

# ПРОГРАММИРОВАНИЕ НА BASH

## Часть I. Краткое введение

---

Данное введение в программирование на bash прельстило меня своей краткостью и содержательностью. В то же время я изменил некоторые примеры, потому что они делали слегка не то, что ожидается новичками. Начинающим текст будет полезен как отправная точка для начала написания скриптов. Опытным – как справочник. Удачного чтения!

Выражая также благодарность Владимиру Черному (начальнику отдела образовательных проектов ALT Linux) за внесенные в текст правки.

Тема программирования на bash из разряда тех, которые могут быть рассмотрены и в пару, и в сотни страниц. Гарольд Родригес (Harold Rodriguez) объясняет эту тему в приведенном ниже руководстве из двух частей. Его прекрасный и яркий стиль позволил ему охватить все существенные черты программирования на bash буквально на нескольких страницах.

Если вы никогда не программирували на bash ранее – сейчас самое время начать. Даже если у вас мало знаний о том, что такое bash, вы вполне можете посмотреть на множество интересных скриптов, разбираемых Гарольдом.

### ВВЕДЕНИЕ

Подобно остальным оболочкам, доступным в Linux, Bourne Again shell (bash) является не только, собственно, командной оболочкой, но и языком для написания сценариев (скриптов) (слова «сценарий» и «скрипт» обычно являются синонимами – прим. перев.). Скрипты позволяют в полной мере использовать возможности оболочки и автоматизировать множество задач, которые иначе потребуют для своего выполнения ввода множества команд. Многие программы, работающие внутри вашего компьютера с Linux – это скрипты. Если вы захотите узнать как они работают или изменить их, важно понимать их синтаксис и семантику. Кроме того, понимая язык bash, вы сможете

писать свои собственные программы, чтобы выполнять разные задачи теми способами, которые выберете сами.

### ТАК ВСЕ-ТАКИ ПРОГРАММИРОВАНИЕ (PROGRAMMING) ИЛИ НАПИСАНИЕ СКРИПТОВ (SCRIPTING)?

Новичков в программировании, как правило, озадачивает разница между, собственно, программированием и языками скриптов. Программы, написанные на каких-то языках программирования, обычно гораздо более мощные по возможностям и работают намного быстрее, чем программы, написанные на языках сценариев. Примеры языков программирования – C, C++ и Java. Создание программы на каком-либо языке программирования обычно начинается с написания исходного кода (текстовый файл, содержащий инструкции о том, как будет работать окончательная программа), затем его необходимо скомпилировать (собрать) в исполняемый файл. Этот исполняемый файл не так легко переносить между различными операционными системами. Например, если вы напишете программу на C для Linux, вы не сможете запустить ее в Windows. Чтобы сделать это, вам придется перекомпилировать исходный код под Windows. Написание скрипта также начинается с написания исходного кода, который не компилируется в исполняемый файл. Вместо этого интерпретатор оболочки последовательно читает инструкции в файле исходного кода и выполняет их. К сожалению, поскольку интерпретатор должен читать каждую инструкцию, скорость исполнения скрипта обычно медленнее (намного медленнее – прим. перев.), чем у скомпилированной программы. Основным преимуществом скриптов является то, что вы можете легко переносить исходный файл в любую операционную систему и просто запускать их (естественно при наличии интерпретатора для этой операционной системы – прим. перев.).

Bash – это язык сценариев. Он отлично подходит для написания небольших программ, но если вы планируете

создавать какие-то мощные приложения, предпочтительнее выбрать для этого какой-нибудь язык программирования. Другие примеры скриптовых языков Perl, Lisp, и Tcl.

### ЧТО НУЖНО ЗНАТЬ ДЛЯ НАПИСАНИЯ СВОИХ СКРИПТОВ?

Для этого необходимо знание основных команд Linux. Например, вы должны знать, как копировать, перемещать и создавать новые файлы. Обязательно умение использовать какой-нибудь текстовый редактор. Существуют три основных текстовых редактора в Linux: vi, emacs и pico (автор еще забыл nano, который лучше всего подходит начинающим, если не учитывать еще и mcedit. – Прим. перев.). Если вы не знакомы с vi или emacs, используйте pico или другой простой в использовании текстовый редактор.

