Интерфейс командной строки (Command line interface, CLI) — разновидность текстового интерфейса (TUI) между человеком и компьютером, в котором инструкции компьютеру даются в основном путём ввода с клавиатуры текстовых строк (команд). Также известен под названиями «консоль» и «терминал».
Терминал в Linux — это программа, которая предоставляет пользователю возможность вести диалог с системой при помощи интерфейса командной строки. Стандартный терминал к системе Linux можно получить на любой текстовой виртуальной консоли, а для того, чтобы получить доступ к командной стоке из графической оболочки, требуются специальные программы — эмуляторы терминала.
Командная оболочка (или интерпретатор команд) — это программа, задача которой состоит в том, чтобы передавать ваши команды операционной системе и прикладным программам, а их ответы — вам. На языке командной оболочки можно писать программы — сценарии (скрипты).
Зарегистрировавшись в системе (введя имя пользователя и пароль), вы увидите приглашение командной строки — строку, оканчивающуюся символом $ (или # для root).
Командный интерпретатор исполняет команды своего языка, заданные в командной строке или поступающие из стандартного ввода или указанного файла.
В качестве команд интерпретируются вызовы системных или прикладных утилит, а также управляющие конструкции. Кроме того, оболочка отвечает за раскрытие шаблонов имен файлов и за перенаправление и связывание ввода-вывода утилит.
В совокупности с набором утилит оболочка представляет собой операционную среду, язык программирования и средство решения как системных, так и некоторых прикладных задач, в особенности автоматизации часто выполняемых последовательностей команд.
В Linux наиболее распространенной является оболочка проекта GNU — *bash* (Bourne Again SHell). bash основывается на оболочке Bourne (sh) созданной Стефеном Борном и включает в себя свойства множества других оболочек - C (csh), Korn (ksh), tc (tcsh).
Свойства оболочки bash делают ее наиболее универсальным и удобным средством взаимодействия с ОС Linux. Bash:

- обеспечивает редактирование командной строки. Курсор может быть перемещен в любую позицию команды для изменения ее содержания
- поддерживает режим истории команд, позволяя отображать и изменять ранее введенные команды. Оболочка bash также имеет несколько переменных, значения которых влияют на сохранение и повторный вызов ранее введенных команд
- обеспечивает завершение частично введенных слов для имен переменных, пользователей, хостов, команд и файлов
- обеспечивает гибкое управление процессами, позволяя приостанавливать их, перезапускать, переключаться между задачами переднего плана и фоновыми, и даже продолжать выполнение задачи при завершении родительского процесса
- позволяет использовать функции и псевдонимы, выполнять арифметические операции и вводить арифметические выражения в качестве команд
- позволяет создавать целые наборы последовательно выполняемых команд (сценарии или скрипты) с использованием гибкого и функционального языка программирования данной оболочки
- имеет гибкие настройки, обеспечивающие изменение внешнего вида командной строки

Оболочка bash может выполняться как в интерактивном, так и в неинтерактивном режимах. В первом случае bash взаимодействует с пользователем, во втором - используется для выполнения скриптов (специально подготовленных текстовых файлов с последовательностью команд).
Команда оболочки Linux - это строка символов из имени команды и аргументов, разделенных пробелами. Аргументы предоставляют команде дополнительные параметры, определяющие ее поведение. Например, команда

echo 12345
выведет на экран строку символов 12345, введенных в качестве аргумента команды.
Команды, являющиеся частью оболочки называются встроенными. Естественно, они могут отличаться для различных оболочек. Кроме того, в качестве команд используются имена исполняемых файлов. В качестве аргументов, командам передаются ключи или опции, состоящие из тире и одного или нескольких символов. Пример такой команды:
ls -l
ls - команда для отображения информации о файлах. При вводе без аргументов, эта команда просто отобразит список файлов в текущем каталоге. При вводе с параметром -l - список в длинном формате - с отображением атрибутов, владельцев, даты и времени.
При использовании нескольких ключей, их можно объединить. Ниже приведенные варианты команд идентичны :
ls -l -d
ls -ld
Клавиша TAB служит для дополнения строки. Например вам нужно ввести длинное и невероятно сложное имя файла. Просто наберите начало названия этого файла и нажмите TAB — оно автоматически дополнится. То же относится ко всем командам.

Как получить справку по использованию команды Linux

При работе в командной оболочке Linux, в большинстве случаев, можно получить справочную информацию по использованию конкретной команды, введя ее имя с параметром --help:

ls --help отобразит справку для команды ls. В некоторых случаях, для получения справочной информации, допускается использование короткого ключа -h

В операционных системах семейства Linux, более подробную справочную информацию можно получить из комплекта документации, известного как man-страницы, поскольку он доступен по команде man.

man (от manual — руководство) - команда предназначенная для форматирования и вывода справочных страниц. Поставляется почти со всеми UNIX-подобными дистрибутивами.

Чтобы вывести справочное руководство по какой-либо команде (или программе, предусматривающей возможность запуска из терминала), можно в консоли ввести:

man <command_name>

Например, чтобы посмотреть справку по команде ls, нужно ввести man ls

Для навигации в справочной системе man можно использовать клавиши ↑ и ↓ для построчного перехода, PgUp и PgDn для постраничного перехода вверх и вниз соответственно.

При просмотре больших страниц удобно воспользоваться поиском, для чего следует нажать /, затем набрать строку поиска (и слеш, и строка поиска отобразятся в нижней части экрана) и нажать Enter. Обратным поиском (снизу вверх) можно воспользоваться, нажав кнопку ?. При этом подсветятся все совпадения с заданным регулярным выражением, и экран прокрутится до первого из них. Для перехода к следующему подсвеченному вхождению нужно нажать n либо оставить строку поиска пустой (/, затем - Enter). Для показа предыдущего совпадения надо также использовать вопросительный знак или же нажимать N

Для получения краткой справки по командам и горячим клавишам справочной системы нужно нажать h

Для выхода из справочной системы используется клавиша q.

Для получения детальной инструкции по использованию команды используется конструкция man man

Перенаправление вывода

В системе по-умолчанию всегда открыты три "файла" — stdin (клавиатура), stdout (экран) и stderr (вывод сообщений об ошибках на экран). Эти, и любые другие открытые файлы, могут быть перенапрвлены. В данном случае, термин "перенаправление" означает получить вывод из файла, команды, программы, сценария или даже отдельного блока в сценарии и передать его на вход в другой файл, команду, программу или сценарий.

