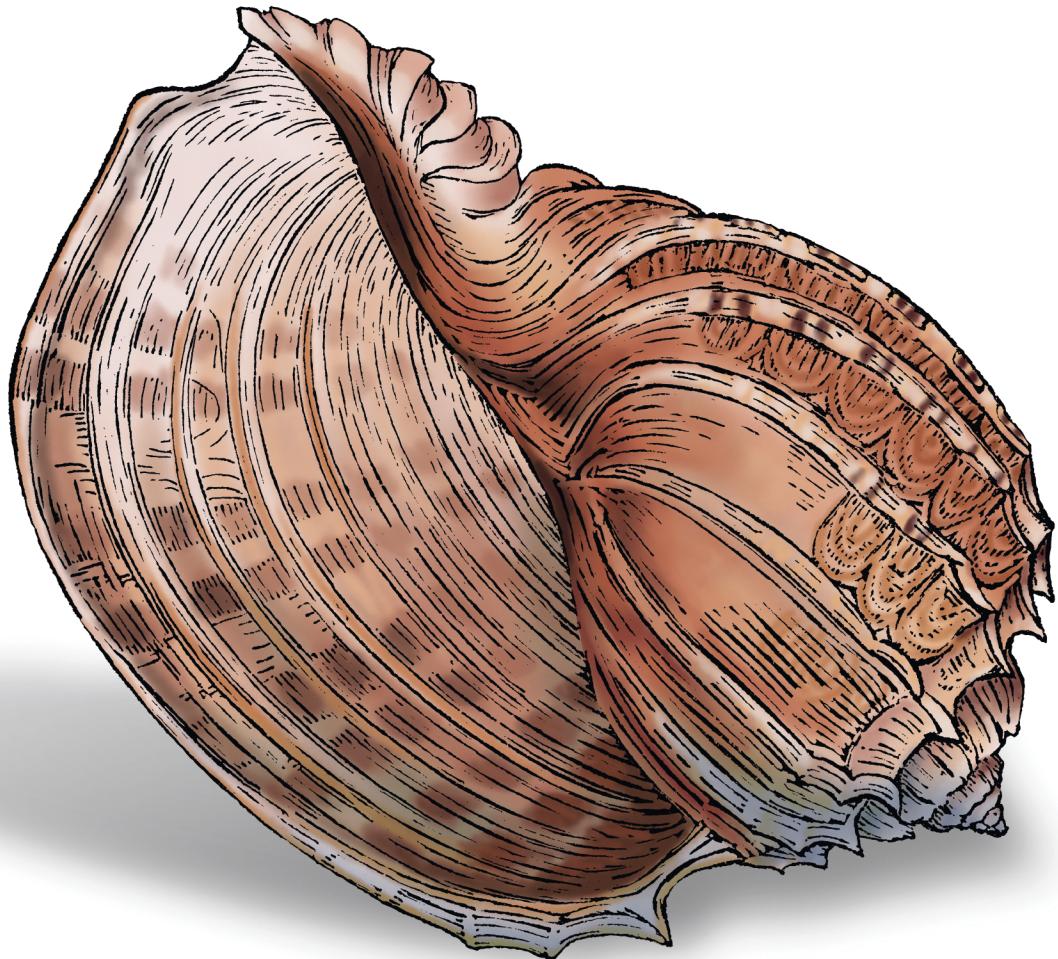


O'REILLY®

# Идиомы bash

мощные, гибкие и понятные сценарии командной оболочки



Карл Олбинг, Джей Пи Фоссен

# Идиомы bash

мощные, гибкие и понятные  
сценарии командной оболочки

Карл Олбинг, Джей Пи Фоссен



Санкт-Петербург · Москва · Минск

2023

ББК 32.973.2-018.2

УДК 004.451.9

О-53

### Олбинг Карл, Фоссен Джей Пи

О-53 Идиомы bash. — СПб.: Питер, 2023. — 208 с.: ил. — (Серия «Бестселлеры O'Reilly»).

ISBN 978-5-4461-2307-0

Сценарии на языке командной оболочки получили самое широкое распространение, особенно написанные на языках, совместимых с bash. Но эти сценарии часто сложны и непонятны. Сложность — враг безопасности и причина неудобочитаемости кода. Эта книга на практических примерах покажет, как расшифровывать старые сценарии и писать новый код, максимально понятный и легко читаемый.

Авторы покажут, как использовать мощь и гибкость командной оболочки. Даже если вы умеете писать сценарии на bash, эта книга поможет расширить ваши знания и навыки. Независимо от используемой ОС — Linux, Unix, Windows или Mac — к концу книги вы научитесь понимать и писать сценарии на экспертном уровне. Это вам обязательно пригодится.

Вы познакомитесь с идиомами, которые следует использовать, и такими, которых следует избегать.

**16+** (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.973.2-018.2

УДК 004.451.9

Права на издание получены по соглашению с O'Reilly.

Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги.

Издательство не несет ответственности за доступность материалов, ссылки на которые вы можете найти в этой книге. На момент подготовки книги к изданию все ссылки на интернет-ресурсы были действующими.

ISBN 978-1492094739 англ.

Authorized Russian translation of the English edition of bash Idioms, ISBN 9781492094753 © 2022 Carl Albing and JP Vossen.

This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

ISBN 978-5-4461-2307-0

© Перевод на русский язык ООО «Прогресс книга», 2022

© Издание на русском языке, оформление ООО «Прогресс книга», 2023

© Серия «Бестселлеры O'Reilly», 2023

# Оглавление

[https://t.me/it\\_boooks](https://t.me/it_boooks)

<b>Вступление.....</b>	<b>10</b>
Запуск bash.....	12
Управление версиями .....	13
Hello World .....	14
Условные обозначения.....	14
Использование исходного кода примеров.....	15
<b>Благодарности .....</b>	<b>17</b>
<b>От издательства .....</b>	<b>19</b>
<b>Глава 1. Идиома «большого» if .....</b>	<b>20</b>
«Большой» if .....	20
Или ELSE.....	22
Выполняем несколько команд.....	23
Еще о случае нескольких команд.....	25
Так делать не нужно! .....	25
В заключение: стиль и удобочитаемость.....	27
<b>Глава 2. Язык циклов .....</b>	<b>29</b>
Циклические конструкции.....	29
Явные значения.....	31
Почти как в Python .....	34
Кавычки и пробелы .....	35
Разработка и тестирование циклов for .....	37
Циклы while и until.....	39
В заключение: стиль и удобочитаемость.....	39