### ВНИМАНИЕ!!! ВНИМАНИЕ!!! ВНИМАНИЕ!!!

Не следует учиться программировать на bash из-под пользователя root! В противном случае – может случиться все что угодно! Я не буду нести никакой ответственности, если вы случайно допустите ошибку и испортите вашу систему. Вы предупреждены! Используйте учетную запись обычного пользователя без каких-либо привилегий. Вы можете даже создать нового пользователя только для обучения написанию сценариев. Таким образом, худшее, что произойдет в данном случае – это исчезновение данных в каталоге этого пользователя.

### ВАША ПЕРВАЯ ПРОГРАММА НА BASH

Нашей первой программой будет классическая «Hello World». Конечно, если уже вы программирували раньше, то, должно быть, устали от таких примеров. Однако это – традиция и кто я такой, чтобы менять ее? Программа «Hello World» просто выводит слова «Hello World» на экран. Запустите текстовый редактор и наберите в нем следующее:

```
#!/bin/bash
echo «Hello World»
```

Первая строка сообщает Linux использовать интерпретатор bash для запуска этого скрипта. В этом случае bash находится в каталоге /bin. Если у вас bash находится где-то еще, сделайте соответствующие изменения в данной строке. Явное указание интерпретатора очень важно, удостоверьтесь еще раз, что указали его, поскольку данная строка говорит Linux какой именно интерпретатор нужно использовать для выполнения инструкций в скрипте. Следующее, что нужно сделать, это сохранить скрипт. Назовем его hello.sh. После этого вам нужно сделать скрипт исполняемым:

```
chmod u+x hello.sh
```

Если вы не понимаете, что делает эта команда, прочтите справочную страницу команды chmod:

```
man chmod
```

Как только это будет сделано, вы сможете запустить программу, просто набрав ее название:

```
./hello.sh
Hello World
```

Получилось! Это ваша первая программа! И хотя она скучная и не делает ничего полезного, она показывает как именно все работает. Просто запомните эту простую последовательность действий: напишите код, сохраните файл, сделайте его исполняемым с помощью chmod и запустите.

### КОМАНДЫ, КОМАНДЫ И КОМАНДЫ

Что именно делает ваша первая программа? Она печатает на экран слова «Hello World». Каким образом она это делает? Она использует команды. В нашей программе мы написали только одну команду – echo «Hello World». Что именно тут команда? echo. Эта программа принимает один аргумент и печатает его на экран.

Аргументом является все, что следует после ввода названия программы. В нашем случае «Hello World» это и есть аргумент, переданный команде echo. При вводе команды ls /home/, аргументом команды ls является /home. Ну и что это все означает? А означает это то, что если у вас есть программа, которая принимает какой-то аргумент и выводит что-то на экран, вы можете использовать ее вместо echo. Предположим, что у нас есть программа под названием foo. Эта программа будет принимать один аргумент (строку из слов) и печатать их на экран. Тогда мы можем переписать нашу программу вот так:

```
#!/bin/bash
foo «Hello World»
```

Сохраните ее, сделайте исполняемой и перезапустите ее (примечание для новичков – этот пример работать не будет. – Прим. перев.):

```
./hello
Hello World
```

Точно такой же результат. Использовался ли тут какой-то уникальный код? Нет. Написали ли мы какую-то программу? Нет, если только вы не являетесь автором программы echo. Все, что вы сделали – просто встроили программу echo в наш скрипт и снабдили ее аргументом. Примером альтернативы использования команды echo

в реальном программировании является команда printf, которая имеет больше возможностей, если вы знакомы с программированием на С. Ну и на самом деле, точно такой же результат можно было бы получить и без написания скрипта:

```
echo «Hello World»
Hello World
```

Написание скриптов на bash предлагает широкий спектр возможностей и этому легко научиться. Как вы только что могли видеть, здесь используются разные команды Linux, чтобы писать собственные скрипты. Ваша программа-оболочка представляет собой несколько других программ, собранных вместе для выполнения какой-либо задачи.