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

```
  COMMAND_OUTPUT >
    # Перенаправление stdout (вывода) в файл.
    # Если файл отсутствовал, то он создется, иначе -- перезаписывается.
    ls -lR > dir-tree.list
    # Создает файл, содержащий список дерева каталогов.

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

  > filename
    # Операция > усекает файл "filename" до нулевой длины.
    # Если до выполнения операции файла не существовало,
    # то создается новый файл с нулевой длиной (тот же эффект дает команда 'touch').

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



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

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

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

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

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

    bad_command1 2>$ERRORFILE       #  Сообщение об ошибке запишется в $ERRORFILE.
    bad_command2 2>>$ERRORFILE      #  Сообщение об ошибке добавится в конец
$ERRORFILE.
    bad_command3                    #  Сообщение об ошибке будет выведено на stderr,
                                    #+ и не попадет в $ERRORFILE.
  2>&1
    # Перенаправляется stderr на stdout.
    # Сообщения об ошибках передаются туда же, куда и стандартный вывод.

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

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

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


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

  |
    # Конвейер (канал).
    # Универсальное средство для объединения команд в одну цепочку.
    # Похоже на ">", но на самом деле -- более обширная.
```

```
    # Используется для объединения команд, сценариев, файлов и программ в одну цепочку
    cat *.txt | sort | uniq > result-file
    # Содержимое всех файлов .txt сортируется, удаляются повторяющиеся строки,
    # результат сохраняется в файле "result-file".
```
Операции перенаправления и/или конвейеры могут комбинироваться в одной командной строке.
```
command < input-file > output-file
command1 | command2 | command3 > output-file
```

Допускается перенаправление нескольких потоков в один файл.
```
ls -yz >> command.log 2>&1
# Сообщение о неверной опции "yz" в команде "ls" будет записано в файл "command.log".
# Поскольку stderr перенаправлен в файл.
```

Закрытие дескрипторов файлов
```
n<&-
```
Закрыть дескриптор входного файла *n*.
```
0<&-, <&-
```
Закрыть stdin.
```
n>&-
```
Закрыть дескриптор выходного файла *n*.
```
1>&-, >&-
```
Закрыть stdout.
Дочерние процессы наследуют дескрипторы открытых файлов. По этой причине и работают конвейеры.

Команда **exec <filename** перенаправляет ввод с stdin на файл. С этого момента весь ввод, вместо stdin, будет производиться из этого файла. Это дает возможность читать содержимое файла, строку за строкой, и анализировать каждую введенную строку
```
#!/bin/bash
# Перенаправление stdin с помощью 'exec'.


exec 6<&0          # Связать дескр. #6 со стандартным вводом (stdin) сохраняя stdin.
exec < data-file   # stdin заменяется файлом "data-file"
read a1            # Читается первая строка из "data-file".
read a2            # Читается вторая строка из "data-file."

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

echo; echo; echo

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

```
echo -n "Введите строку  "
read b1  # Теперь функция "read", как и следовало ожидать, принимает данные с обычного stdin.
echo "Строка, принятая со stdin."
echo "--------------------------"
echo "b1 = $b1"
echo
exit 0
```

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

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

LOGFILE=logfile.txt

exec 6>&1           # Связать дескр. #6 со stdout. Сохраняя stdout.
exec > $LOGFILE     # stdout замещается файлом "logfile.txt".
# Весь вывод от команд, в данном блоке, записывается в файл $LOGFILE.
echo -n "Logfile: "
date
echo "-------------------------------------"
echo

echo "Вывод команды \"ls -al\""
echo
ls -al
echo; echo
echo "Вывод команды \"df\""
df
exec 1>&6 6>&-      # Восстановить stdout и закрыть дескр. #6.
echo "== stdout восстановлено в значение по-умолчанию == "
ls -al
exit 0
```

Выполнение программ в фоновом режиме

Запуск команды в качестве фоновой означает, что команда выполняется в оперативной памяти, в то время как управление командной строкой оболочки возвращается вашей консоли. Это удобный способ работы в Linux, такие задачи, как сортировка больших файлов или поиск в каталогах и других файловых системах, неплохие кандидаты на выполнение в фоновом режиме.

Для запуска программ в фоновом режиме в конец командной строки добавляется символ амперсанда (&). Например: command &

В выводе терминала будут отображены порядковый номер задачи (в квадратных скобках) и идентификатор процесса. Вы можете остановить процесс с помощью команды оболочки kill, указав номер процесса для программы.

Работая в фоновом режиме, команда все равно продолжает выводить сообщения в терминал, из которого была запущена. Для этого она использует потоки stdout и stderr, которые можно закрыть например так:

command > /dev/null 2>&1 &

Узнать состояние всех остановленных и выполняемых в фоновом режиме задач в рамках текущей сессии терминала можно при помощи утилиты jobs с использованием опции **-l**:

jobs -l

Вывод содержит порядковый номер задачи, идентификатор фонового процесса, состояние задачи и название команды, которая запустила задание.

В любое время можно вернуть процесс из фонового режима на передний план. Для этого служит команда fg. Если в фоновом режиме выполняется несколько программ, следует также указывать номер. Например: fg %1

Если изначально процесс был запущен обычным способом, его можно перевести в фоновый режим, выполнив следующие действия:

Остановить выполнение команды, нажав комбинацию клавиш **Ctrl+Z**.

Перевести процесс в фоновый режим при помощи команды **bg**.

Закрытие терминала влечет за собой завершение всех фоновых процессов. Впрочем, есть несколько способов сохранить их после того как связь с интерактивной оболочкой прервется. Команда nohup выполняет другую команду, которая была указана в качестве аргумента, при этом игнорирует все сигналы SIGHUP (те, которые получает процесс при закрытии терминала). Для запуска команды в фоновом режиме нужно написать команду в виде: nohup command &

вывод команды перенаправляется в файл nohup.out. При этом после выхода из системы или закрытия терминала процесс не завершается. Существует ряд программ, которые позволяют запускать несколько интерактивных сессий одновременно. Наиболее популярные из них — Screen и Tmux.

Screen - это терминальный мультиплексор, который позволяет запустить один рабочий сеанс и в рамках него открыть любое количество окон (виртуальных терминалов). Процессы, запущенные в этой программе, будут выполняться, даже если их окна невидимы или программа прекратила работу.

Tmux - более современная альтернатива GNU Screen. Впрочем, возможности Tmux не имеют принципиальных отличий — в этой программе точно так же можно открывать множество окон в рамках одного сеанса. Задачи, запущенные в Tmux, продолжают выполняться, если терминал был закрыт.

Операторы объединения команд

Цепочка команд Linux означает объединение нескольких команд и выполнение их на основе оператора, используемого между ними. Цепочка команд в Linux — это то, когда вы пишете короткие скрипты оболочки в самой оболочке и выполняете их непосредственно из терминала.

Оператор точка с запятой (;) позволяет запускать несколько команд за один раз, и выполнение команды происходит последовательно.

command 1 ; command 2 ; command 3

Оператор AND (&&) будет выполнять вторую команду только в том случае, если при успешном выполнении первой команды.

