

Справочный лист по Bash

Содержание

1. Справочный лист по Bash
2. Синтаксис
3. Основные структуры
 1. Составные команды
 1. Списки команд
 2. Выражения
 3. Циклы
 2. Встроенные
 1. Чайники
 2. Декларативный
 3. Ввод
 4. Вывод
 5. Выполнение
 6. Задания /процессы
 7. Условные обозначения и циклы
 8. Аргументы скрипта
4. Стримы
 1. Файловые дескрипторы
 2. Перенаправление
 3. Трубопровод
 4. Расширения
 5. Распространенные комбинации
5. Тесты
 1. Коды выхода
 1. Тестирование кода выхода
 2. Шаблоны
 1. Синтаксис глобуса
 3. Тестирование
6. Параметры
 1. Специальные параметры
 2. Операции с параметрами
 3. Массивы
 1. Создание массивов
 2. Использование массивов
7. Примеры: Базовые структуры
 1. Составные команды
 1. Списки команд
 2. Выражения
 3. Циклы
 2. Встроенные
 1. Чайники
 2. Декларативный
 3. Ввод
 4. Вывод
 5. Выполнение

Синтаксис

- *[слово]* **[пробел]** *[слово]*

Пробелы разделяют слова. В bash *слово* - это группа символов, которые принадлежат друг другу. Примерами являются имена команд и аргументы команд. Чтобы поместить пробелы внутри аргумента (или *слова*), заключите аргумент в одинарные или двойные кавычки (см. Следующие два пункта).

- `' [Строка в одинарных кавычках] '`

Отключает синтаксическое значение всех символов внутри строки.

Всякий раз, когда вам нужны литеральные строки в вашем коде, рекомендуется заключать их в одинарные кавычки, чтобы вы не рисковали случайно использовать символ, который также имеет синтаксическое значение для Bash .

- `" [Строка в двойных кавычках] "`

Отключает синтаксическое значение всех символов, кроме расширений внутри строки. Используйте эту форму вместо одинарных кавычек, если вам нужно развернуть параметр или команду подстановки в вашей строке.

Помните: Важно всегда заключать свои расширения (`"$var"` или `"$(command)"`) в двойные кавычки. Это, в свою очередь, безопасно отключит значение синтаксических символов, которые могут встречаться внутри расширенного результата.

- `[команда] ; [команда] [новая строка]`

Точки с запятой и новые строки отделяют синхронные команды друг от друга. Используйте точку с запятой *или* новую строку, чтобы закончить команду и начать новую. Первая команда будет выполняться синхронно, что означает, что Bash будет ждать ее завершения перед выполнением следующей команды.

- `[команда] & [команда]`

Один амперсанд завершает выполнение асинхронной команды.

Амперсанд выполняет то же самое, что точка с запятой или перевод строки, поскольку он указывает конец команды, но это заставляет Bash выполнять команду асинхронно. Это означает, что Bash запустит его в фоновом режиме и сразу же запустит следующую команду, не дожидаясь завершения первой. Только команда перед `&` выполняется асинхронно, и вы не должны ставить `;` после `&`, `&` заменяет `;` .

- `[команда] | [команда]`

Вертикальная линия или символ канала соединяет вывод одной команды с вводом следующей. Любые символы, передаваемые первой командой в `stdout`, будут прочитаны второй командой в `stdin`.

- `[команда] && [команда]`

Условие И вызывает выполнение второй команды только в том случае, если первая команда завершается и завершается успешно.

- `[команда] || [команда]`

Условие ИЛИ вызывает выполнение второй команды только в том случае, если первая команда завершается и завершается с кодом выхода с ошибкой (любым ненулевым кодом выхода).

Основные структуры

Смотрите `BashSheet#Examples:_Basic_Structures` для некоторых примеров синтаксиса ниже.

Составные команды

Составные команды - это операторы, которые могут выполнять несколько команд, но рассматриваются Bash как своего рода группа команд.

Списки команд

- [список команд] {; }

Выполните список команд в текущей оболочке, как если бы они были одной командой.

Группировка команд сама по себе не очень полезна. Однако это вступает в игру везде, где синтаксис Bash принимает только одну команду, в то время как вам нужно выполнить несколько. Например, вы можете захотеть передать вывод нескольких команд через канал на вход другой команды:

```
{cmd1; cmd2; } | cmd3
```

Или вы можете захотеть выполнить несколько команд после `||` operator:

```
rm-файл || { echo "Ошибка удаления, прерывание".; выход 1; }
```

Он также используется для тел функций. Технически, это также можно использовать для тел циклов, хотя это **недокументировано, не переносимо**, и мы обычно предпочитаем **делать ...; сделано** для этого):

```
для digit в 1 9 7; {echo "$digit"; } #
непереносимый, недокументированный,
неподдерживаемый
```

```
для цифры в 1 9 7; повторите "$ digit"; сделано
# предпочтительно
```

Примечание: Вам **нужен ;** перед закрытием `}` (или он должен быть на новой строке).

- ([список команд])

Выполните список команд в подоболочке.

Это точно то же самое, что и приведенная выше группировка команд, только команды выполняются в подоболочке. Любой код, который влияет на среду, такой как присвоение переменных, **cd**, **экспорт** и т. Д., Не влияет на среду основного скрипта, но заключен в квадратные скобки.

Примечание: Вам **не нужен ;** перед закрытием `)`.

Выражения

- (([арифметическое выражение]))

Вычисляет данное выражение в арифметическом контексте.

Это означает, что строки считаются именами целочисленных переменных, все операторы считаются арифметическими операторами (такими как **++**, **==**, **>**, **<=**, и т.д.) Вы всегда должны использовать это для выполнения тестов с числами!

- \$(([арифметическое выражение]))

Расширяет результат данного выражения в арифметическом контексте.

Этот синтаксис похож на предыдущий, но расширяется в результате расширения. Мы используем его внутри других команд, когда хотим, чтобы результат арифметического выражения стал частью другой команды.

- [[[тестовое выражение]]]

Вычисляет данное выражение как выражение, совместимое с ТЕСТОМ.

Поддерживаются все операторы **ТЕСТИРОВАНИЯ**, но вы также можете выполнить *сопоставление с шаблоном* глобуса и несколько других более сложных тестов. Хорошо отметить, что разделение слов **не** будет выполняться при расширении параметров без кавычек здесь. Вы всегда должны использовать это для выполнения тестов на строках и именах файлов!

Циклы

Если вы новичок в циклах или ищете более подробную информацию, объяснение и / или примеры их использования, прочитайте раздел BashGuide об условных циклах.

- **сделать** *[список команд]*; **ГОТОВО**

Это представляет собой фактический цикл, который используется следующими несколькими командами.

Список команд между **do** и **done** - это команды, которые будут выполняться на каждой итерации цикла.

- **для** *[имени]* **в** *[словах]*

Следующий цикл будет повторять каждое **СЛОВО** после ключевого слова **in**.

Команды цикла будут выполняться со значением переменной, обозначаемой **ИМЕНЕМ**, равным слову.

- **для** ((*[арифметическое выражение]*; *[арифметическое выражение]*; *[арифметическое выражение]*))

Следующий цикл будет выполняться до тех пор, пока второе **арифметическое выражение** остается истинным.