<b>Глава 3. На всякий случай: оператор Case .....</b>	<b>42</b>
Сделайте свой выбор.....	42
Применение на практике .....	44
Задача.....	45
Наш сценарий .....	45
Сценарии-обертки .....	47
Еще один важный момент .....	54
В заключение: стиль и удобочитаемость.....	55
<b>Глава 4. Язык переменных .....</b>	<b>57</b>
Ссылка на переменную.....	57
Дополнительные параметры .....	59
Сокращенный вариант команды basename .....	59
Удаление пути или префикса .....	60
Сокращенный вариант команды dirname или удаление суффикса .....	61
Другие модификаторы .....	62
Условные подстановки .....	66
Значения по умолчанию.....	66
Списки значений, разделенных запятыми.....	67
Изменение значения.....	68
\$RANDOM .....	68
Подстановка команд.....	69
В заключение: стиль и удобочитаемость.....	71
<b>Глава 5. Выражения и арифметика .....</b>	<b>72</b>
Арифметика .....	73
Круглые скобки не нужны.....	76
Составные команды .....	77
В заключение: стиль и удобочитаемость.....	80
<b>Глава 6. Функции .....</b>	<b>82</b>
Вызов функций .....	82
Определение функций.....	83
Параметры функций.....	83
Возвращаемые значения функций .....	85
Локальные переменные .....	86

Особые случаи.....	87
Функция printf.....	88
Вывод POSIX .....	89
Получение и использование даты и времени.....	90
printf для повторного использования или отладки.....	91
В заключение: стиль и удобочитаемость.....	91
<b>Глава 7. Списки и хеши.....</b>	<b>93</b>
Сходные черты .....	95
Списки .....	96
Хеши .....	101
Пример подсчета слов .....	106
В заключение: стиль и удобочитаемость.....	109
<b>Глава 8. Аргументы.....</b>	<b>110</b>
Ваш первый аргумент.....	110
Поддержка ключей .....	112
Анализ ключей.....	113
Длинные ключи .....	115
HELP! .....	118
Отладочный и подробный режимы вывода .....	122
Версия.....	123
В заключение: стиль и удобочитаемость.....	124
<b>Глава 9. Файлы и не только .....</b>	<b>125</b>
Чтение файлов .....	125
read.....	125
mapfile.....	126
Метод «грубой силы» .....	130
Изменяем \$IFS при чтении файлов .....	130
Имитации файлов .....	133
Настроечные каталоги.....	134
Организация библиотек.....	135
Shebang!.....	136
Строгий режим bash.....	138
Код выхода.....	139

Это ловушка! .....	140
Встроенные документы и строки.....	142
Код выполняется в интерактивном режиме? .....	143
В заключение.....	144
<b>Глава 10. Помимо идиом: работа с bash.....</b>	<b>145</b>
Приглашения к вводу .....	146
Часовой пояс в приглашении .....	149
Получение ввода пользователя .....	149
read .....	150
pause.....	152
select .....	152
Псевдонимы.....	153
Функции.....	155
Локальные переменные.....	156
Возможности Readline.....	156
Журналирование в bash.....	158
Обработка JSON с помощью jq.....	159
Поиск в списке процессов.....	160
Ротация старых файлов .....	161
Встроенная документация .....	163
Отладка в bash .....	169
Модульное тестирование в bash.....	172
В заключение.....	173
<b>Глава 11. Разработка своего руководства по стилю .....</b>	<b>174</b>
Удобочитаемость .....	177
Комментарии.....	179
Имена .....	180
Функции.....	182
Кавычки.....	183
Форматирование.....	184
Синтаксис .....	186
Другие рекомендации.....	187
Шаблон сценария .....	187
Другие руководства по стилю .....	189

---

Инструмент проверки оформления кода на bash.....	190
В заключение.....	191
<b>Приложение. Руководство по стилю .....</b>	<b>192</b>
Удобочитаемость .....	193
Комментарии.....	194
Имена .....	194
Функции.....	195
Кавычки.....	197
Форматирование.....	198
Синтаксис .....	198
Другие рекомендации.....	199
Шаблон сценария .....	200
<b>Об авторах .....</b>	<b>202</b>
<b>Иллюстрация на обложке.....</b>	<b>203</b>

# Вступление

Вот как словарь Уэбстера определяет термин *идиома*:<sup>1</sup>

1. Специфический оборот речи, употребляющийся как единое целое, значение которого не определяется значением входящих в него слов (как, например, оборот «в подвешенном состоянии», означающий «неопределенность»). В данном обороте может иметь место нетипичное грамматическое использование слов (например, «дать дорогу»).
- 2а. Язык, свойственный народу, географической области, сообществу или классу, диалект.
- 2б. Синтаксическая, грамматическая или структурная форма, характерная для языка.
3. Стиль или форма художественного выражения, характерные для человека, периода или движения, средства или инструмента.

Почему для книги выбрано название «Идиомы bash»? Для простоты. Или, если хотите, чтобы было понятнее. В этой книге «простота» — синоним «понятности». Мы не собираемся убеждать вас в важности удобочитаемости: если это не первая книга по программированию, которую вы читаете, значит, уже должны это понимать. *Удобочитаемость* означает простоту чтения и понимания кода, особенно если он написан не вами. Не менее важно научиться писать код так, чтобы в будущем вы или кто-то другой смогли его понять. Очевидно, что эти аспекты являются разными сторонами одной медали, поэтому

---

<sup>1</sup> <https://oreil.ly/pgx8b>.

мы рассмотрим и идиомы, которые следует использовать, и такие, которых следует избегать.

Между нами говоря, мы считаем bash языком «управления». Для сложной обработки данных он малопригоден: ее можно реализовать, но код получится слишком сложным. Однако, если все инструменты, необходимые для обработки данных, уже имеются и требуется лишь «склеить» их, то bash подойдет на эту роль как нельзя лучше.

Если мы собираемся использовать bash только для управления, то зачем беспокоиться об идиомах или «структурной форме» языка? Программы развиваются, возможности ширятся, ситуация меняется, но нет ничего более постоянного, чем временное. Рано или поздно кому-то придется прочитать ваш код, понять его и изменить. Если он написан с использованием непонятных идиом, то сделать это будет намного сложнее.

Во многих отношениях bash не похож на другие языки. У него богатая история (некоторые могут сказать «багаж»), и есть причины, почему он выглядит и работает определенным образом. Мы не будем много говорить об этом. Если вам интересна эта тема, обратитесь к нашей книге «*bash Cookbook*»<sup>1</sup>. Сценарии командной оболочки «управляют миром», по крайней мере в мирах Unix и Linux (а Linux фактически управляет облачным миром), причем подавляющее большинство этих сценариев написаны на bash. Поддержание обратной совместимости с самыми первыми командными оболочками Unix часто критически важно, но накладывает некоторые... ограничения.

Теперь о «диалектах». Наиболее важным, особенно для обратной совместимости, является стандарт POSIX. Об этом тоже не будем много говорить, в конце концов эта книга посвящена идиомам bash, а не POSIX. Появление других диалектов возможно, когда программисты пишут код на bash в стиле другого известного им языка. Однако поток, имеющий смысл в C, может показаться бессвязным в bash.

<sup>1</sup> <https://learning.oreilly.com/library/view/bash-cookbook-2nd/9781491975329/>.