## ДРУГИЕ ПОЛЕЗНЫЕ ПРОГРАММЫ

Сейчас мы напишем программу, которая переместит все файлы в каталог, удалит его вместе с содержимым, а затем создаст это каталог заново. Это может быть сделано с помощью следующих команд (В примере, приведенном в оригинале автор показывает, что не зря рекомендовал делать приведенные упражнения под специально созданным пользователем. Результатом выполнения данной последовательности команд будет чистый каталог, в котором вы работаете. Скорее всего это будет ваша домашняя директория. Поэтому, если вы НЕ хотите удаления всех файлов в ней – НЕ выполняйте команды из оригинала статьи. А лучше последуйте совету автора и создайте отдельного пользователя специально для тренировки написания скриптов. Этот пример я немного расширил и теперь он не такой опасный. – Прим. перев.):

```
touch file1
mkdir trash
mv file1 trash
rm -rf trash
mkdir trash
```

Вместо того, чтобы вводить это все в интерактивном режиме, напишем скрипт, выполняющий эти команды:

```
#!/bin/bash
touch file1
mkdir trash
mv file1 trash
rm -rf trash
mkdir trash
echo “Файл удален!”
```

Сохраните его под именем clean.sh и теперь все, что нужно сделать – запустить его. Он переместит все файлы в каталог, удалит его, создаст заново каталог и даже напечатает сообщение об удалении файлов. Запомните, если

вы обнаружите, что регулярно делаете нечто требующее набора одной и той же последовательности команд – это вполне можно автоматизировать написанием скрипта.

## КОММЕНТАРИИ

Комментарии помогают сделать ваш код более читабельным. Они не влияют на то, что выводит программа. Они написаны специально для того, чтобы вы их прочли. Все комментарии в Bash начинаются с хэш-символа #, за исключением первой строки (#!/bin/bash), имеющей специальное назначение. Первая строка – не комментарий. Возьмем для примера следующий код:

```
#!/bin/bash
# Эта программа считает от 1 до 10:
for i in 1 2 3 4 5 6 7 8 9 10; do
echo $i
done
```

Даже если вы пока не понимаете скрипты на Bash, вы сразу же поймете, что делает приведенный выше пример, благодаря комментарию. Комментирование кода – хорошая практика. Со временем вы поймете, что, если вам нужно будет поддерживать ваши скрипты, то при наличии комментированного кода – делать это станет легче.

## ПЕРЕМЕННЫЕ

Переменные это просто «контейнеры», которые содержат некоторые значения. Создавать их нужно по многим причинам. Вам нужно будет как-то сохранять вводимые пользователем данные, аргументы или числовые величины. Например:

```
#!/bin/bash
x=12
echo "Значение переменной x - $x"
```

Здесь мы присвоили переменной x значение 12. Стока echo “Значение переменной x - \$x” напечатает текущее значение x. При определении переменной не допускается наличие каких-то пробелов между именем переменной и оператором присваивания: «=». Синтаксис следующий:

имя\_переменной=ее\_значение

Обращение к переменным выполняется с помощью префикса «\$» перед именем переменной. Именно таким образом мы получаем доступ к значению переменной x с помощью команды echo \$x.

Есть два типа переменных – локальные и переменные окружения (среды). Переменные окружения устанавлива-

ются системой и имеют специальной назначение. Обычно их значение может быть выведено с помощью команды echo. Например, если ввести:

```
echo $SHELL
/bin/bash
```

Вы получили имя оболочки, запущенной в данный момент. Переменные среды задаются в файле /etc/profile и в ~/.bash\_profile. Команда echo может применяться для проверки текущего значения переменной.

Примечание: задание переменных среды подробно описано в этой статье – «Как задавать переменные среды» – <http://learnbyexamples.org/linux/linux-tip-how-to-set-shell-environment-variables-bash-shell.html>. В статье также описаны некоторые особенности оболочки Bash.

Если у вас все еще возникают проблемы с пониманием того, зачем нужно использовать переменные, приведем пример:

```
#!/bin/bash
echo «Значение x - 12».
echo «У меня есть 12 карандашей».
echo «Он сказал мне, что значение x равно
12».
echo «Мне 12 лет.»
echo «Как получилось, что значение x равно 12?»
```

Хорошо, теперь предположим, что вы решите поменять значение x на 8 вместо 12. Что для этого нужно сделать? Вы должны изменить все строки кода, в которых говорится, что x равно 12. Но погодите... Есть другие строки кода, где упоминается это число, поэтому простую автозамену использовать не получится. Теперь приведем аналогичный пример, только с использованием переменных:

```
#!/bin/bash
x=12 # задаем переменной x значение 12
echo «Значение x = $x»
echo «У меня есть 12 карандашей»
echo «Он сказал мне, что значение x равно
$x»
echo «Мне 12 лет»
echo «Как получилось, что значение x равно
но $x?»
```

Здесь мы видим, что \$x выводит текущее значение переменной x равное 12. Поэтому теперь, если вы хотите задать новое значение x равное 8, то все что вам нужно сделать, это изменить одну строчку с x=12 на x=8, и в выводе все строки с упоминанием x также изменятся. Поэтому вам не нужно руками модифицировать остальные строки.