Первое **арифметическое выражение** будет выполнено перед запуском цикла. Третье **арифметическое выражение** будет выполняться после выполнения последней команды в каждой итерации.

- **в то время как** *[список команд]*

Следующий цикл будет повторяться до тех пор, пока последняя команда, выполненная в *списке команд*, не завершится успешно.

- **пока** *[список команд]*

Следующий цикл будет повторяться до тех пор, пока последняя команда, выполненная в *списке команд*, завершается неудачно ("сбой").

- **выберите** *[имя]* **в** *[словах]*

Следующий цикл будет повторяться вечно, позволяя пользователю выбирать между заданными словами.

Команды итерации выполняются с переменной, обозначаемой значением **name**, установленным на слово, выбранное пользователем. Естественно, вы можете использовать **break** для завершения этого цикла.

Встроенные

Встроенные команды - это команды, которые выполняют определенную функцию, которая была скомпилирована в Bash. Понятно, что они также являются единственными типами команд (кроме приведенных выше), которые могут изменять среду оболочки Bash.

Чайники

- **true** (или **:**): эти команды вообще ничего не делают.
Это *NOP*, которые всегда возвращаются успешно.
- **false**: то же, что и выше, за исключением того, что команда всегда "завершается ошибкой".
Он возвращает код выхода **1**, указывающий на сбой.

Декларативный

- **ПСЕВДОНИМ**: устанавливает псевдоним **Bash** или печатает псевдоним **bash** с заданным именем.
Псевдонимы заменяют слово в начале команды на что-то другое. Они работают только в интерактивных оболочках (не в скриптах).
- **ОБЪЯВИТЬ** (или **ВВЕСТИ**): присвоить значение переменной.
Каждый аргумент представляет собой новое присвоение переменной. Часть каждого аргумента перед знаком равенства - это имя переменной, а после идут данные переменной. Параметры для объявления могут использоваться для переключения специальных флагов переменных (например, **_read-only** / **_export** / **_integer** / **_array**).
- **ЭКСПОРТ**: экспортируйте данную переменную в среду, чтобы дочерние процессы наследовали ее.
Это то же самое, что и **declare -x**. Помните, что для дочернего процесса переменная не совпадает с той, которую вы экспортировали. Он просто содержит те же данные. Это означает, что вы не можете изменять переменные данные и ожидать, что они также изменятся в родительском процессе.
- **local**: объявить переменную, область действия которой ограничена текущей функцией.
Как только функция завершает работу, переменная исчезает. Присвоение ему в функции также не изменяет глобальную переменную с тем же именем, если она существует. Те же параметры, что и в **declare**, могут быть переданы в **local**.
- **ТИП**: показывает тип имени команды, указанного в качестве аргумента.
Тип может быть любым: псевдоним, ключевое слово, функция, встроенный или файл.


Ввод

- **ЧТЕНИЕ**: прочитайте строку (если опция **-d** не используется для изменения разделителя с новой строки на что-то другое) и поместите ее в переменные, обозначенные аргументами, заданными для **ЧТЕНИЯ**.
Если задано более одного имени переменной, разделите строку, используя символы в IFS в качестве разделителей. Если задано меньше имен переменных, чем разделенных фрагментов в строке, последняя переменная получает все данные, оставшиеся неразделенными.

Вывод

- **echo**: выводите каждый аргумент, заданный **echo**, в одну строку, разделенную одним пробелом.
Первыми аргументами могут быть параметры, которые переключают специальное поведение (например, не переводить строку в конце / оценивать последовательности `scare`).
- **printf**: используйте первый аргумент в качестве спецификатора формата для вывода других аргументов.
Смотрите справку `printf`.
- **pwd**: выведите абсолютный путь к текущему рабочему каталогу.
Вы можете использовать опцию `-P`, чтобы заставить `pwd` разрешать любые символические ссылки в имени пути.

Выполнение

- **cd**: изменяет текущий каталог на указанный путь.
Если путь не начинается с косой черты, он относится к текущему каталогу.
- **КОМАНДА**: запустите первый аргумент как команду.
Это говорит Bash пропустить поиск псевдонима, функции или ключевого слова по этому имени; и вместо этого предположим, что имя команды является встроенным или программой в **PATH**.
- **coproc**: запуск команды или составной команды в качестве совместного процесса.
Работает в `bg`, настраивая каналы для связи. Посмотреть  <http://wiki.bash-hackers.org/syntax/keywords/coproc> за подробностями.
- **SOURCE** или `.`: Заставляет Bash считывать имя файла, указанное в качестве первого аргумента, и выполнять его содержимое в текущей оболочке.
Это похоже на **ВКЛЮЧЕНИЕ** в другие языки. Если в исходном коде указано больше аргументов, чем просто имя файла, эти аргументы устанавливаются в качестве позиционных параметров во время выполнения исходного кода. Если в исходном имени файла нет косой черты, для него выполняется поиск **PATH**.
- **exes**: Запустите команду, указанную в качестве первого аргумента, и замените на нее текущую оболочку.
Другие аргументы передаются команде в качестве ее аргументов. Если для `exes` не задано никаких аргументов, но вы указываете *перенаправления* в команде `exes`, перенаправления будут применены к текущей оболочке.
- **ВЫХОД**: завершите выполнение текущего скрипта.
Если задан аргумент, это статус завершения текущего скрипта (целое число от 0 до 255).
- **ВЫХОД** из системы: завершение выполнения оболочки входа в систему.
- **ВОЗВРАТ**: завершение выполнения текущей функции.
Статус выхода может быть указан точно так же, как при встроенном выходе.
- **ulimit**: измените ограничения ресурсов текущего процесса оболочки.
Эти ограничения наследуются дочерними процессами.

Задания /процессы

- **ЗАДАНИЯ:** перечислите активные задания текущей оболочки.
- **bg:** Отправьте предыдущее задание (или задание, обозначенное данным аргументом) для запуска в фоновом режиме.
Оболочка продолжает выполняться во время выполнения задания. Ввод оболочки обрабатывается сам по себе, а не задание.
- **fg:** отправить предыдущее задание (или задание, обозначенное данным аргументом) для запуска на переднем плане.
Оболочка ожидает завершения задания, и задание может получать входные данные от оболочки.
- **УБИТЬ:** отправить сигнал (3) процессу или заданию.
В качестве аргумента укажите идентификатор процесса процесса или спецификацию задания, которому вы хотите отправить сигнал.
- **ЛОВУШКА:** обрабатывает сигнал (3), отправленный в текущую оболочку.
Код, который находится в первом аргументе, выполняется всякий раз, когда принимается сигнал, обозначаемый любым из других аргументов для перехвата.
- **ПРИОСТАНОВИТЬ:** останавливает выполнение текущей оболочки до тех пор, пока она не получит сигнал *SIGCONT*.
Это очень похоже на то, что происходит, когда оболочка получает сигнал *SIGSTOP*.
- **ЖДАТЬ:** останавливает выполнение текущей командной строки до завершения активных заданий.
В аргументах вы можете указать, какие задания (по *jobspec*) или процессы (по *PID*) следует ожидать.