Итак, в этой книге мы намерены продемонстрировать «стиль или форму... выражения, характерную» для bash (в духе третьего определения в словаре Уэбстера). Программисты на Python называют свой стиль *pythonic*. А мы бы хотели в этой книге показать стиль *bashy*.

К концу книги читатель приобретет следующие знания и навыки:

- научится писать полезный, гибкий, удобочитаемый и... стильный код на bash;
- узнает, как расшифровываются идиомы bash, такие как  `${MAKEMELC,,}` и  `${PATHNAME##*/}`;
- освоит приемы, экономящие время и обеспечивающие согласованность при автоматизации задач;
- сможет удивить и впечатлить коллег идиомами bash;
- узнает, как идиомы bash помогают сделать код чистым и лаконичным.

## Запуск bash

Мы предполагаем, что вы уже имеете опыт программирования на bash и нет нужды рассказывать, где его найти или как установить. Конечно, bash есть почти во всех дистрибутивах Linux, и он уже установлен по умолчанию или может быть установлен практически в любой операционной системе. Получить версию для Windows можно с помощью Git for Windows<sup>1</sup> подсистемы Windows для Linux (Windows Subsystem for Linux, WSL) или другими способами.

## bash на Mac

Обратите внимание на версию bash, которая по умолчанию устанавливается в Mac: она довольно старая и не поддерживает многие новые идиомы версий 4.0 и выше. Более свежую версию можно получить

---

<sup>1</sup> <https://gitforwindows.org/>.

с помощью MacPorts, Homebrew или Fink. По информации от Apple<sup>1</sup>, проблема в том, что новые версии bash используют GPLv3, что является проблемой для macOS.

Apple также сообщает, что macOS Catalina и более новые версии будут использовать Zsh в качестве интерактивной оболочки и оболочки входа по умолчанию. Zsh в значительной мере совместима с bash, но некоторые примеры в этой книге потребуют изменений. Командная оболочка bash на компьютерах Mac не исчезнет (по крайней мере, пока), причем использование Zsh в качестве оболочки по умолчанию не повлияет на строку «shebang» (см. раздел «Shebang!» в главе 9), но имейте в виду, что если не обновить версию bash, вы застрянете в каменном веке.

Мы отметили примеры сценариев, несовместимые с Zsh, комментарием: «Не работает в Zsh 5.4.2!».

## bash в контейнерах

Будьте осторожны, используя Docker и другие создатели контейнеров, где /bin/sh ссылается не на bash, а /bin/bash может вообще не существовать! Это особенно актуально для ограниченных окружений, включая системы интернета вещей и промышленные контроллеры.

/bin/sh может ссылаться на bash (согласно POSIX), но также на Ash, Dash, BusyBox (которым чаще всего является Dash) или что-то еще. Будьте внимательны (см. раздел «Shebang!» в главе 9) и убедитесь, что bash действительно установлен, или придерживайтесь стандарта POSIX и избегайте «башизмов».

# Управление версиями

Очень надеемся, что вы используете какую-либо систему управления версиями. Если да, то можете пропустить этот абзац. Если нет, то внедрите

---

<sup>1</sup> <https://oreil.ly/2PZRm>.

их в свою работу, прежде чем продолжать чтение. Мы посвятили этому вопросу целое приложение в «*bash Cookbook*»<sup>1</sup>, но вообще в интернете можно найти огромный объем информации о таких системах, в том числе от одного из авторов этой книги<sup>2</sup>.

## Hello World

Во многих трудах нужно добраться до конца главы 1, 2 или даже 3, прежде чем вы узнаете, как вывести на экран «Hello World». Мы же перейдем к этому вопросу немедленно! Впрочем, поскольку вы уже писали код на bash и храните его в системе управления версиями (верно?), то говорить об `echo 'Hello, World'` было бы довольно глупо, а потому и не будем. Упс.

## Условные обозначения

В этой книге используются следующие условные обозначения.

### *Курсив*

Курсивом выделены новые термины или важные понятия.

### Моношириинный шрифт

Обозначает листинги программ, а также используется внутри абзацев для переменных, функций, баз данных, типов данных, переменных среды, операторов и ключевых слов, имен файлов и их расширений.

### Моношириинный полужирный шрифт

Обозначает элементы, которые пользователь должен ввести самостоятельно.

---

<sup>1</sup> <https://github.com/vossenjp/bashidioms-examples/blob/main/bcb2-appd.pdf>.

<sup>2</sup> <https://oreil.ly/fPHy8>.

### Моноширинный курсив

Обозначает текст, который должен быть заменен значениями, введенными пользователем или определяемыми контекстом.

### Шрифт без засечек

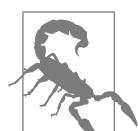
Используется для обозначения URL, адресов электронной почты, названий кнопок и других элементов интерфейса, а также каталогов.



Этот рисунок указывает на совет или предложение.



Этот рисунок указывает на общее примечание.



Этот рисунок указывает на предупреждение.

## Использование исходного кода примеров

Вспомогательные материалы (примеры кода, упражнения и т. д.) доступны для загрузки по адресу: <https://github.com/vossenjp/bashidioms-examples>. Если у вас возникнут вопросы технического характера по использованию примеров кода, направляйте их по электронной почте на адрес [bookquestions@oreilly.com](mailto:bookquestions@oreilly.com).

В общем случае все примеры кода из книги вы можете использовать в своих программах и в документации. Вам не нужно обращаться

в издательство за разрешением, если вы не собираетесь воспроизводить существенные части программного кода. Если вы разрабатываете программу и используете в ней несколько фрагментов кода из книги, вам не нужно обращаться за разрешением. Но для продажи или распространения примеров из книги вам потребуется разрешение от издательства O'Reilly. Вы можете отвечать на вопросы, цитируя данную книгу или примеры из нее, но для включения существенных объемов программного кода из книги в документацию вашего продукта потребуется разрешение.

Мы рекомендуем, но не требуем добавлять ссылку на первоисточник при цитировании. Под ссылкой на первоисточник мы подразумеваем указание авторов, издательства и ISBN.

За получением разрешения на использование значительных объемов программного кода из книги обращайтесь по адресу [permissions@oreilly.com](mailto:permissions@oreilly.com).

# Благодарности

## Bash

Спасибо GNU Software Foundation и Брайану Фоксу (Brian Fox) за создание bash. Благодарим также Чета Рэми (Chet Ramey), поддерживающего и совершенствующего bash, начиная с версии 1.14, которая вышла в первой половине 1990-х. Все вы дали нам отличный инструмент.

## Рецензентам

Большое спасибо рецензентам: Дугу Макилрою (Doug McIlroy), Яну Миеллу (Ian Miell), Кертису Олду (Curtis Old) и Полу Тронконе (Paul Troncone), которые помогли значительно улучшить книгу! Все они дали ценные отзывы, а в некоторых случаях предложили альтернативные решения, указав на проблемы, которые мы упустили из виду. Если в этой книге есть ошибки или упущения, то это не их, а наша вина.