Оператор OR (||) очень похож на оператор «**else**» в программировании. Вышеуказанный оператор позволяет вам выполнять вторую команду только в случае сбоя при выполнении первой команды, то есть скогда программа выполнена НЕ успешно». Если первая команда выполнена успешно, со статусом выхода 0, то вторая команда не будет выполнена.

Оператор комбинации команд {} объединяет две или более команды.

Например, проверьте, доступен ли каталог «bin»**,** и выведите соответствующий вывод.

[-d"bin"] || { echo Directory does not exist, creating directory now.; mkdir bin; } && echo Directory exists.

Оператор приоритета ()

Оператор позволяет выполнить команду в порядке приоритета.

Command_x1 && Command_x2 || Command_x3 && Command_x4

В приведенной выше псевдокоманде, что если Command_x1 завершится неудачно? Ни один из Command_x2, Command_x3, Command_x4 не будет выполнен. Если же мы используем оператор приоритета вот так:

(Command_x1 &&Command_x2) || (Command_x3 && Command_x4)

то, если Command_x1 завершается ошибкой, Command_x2 также завершается ошибкой, но все же выполнение Command_x3 и Command_x4 зависит от состояния выхода Command_x3.

Alias

Alias - это, по сути, ярлыки для команд Linux. Команда alias позволяет пользователю запускать любую команду или даже группу команд, в том числе с опциями, параметрами и файлами, вводом одного слова или даже символа. Это очень удобно во многих ситуациях. Чтобы посмотреть какие псевдонимы linux команд уже заданны в вашей системе просто выполните: alias

Команда покажет все alias команд linux определенные для текущего пользователя. Вывод очень сильно зависит от вашего дистрибутива. Общий синтаксис команды выглядит следующим образом:

```
$ alias имя="значение"
$ alias имя="команда аргумент1 аргумент2"
$ alias имя="/путь/к/исполняемому/файлу"
```

Вы можете создавать новые псевдонимы, просто выполняя эту команду в терминале. Но созданные таким образом алиасы linux будут работать только в этом терминале и только до его закрытия. Для примера создадим alias linux для такой часто используемой команды, как clear (очистить вывод терминала):

alias c='clear'

Теперь, чтобы очистить терминал достаточно выполнить: c

Удалить созданный alias можно с помощью команды unalias:

unalias c

При запуске терминала, каждый раз выполняется скрипт **~/.bashrc**, чтобы установить переменные окружения и подготовить оболочку. Таким образом добавив нужные строки в конец файла мы получим работающие alias linux в каждом терминале.

Список полезных алиасов.

Цветной вывод ls:

alias ls='ls --color=auto'

Показывать скрытые файлы и представлять вывод в виде списка:

alias ll='ls -la'

Показать только скрытые файлы:

alias l.='ls -d .* --color=auto'

Исправляем опечатку в cd:

alias cd..='cd ..'

Создавать дерево каталогов, если оно не существует:

alias mkdir='mkdir -pv'

Сделаем вывод mount читаемым:

alias mount='mount | column -t'

Сократим команды для экономии времени:

alias h='history'

alias j='jobs -l'

Посылать только пять запросов ping:

alias ping='ping -c 5'

Открытые порты:

alias ports='netstat -tulanp'

В семействе дистрибутивов Red Hat используется пакетный менеджер yum:

alias update='yum -y update'

Показать топ 10 процессов по использованию ЦПУ:

alias pscpu10='ps auxf |sort -nr -k 3 |head -10'

Переменные окружения

Окружение/среда (environment) - это набор пар ПЕРЕМЕННАЯ=ЗНАЧЕНИЕ, которые могут использоваться запускаемыми процессами.

Для того чтобы посмотреть свое окружение введите команду без аргументов:

env

В зависимости от конфигурации системы, вывод списка переменных окружения может занять несколько экранов, поэтому лучше использовать команду:

env |more

Переменные окружения могут формироваться как из заглавных, так и из строчных символов, однако исторически повелось именовать их в верхнем регистре.

Чтобы вывести на экран значение какой-нибудь переменной окружения, достаточно набрать echo $ИМЯ_ПЕРЕМЕННОЙ, например, просмотр домашней директории пользователя, хранящийся в переменной окружения HOME:

echo $HOME

Для перехода в домашнюю директорию можно использовать команду:

cd $HOME

Для установки значений переменной окружения введите команду:

VAR=value

unset вызывает уничтожение переменных оболочки

Некоторые зарезервированные переменные:

$DIRSTACK - содержимое вершины стека каталогов

$EDITOR - текстовый редактор по умолчанию

$EUID - Эффективный UID. Если вы использовали программу su для выполнения команд от другого пользователя, то эта переменная содержит UID этого пользователя, в то время как...

$UID - ...содержит реальный идентификатор, который устанавливается только при логине.

$FUNCNAME - имя текущей функции в скрипте.

$GROUPS - массив групп к которым принадлежит текущий пользователь

$HOME - домашний каталог пользователя

$HOSTNAME - ваш hostname

$HOSTTYPE - архитектура машины.

$LC_CTYPE - внутренняя переменная, котороя определяет кодировку символов

$OLDPWD - прежний рабочий каталог

$OSTYPE - тип ОС

$PATH - путь поиска программ

$PPID - идентификатор родительского процесса

$SECONDS - время работы скрипта(в сек.)

$# - общее количество параметров переданных скрипту

$* - все аргументы переданыне скрипту(выводятся в строку)

$@ - тоже самое, что и предыдущий, но параметры выводятся в столбик

$! - PID последнего запущенного в фоне процесса

$$ - PID самого скрипта

Файлы и каталоги

В linux существуют следующие типов файлов:
1.Обычный файл (regular file)
2.Каталог (directory)
3.Именованный канал (named pipe)
4.Символическая ссылка (soft link)
5.Специальный файл устройства (device file)
6.Сокет (socket)