Условные обозначения и циклы

- **break:** выход из текущего цикла.
Если активно более одного цикла, выделите последний объявленный.
Когда число задается в качестве аргумента для прерывания, прерывайте числовые циклы, начиная с последнего объявленного.
- **ПРОДОЛЖЕНИЕ:** пропустите код, который остался в текущем цикле, и запустите новую итерацию этого цикла.
Как и в случае с **break**, может быть задано число, чтобы пропустить больше циклов.

Аргументы скрипта

- **set:** команда **set** обычно устанавливает различные параметры оболочки, но также может устанавливать позиционные параметры.
Параметры оболочки - это параметры, которые могут быть переданы в оболочку, такие как **bash -x** или **bash -e**. **set** переключает параметры оболочки следующим образом: **set -x**, **set +x**, **set -e**, ... Позиционные параметры - это параметры, содержащие аргументы, которые были переданы скрипту или оболочке, такие как **bash myscript -foo /bar**. **set** присваивает позиционные параметры следующим образом: **set -- -foo /bar**.

- **СДВИГ: перемещает значения всех позиционных параметров на один параметр назад.**

Таким образом, значения, которые были в \$ 1, отбрасываются, значения из \$ 2 переходят в \$ 1, значения из \$ 3 переходят в \$ 2 и так далее. Вы можете указать аргумент для shift, который является целым числом, указывающим, сколько раз повторять этот сдвиг.

- **getopts: помещает параметр, указанный в аргументах, в переменную.**

getopts использует первый аргумент в качестве спецификации того, какие параметры следует искать в аргументах. Затем он принимает первый вариант в аргументах, который упоминается в спецификации этого параметра (или следующий вариант, если **getopts** был запущен ранее), и помещает этот параметр в переменную, обозначенную именем во втором аргументе **getopts** . Эта команда практически всегда используется в цикле:

```
в то время как getopts abc opt
делает
case $opt в
a) ...;;
b) ...;;
c) ...;;
esac
выполнен
```

Таким образом, анализируются все параметры в аргументах, и когда они имеют значение **-a**, **-b** или **-c**, выполняется соответствующий код в инструкции **case**. Следующий краткий стиль также действителен для указания нескольких параметров в аргументах, которые анализирует **getopts: -ac** .

Стримы

Если вы новичок в обработке ввода и вывода в **bash** или ищете больше примеров, деталей и / или объяснений, прочитайте **BashGuide/ InputAndOutput** .

Bash - отличный инструмент для управления потоками данных между процессами. Благодаря отличным операторам для подключения файловых дескрипторов, мы берем данные практически из любого места и отправляем их практически куда угодно. Понимание потоков и того, как вы управляете ими в **Bash**, является ключом к безграничности возможностей **Bash**.

Файловые дескрипторы

Дескриптор файла подобен дороге между файлом и процессом. Он используется процессом для отправки данных в файл или чтения данных из файла. Процесс может иметь множество файловых дескрипторов, но по умолчанию для стандартных задач используются три.

- **0: стандартный ввод**

Это то место, откуда процессы обычно считывают информацию. Например, процесс может запросить у вас ваше имя, после того, как вы его введете, информация считывается через **FD 0**.

- **1: Стандартный вывод**

Это то место, куда процессы обычно записывают все свои выходные данные. Например, процесс может объяснить, что он делает, или вывести результат операции.

- **2: Стандартная ошибка**

Это то место, куда процессы обычно записывают свои сообщения об ошибках. Например, процесс может жаловаться на неверный ввод или недопустимые аргументы.

Перенаправление

- `[файл] > [команда], [команда] [n] > [файл], [команда] 2> [файл]`

Перенаправление файлов: оператор > перенаправляет стандартный вывод команды (или FD N) в заданный файл.

Это означает, что все стандартные выходные данные, сгенерированные командой, будут записаны в файл.

Вы можете дополнительно указать число перед оператором >. Если не указано, число по умолчанию равно 1. Число указывает, с какого файлового дескриптора процесса перенаправлять выходные данные.

Примечание: файл будет усечен (очищен) перед запуском команды!

- `[fd] >&[команда], [команда] [fd] >&[fd], [команда] 2>&1`

Дублирование файловых дескрипторов: оператор X>&Y копирует цель FD Y в FD X.

В последнем примере текущая цель FD 1 (стандартный вывод команды) копируется в FD 2 (стандартный вывод команды).

В результате, когда команда записывает в свой stderr, байты окажутся в том же месте, в котором они были бы, если бы они были записаны в стандартный вывод команды.

- `[файл] >> [команда], [команда] [n] >> [файл]`

Перенаправление файлов: оператор >> перенаправляет стандартный вывод команды в данный файл, добавляя к нему.

Это означает, что все стандартные выходные данные, сгенерированные командой, будут добавлены в конец файла.

Примечание: файл не усечен. Вывод просто добавляется в конец.

- `[файл] < [команда], [команда] [n] < [файл]`

Перенаправление файла: оператор < перенаправляет данный файл на стандартный ввод команды.

Вы можете дополнительно указать число перед оператором <. Если не указано, число по умолчанию равно 0. Число указывает, в какой файловый дескриптор процесса перенаправлять ввод.

- `[команда] &> [файл]`

Перенаправление файлов: оператор &> перенаправляет стандартный вывод команды и стандартную ошибку в данный файл.

Это означает, что все стандартные выходные данные и ошибки, сгенерированные командой, будут записаны в файл.

- `[команда] &>> [файл] (Bash 4+)`

Перенаправление файлов: оператор &>> перенаправляет стандартный вывод команды и стандартную ошибку в данный файл, добавляя к нему.

Это означает, что все стандартные выходные данные и ошибки, сгенерированные командой, будут добавлены в конец файла.

- `[команда] <<< "[строка данных]"`

Здесь-Строка: перенаправляет единственную строку данных на стандартный ввод команды.

Это хороший способ отправить одну строку текста на ввод команды.

Обратите внимание, что, поскольку строка заключена в кавычки, вы также можете безопасно вводить в нее новые строки и превращать ее в несколько строк данных.

- `[команда] <<[СЛОВО]`
`[строки данных]`
`[СЛОВО]`

Здесь-Document: перенаправляет строки данных на стандартный ввод команды.

Это хороший способ отправки нескольких строк текста на ввод команды.

Примечание: Слово после <

Примечание: Вы можете "заключить в кавычки" слово после <<. Если вы это сделаете, все в строках данных, которые выглядят как расширения, не будут расширены с помощью bash.

Примечание: Если после << (<<- [WORD]) добавляется дефис (-), то все начальные символы табуляции игнорируются в каждой строке.

Трубопровод

- `[команда] | [другая команда]`

Канал: оператор | соединяет стандартный вывод первой команды со стандартным вводом второй команды.

В результате вторая команда будет считывать свои данные из выходных данных первой команды.

- `[команда] | & [другая команда] (Bash 4+)`

Канал: оператор | & соединяет стандартный вывод первой команды и стандартную ошибку со стандартным вводом второй команды.

В результате вторая команда будет считывать свои данные из выходных данных первой команды вместе с ошибками.

Расширения

- `) "[список команд]" $([command], [command] "` [список команд] "``

Подстановка команд: захватывает выходные данные команды и разворачивает их в строке.

Мы используем подстановку команд только внутри других команд, когда хотим, чтобы вывод одной команды стал частью другого оператора.