## O'Reilly

Спасибо всем сотрудникам издательства O'Reilly, без которых эта книга не появилась бы на свет, а если и появилась бы, то по содержанию и оформлению была бы беднее.

Спасибо Майку Лукидесу (Mike Loukides) за оригинальную идею и то, что предложил и доверил реализовать ее нам. Спасибо Сюзанне «Зан»

МакКуэйд (Suzanne «Zan» McQuade), что помогла воплотить идею в жизнь. Огромная благодарность Николь Таше (Nicole Taché) и Кристен Браун (Kristen Brown) за правки и то, что терпели нас в ходе долгой работы над книгой и ее подготовки к печати. Особое спасибо Нику Адамсу (Nick Adams) из Tools за исправление наших многочисленных (и порой волнистых) ошибок в AsciiDoc, а также помочь в вопросах, не связанных с набором. Спасибо литературному редактору Ким Сандал (Kim Sandoval), составителю индекса Шерил Ленсер (Cheryl Lenser), корректору Лиз Уилер (Liz Wheeler), дизайнерам Дэвиду Футато (David Futato) и Карен Монтгомери (Karen Montgomery), а также другим сотрудникам O'Reilly.

## От Карла

Спасибо Джей Пи за его работу, внимание к деталям и готовность к сотрудничеству. Спасибо всем сотрудникам O'Reilly за помощь в издании этой книги.

Эту книгу я посвящаю моей супруге Синтии, которая мерились с моими писательскими амбициями и достаточно убедительно делала вид, что ей интересно, о чем я пишу. Моя работа над этой книгой направлена, как говорят в Бетельском университете, во славу Божию и на благо моих близких.

## От Джей Pi

Спасибо Карлу за его работу; похоже, нам снова удалось уложиться в график. Спасибо Майку за то, что опять привел все в движение, и Николь за ее труд, а также терпеливое отношение к нашим работе, жизни и неумению правильно распределять время.

Эту книгу я посвящаю моей супруге Карен — «исполнительному вице-президенту», отвечающему за все. Спасибо за твою невероятную поддержку, терпение и понимание, без тебя я бы не состоялся в жизни. Наконец, спасибо Кейт и Сэмю за то, что нормально воспринимали мой ответ: «Если вы не истекаете кровью и не горите, то не мешайте: я занят книгой».

# **От издательства**

Ваши замечания, предложения, вопросы отправляйте по адресу [comp@piter.com](mailto:comp@piter.com) (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства [www.piter.com](http://www.piter.com) вы найдете подробную информацию о наших книгах.

## ГЛАВА 1

---

# Идиома «большого» if

[https://t.me/it\\_boooks](https://t.me/it_boooks)

Знакомство с идиомами bash мы начнем с конструкции, которая позволяет делать то же самое, что и привычные операторы `if/then/else`, но имеет более компактный синтаксис. Идиоматическая конструкция, которую мы рассмотрим в этой главе, не только дает реальные преимущества (в основном — краткость), но также таит некоторые ловушки. Кроме того, не зная этой идиомы bash, можно вообще не понять смысл кода.

Взгляните на фрагмент кода:

```
[[ -n "$DIR" ]] && cd "$DIR"
```

Как вы думаете, похож он на оператор `if`? Знакомые с bash заметят, что функционально это тождественно оператору `if`, даже при том, что ключевое слово `if` отсутствует.

Давайте разберем, что делает этот код.

## «Большой» if

Прежде чем объяснить эту идиому, рассмотрим похожий, но более простой пример:

```
cd tmp && rm scratchfile
```

По сути, это тоже оператор `if`. Если команда `cd` выполнится успешно, то выполнится и команда `rm`. «Идиома» здесь — это использование для разделения команд пары амперсандов (`&&`), которая обычно читается как «И».

На уроках логики и философии учат правилу: выражение «А И Б» истинно тогда и только тогда, когда оба условия, А и Б, истинны. Следовательно, если А ложно, то нет необходимости даже рассматривать Б. Например, возьмем такое выражение: «У меня есть собака, И у меня есть кошка». Если у меня нет собаки, то это составное выражение неверно, независимо от наличия у меня кошки.

Применим это правило в bash. Напомним, что основная функция bash – выполнять команды. В первой части нашего примера выполняется команда `cd`. В соответствии с логикой, если эта первая команда потерпела неудачу, то bash не будет выполнять вторую команду `rm`.

Оператор `&&` позволяет использовать логическое «И». На самом деле bash не выполняет логическую операцию с двумя результатами (в C/C++ была бы другая логика при таком же синтаксисе). Эта идиома просто обеспечивает условное выполнение второй команды, которая не запускается, если первая команда завершилась с ошибкой.

Теперь вернемся к исходному примеру:

```
[[ -n "$DIR" ]] && cd "$DIR"
```

Стал ли он теперь более понятным? Первое выражение проверяет, отличается ли длина значения переменной `DIR` от нуля. Если переменная имеет некоторое значение, то есть длина этого значения отлична от нуля, то команда `cd` попытается перейти в каталог, имя которого соответствует значению `DIR`.

То же самое можно было бы записать, явно использовав оператор `if`:

```
if [[ -n "$DIR" ]]; then
    cd "$DIR"
fi
```

Для тех, кто плохо знаком с bash, этот последний фрагмент, безусловно, выглядит понятнее. Но внутри ветки `then` выполняется очень мало действий, только команда `cd`, поэтому такой синтаксис условного выполнения кажется излишне громоздким. Вам придется решать, какой синтаксис использовать, исходя из того, кто будет читать ваш код

и насколько высока вероятность добавления других команд в ветку `then`. В следующих разделах мы рассмотрим несколько возможных вариантов.



### Справка bash

Команда `help` в bash служит отличным источником информации о встроенных командах, а команда `help test` дает ценные подсказки о выражениях проверки условий, таких как `-n`. Также можно заглянуть в справочное руководство `man bash`, тогда придется отыскать раздел Conditional expressions (условные выражения). Но справочник `man` — очень объемный, а команда `help` дает короткие и содержательные подсказки. Если вы не уверены, встроена ли та или иная команда в bash, то просто передайте ее команде `help` или введите: `type -a команда`.

Если бы вы знали, что `help test` расскажет вам о значении `-n`, то, может, и не купили бы эту книгу. И еще один тонкий момент: попробуйте команду `help [`. Пожалуйста.

## Или ELSE...

Существует похожая идиома, основанная на использовании символов `||` для разделения двух элементов в команде bash. Эта пара символов читается как «ИЛИ»: вторая часть команды будет выполнена, только если первая закончится неудачей. Такой образ действий напоминает логическое «ИЛИ», например: А ИЛИ Б. Все выражение истинно, если истинна хотя бы одна из его частей, А или Б. Иными словами, если А истинно, то не имеет значения, истинно ли Б. Например, рассмотрим такое выражение: «У меня есть собака ИЛИ кошка». Если у меня действительно есть собака, то это выражение истинно, независимо от наличия кошки.