Как вы увидите позже, переменные имеют и другие способы применения.

## УПРАВЛЯЮЩИЕ ОПЕРАТОРЫ

Управляющие операторы делают вашу программу компактнее и позволяют ей принимать решения. И, что еще более важно, они позволяют нам выполнять проверку на наличие ошибок. До сих пор все, что мы сделали, это писали скрипты, которые просто исполняют набор инструкций в файле. Например:

```
#!/bin/bash
cp /etc/foo .
echo "Готово"
```

Это небольшой скрипт, назовем его bar.sh, копирует файл с именем /etc/foo в текущий каталог и выводит «Готово» на экране. Эта программа будет работать при одном условии - файл /etc/foo должен существовать. В противном случае вот что произойдет:

```
./bar.sh
cp: /etc/foo: No such file or directory
Готово
```

Таким образом, как вы видите, существует проблема. Не у каждого, кто будет запускать вашу программу, будет файл /etc/foo. Наверное, было бы лучше, если бы ваша программа сначала проверяла наличие данного файла, а затем при положительном ответе – выполняла бы копирование, в противном случае – просто бы завершала работу. В псевдо-коде это выглядит так:

```
если /etc/foo существует, то
скопировать /etc/foo в текущую директо-
рию
напечатать «Готово» на экране
в противном случае,
напечатать на экране «Этот файл не суще-
ствует»
выход
```

Можно ли это сделать в Bash? Конечно! Набор управляющих операторов Bash включает в себя: if, while, until, for и case. Каждый из этих операторов является парным, то есть начинается он одним тегом и заканчивается другим. Например, если условный оператор if начинается с if и заканчивается fi. Управляющие операторы cp /etc/foo . – это не отдельные программы в системе, они встроены в bash.

```
if ... else ... elif ... fi
```

Это один из наиболее распространенных операторов. Он позволяет программе принимать решения следующим образом – «если условие верно – делаем одно, если нет

– делаем что-то другое». Чтобы эффективно его использовать, сначала нужно научиться пользоваться командой test. Эта программа выполняет проверку условия (например, существует ли файл, есть ли необходимые права доступа). Вот переписанный вариант bar.sh :

```
#!/bin/bash
if test -f /etc/foo
then
```

```
# Файл существует, копируем его и печатаем сообщение на экране
cp /etc/foo .
echo «Готово».
```

```
else # Файл не существует, поэтому мы печатаем сообщение
```

```
#и завершаем работу
echo «Этот файл не существует.»
exit
fi
```

Обратите внимание на переводы строки после then и else. Они не являются обязательными, но делают чтение кода гораздо более простым в том смысле, что делают логику программы более наглядной. Теперь запустите программу. Если у вас есть файл /etc/foo – он будет скопирован, в противном случае будет напечатано сообщение об ошибке. Команда test проверяет существование файла. Ключ -f проверяет, является ли аргумент обычным файлом. Ниже приведен список опций test (Не стоит пытаться запомнить их все, т.к. это все равно нереально. Его всегда можно посмотреть в руководстве команды test – man test. Прим. перев.):

## КЛЮЧИ КОМАНДЫ TEST:

- d проверяет наличие файла и то, что он является каталогом
- e проверяет существование файла
- f проверяет наличие файла и то, что это обычный файл
- g проверяет наличие у файла SGID-бита
- r проверяет наличие файла и то, что он доступен на чтение
- s проверяет наличие файла и то, что его размер больше нуля
- u проверяет наличие у файла SUID-бита
- w проверяет наличие файла и то, что он доступен на запись
- x проверяет наличие файла и наличие у него прав на запуск

Оператор else используется, когда вы хотите, чтобы ваша программа еще что-то делала, если первое условие не выполняется. Существует также оператор elif, который

может использоваться вместо еще одного if. elif означает «else if». Он используется, когда первое условие не выполняется, и вы хотите проверить еще одно условие.

Если вам не нравится приведенная форма записи if и test, есть сокращенный вариант.