По распечатке списка файлов командой ls -l можно определить тип файла. Первый символ сообщает нам о типе, а именно '-' обозначает обычный файл, 'd' — каталог, 'p' — именованный канал, 'l' — символическую ссылку, 'c' и 'b' — символьные и блочные файлы устройств, 's' — сокет.
Изначально каждый файл имел три параметра доступа:
Чтение - разрешает получать содержимое файла. Для каталога позволяет получить список файлов и каталогов, расположенных в нем;
Запись - разрешает записывать новые данные в файл или изменять существующие, а также позволяет создавать и изменять файлы и каталоги;
Выполнение - вы не можете выполнить программу, если у нее нет флага выполнения. Этот атрибут устанавливается для всех программ и скриптов, именно с помощью него система может понять, что этот файл нужно запускать как программу.
Каждый файл имеет три категории пользователей, для которых можно устанавливать различные сочетания прав доступа:
Владелец - набор прав для владельца файла, пользователя, который его создал или сейчас установлен его владельцем.
Группа - любая группа пользователей, существующая в системе и привязанная к файлу. Но это может быть только одна группа и обычно это группа владельца, хотя для файла можно назначить и другую группу.
Остальные - все пользователи, кроме владельца и пользователей, входящих в группу файла.
Только пользователь root может работать со всеми файлами независимо от их набора их полномочий.
Так же существуют дополнительные флаги, которые предоставляют специальные права доступа. Для того, чтобы позволить обычным пользователям выполнять программы от имени суперпользователя без знания его пароля была придумана такая вещь, как SUID и SGID биты. Рассмотрим эти полномочия подробнее.
SUID - если этот бит установлен, то при выполнении программы, id пользователя, от которого она запущена заменяется на id владельца файла. Фактически, это позволяет обычным пользователям запускать программы от имени суперпользователя;
SGID - этот флаг работает аналогичным образом, только разница в том, что пользователь считается членом группы, с которой связан файл, а не групп, к которым он действительно принадлежит. Если SGID флаг установлен на каталог, все файлы, созданные в нем, будут связаны с группой каталога, а не пользователя. Такое поведение используется для организации общих папок;
Sticky-bit - этот бит тоже используется для создания общих папок. Если он установлен, то пользователи могут только создавать, читать и выполнять файлы, но не могут удалять файлы, принадлежащие другим пользователям.
Теперь давайте рассмотрим как посмотреть и изменить права на файлы в Linux.
Для просмотра подробной информации обо всех флагах, в том числе специальных, нужно использовать команду ls с параметром -l. Все файлы из каталога будут выведены в виде списка, и там будут показаны все атрибуты и биты.

Для изменения прав доступа используется команда chmod (change mode).

Синтаксис:
chmod [options] mode[,mode] file1 [file2 ...]
Опции:
-R рекурсивное изменение прав доступа для каталогов и их содержимого
-f не выдавать сообщения об ошибке для файлов, чьи права не могут быть изменены.
-v подробно описывать действие или отсутствие действия для каждого файла.
Команда не изменяет права на символьные ссылки, но для каждой символьной ссылки, заданной в командной строке, изменяет права доступа связанного с ней файла. Команда игнорирует символьные ссылки, встречающиеся во время рекурсивной обработки каталогов.
Аргумент команды chmod, задающий разрешения, может быть записан в двух форматах: в числовом и в символьном.
В числовом формате права записываются одной строкой сразу для трёх типов пользователей: владельца файла (u), группы владельца (g) и всех прочих пользователей (o).
В числовом виде файлу или каталогу устанавливаются абсолютные права, в то же время в символьном виде можно установить отдельные права для разных типов пользователей.
Варианты записи прав пользователя

двоичная	восьмеричная	символьная	права на файл	права на каталог
000	0	`---`	нет	нет
001	1	`--x`	выполнение	чтение свойств файлов
010	2	`-w-`	запись	нет
011	3	`-wx`	запись и выполнение	всё, кроме получения имени файлов
100	4	`r--`	чтение	чтение имён файлов
101	5	`r-x`	чтение и выполнение	доступ на чтение файлов/их свойств
110	6	`rw-`	чтение и запись	чтение имён файлов
111	7	`rwx`	все права	все права

Помимо стандартных разрешений 'rwx', команда chmod осуществляет также управление битами SGID, SUID и T.
Для SUID вес — 4000, а для SGID — 2000. Данные атрибуты имеют смысл при установленном соответствующем бите исполнения и обозначаются при символьной записи буквой «s»: «rwsrwxrwx» и «rwxrwsrwx» соответственно.
sticky bit имеет вес 1000.
Пример: chmod 4555 {имяфайла} — все имеют право на чтение и выполнение, но запускаться файл на исполнение будет с правами владельца.
В соответствии с концепцией файловой системы POSIX если на каталоге стоит возможность записи (w), то файл внутри этого каталога можно будет удалить, даже если право на запись для него не установлено.

В символьном виде использование команды позволяет более гибко добавлять, устанавливать или убирать права на файлы или каталоги:
$ chmod [references][operator][modes] file ...
References определяют пользователей, которым будут меняться права. References определяются одной или несколькими буквами:

Reference	Class	Описание
`u`	user	Владелец файла
`g`	group	Пользователи, входящие в группу владельца файла
`o`	others	Остальные пользователи
`a`	all	Все пользователи (или `ugo`)

Operator определяет операцию, которую будет выполнять chmod:

Operator	Описание
`+`	добавить определённые права
`-`	удалить определённые права
`=`	установить определённые права

Modes определяет, какие именно права будут установлены, добавлены или удалены:

Mode	Name	Описание
`r`	read	чтение файла или содержимого каталога
`w`	write	запись в файл или в каталог
`x`	execute	выполнение файла или чтение содержимого каталога
`X`	special execute	выполнение, если файл является каталогом или уже имеет право на выполнение для какого-нибудь пользователя
`s`	setuid/gid	установленные атрибуты SUID или SGID позволяют запускать файл на выполнение с правами владельца файла или группы соответственно
`t`	sticky	устанавливая t-бит на каталог, мы меняем это правило таким образом, что удалить файл может только владелец этого файла

Пример, установить права «rwxr-xr-x» (0755) дл яфайла:
chmod u=rwx,g=rx,o=rx filename

Скрипты bash

Введение

В первой строке скрипта принято указывать интерпретатор, который будет выполнять данный скрипт. Делается это с помощью шебанга.

Шебанг (shebang, hashbang) — в программировании последовательность из двух символов: решётки и восклицательного знака ("#!") в начале файла скрипта.

Когда скрипт с шебангом выполняется как программа в Unix-подобных операционных системах, загрузчик программ рассматривает остаток строки после шебанга как имя файла программы-интерпретатора. Загрузчик запускает эту программу и передаёт ей в качестве параметра имя файла скрипта с шебангом. Указание интерпретатора в строке шебанга позволяет использовать файлы скриптов и данных как системные команды, скрывая детали реализации от пользователей и других программ, так как устраняется необходимость указывать файл интерпретатора в командной строке перед файлом скрипта. Строки с шебангом могут содержать дополнительные аргументы, которые передаются интерпретатору. Пример:

#!/bin/sh — выполнить файл с помощью sh (Bourne shell) или другой совместимой оболочки;

•#!/bin/csh -f — выполнить файл с помощью csh (C shell) или другой совместимой оболочки, с отключением выполнения .cshrc файла пользователя;

•#!/usr/bin/perl -T — выполнить файл при помощи Perl в режиме Taint checking;

•#!/usr/bin/env python2 — выполнить файл как программу на Python, используя env для получения пути к файлу интерпретатора.

В других строках скрипта символ решётки используется для обозначения комментариев, которые оболочка не обрабатывает.

Переменные

Переменные позволяют хранить в файле сценария информацию, например — результаты работы команд для использования их другими командами. Существуют два вида переменных, которые можно использовать в bash-скриптах: переменные среды (окружения) и пользовательские переменные.