Древний и опрометчивый альтернативный синтаксис для замены команд - это обратная кавычка: ``command``. Этот синтаксис имеет тот же результат, но он плохо вложен и его слишком легко спутать с кавычками (обратные

кавычки не имеют ничего общего с цитированием!). Избегайте этого синтаксиса и замените его на **`$ (command)`**, когда найдете его.

Это все равно, что запустить вторую команду, взять ее выходные данные и вставить их в первую команду, куда вы бы поместили **`$ (. . .)`**.

- **`[команда] < ([список команд])`**

Замена процесса: оператор `< (. . .)` расширяется в новый файл, созданный `bash`, который содержит выходные данные другой команды. Файл предоставляет любому, кто читает из него, выходные данные второй команды.

Это похоже на перенаправление вывода второй команды в файл с именем **`foo`**, а затем выполнение первой команды и предоставление ей **`foo`** в качестве *аргумента*. Только в одном операторе, и **`foo`** создается и очищается автоматически после этого.

ПРИМЕЧАНИЕ: НЕ ПУТАЙТЕ ЭТО С ПЕРЕНАПРАВЛЕНИЕМ ФАЙЛОВ. `<` здесь **не** означает *перенаправление файла*. Это просто символ, который является частью оператора `< (. . .)`! Этот оператор **не** выполняет никакого перенаправления. Он *просто* расширяется в *путь к файлу*.

- **`[команда] > ([список команд])`**

Замена процесса: оператор `> (. . .)` расширяется в новый файл, созданный `bash`, который отправляет данные, которые вы записываете в него, на *стандартный ввод* второй команды.

Когда первая команда записывает что-то в файл, эти данные передаются второй команде в качестве входных данных.

Это все равно, что перенаправить файл с именем **`foo`** на ввод второй команды, а затем запустить первую команду, указав ей **`foo`** в качестве *аргумента*. Только в одном операторе, и **`foo`** автоматически создается и очищается после этого

Распространенные комбинации

- **`[команда] << ([список команд])`**

Перенаправление файлов и замена процессов: `< (. . .)` заменяется файлом, созданным `bash`, а оператор `<` принимает этот новый файл и перенаправляет его на *стандартный ввод* команды.

Это почти то же самое, что передача второй команды в первую (**`secondcommand | firstcommand`**), но первая команда не имеет вложенной оболочки, как в канале. В основном он используется, когда нам нужна первая команда для изменения среды оболочки (что невозможно, если она имеет подоболочку). Например, чтение в переменную:

прочитайте `var << (файл grep foo)`. Это не сработало бы: **`grep foo file | read var`**, потому что **`var`** будет назначен только в его крошечной подболочке и исчезнет, как только канал будет завершен.

Примечание: Не забывайте о *пробелах* между оператором `<` и оператором `< (. . .)`. Если вы забудете это пространство и превратите его в **`<< (. . .)`**, это приведет к ошибкам!

Примечание: Это создает (и очищает) временный файл, зависящий от конкретной реализации (обычно, FIFO), который передает вывод из второй

команды в первую.

- `[команда] <<< "$([список команд])"`

Здесь -Подстановка строк и команд: `$ (. . .)` заменяется выводом второй команды, и оператор `<<<` отправляет эту строку на **стандартный ввод** первой команды.

Это почти то же самое, что и команда выше, с небольшим побочным эффектом, заключающимся в том, что `$ ()` удаляет все конечные новые строки из выходных данных и `<<<` добавляет к ним одну обратно.

Примечание: сначала считывается весь вывод из второй команды, сохраняя его в памяти. Когда вторая команда завершена, первая вызывается с выводом. В зависимости от объема вывода это может занять больше памяти.

Тесты

Если вы новичок в `bash`, не до конца понимаете, что такое команды и коды выхода, или хотите получить некоторые подробности, пояснения и / или примеры по тестированию команд, строк или файлов, прочитайте раздел `BashGuide` о тестах и условных обозначениях.

Коды выхода

Код выхода или статус выхода - это 8-разрядное целое число без знака, возвращаемое командой, которое указывает, как прошло ее выполнение. Согласовано, что код выхода `0` указывает на то, что команда успешно выполнила то, что должна была сделать. Любой другой код выхода указывает на то, что что-то пошло не так. Приложения могут сами выбирать, какое число указывает на то, что пошло не так; поэтому обратитесь к руководству приложения, чтобы узнать, что означает код выхода приложения.

Тестирование кода выхода

- `если [список команд]; затем [список команд]; elif [список команд]; затем [список команд]; else [список команд]; fi`

Команда **`if`** проверяет, имела ли последняя команда в первом списке команд код выхода `0`.

Если это так, он выполняет список команд, который следует за **`then`**. Если нет, то следующий **`elif`** выполняется таким же образом. Если нет **`elif`**-ов, выполняется список команд, следующий за **`else`**, если нет инструкции **`else`**. Подводя итог, **`if`** выполняет список * команд *. Он проверяет код выхода. В случае успеха выполняются команды **`then`**. части **`elif`** и **`else`** являются необязательными. Часть **`fi`** завершает весь блок **`if`** (не забывайте об этом!).

- `[список команд] пока и пока [список команд]`

Выполните следующую итерацию в зависимости от кода выхода последней команды в списке команд.

Мы обсуждали их раньше, но стоит повторить их в этом разделе, поскольку они фактически выполняют то же самое, что и оператор **`if`**; за

исключением того, что они выполняют цикл до тех пор, пока тестируемый код выхода соответственно равен 0 или не равен 0.

Шаблоны

Bash знает два типа шаблонов. *Шаблоны глобусов являются наиболее важными, наиболее часто используемыми и наиболее удобочитаемыми.* Более поздние версии Bash также поддерживают "модные" регулярные выражения. Однако не рекомендуется использовать регулярные выражения в скриптах, если у вас нет абсолютно никакого другого выбора или преимущества их использования намного больше, чем при использовании globs. Вообще говоря, если вам нужно регулярное выражение, вы будете использовать `awk(1)`, `sed(1)` или `grep(1)` вместо Bash.

Если вы новичок в bash или хотите получить некоторые подробности, пояснения и / или примеры по сопоставлению шаблонов, прочитайте раздел BashGuide о шаблонах.

Синтаксис глобуса

- **?: Знак вопроса соответствует любому символу.**
Это один единственный символ.
- ***: Звезда соответствует любому количеству любых символов.**
Это ноль или более любых символов.
- **]...[: Это соответствует * одному из * любых символов внутри фигурных скобок.**

Это один символ, который упоминается внутри фигурных скобок.

- **[abc]: соответствует либо a, b, либо c, но не строке abc.**
- **[a - c]: Тире указывает Bash использовать диапазон.**

Соответствует любому символу между (включительно) a и c. Так что это то же самое, что и в примере чуть выше.

- **[!a - c] или [^a - c]: The ! или ^ в начале указывает Bash инвертировать совпадение.**

Соответствует любому символу, который *не* a, b или c. Это означает любую другую букву, но * также * число, точку, запятую или любой другой символ, который вы можете придумать.

