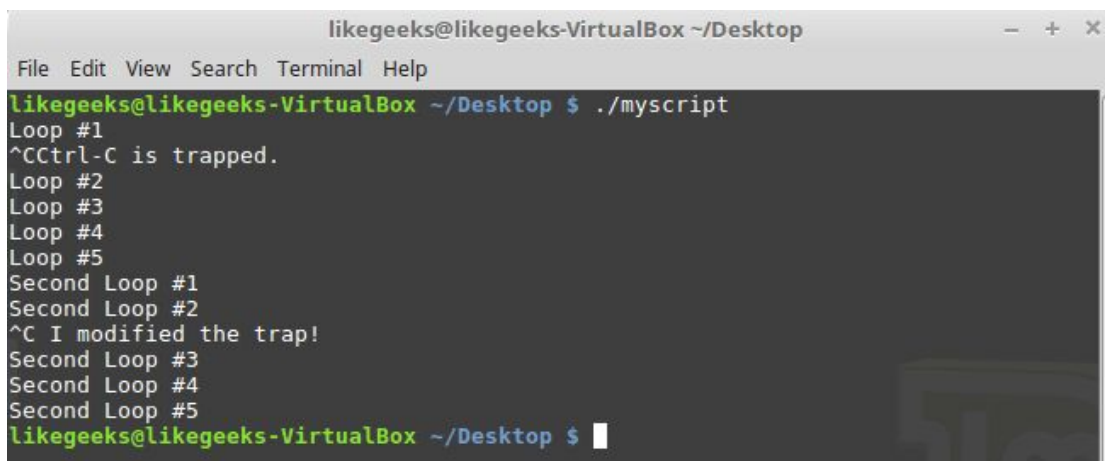
trap "echo ' I modified the trap!'" SIGINT

count=1

while [ $count -le 5 ]
do
    echo "Second Loop #$count"

    sleep 1

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



```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript
Loop #1
^C Ctrl-C is trapped.
Loop #2
Loop #3
Loop #4
Loop #5
Second Loop #1
Second Loop #2
^C I modified the trap!
Second Loop #3
Second Loop #4
Second Loop #5
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

### ***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

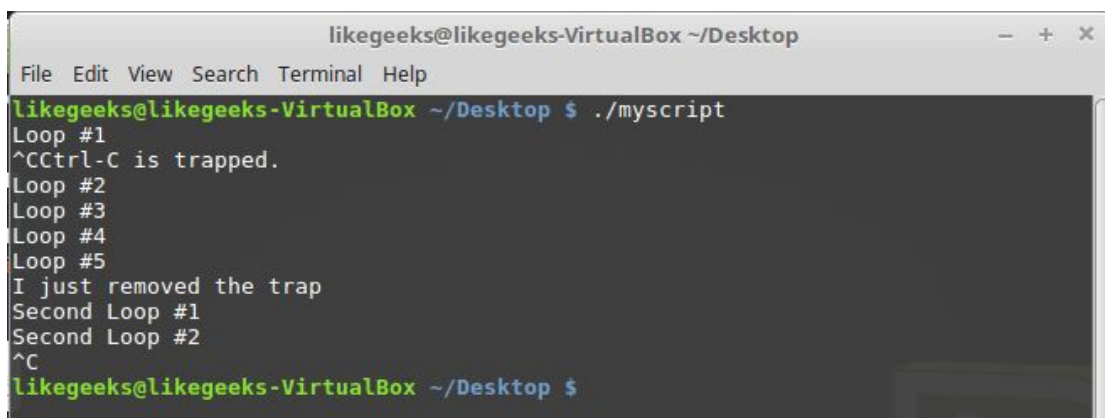
    count=$(( $count + 1 ))
done
```

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.



```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript
Loop #1
^C Ctrl-C is trapped.
Loop #2
Loop #3
Loop #4
Loop #5
I just removed the trap
Second Loop #1
Second Loop #2
^C
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

*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

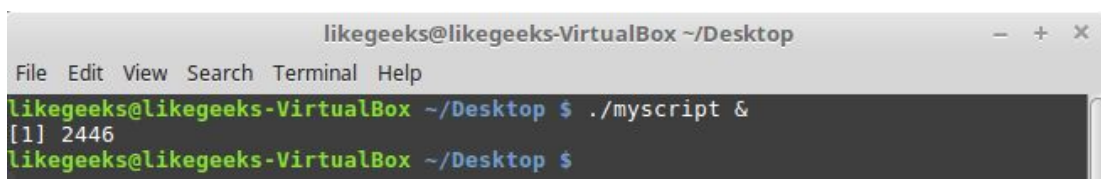
while [ $count -le 10 ]
do
    sleep 1

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

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

```
$ ./myscript &
```

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

A screenshot of a terminal window titled 'likegeeks@likegeeks-VirtualBox ~/Desktop'. The terminal shows the command './myscript &' being entered. The prompt changes to '[1] 2446', indicating the script has been launched as a background process with PID 2446. The prompt then returns to '\$'.

### ***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.

```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript &
[1] 2552
likegeeks@likegeeks-VirtualBox ~/Desktop $ ps -l
F S  UID  PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S   1000  2536  2532  0  80   0  -  5592 wait  pts/0    00:00:00 bash
0 S   1000  2552  2536  0  80   0  -  3134 wait  pts/0    00:00:00 myscript
0 S   1000  2555  2552  0  80   0  -  1823 hrtime pts/0    00:00:00 sleep
0 R   1000  2556  2536  0  80   0  -  7229 -      pts/0    00:00:00 ps
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

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.

```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ nohup ./myscript &
[1] 2605
likegeeks@likegeeks-VirtualBox ~/Desktop $ nohup: ignoring input and appending o
utput to 'nohup.out'
```

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
```

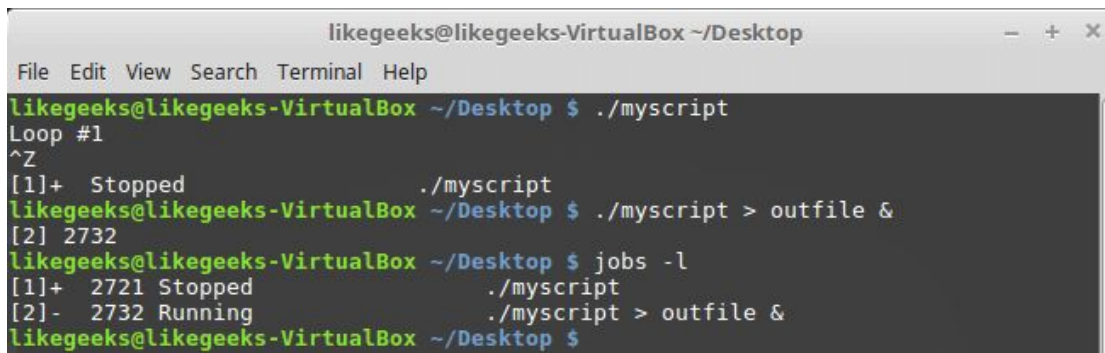
И временно остановим комбинацией клавиш CTRL + Z.

### *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.



```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript
Loop #1
^Z
[1]+  Stopped                  ./myscript
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript > outfile &
[2] 2732
likegeeks@likegeeks-VirtualBox ~/Desktop $ jobs -l
[1]+  2721 Stopped              ./myscript
[2]-  2732 Running              ./myscript > outfile &
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

### *Получение сведений о скриптах*

Ключ -l при вызове команды jobs указывает на то, что нам нужны сведения об ID процессов.

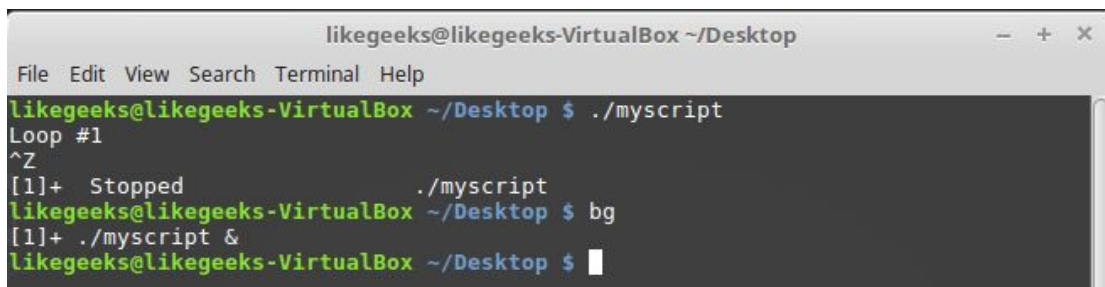
## **Перезапуск приостановленных заданий**

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
```



```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript
Loop #1
^Z
[1]+  Stopped                  ./myscript
likegeeks@likegeeks-VirtualBox ~/Desktop $ bg
[1]+  ./myscript &
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

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 -М 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:

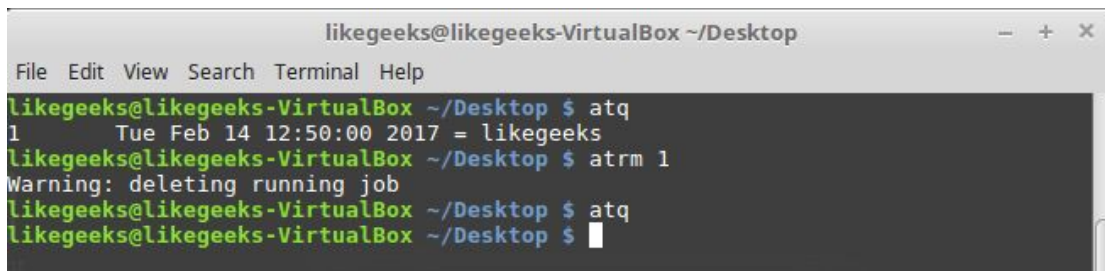
```
$ atq
```

### *Список заданий, ожидающих выполнения*

## **Удаление заданий, ожидающих выполнения**

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

```
$ atrm 18
```

A screenshot of a terminal window titled 'likegeeks@likegeeks-VirtualBox ~/Desktop'. The terminal shows the following commands and output:   
1. Command: `atq`   
Output: `1 Tue Feb 14 12:50:00 2017 = likegeeks`   
2. Command: `atrm 1`   
Output: `Warning: deleting running job`   
3. Command: `atq`   
Output: (empty)   
4. Command: `$` (prompt)   
The terminal has a menu bar with 'File', 'Edit', 'View', 'Search', 'Terminal', and 'Help'.

### *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](#) 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`.

### ***Итоги***

Сегодня мы разобрали вопросы, касающиеся управления жизненным циклом сценариев, поговорили о том, как запускать скрипты в фоне, как планировать их выполнение по расписанию. В следующий раз читайте о функциях в `bash`-скриптах и о разработке библиотек.

## Bash-скрипты, часть 6: функции и разработка библиотек

Занимаясь разработкой bash-скриптов, вы рано или поздно столкнётесь с тем, что вам периодически приходится использовать одни и те же фрагменты кода. Постоянно набирать их вручную скучно, а копирование и вставка — не наш метод. Как быть? Хорошо бы найти средство, которое позволяет один раз написать блок кода и, когда он понадобится снова, просто сослаться на него в скрипте.

Оболочка bash предоставляет такую возможность, позволяя создавать функции. Функции bash — это именованные блоки кода, которые можно повторно использовать в скриптах.

### ***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.

```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript.sh
This is an example of using a function
This is an example of using a function
This is an example of using a function
This is the end of the loop
This is an example of using a function
End of the script
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

### *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 {

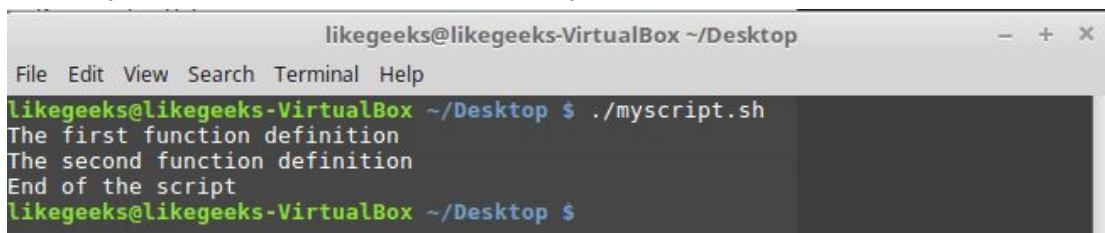
echo "The second function definition"

}

myfunc

echo "End of the script"
```

Как видно, новая функция преспокойно затёрла старую.



```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript.sh
The first function definition
The second function definition
End of the script
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

*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.



```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript.sh
Enter a value: 10
adding value
The new value is 20
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

*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

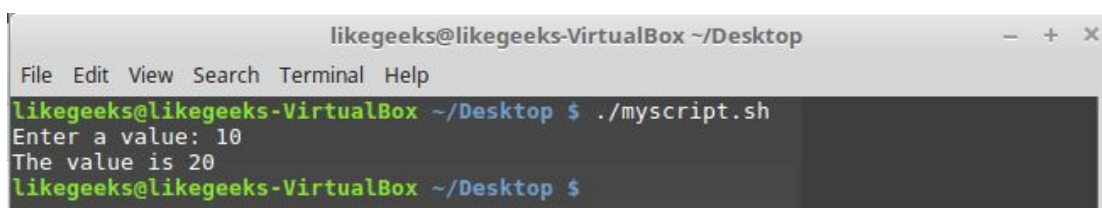
echo $(( $value + 10 ))

}

result=$( myfunc)

echo "The value is $result"
```

Вот что получится после вызова данного скрипта.



```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript.sh
Enter a value: 10
The value is 20
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

*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](#) 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, напомним такой сценарий:

```
#!/bin/bash

function myfunc {

echo $(( $1 + $2 ))

}

if [ $# -eq 2 ]

then

value=$(( myfunc))

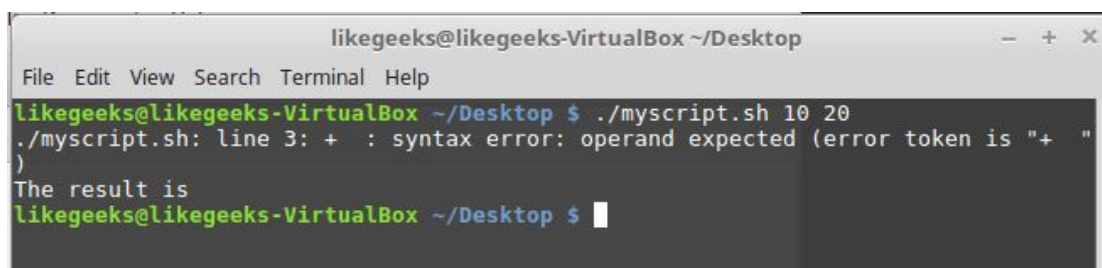
echo "The result is $value"

else

echo "Usage: myfunc a b"

fi
```

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



```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript.sh 10 20
./myscript.sh: line 3: + : syntax error: operand expected (error token is '+ ')
The result is
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

*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

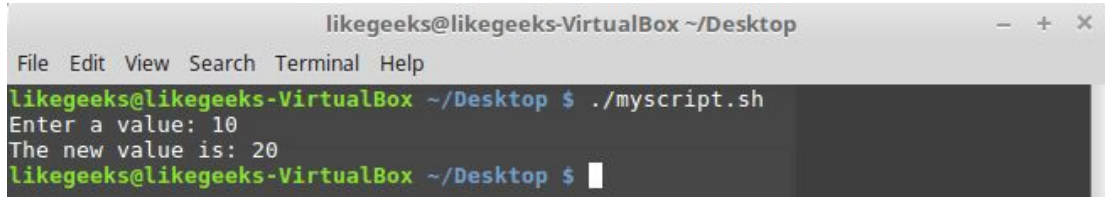
function myfunc {

value=$(( $value + 10 ))

```

```
}  
  
read -p "Enter a value: " value  
  
myfunc  
  
echo "The new value is: $value"
```

Вот что выведет этот сценарий.

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 following commands and output:

```
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript.sh  
Enter a value: 10  
The new value is: 20  
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

*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:

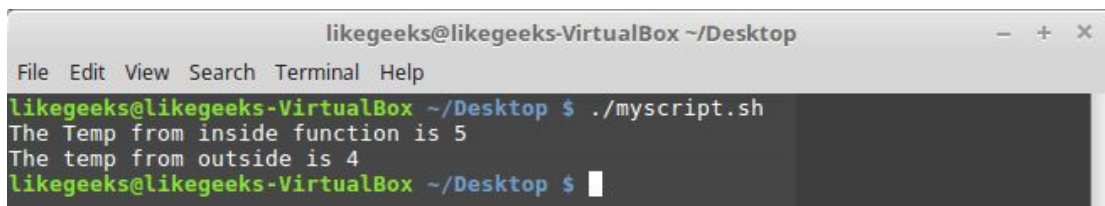
```
local temp=$(( $value + 5 ))
```

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.

A screenshot of a terminal window titled 'likegeeks@likegeeks-VirtualBox ~/Desktop'. The terminal shows a menu with 'File', 'Edit', 'View', 'Search', 'Terminal', and 'Help'. Below the menu, the prompt 'likegeeks@likegeeks-VirtualBox ~/Desktop \$' is followed by the command './myscript.sh'. The output of the script is displayed on two lines: 'The Temp from inside function is 5' and 'The temp from outside is 4'. The prompt then shows 'likegeeks@likegeeks-VirtualBox ~/Desktop \$' with a cursor.

### *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[*]}"
```

```

}

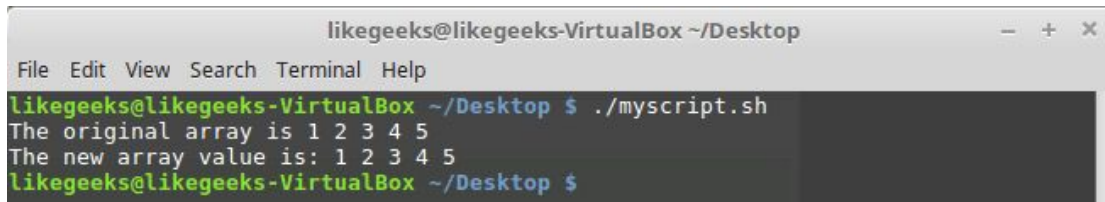
myarray=(1 2 3 4 5)

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

myfunc ${myarray[*]}

```

Запустим сценарий.



```

likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./myscript.sh
The original array is 1 2 3 4 5
The new array value is: 1 2 3 4 5
likegeeks@likegeeks-VirtualBox ~/Desktop $

```

*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](#) 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 понадобится 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.

## **Итоги**

Функции в `bash`-скриптах позволяют оформлять блоки кода и вызывать их в скриптах. А наиболее часто используемые функции стоит выделить в библиотеки, которые можно подключать к скриптам, используя оператор `source`. Если же среди ваших функций найдутся такие, без которых вы прямо таки жить не можете — библиотеки с ними можно подключить в файле `.bashrc`. Это позволит удобно пользоваться ими в командной строке или в других скриптах. Главное — чтобы имена ваших функций не совпадали с именами встроенных команд.

На сегодня это всё. В следующий раз поговорим об утилите `sed` — мощном средстве обработки строк.

## Bash-скрипты, часть 7: sed и обработка текстов

В прошлый раз мы говорили о функциях в bash-скриптах, в частности, о том, как вызывать их из командной строки. Наша сегодняшняя тема — весьма полезный инструмент для обработки строковых данных — утилита Linux, которая называется *sed*. Её часто используют для работы с текстами, имеющими вид лог-файлов, конфигурационных и других файлов.

Если вы, в bash-скриптах, каким-то образом обрабатываете данные, вам не помешает знакомство с инструментами [sed](#) и *gawk*. Тут мы сосредоточимся на *sed* и на работе с текстами, так как это — очень важный шаг в нашем путешествии по бескрайним просторам разработки bash-скриптов.

Сейчас мы разберём основы работы с *sed*, а так же рассмотрим более трёх десятков примеров использования этого инструмента.

### Основы работы с sed

Утилиту *sed* называют потоковым текстовым редактором. В интерактивных текстовых редакторах, наподобие *nano*, с текстами работают, используя клавиатуру, редактируя файлы, добавляя, удаляя или изменяя тексты. *Sed* позволяет редактировать потоки данных, основываясь на заданных разработчиком наборах правил. Вот как выглядит схема вызова этой команды:

```
$ sed options file
```

По умолчанию *sed* применяет указанные при вызове правила, выраженные в виде набора команд, к STDIN. Это позволяет передавать данные непосредственно *sed*. Например, так:

```
$ echo "This is a test" | sed 's/test/another test/'
```

Вот что получится при выполнении этой команды.

#### Простой пример вызова sed

В данном случае *sed* заменяет слово «test» в строке, переданной для обработки, словами «another test». Для оформления правила обработки текста, заключённого в кавычки, используются прямые слэши. В нашем случае применена команда вида *s/pattern1/pattern2/*. Буква «s» — это сокращение слова «substitute», то есть — перед нами команда замены. *Sed*, выполняя эту команду, просмотрит переданный текст и заменит найденные в нём фрагменты (о том — какие именно, поговорим ниже), соответствующие *pattern1*, на *pattern2*.

Выше приведён примитивный пример использования *sed*, нужный для того, чтобы ввести вас в курс дела. На самом деле, *sed* можно применять в гораздо более сложных сценариях обработки текстов, например — для работы с файлами.

Ниже показан файл, в котором содержится фрагмент текста, и результаты его обработки такой командой:

```
$ sed 's/test/another test' ./myfile
```

### Текстовый файл и результаты его обработки

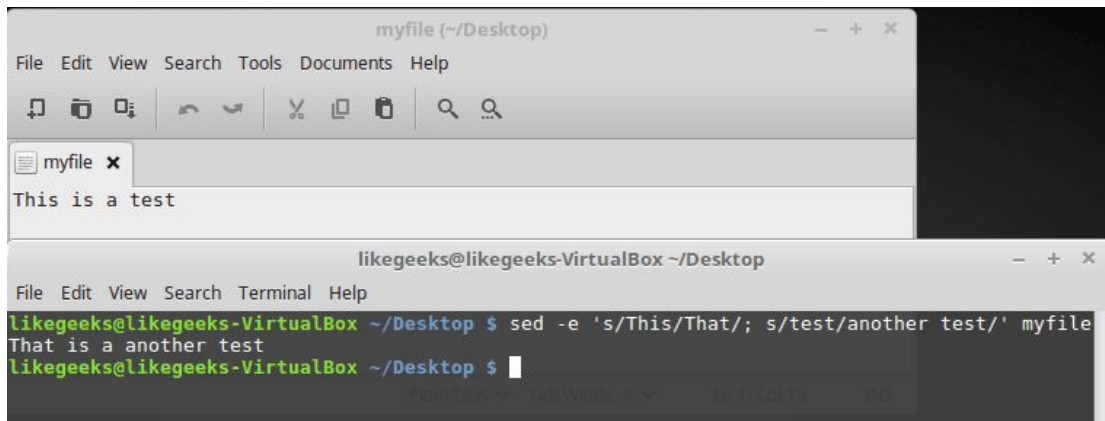
Здесь применён тот же подход, который мы использовали выше, но теперь sed обрабатывает текст, хранящийся в файле. При этом, если файл достаточно велик, можно заметить, что sed обрабатывает данные порциями и выводит то, что обработано, на экран, не дожидаясь обработки всего файла.

Sed не меняет данные в обрабатываемом файле. Редактор читает файл, обрабатывает прочитанное, и отправляет то, что получилось, в STDOUT. Для того, чтобы убедиться в том, что исходный файл не изменился, достаточно, после того, как он был передан sed, открыть его. При необходимости вывод sed можно перенаправить в файл, возможно — перезаписать старый файл. Если вы знакомы с одним из предыдущих [материалов](#) этой серии, где речь идёт о перенаправлении потоков ввода и вывода, вы вполне сможете это сделать.