Применим это правило в bash:

```
[[ -z "$DIR" ]] || cd "$DIR"
```

Сможете объяснить, как работает это выражение? Если значение переменной имеет нулевую длину, то первая часть будет оценена как «истинная» и вторая половина выполняться не будет, то есть команда `cd`

не будет выполнена. Но если значение \$DIR имеет ненулевую длину, то проверка вернет «ложь» и команда cd выполнится.

Эту строку на языке bash можно прочитать так: «Либо \$DIR имеет нулевую длину, либо попытаться перейти в этот каталог».

Запись тех же действий с использованием оператора if выглядит немного странно, потому что ветка then — пустая. Код после || похож на ветку else:

```
if [[ -z "$DIR" ]]; then
:
else
    cd "$DIR"
fi
```

Двоеточие (:) — это пустая инструкция, которая ничего не делает.

Итак, две команды, разделенные символами &&, похожи на оператор if и его ветку then; две команды, разделенные символами ||, похожи на оператор if и его ветку else.

## Выполняем несколько команд

Если требуется выполнить несколько команд после пары символов ||, как в ветке else, или после &&, как в ветке then, то нередко допускаются ошибки. Например, может возникнуть соблазн написать такой код:

```
# Внимание: этот код работает не так, как можно предположить!
cd /tmp || echo "cd to /tmp failed." ; exit
```

Оператор «ИЛИ» говорит нам, что в случае сбоя cd выполнится команда echo, которая сообщит пользователю, что cd потерпела неудачу. Но вот в чем загвоздка: exit выполнится в любом случае. Вы этого не ожидали, верно?

Интерпретируйте точку с запятой (;) как эквивалент перевода строки, и все сразу станет на свои места (и выяснится, что это не то, чего вы хотели):

```
cd /tmp || echo "cd to /tmp failed."  
exit
```

Можно ли добиться желаемого результата? Да, для этого следует сгруппировать `echo` и `exit` в одно предложение справа от «ИЛИ», например:

```
# Или cd выполнится успешно, или сценарий завершится с сообщением об ошибке  
cd /tmp || { echo "cd to /tmp failed." ; exit ; }
```

Фигурные скобки в bash используются для определения составных команд, то есть для группировки инструкций. Возможно, вы видели нечто подобное с использованием круглых скобок, но инструкции, заключенные в круглые скобки, выполняются в подоболочке, также называемой дочерним процессом. Это связано с ненужными в данном случае расходами ресурсов, к тому же выход по команде `exit` произойдет из подоболочки, что не даст желаемого результата.



### Завершение составных команд

Синтаксис bash требует в обязательном порядке завершать составные команды точкой с запятой или переводом строки перед закрывающей фигурной скобкой. Если используется точка с запятой, то она должна отделяться пробелом от закрывающей фигурной скобки, чтобы интерпретатор распознал ее как служебный символ (иначе она будет перепутана с закрывающей фигурной скобкой синтаксиса переменных оболочки, например `$(VAR)`). Вот почему предыдущий пример заканчивается, казалось бы, лишней точкой с запятой: `{ echo "..."; exit; }`. При использовании перевода строки завершающая точка с запятой не нужна:

```
# Или cd выполнится успешно, или сценарий завершится с сообщением  
об ошибке  
cd /tmp || { echo "cd to /tmp failed." ; exit  
}
```

но такой код воспринимается неоднозначно. Закрывающая фигурная скобка у левого края будет выглядеть странно; если же добавить отступ, как в примере выше, то такая строка кажется голой. Мы рекомендуем использовать дополнительную точку с запятой, не забывая о пробеле между ней и закрывающей фигурной скобкой.

## Еще о случае нескольких команд

Что, если требуется реализовать более сложную логику, например с несколькими конструкциями «И» и «ИЛИ»? Как их объединить? Рассмотрим следующую строку кода:

```
[ -n "$DIR" ] && [ -d "$DIR" ] && cd "$DIR" || exit 4
```

Если переменная `DIR` — непустая и существует каталог с таким именем, то `cd` выполнит переход в этот каталог; иначе сценарий завершится с кодом 4. Эта группа команд делает именно то, что можно было бы ожидать, но правильно ли вы понимаете логику?

При взгляде на этот пример можно подумать, что оператор `&&` имеет более высокий приоритет, чем `||`, но в действительности это не так. Они выполняются в порядке следования слева направо. В `bash` операторы `&&` и `||` имеют одинаковый приоритет и являются левоассоциативными. Хотите доказательств? Взгляните на следующие примеры:

```
# Пример 1
$ echo 1 && echo 2 || echo 3
1
2
$
```

```
# Пример 2
$ echo 1 || echo 2 && echo 3
1
3
$
```

Обратите внимание, что крайняя левая команда выполняется всегда, независимо от следующего за ней оператора: порядок вычислений определяется не приоритетом операторов, а их последовательностью.

## Так делать не нужно!

Пока мы не ушли далеко от главной темы главы, рассмотрим примеры использования оператора `if`, которые типичны для сценариев, написанных

много лет назад. Мы показываем их, чтобы дополнительно пояснить идиому «большого» if, а также убедить вас никогда не подражать этому стилю. Итак, вот код:

```
### Не используйте операторы if для таких проверок
if [ $VAR"X" = X ]; then
    echo empty
fi

### Или таких
if [ "x$VAR" == x ]; then
    echo empty
fi
### И других, подобных им
```

Здесь всего лишь выполняется проверка, не является ли переменная VAR пустой. Для этого к ее значению добавляется некоторый символ (в этих примерах X и x), и, если в результате получается строка, совпадающая только с этим символом, значит, переменная имеет пустое значение. Не делайте так. Есть лучшие способы выполнить такую проверку. Вот простая альтернатива:

```
# Значение переменной имеет нулевую длину?
if [[ -z "$VAR" ]]; then
    echo empty
fi
```



### Одиночные и двойные квадратные скобки

В примерах кода выше проверяемое условие заключено в одиночные квадратные скобки [ ]. Но главная проблема не в них. В первую очередь, мы рекомендуем избегать приема с добавлением значения и сравнением строк — для таких проверок используйте ключи `-z` или `-n`. Почему же в других наших примерах в операторах `if` и заменяющих их конструкциях используются двойные квадратные скобки [[ и ]]? Они являются дополнением, появившимся в bash и отсутствующим в оригинальной командной оболочке sh. Благодаря этому исключаются некоторые чреватые ошибками ситуации, например когда имя переменной в одних случаях заключено в кавычки, а в других — нет. Мы использовали в двух примерах выше одиночные квадратные скобки, потому что код такого вида часто встречается в старых сценариях. Возможно, вам придется использовать одиночные скобки, если приоритетом является совместимость между различными платформами, в том числе не поддерживающими bash

(например, использующими dash). Дополнительно отметим, что двойные квадратные скобки — это ключевые слова, тогда как левая одиночная квадратная скобка — это встроенная функция. Это объясняет некоторые тонкие отличия в их свойствах. Мы советуем по возможности всегда использовать двойные квадратные скобки.