- **[[:digit:]]: Синтаксис [:class:] указывает Bash использовать символьный класс.**

Классы символов - это группы символов, которые предопределены и названы для удобства. Вы можете использовать следующие классы:

`alnum`, альфа, `ascii`, пробел, `cntrl`, цифра, график, нижний, печать, точка, пробел, верхний, слово, `xdigit`

Тестирование

- `esac ; [список команд]) [глобальный шаблон] ; [список команд]) [глобальный шаблон]` в [строковом] случае:

Использование **CASE** удобно, если вы хотите протестировать определенную строку, которая может соответствовать любому из нескольких разных шаблонов глобусов.

Будет выполнен список команд, следующий за шаблоном ** first * glob*, который соответствует вашей строке. Вы можете указать столько комбинаций шаблонов глобусов и списков команд, сколько вам нужно.

- `]] "[строка]"="[строка]" [[, [[[строка] = [шаблон глобуса]]]` или `[[[строка] =~ [регулярное выражение]]]`:

Проверьте, соответствует ли левая **СТРОКА** правой **СТРОКЕ** (если она в кавычках), **ГЛОБУСУ** (если без кавычек и с использованием `=`) или **РЕГУЛЯРНОМУ ВЫРАЖЕНИЮ** (если без кавычек и с использованием `=~`).

[и **test** - это команды, которые вы часто видите в **sh**-скриптах для выполнения этих тестов. [[может делать все эти вещи (но лучше и безопаснее), а также предоставляет вам сопоставление с шаблоном.

НЕ используйте [или не **тестируйте** в коде **bash**. Всегда используйте [[вместо. У него много преимуществ и никаких недостатков.

НЕ используйте [[для выполнения тестов над командами или числовыми операциями. Для первого используйте **if**, а для второго используйте **((**.

[[можно выполнить множество других тестов, например, для файлов. Все типы тестов, которые он может выполнить для вас, см. в разделе **help test**.

- `)) [арифметическое выражение] ((:`

Это ключевое слово специализируется на выполнении числовых тестов и операций.

См. `ArithmeticExpression`

Параметры

Параметры - это то, что **Bash** использует для хранения данных вашего скрипта. Существуют *специальные параметры* и *переменные*.

Любые параметры, которые вы создаете, будут переменными, поскольку специальные параметры - это параметры, доступные только для чтения, управляемые **Bash**.

Рекомендуется использовать имена в нижнем регистре для ваших собственных параметров, чтобы не путать их с именами переменных в верхнем регистре, используемыми внутренними переменными **Bash** и переменными среды. Также рекомендуется использовать понятные и прозрачные имена для ваших переменных.

Избегайте **x**, **i**, **t**, **tmp**, **foo** и т. Д. Вместо этого используйте имя переменной для описания типа данных, которые должна хранить переменная.

Также важно, чтобы вы понимали необходимость цитирования. Вообще говоря, всякий раз, когда вы используете параметр, вы должны заключать его в кавычки:

echo "Файл находится в: \$filePath". Если вы этого не сделаете, **bash** разорвет содержимое вашего параметра на биты, удалит из него все пробелы и передаст биты в качестве аргументов команде. Да, **Bash** искажает ваши расширения параметров по умолчанию - это называется *разделением слов* - поэтому используйте кавычки, чтобы предотвратить это.

Исключение составляют *ключевые слова* и *назначение*. После `myvar=` и внутри `[`, `case` и т. Д. кавычки вам не *нужны*, но они также не повредят - так что, если вы не уверены: цитируйте!

И последнее, но не менее важное: помните, что параметры - это *структуры данных* bash. Они хранят данные вашего приложения. Они **НЕ** должны использоваться для хранения логики вашего приложения. Поэтому, хотя многие плохо написанные скрипты могут использовать такие вещи, как `GREP=/usr/bin/grep` или `command='mplayer -vo x11 -ao alsa'`, вы **НЕ** должны этого делать. Основная причина в том, что вы не можете сделать это полностью правильно, безопасно и доступно для чтения / обслуживания. Если вы хотите избежать повторного ввода одной и той же команды несколько раз или создать одно место для управления командной строкой команды, вместо этого используйте *функцию*. Не параметры.

Специальные параметры

Если вы новичок в bash или хотите получить некоторые подробности, пояснения и / или примеры параметров, ознакомьтесь с разделом BashGuide, посвященным специальным параметрам.

- **1, 2, ...:** **Позиционные параметры** - это аргументы, которые были переданы вашему скрипту или вашей функции.

Когда ваш скрипт запускается с `./script foo bar`, `"$ 1"` станет `"foo"`, а `"$ 2"` станет `"bar"`. Скрипт, запущенный как `./script "foo bar"` хаббл расширит `"$ 1"` как `"foo bar"` и `"$ 2"` как `"хаббл"`.

- *****: При расширении он равен единственной строке, которая объединяет все позиционные параметры, используя первый символ IFS для их разделения (- по умолчанию это пробел).

Короче говоря, `"$ *"` - это то же самое, что и `"$ 1x $ 2x $ 3x $ 4x ..."`, где `x` - первый символ IFS.

С IFS по умолчанию это станет простым `"$1 $2 $3 $4 ..."`.

- **@:** Это расширится на несколько аргументов: каждый установленный позиционный параметр будет расширен как один аргумент.

Итак, по сути, `"$ @"` - это то же самое, что `"$1" "$2" "$3" ...`, все они указаны отдельно.

ПРИМЕЧАНИЕ: вы всегда должны использовать `"$ @"` перед `"$ *"`, потому что `"$ @"` сохраняет тот факт, что каждый аргумент является его отдельной сущностью. С помощью `"$ *"` вы потеряете эти данные! `"$*"` действительно полезен, только если вы хотите разделить свои аргументы чем-то, что не является пробелом; например, запятой: `(IFS=,; echo "Вы запустили скрипт с аргументами: $ *")` - выведете все свои аргументы, разделяя их запятыми.

- **#:** Этот параметр превращается в число, представляющее, сколько позиционных параметров задано.

Скрипт, выполняемый с 5 аргументами, будет иметь значение `"$ #"` до 5. В основном это полезно только для проверки того, были ли заданы какие-

либо аргументы: `if ((! $ #));` затем `echo "Аргументы не были переданы". > & 2;` выход `1;`
`fi`

- **?:** Расширяется в код выхода ранее выполненной команды переднего плана.

Мы используем `$?` в основном, если мы хотим использовать код выхода команды в нескольких местах; или для проверки его на множество возможных значений в операторе `case`.

- **-:** Параметр `dash` расширяется до флагов параметров, которые в настоящее время установлены в процессе `Bash`.

См. `set` для объяснения того, что такое флаги опций, какие существуют и что они означают.

- **\$:** параметр `dollar` расширяется до идентификатора *процесса* процесса `Bash`.

Удобно в основном для создания файла PID для вашего процесса `bash` (`echo "$$" > /var/run/foo.pid`); так что вы можете легко завершить его, например, из другого процесса `bash`.

- **!:** Расширяется до идентификатора *процесса* самой последней фоновой команды.

Используйте это для управления фоновыми командами из вашего Bash-скрипта: `foo ./bar & pid=$!; sleep 10; kill "$pid";` подождите `"$pid"`

- **_:** Расширение аргумента подчеркивания дает вам последний аргумент последней выполненной вами команды.

Этот используется в основном в интерактивных оболочках, чтобы немного сократить ввод текста: `mkdir -p /foo /bar && mv myfile "$_".`

Операции с параметрами

Если вы новичок в `bash` или хотите получить некоторые подробности, пояснения и / или примеры операций с параметрами, прочитайте раздел `BashGuide` о расширении параметров и `BashFAQ / 073`.