О переменных окружения было сказано выше. Напомню, обращение к переменной делаем по имени со знаком $ в начале. Пример:

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

Если необходимо в команде echo напечатать знак доллара, то его необходимо экранировать с помощью управляющего символа - обратной косой черты:

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

В дополнение к переменным среды, bash-скрипты позволяют задавать и использовать в сценарии собственные (пользовательские) переменные. Подобные переменные хранят значение до тех пор, пока не завершится выполнение сценария. В отличие от большинства других языков программирования, Bash не производит разделения переменных по "типам". По сути, переменные Bash являются строковыми переменными, но, в зависимости от контекста, Bash допускает целочисленную арифметику с переменными. Определяющим фактором здесь служит содержимое переменных. Как и в случае с системными переменными, к пользовательским переменным можно обращаться, используя знак доллара:

```
#!/bin/bash
# testing variables
person="Ivan"
grade=15
echo "$person is a cool boy, he is in grade $grade"
```

Одна из 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:
currentdir=`pwd`
При втором подходе то же самое записывают так:
currentdir=$(pwd)

Объявление переменных с использованием **declare** и **typeset.**
Инструкции **declare** и **typeset** являются встроенными инструкциями (они абсолютно идентичны друг другу и являются синонимами) и предназначена для наложения ограничений на переменные (контроля над типами). Ключи инструкций declare/typeset
-r *readonly* (только для чтения)
declare -r var1
Это грубый эквивалент констант (const) в языке С. Попытка изменения таких переменных завершается сообщением об ошибке.
-i *integer*
declare -i number
# Сценарий интерпретирует переменную "number" как целое число.
number=3
echo "number = $number"   # number = 3
-a *array*
declare -a ind
Переменная ind объявляется массивом.
-f *functions*
declare -f
Инструкция declare -f, без аргументов, приводит к выводу списка ранее объявленных функций в сценарии. Инструкция declare -f function_name выводит имя функции function_name, если она была объявлена ранее.
Инструкция declare допускает совмещение объявления и присваивания значения переменной одновременно: declare -i var3=373

Математические операции. Внутри **((…))** вычисляются арифметические выражения и возвращается их результат. В простейшем случае, конструкция a=$((5 + 3)) присвоит переменной "a" значение выражения "5 + 3", т.е. 8. Но, кроме того, двойные круглые скобки позволяют работать с переменными в стиле языка С.

```
#!/bin/bash
# Работа с переменными в стиле языка С
(( a = 23 ))  # Присвоение переменной в стиле С, с обоих строн от "=" стоят пробелы.
echo "a (начальное значение) = $a"
(( a++ ))     # Пост-инкремент 'a', в стиле С.
echo "a (после a++) = $a"
(( a-- ))     # Пост-декремент 'a', в стиле С.
echo "a (после a--) = $a"
(( ++a ))     # Пред-инкремент 'a', в стиле С.
echo "a (после ++a) = $a"
(( --a ))     # Пред-декремент 'a', в стиле С.
echo "a (после --a) = $a"
(( t = a<45?7:11 ))   # Трехместный оператор в стиле языка С.
echo "If a < 45, then t = 7, else t = 11."
echo "t = $t "
```

Работа со строками

Bash поддерживает на удивление большое количество операций над строками.

Длина строки

${#string}

expr length $string

expr "$string" : '.*'

Пример:

```
stringZ=abcABC123ABCabc
echo ${#stringZ}                 # 15
echo `expr length $stringZ`      # 15
echo `expr "$stringZ" : '.*'`    # 15
```

Длина подстроки в строке (подсчет совпадающих символов ведется с начала строки)

expr match "$string" '$substring'

где *$substring* - регулярное выражение.

expr "$string" : '$substring'

где *$substring* - регулярное выражение.

Пример:

```
stringZ=abcABC123ABCabc
echo `expr match "$stringZ" 'abc[A-Z]*.2'`   # 8
echo `expr "$stringZ" : 'abc[A-Z]*.2'`       # 8
```

Поиск подстроки в строке expr index $string $substring

Номер позиции первого совпадения в $string с первым символом в $substring.

```
stringZ=abcABC123ABCabc
echo `expr index "$stringZ" C12`            # 6 - позиция символа C.
echo `expr index "$stringZ" 1c`             # 3
# символ 'c' (в #3 позиции) совпал раньше, чем '1'.
```

Извлечение подстроки

${string:position}

Извлекает подстроку из *$string*, начиная с позиции *$position*.

${string:position:length}

Извлекает *$length* символов из *$string*, начиная с позиции *$position*.

```
stringZ=abcABC123ABCabc
#       Индексация начинается с 0.

echo ${stringZ:0}                       # abcABC123ABCabc
echo ${stringZ:1}                       # bcABC123ABCabc
echo ${stringZ:7}                       # 23ABCabc
echo ${stringZ:7:3}                      # 23A

# Возможна индексация с "правой" стороны строки
# Круглые скобки или дополнительный пробел "экранируют" параметр позиции.
echo ${stringZ:(-4)}                     # Cabc
echo ${stringZ: -4}                      # Cabc
```

expr substr $string $position $length

Извлекает *$length* символов из *$string*, начиная с позиции *$position*.

```
stringZ=abcABC123ABCabc
#       Индексация начинается с 1.
```

```
echo `expr substr $stringZ 1 2`            # ab
echo `expr substr $stringZ 4 3`            # ABC
```

expr match "$string" '\($substring\)'
Находит и извлекает первое совпадение *$substring* в *$string*, где *$substring* — это регулярное выражение.
expr "$string" : '\($substring\)'
Находит и извлекает первое совпадение *$substring* в *$string*, где *$substring* — это регулярное выражение.

```
stringZ=abcABC123ABCabc
echo `expr match "$stringZ" '\(.[b-c]*[A-Z]..[0-9]\)'`   # abcABC1
echo `expr "$stringZ" : '\(.[b-c]*[A-Z]..[0-9]\)'`       # abcABC1
echo `expr "$stringZ" : '\(.......\)'`                   # abcABC1
# Все вышеприведенные операции дают один и тот же результат.
```

Аналогично, но поиск начинается с конца строки:

```
stringZ=abcABC123ABCabc
echo `expr match "$stringZ" '.*\([A-C][A-C][A-C][a-c]*\)'`    # ABCabc
echo `expr "$stringZ" : '.*\(......\)'`                       # ABCabc
```

${string#substring}
Удаление самой короткой, из найденных, подстроки *$substring* в строке *$string*. Поиск ведется с начала строки
${string##substring}
Удаление самой длинной, из найденных, подстроки *$substring* в строке *$string*. Поиск ведется с начала строки

```
stringZ=abcABC123ABCabc
echo ${stringZ#a*C}      # 123ABCabc
echo ${stringZ##a*C}     # abc
```

${string%substring}
Удаление самой короткой, из найденных, подстроки *$substring* в строке *$string*. Поиск ведется с конца строки
${string%%substring}
Удаление самой длинной, из найденных, подстроки *$substring* в строке *$string*. Поиск ведется с конца строки

```
stringZ=abcABC123ABCabc
echo ${stringZ%b*c}      # abcABC123ABCa
echo ${stringZ%%b*c}     # a
```

${string/substring/replacement}
Замещает первое вхождение *$substring* строкой *$replacement*.
${string//substring/replacement}
Замещает все вхождения *$substring* строкой *$replacement*.

```
stringZ=abcABC123ABCabc
echo ${stringZ/abc/xyz}          # xyzABC123ABCabc
echo ${stringZ//abc/xyz}         # xyzABC123ABCxyz
```

${string/#substring/replacement}
Подстановка строки *$replacement* вместо *$substring*. Поиск ведется с начала строки *$string*.