## Выполнение наборов команд при вызове sed

Для выполнения нескольких действий с данными, используйте ключ -e при вызове sed. Например, вот как организовать замену двух фрагментов текста:

```
$ sed -e 's/This/That/; s/test/another test/' ./myfile
```



### Использование ключа -e при вызове sed

К каждой строке текста из файла применяются обе команды. Их нужно разделить точкой с запятой, при этом между окончанием команды и точкой с запятой не должно быть пробела.

Для ввода нескольких шаблонов обработки текста при вызове sed, можно, после ввода первой одиночной кавычки, нажать Enter, после чего вводить каждое правило с новой строки, не забыв о закрывающей кавычке:

```
$ sed -e '  
> s/This/That/
```



```
> s/test/another test/' ./myfile
```

Вот что получится после того, как команда, представленная в таком виде, будет выполнена.

*Другой способ работы с sed*

## Чтение команд из файла

Если имеется множество команд sed, с помощью которых надо обработать текст, обычно удобнее всего предварительно записать их в файл. Для того, чтобы указать sed файл, содержащий команды, используют ключ -f:

Вот содержимое файла mycommands:

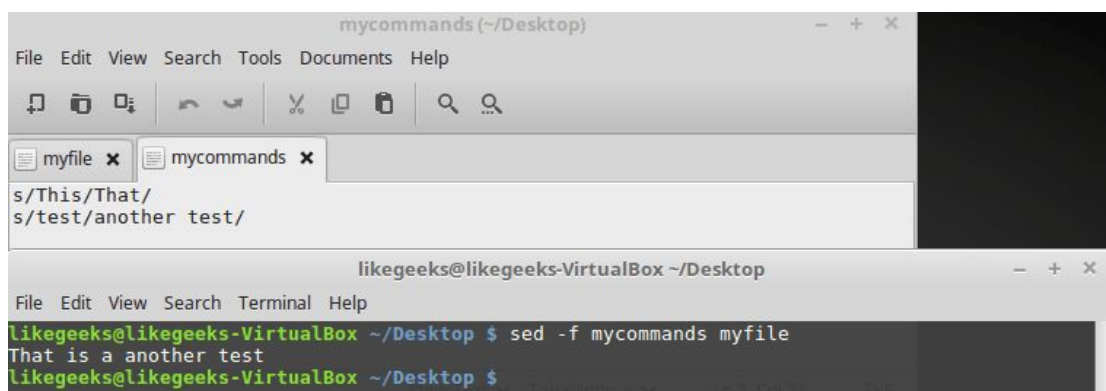
```
s/This/That/
```

```
s/test/another test/
```

Вызовем sed, передав редактору файл с командами и файл для обработки:

```
$ sed -f mycommands myfile
```

Результат при вызове такой команды аналогичен тому, который получался в предыдущих примерах.



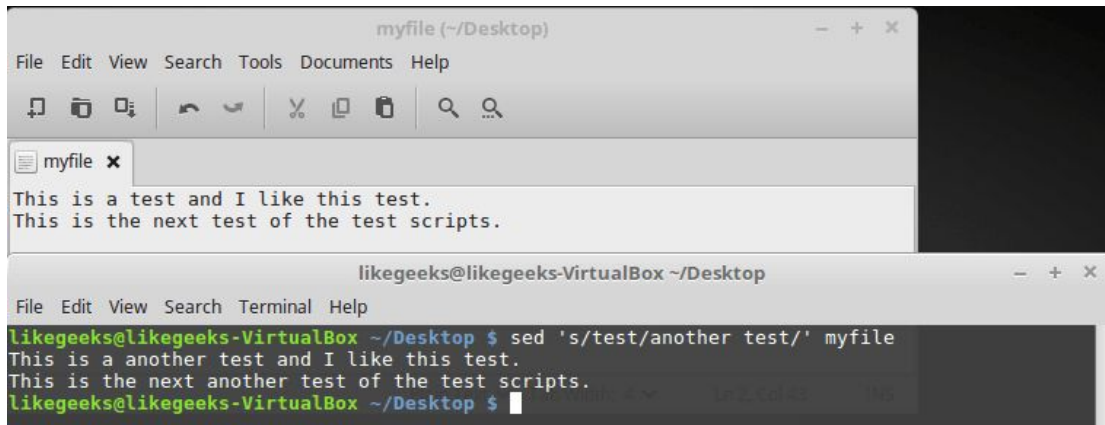
*Использование файла с командами при вызове sed*

## Флаги команды замены

Внимательно посмотрите на следующий пример.

```
$ sed 's/test/another test/' myfile
```

Вот что содержится в файле, и что будет получено после его обработки sed.



#### *Исходный файл и результаты его обработки*

Команда замены нормально обрабатывает файл, состоящий из нескольких строк, но заменяются только первые вхождения искомого фрагмента текста в каждой строке. Для того, чтобы заменить все вхождения шаблона, нужно использовать соответствующий флаг.

Схема записи команды замены при использовании флагов выглядит так:

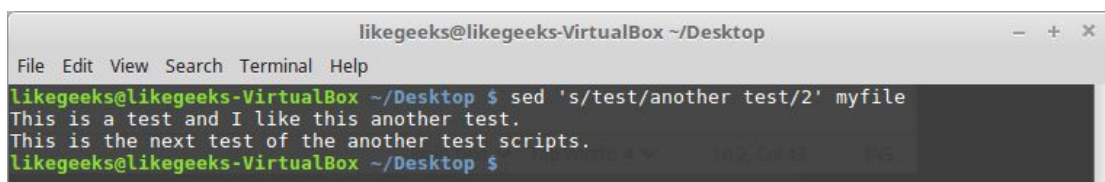
`s/pattern/replacement/flags`

Выполнение этой команды можно модифицировать несколькими способами.

- При передаче номера учитывается порядковый номер вхождения шаблона в строку, заменено будет именно это вхождение.
- Флаг `g` указывает на то, что нужно обработать все вхождения шаблона, имеющиеся в строке.
- Флаг `r` указывает на то, что нужно вывести содержимое исходной строки.
- Флаг вида `w file` указывает команде на то, что нужно записать результаты обработки текста в файл.

Рассмотрим использование первого варианта команды замены, с указанием позиции заменяемого вхождения искомого фрагмента:

```
$ sed 's/test/another test/2' myfile
```



#### *Вызов команды замены с указанием позиции заменяемого фрагмента*

Тут мы указали, в качестве флага замены, число 2. Это привело к тому, что было заменено лишь второе вхождение искомого шаблона в каждой строке.

Теперь опробуем флаг глобальной замены — `g`:

```
$ sed 's/test/another test/g' myfile
```

Как видно из результатов вывода, такая команда заменила все вхождения шаблона в тексте.

```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ sed 's/test/another test/g' myfile
This is a another test and I like this another test.
This is the next another test of the another test scripts.
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

### Глобальная замена

Флаг команды замены `r` позволяет выводить строки, в которых найдены совпадения, при этом ключ `-n`, указанный при вызове `sed`, подавляет обычный вывод:

```
$ sed -n 's/test/another test/p' myfile
```

Как результат, при запуске `sed` в такой конфигурации на экран выводятся лишь строки (в нашем случае — одна строка), в которых найден заданный фрагмент текста.

### Использование флага команды замены `r`

Воспользуемся флагом `w`, который позволяет сохранить результаты обработки текста в файл:

```
$ sed 's/test/another test/w output' myfile
```

```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ sed 's/test/another test/w output' myfile
This is a another test.
This is a different one.
likegeeks@likegeeks-VirtualBox ~/Desktop $ cat output
This is a another test.
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

### Сохранение результатов обработки текста в файл

Хорошо видно, что в ходе работы команды данные выводятся в [STDOUT](#), при этом обработанные строки записываются в файл, имя которого указано после `w`.

## Символы-разделители

Представьте, что нужно заменить `/bin/bash` на `/bin/csh` в файле `/etc/passwd`. Задача не такая уж и сложная:

```
$ sed 's/\/bin\/bash/\/bin\/csh/' /etc/passwd
```

Однако, выглядит всё это не очень-то хорошо. Всё дело в том, что так как прямые слэши используются в роли символов-разделителей, такие же символы в передаваемых `sed` строках приходится экранировать. В результате страдает читаемость команды.

К счастью, `sed` позволяет нам самостоятельно задавать символы-разделители для использования их в команде замены. Разделителем считается первый символ, который будет встречен после `s`:

```
$ sed 's!/bin/bash!/bin/csh!' /etc/passwd
```

В данном случае в качестве разделителя использован восклицательный знак, в результате код легче читать и он выглядит куда опрятнее, чем прежде.

## ***Выбор фрагментов текста для обработки***

До сих пор мы вызывали `sed` для обработки всего переданного редактору потока данных. В некоторых случаях с помощью `sed` надо обработать лишь какую-то часть текста — некую конкретную строку или группу строк. Для достижения такой цели можно воспользоваться двумя подходами:

- Задать ограничение на номера обрабатываемых строк.
- Указать фильтр, соответствующие которому строки нужно обработать.

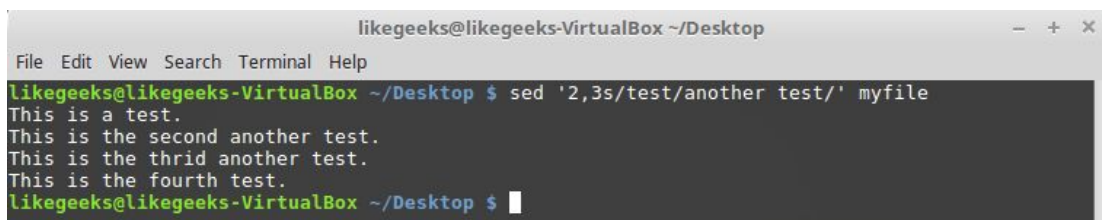
Рассмотрим первый подход. Тут допустимо два варианта. Первый, рассмотренный ниже, предусматривает указание номера одной строки, которую нужно обработать:

```
$ sed '2s/test/another test/' myfile
```

*Обработка только одной строки, номер который задан при вызове `sed`*

Второй вариант — диапазон строк:

```
$ sed '2,3s/test/another test/' myfile
```

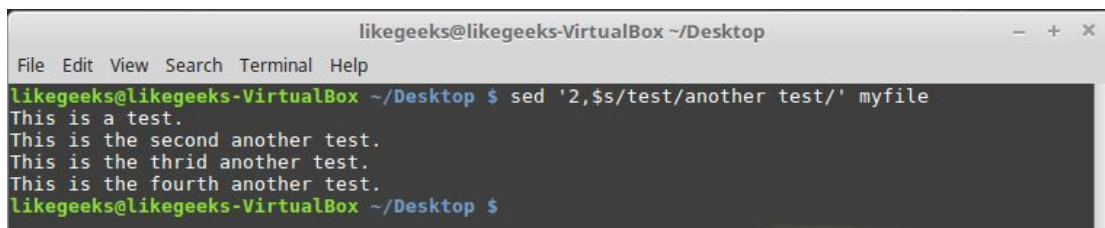


```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ sed '2,3s/test/another test/' myfile
This is a test.
This is the second another test.
This is the thrid another test.
This is the fourth test.
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

*Обработка диапазона строк*

Кроме того, можно вызвать команду замены так, чтобы файл был обработан начиная с некоей строки и до конца:

```
$ sed '2,$s/test/another test/' myfile
```

A screenshot of a terminal window titled 'likegeeks@likegeeks-VirtualBox ~/Desktop'. The terminal shows a menu bar with 'File', 'Edit', 'View', 'Search', 'Terminal', and 'Help'. Below the menu, the command 'likegeeks@likegeeks-VirtualBox ~/Desktop \$ sed '2,\$s/test/another test/' myfile' is entered. The output of the command is displayed: 'This is a test.', 'This is the second another test.', 'This is the thrid another test.', and 'This is the fourth another test.'. The prompt 'likegeeks@likegeeks-VirtualBox ~/Desktop \$' is visible at the bottom.

*Обработка файла начиная со второй строки и до конца*

Для того, чтобы обрабатывать с помощью команды замены только строки, соответствующие заданному фильтру, команду надо вызвать так:

```
$ sed '/likegeeks/s/bash/csh/' /etc/passwd
```

По аналогии с тем, что было рассмотрено выше, шаблон передаётся перед именем команды s.

*Обработка строк, соответствующих фильтру*

Тут мы использовали очень простой фильтр. Для того, чтобы в полной мере раскрыть возможности данного подхода, можно воспользоваться регулярными выражениями. О них мы поговорим в одном из следующих материалов этой серии.

## **Удаление строк**

Утилита sed годится не только для замены одних последовательностей символов в строках на другие. С её помощью, а именно, используя команду d, можно удалять строки из текстового потока.

Вызов команды выглядит так:

```
$ sed '3d' myfile
```

Мы хотим, чтобы из текста была удалена третья строка. Обратите внимание на то, что речь не идёт о файле. Файл останется неизменным, удаление отразится лишь на выводе, который сформирует sed.

*Удаление третьей строки*

Если при вызове команды d не указать номер удаляемой строки, удалены будут все строки потока. Вот как применить команду d к диапазону строк:

```
$ sed '2,3d' myfile
```

```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ sed '2,3d' myfile
This is a test.
This is the fourth test.
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

*Удаление диапазона строк*

А вот как удалить строки, начиная с заданной — и до конца файла:

```
$ sed '3,$d' myfile
```

```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ sed '3,$d' myfile
This is a test.
This is the second test.
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

*Удаление строк до конца файла*

Строки можно удалять и по шаблону:

```
$ sed '/test/d' myfile
```

*Удаление строк по шаблону*

При вызове `d` можно указывать пару шаблонов — будут удалены строки, в которых встретится шаблон, и те строки, которые находятся между ними:

```
$ sed '/second/,/fourth/d' myfile
```

```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ cat myfile
This is a test.
This is the second test.
This is the thrid test.
This is the fourth test.
This is another line
likegeeks@likegeeks-VirtualBox ~/Desktop $ sed '/second/,/fourth/d' myfile
This is a test.
This is another line
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

*Удаление диапазона строк с использованием шаблонов*

## **Вставка текста в поток**

С помощью `sed` можно вставлять данные в текстовый поток, используя команды `i` и `a`:

- Команда `i` добавляет новую строку перед заданной.
- Команда `a` добавляет новую строку после заданной.

Рассмотрим пример использования команды `i`:

```
$ echo "Another test" | sed 'i\First test '
```

*Команда `i`*

Теперь взглянем на команду `a`:

```
$ echo "Another test" | sed 'a\First test '
```

*Команда `a`*

Как видно, эти команды добавляют текст до или после данных из потока. Что если надо добавить строку где-нибудь посередине?

Тут нам поможет указание номера опорной строки в потоке, или шаблона. Учтите, что адресация строк в виде диапазона тут не подойдёт. Вызовем команду `i`, указав номер строки, перед которой надо вставить новую строку:

```
$ sed '2i\This is the inserted line.' myfile
```

*Команда `i` с указанием номера опорной строки*

Проделаем то же самое с командой `a`:

```
$ sed '2a\This is the appended line.' myfile
```



```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ cat myfile
This is a test.
This is the second test.
This is the thrid test.
This is the fourth test.
This is another line
likegeeks@likegeeks-VirtualBox ~/Desktop $ sed '2a\This is the appended line.' myfile
This is a test.
This is the second test.
This is the appended line.
This is the thrid test.
This is the fourth test.
This is another line
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

*Команда а с указанием номера опорной строки*

Обратите внимание на разницу в работе команд `i` и `a`. Первая вставляет новую строку до указанной, вторая — после.

## Замена строк

Команда `s` позволяет изменить содержимое целой строки текста в потоке данных. При её вызове нужно указать номер строки, вместо которой в поток надо добавить новые данные:

```
$ sed '3c\This is a modified line.' myfile
```

```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ cat myfile
This is a test.
This is the second test.
This is the thrid test.
This is the fourth test.
This is another line
likegeeks@likegeeks-VirtualBox ~/Desktop $ sed '3c\This is a modified line.' myfile
This is a test.
This is the second test.
This is a modified line.
This is the fourth test.
This is another line
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

*Замена строки целиком*

Если воспользоваться при вызове команды шаблоном в виде обычного текста или регулярного выражения, заменены будут все соответствующие шаблону строки:

```
$ sed '/This is/c This is a changed line of text.' myfile
```

```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ cat myfile
This is a test.
This is the second test.
This is the thrid test.
This is the fourth test.
This is another line
likegeeks@likegeeks-VirtualBox ~/Desktop $ sed '/This is/c This is a changed line of text.' myfile
This is a changed line of text.
This is a changed line of text.
This is a changed line of text.
This is a changed line of text.
This is another line
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

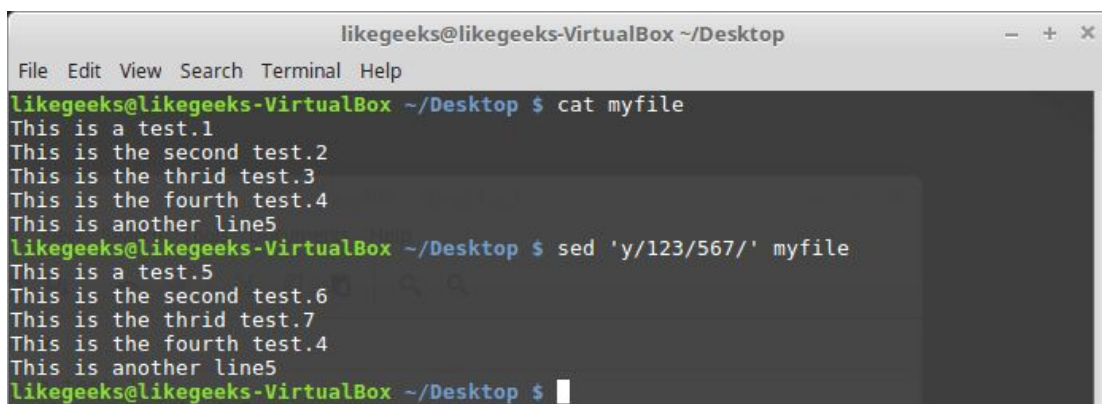
*Замена строк по шаблону*

## Замена символов



Команда `y` работает с отдельными символами, заменяя их в соответствии с переданными ей при вызове данными:

```
$ sed 'y/123/567/' myfile
```



```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ cat myfile
This is a test.1
This is the second test.2
This is the thrid test.3
This is the fourth test.4
This is another line5
likegeeks@likegeeks-VirtualBox ~/Desktop $ sed 'y/123/567/' myfile
This is a test.5
This is the second test.6
This is the thrid test.7
This is the fourth test.4
This is another line5
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

#### *Замена символов*

Используя эту команду, нужно учесть, что она применяется ко всему текстовому потоку, ограничить её конкретными вхождениями символов нельзя.

## **Вывод номеров строк**

Если вызвать `sed`, используя команду `=`, утилита выведет номера строк в потоке данных:

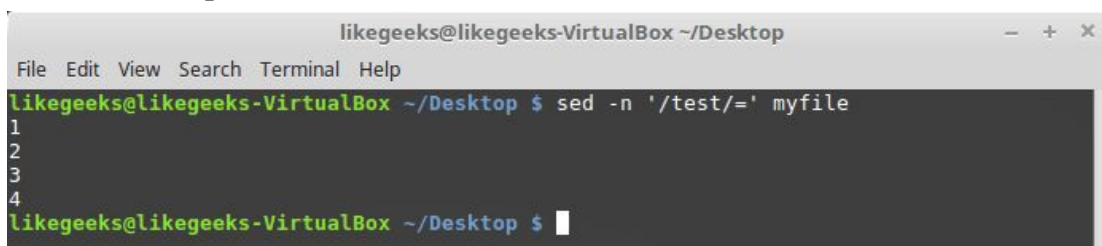
```
$ sed '=' myfile
```

#### *Вывод номеров строк*

Потоковый редактор вывел номера строк перед их содержимым.

Если передать этой команде шаблон и воспользоваться ключом `sed -n`, выведены будут только номера строк, соответствующих шаблону:

```
$ sed -n '/test/=' myfile
```



```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ sed -n '/test/=' myfile
1
2
3
4
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

#### *Вывод номеров строк, соответствующих шаблону*

## **Чтение данных для вставки из файла**

Выше мы рассматривали приёмы вставки данных в поток, указывая то, что надо вставить, прямо при вызове `sed`. В качестве источника данных можно воспользоваться и файлом. Для этого служит команда `г`, которая позволяет вставлять в поток данные из указанного файла. При её вызове можно указать номер строки, после которой надо вставить содержимое файла, или шаблон.

Рассмотрим пример:

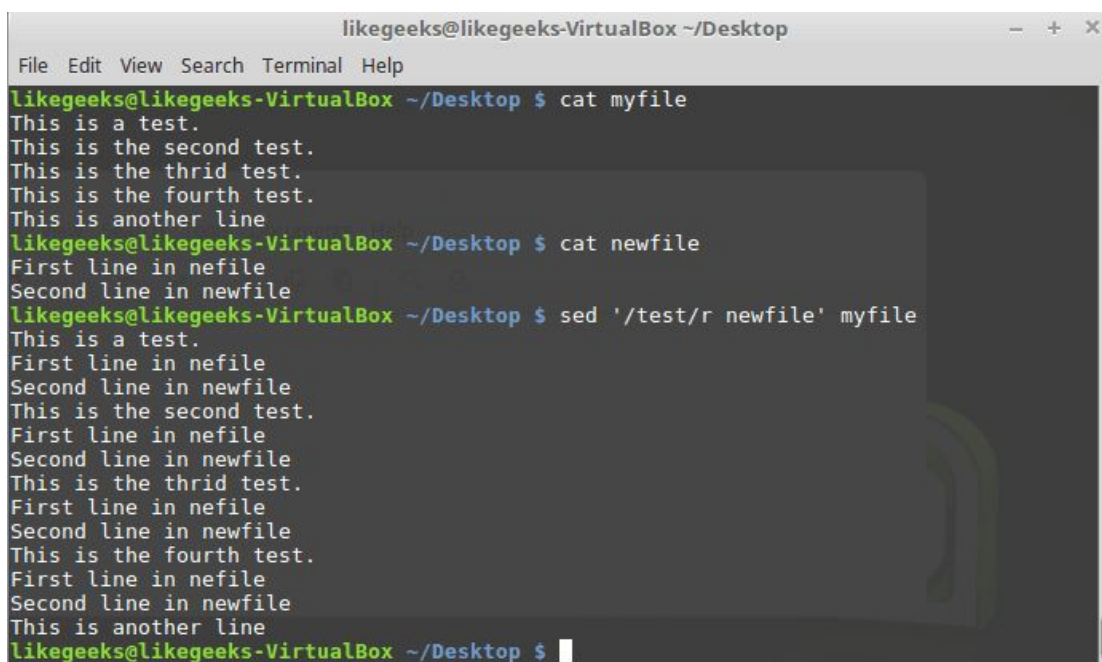
```
$ sed '3r newfile' myfile
```

#### *Вставка в поток содержимого файла*

Тут содержимое файла `newfile` было вставлено после третьей строки файла `myfile`.

Вот что произойдёт, если применить при вызове команды `г` шаблон:

```
$ sed '/test/r newfile' myfile
```



```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ cat myfile
This is a test.
This is the second test.
This is the thrird test.
This is the fourth test.
This is another line
likegeeks@likegeeks-VirtualBox ~/Desktop $ cat newfile
First line in nefile
Second line in newfile
likegeeks@likegeeks-VirtualBox ~/Desktop $ sed '/test/r newfile' myfile
This is a test.
First line in nefile
Second line in newfile
This is the second test.
First line in nefile
Second line in newfile
This is the thrird test.
First line in nefile
Second line in newfile
This is the fourth test.
First line in nefile
Second line in newfile
This is another line
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

#### *Использование шаблона при вызове команды `г`*

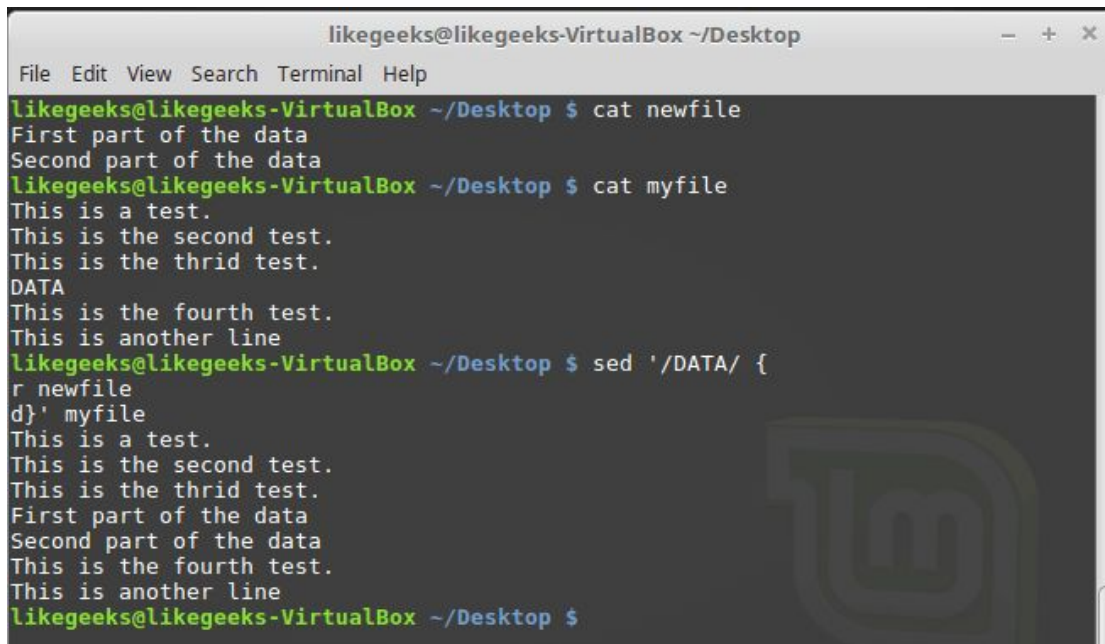
Содержимое файла будет вставлено после каждой строки, соответствующей шаблону.