В случае, когда длина значения переменной или строки не равна нулю, можно использовать ключ `-n` или просто сослаться на переменную:

```
# Проверяет, что значение переменной имеет ненулевую длину
if [[ -n "$VAR" ]]; then
    echo "VAR has a value:" $VAR
fi

# То же самое
if [[ "$VAR" ]]; then
    echo even easier this way
fi
```

Как видите, нет необходимости использовать подход, который был нужен в устаревших версиях команды `test` («[»). Однако мы посчитали, что вы должны знать о нем, чтобы понимать старые сценарии. Кроме того, теперь вы знаете лучший способ добиться того же результата.

## В заключение: стиль и удобочитаемость

В этой главе мы рассмотрели важную идиому bash — проверку условия без оператора `if`. Такая проверка не похожа на традиционную конструкцию `if/then/else`, но позволяет получать те же результаты. Если не знать о ней, то некоторые сценарии могут остаться для вас неясными. Эту идиому целесообразно использовать для проверки всех предварительных условий перед выполнением команды или быстрой проверки ошибок, не нарушая основной логики сценария.

С помощью операторов `&&` и `||` можно реализовать логику `if/then/else`. Но в bash есть также и ключевые слова `if`, `then` и `else`, поэтому возникает вопрос: когда использовать их, а когда сокращенные конструкции? Ответ: все зависит от удобочитаемости.

Для определения сложной логики лучше использовать знакомые ключевые слова. А для простых проверок с одиночными командами часто удобнее операторы `&&` и `||`, потому что они не отвлекают внимание от основного алгоритма. Используйте `help test`, чтобы вспомнить, какие проверки выполняют, например, ключи `-n` и `-r`, и скопируйте текст справки в памятку на будущее.

В любом случае и в знакомых операторах `if`, и в идиоматических проверках без `if` мы рекомендуем использовать синтаксис с двойными квадратными скобками.

Теперь, подробно рассмотрев одну идиому, давайте взглянем на другие, чтобы поднять на новый уровень ваши умения программировать на bash.

## ГЛАВА 2

---

# ЯЗЫК ЦИКЛОВ

[https://t.me/it\\_boooks](https://t.me/it_boooks)

В bash имеются не только циклы `for` в стиле C, но также другие виды и стили организации циклического выполнения инструкций. Некоторые из них ближе программистам на Python, но у каждого есть свои особые задачи. Существует цикл `for` без очевидных аргументов, который удобно использовать как в сценариях, так и внутри функций. Также есть похожий на итератор цикл `for`, значения для которого могут задаваться явно или возвращаться другими командами.

## Циклические конструкции

Циклические конструкции распространены в языках программирования. С момента изобретения языка C многие языки программирования заимствовали из него цикл `for`. Это мощная и понятная конструкция, объединяющая код инициализации, условие завершения и код, выполняемый в начале каждой итерации. В C, Java и многих других языках цикл `for` выглядит следующим образом:

```
/* HE bash */
for (i=0; i<10; i++) {
    printf("%d\n", i);
}
```

В bash используется тот же подход, хотя и с некоторыми различиями в синтаксисе:

```
for ((i=0; i<10; i++)); do
    printf '%d\n' "$i"
done
```

Во-первых, обратите внимание на двойные круглые скобки. Во-вторых, для обозначения блока инструкций, составляющих тело цикла, в bash вместо фигурных скобок используются ключевые слова `do` и `done`. Так же как в C/C++, идиоматическое использование цикла `for` — это пустой бесконечный цикл (позже мы также покажем конструкцию `while true; do`):

```
for ((;)); do
    printf 'forever'
done
```

Но это не единственный вид цикла `for` в bash. В сценариях широко используется следующая идиома:

```
for value; do
    echo "$value"
    # Что-то сделать со значением $value...
done
```

На первый взгляд, этой конструкции чего-то не хватает, не так ли? Откуда `value` получает свои значения? Если записать такой цикл в командной строке, он ничего не выведет. Однако в сценарии оболочки такой цикл `for` будет перебирать параметры командной строки. То есть он последовательно будет присваивать переменной `value` значения `$1`, `$2`, `$3` и т. д.

Поместите этот цикл `for` в файл с именем `myloop.sh` и запустите его, как показано ниже. В результате он выведет три аргумента (`-c`, `17`, `core`):

```
$ bash myloop.sh -c 17 core
-c
17
core
$
```

Такую краткую форму цикла `for` также часто можно встретить в определениях функций:

```
function Listem {  
    for arg; do  
        echo "arg to func: '$arg'"  
    done  
    echo "Inside func: \$0 is still: '\$0'"  
}
```

Внутри определения функции переменные `$1`, `$2` и т. д. представляют собой параметры функции, а не сценария. Поэтому цикл `for` будет перебирать параметры, переданные функции.

Этот минималистский цикл `for` выполняет итерацию по подразумеваемому списку значений — параметров, передаваемых сценарию или функции. При использовании в основной части сценария он перебирает параметры, переданные сценарию; внутри функции он перебирает параметры, переданные этой функции.

Это одна из малоизвестных идиом bash. Но вы должны понимать, как она работает, и в следующем разделе мы вернемся к ее обсуждению (как говорят adeptы Python, явное лучше неявного).

Вам может понравиться такая же простая запись цикла, но с явными значениями на наш выбор, не ограниченный параметрами. В bash есть все, что для этого нужно.

## Явные значения

В bash можно передать в цикл `for` список значений, например:

```
for num in 1 2 3 4 5; do  
    echo "$num"  
done
```

Поскольку bash поддерживает работу со строками, мы не ограничены только числами:

```
for person in Sue Neil Pat Harry; do  
    echo $person  
done
```

Список значений может включать не только литералы, но и переменные:

```
for person in $ME $3 Pat ${RA[2]} Sue; do
    echo $person
done
```

Также источником значений для цикла `for` могут быть результаты выполнения команд — отдельных или их конвейеров:

```
for arg in $(some cmd or other | sort -u)
```

Вот еще несколько примеров:

```
for arg in $(cat /some/file)
for arg in $(< /some/file) # Faster than shelling out to cat
for pic in $(find . -name '*.jpg')
for val in $(find . -type d | LC_ALL=C sort)
```

Типичная запись цикла `for`, особенно в старых сценариях, выглядит примерно так:

```
for i in $(seq 1 10)
```

Здесь команда `seq` генерирует последовательность чисел. Эквивалент такой записи цикла:

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

Последний вариант представления цикла `for` более эффективен и, вероятно, более понятен (обратите внимание, что в этих двух формах значение `i` будет различаться после завершения цикла — 10 в первом случае и 11 во втором, хотя вне цикла это значение обычно не используется).