${string/%substring/replacement}
Подстановка строки *$replacement* вместо *$substring*. Поиск ведется с конца строки *$string*.

```
stringZ=abcABC123ABCabc
echo ${stringZ/#abc/XYZ}          # XYZABC123ABCabc
echo ${stringZ/%abc/XYZ}          # abcABC123ABCXYZ
```

Массивы

Новейшие версии Bash поддерживают одномерные массивы. Инициализация элементов массива может быть произведена в виде: **variable[xx]**. Можно явно объявить массив в сценарии, с помощью директивы declare: **declare -a variable**. Обращаться к отдельным элементам массива можно с помощью *фигурных скобок*, т.е.: **${variable[xx]}**. Массив целиком может быть определен путем заключения записей в круглые скобки:

```
arr=(Hello World)
```

или

```
arr[0]=Hello
arr[1]=World
```

Массивы не требуют, чтобы последовательность элементов в массиве была непрерывной. Некоторые элементы массива могут оставаться не инициализированными. "Дырки" в массиве не являются ошибкой.

```
area[11]=23
area[13]=37
area[51]=UFOs
```

Обращение к неинициализированным элементам дает пустую строку.

Обратиться к определенному элементу массива:

```
echo ${arr[0]} ${arr[1]}
```

Фигурные скобки нужны для предотвращения конфликтов при разворачивании полных путей к файлам. Кроме того, доступны следующие конструкции:

```
  ${arr[*]} # Все записи в массиве
  ${!arr[*]}# Все индексы в массиве
  ${#arr[*]}# Количество записей в массиве
  ${#arr[0]}# Длина первой записи (нумерация с нуля)
```

Сумма двух элементов массива, записанная в третий элемент

```
area[5]=`expr ${area[11]} + ${area[13]}`
echo "area[5] = area[11] + area[13]"
echo -n "area[5] = "
echo ${area[5]}
```

Индексация начинается с нуля (первый элемент массива имеет индекс [0], а не [1]).

Bash позволяет оперировать переменными, как массивами, даже если они не были явно объявлены таковыми.

```
string=abcABC123ABCabc
echo ${string[@]}            # abcABC123ABCabc
echo ${string[*]}            # abcABC123ABCabc
echo ${string[0]}            # abcABC123ABCabc
echo ${string[1]}            # Ничего не выводится!
                             # Почему?
echo ${#string[@]}       # 1
```

При работе с отдельными элементами массива можно использовать специфический синтаксис:

```
array2=( [0]="первый элемент" [1]="второй элемент" [3]="четвертый элемент" )
echo ${array2[0]}     # первый элемент
echo ${array2[1]}     # второй элемент
echo ${array2[2]}     #
                # Элемент неинициализирован, поэтому на экран ничего не выводится.
echo ${array2[3]}     # четвертый элемент
```

При работе с массивами, некоторые встроенные команды Bash имеют несколько иной смысл. Например, unset - удаляет отдельные элементы массива, или даже массив целиком.
Пример

```
#!/bin/bash
declare -a colors
# Допускается объявление массива без указания его размера.
echo "Введите ваши любимые цвета (разделяя их пробелами)."
read -a colors    # Введите хотя бы 3 цвета для демонстрации некоторых свойств массивов.
#  Специфический ключ команды 'read', позволяющий вводить несколько элементов массива.
echo
element_count=${#colors[@]}

# Получение количества элементов в массиве.
#     element_count=${#colors[*]} - дает тот же результат.
#
#  Переменная "@" позволяет "разбивать" строку в кавычках на отдельные слова
echo
# Способ вывода списка всех элементов массива.
  echo ${colors[@]}          # ${colors[*]} дает тот же результат.
echo
# Команда "unset" удаляет элементы из массива, или даже массив целиком.
unset 'colors[1]'             # Удаление 2-го элемента массива.
                        # Тот же эффект дает команда   colors[1]=
echo  ${colors[@]}          # Список всех элементов массива -- 2-й элемент отсутствует.

unset colors                # Удаление всего массива.
                        #  Тот же эффект имеют команды unset colors[*]
                        #+ и unset colors[@].
echo; echo -n "Массив цветов опустошен."
echo ${colors[@]}           # Список элементов массива пуст.
exit 0
```

Копирование массивов:

```
array2=( "${array1[@]}" )
```

или

```
array2="${array1[@]}"
```

Добавить элемент в массив:

```
array=( "${array[@]}" "новый элемент" )
```

Операция подстановки команд позволяет загружать содержимое текстовых файлов в массивы.

```
#!/bin/bash
filename=/sample_file
#           cat sample_file
#           1 a b c
#           2 d e fg
declare -a array1
array1=( `cat "$filename" | tr '\n' ' '`)  # Загрузка содержимого файла $filename в массив array1.
#         tr заменяет символов перевода строки на пробелы.
echo ${array1[@]}           # список элементов массива.
#                       1 a b c 2 d e fg
```

```
#  Каждое "слово", в текстовом файле, отделяемое от других пробелами
#  заносится в отдельный элемент массива.
element_count=${#array1[*]}
echo $element_count          # 8
```

Ветвление кода

Управление ходом исполнения - один из ключевых моментов структурной организации сценариев. Условные переходы являются средством, которое обеспечивает управление порядком исполнения команд. В Bash, для проверки условий, имеется команда **test**, различного вида скобочные операторы и условный оператор **if/then**.