- `"$var"`, `"${var}"`

Разверните значение, содержащееся в параметре **var**. Синтаксис расширения параметра заменяется содержимым переменной.

- `"${var: -расширенное значение по умолчанию}"`

Разверните значение, содержащееся в параметре **var**, или **расширенное значение по умолчанию** в строке, если значение **var** пустое. Используйте это, чтобы развернуть значение по умолчанию в случае, если значение параметра пустое (не задано или не содержит символов).

- `"${var:=расширенное и присвоенное значение по умолчанию}"`

Разверните значение, содержащееся в параметре **var**, но сначала назначьте параметру **расширенное и присвоенное значение по умолчанию**, если оно пустое. Этот синтаксис часто

используется с командой двоеточия (:): : "\${name:=\$USER}", но также подойдет обычное присвоение с указанным выше:
name="\${name:- \$USER}".

- "\$ {var:?Сообщение об ошибке, если не задано}",
"\${name:?Ошибка: требуется имя.}"

Разверните значение, содержащееся в ИМЕНИ параметра, или покажите сообщение об ошибке, если оно пустое. Сценарий (или функция, если в интерактивной оболочке) прерывается.

- \${name:+Замещающее значение}, \${name:-- name
"\$name"}

Разверните заданную строку, если ИМЯ параметра не является пустым. Это расширение используется в основном для расширения параметра вместе с некоторым контекстом. Пример расширяет **два аргумента**: обратите внимание, что, в отличие от всех других примеров, основное расширение не заключено в кавычки, что позволяет разбивать внутреннюю строку на слова. Однако не забудьте заключить параметр во внутреннюю строку в кавычки!

- "\${строка: 5}", "\${строка:5:10}", "\${
линия: смещение: длина}"

Разверните подстроку значения, содержащегося в строке параметра. Подстрока начинается с символа с номером **5** (или числа, содержащегося в параметре **offset**, во втором примере) и имеет длину **10** символов (или числа, содержащегося в параметре **length**). Смещение основано на **0**. Если длина опущена, подстрока доходит до конца значения параметра.

- "\${@:5}", "\${@:2:4}", "\${ массив:начало:количество}"

Развернуть элементы из массива, начиная с начального индекса и расширяя все или заданное КОЛИЧЕСТВО элементов. Все элементы раскрываются как отдельные аргументы из-за кавычек. Если вы используете **@** в качестве имени параметра, элементы берутся из позиционных параметров (аргументы вашего скрипта - второй пример становится: "\$2" "\$3" "\$4" "\$5").

- "\${!var}"

Разверните значение параметра, названного значением параметра var. Это **плохая практика**! Это расширение делает ваш код крайне непрозрачным и непредсказуемым в будущем. Вероятно, вместо этого вам нужен ассоциативный массив.

- "\${#var}", "\${#myarray[@]}"

Разверните в длину значения параметра var. Второй пример расширяется до количества элементов, содержащихся в массиве с именем **myarray**.

- "\${var#Префикс}", "\${PWD#*/}", "\${PWD##*/}"

Разверните значение, содержащееся в параметре var, после удаления префикса строки A из его начала. Если значение не имеет указанного префикса, оно расширяется как есть. Префикс также может быть шаблоном глобуса, и в этом случае строка, соответствующая шаблону, удаляется спереди. Вы можете удвоить знак **#**, чтобы сделать шаблон жадным.

- `"${var%суффикс}", "${PWD%/*}", "${PWD%%/*}"`

Разверните значение, содержащееся в параметре **var**, после удаления **суффикса строки A** из его конца. Работает точно так же, как операция обрезки префикса, только убирает конец.

- `"${var/pattern/replacement}", "${HOME/$USER/bob}", "${PATH//:/ }"`

Разверните значение, содержащееся в параметре **var**, после замены **заданного шаблона заданной строкой замены**. Шаблон - это глобус, используемый для поиска строки для замены в значении **var**. Первое совпадение заменяется заменяющей строкой. Вы можете удвоить первый `/`, чтобы заменить все совпадения: В третьем примере все двоеточия в значении **ПУТИ** заменяются пробелами.

- `"${var^}", "${var^^}", "${var^^[ac]}"`

Разверните значение, содержащееся в параметре **var**, после ввода всех **символов, соответствующих шаблону**. Шаблон *должен* соответствовать одному символу, а шаблон `?(любой символ)` используется, если он опущен. В первом примере в верхнем регистре указан первый символ из значения **переменной**, во втором - все символы в верхнем регистре. В третьем верхнем регистре все символы, которые являются либо **a**, либо **c**.

- `"${var,,}", "${var,,,}", "${var,,,[AC]}"`

Разверните значение, содержащееся в параметре **var**, после того, как **все символы в нижнем регистре соответствуют шаблону**. Работает точно так же, как операция верхнего регистра, только символы в нижнем регистре соответствуют символам.

Массивы

Массивы - это переменные, которые содержат несколько строк. Всякий раз, когда вам нужно сохранить несколько элементов в переменной, **используйте массив, а НЕ строковую** переменную. Массивы позволяют вам четко разделять элементы и позволяют аккуратно разложить элементы на отдельные аргументы. Это невозможно сделать, если вы объединяете свои элементы в строку!

Если вы новичок в `bash` или не до конца понимаете, что такое массивы и почему их следует использовать вместо обычных переменных, или вы ищете дополнительные объяснения и / или примеры массивов, прочитайте раздел `BashGuide` о массивах и `BashFAQ / 005`

Создание массивов

- `myarray=(foo bar quux)`

Создайте массив **myarray**, содержащий три элемента. Массивы создаются с использованием синтаксиса `X= (y)`, а элементы массива отделяются друг от друга пробелами.

- `myarray=("foo bar" quux)`

Создайте массив **myarray**, содержащий два элемента. Чтобы поместить элементы в массив, содержащие пробелы, заключите их в кавычки, чтобы указать `bash`, что заключенный в кавычки текст принадлежит одному элементу массива.

- `myfiles=(*.txt)`

Создайте массив **myfiles**, содержащий все имена файлов в текущем каталоге, которые заканчиваются на **.txt**. Мы можем использовать любой тип расширения внутри синтаксиса назначения массива. В примере используется расширение имени пути для замены шаблона глобуса на все имена файлов, которым он соответствует. После замены назначение массива происходит, как в первых двух примерах.

- `myfiles+=(*.html)`

Добавьте все HTML-файлы из текущего каталога в массив **myfiles**.

Синтаксис `X += (y)` можно использовать так же, как и обычный синтаксис назначения массива, но добавлять элементы в конец массива.

- `имена[5]= "Большой Джон", имена[n + 1]= "Длинный Джон"`

Присвойте строку определенному индексу в массиве. Используя этот синтаксис, вы явно указываете Bash, по какому индексу в вашем массиве вы хотите сохранить строковое значение. Индекс фактически интерпретируется как *арифметическое выражение*, поэтому вы можете легко выполнять математические вычисления там.

- читать - `ra myarray`

Разделите строку на поля и сохраните поля в массиве **myarray**.