### **Пример**

Представим себе такую задачу. Есть файл, в котором имеется некая последовательность символов, сама по себе бессмысленная, которую надо заменить на данные, взятые из другого файла. А именно, пусть это будет файл `newfile`, в котором роль указателя места заполнения играет последовательность символов `DATA`. Данные, которые нужно подставить вместо `DATA`, хранятся в файле `data`.

Решить эту задачу можно, воспользовавшись командами `r` и `d` потокового редактора `sed`:

```
$ Sed '/DATA>/ {  
r newfile  
d}' myfile
```



```
likegeeks@likegeeks-VirtualBox ~/Desktop  
File Edit View Search Terminal Help  
likegeeks@likegeeks-VirtualBox ~/Desktop $ cat newfile  
First part of the data  
Second part of the data  
likegeeks@likegeeks-VirtualBox ~/Desktop $ cat myfile  
This is a test.  
This is the second test.  
This is the thrid test.  
DATA  
This is the fourth test.  
This is another line  
likegeeks@likegeeks-VirtualBox ~/Desktop $ sed '/DATA/ {  
r newfile  
d}' myfile  
This is a test.  
This is the second test.  
This is the thrid test.  
First part of the data  
Second part of the data  
This is the fourth test.  
This is another line  
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

*Замена указателя места заполнения на реальные данные*

Как видите, вместо заполнителя `DATA` `sed` добавил в выходной поток две строки из файла `data`.

## Итоги

Сегодня мы рассмотрели основы работы с потоковым редактором `sed`. На самом деле, `sed` — это огромнейшая тема. Его изучение вполне можно сравнить с изучением нового языка программирования, однако, поняв основы, вы сможете освоить `sed` на любом необходимом вам уровне. В результате ваши возможности по обработке с его помощью текстов будет ограничивать лишь воображение.

На сегодня это всё. В следующий раз поговорим о языке обработки данных `awk`.

## Bash-скрипты, часть 8: язык обработки данных awk

В прошлый раз мы говорили о потоковом редакторе [sed](#) и рассмотрели немало примеров обработки текста с его помощью. Sed способен решать многие задачи, но есть у него и ограничения. Иногда нужен более совершенный инструмент для обработки данных, нечто вроде языка программирования. Собственно говоря, такой инструмент — awk.

Утилита awk, или точнее GNU awk, в сравнении с sed, выводит обработку потоков данных на более высокий уровень. Благодаря awk в нашем распоряжении оказывается язык программирования, а не довольно скромный набор команд, отдаваемых редактору. С помощью языка программирования awk можно выполнять следующие действия:

- Объявлять переменные для хранения данных.
- Использовать арифметические и строковые операторы для работы с данными.
- Использовать структурные элементы и управляющие конструкции языка, такие, как оператор if-then и циклы, что позволяет реализовать сложные алгоритмы обработки данных.
- Создавать форматированные отчёты.

Если говорить лишь о возможности создавать форматированные отчёты, которые удобно читать и анализировать, то это оказывается очень кстати при работе с лог-файлами, которые могут содержать миллионы записей. Но awk — это намного больше, чем средство подготовки отчётов.

### **Особенности вызова awk**

Схема вызова awk выглядит так:

```
$ awk options program file
```

Awk воспринимает поступающие к нему данные в виде набора записей. Записи представляют собой наборы полей. Упрощенно, если не учитывать возможности настройки awk и говорить о некоем вполне обычном тексте, строки которого разделены символами перевода строки, запись — это строка. Поле — это слово в строке.

Рассмотрим наиболее часто используемые ключи командной строки awk:

- F fs — позволяет указать символ-разделитель для полей в записи.
- f file — указывает имя файла, из которого нужно прочесть awk-скрипт.
- v var=value — позволяет объявить переменную и задать её значение по умолчанию, которое будет использовать awk.
- mf N — задаёт максимальное число полей для обработки в файле данных.
- mr N — задаёт максимальный размер записи в файле данных.
- W keyword — позволяет задать режим совместимости или уровень выдачи предупреждений awk

Настоящая мощь awk скрывается в той части команды его вызова, которая помечена выше как program. Она указывает на файл awk-скрипта, написанный программистом и предназначенный для чтения данных, их обработки и вывода результатов.

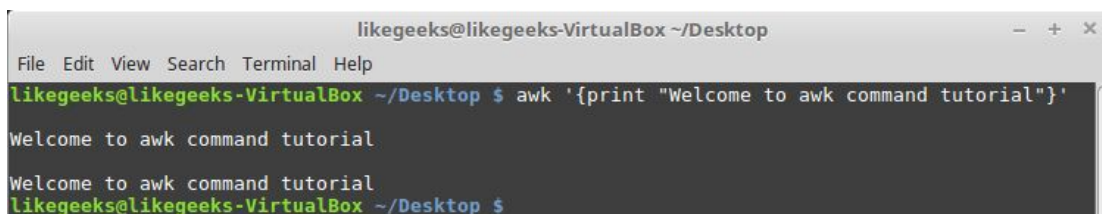
### **Чтение awk-скриптов из командной строки**

Скрипты `awk`, которые можно писать прямо в командной строке, оформляются в виде текстов команд, заключённых в фигурные скобки. Кроме того, так как `awk` предполагает, что скрипт представляет собой текстовую строку, его нужно заключить в одинарные кавычки:

```
$ awk '{print "Welcome to awk command tutorial"}'
```

Запустим эту команду... И ничего не произойдёт. Дело тут в том, что мы, при вызове `awk`, не указали файл с данными. В подобной ситуации `awk` ожидает поступления данных из [STDIN](#). Поэтому выполнение такой команды не приводит к немедленно наблюдаемым эффектам, но это не значит, что `awk` не работает — он ждёт входных данных из STDIN.

Если теперь ввести что-нибудь в консоль и нажать Enter, `awk` обработает введённые данные с помощью скрипта, заданного при его запуске. `Awk` обрабатывает текст из потока ввода построчно, этим он похож на `sed`. В нашем случае `awk` ничего не делает с данными, он лишь, в ответ на каждую новую полученную им строку, выводит на экран текст, заданный в команде `print`.

A screenshot of a terminal window titled 'likegeeks@likegeeks-VirtualBox ~/Desktop'. The terminal shows the command 'awk '{print "Welcome to awk command tutorial"}'' being entered and executed. The output is 'Welcome to awk command tutorial' printed twice, once for each line of input. The prompt 'likegeeks@likegeeks-VirtualBox ~/Desktop \$' is visible at the bottom.

*Первый запуск `awk`, вывод на экран заданного текста*

Что бы мы ни ввели, результат в данном случае будет одним и тем же — вывод текста.

Для того, чтобы завершить работу `awk`, нужно передать ему символ конца файла (EOF, End-of-File). Сделать это можно, воспользовавшись сочетанием клавиш CTRL + D.

Неудивительно, если этот первый пример показался вам не особо впечатляющим. Однако, самое интересное — впереди.

## ***Позиционные переменные, хранящие данные полей***

Одна из основных функций `awk` заключается в возможности манипулировать данными в текстовых файлах. Делается это путём автоматического назначения переменной каждому элементу в строке. По умолчанию `awk` назначает следующие переменные каждому полю данных, обнаруженному им в записи:

- `$0` — представляет всю строку текста (запись).
- `$1` — первое поле.
- `$2` — второе поле.
- `$n` — n-ное поле.

Поля выделяются из текста с использованием символа-разделителя. По умолчанию — это пробельные символы вроде пробела или символа табуляции.

Рассмотрим использование этих переменных на простом примере. А именно, обработаем файл, в котором содержится несколько строк (этот файл показан на рисунке ниже) с помощью такой команды:

```
$ awk '{print $1}' myfile
```

```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ cat myfile
This is a test.
This is the second test.
This is the thrid test.
This is the fourth test.
likegeeks@likegeeks-VirtualBox ~/Desktop $ awk '{print $1}' myfile
This
This
This
This
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

*Вывод в консоль первого поля каждой строки*

Здесь использована переменная \$1, которая позволяет получить доступ к первому полю каждой строки и вывести его на экран.

Иногда в некоторых файлах в качестве разделителей полей используется что-то, отличающееся от пробелов или символов табуляции. Выше мы упоминали ключ awk -F, который позволяет задать необходимый для обработки конкретного файла разделитель:

```
$ awk -F: '{print $1}' /etc/passwd
```

```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ awk -F: '{print $1}' /etc/passwd
root
daemon
bin
sys
sync
games
man
lp
mail
news
uucp
proxy
www-data
backup
list
```

*Указание символа-разделителя при вызове awk*

Эта команда выводит первые элементы строк, содержащихся в файле /etc/passwd. Так как в этом файле в качестве разделителей используются двоеточия, именно этот символ был передан awk после ключа -F.

## **Использование нескольких команд**

Вызов awk с одной командой обработки текста — подход очень ограниченный. Awk позволяет обрабатывать данные с использованием многострочных скриптов. Для того, чтобы передать awk многострочную команду при вызове его из консоли, нужно разделить её части точкой с запятой:

```
$ echo "My name is Tom" | awk '{ $4="Adam"; print $0 }'
```

*Вызов awk из командной строки с передачей ему многострочного скрипта*

В данном примере первая команда записывает новое значение в переменную \$4, а вторая выводит на экран всю строку.

## **Чтение скрипта awk из файла**

Awk позволяет хранить скрипты в файлах и ссылаться на них, используя ключ -f. Подготовим файл testfile, в который запишем следующее:

```
{print $1 " has a home directory at " $6}
```

Вызовем awk, указав этот файл в качестве источника команд:

```
$ awk -F: -f testfile /etc/passwd
```

### *Вызов awk с указанием файла скрипта*

Тут мы выводим из файла /etc/passwd имена пользователей, которые попадают в переменную \$1, и их домашние директории, которые попадают в \$6. Обратите внимание на то, что файл скрипта задают с помощью ключа -f, а разделитель полей, двоеточие в нашем случае, с помощью ключа -F.

В файле скрипта может содержаться множество команд, при этом каждую из них достаточно записывать с новой строки, ставить после каждой точку с запятой не требуется.

Вот как это может выглядеть:

```
{  
  
text = " has a home directory at "  
  
print $1 text $6  
  
}
```

Тут мы храним текст, используемый при выводе данных, полученных из каждой строки обрабатываемого файла, в переменной, и используем эту переменную в команде print. Если воспроизвести предыдущий пример, записав этот код в файл testfile, выведено будет то же самое.

## **Выполнение команд до начала обработки данных**

Иногда нужно выполнить какие-то действия до того, как скрипт начнёт обработку записей из входного потока. Например — создать шапку отчёта или что-то подобное.

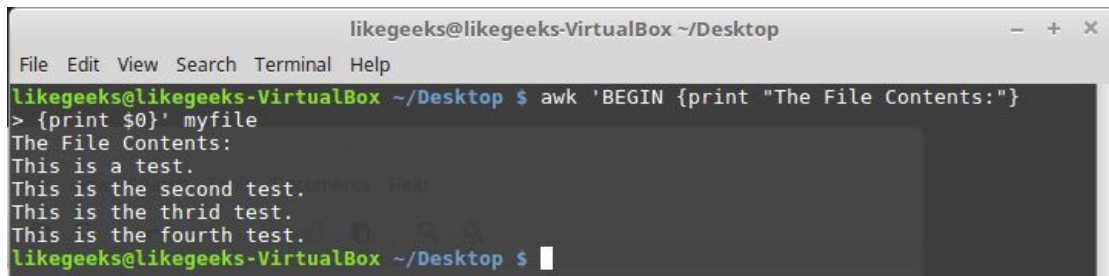


Для этого можно воспользоваться ключевым словом BEGIN. Команды, которые следуют за BEGIN, будут исполнены до начала обработки данных. В простейшем виде это выглядит так:

```
$ awk 'BEGIN {print "Hello World!"}'
```

А вот — немного более сложный пример:

```
$ awk 'BEGIN {print "The File Contents:"}  
{print $0}' myfile
```



```
likegeeks@likegeeks-VirtualBox ~/Desktop  
File Edit View Search Terminal Help  
likegeeks@likegeeks-VirtualBox ~/Desktop $ awk 'BEGIN {print "The File Contents:"}  
> {print $0}' myfile  
The File Contents:  
This is a test.  
This is the second test.  
This is the thrid test.  
This is the fourth test.  
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

*Выполнение команд до начала обработки данных*

Сначала awk исполняет блок BEGIN, после чего выполняется обработка данных. Будьте внимательны с одинарными кавычками, используя подобные конструкции в командной строке. Обратите внимание на то, что и блок BEGIN, и команды обработки потока, являются в представлении awk одной строкой. Первая одинарная кавычка, ограничивающая эту строку, стоит перед BEGIN. Вторая — после закрывающей фигурной скобки команды обработки данных.

## ***Выполнение команд после окончания обработки данных***

Ключевое слово END позволяет задавать команды, которые надо выполнить после окончания обработки данных:

```
$ awk 'BEGIN {print "The File Contents:"}  
{print $0}  
END {print "End of File"}}' myfile
```

*Результаты работы скрипта, в котором имеются блоки BEGIN и END*

После завершения вывода содержимого файла, awk выполняет команды блока END. Это полезная возможность, с её помощью, например, можно сформировать подвал отчёта.

Теперь напишем скрипт следующего содержания и сохраним его в файле myscript:

```
BEGIN {  
  
print "The latest list of users and shells"
```



```

print " UserName \t HomePath"

print "----- \t -----"

FS=":"

}

{

print $1 " \t " $6

}

END {

print "The end"

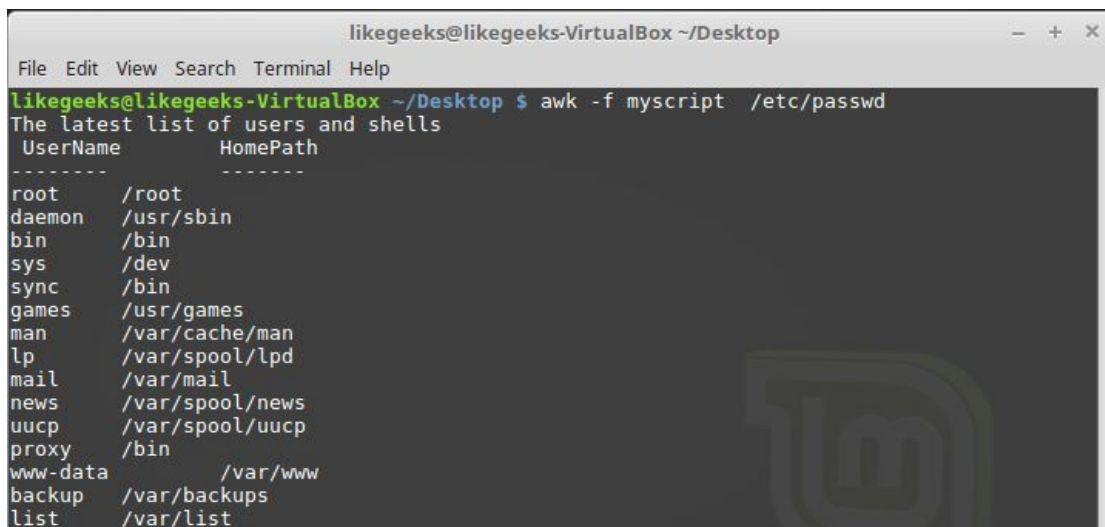
}

```

Тут, в блоке BEGIN, создаётся заголовок табличного отчёта. В этом же разделе мы указываем символ-разделитель. После окончания обработки файла, благодаря блоку END, система сообщит нам о том, что работа окончена.

Запустим скрипт:

```
$ awk -f myscript /etc/passwd
```



```

likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ awk -f myscript /etc/passwd
The latest list of users and shells
UserName      HomePath
-----
root          /root
daemon        /usr/sbin
bin            /bin
sys           /dev
sync          /bin
games         /usr/games
man           /var/cache/man
lp            /var/spool/lpd
mail          /var/mail
news          /var/spool/news
uucp          /var/spool/uucp
proxy         /bin
www-data      /var/www
backup        /var/backups
list          /var/list

```

*Обработка файла /etc/passwd с помощью awk-скрипта*

Всё, о чём мы говорили выше — лишь малая часть возможностей awk. Продолжим освоение этого полезного инструмента.

## ***Встроенные переменные: настройка процесса обработки данных***

Утилита awk использует встроенные переменные, которые позволяют настраивать процесс обработки данных и дают доступ как к обрабатываемым данным, так и к некоторым сведениям о них.

Мы уже рассматривали позиционные переменные — \$1, \$2, \$3, которые позволяют извлекать значения полей, работали мы и с некоторыми другими переменными. На самом деле, их довольно много. Вот некоторые из наиболее часто используемых:

FIELDWIDTHS — разделённый пробелами список чисел, определяющий точную ширину каждого поля данных с учётом разделителей полей.

FS — уже знакомая вам переменная, позволяющая задавать символ-разделитель полей.

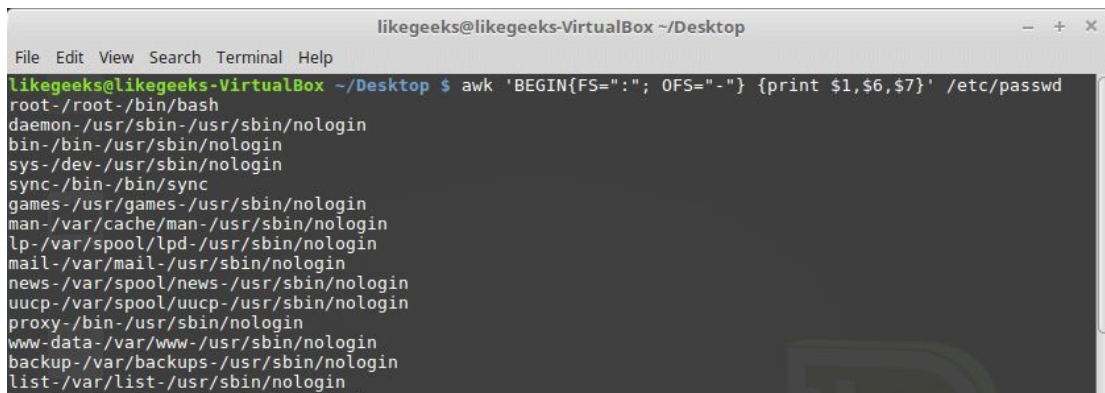
RS — переменная, которая позволяет задавать символ-разделитель записей.

OFS — разделитель полей на выводе awk-скрипта.

ORS — разделитель записей на выводе awk-скрипта.

По умолчанию переменная OFS настроена на использование пробела. Её можно установить так, как нужно для целей вывода данных:

```
$ awk 'BEGIN{FS=":"; OFS="-"} {print $1,$6,$7}' /etc/passwd
```



```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ awk 'BEGIN{FS=":"; OFS="-"} {print $1,$6,$7}' /etc/passwd
root-/root-/bin/bash
daemon-/usr/sbin-/usr/sbin/nologin
bin-/bin-/usr/sbin/nologin
sys-/dev-/usr/sbin/nologin
sync-/bin-/bin/sync
games-/usr/games-/usr/sbin/nologin
man-/var/cache/man-/usr/sbin/nologin
lp-/var/spool/lpd-/usr/sbin/nologin
mail-/var/mail-/usr/sbin/nologin
news-/var/spool/news-/usr/sbin/nologin
uucp-/var/spool/uucp-/usr/sbin/nologin
proxy-/bin-/usr/sbin/nologin
www-data-/var/www-/usr/sbin/nologin
backup-/var/backups-/usr/sbin/nologin
list-/var/list-/usr/sbin/nologin
```

#### Установка разделителя полей выходного потока

Переменная FIELDWIDTHS позволяет читать записи без использования символа-разделителя полей. В некоторых случаях, вместо использования разделителя полей, данные в пределах записей расположены в колонках постоянной ширины. В подобных случаях необходимо задать переменную FIELDWIDTHS таким образом, чтобы её содержимое соответствовало особенностям представления данных.

При установленной переменной FIELDWIDTHS awk будет игнорировать переменную FS и находить поля данных в соответствии со сведениями об их ширине, заданными в FIELDWIDTHS.

Предположим, имеется файл testfile, содержащий такие данные:

```
1235.9652147.91
```

```
927-8.365217.27
```

```
36257.8157492.5
```

Известно, что внутренняя организация этих данных соответствует шаблону 3-5-2-5, то есть, первое поле имеет ширину 3 символа, второе — 5, и так далее. Вот скрипт, который позволит разобрать такие записи:

```
$ awk 'BEGIN{FIELDWIDTHS="3 5 2 5"}{print $1,$2,$3,$4}' testfile
```

## Использование переменной *FIELDWIDTHS*

Посмотрим на то, что выведет скрипт. Данные разобраны с учётом значения переменной *FIELDWIDTHS*, в результате числа и другие символы в строках разбиты в соответствии с заданной шириной полей.

Переменные *RS* и *ORS* задают порядок обработки записей. По умолчанию *RS* и *ORS* установлены на символ перевода строки. Это означает, что *awk* воспринимает каждую новую строку текста как новую запись и выводит каждую запись с новой строки.

Иногда случается так, что поля в потоке данных распределены по нескольким строкам. Например, пусть имеется такой файл с именем *addresses*:

```
Person Name
123 High Street
(222) 466-1234
```

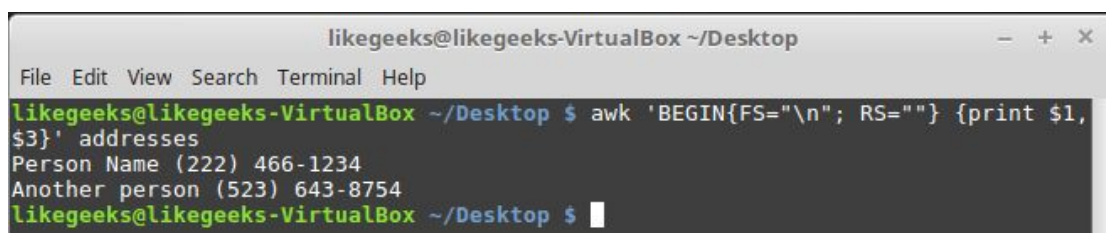
```
Another person
487 High Street
(523) 643-8754
```

Если попытаться прочесть эти данные при условии, что *FS* и *RS* установлены в значения по умолчанию, *awk* сочтёт каждую новую строку отдельной записью и выделит поля, опираясь на пробелы. Это не то, что нам в данном случае нужно.

Для того, чтобы решить эту проблему, в *FS* надо записать символ перевода строки. Это укажет *awk* на то, что каждая строка в потоке данных является отдельным полем.

Кроме того, в данном примере понадобится записать в переменную *RS* пустую строку. Обратите внимание на то, что в файле блоки данных о разных людях разделены пустой строкой. В результате *awk* будет считать пустые строки разделителями записей. Вот как всё это сделать:

```
$ awk 'BEGIN{FS="\n"; RS=""} {print $1,$3}' addresses
```



## Результаты настройки переменных *RS* и *FS*

Как видите, *awk*, благодаря таким настройкам переменных, воспринимает строки из файла как поля, а разделителями записей становятся пустые строки.

## **Встроенные переменные: сведения о данных и об окружении**

Помимо встроенных переменных, о которых мы уже говорили, существуют и другие, которые предоставляют сведения о данных и об окружении, в котором работает *awk*:

ARGC — количество аргументов командной строки.

ARGV — массив с аргументами командной строки.

ARGIND — индекс текущего обрабатываемого файла в массиве ARGV.

ENVIRON — ассоциативный массив с переменными окружения и их значениями.

ERRNO — код системной ошибки, которая может возникнуть при чтении или закрытии входных файлов.

FILENAME — имя входного файла с данными.

FNR — номер текущей записи в файле данных.

IGNORECASE — если эта переменная установлена в ненулевое значение, при обработке игнорируется регистр символов.

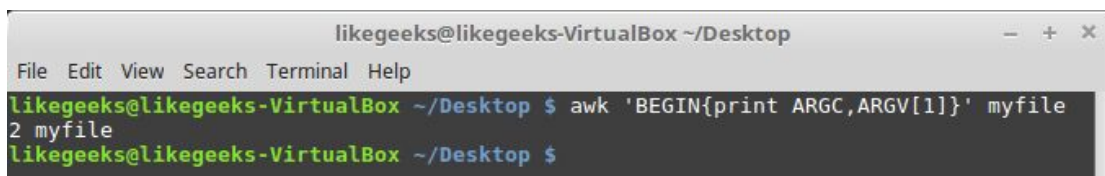
NF — общее число полей данных в текущей записи.

NR — общее число обработанных записей.

Переменные ARGC и ARGV позволяют работать с аргументами командной строки. При этом скрипт, переданный awk, не попадает в массив аргументов ARGV. Напишем такой скрипт:

```
$ awk 'BEGIN{print ARGC,ARGV[1]}' myfile
```

После его запуска можно узнать, что общее число аргументов командной строки — 2, а под индексом 1 в массиве ARGV записано имя обрабатываемого файла. В элементе массива с индексом 0 в данном случае будет «awk».

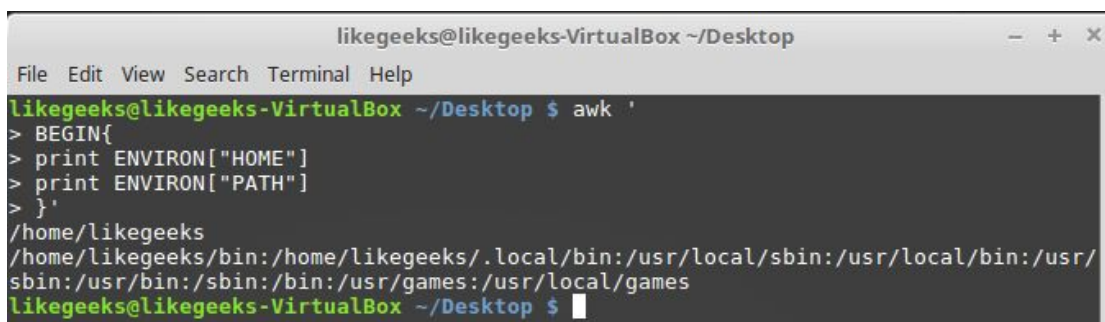


```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ awk 'BEGIN{print ARGC,ARGV[1]}' myfile
2 myfile
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

#### *Работа с параметрами командной строки*

Переменная ENVIRON представляет собой ассоциативный массив с переменными среды. Опробуем её:

```
$ awk '
BEGIN{
print ENVIRON["HOME"]
print ENVIRON["PATH"]
}'
```

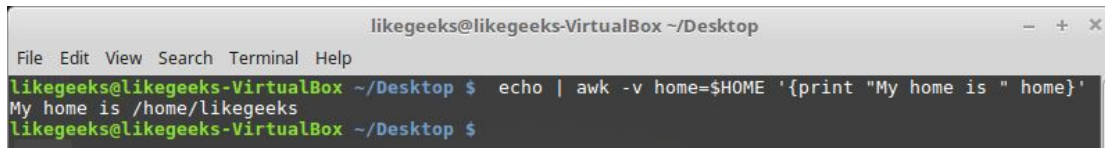


```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ awk '
> BEGIN{
> print ENVIRON["HOME"]
> print ENVIRON["PATH"]
> }'
/home/likegeeks
/home/likegeeks/bin:/home/likegeeks/.local/bin:/usr/local/sbin:/usr/local/bin:/usr/
sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

#### *Работа с переменными среды*

Переменные среды можно использовать и без обращения к ENVIRON. Сделать это, например, можно так:

```
$ echo | awk -v home=$HOME '{print "My home is " home}'
```

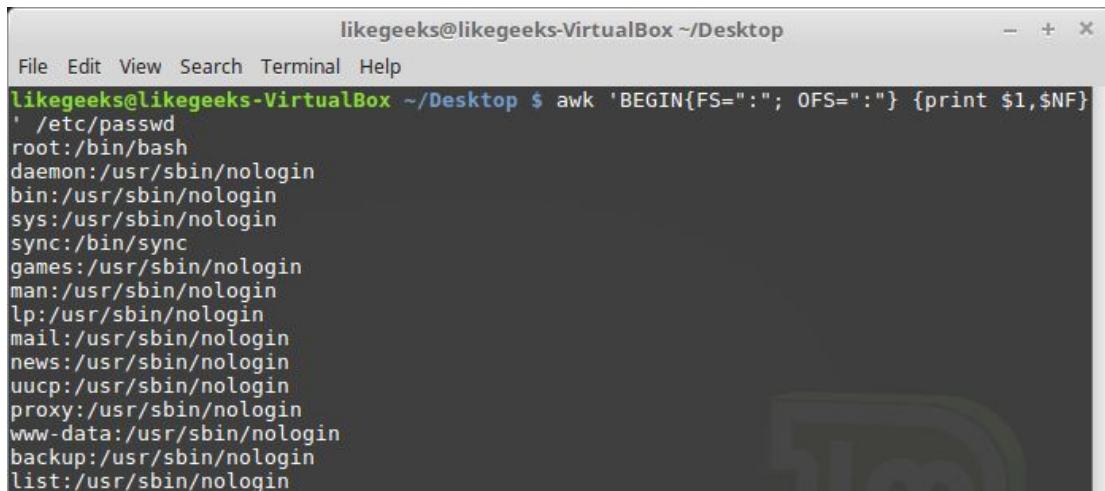


```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ echo | awk -v home=$HOME '{print "My home is " home}'
My home is /home/likegeeks
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

### Работа с переменными среды без использования ENVIRON

Переменная NF позволяет обращаться к последнему полю данных в записи, не зная его точной позиции:

```
$ awk 'BEGIN{FS=":"; OFS=":"} {print $1,$NF}' /etc/passwd
```



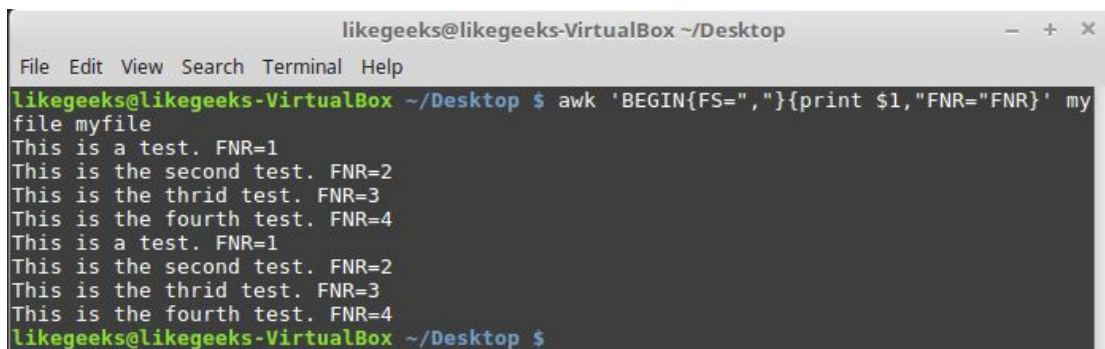
```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ awk 'BEGIN{FS=":"; OFS=":"} {print $1,$NF}' /etc/passwd
root:/bin/bash
daemon:/usr/sbin/nologin
bin:/usr/sbin/nologin
sys:/usr/sbin/nologin
sync:/bin/sync
games:/usr/sbin/nologin
man:/usr/sbin/nologin
lp:/usr/sbin/nologin
mail:/usr/sbin/nologin
news:/usr/sbin/nologin
uucp:/usr/sbin/nologin
proxy:/usr/sbin/nologin
www-data:/usr/sbin/nologin
backup:/usr/sbin/nologin
list:/usr/sbin/nologin
```

### Пример использования переменной NF

Эта переменная содержит числовой индекс последнего поля данных в записи. Обратиться к данному полю можно, поместив перед NF знак \$.

Переменные FNR и NR, хотя и могут показаться похожими, на самом деле различаются. Так, переменная FNR хранит число записей, обработанных в текущем файле. Переменная NR хранит общее число обработанных записей. Рассмотрим пару примеров, передав awk один и тот же файл дважды:

```
$ awk 'BEGIN{FS=","}{print $1,"FNR="FNR}' myfile myfile
```



```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ awk 'BEGIN{FS=","}{print $1,"FNR="FNR}' myfile myfile
This is a test. FNR=1
This is the second test. FNR=2
This is the thrid test. FNR=3
This is the fourth test. FNR=4
This is a test. FNR=1
This is the second test. FNR=2
This is the thrid test. FNR=3
This is the fourth test. FNR=4
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

### Исследование переменной FNR

Передача одного и того же файла дважды равносильна передаче двух разных файлов. Обратите внимание на то, что FNR сбрасывается в начале обработки каждого файла.

Взглянем теперь на то, как ведёт себя в подобной ситуации переменная NR:

```
$ awk '
BEGIN {FS=","}

{print $1,"FNR="FNR,"NR="NR}

END{print "There were",NR,"records processed"}' myfile myfile
```

#### *Различие переменных NR и FNR*

Как видно, FNR, как и в предыдущем примере, сбрасывается в начале обработки каждого файла, а вот NR, при переходе к следующему файлу, сохраняет значение.

### ***Пользовательские переменные***

Как и любые другие языки программирования, awk позволяет программисту объявлять переменные. Имена переменных могут включать в себя буквы, цифры, символы подчёркивания. Однако, они не могут начинаться с цифры. Объявить переменную, присвоить ей значение и воспользоваться ей в коде можно так:

```
$ awk '
BEGIN{
test="This is a test"

print test

}'
```

#### *Работа с пользовательской переменной*

### ***Условный оператор***

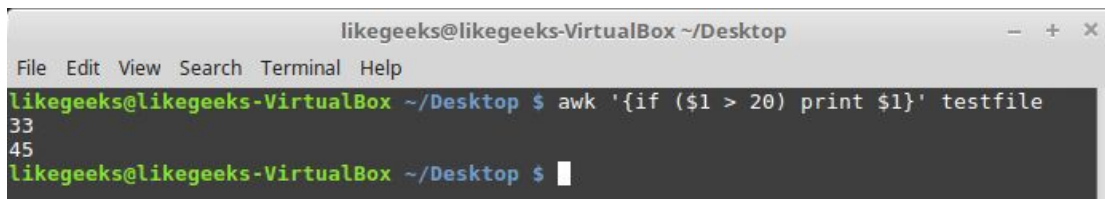
Awk поддерживает стандартный во многих языках программирования формат условного оператора if-then-else. Однострочный вариант оператора представляет собой ключевое слово if, за которым, в скобках, записывают проверяемое выражение, а затем — команду, которую нужно выполнить, если выражение истинно.

Например, есть такой файл с именем testfile:

```
10
15
6
33
45
```

Напишем скрипт, который выводит числа из этого файла, большие 20:

```
$ awk '{if ($1 > 20) print $1}' testfile
```

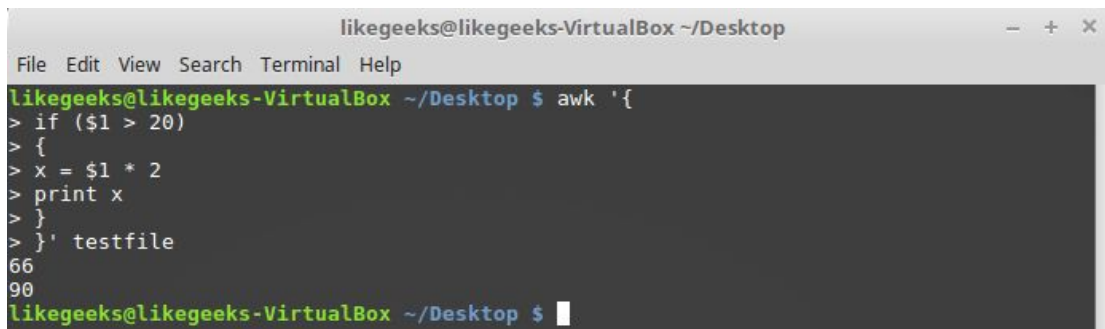


```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ awk '{if ($1 > 20) print $1}' testfile
33
45
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

*Однострочный оператор if*

Если нужно выполнить в блоке if несколько операторов, их нужно заключить в фигурные скобки:

```
$ awk '{
if ($1 > 20)
{
x = $1 * 2
print x
}
}' testfile
```



```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ awk '{
> if ($1 > 20)
> {
> x = $1 * 2
> print x
> }
> }' testfile
66
90
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

*Выполнение нескольких команд в блоке if*

Как уже было сказано, условный оператор awk может содержать блок else:

```
$ awk '{
if ($1 > 20)
{
```



```

x = $1 * 2

print x

} else

{

x = $1 / 2

print x

}}' testfile

```

```

likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ awk '{
> if ($1 > 20)
> {
> x = $1 * 2
> print x
> } else
> {
> x = $1 / 2
> print x
> } }' testfile
5
10
7.5
15
3
1.5
66
33
90
45
likegeeks@likegeeks-VirtualBox ~/Desktop $

```

### Условный оператор с блоком else

Ветвь else может быть частью однострочной записи условного оператора, включая в себя лишь одну строку с командой. В подобном случае после ветви if, сразу перед else, надо поставить точку с запятой:

```
$ awk '{if ($1 > 20) print $1 * 2; else print $1 / 2}' testfile
```

*Условный оператор, содержащий ветви if и else, записанный в одну строку*

## Цикл while

Цикл while позволяет перебирать наборы данных, проверяя условие, которое остановит цикл. Вот файл myfile, обработку которого мы хотим организовать с помощью цикла:

```

124 127 130
112 142 135
175 158 245

```

Напишем такой скрипт:

```
$ awk '{
```



```

total = 0

i = 1

while (i < 4)

{

total += $i

i++

}

avg = total / 3

print "Average:", avg

}' testfile

```

### *Обработка данных в цикле while*

Цикл `while` перебирает поля каждой записи, накапливая их сумму в переменной `total` и увеличивая в каждой итерации на 1 переменную-счётчик `i`. Когда `i` достигнет 4, условие на входе в цикл окажется ложным и цикл завершится, после чего будут выполнены остальные команды — подсчёт среднего значения для числовых полей текущей записи и вывод найденного значения.

В циклах `while` можно использовать команды `break` и `continue`. Первая позволяет досрочно завершить цикл и приступить к выполнению команд, расположенных после него. Вторая позволяет, не завершая до конца текущую итерацию, перейти к следующей.

Вот как работает команда `break`:

```

$ awk '{

total = 0

i = 1

while (i < 4)

{

total += $i

if (i == 2)

```

```

break

i++

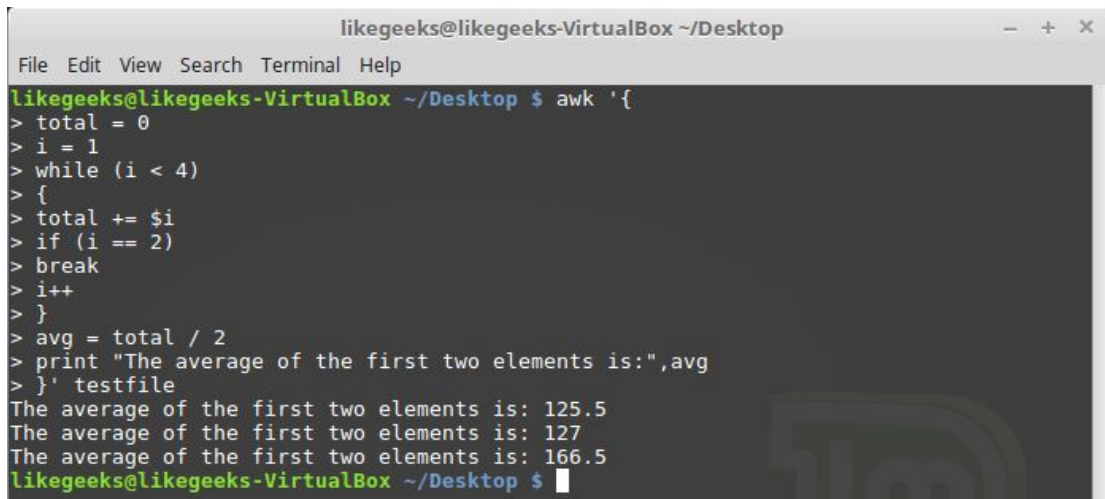
}

avg = total / 2

print "The average of the first two elements is:",avg

}' testfile

```



```

likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ awk '{
> total = 0
> i = 1
> while (i < 4)
> {
> total += $i
> if (i == 2)
> break
> i++
> }
> avg = total / 2
> print "The average of the first two elements is:",avg
> }' testfile
The average of the first two elements is: 125.5
The average of the first two elements is: 127
The average of the first two elements is: 166.5
likegeeks@likegeeks-VirtualBox ~/Desktop $

```

Команда *break* в цикле *while*

## Цикл *for*

Циклы *for* используются во множестве языков программирования. Поддерживает их и *awk*. Решим задачу расчёта среднего значения числовых полей с использованием такого цикла:

```

$ awk '{

total = 0

for (i = 1; i < 4; i++)

{

total += $i

}

avg = total / 3

print "Average:",avg

}' testfile

```

```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ awk '{
> total = 0
> for (i = 1; i < 4; i++)
> {
> total += $i
> }
> avg = total / 3
> print "Average:",avg
> }' testfile
Average: 127
Average: 129.667
Average: 192.667
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

### Цикл for

Начальное значение переменной-счётчика и правило её изменения в каждой итерации, а также условие прекращения цикла, задаются в начале цикла, в круглых скобках. В итоге нам не нужно, в отличие от случая с циклом while, самостоятельно инкрементировать счётчик.

## Форматированный вывод данных

Команда printf в awk позволяет выводить форматированные данные. Она даёт возможность настраивать внешний вид выводимых данных благодаря использованию шаблонов, в которых могут содержаться текстовые данные и спецификаторы форматирования.

Спецификатор форматирования — это специальный символ, который задаёт тип выводимых данных и то, как именно их нужно выводить. Awk использует спецификаторы форматирования как указатели мест вставки данных из переменных, передаваемых printf. Первый спецификатор соответствует первой переменной, второй спецификатор — второй, и так далее.

Спецификаторы форматирования записывают в таком виде:

`%[modifier]control-letter`

Вот некоторые из них:

- c — воспринимает переданное ему число как код ASCII-символа и выводит этот символ.
- d — выводит десятичное целое число.
- i — то же самое, что и d.
- e — выводит число в экспоненциальной форме.
- f — выводит число с плавающей запятой.
- g — выводит число либо в экспоненциальной записи, либо в формате с плавающей запятой, в зависимости от того, как получается короче.
- o — выводит восьмеричное представление числа.
- s — выводит текстовую строку.

Вот как форматировать выводимые данные с помощью printf:

```
$ awk 'BEGIN{
x = 100 * 100
printf "The result is: %e\n", x
}'
```

### Форматирование выходных данных с помощью printf

Тут, в качестве примера, мы выводим число в экспоненциальной записи. Полагаем, этого достаточно для того, чтобы вы поняли основную идею, на которой построена работа с printf.

## Встроенные математические функции

При работе с awk программисту доступны [встроенные функции](#). В частности, это математические и строковые функции, функции для работы со временем. Вот, например, список математических функций, которыми можно пользоваться при разработке awk-скриптов:

cos(x) — косинус x (x выражено в радианах).

sin(x) — синус x.

exp(x) — экспоненциальная функция.

int(x) — возвращает целую часть аргумента.

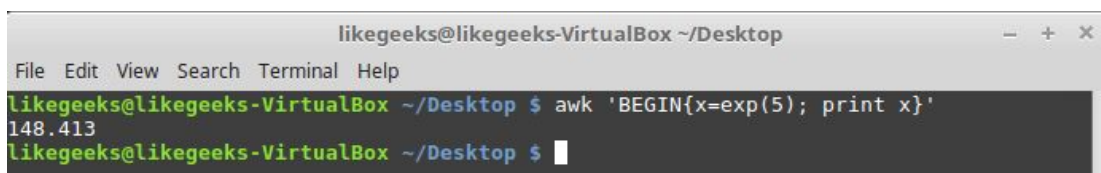
log(x) — натуральный логарифм.

rand() — возвращает случайное число с плавающей запятой в диапазоне 0 - 1.

sqrt(x) — квадратный корень из x.