Конструкции проверки условий

- Оператор **if/then** проверяет - является ли код завершения списка команд 0 (поскольку 0 означает "успех"), и если это так, то выполняет одну, или более, команд, следующие за словом then.
- Существует специальная команда – **[** (левая квадратная скобка). Она является синонимом команды **test**, и является встроенной командой (т.е. более эффективной, в смысле производительности). Эта команда воспринимает свои аргументы как выражение сравнения или как файловую проверку и возвращает код завершения в соответствии с результатами проверки (0 - истина, 1 - ложь).
- Начиная с версии 2.02, Bash предоставляет в распоряжение программиста конструкцию [[...]] расширенный вариант команды test, которая выполняет сравнение способом более знакомым программистам, пишущим на других языках программирования. Обратите внимание: [[-- это зарезервированное слово, а не команда.
  Bash исполняет **[[$a -lt $b]]** как один элемент, который имеет код возврата.
  Круглые скобки ((...)) и предложение let так же возвращают код 0, если результатом арифметического выражения является ненулевое значение. Таким образом, арифметические выражения могут участвовать в операциях сравнения.
- Условный оператор **if** проверяет код завершения любой команды, а не только результат выражения, заключенного в квадратные скобки.
- Оператор **if/then** допускает наличие вложенных проверок.

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 n1больше или равно n2.
n1 -gt n2 Возвращает истинное значение, если n1 больше n2.
n1 -le n2 Возвращает истинное значение, если n1меньше или равно n2.
n1 -lt n2 Возвращает истинное значение, если n1 меньше n2.
n1 -ne n2 Возвращает истинное значение, если n1не равно n2.

Пример

```
#!/bin/bash
val1=6
if [ $val1 -gt 5 ]
then
        echo "The test value $val1 is greater than 5"
else
        echo "The test value $val1 is not greater than 5"
fi
```

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.

-n str1 Возвращает истину, если длина str1 больше нуля.
-z str1Возвращает истину, если длина str1 равна нулю.
str1 < str2 меньше, в смысле величины ASCII-кодов if ["$a" \< "$b"]
str1 > str2 больше, в смысле величины ASCII-кодов if ["$a" \> "$b"]
Обратите внимание! Символы > и < необходимо экранировать внутри **[ ]**
Пример сравнения строк:

```
if [ -n "$string1" ]
then
        echo "Строка \"string1\" не пустая."
else
        echo "Строка \"string1\" пустая."
fi    # Внутри квадратных скобок заключайте строки в кавычки!
```

Проверки файлов
-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 file2Проверяет, старше ли file1, чем file2.
-O file Проверяет, существует ли файл, и является ли его владельцем текущий пользователь.
-G fileПроверяет, существует ли файл, и соответствует ли его идентификатор группы идентификатору группы текущего пользователя.

Оператор выбора **case** управляет ходом исполнения программы, в зависимости от начальных условий. Конструкция **case (in) / esac** эквивалентна конструкции **switch** в языке C/C++. Она позволяет выполнять тот или иной участок кода, в зависимости от результатов проверки условий. Она является, своего рода, краткой формой записи большого количества операторов if/then/else.
Каждая строка с условием должна завершаться закрывающей круглой скобкой).
Каждый блок команд, отрабатывающих по заданному условию, должен завершаться двумя символами точка-с-запятой ;;
Блок case должен завершаться ключевым словом **esac** (*case* записанное в обратном порядке).
Пример:

```
OS=$(uname -s)
case $OS in
Linux) echo Linux;;
AIX)   echo AIX;;
*)     echo other;;
esac
```

Циклы

Цикл - это блок команд, который исполняется многократно до тех пор, пока не будет выполнено условие выхода из цикла.

Циклы for **-** это одна из основных разновидностей циклов. И она значительно отличается от аналога в языке C.

```
for arg in [list]
do
команда(ы)...
done
```

На каждом проходе цикла, переменная-аргумент цикла *arg* последовательно, одно за другим, принимает значения из списка *list*

Элементы списка могут включать в себя шаблонные символы.

Есл ключевое слово **do** находится в одной строке со словом **for**, то после списка аргументов (перед do) необходимо ставить точку с запятой **for** *arg* in [*list*] ; do

Пример

```
#!/bin/bash
# Список планет.
for planet in Меркурий Венера Земля Марс Юпитер Сатурн Уран Нептун Плутон
do
  echo $planet
done
# Если 'список аргументов' заключить в кавычки, то он будет восприниматься как единственный аргумент .
```

В качестве списка, в цикле for, можно использовать переменную:

```
#!/bin/bash
FILES="/usr/sbin/privatepw
/usr/sbin/pwck
/usr/sbin/go500gw
/usr/bin/fakefile
/sbin/mkreiserfs
/sbin/ypbind"
# Список интересующих нас файлов. В список добавлен фиктивный файл /usr/bin/fakefile.
for file in $FILES
do
  if [ ! -e "$file" ]       # Проверка наличия файла.
  then
    echo "Файл $file не найден."; echo
    continue              # Переход к следующей итерации.
  fi
  ls -l $file | awk '{ print $8 "     размер: " $5 }'  # Печать 2 полей.
   echo
done
```

Оператор цикла **for** имеет и альтернативный синтаксис записи - очень похожий на синтаксис оператора for в языке C. Для этого используются двойные круглые скобки:

```
#!/bin/bash
LIMIT=10
for ((a=1; a <= LIMIT ; a++))  # Двойные круглые скобки и "LIMIT" без "$".
do
  echo -n "$a "
done
```

```
echo; echo
# Попробуем и C-шный оператор "запятая".
for ((a=1, b=1; a <= LIMIT ; a++, b++))  # Запятая разделяет две операции, которые
выполняются совместно.
do
  echo -n "$a-$b "
done
echo; echo
```

Оператор **while** проверяет условие перед началом каждой итерации и если условие истинно (если код возврата равен 0), то управление передается в тело цикла. В отличие от циклов for, циклы while используются в тех случаях, когда количество итераций заранее не известно.

while [*condition*]
do
  command...
done

Как и в случае с циклами for, при размещении ключевого слова **do** в одной строке с объявлением цикла, необходимо вставлять символ ";" перед **do: while** [*condition*] ; do

Пример

```
#!/bin/bash
var0=0
LIMIT=10
while [ "$var0" -lt "$LIMIT" ]
do
  echo -n "$var0 "        # -n подавляет перевод строки.
  var0=`expr $var0 + 1`   # допускается var0=$(($var0+1)).
done
echo
exit 0
```

Как и в случае с **for**, цикл **while** может быть записан в C-подобной нотации, с использованием двойных круглых скобок.

```
#!/bin/bash
LIMIT=10
((a = 1))
# Двойные скобки допускают наличие лишних пробелов в выражениях.
while (( a <= LIMIT ))   # В двойных скобках символ "$" перед переменными опускается.
do
  echo -n "$a "
  ((a += 1))   # let "a+=1"
  # Двойные скобки позволяют наращивание переменной в стиле языка C.
done
echo
exit 0
```

Оператор цикла **until** проверяет условие в начале каждой итерации, но в отличие от **while** итерация возможна только в том случае, если условие ложно.

until [*condition-is-true*]
do
  command...
done

Обратите внимание: оператор **until** проверяет условие завершения цикла ПЕРЕД очередной итерацией, а не после, как это принято в некоторых языках программирования.
Как и в случае с циклами for, при размещении ключевого слова **do** в одной строке с объявлением цикла, необходимо вставлять символ ";" перед **do: until** [*condition-is-true*] ; do
```
#!/bin/bash
count=0
until [ $count -gt 10 ]
do
        (( count++ ))
        echo $count
done
```