Команды ЧТЕНИЯ считывают строку из *stdin* и используют каждый символ в переменной IFS в качестве разделителя, чтобы разделить эту строку на поля.

- `ЕСЛИ=, прочитайте имена <<< "Джон, Лукас, Смит, Иоланда"`

Разделите строку на поля, используя в качестве разделителя, и сохраните поля в массиве с именем **names**. Мы используем синтаксис `<<<` для передачи строки в *stdin* команды чтения. Значение IFS установлено на время выполнения команды ЧТЕНИЯ, что приводит к разделению строки ввода на поля, разделенные запятой. Каждое поле хранится как элемент в массиве ИМЕН.

- `IFS=$'\n' читать -d " -ra строки`

Считайте все строки из *stdin* в элементы массива с именем **lines**. Мы используем переключатель ЧТЕНИЯ `-d "`, чтобы указать ему не прекращать чтение после первой строки, заставляя его читать во всех стандартных файлах. Затем мы устанавливаем IFS в символ новой строки, заставляя `read` разбивать ввод на поля всякий раз, когда начинается новая строка.

- `files=(); в то время как IFS= read -d " -r file; do files+=("$file"); готово < <(найти . -имя '*.txt' -print0)`

Безопасное чтение всех текстовых файлов, содержащихся рекурсивно в текущем каталоге, в массив именованных **файлов**.

Мы начинаем с создания пустого массива с именем **files**. Затем мы запускаем цикл `while`, который запускает инструкцию `read` для чтения в имени файла из *stdin*, а затем добавляет это имя файла (содержащееся в

переменной `file`) в массив `files`. Для инструкции `read` мы устанавливаем значение `IFS` в значение `empty`, избегая поведения `read` по удалению начальных пробелов из входных данных, и мы устанавливаем `-d "`, чтобы сообщить `read` продолжить чтение, пока он не увидит **нулевой** байт (имена файлов **МОГУТ** занимать несколько строк, поэтому мы не хотим, чтобы `read` прекращал чтение имени файла после одной строки!). Для **ВВОДА** мы подключаем команду `find` к стандартному интерфейсу `while`. Команда `find` использует `-print0` для вывода своих имен файлов, разделяя их **нулевыми** байтами (см. `-d "` при **ЧТЕНИИ**). **ПРИМЕЧАНИЕ:** Это **единственный** по-настоящему безопасный способ создания массива имен файлов из выходных данных команды! Вы **должны** разделять свои имена файлов **нулевыми** байтами, потому что это **единственный** байт, который на самом деле не может отображаться внутри имени файла! **НИКОГДА** не используйте `ls` для перечисления имен файлов! Сначала попробуйте использовать приведенные выше примеры `glob`, они так же безопасны (не нужно анализировать внешнюю команду), намного проще и быстрее.

- объявить `-A homedirs=(["Питер"]=~пит ["Йохан"]=~джо ["Роберт"]=~роб)`

Создайте **ассоциативный** массив, сопоставляя имена с домашними каталогами пользователей. В отличие от обычных массивов, индексы ассоциативных массивов представляют собой строки (как и значения).
Примечание: вы **должны** использовать `declare -A` при создании ассоциативного массива, чтобы указать `bash`, что индексы этого массива являются строками, а не целыми числами.

- `homedirs["Джон"]=~джон`

Добавьте элемент в ассоциативный массив с ключом **"John"**, сопоставленный с домашним каталогом **Джона**.

Использование массивов

- `echo "${имена[5]}", echo "${имена[n + 1]}"`

Разверните один элемент из массива, на который ссылается его индекс. Этот синтаксис позволяет получить значение элемента по индексу элемента. Индекс фактически интерпретируется как *арифметическое выражение*, поэтому вы можете легко выполнять математические вычисления там.

- `echo "${имена[@]}"`

Разверните каждый элемент массива как отдельный аргумент. Это предпочтительный способ расширения массивов. Каждый элемент в массиве расширяется, как если бы он передавался как новый аргумент, правильно заключенный в кавычки.

- `cp "${myfiles[@]}" /destinationdir/`

Скопируйте все файлы, на которые ссылаются имена файлов в массиве `myfiles`, в `/destinationdir/`. Расширение массива происходит с использованием синтаксиса `"${array[@]}"`. Он эффективно заменяет этот синтаксис расширения списком всех элементов, содержащихся в массиве, правильно цитируемых как отдельные аргументы.

- `rm "${myfiles[@]}"`

Удалите все файлы, на которые ссылаются имена файлов в массиве **myfiles**. Как правило, присоединять строки к синтаксису расширения массива - *плохая* идея. Что происходит: строка имеет только префикс к *первому* элементу, расширенному из массива (или суффикс к последнему, если вы прикрепили строку к концу синтаксиса расширения массива). Если мои файлы **содержат** элементы - `foo.txt` и `bar-.html`, эта команда расширилась бы до: `rm "./-foo.txt " "bar-.html "`. Обратите внимание, что только первый элемент имеет префикс `./`. В данном конкретном случае это удобно, потому что `rm` завершается ошибкой, если первое имя файла начинается с тире. Теперь он начинается с точки.

- `(IFS=,; echo "${имена[*]}")`

Разверните **ИМЕНА** массивов в *одну строку*, содержащую все элементы массива, объединив их, разделив запятой (,). Синтаксис `"${array[*]}"` очень редко бывает полезен. Как правило, когда вы видите это в скриптах, это ошибка. Единственное его применение - объединить все элементы массива в одну строку для отображения пользователю. Обратите внимание, что мы заключили оператор в (квадратные скобки), создав подоболочку: это расширит назначение IFS, сбросив его после завершения подоболочки.

- для файла в `"${myfiles[@]}"`; нужно ли `read -p "Удалить $file? " && [[$REPLY = y]] && rm "$file";` готово

Выполните итерацию по всем элементам массива **myfiles** после их расширения в операторе **for**. Затем для каждого файла спросите пользователя, хочет ли он его удалить.

- для индекса в `"${!myfiles[@]}"`; сделайте эхо "Номер файла \$index равен \${myfiles[index]}"; готово

Выполните итерацию по всем ключам массива **myfiles** после их расширения в операторе **for**. Синтаксис `"${!array[@]}"` (обратите внимание !) расширяется в список *ключей* массива, а не значений. Ключами обычных массивов являются числа, начинающиеся с 0. Синтаксис для перехода к определенному элементу в массиве - `"${array[index]}"`, где **индекс** - это ключ элемента, к которому вы хотите получить доступ.

- `имена=(Джон Пит Роберт); echo "${имена[@]}/#/Длинные}"`

Выполните операцию расширения параметров для каждого элемента массива **ИМЕН**. При добавлении операции расширения параметра к расширению массива операция применяется к каждому отдельному элементу массива по мере его расширения.

- `имена=(Джон Пит Роберт); echo "${имена[@]:начало:длина}"; echo "${имена[@]:1:2}"`

Разверните элементы массива **ДЛИНЫ**, начиная с **начала** индекса. Аналогично простому `"${names[@]}"`, но расширяет *подраздел*

массива. Если значение `length` опущено, остальные элементы массива расширяются.

- `printf '%s\n' "${имена[@]}"`

Выводите каждый элемент массива в новой строке. Этот оператор `printf` - очень удобный метод для вывода элементов массива обычным способом (в данном случае, добавления новой строки к каждому). Строка формата, указанная в `printf`, применяется к каждому элементу (если, конечно, в нем не указано несколько `%S`).