Вот как пользоваться этими функциями:

```
$ awk 'BEGIN{x=exp(5); print x}'
```



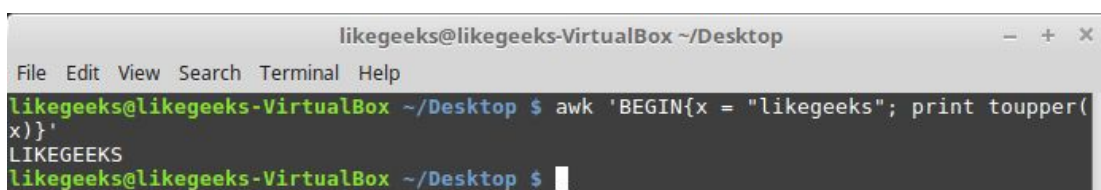
```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ awk 'BEGIN{x=exp(5); print x}'
148.413
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

Работа с математическими функциями

## Строковые функции

Awk поддерживает множество [строковых функций](#). Все они устроены более или менее одинаково. Вот, например, функция toupper:

```
$ awk 'BEGIN{x = "likegeeks"; print toupper(x)}'
```



```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ awk 'BEGIN{x = "likegeeks"; print toupper(x)}'
LIKEGEEKS
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

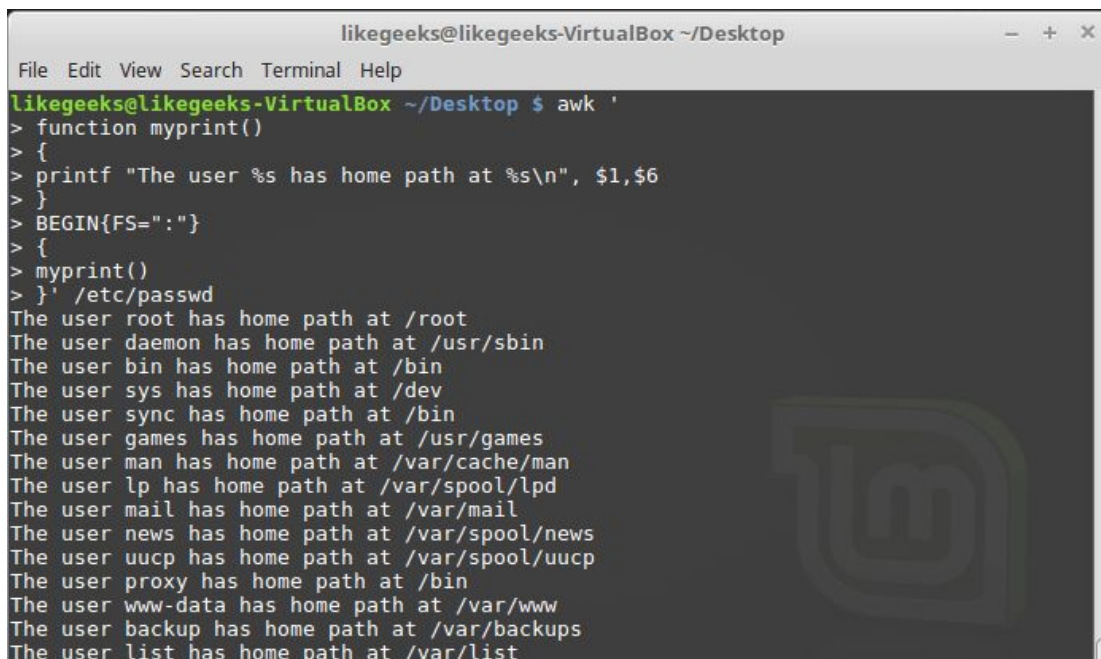
Использование строковой функции toupper

Эта функция преобразует символы, хранящиеся в переданной ей строковой переменной, к верхнему регистру.

## Пользовательские функции

При необходимости вы можете создавать собственные функции awk. Такие функции можно использовать так же, как встроенные:

```
$ awk '  
  
function myprint()  
  
{  
  
printf "The user %s has home path at %s\n", $1,$6  
  
}  
  
BEGIN{FS=":"}  
  
{  
  
myprint()  
  
}' /etc/passwd
```



```
likegeeks@likegeeks-VirtualBox ~/Desktop  
File Edit View Search Terminal Help  
likegeeks@likegeeks-VirtualBox ~/Desktop $ awk '  
> function myprint()  
> {  
> printf "The user %s has home path at %s\n", $1,$6  
> }  
> BEGIN{FS=":"}  
> {  
> myprint()  
> }' /etc/passwd  
The user root has home path at /root  
The user daemon has home path at /usr/sbin  
The user bin has home path at /bin  
The user sys has home path at /dev  
The user sync has home path at /bin  
The user games has home path at /usr/games  
The user man has home path at /var/cache/man  
The user lp has home path at /var/spool/lpd  
The user mail has home path at /var/mail  
The user news has home path at /var/spool/news  
The user uucp has home path at /var/spool/uucp  
The user proxy has home path at /bin  
The user www-data has home path at /var/www  
The user backup has home path at /var/backups  
The user list has home path at /var/list
```

### Использование собственной функции

В примере используется заданная нами функция myprint, которая выводит данные.

## Итоги

Сегодня мы разобрали основы awk. Это мощнейший инструмент обработки данных, масштабы которого сопоставимы с отдельным языком программирования.

Вы не могли не заметить, что многое из того, о чём мы говорим, не так уж и сложно для понимания, а зная основы, уже можно что-то автоматизировать, но если копнуть поглубже, вникнуть в документацию... Вот, например, [The GNU Awk User's Guide](#). В этом руководстве впечатляет уже одно то, что оно ведёт свою историю с 1989-го (первая версия awk, кстати, появилась в 1977-м). Однако, сейчас вы знаете об awk достаточно для того, чтобы не потеряться в официальной документации и познакомиться с ним настолько близко, насколько вам того хочется. В следующий раз, кстати, мы

поговорим о регулярных выражениях. Без них невозможно заниматься серьёзной обработкой текстов в bash-скриптах с применением `sed` и `awk`.

## Bash-скрипты, часть 9: регулярные выражения

Для того, чтобы полноценно обрабатывать тексты в bash-скриптах с помощью `sed` и `awk`, просто необходимо разобраться с регулярными выражениями. Реализации этого полезнейшего инструмента можно найти буквально повсюду, и хотя устроены все регулярные выражения схожим образом, основаны на одних и тех же идеях, в разных средах работа с ними имеет определённые особенности. Тут мы поговорим о регулярных выражениях, которые подходят для использования в сценариях командной строки Linux.

Этот материал задуман как введение в регулярные выражения, рассчитанное на тех, кто может совершенно не знать о том, что это такое. Поэтому начнём с самого начала.

### Что такое регулярные выражения

У многих, когда они впервые видят регулярные выражения, сразу же возникает мысль, что перед ними бессмысленное нагромождение символов. Но это, конечно, далеко не так. Взгляните, например, на это регулярное выражение

```
^([a-zA-Z0-9_-\.\+])@([a-zA-Z0-9_-\.\+])\.([a-zA-Z]{2,5})$
```

На наш взгляд даже абсолютный новичок сходу поймёт, как оно устроено и зачем нужно :-). Если же вам не вполне понятно — просто читайте дальше и всё встанет на свои места.

Регулярное выражение — это шаблон, пользуясь которым программы вроде `sed` или `awk` фильтруют тексты. В шаблонах используются обычные ASCII-символы, представляющие сами себя, и так называемые метасимволы, которые играют особую роль, например, позволяя ссылаться на некие группы символов.

### Типы регулярных выражений

Реализации регулярных выражений в различных средах, например, в языках программирования вроде Java, Perl и Python, в инструментах Linux вроде `sed`, `awk` и `grep`, имеют определённые особенности. Эти особенности зависят от так называемых движков обработки регулярных выражений, которые занимаются интерпретацией шаблонов.

В Linux имеется два движка регулярных выражений:

- Движок, поддерживающий стандарт POSIX Basic Regular Expression (BRE).
- Движок, поддерживающий стандарт POSIX Extended Regular Expression (ERE).

Большинство утилит Linux соответствуют, как минимум, стандарту POSIX BRE, но некоторые утилиты (в их числе — `sed`) понимают лишь некое подмножество стандарта BRE. Одна из причин такого ограничения — стремление сделать такие утилиты как можно более быстрыми в деле обработки текстов.

Стандарт POSIX ERE часто реализуют в языках программирования. Он позволяет пользоваться большим количеством средств при разработке регулярных выражений. Например, это могут быть специальные последовательности символов для часто используемых шаблонов, вроде поиска в тексте отдельных слов или наборов цифр. `Awk` поддерживает стандарт ERE.

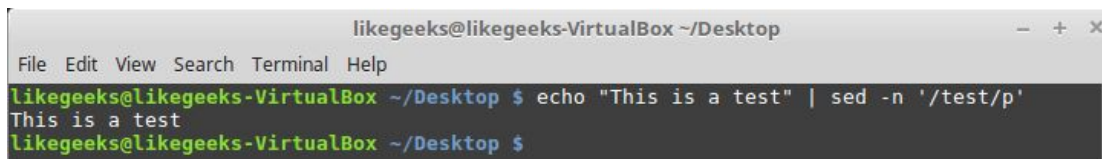
Существует много способов разработки регулярных выражений, зависящих и от мнения программиста, и от особенностей движка, под который их создают. Непросто писать универсальные регулярные выражения, которые сможет понять любой движок. Поэтому мы сосредоточимся на наиболее часто используемых регулярных выражениях и рассмотрим особенности их реализации для sed и awk.

## Регулярные выражения *POSIX BRE*

Пожалуй, самый простой шаблон BRE представляет собой регулярное выражение для поиска точного вхождения последовательности символов в тексте. Вот как выглядит поиск строки в sed и awk:

```
$ echo "This is a test" | sed -n '/test/p'

$ echo "This is a test" | awk '/test/{print $0}'
```

A screenshot of a terminal window titled 'likegeeks@likegeeks-VirtualBox ~/Desktop'. The terminal shows a command prompt where the user enters 'echo "This is a test" | sed -n "/test/p"'. The output of the command is 'This is a test'.

*Поиск текста по шаблону в sed*

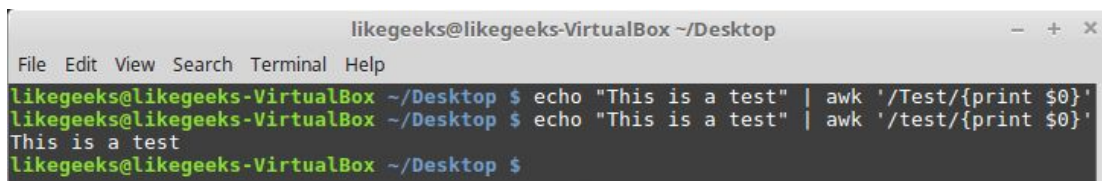
*Поиск текста по шаблону в awk*

Можно заметить, что поиск заданного шаблона выполняется без учёта точного места нахождения текста в строке. Кроме того, не имеет значение и количество вхождений. После того, как регулярное выражение найдёт заданный текст в любом месте строки, строка считается подходящей и передаётся для дальнейшей обработки.

Работая с регулярными выражениями нужно учитывать то, что они чувствительны к регистру символов:

```
$ echo "This is a test" | awk '/Test/{print $0}'

$ echo "This is a test" | awk '/test/{print $0}'
```

A screenshot of a terminal window titled 'likegeeks@likegeeks-VirtualBox ~/Desktop'. It shows two commands being executed. The first command is 'echo "This is a test" | awk "/Test/{print \$0}"', which produces no output. The second command is 'echo "This is a test" | awk "/test/{print \$0}"', which produces the output 'This is a test'.

*Регулярные выражения чувствительны к регистру*

Первое регулярное выражение совпадений не нашло, так как слово «test», начинающееся с заглавной буквы, в тексте не встречается. Второе же, настроенное на поиск слова, написанного прописными буквами, обнаружило в потоке подходящую строку.

В регулярных выражениях можно использовать не только буквы, но и пробелы, и цифры:

```
$ echo "This is a test 2 again" | awk '/test 2/{print $0}'
```



*Поиск фрагмента текста, содержащего пробелы и цифры*

Пробелы воспринимаются движком регулярных выражений как обычные символы.

## **Специальные символы**

При использовании различных символов в регулярных выражениях надо учитывать некоторые особенности. Так, существуют некоторые специальные символы, или метасимволы, использование которых в шаблоне требует особого подхода. Вот они:

```
. * [ ] ^ $ { } \ + ? | ( )
```

Если один из них нужен в шаблоне, его нужно будет экранировать с помощью обратной косой черты (обратного слэша) — \.

Например, если в тексте нужно найти знак доллара, его надо включить в шаблон, предварив символом экранирования. Скажем, имеется файл `myfile` с таким текстом:

```
There is 10$ on my pocket
```

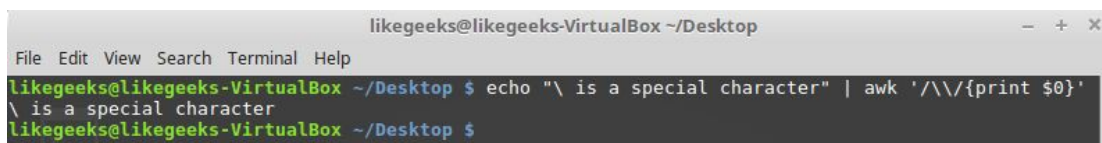
Знак доллара можно обнаружить с помощью такого шаблона:

```
$ awk '/\$/ {print $0}' myfile
```

### *Использование в шаблоне специального символа*

Кроме того, обратная косая черта — это тоже специальный символ, поэтому, если нужно использовать его в шаблоне, его тоже надо будет экранировать. Выглядит это как два слэша, идущих друг за другом:

```
$ echo "\" is a special character" | awk '/\\/ {print $0}'
```



```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ echo "\" is a special character" | awk '/\\/ {print $0}'
\" is a special character
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

### *Экранирование обратного слэша*

Хотя прямой слэш и не входит в приведённый выше список специальных символов, попытка воспользоваться им в регулярном выражении, написанном для `sed` или `awk`, приведёт к ошибке:

```
$ echo "3 / 2" | awk '/// {print $0}'
```

## Неправильное использование прямого слэша в шаблоне

Если он нужен, его тоже надо экранировать:

```
$ echo "3 / 2" | awk '/\//{print $0}'
```

## Экранирование прямого слэша

## Якорные символы

Существуют два специальных символа для привязки шаблона к началу или к концу текстовой строки. Символ «крышка» — ^ позволяет описывать последовательности символов, которые находятся в начале текстовых строк. Если искомый шаблон окажется в другом месте строки, регулярное выражение на него не отреагирует. Выглядит использование этого символа так:

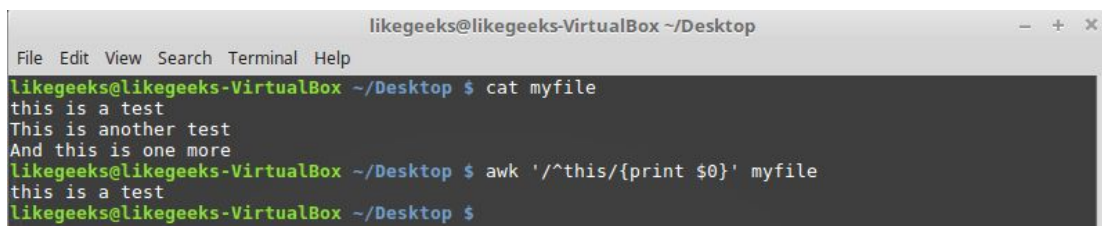
```
$ echo "welcome to likegeeks website" | awk '/^likegeeks/{print $0}'
```

```
$ echo "likegeeks website" | awk '/^likegeeks/{print $0}'
```

## Поиск шаблона в начале строки

Символ ^ предназначен для поиска шаблона в начале строки, при этом регистр символов так же учитывается. Посмотрим, как это отразится на обработке текстового файла:

```
$ awk '/^this/{print $0}' myfile
```

A screenshot of a terminal window titled 'likegeeks@likegeeks-VirtualBox ~/Desktop'. The terminal shows the following commands and output:

```
likegeeks@likegeeks-VirtualBox ~/Desktop $ cat myfile
this is a test
This is another test
And this is one more
likegeeks@likegeeks-VirtualBox ~/Desktop $ awk '/^this/{print $0}' myfile
this is a test
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

## Поиск шаблона в начале строки в тексте из файла

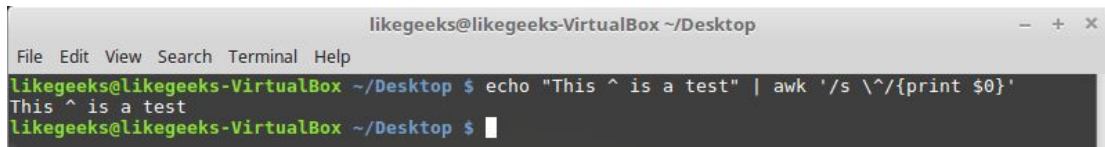
При использовании sed, если поместить крышку где-нибудь внутри шаблона, она будет восприниматься как любой другой обычный символ:

```
$ echo "This ^ is a test" | sed -n '/s ^/p'
```

## Крышка, находящаяся не в начале шаблона в sed

В awk, при использовании такого же шаблона, данный символ надо экранировать:

```
$ echo "This ^ is a test" | awk '/s \^/{print $0}'
```



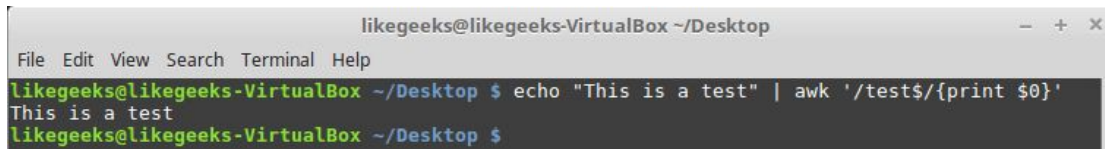
```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ echo "This ^ is a test" | awk '/s \^/{print $0}'
This ^ is a test
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

*Крышка, находящаяся не в начале шаблона в awk*

С поиском фрагментов текста, находящихся в начале строки мы разобрались. Что, если надо найти нечто, расположенное в конце строки?

В этом нам поможет знак доллара — \$, являющийся якорным символом конца строки:

```
$ echo "This is a test" | awk '/test$/{print $0}'
```

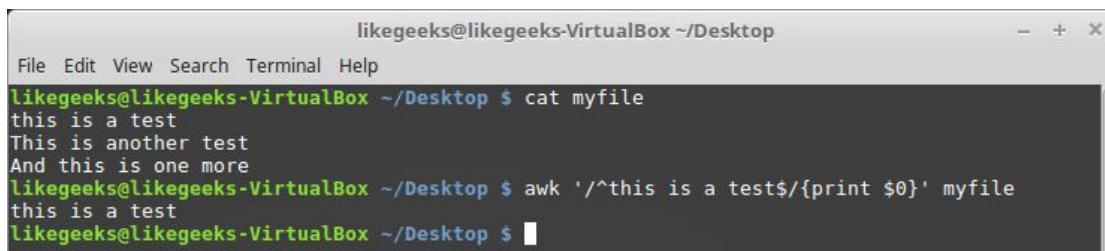


```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ echo "This is a test" | awk '/test$/{print $0}'
This is a test
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

*Поиск текста, находящегося в конце строки*

В одном и том же шаблоне можно использовать оба якорных символа. Выполним обработку файла myfile, содержимое которого показано на рисунке ниже, с помощью такого регулярного выражения:

```
$ awk '/^this is a test$/{print $0}' myfile
```



```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ cat myfile
this is a test
This is another test
And this is one more
likegeeks@likegeeks-VirtualBox ~/Desktop $ awk '/^this is a test$/{print $0}' myfile
this is a test
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

*Шаблон, в котором использованы специальные символы начала и конца строки*

Как видно, шаблон среагировал лишь на строку, полностью соответствующую заданной последовательности символов и их расположению.

Вот как, пользуясь якорными символами, отфильтровать пустые строки:

```
$ awk '!/^$/{print $0}' myfile
```

В данном шаблоне использовал символ отрицания, восклицательный знак — !. Благодаря использованию такого шаблона выполняется поиск строк, не содержащих ничего между началом и концом строки, а благодаря восклицательному знаку на печать выводятся лишь строки, которые не соответствуют этому шаблону.

## Символ «точка»

Точка используется для поиска любого одиночного символа, за исключением символа перевода строки. Передадим такому регулярному выражению файл myfile, содержимое которого приведено ниже:

```
$ awk '/.st/{print $0}' myfile
```

```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ cat myfile
this is a test
This is another test
And this is one more
start with this
likegeeks@likegeeks-VirtualBox ~/Desktop $ awk '/.st/{print $0}' myfile
this is a test
This is another test
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

*Использование точки в регулярных выражениях*

Как видно по выведенным данным, шаблону соответствуют лишь первые две строки из файла, так как они содержат последовательность символов «st», предварённую ещё одним символом, в то время как третья строка подходящей последовательности не содержит, а в четвёртой она есть, но находится в самом начале строки.

## Классы символов

Точка соответствует любому одиночному символу, но что если нужно более гибко ограничить набор искомых символов? В подобной ситуации можно воспользоваться классами символов.

Благодаря такому подходу можно организовать поиск любого символа из заданного набора. Для описания класса символов используются квадратные скобки — []:

```
$ awk '/[oi]th/{print $0}' myfile
```

```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ cat myfile
this is a test
This is another test
And this is one more
start with this
likegeeks@likegeeks-VirtualBox ~/Desktop $ awk '/[oi]th/{print $0}' myfile
This is another test
start with this
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

*Описание класса символов в регулярном выражении*

Тут мы ищем последовательность символов «th», перед которой есть символ «o» или символ «i». Классы оказываются очень кстати, если выполняется поиск слов, которые могут начинаться как с прописной, так и со строчной буквы:

```
$ echo "this is a test" | awk '/[Tt]his is a test/{print $0}'
```

```
$ echo "This is a test" | awk '/[Tt]his is a test/{print $0}'
```

```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ echo "this is a test" | awk '/[Tt]his is a test/{print $0}'
this is a test
likegeeks@likegeeks-VirtualBox ~/Desktop $ echo "This is a test" | awk '/[Tt]his is a test/{print $0}'
This is a test
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

*Поиск слов, которые могут начинаться со строчной или прописной буквы*

Классы символов не ограничены буквами. Тут можно использовать и другие символы. Нельзя заранее сказать, в какой ситуации понадобятся классы — всё зависит от решаемой задачи.

## Отрицание классов символов

Классы символов можно использовать и для решения задачи, обратной описанной выше. А именно, вместо поиска символов, входящих в класс, можно организовать поиск всего, что в класс не входит. Для того, чтобы добиться такого поведения регулярного выражения, перед списком символов класса нужно поместить знак ^. Выглядит это так:

```
$ awk '/[^oi]th/{print $0}' myfile
```

#### *Поиск символов, не входящих в класс*

В данном случае будут найдены последовательности символов «th», перед которыми нет ни «o», ни «i».

## **Диапазоны символов**

В символьных классах можно описывать диапазоны символов, используя тире:

```
$ awk '/[e-p]st/{print $0}' myfile
```

#### *Описание диапазона символов в символьном классе*

В данном примере регулярное выражение реагирует на последовательность символов «st», перед которой находится любой символ, расположенный, в алфавитном порядке, между символами «e» и «p».