Например, код:

```
if test -f /etc/foo
then
```

Можно записать вот так:

```
if [ -f /etc/foo ]; then
```

Квадратные скобки – это еще один вариант записи test. Если у вас есть опыт в программировании на C, этот синтаксис для вас может быть более удобным. Обратите внимание на наличие пробелов до и после каждой из скобок (Наличие пробелов объясняется просто: открывающая квадратная скобка – это команда оболочки). В этом можно легко убедиться набрав в консоли команду which [. А раз это отдельная команда, то ее нужно отделить пробелами от остальных опций. – Прим. перев.). Точка с запятой: «;» говорит оболочке о завершении одного оператора и начале следующего. Все, что находится после этого символа будет работать так, как будто находится в отдельной строке. Это делает код более удобным для чтения и, естественно, такая запись необязательна. Если вы предпочитаете другой вариант записи – then можно сразу поместить в другой строке.

Если вы используете переменные – их нужно помещать в кавычки. Например:

```
if [ «$name» -eq 5 ]; then
```

оператор -eq будет объяснен далее в этой статье.

while ... do ... done

Оператор while используется для организации циклов. Он работает так «пока (while) условие истинно, делать что-то». Рассмотрим это на примере:

```
#!/bin/bash
while true; do
echo «Нажмите CTRL-C для выхода»
done
```

true – это тоже программа. Единственное, что она тут делает – это запускает тело цикла снова и снова. Использование true считается медленным, потому что ваш скрипт должен запускать ее раз за разом. Можно использовать альтернативный вариант:

```
#!/bin/bash
while :; do
```

```
echo «Нажмите CTRL-C для выхода»
done
```

Это позволяет добиться точно такого же эффекта, но быстрее, потому что «`:`» – это встроенная функция bash. Единственное отличие состоит в принесении в жертву читабельности кода. Используйте из приведенных вариантов тот, который вам нравится больше. Ниже приведен гораздо более полезный вариант использования переменных:

```
#!/bin/bash
x=0;
while [ «$x» -le 10 ]; do
echo «Текущее значение x: $ x»
# Увеличиваем значение x:
x=$(expr $x + 1)
sleep 1
done
```

Здесь мы используем для проверки состояния переменной `x` запись с квадратными скобками. Опция `-le` означает «меньше или равно (less or equal)». Говоря обычным языком приведенный код говорит: «пока (`while`) `x` меньше или равен 10, выводить на экран текущее значение `x`, после чего добавлять к текущему значению `x` единицу». Оператор `sleep 1` приостанавливает выполнение программы на одну секунду.

Ниже приведен список возможных операций сравнения целых чисел (полный список приведен в `man test`. – Прим. перев.):

```
x -eq y – x = y (equal)
x -ne y – x не равен y (not equal)
x -gt y – x больше либо равен y (greater than)
x -lt y – x меньше либо равен y (lesser than)
```

### ОПЕРАТОРЫ СРАВНЕНИЯ СТРОК:

```
x = y – строка x идентична y
x != y – строка x не совпадает y
-n x – выражение истинно, если строка x ненулевой длины
-z x – выражение истинно, если строка x имеет нулевую длину
```

Скрипт, приведенный выше, нетрудно понять, за исключением, может быть, только этой строки:

```
x=$(expr $x + 1)
```

Комментарий приведенный выше он говорит нам, что он увеличивает `x` на 1. Но что означает запись `$ (...)?` Это переменная? Нет. На самом деле это способ сказать оболочке, что вы хотите запустить команду `expr $x + 1`, и присвоить результат ее выполнения переменной `x`. Любая ко-

манда, заключенная в `$ (...)` будет выполняться:

```
#!/bin/bash
me=$(whoami)
echo «Привет! Меня зовут $me»
```

Попробуйте сделать приведенный пример, и вы поймете, что я имею в виду. Приведенный выше код можно было бы сократить без каких-либо потерь вот так:

```
#!/bin/bash
echo «Привет! Меня зовут $(whoami)»
```

Вы сами можете выбрать, какая из записей вам ближе и понятнее. Существует и другой способ для выполнения команд или передачи результата их выполнения переменной. Как это сделать – будет объяснено позже. Пока используйте запись вида `$(...).`

`until ... do ... done`

Оператор `until` применяется аналогично приведенному выше `while`. Разница лишь в том, что условие работает наоборот. Цикл `while` выполняется до тех пор, пока условие истинно. Цикл `until` – до тех пор, пока условие не станет истинным. Например:

```
#!/bin/bash
x=0
until [ «$x» -ge 10 ]; do
echo «Текущее значение x равно $ x»
x=$(expr $x + 1)
sleep 1
done
```

Эта часть кода выглядит знакомой. Попробуйте ее набрать и посмотреть, что он делает. Приведенный цикл будет работать, пока `x` не станет больше или равен 10. Когда величина `x` достигнет значения 10, цикл остановится. Таким образом, последнее значение напечатанное значение `x` будет 9.