- для имени в `"${!homedirs[@]}"`; повторите `"$name живет в $ {homedirs[$name]}"`; готово

Выполните итерацию по всем КЛЮЧАМ массива `homedirs` после их расширения в операторе `for`. Синтаксис для доступа к ключам ассоциативных массивов такой же, как и для обычных массивов. Вместо чисел, начинающихся с `0`, мы теперь получаем ключи, для которых мы сопоставили значения нашего ассоциативного массива. Позже мы можем использовать эти ключи для поиска значений в массиве, как в обычных массивах.

- `printf '%s\n' "${#имена[@]}"`

Выведите количество элементов в массиве. В этом операторе `printf` расширение расширяется только до одного аргумента, независимо от количества элементов в массиве. Расширенный аргумент - это число, которое указывает количество элементов в массиве **ИМЕН**.

Примеры: Базовые структуры

Составные команды

Списки команд

- `[[$1]] || { echo "Вам нужно указать аргумент!" >&2; выход 1; }`

Мы используем здесь группу команд, потому что оператор `||` принимает только одну команду.

Мы хотим, чтобы команды `echo` и `exit` выполнялись, если `$1` пуст.

- `(IFS=' '; echo "Массив содержит эти элементы: ${array[*]}")`

Мы используем круглые скобки, чтобы вызвать здесь подболочку. Когда мы устанавливаем переменную `IFS`, она будет меняться только в подболочке, а не в нашем основном скрипте. Это позволяет избежать необходимости сбрасывать значение по умолчанию после расширения в операторе `echo` (что в противном случае нам пришлось бы сделать, чтобы избежать неожиданного поведения позже).

- `(cd "$ 1" && tar -архив cvjpf.tbz2 .)`

Здесь мы используем подболочку, чтобы временно изменить текущий каталог на то, что находится в `$ 1`.

После операции `tar` (когда дочерняя оболочка заканчивается) мы

возвращаемся к тому, что было до команды **cd**, потому что текущий каталог основного скрипта никогда не менялся.

Выражения

- `((завершение = текущее * 100 / всего))`
Обратите внимание, что арифметический контекст следует совершенно иным правилам синтаксического анализа, чем обычные операторы **bash**.
- `[[$foo = /*]] && echo "foo содержит абсолютный путь"`.
Мы можем использовать команду `[` для выполнения всех тестов, которые может выполнить **test(1)**.
Но, как показано в примере, он может делать гораздо больше, чем **test(1)**; например, сопоставление шаблонов глобусов, сопоставление регулярных выражений, группирование тестов и т. Д.

Циклы

- для файла в формате *.mp3; сделайте `openssl md5 "$file"`; готово `> mysongs.md5`
Циклы **For** перебирают все аргументы после ключевого слова **in**.
Один за другим каждый аргумент помещается в файл с именем переменной, и выполняется тело цикла.
НЕ ПЕРЕДАВАЙТЕ ВЫВОД КОМАНДЫ В for ВСЛЕПУЮ!
for будет перебирать СЛОВА в выводе команды; что почти НИКОГДА не является тем, чего вы действительно хотите!
- для файла; сделайте `cp "$file" /backup/`; готово
Эта краткая версия цикла **for** повторяет позиционные параметры.
По сути, это эквивалент `for file в "$@"`.
- для `((i = 0; i < 50; i++))`; сделать `printf "%02d", "$i"`; готово
Генерирует список чисел, разделенных запятыми, с добавлением нуля до двух цифр.
(Последним символом будет запятая, да, если вы действительно хотите от нее избавиться; вы можете - но это противоречит простоте этого примера)
- при чтении `_ line`; выполнить `echo "$line"`; готово `< файл`
Этот цикл **while** продолжается до тех пор, пока команда **read** выполняется успешно.
(Имеется в виду, до тех пор, пока строки могут быть прочитаны из файла).
Пример в основном просто выбрасывает первый столбец данных из файла и печатает остальные.
- до тех пор, пока `myserver`; не повторит "Мой сервер разбился с кодом выхода: \$?"; перезапуск его через 2 секунды `.."`; сон 2; готово

Этот цикл перезапускает **myserver** каждый раз, когда он завершается с неуспешным кодом выхода.

Предполагается, что когда **myserver** завершает работу с неуспешным кодом выхода; он разбился и нуждается в перезапуске; и если он существует с успешным кодом выхода; вы приказали ему завершить работу, и его не нужно перезапускать.

- выберите фрукты в Яблоко Груша Виноград Банан Клубника; `do ((кредит - = 2, здоровье + = 5)); echo "Вы купили фрукты за несколько долларов. Наслаждайтесь!"; готово`

Простая программа, которая преобразует кредиты в здоровье. Удивительные.

Встроенные

Чайники

- пока! `ssh lhunath@lyndir.com ; сделать :: сделано`
Повторное подключение при сбое.

Декларативный

- псевдоним `l='ls -al'`
Создайте псевдоним с именем **l**, который заменяется на **ls -al**.
Удобно для быстрого просмотра подробного содержимого каталога.
- объявить `-i myNumber=5`
Объявите целое число с именем **myNumber**, инициализированное значением **5**.
- экспорт `AUTOSSH_PORT=0`
Экспортируйте переменную в среду процесса **bash** с именем **AUTOSSH_PORT**, которая будет унаследована любым процессом, вызываемым этим процессом **bash**.
- `foo() { local bar=fooBar; echo "Внутри foo(), bar равен $bar"; }; echo "Установка bar в 'normalBar'"; bar= normalBar; foo; echo "Вне foo(), bar равен $bar"`
Упражнение в области переменных.
- если ! введите `-P ssh >/dev/null; затем повторите "Пожалуйста, установите OpenSSH". >&2; выход 1; fi`
Проверьте, доступен ли **ssh**.
Предложите пользователю установить *OpenSSH*, если это не так, и выйдите.

Ввод

- прочитайте Имя Фамилия Номер телефона адрес

Считывать данные из одной строки с четырьмя полями в четыре именованные переменные.

Вывод

- `echo "Мне действительно не нравится $nick. Он может быть таким придурком ".`

Выведите простую строку на стандартный вывод.

- `printf "Мне действительно не нравится %s . Он может быть таким придурком ". "$nick"`

То же самое, используя **printf** вместо **echo**, красиво отделяя текст от данных.

Выполнение

- `cd ~ лунат`

Измените текущий каталог на домашний каталог `lhunath`.

- `cd() { встроенный cd "$@" && echo "$PWD"; }`

Внутри функции выполните встроенную команду **cd**, а не функцию (что приведет к бесконечной рекурсии), и если это удастся, **повторите** новый текущий рабочий каталог.

- источник `bashlib`; источник `./foorc`

Запустите весь код `bash` в файле с именем **bashlib**, который существует где-то в **PATH**; затем сделайте то же самое для file `.foorc` в текущем каталоге.

- `exec 2>/var/log/foo.log`

С этого момента отправляйте все выходные данные в стандартную ошибку в файл журнала.

- `echo "Произошла фатальная ошибка! Завершение!";`
`выход 1`

Отобразите сообщение об ошибке и выйдите из скрипта.

Категория Shell