Диапазоны можно создавать и из чисел:

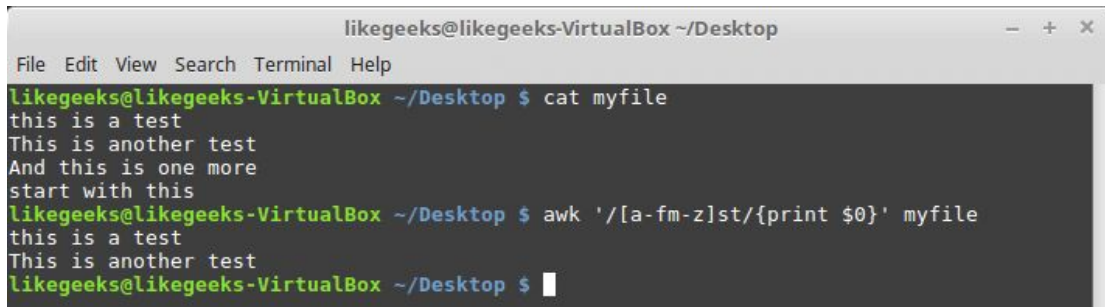
```
$ echo "123" | awk '/[0-9][0-9][0-9]/'
```

```
$ echo "12a" | awk '/[0-9][0-9][0-9]/'
```

#### *Регулярное выражение для поиска трёх любых чисел*

В класс символов могут входить несколько диапазонов:

```
$ awk '/[a-fm-z]st/{print $0}' myfile
```



```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ cat myfile
this is a test
This is another test
And this is one more
start with this
likegeeks@likegeeks-VirtualBox ~/Desktop $ awk '/[a-fm-z]st/{print $0}' myfile
this is a test
This is another test
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

*Класс символов, состоящий из нескольких диапазонов*

Данное регулярное выражение найдёт все последовательности «st», перед которыми есть символы из диапазонов a-f и m-z.

## Специальные классы символов

В BRE имеются специальные классы символов, которые можно использовать при написании регулярных выражений:

- `[:alpha:]` — соответствует любому алфавитному символу, записанному в верхнем или нижнем регистре.
- `[:alnum:]` — соответствует любому алфавитно-цифровому символу, а именно — символам в диапазонах 0-9, A-Z, a-z.
- `[:blank:]` — соответствует пробелу и знаку табуляции.
- `[:digit:]` — любой цифровой символ от 0 до 9.
- `[:upper:]` — алфавитные символы в верхнем регистре — A-Z.
- `[:lower:]` — алфавитные символы в нижнем регистре — a-z.
- `[:print:]` — соответствует любому печатаемому символу.
- `[:punct:]` — соответствует знакам препинания.
- `[:space:]` — пробельные символы, в частности — пробел, знак табуляции, символы NL, FF, VT, CR.

Использовать специальные классы в шаблонах можно так:

```
$ echo "abc" | awk '/[:alpha:]/{print $0}'
$ echo "abc" | awk '/[:digit:]/{print $0}'
$ echo "abc123" | awk '/[:digit:]/{print $0}'
```

*Специальные классы символов в регулярных выражениях*

## Символ «звёздочка»

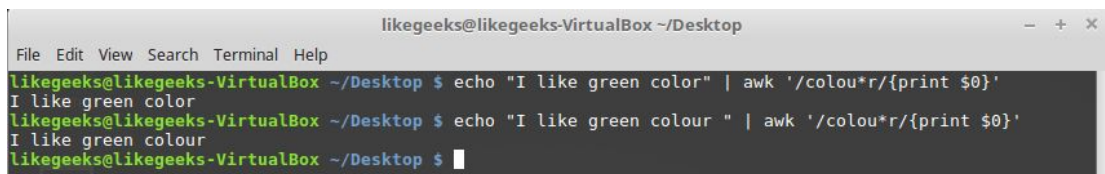
Если в шаблоне после символа поместить звёздочку, это будет означать, что регулярное выражение сработает, если символ появляется в строке любое количество раз — включая и ситуацию, когда символ в строке отсутствует.

```
$ echo "test" | awk '/tes*t/{print $0}'  
$ echo "tessst" | awk '/tes*t/{print $0}'
```

### *Использование символа \* в регулярных выражениях*

Этот шаблонный символ обычно используют для работы со словами, в которых постоянно встречаются опечатки, или для слов, допускающих разные варианты корректного написания:

```
$ echo "I like green color" | awk '/colou*r/{print $0}'  
$ echo "I like green colour " | awk '/colou*r/{print $0}'
```



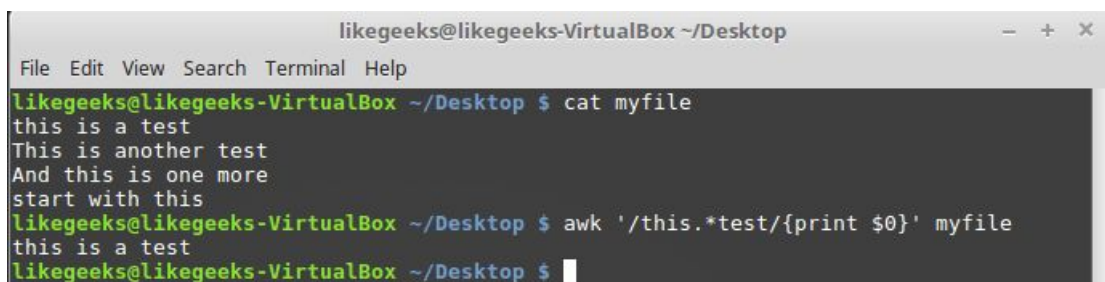
```
likegeeks@likegeeks-VirtualBox ~/Desktop  
File Edit View Search Terminal Help  
likegeeks@likegeeks-VirtualBox ~/Desktop $ echo "I like green color" | awk '/colou*r/{print $0}'  
I like green color  
likegeeks@likegeeks-VirtualBox ~/Desktop $ echo "I like green colour " | awk '/colou*r/{print $0}'  
I like green colour  
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

### *Поиск слова, имеющего разные варианты написания*

В этом примере одно и то же регулярное выражение реагирует и на слово «color», и на слово «colour». Это так благодаря тому, что символ «и», после которого стоит звёздочка, может либо отсутствовать, либо встречаться несколько раз подряд.

Ещё одна полезная возможность, вытекающая из особенностей символа звёздочки, заключается в комбинировании его с точкой. Такая комбинация позволяет регулярному выражению реагировать на любое количество любых символов:

```
$ awk '/this.*test/{print $0}' myfile
```



```
likegeeks@likegeeks-VirtualBox ~/Desktop  
File Edit View Search Terminal Help  
likegeeks@likegeeks-VirtualBox ~/Desktop $ cat myfile  
this is a test  
This is another test  
And this is one more  
start with this  
likegeeks@likegeeks-VirtualBox ~/Desktop $ awk '/this.*test/{print $0}' myfile  
this is a test  
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

### *Шаблон, реагирующий на любое количество любых символов*

В данном случае неважно сколько и каких символов находится между словами «this» и «test». Звёздочку можно использовать и с классами символов:

```
$ echo "st" | awk '/s[ae]*t/{print $0}'  
$ echo "sat" | awk '/s[ae]*t/{print $0}'  
$ echo "set" | awk '/s[ae]*t/{print $0}'
```



```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ echo "st" | awk '/s[ae]*t/{print $0}'
st
likegeeks@likegeeks-VirtualBox ~/Desktop $ echo "sat" | awk '/s[ae]*t/{print $0}'
sat
likegeeks@likegeeks-VirtualBox ~/Desktop $ echo "set" | awk '/s[ae]*t/{print $0}'
set
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

### *Использование звёздочки с классами символов*

Во всех трёх примерах регулярное выражение срабатывает, так как звёздочка после класса символов означает, что если будет найдено любое количество символов «а» или «е», а также если их найти не удастся, строка будет соответствовать заданному шаблону.

## **Регулярные выражения POSIX ERE**

Шаблоны стандарта POSIX ERE, которые поддерживают некоторые утилиты Linux, могут содержать дополнительные символы. Как уже было сказано, awk поддерживает этот стандарт, а вот sed — нет.

Тут мы рассмотрим наиболее часто используемые в ERE-шаблонах символы, которые пригодятся вам при создании собственных регулярных выражений.

## **Вопросительный знак**

Вопросительный знак указывает на то, что предшествующий символ может встретиться в тексте один раз или не встретиться вовсе. Этот символ — один из метасимволов повторений. Вот несколько примеров:

```
$ echo "tet" | awk '/tes?t/{print $0}'
tet
$ echo "test" | awk '/tes?t/{print $0}'
test
$ echo "tesst" | awk '/tes?t/{print $0}'
tesst
```

### *Вопросительный знак в регулярных выражениях*

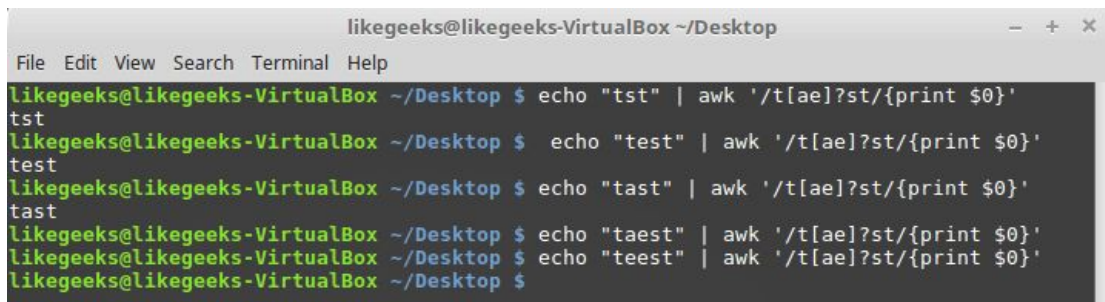
Как видно, в третьем случае буква «s» встречается дважды, поэтому на слово «tesst» регулярное выражение не реагирует.

Вопросительный знак можно использовать и с классами символов:

```
$ echo "tst" | awk '/t[ae]?st/{print $0}'
tst
$ echo "test" | awk '/t[ae]?st/{print $0}'
test
$ echo "tast" | awk '/t[ae]?st/{print $0}'
tast
$ echo "taest" | awk '/t[ae]?st/{print $0}'
taest
```



```
$ echo "teest" | awk '/t[ae]?st/{print $0}'
```



```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ echo "tst" | awk '/t[ae]?st/{print $0}'
tst
likegeeks@likegeeks-VirtualBox ~/Desktop $ echo "test" | awk '/t[ae]?st/{print $0}'
test
likegeeks@likegeeks-VirtualBox ~/Desktop $ echo "tast" | awk '/t[ae]?st/{print $0}'
tast
likegeeks@likegeeks-VirtualBox ~/Desktop $ echo "taest" | awk '/t[ae]?st/{print $0}'
likegeeks@likegeeks-VirtualBox ~/Desktop $ echo "teest" | awk '/t[ae]?st/{print $0}'
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

### Вопросительный знак и классы символов

Если символов из класса в строке нет, или один из них встречается один раз, регулярное выражение срабатывает, однако стоит в слове появиться двум символам и система уже не находит в тексте соответствия шаблону.

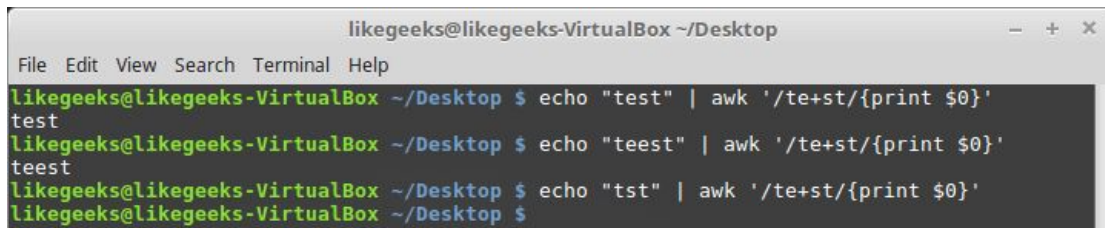
## Символ «плюс»

Символ «плюс» в шаблоне указывает на то, что регулярное выражение обнаружит искомое в том случае, если предшествующий символ встретится в тексте один или более раз. При этом на отсутствие символа такая конструкция реагировать не будет:

```
$ echo "test" | awk '/te+st/{print $0}'
```

```
$ echo "teest" | awk '/te+st/{print $0}'
```

```
$ echo "tst" | awk '/te+st/{print $0}'
```



```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ echo "test" | awk '/te+st/{print $0}'
test
likegeeks@likegeeks-VirtualBox ~/Desktop $ echo "teest" | awk '/te+st/{print $0}'
teest
likegeeks@likegeeks-VirtualBox ~/Desktop $ echo "tst" | awk '/te+st/{print $0}'
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

### Символ «плюс» в регулярных выражениях

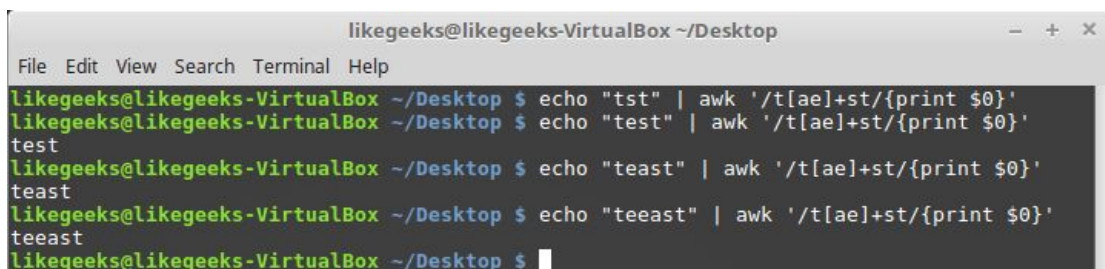
В данном примере, если символа «е» в слове нет, движок регулярных выражений не найдёт в тексте соответствий шаблону. Символ «плюс» работает и с классами символов — этим он похож на звёздочку и вопросительный знак:

```
$ echo "tst" | awk '/t[ae]+st/{print $0}'
```

```
$ echo "test" | awk '/t[ae]+st/{print $0}'
```

```
$ echo "teast" | awk '/t[ae]+st/{print $0}'
```

```
$ echo "teeast" | awk '/t[ae]+st/{print $0}'
```



```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ echo "tst" | awk '/t[ae]+st/{print $0}'
likegeeks@likegeeks-VirtualBox ~/Desktop $ echo "test" | awk '/t[ae]+st/{print $0}'
test
likegeeks@likegeeks-VirtualBox ~/Desktop $ echo "teast" | awk '/t[ae]+st/{print $0}'
teast
likegeeks@likegeeks-VirtualBox ~/Desktop $ echo "teeast" | awk '/t[ae]+st/{print $0}'
teeast
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

### Знак «плюс» и классы символов

В данном случае если в строке имеется любой символ из класса, текст будет сочтён соответствующим шаблону.

## Фигурные скобки

Фигурные скобки, которыми можно пользоваться в ERE-шаблонах, похожи на символы, рассмотренные выше, но они позволяют точнее задавать необходимое число вхождений предшествующего им символа.

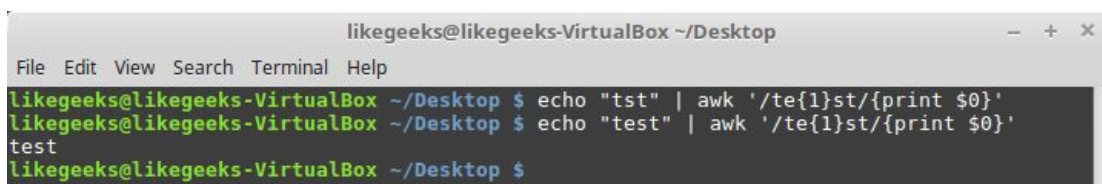
Указывать ограничение можно в двух форматах:

- $n$  — число, задающее точное число искомых вхождений
- $n, m$  — два числа, которые трактуются так: «как минимум  $n$  раз, но не больше чем  $m$ ».

Вот примеры первого варианта:

```
$ echo "tst" | awk '/te{1}st/{print $0}'
```

```
$ echo "test" | awk '/te{1}st/{print $0}'
```



```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ echo "tst" | awk '/te{1}st/{print $0}'
likegeeks@likegeeks-VirtualBox ~/Desktop $ echo "test" | awk '/te{1}st/{print $0}'
test
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

*Фигурные скобки в шаблонах, поиск точного числа вхождений*

В старых версиях awk нужно было использовать ключ командной строки `--re-interval` для того, чтобы программа распознавала интервалы в регулярных выражениях, но в новых версиях этого делать не нужно.

```
$ echo "tst" | awk '/te{1,2}st/{print $0}'
```

```
$ echo "test" | awk '/te{1,2}st/{print $0}'
```

```
$ echo "teest" | awk '/te{1,2}st/{print $0}'
```

```
$ echo "teeest" | awk '/te{1,2}st/{print $0}'
```

*Интервал, заданный в фигурных скобках*

В данном примере символ «e» должен встретиться в строке 1 или 2 раза, тогда регулярное выражение отреагирует на текст.

Фигурные скобки можно применять и с классами символов. Тут действуют уже знакомые вам принципы:

```
$ echo "tst" | awk '/t[ae]{1,2}st/{print $0}'
```

```
$ echo "test" | awk '/t[ae]{1,2}st/{print $0}'  
$ echo "teest" | awk '/t[ae]{1,2}st/{print $0}'  
$ echo "teeast" | awk '/t[ae]{1,2}st/{print $0}'
```

### *Фигурные скобки и классы символов*

Шаблон отреагирует на текст в том случае, если в нём один или два раза встретится символ «а» или символ «е».

## **Символ логического «или»**

Символ | — вертикальная черта, означает в регулярных выражениях логическое «или». Обработывая регулярное выражение, содержащее несколько фрагментов, разделённых таким знаком, движок сочтёт анализируемый текст подходящим в том случае, если он будет соответствовать любому из фрагментов. Вот пример:

```
$ echo "This is a test" | awk '/test|exam/{print $0}'  
$ echo "This is an exam" | awk '/test|exam/{print $0}'  
$ echo "This is something else" | awk '/test|exam/{print $0}'
```

### *Логическое «или» в регулярных выражениях*

В данном примере регулярное выражение настроено на поиск в тексте слов «test» или «exam». Обратите внимание на то, что между фрагментами шаблона и разделяющим их символом | не должно быть пробелов.

## **Группировка фрагментов регулярных выражений**

Фрагменты регулярных выражений можно группировать, пользуясь круглыми скобками. Если сгруппировать некую последовательность символов, она будет восприниматься системой как обычный символ. То есть, например, к ней можно будет применить метасимволы повторений. Вот как это выглядит:

```
$ echo "Like" | awk '/Like(Geeks)?/{print $0}'  
$ echo "LikeGeeks" | awk '/Like(Geeks)?/{print $0}'
```

### *Группировка фрагментов регулярных выражений*

В данных примерах слово «Geeks» заключено в круглые скобки, после этой конструкции идёт знак вопроса. Напомним, что вопросительный знак означает «0 или 1 повторение», в результате регулярное выражение отреагирует и на строку «Like», и на строку «LikeGeeks».

## **Практические примеры**

После того, как мы разобрали основы регулярных выражений, пришло время сделать с их помощью что-нибудь полезное.

### **Подсчёт количества файлов**

Напишем `bash`-скрипт, который подсчитывает файлы, находящиеся в директориях, которые записаны в переменную окружения `PATH`. Для того, чтобы это сделать, понадобится, для начала, сформировать список путей к директориям. Сделаем это с помощью `sed`, заменив двоеточия на пробелы:

```
$ echo $PATH | sed 's:/:/g'
```

Команда замены поддерживает регулярные выражения в качестве шаблонов для поиска текста. В данном случае всё предельно просто, ищем мы символ двоеточия, но никто не мешает использовать здесь и что-нибудь другое — всё зависит от конкретной задачи.

Теперь надо пройти по полученному списку в цикле и выполнить там необходимые для подсчёта количества файлов действия. Общая схема скрипта будет такой:

```
mypath=$(echo $PATH | sed 's:/:/g')
```

```
for directory in $mypath
```

```
do
```

```
done
```

Теперь напишем полный текст скрипта, воспользовавшись командой `ls` для получения сведений о количестве файлов в каждой из директорий:

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

```
mypath=$(echo $PATH | sed 's:/:/g')
```

```
count=0
```

```
for directory in $mypath
```

```
do
```

```
check=$(ls $directory)
```

```

for item in $check
do

count=$(( $count + 1 ))

done

echo "$directory - $count"

count=0

done

```

При запуске скрипта может оказаться, что некоторых директорий из PATH не существует, однако, это не мешает ему посчитать файлы в существующих директориях.

### *Подсчёт файлов*

Главная ценность этого примера заключается в том, что пользуясь тем же подходом, можно решать и куда более сложные задачи. Какие именно — зависит от ваших потребностей.

## **Проверка адресов электронной почты**

Существуют веб-сайты с огромными коллекциями регулярных выражений, которые позволяют проверять адреса электронной почты, телефонные номера, и так далее. Однако, одно дело — взять готовое, и совсем другое — создать что-то самому. Поэтому напомним регулярное выражение для проверки адресов электронной почты. Начнём с анализа исходных данных. Вот, например, некий адрес:

```
username@hostname.com
```

Имя пользователя, username, может состоять из алфавитно-цифровых и некоторых других символов. А именно, это точка, тире, символ подчёркивания, знак «плюс». За именем пользователя следует знак @. Вооружившись этими знаниями, начнём сборку регулярного выражения с его левой части, которая служит для проверки имени пользователя. Вот что у нас получилось:

```
^([a-zA-Z0-9_-\.\+]+)@
```

Это регулярное выражение можно прочитать так: «В начале строки должен быть как минимум один символ из тех, которые имеются в группе, заданной в квадратных скобках, а после этого должен идти знак @».

Теперь — очередь имени хоста — hostname. Тут применимы те же правила, что и для имени пользователя, поэтому шаблон для него будет выглядеть так:

```
([a-zA-Z0-9_-\.\.]+)
```

Имя домена верхнего уровня подчиняется особым правилам. Тут могут быть лишь алфавитные символы, которых должно быть не меньше двух (например, такие домены обычно содержат код страны), и не больше пяти. Всё это значит, что шаблон для проверки последней части адреса будет таким:

```
\.([a-zA-Z]{2,5})$
```

Прочесть его можно так: «Сначала должна быть точка, потом — от 2 до 5 алфавитных символов, а после этого строка заканчивается».

Подготовив шаблоны для отдельных частей регулярного выражения, соберём их вместе:

```
^([a-zA-Z0-9_-\.\.]+)@([a-zA-Z0-9_-\.\.]+)\.([a-zA-Z]{2,5})$
```

Теперь осталось лишь протестировать то, что получилось:

```
$ echo "name@host.com" | awk  
'/^([a-zA-Z0-9_-\.\.]+)@([a-zA-Z0-9_-\.\.]+)\.([a-zA-Z]{2,5})$/{print $0}'
```

```
$ echo "name@host.com.us" | awk  
'/^([a-zA-Z0-9_-\.\.]+)@([a-zA-Z0-9_-\.\.]+)\.([a-zA-Z]{2,5})$/{print $0}'
```

### *Проверка адреса электронной почты с помощью регулярных выражений*

То, что переданный awk текст выводится на экран, означает, что система распознала в нём адрес электронной почты.

## **Итоги**

Если регулярное выражение для проверки адресов электронной почты, которое встретилось вам в самом начале статьи, казалось тогда совершенно непонятным, надеемся, сейчас оно уже не выглядит бессмысленным набором символов. Если это действительно так — значит данный материал выполнил своё предназначение. На самом деле, регулярные выражения — это тема, которой можно заниматься всю жизнь, но даже то небольшое, что мы разобрали, уже способно помочь вам в написании скриптов, которые довольно продвинуто обрабатывают тексты.

В этой серии материалов мы обычно показывали очень простые примеры bash-скриптов, которые состояли буквально из нескольких строк. В следующий раз рассмотрим кое-что более масштабное.

## Bash-скрипты, часть 10: практические примеры

В предыдущих материалах мы обсуждали различные аспекты разработки bash-скриптов, говорили о полезных инструментах, но до сих пор рассматривали лишь небольшие фрагменты кода. Пришло время более масштабных проектов. А именно, здесь вы найдёте два примера. Первый — скрипт для отправки сообщений, второй пример — скрипт, выводящий сведения об использовании дискового пространства.

Главная ценность этих примеров для тех, кто изучает bash, заключается в методике разработки. Когда перед программистом встаёт задача по автоматизации чего бы то ни было, его путь редко бывает прямым и быстрым. Задачу надо разбить на части, найти средства решения каждой из подзадач, а потом собрать из частей готовое решение.

### *Отправка сообщений в терминал пользователя*

В наши дни редко кто прибегает к одной из возможностей Linux, которая позволяет общаться, отправляя сообщения в терминалы пользователей, вошедших в систему. Сама по себе команда отправки сообщений, `write`, довольно проста. Для того, чтобы ей воспользоваться, достаточно знать имя пользователя и имя его терминала. Однако, для успешной отправки сообщения, помимо актуальных данных о пользователе и терминале, надо знать, вошёл ли пользователь в систему, не запретил ли он запись в свой терминал. В результате, перед отправкой сообщения нужно выполнить несколько проверок.