Функции

Функция - это подпрограмма, блок кода который реализует набор операций, своего рода "черный ящик", предназначенный для выполнения конкретной задачи. Функции могут использоваться везде, где имеются участки повторяющегося кода.

function *function_name* {
command...
}
или
function_name ()
{
command...
}

Вызов функции осуществляется простым указанием ее имени в тексте сценария. Функция должна быть объявлена раньше, чем ее можно будет использовать.

```
#!/bin/bash
funky ()
{
  echo "Это обычная функция."
}
funky # Вызов функции
exit 0
```

Допускается даже создание вложенных функций, хотя пользы от этого немного.

```
f1 ()
{
  f2 () # вложенная
  {
    echo "Функция \"f2\", вложенная в \"f1\"."
  }
}
f2  #  Вызывает сообщение об ошибке.
echo
f1  #  Ничего не происходит, простой вызов "f1", не означает автоматический вызов "f2".
f2  #  Теперь все нормально, вызов "f2" не приводит к появлению ошибки,
    #   поскольку функция "f2" была определена в процессе вызова "f1".
```

Объявление функции может размещаться в самых неожиданных местах. Можно подменить встроенную команду на свою функцию:

```
NO_EXIT=1
[[ $NO_EXIT -eq 1 ]] && exit() { true; }
# Если $NO_EXIT равна 1, то объявляется "exit ()".
# Тем самым, функция "exit" подменяет встроенную команду "exit".
exit  # Вызывается функция "exit ()", а не встроенная команда "exit".
```

Функции могут принимать входные аргументы и возвращать код завершения.

function_name $arg1 $arg2

Доступ к входным аргументам, в функциях, производится посредством позиционных параметров, т.е. $1, $2 и так далее. Пример

```
#!/bin/bash
DEFAULT=default                    # Значение аргумента по-умолчанию.
func2 () {
  if [ -z "$1" ]                   # Длина аргумента #1 равна нулю?
```

```
  then
    echo "-Аргумент #1 имеет нулевую длину.-"  # Или аргумент не был передан функции.
  else
    echo "-Аргумент #1: \"$1\".-"
  fi

  variable=${1-$DEFAULT} #Что делает подстановка параметра?
  echo "variable = $variable" #Она различает отсутствующий аргумент от "пустого" аргумента

  if [ "$2" ]
  then
    echo "-Аргумент #2: \"$2\".-"
  fi
  return 0
}
echo "Вызов функции без аргументов."
func2
echo "Вызов функции с \"пустым\" аргументом."
func2 ""
echo "Вызов функции с неинициализированным аргументом."
func2 "$uninitialized_param"
echo "Вызов функции с одним аргументом."
func2 first
echo "Вызов функции с двумя аргументами."
func2 first second
echo "Вызов функции с аргументами \"\" \"second\"."
func2 "" second       # Первый параметр "пустой"  и второй параметр -- ASCII-строка
exit 0
```

В отличие от других языков программирования, в сценариях на языке командной оболочки, в функции передаются аргументы по значению. Если имена переменных (которые фактически являются указателями) передаются функции в виде аргументов, то они интерпретируются как обычные строки символов и не могут быть разыменованы. Функции интерпретируют свои аргументы буквально.
Функции возвращают значение в виде кода завершения. Код завершения может быть задан явно, с помощью команды return, в противном случае будет возвращен код завершения последней команды в функции (0 - в случае успеха, иначе - ненулевой код ошибки). Код завершения в сценарии может быть получен через переменную $?
return завершает исполнение функции. Команда return может иметь необязательный аргумент типа *integer*, который возвращается в вызывающий сценарий как "код завершения" функции, это значение так же записывается в переменную $?
Наибольшее положительное целое число, которое может вернуть функция -- 255. Команда return очень тесно связана с понятием код завершения, что объясняет это специфическое ограничение. К счастью существуют различные способы преодоления этого ограничения. Например, функции могут возвращать большие отрицательные значения (имеются ввиду - большие по своему абсолютному значению). Используя эту особенность, можно обыграть возможность получения от функций большие положительные значения. Еще один способ - использовать глобальные переменные для хранения "возвращаемого значения".

```
#!/bin/bash
return_test ()
{
```

```
  return $1
}
return_test 27      # o.k.
echo $?             # Возвращено число 27.
return_test 255     # o.k.
echo $?             # Возвращено число 255.
return_test 257     # Ошибка!
echo $?             # Возвращено число 1.
return_test -151892    # o.k.
echo $?                # Возвращено число -151892
Return_Val=   # Глобальная переменная, которая хранит значение, возвращаемое функцией.
alt_return_test ()
{
  fvar=$1
  Return_Val=$fvar
  return   # Возвратить 0 (успешное завершение).
}
alt_return_test 1
echo $?                          # 0
echo "Функция вернула число $Return_Val" # 1
alt_return_test 255
echo "Функция вернула число $Return_Val" # 255
alt_return_test 257
echo "Функция вернула число $Return_Val" # 257
alt_return_test 25701
echo "Функция вернула число $Return_Val" #25701
```

Для случаев, когда функция должна возвращать строку или массив, используйте специальные переменные:

```
count_lines_in_etc_passwd()
{
  [[ -r /etc/passwd ]] && REPLY=$(echo $(wc -l < /etc/passwd))
  # Если файл /etc/passwd доступен на чтение,
  # то в переменную REPLY заносится число строк.
}

if count_lines_in_etc_passwd
then
  echo "В файле /etc/passwd найдено $REPLY строк."
else
  echo "Невозможно подсчитать число строк в файле /etc/passwd."
fi
```

Перенаправление ввода для функций. Функции — по сути блок кода, а это означает, что устройство stdin для функций может быть переопределено.

```
#!/bin/bash
# По логину пользователя получим его "настоящее имя" из /etc/passwd.
ARGCOUNT=1  # Ожидается один аргумент.
E_WRONGARGS=65
file=/etc/passwd
pattern=$1
```

```
if [ $# -ne "$ARGCOUNT" ]
then
  echo "Порядок использования: `basename $0` USERNAME"
  exit $E_WRONGARGS
fi

file_excerpt ()
{
while read line
do
  echo "$line" | grep $1 | awk -F":" '{ print $5 }'  # Указывет awk использовать ":" как
разделитель полей.
done
} <$file  # Подменить stdin для функции.
file_excerpt $pattern
exit 0
```

Допускается также перенаправление ввода для блока кода, заключенного в фигурные скобки, в пределах функции.
Вместо:

```
Function ()
{
 ...
 } < file
```

Пишем:

```
Function ()
{
  {
    ...
   } < file
}
```