Существует еще один вариант, но у него есть проблемы совместимости с разными версиями `bash`, потому что поддержка фигурных скобок появилась в версии 3.0, а дополнение нулями числовых значений — в версии 4.0:

```
for i in {01..10}; do echo "$i"; done
```



### Нули в начале числа

В bash версии 4.0 и выше, если любой из первых двух членов в выражении вида `{начало..конец..шаг}` начинается с нуля, то переменная цикла будет принимать значения одинаковой длины и дополняться нулями слева. То есть при использовании выражения `{098..100}` переменная цикла будет последовательно принимать значения `098, 099, 100`, а в случае `{98..0100}` каждое значение будет иметь длину в 4 символа: `0098, 0099, 0100`.

Конструкция в фигурных скобках особенно удобна, когда требуется, чтобы генерируемые числа были частью строки. Для этого достаточно поместить конструкцию в фигурных скобках в запись строки. Например, пять имен файлов от `log01.txt` до `log05.txt` можно сгенерировать следующим образом:

```
for filename in log{01..5}.txt ; do
    # Сделать что-то с очередным именем файла
    echo $filename
done
```



### Фигурные скобки или printf -v?

Тот же результат можно получить с помощью числового цикла `for`, использовав команду `printf -v` для конструирования имени файла из чисел, но применение фигурных скобок выглядит проще. Используйте числовой цикл `for` и `printf` для более сложных задач, чем генерация имен файлов.

Для создания последовательности чисел с плавающей точкой очень удобно использовать команду `seq`. Вы указываете начальное значение, приращение на каждом шаге и конечное значение, например:

```
for value in $(seq 2.1 0.3 3.2); do
    echo $value
done
```

Получаем следующую последовательность:

2.1  
2.4

2.7

3.0

Помните, что bash не поддерживает арифметику с плавающей точкой. Но такие значения могут понадобиться для передачи из сценария какой-то другой программе.

## Почти как в Python

Вот еще одна конструкция, часто встречающаяся в циклах `for` в bash:

```
for person in ${name_list[@]}; do
    echo $person
done
```

Такой цикл может произвести, например, следующий вывод:

```
Arthur
Ann
Henry
John
```

Внешне эта конструкция похожа на цикл `for` в Python, где можно перебирать значения, возвращаемые итератором. В этом примере bash перебирает ряд значений, но они поступают не от итератора — все имена известны до начала цикла.

`${name_list[@]}` — это конструкция перечисления всех значений в массиве, который далее мы будем называть *списком* (подробное обсуждение терминологии вы найдете во введении к главе 7; в этом примере список называется `name_list`). Подстановка производится при подготовке команды к выполнению. Содержимое массива извлекается до передачи управления оператору `for`, то есть цикл получает значения, как если бы они были введены явно:

```
for person in Arthur Ann Henry John
```

А как обстоит дело со словарями? Для коллекций, которые в Python называются «словарями», в bash используется термин «ассоциативные массивы», а в некоторых других языках — «пары ключ/значение» или

«хеши» (см. введение к главе 7). Для работы с парами ключ/значение можно использовать конструкцию  `${hash[@]}` . Чтобы выполнить итерации только по ключам (то есть индексам) хеша, добавьте восклицательный знак и используйте конструкцию  `${!hash[@]}` , как показано ниже:

```
# Объявляем хеш (то есть массив пар ключ/значение)
declare -A hash
# Записываем данные в хеш
while read key value; do
    hash[$key]="$value"
done
# Показываем содержимое хеша, хотя оно может
# выводиться не в порядке записи
for key in "${!hash[@]}"; do
    echo "key $key ==> value ${hash[$key]}"
done
```

Вот еще один пример:

```
# Объявляем хеш (то есть массив пар ключ/значение)
declare -A hash
# Записываем данные в хеш: слова и количества вхождений
while read word count; do
    let hash[$word]+="$count"
done
# Показываем содержимое хеша, хотя мы не можем управлять
# порядком элементов
for key in "${!hash[@]}";do
    echo "word $key count = ${hash[$key]}"
done
```

Эта глава в основном посвящена конструированию циклов, таких как `for`, поэтому за дополнительными подробностями и примерами списков и хешей обращайтесь к главе 7.

## Кавычки и пробелы

Есть еще один важный аспект, который следует учитывать при написании циклов `for`. Вы наверняка обратили внимание на использование кавычек в предыдущем примере. Причина в том, что если значения в списке имеют пробелы (например, если каждый элемент списка

содержит имя и фамилию), то мы можем получить неожиданный результат. Рассмотрим пример:

```
for person in ${namelist[@]}; do
    echo $person
done
```

Такой цикл может произвести следующий вывод:

```
Art
Smith
Ann
Arundel
Hank
Till
John
Jakes
```

Получив список с четырьмя именами и фамилиями, цикл `for` вывел восемь отдельных значений. Почему? Ответ заключается в механизме подстановки в конструкцию  `${namelist[@]}`  — bash просто помещает значения из массива на место выражения с переменной. В результате получается список из восьми слов:

```
for person in Art Smith Ann Arundel Hank Till John Jakes
```

Цикл `for` получает список слов, но не знает, откуда они взялись.

Для решения этой проблемы в bash предусмотрен синтаксис с кавычками: поместите выражение списка в кавычки, и тогда каждый элемент будет заключен в кавычки. Выражение:

```
for person in "${namelist[@]}"
```

будет преобразовано в:

```
for person in "Art Smith" "Ann Arundel" "Hank Till" "John Jakes"
```

и даст желаемый результат:

```
Art Smith
Ann Arundel
Hank Till
John Jakes
```

Если вы собираетесь использовать цикл `for` для перебора списка имен файлов, обязательно используйте кавычки, потому что в именах файлов могут встречаться пробелы.

И еще один момент, о котором стоит упомянуть. В синтаксисе списка можно использовать знаки `*` или `@` для перечисления всех его элементов. Конструкция  `${namelist[*]}`  дает тот же результат, за исключением случая, когда она заключена в кавычки: `"${namelist[*]}"`  вернет все значения внутри одной строки. Например:

```
for person in "${namelist[*]}"; do  
    echo $person  
done
```

выведет одну строку:

```
Art Smith Ann Arundel Hank Till John Jakes
```

Иногда требуется именно вывод в одну строку, но в цикле `for` это означает выполнение только одной итерации. Мы советуем использовать символ `@`, если вы не уверены, что вам нужен именно символ `*`.

Дополнительную информацию вы найдете в разделе «Кавычки» главы 11.

## Разработка и тестирование циклов for

Обработка списка в цикле чрезвычайно удобна. Два очевидных случая ее применения: запуск SSH-команд со списком серверов и переименование файлов, например `for file in *.JPEG; do mv -v $file ${file/JPEG/jpg}; done`. Но как разработать и протестировать сценарий или даже простую команду `for`? Точно так же, как разрабатывается все остальное: начните с простого и постепенно двигайтесь вперед. Для проверки используйте `echo` (см. пример 2.1). Обратите внимание, что встроенная команда `echo` поддерживает ряд интересных параметров, но не соответствует стандарту POSIX (см. раздел «Вывод POSIX» в главе 6). Наиболее интересными и часто используемыми операторами являются `-e` (выключает интерпретацию символа, следующего за обратной косой чертой) и `-n` (подавляет автоматический переход на новую строку).