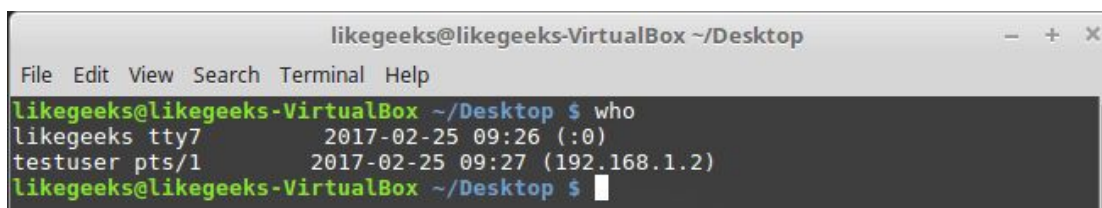
Как видите, задача: «отправить сообщение», при ближайшем рассмотрении, оказалась задачей: «проверить возможность отправки сообщения, и, если нет препятствий, отправить его». Займёмся решением задачи, то есть — разработкой bash-скрипта.

### Команды `who` и `mesg`

Ядром скрипта являются несколько команд, которые мы ещё не обсуждали. Всё остальное должно быть вам знакомо по предыдущим материалам.

Первое, что нам тут понадобится — команда `who`. Она позволяет узнать сведения о пользователях, работающих в системе. В простейшем виде её вызов выглядит так:

```
$ who
```



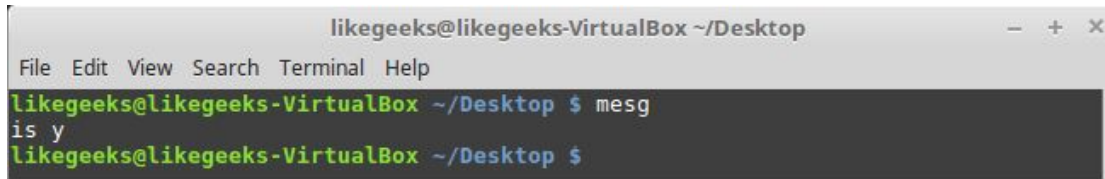
```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ who
likegeeks tty7      2017-02-25 09:26 (:0)
testuser pts/1     2017-02-25 09:27 (192.168.1.2)
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

*Результаты вызова команды `who`*

В каждой строке, которую выводит команда `who`, нас интересуют первых два показателя — имя пользователя и сведения о его терминале.

По умолчанию запись в терминал разрешена, но пользователь может, с помощью команды `mesg`, запретить отправку ему сообщений. Таким образом, прежде чем пытаться что-то кому-то отправить, неплохо будет проверить, разрешена ли отправка сообщений. Если нужно узнать собственный статус, достаточно ввести команду `mesg` без параметров:

```
$ mesg
```



```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ mesg
is y
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

#### Команда *mesg*

В данном случае команда вывела «is y», это значит, что пользователь, под которым мы работаем в системе, может принимать сообщения, отправленные в его терминал. В противном случае *mesg* выведет «is n».

Для проверки того, разрешена ли отправка сообщений какому-то другому пользователю, можно использовать уже знакомую вам команду *who* с ключом *-T*:

```
$ who -T
```

При этом проверка возможна только для пользователей, которые вошли в систему. Если такая команда, после имени пользователя, выведет чёрточку (-), это означает, что пользователь запретил запись в свой терминал, то есть, сообщения ему отправлять нельзя. О том, что пользователю можно отправлять сообщения, говорит знак «плюс» (+).

Если у вас приём сообщений отключён, а вы хотите позволить другим пользователям отправлять вам сообщения, можно воспользоваться такой командой:

```
$ mesg y
```

#### *Включение приёма сообщений от других пользователей*

После включения приёма сообщений *mesg* возвращает «is y».

Конечно, для обмена сообщениями нужны два пользователя, поэтому мы, после обычного входа в систему, подключились к компьютеру по *ssh*. Теперь можно поэкспериментировать.

### Команда **write**

Основной инструмент для обмена сообщениями между пользователями, вошедшими в систему — команда *write*. Если приём сообщений у пользователя разрешён, с помощью этой команды ему можно отправлять сообщения, используя его имя и сведения о терминале.

Обратите внимание на то, что с помощью *write* можно отправлять сообщения пользователям, вошедшим в виртуальную консоль. Пользователи, которые работают в графическом окружении (KDE, Gnome, Cinnamon, и так далее), не могут получать подобные сообщения.

Итак, мы, работая под пользователем *likegeeks*, иницилируем сеанс связи с пользователем *testuser*, который работает в терминале *pts/1*, следующим образом:



```
$ write testuser pts/1
```

### *Проверка возможности отправки сообщений и отправка сообщения*

После выполнения вышеуказанной команды перед нами окажется пустая строка, в которую нужно ввести первую строку сообщения. Нажав клавишу ENTER, мы можем ввести следующую строку сообщения. После того, как ввод текста завершён, окончить сеанс связи можно, воспользовавшись комбинацией клавиш CTRL + D, которая позволяет ввести [символ конца файла](#).

Вот что увидит в своём терминале пользователь, которому мы отправили сообщение.

### *Новое сообщение, пришедшее в терминал*

Получатель может понять от кого пришло сообщение, увидеть время, когда оно было отправлено. Обратите внимание на признак конца файла, EOF, расположенный в нижней части окна терминала. Он указывает на окончание текста сообщения.

Полагаем, теперь у нас есть всё необходимое для того, чтобы автоматизировать отправку сообщений с помощью сценария командной строки.

## **Создание скрипта для отправки сообщений**

Прежде чем заниматься отправкой сообщений, нужно определить, вошёл ли интересующий нас пользователь в систему. Сделать это можно с помощью такой команды:

```
logged_on=$(who | grep -i -m 1 $1 | awk '{print $1}')
```

Здесь результаты работы команды who передаются команде grep. Ключ -i этой команды позволяет игнорировать регистр символов. Ключ -m 1 включён в вызов команды на тот случай, если пользователь вошёл в систему несколько раз. Эта команда либо не выведет ничего, либо выведет имя пользователя (его мы укажем при вызове скрипта, оно попадёт в позиционную переменную \$1), соответствующее первому найденному сеансу. Вывод grep мы передаём awk. Эта команда, опять же, либо не выведет ничего, либо выведет элемент, записанный в собственную переменную \$1, то есть — имя пользователя. В итоге то, что получилось, попадает в переменную logged\_on.

Теперь надо проверить переменную logged\_on, посмотреть, есть ли в ней что-нибудь:

```
if [ -z $logged_on ]  
then  
  
echo "$1 is not logged on."
```

```
echo "Exit"

exit

fi
```

Если вы не вполне уверенно чувствуете себя, работая с конструкцией if, взгляните на [ЭТОТ](#) материал. Скрипт, содержащий вышеописанный код, сохраним в файле senderscript и вызовем, передав ему, в качестве параметра командной строки, имя пользователя testuser.

### *Проверка статуса пользователя*

Тут мы проверяем, является ли logged\_on переменной с нулевой длиной. Если это так, нам сообщат о том, что в данный момент пользователь в систему не вошёл и скрипт завершит работу с помощью команды exit. В противном случае выполнение скрипта продолжится.

## Проверка возможности записи в терминал пользователя

Теперь надо проверить, принимает ли пользователь сообщения. Для этого понадобится такая конструкция, похожая на ту, которую мы использовали выше:

```
allowed=$(who -T | grep -i -m 1 $1 | awk '{print $2}')

if [ $allowed != "+" ]

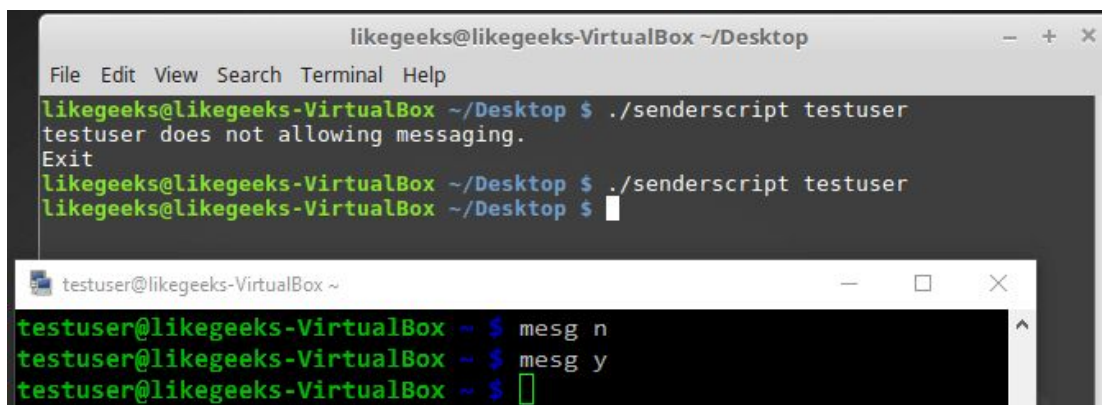
then

echo "$1 does not allowing messaging."

echo "Exit"

exit

fi
```



```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./senderscript testuser
testuser does not allowing messaging.
Exit
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./senderscript testuser
likegeeks@likegeeks-VirtualBox ~/Desktop $

testuser@likegeeks-VirtualBox ~
testuser@likegeeks-VirtualBox ~ $ mesg n
testuser@likegeeks-VirtualBox ~ $ mesg y
testuser@likegeeks-VirtualBox ~ $
```

### *Проверка возможности отправки сообщений пользователю*

Сначала мы вызываем команду who с ключом -T. В строке сведений о пользователе, который может принимать сообщения, окажется знак «плюс» (+), если же пользователь принимать сообщения не

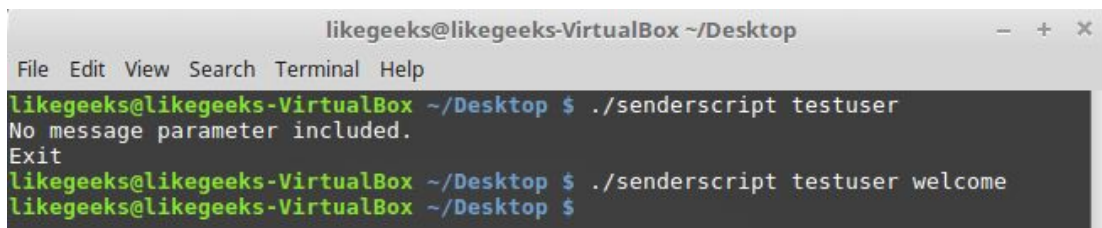
может — там будет чёрточка (-). То, что получилось после вызова `who`, передаётся `grep`, а потом — `awk`, формируя переменную `allowed`.

Далее, используя условный оператор, мы проверяем то, что оказалось в переменной `allowed`. Если знака «плюс» в ней нет, сообщим о том, что отправка сообщений пользователю запрещена и завершим работу. В противном случае выполнение сценария продолжится.

## Проверка правильности вызова скрипта

Первым параметром скрипта является имя пользователя, которому мы хотим отправить сообщение. Вторым — текст сообщения, в данном случае — состоящий из одного слова. Для того, чтобы проверить, передано ли скрипту сообщение для отправки, воспользуемся таким кодом:

```
if [ -z $2 ]  
  
then  
  
echo "No message parameter included."  
  
echo "Exit"  
  
exit  
  
fi
```



```
likegeeks@likegeeks-VirtualBox ~/Desktop  
File Edit View Search Terminal Help  
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./senderscript testuser  
No message parameter included.  
Exit  
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./senderscript testuser welcome  
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

*Проверка параметров командной строки, указанных при вызове скрипта*

Тут, если при вызове скрипта ему не было передано сообщение для отправки, мы сообщаем об этом и завершаем работу. В противном случае — идём дальше.

## Получение сведений о термине пользователя

Прежде чем отправить пользователю сообщение, нужно получить сведения о термине, в котором он работает и сохранить имя терминала в переменной. Делается это так:

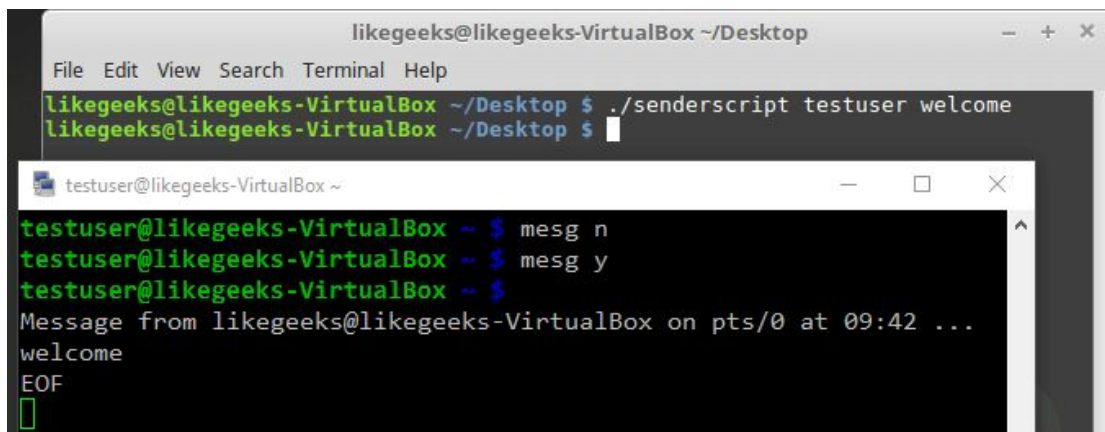
```
terminal=$(who | grep -i -m 1 $1 | awk '{print $2}')
```

Теперь, после того, как все необходимые данные собраны, осталось лишь отправить сообщение:

```
echo $2 | write $logged_on $terminal
```

Вызов готового скрипта выглядит так:

```
$ ./senderscript testuser welcome
```

A screenshot of a terminal window titled 'likegeeks@likegeeks-VirtualBox ~/Desktop'. The terminal shows the command './senderscript testuser welcome' being executed. Below it, a second terminal window titled 'testuser@likegeeks-VirtualBox ~' is shown, displaying the output of the script: 'Message from likegeeks@likegeeks-VirtualBox on pts/0 at 09:42 ...', 'welcome', and 'EOF'.

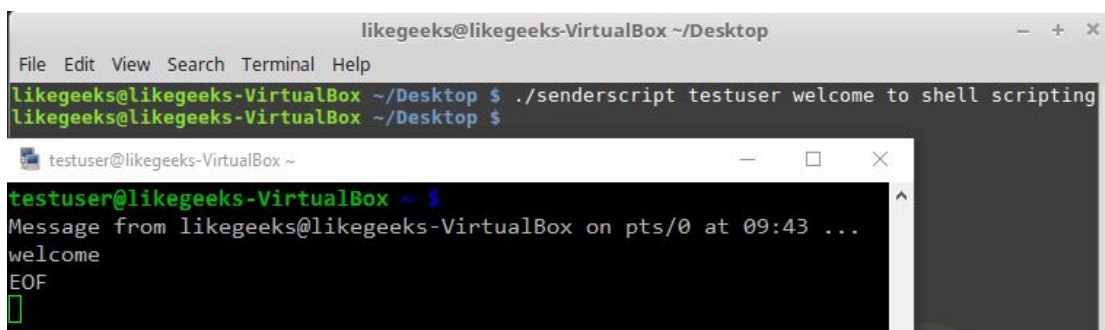
Успешная отправка сообщения с помощью bash-скрипта

Как видно, всё работает как надо. Однако, с помощью такого сценария можно отправлять лишь сообщения, состоящие из одного слова. Хорошо бы получить возможность отправлять более длинные сообщения.

## Отправка длинных сообщений

Попробуем вызвать сценарий senderscript, передав ему сообщение, состоящее из нескольких слов:

```
$ ./senderscript likegeeks welcome to shell scripting
```

A screenshot of a terminal window titled 'likegeeks@likegeeks-VirtualBox ~/Desktop'. The terminal shows the command './senderscript testuser welcome to shell scripting' being executed. Below it, a second terminal window titled 'testuser@likegeeks-VirtualBox ~' is shown, displaying the output of the script: 'Message from likegeeks@likegeeks-VirtualBox on pts/0 at 09:43 ...', 'welcome', and 'EOF'. The message 'welcome to shell scripting' was truncated to just 'welcome'.

Попытка отправки длинного сообщения

Как видно, отправлено было лишь первое слово. Всё дело в том, что каждое слово сообщения воспринимается внутри скрипта как отдельная позиционная переменная. Для того, чтобы получить возможность отправки длинных сообщений, обработаем параметры командной строки, переданные сценарию, воспользовавшись [командой](#) shift и циклом while.

```
shift
while [ -n "$1" ]
do
whole_message=$whole_message' '$1
shift
done
```

После этого, в команде отправки сообщения, воспользуемся, вместо применяемой ранее позиционной переменной \$2, переменной whole\_message:

```
echo $whole_message | write $logged_on $terminal
```

**Вот полный текст сценария:**

```
#!/bin/bash

logged_on=$(who | grep -i -m 1 $1 | awk '{print $1}')

if [ -z $logged_on ]

then

echo "$1 is not logged on."

echo "Exit"

exit

fi

allowed=$(who -T | grep -i -m 1 $1 | awk '{print $2}')

if [ $allowed != "+" ]

then

echo "$1 does not allowing messaging."

echo "Exit"

exit

fi

if [ -z $2 ]

then

echo "No message parameter included."

echo "Exit"

exit

fi

terminal=$(who | grep -i -m 1 $1 | awk '{print $2}')

shift

while [ -n "$1" ]

do

whole_message=$whole_message' '$1

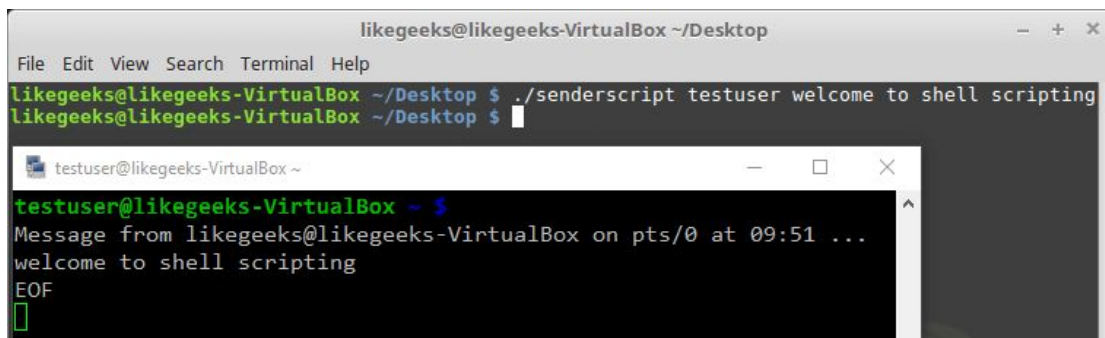
shift

done

echo $whole_message | write $logged_on $terminal
```

**Испытаем его:**

```
$ ./senderscript likegeeks welcome to shell scripting
```



Успешная отправка длинного сообщения:

Длинное сообщение успешно дошло до адресата.

Теперь рассмотрим следующий пример.

## Скрипт для мониторинга дискового пространства

Сейчас мы собираемся создать сценарий командной строки, который предназначен для поиска в заданных директориях первой десятки папок, на которые приходится больше всего дискового пространства. В этом нам поможет [команда](#) `du`, которая выводит сведения о том, сколько места на диске занимают файлы и папки. По умолчанию она выводит сведения лишь о директориях, с ключом `-a` в отчёт попадают и отдельные файлы. Её ключ `-s` позволяет вывести сведения о размерах директорий. Эта команда позволяет, например, узнать объём дискового пространства, который занимают данные некоего пользователя. Вот как выглядит вызов этой команды:

```
$ du -s /var/log/
```

Для наших целей лучше подойдёт ключ `-S` (заглавная S), так как он позволяет получить сведения как по корневой папке, так и по вложенным в неё директориям:

```
$ du -S /var/log/
```

*Вызов команды `du` с ключами `-s` и `-S`*

Нам нужно найти директории, на которые приходится больше всего дискового пространства, поэтому список, который выдаёт `du`, надо отсортировать, воспользовавшись командой `sort`:

```
$ du -S /var/log/ | sort -rn
```

```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ du -S /var/log/ | sort -rn
5184    /var/log/
828     /var/log/installer
68      /var/log/ConsoleKit
60      /var/log/apt
36      /var/log/cups
24      /var/log/mdm
12      /var/log/fsck
4       /var/log/upstart
4       /var/log/samba
4       /var/log/ntpstats
4       /var/log/hp/tmp
4       /var/log/hp
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

### Отсортированный список объектов

Ключ `-n` указывает команде на то, что нужна числовая сортировка, ключ `-r` — на обратный порядок сортировки (самое большое число окажется в начале списка). Полученные данные вполне подходят для наших целей.