`for ... in ... do ... done`

Цикл `for` используется, когда вам надо перебрать несколько значений переменной. Например, вы можете написать небольшую программу, которая печатает 10 точек:

```
#!/bin/bash
echo -n «Проверка системы на наличие ошибок»
for dots in 1 2 3 4 5 6 7 8 9 10; do
echo -n «.»
done
echo «Ошибка не обнаружено»
```

Опция `-n` команды `echo` предотвращает автоматический перевод строки. Попробуйте один раз вариант с `-n` и вариант без этой опции, чтобы понять, что я имею в виду.

Переменная dots последовательно принимает значения от 1 до 10 и одновременно скрипт печатает на экране точку.

Приведенный дальше пример показывает, что я имею в виду под выражением «переменная последовательно принимает несколько значений»:

```
#!/bin/bash
for x in paper pencil pen; do
echo «значение переменной x равно $x»
sleep 1
done
```

При запуске программы, вы видите, что x сначала имеет значение «pencil», а затем она принимает значение «pen». Когда у переменной заканчивается список возможных значений, цикл завершается.

Ниже приведен гораздо более полезный пример. Этот скрипт добавляет расширение .html для всех файлов в текущей директории (Этот скрипт действительно так и поступает и вам возможно это не нужно. Поэтому все-таки создайте отдельного пользователя, если вы еще до сих пор этого не сделали, и экспериментируйте под ним. – Прим. перев.):

```
#!/bin/bash
for file in *; do
echo «добавляем расширение .html для файла $file ...»
mv $file $file.html
sleep 1
done
```

Символ \* имеет специальное значение, которое в данном случае означает «все в текущем каталоге», т.е. все файлы в каталоге. Переменная file последовательно принимает значения, соответствующие именам файлов в текущем каталоге. Затем используется программа mv для переименования файла в файл с расширением .html.

case ... in ... esac

Оператор case очень похож на if. Он отлично подходит для тех случаев, когда нужно проверить несколько условий и вы не хотите для этого использовать несколько вложенных операторов if. Поясним на примере:

```
#!/bin/bash
x=5 # инициализируем x значением 5
# проверяем значение x:
case $x in
0) echo «значение x равно 0»
;;
5) echo «значение x равно 5»
;;
9) echo «значение x равно 9»
;;
*) echo «значение неизвестно»
```

```
;;
esac
```

Оператор case проверяет переменную x на равенство трем значениям. В приведенном примере, он сначала проверит, равен ли x нулю 0, затем равен ли он 5, затем равен ли он 9. И, если все проверки завершились неудачно, скрипт выведет сообщение, что значение x определить не получилось. Помните, что «\*» означает «все», а в этом случае, «любое другое значение, помимо указанных явно». Если x имеет любое другое значение, отличное от 0, 5 или 9, то это значение попадает во категорию «\*». При использовании case каждое условие должно заканчиваться двумя точками с запятой.

Зачем нужно использовать case, когда вы можно использовать if? Ниже приведен пример эквивалентного скрипта, написанного с использованием if. Решение о том, что быстрее написать и легче прочесть, предлагается принять самостоятельно ;):

```
#!/bin/bash
x=5 # инициализируем x значением 5
if [ «$x» -eq 0 ]; then
echo «Значение x равно 0»
elif [ «$x» -eq 5 ]; then
echo «значение x равно 5»
elif [ «$x» -eq 9 ]; then
echo «значение x равно 9»
else
echo «Значение x определить не удалось»
fi
```

## ИСПОЛЬЗОВАНИЕ КАВЫЧЕК

Кавычки играют важную роль в написании скриптов оболочки. Существует три типа кавычек. Это: двойные кавычки «, одинарные ' (апостроф) и обратные ` (находятся слева от клавиши 1. – Прим. перев.). Имеет ли каждый из приведенных видов какое-то особое значение? Да.

Примечание: Статья Wildcards, Quotes, Back Quotes, Apostrophes in shell commands (\* ? [] " ' ') прекрасно описывает использование специальных символов. Пожалуйста, ознакомьтесь с ней в случае, если вы не знакомы с использованием этих специальных символов в скриптах оболочки. Ниже приведено краткое объяснение использования некоторых из них.