Для того, чтобы ограничить полученный список первыми десятью записями, воспользуемся [ПОТОКОВЫМ редактором](#) `sed`, который позволит удалить из полученного списка все строки, начиная с одиннадцатой. Следующий шаг — добавить к каждой полученной строке её номер. Тут также поможет `sed`, а именно — его команда `N`:

```
sed '{11,$D; =}' |
```

```
sed 'N; s/\n/ /' |
```

Приведём полученные данные в порядок, воспользовавшись `awk`. Передадим `awk` то, что получилось после обработки данных с помощью `sed`, применив, как и в других случаях, конвейер, и выведем полученные данные с помощью команды `printf`:

```
awk '{printf $1 ":" "\t" $2 "\t" $3 "\n"}'
```

В начале строки выводится её номер, потом идёт двоеточие и знак табуляции, далее — объём дискового пространства, следом — ещё один знак табуляции и имя папки. Соберём вместе всё то, о чём мы говорили:

```
$ du -S /var/log/ |
```

```
sort -rn |
```

```
sed '{11,$D; =}' |
```

```
sed 'N; s/\n/ /' |
```

```
awk '{printf $1 ":" "\t" $2 "\t" $3 "\n"}'
```

```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ du -S /var/log/ |
> sort -rn |
> sed '{11,$D; =}' |
> sed 'N; s/\n/ /' |
> awk '{printf $1 ":" "\t" $2 "\t" $3 "\n"}'
1: 5184 /var/log/
2: 828 /var/log/installer
3: 68 /var/log/ConsoleKit
4: 60 /var/log/apt
5: 36 /var/log/cups
6: 24 /var/log/mdm
7: 12 /var/log/fsck
8: 4 /var/log/upstart
9: 4 /var/log/samba
10: 4 /var/log/ntpstats
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

### Вывод сведений о дисковом пространстве

Для того, чтобы повысить эффективность работы скрипта, код которого вы совсем скоро увидите, реализуем возможность получения данных сразу по нескольким директориям. Для этого создадим переменную MY\_DIRECTORIES и внесём в неё список интересующих нас директорий:

```
MY_DIRECTORIES="/home /var/log"
```

Переберём список с помощью цикла for и вызовем вышеописанную последовательность команд для каждого элемента списка. Вот что получилось в результате:

```
#!/bin/bash

MY_DIRECTORIES="/home /var/log"

echo "Top Ten Disk Space Usage"

for DIR in $MY_DIRECTORIES
do

echo "The $DIR Directory:"

du -S $DIR 2>/dev/null |

sort -rn |

sed '{11,$D; =}' |

sed 'N; s/\n/ /' |

awk '{printf $1 ":" "\t" $2 "\t" $3 "\n"}'

done

exit
```



### *Получение сведений о нескольких директориях*

Как видите, скрипт выводит, в виде удобного списка, сведения о директориях, список которых хранится в MY\_DIRECTORIES.

Команду `du` в этом скрипте можно вызвать с другими ключами, полученный список объектов вполне можно отфильтровать, в целом — тут открывается широкий простор для самостоятельных экспериментов. В результате, вместо работы со списком папок, можно, например, найти самые большие файлы с расширением `.log`, или реализовать более сложный алгоритм поиска самых больших (или самых маленьких) файлов и папок.

## **Итоги**

Сегодня мы подробно разобрали пару примеров разработки скриптов. Тут хотелось бы напомнить, что наша главная цель — не в том, чтобы написать скрипт для отправки сообщений с помощью команды `write`, или сценарий, который помогает в поиске файлов и папок, занимающих много места на диске, а в описании самого процесса разработки. Освоив эти примеры, поэкспериментировав с ними, возможно — дополнив их или полностью переработав, вы научитесь чему-то новому, улучшите свои навыки разработки `bash`-скриптов.

На сегодня это всё. В следующий раз поговорим об автоматизации работы с интерактивными утилитами с помощью `exrc`.

# Bash-скрипты, часть 11: expect и автоматизация интерактивных утилит

В прошлый раз мы говорили о методике разработки bash-скриптов. Если же суммировать всё, что мы разобрали в предыдущих десяти материалах, то вы, если начинали читать их, ничего не зная о bash, теперь можете сделать уже довольно много всего полезного.

Сегодняшняя тема, заключительная в этой серии материалов, посвящена автоматизации работы с интерактивными утилитами, например, со скриптами, которые, в процессе выполнения, взаимодействуют с пользователем. В этом деле нам поможет expect — инструмент, основанный на языке Tcl.

Expect позволяет создавать программы, ожидающие вопросов от других программ и дающие им ответы. Expect можно сравнить с роботом, который способен заменить пользователя при взаимодействии со сценариями командной строки.

## Основы expect

Если expect в вашей системе не установлен, исправить это, например, в Ubuntu, можно так:

```
$ apt-get install expect
```

В чём-то вроде CentOS установка выполняется такой командой:

```
$ yum install expect
```

Expect предоставляет набор команд, позволяющих взаимодействовать с утилитами командной строки. Вот его основные команды:

- spawn — запуск процесса или программы. Например, это может быть командная оболочка, [FTP](#), Telnet, ssh, scp и так далее.
- expect — ожидание данных, выводимых программой. При написании скрипта можно указать, какого именно вывода он ждёт и как на него нужно реагировать.
- send — отправка ответа. Expect-скрипт с помощью этой команды может отправлять входные данные автоматизируемой программе. Она похожа на знакомую вам команду echo в обычных bash-скриптах.
- interact — позволяет переключиться на «ручной» режим управления программой.

## Автоматизация bash-скрипта

Напишем скрипт, который взаимодействует с пользователем и автоматизируем его с помощью expect. Вот код bash-скрипта questions:

```
#!/bin/bash

echo "Hello, who are you?"

read $REPLY

echo "Can I ask you some questions?"

read $REPLY
```

```
echo "What is your favorite topic?"
```

```
read $REPLY
```

Теперь напишем expect-скрипт, который запустит скрипт questions и будет отвечать на его вопросы:

```
#!/usr/bin/expect -f

set timeout -1

spawn ./questions

expect "Hello, who are you?\r"

send -- "Im Adam\r"

expect "Can I ask you some questions?\r"

send -- "Sure\r"

expect "What is your favorite topic?\r"

send -- "Technology\r"

expect eof
```

Сохраним скрипт, дав ему имя answerbot.

В начале скрипта находится строка идентификации, которая, в данном случае, содержит путь к expect, так как интерпретировать скрипт будет именно expect.

Во второй строке мы отключаем тайм-аут, устанавливая переменную expect timeout в значение -1. Остальной код — это и есть автоматизация работы с bash-скриптом.

Сначала, с помощью команды spawn, мы запускаем bash-скрипт. Естественно, тут может быть вызвана любая другая утилита командной строки. Далее задана последовательность вопросов, поступающих от bash-скрипта, и ответов, которые даёт на них expect. Получив вопрос от подпроцесса, expect выдаёт ему заданный ответ и ожидает следующего вопроса.

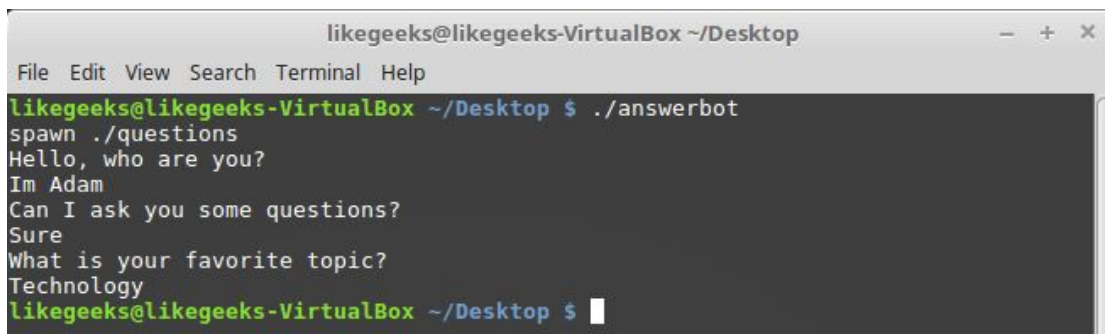
В последней команде expect ожидает признака конца файла, скрипт, дойдя до этой команды, завершается.

Теперь пришло время всё это опробовать. Сделаем answerbot исполняемым файлом:

```
$ chmod +x ./answerbot
```

И вызовем его:

```
$ ./answerbot
```



```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./answerbot
spawn ./questions
Hello, who are you?
Im Adam
Can I ask you some questions?
Sure
What is your favorite topic?
Technology
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

*Ехрест-скрипт отвечает на вопросы bash-скрипта*

Как видно, ехрест-скрипт верно ответил на вопросы bash-скрипта. Если на данном этапе вы столкнулись с ошибкой, вызванной тем, что неправильно указано расположение ехрест, выяснить его адрес можно так:

```
$ which expect
```

Обратите внимание на то, что после запуска скрипта answerbot всё происходит в полностью автоматическом режиме. То же самое можно проделать для любой утилиты командной строки. Тут надо отметить, что наш bash-скрипт устроен очень просто, мы точно знаем, какие именно данные он выводит, поэтому написать ехрест-скрипт для взаимодействия с ним несложно. Задача усложняется при работе с программами, которые написаны другими разработчиками. Однако, здесь на помощь приходит средство для автоматизированного создания ехрест-скриптов.

## ***Autoexpect — автоматизированное создание ехрест-скриптов***

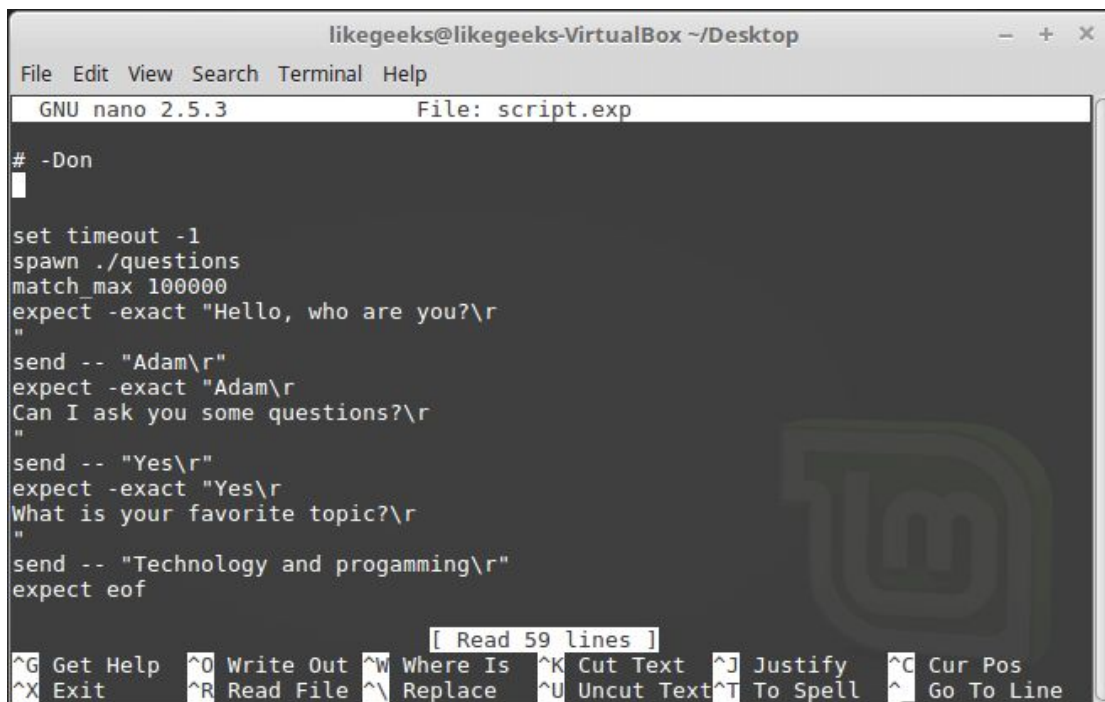
Autoexpect позволяет запускать программы, которые надо автоматизировать, после чего записывает то, что они выводят, и то, что пользователь вводит, отвечая на их вопросы. Вызовем autoexpect, передав этой утилите имя нашего скрипта:

```
$ autoexpect ./questions
```

В этом режиме взаимодействие с bash-скриптом ничем не отличается от обычного: мы сами вводим ответы на его вопросы.

### *Запуск bash-скрипта с помощью autoexpect*

После завершения работы с bash-скриптом, autoexpect сообщит о том, что собранные данные записаны в файл script.exp. Взглянем на этот файл.



```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
GNU nano 2.5.3 File: script.exp

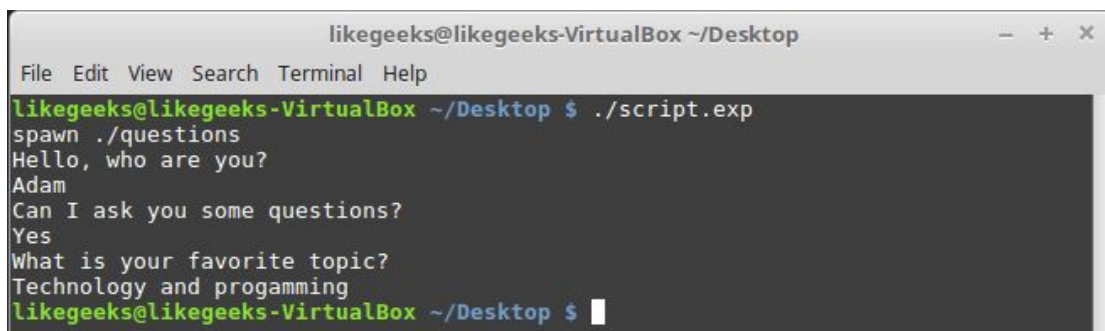
# -Don

set timeout -1
spawn ./questions
match_max 100000
expect -exact "Hello, who are you?\r"
"
send -- "Adam\r"
expect -exact "Adam\r"
Can I ask you some questions?\r
"
send -- "Yes\r"
expect -exact "Yes\r"
What is your favorite topic?\r
"
send -- "Technology and programming\r"
expect eof

[ Read 59 lines ]
^G Get Help ^O Write Out ^W Where Is ^K Cut Text ^J Justify ^C Cur Pos
^X Exit ^R Read File ^\ Replace ^U Uncut Text ^T To Spell ^_ Go To Line
```

Файл script.exp

В целом, за исключением некоторых деталей, перед нами такой же скрипт, который мы писали самостоятельно. Если запустить этот скрипт, результат будет тем же.



```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./script.exp
spawn ./questions
Hello, who are you?
Adam
Can I ask you some questions?
Yes
What is your favorite topic?
Technology and programming
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

Запуск expect-скрипта, созданного автоматически

При записи сеансов взаимодействия с некоторыми программами, вроде FTP-клиентов, вы можете столкнуться с тем, что они используют в выводимых данных сведения о времени проведения операции, или выводят данные, отражающие процесс выполнения неких продолжительных действий. В целом, речь идёт о том, что вывод программы при каждом её запуске, правильно воспринимаемый человеком и вызывающий ввод одних и тех же ответов, будет, в тех же условиях, выглядеть по-новому для expect. Если в expect-скрипте строки, ожидаемые от такой программы, будут жёстко зафиксированы, такой скрипт не сможет нормально работать. Справиться с этим можно, либо удалив из expect-скрипта данные, которые выглядят по-новому при каждом запуске программы, либо используя шаблоны, пользуясь которыми, expect сможет правильно понять то, что хочет от него программа.

Как видите, autoexpect — это весьма полезный инструмент, но и он не лишён недостатков, исправить которые можно только вручную. Поэтому продолжим осваивать язык expect-скриптов.

## Работа с переменными и параметрами командной строки

Для объявления переменных в expect-скриптах используется команда set. Например, для того, чтобы присвоить значение 5 переменной VAR1, используется следующая конструкция:

```
set VAR1 5
```

Для доступа к значению переменной перед её именем надо добавить знак доллара — \$. В нашем случае это будет выглядеть как \$VAR1.

Для того, чтобы получить доступ к аргументам командной строки, с которыми вызван expect-скрипт, можно поступить так:

```
set VAR [lindex $argv 0]
```

Тут мы объявляем переменную VAR и записываем в неё указатель на первый аргумент командной строки, \$argv 0.

Для целей обновлённого expect-скрипта мы собираемся записать значение первого аргумента, представляющее собой имя пользователя, которое будет использовано в программе, в переменную my\_name. Второй аргумент, символизирующий то, что пользователю нравится, попадёт в переменную my\_favorite. В результате объявление переменных будет выглядеть так:

```
set my_name [lindex $argv 0]
```

```
set my_favorite [lindex $argv 1]
```

Отредактируем скрипт answerbot, приведя его к такому виду:

```
#!/usr/bin/expect -f

set my_name [lindex $argv 0]

set my_favorite [lindex $argv 1]

set timeout -1

spawn ./questions

expect "Hello, who are you?\r"

send -- "Im $my_name\r"

expect "Can I ask you some questions?\r"

send -- "Sure\r"

expect "What is your favorite topic?\r"

send -- "$my_favorite\r"

expect eof
```

Запустим его, передав в качестве первого параметра SomeName, в качестве второго — Programming:

```
$ ./answerbot SomeName Programming
```

Как видите, всё работает так, как ожидалось. Теперь expect-скрипт отвечает на вопросы bash-скрипта, пользуясь переданными ему параметрами командной строки.

## **Ответы на разные вопросы, которые могут появиться в одном и том же месте**

Если автоматизируемая программа может, в одной ситуации, выдать одну строку, а в другой, в том же самом месте — другую, в expect можно использовать блоки, заключённые в фигурные скобки и содержащие варианты реакции скрипта на разные данные, полученные от программы. Выглядит это так:

```
expect {  
  
    "something" { send -- "send this\r" }  
  
    "*another" { send -- "send another\r" }  
  
}
```

Здесь, если expect-скрипт увидит строку «something», он отправит ответ «send this». Если же это будет некая строка, оканчивающаяся на «another», он отправит ответ «send another».

Напишем новый скрипт, записав его в файл questions, случайным образом задающий в одном и том же месте разные вопросы:

```
#!/bin/bash  
  
let number=$RANDOM  
  
if [ $number -gt 25000 ]  
  
then  
  
echo "What is your favorite topic?"  
  
else  
  
echo "What is your favorite movie?"  
  
fi  
  
read $REPLY
```

Тут мы генерируем случайное число при каждом запуске скрипта, и, проанализировав его, выводим один из двух вопросов.

Для автоматизации такого скрипта нам и пригодится вышеописанная конструкция:

```
#!/usr/bin/expect -f  
  
set timeout -1  
  
spawn ./questions  
  
expect {
```

```

"*topic?" { send -- "Programming\r" }

"*movie?" { send -- "Star wars\r" }

}

```

```

likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./answerbot
spawn ./questions
What is your favorite movie?
Star wars
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./answerbot
spawn ./questions
What is your favorite topic?
Programming
likegeeks@likegeeks-VirtualBox ~/Desktop $

```

*Ответы на разные вопросы, появляющиеся в одном и том же месте*

Как видно, когда автоматизированный скрипт выводит строку, оканчивающуюся на «topic?», expect-скрипт передаёт ему строку «Programming». Получив в том же месте, при другом запуске программы, вопрос, оканчивающийся на «movie?», expect-скрипт отвечает: «Star wars». Это очень полезная техника.

## Условный оператор

Expect поддерживает условный оператор if-else и другие управляющие конструкции. Вот пример использования условного оператора:

```

#!/usr/bin/expect -f

set TOTAL 1

if { $TOTAL < 5 } {

puts "\nTOTAL is less than 5\n"

} elseif { $TOTAL > 5 } {

puts "\nTOTAL greater than 5\n"

} else {

puts "\nTOTAL is equal to 5\n"

}

expect eof

```

*Условный оператор в expect*



Тут мы присваиваем переменной TOTAL некое число, после чего проверяем его и выводим текст, зависящий от результата проверки.

Обратите внимание на конфигурацию фигурных скобок. Очередная открывающая скобка должна быть расположена на той же строке, что и предыдущие конструкции.

## Цикл *while*

Циклы *while* в *exrcpt* очень похожи на те, что используются в обычных *bash*-скриптах, но, опять же, тут применяются фигурные скобки:

```
#!/usr/bin/expect -f

set COUNT 0

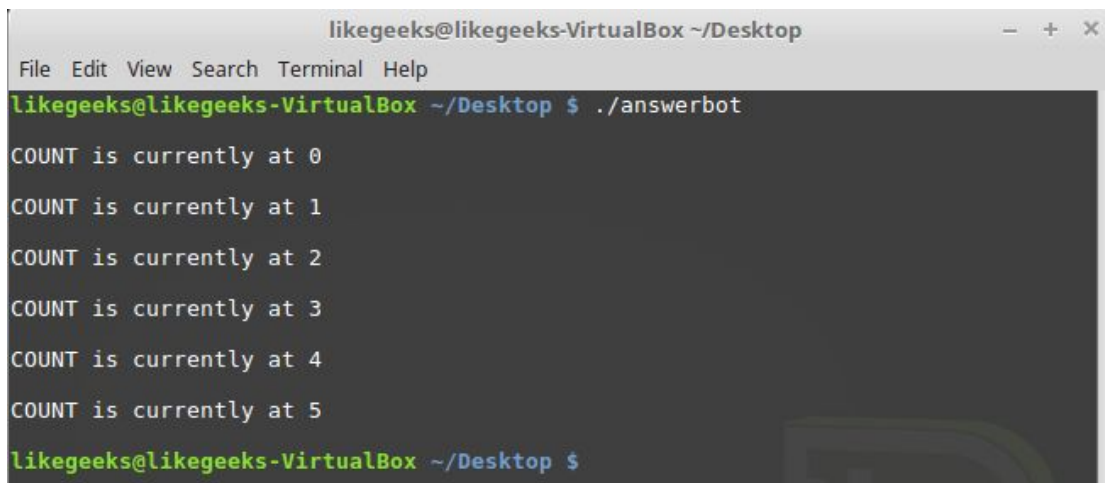
while { $COUNT <= 5 } {

puts "\nCOUNT is currently at $COUNT"

set COUNT [ expr $COUNT + 1 ]

}

puts ""
```

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 execution of a script named 'answerbot'. The output of the script is: 'COUNT is currently at 0', 'COUNT is currently at 1', 'COUNT is currently at 2', 'COUNT is currently at 3', 'COUNT is currently at 4', and 'COUNT is currently at 5'. The prompt 'likegeeks@likegeeks-VirtualBox ~/Desktop \$' is visible at the bottom.

*Цикл while в expect*

## Цикл *for*

Цикл *for* в *exrcpt* устроен по-особому. В начале цикла, в самостоятельных парах фигурных скобок, надо указать переменную-счётчик, условие прекращения цикла и правило модификации счётчика. Затем, опять же в фигурных скобках, идёт тело цикла:

```
#!/usr/bin/expect -f

for {set COUNT 0} {$COUNT <= 5} {incr COUNT} {

puts "\nCOUNT is at $COUNT"

}

puts ""
```

*Цикл for в expect*

## **Объявление и использование функций**

Expect позволяет программисту объявлять функции, используя ключевое слово `proc`:

```
proc myfunc { MY_COUNT } {  
    set MY_COUNT [expr $MY_COUNT + 1]  
    return "$MY_COUNT"  
}
```

Вот как выглядит expect-скрипт, в котором используется объявленная в нём же функция:

```
#!/usr/bin/expect -f  
  
proc myfunc { MY_COUNT } {  
    set MY_COUNT [expr $MY_COUNT + 1]  
    return "$MY_COUNT"  
}  
  
set COUNT 0  
while {$COUNT <= 5} {  
    puts "\nCOUNT is currently at $COUNT"  
    set COUNT [myfunc $COUNT]  
}
```

```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./answerbot
spawn ./questions
Hello, who are you?
Hi Im Adam
What is you password?
mypass
What is your favorite topic?
Technology
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

puts ""

```
likegeeks@likegeeks-VirtualBox ~/Desktop
File Edit View Search Terminal Help
likegeeks@likegeeks-VirtualBox ~/Desktop $ ./answerbot
COUNT is currently at 0
COUNT is currently at 1
COUNT is currently at 2
COUNT is currently at 3
COUNT is currently at 4
COUNT is currently at 5
likegeeks@likegeeks-VirtualBox ~/Desktop $
```

Функции в exect

## Команда interact

Случается так, что автоматизируемые с помощью exect программы требуют ввода конфиденциальных данных, вроде паролей, которые вам не хотелось бы хранить в виде обычного текста в коде скрипта. В подобной ситуации можно воспользоваться командой `interact`, которая позволит вам, автоматизировав некую часть взаимодействия с программой, самостоятельно ввести, скажем, пароль, а потом опять передать управление exect.

Когда выполняется эта команда, exect-скрипт переключается на чтение ответа на вопрос программы с клавиатуры, вместо того, чтобы передавать ей ранее записанные в нём данные.

Вот bash-скрипт, в общем-то, точно такой же, как мы рассматривали ранее, но теперь ожидающий ввод пароля в ответ на один из своих вопросов:

```
#!/bin/bash
echo "Hello, who are you?"
read $REPLY
echo "What is you password?"
read $REPLY
echo "What is your favorite topic?"
read $REPLY
```

Напишем exect-скрипт, который, когда ему предлагают предоставить пароль, передаёт управление нам:

```
#!/usr/bin/expect -f

set timeout -1

spawn ./questions

expect "Hello, who are you?\r"

send -- "Hi Im Adam\r"

expect "*password?\r"

interact ++ return

send "\r"

expect "*topic?\r"

send -- "Technology\r"

expect eof
```

### *Команда interact в expect-скрипте*

Встретив команду interact, expect-скрипт остановится, предоставив нам возможность ввести пароль. После ввода пароля надо ввести «++» и expect-скрипт продолжит работу, снова получив управление.

## **Итоги**

Возможностями expect можно пользоваться в программах, написанных на разных языках программирования благодаря соответствующим библиотекам. Среди этих языков — C#, Java, Perl, Python, Ruby, и другие. То, что expect доступен для разных сред разработки — далеко не случайность. Всё дело в том, что это действительно важный и полезный инструмент, который используют для решения множества задач. Здесь и проверка качества ПО, и выполнение различных работ по сетевому администрированию, автоматизация передачи файлов, автоматическая установка обновлений и многое другое.

Освоив этот материал, вы ознакомились с основными концепциями expect и научились пользоваться инструментом autoexpect для автоматического формирования скриптов. Теперь вы вполне можете продолжить изучение expect, воспользовавшись дополнительными источниками. Вот — [сборник](#) учебных и справочных материалов. Вот — достойная внимания серия из трёх статей ([1](#), [2](#), [3](#)). А вот — [официальная страница](#) expect, на которой можно найти ссылки на исходный код программы и список публикаций.

На этом мы завершаем серию материалов о bash-скриптах. Надеемся, её одиннадцать частей, а также бессчётное число комментариев к ним, помогли в достижении цели тем, кто хотел научиться писать сценарии командной строки.