Двойные кавычки используются главным образом для объединения нескольких слов в строку и сохранения в ней пробелов. Например, «Эта строка содержит пробелы». Стока, заключенная в двойные кавычки рассматривается как единое целое. Например:

```
mkdir hello world
ls -F
```

```
hello/ world/
```

Здесь мы создали две директории. Команда `mkdir` принимает два слова `hello` и `world`, как два отдельных аргумента, и поэтому создает два каталога. Теперь посмотрим, что произойдет, если написать код таким образом:

```
$ mkdir "hello world"
$ ls -F
hello/ hello world/ world/
```

Команда создала каталог с именем из двух слов. Кавычки объединили два слова в один аргумент. (Главным образом, дело в том, что `bash` воспринимает пробел как разделитель всего, что только можно — опций, аргументов, отдельных команд. Внутри двойных кавычек пробел теряет свое специальное значение. — Прим. перев.).

Одинарные кавычки в основном используются для работы с переменными. Если переменная находится в двойных кавычках, то к ней можно обратиться через `$имя_переменной`. Если переменная находится в одинарных кавычках — это невозможно. Чтобы пояснить это, приведем пример:

```
#!/bin/bash
x=5 # задаем x равным 5
# используем двойные кавычки
echo «Используем двойные кавычки, значение x равно $x»
# используем одинарные кавычки
echo 'Используем одинарные кавычки, значение x равно $x'
```

Почувствовали разницу? Вы можете использовать двойные кавычки, если вы не планируете использовать переменные для строк, которая в них находится. И да, если вам интересно, прямые кавычки также можно использовать для сохранения пробелов в строке тем же способом, что и двойные кавычки

```
mkdir 'hello world'
ls -F
hello world/
```

Обратные кавычки сильно отличаются от двойных и одинарных. Они не могут использоваться для сохранения пробелов. Если вы помните, выше мы использовали такую строку:

```
x=$(expr $x + 1)
```

Как вы уже знаете, результатом работы этой команды будет то, что выражение `$x + 1` присваивается переменной `x`. Того же результата можно достичь и с использованием обратных кавычек:

```
x='expr $x + 1'
```

Какой тип кавычек лучше использовать? Тот, что вам больше нравится. Изучая скрипты вы найдете, что обратные кавычки используются чаще, чем запись `$(...)`. Тем не менее, я считаю, `$(...)` легче читать, особенно если у вас код наподобие этого:

```
#!/bin/bash
echo "I am 'whoami'"
```

(На мой взгляд, лучше использовать именно запись типа `$(...)`, потому что запись в обратных кавычках и одинарных можно легко перепутать при наборе кода и при его чтении. — Прим. перев.)

Это только начало. Вы узнаете еще много интересного в заключительной части этой статьи (надеюсь со временем, как только текст появится, осилить и его перевод. — Прим. перев.). А пока вы ждете — удачного вам написания скриптов...

Перевод Чернышов Антон,  
Linux-преподаватель УЦ R-Style  
[www.tux-the-penguin.blogspot.com](http://www.tux-the-penguin.blogspot.com)

## Включаем NumLock при старте

Больше всего по утрам меня когда-то раздражал выключенный по-умолчанию NumLock в Ubuntu/Xubuntu/Kubuntu и иже с ними. Если у вас в пароле к учетной записи есть цифры — вы меня поймете. При нажатии на клавиши NumPad'a фокус вдруг начинал скакать по элементам управления, словно строптивый конь.

Как оказалось, решение такой серьезной проблемы для всех, у кого есть цифровая клавиатура, оказалось очень простым. Вот тут — [www.help.ubuntu.com/community/NumLock](http://www.help.ubuntu.com/community/NumLock) — есть целая страница помощи с множеством способов для различных систем.

Я же предлагаю вам довериться мне и сделать все универсально и через консоль :)

Для начала установим маленькую (10 Кб) утилиту `numlockx`, управляющую NumLock'ом:

```
sudo apt-get install numlockx
```

А теперь добавим в конфигурационный файл «иксов» загрузку этой утилиты и включение NumLock'a:

```
sudo su -
echo /usr/bin/numlockx on >> /etc/X11/xinit/xinitrc
```

Вот и всё :)

[www.pingvinus.ru](http://www.pingvinus.ru)



And  
**User** Linux