### Пример 2.1. Переименование файлов — тестовая версия

```
### Конструируем и проверяем команду rename, обратите внимание на echo
for file in *.JPEG; do echo mv -v $file ${file/JPEG/jpg}; done

### Выполняем команды SSH на нескольких узлах, обратите внимание на первую
### команду echo (весь код можно записать в одну строку, но мы
### отформатировали его, чтобы уместить по ширине книжной страницы)
for node in web-server{00..09}; do
    echo ssh $node 'echo -e "$HOSTNAME\t$(date "+%F") $(uptime)"'; 
done
```

Как только код заработает должным образом, удалите первую команду `echo`. Но имейте в виду, что при использовании перенаправления в блоке вам, возможно, придется заменить `|` на `.p.`, `>` на `.gt.` и т. д., пока не будут пройдены все этапы.



### Выполнение одной и той же команды на нескольких хостах

Хотя этот вопрос выходит за рамки данной книги, мы бы хотели отметить, что для запуска одной и той же команды на нескольких хостах предпочтительно использовать специальные инструменты, например Ansible, Chef или Puppet. Однако если нужно быстро выполнить пару команд, то подойдет один из следующих инструментов:

#### *clusterssh*

Написан на Perl, открывает множество неуправляемых терминалов в окнах.

#### *mssh (MultiSSH)*

SSH-клиент с графическим интерфейсом на основе GTK+, выполняется в одном окне.

#### *mussh*

Сценарий командной оболочки для работы с несколькими хостами.

#### *pconsole*

Консольный тайловый оконный менеджер, создающий отдельный терминал для каждого хоста.

#### *multixterm*

Написан на Expect и Tk, управляет несколькими терминалами xterm.

#### *PAC Manager*

Графический интерфейс в стиле SecureCRT для Linux, написанный на Perl.

## Циклы `while` и `until`

Мы уже упоминали циклы `while` выше. В bash они работают в полном соответствии с ожиданиями — тело цикла выполняется, пока код условия выхода не обнулится:

```
while <УСЛОВИЕ>; do <ТЕЛО>; done
```

Такие циклы часто используются для чтения файлов, как будет показано в главе 9, а иногда — для анализа параметров, как изложено в разделе «Анализ ключей» в главе 8.

В отличие от других языков, в bash цикл `until` полностью эквивалентен циклу `! while`. Он действует по принципу: «выполнять тело цикла, пока код условия выхода нулевой»:

```
until <УСЛОВИЕ>; do <ТЕЛО>; done  
### То же самое:  
! while <УСЛОВИЕ>; do <ТЕЛО>; done
```

Это очень удобно в случае ожидания какого-либо события, например перезагрузки узла (пример 2.2).

### Пример 2.2. Ожидание перезагрузки

```
until ssh user@10.10.10.10; do sleep 3; done
```

## В заключение: стиль и удобочитаемость

В начале этой главы мы кратко рассмотрели цикл `for` в стиле C/C++. Поскольку bash в большей степени ориентирован на работу со строками, он поддерживает еще несколько стилей цикла `for`, о которых следует знать. Минималистская конструкция `for variable` обеспечивает неявный (и, возможно, малопонятный) обход аргументов сценария или функции. Явная передача в цикл `for` списка строковых или иных значений обеспечивает идеальный механизм для обхода всех элементов списка или всех ключей в хеше.

Теперь вы знаете, что  `${namelist[@]}`  и  `${namelist[*]}`  извлекают все значения из списка, но если эти две конструкции заключить в двойные кавычки, то они дадут разный результат: первая вернет каждый элемент в отдельной строке, а вторая — все элементы в одной строке. То же относится к специальным переменным командной оболочки  `$@`  и  `$*` . Обе вызывают список всех аргументов сценария (например,  `$1` ,  `$2`  и т. д.). Однако при заключении в двойные кавычки они дают разный результат: несколько строк и одну строку. Мы вспомнили об этом, только чтобы вернуться к простейшему циклу  `for` :

```
for param
```

и отметить, что он эквивалентен следующей конструкции:

```
for param in "$@"
```

Мы считаем вторую форму более удачной, потому что она явно показывает, какие значения перебирает цикл. Однако кто-то может возразить, что само имя переменной  `$@`  и необходимость заключать ее в кавычки относятся к специфическим особенностям  `bash` , которые менее понятны для неискушенного пользователя, чем первая минималистская форма. Если вы предпочитаете первый вариант, то просто добавьте комментарий:

```
for param # Итерации по всем аргументам сценария
```

Когда итерации выполняются по последовательности целочисленных значений, наиболее читабельным и эффективным, пожалуй, является цикл  `for`  в стиле С с двойными круглыми скобками. Кстати, если эффективность имеет большое значение, объявите переменную цикла как целочисленную, добавив инструкцию  `declare -i i`  в начало сценария, это позволит избежать ресурсоемких преобразований из строки в число и обратно.

Каковы возможности обработки данных в циклах? Данные можно анализировать, принимая решения на основе результатов этого анализа. Здесь мы подошли к еще одной важной особенности  `bash`  —

сверхмощному и чрезвычайно гибкому оператору `case`, о котором мы поговорим в следующей главе.

Циклы `for` чрезвычайно полезны, но могут приводить к ошибкам. Начинайте с простой конструкции и используйте `echo`, пока не убедитесь, что ваша команда работает должным образом. И помните о «синтаксическом сахаре» `while` и `until`, помогающем улучшить читаемость кода.

## ГЛАВА 3

---

# На всякий случай: оператор Case

[https://t.me/it\\_boooks](https://t.me/it_boooks)

Во многих языках программирования для множественного ветвления предусмотрен специальный оператор, заменяющий цепочки операторов `if/then/else`. В bash имеется оператор `case`, который обеспечивает мощные возможности сравнения с образцом и очень удобен во многих сценариях.

## Сделайте свой выбор

Ключевые слова `case` и `in` определяют значение, которое требуется сравнить с различными шаблонами. Вот пример:

```
case "$var" in
    yes ) echo "glad you agreed" ;;
    no  )
        echo "sorry; good bye"
        exit
    ;;
    *   ) echo "invalid answer. try again" ;;
esac
```

Скорее всего, вы поняли смысл приведенного кода. Он проверяет совпадение значения переменной `$var` со словами «yes» и «no» и выполняет соответствующие инструкции. Также предусмотрено действие по умолчанию. Конец конструкции отмечается ключевым словом `esac`, то есть перевернутым словом `case`. Этот пример довольно легко читается, но демонстрирует лишь малую часть возможностей оператора `case`.

Обратите внимание также на разные стили оформления блоков: «однострочный» для «yes» и более типичный блок, заканчивающийся ;;, о котором мы поговорим ниже, для «no». Стиль оформления выбирается в зависимости от действий, которые необходимо выполнить, и из соображений удобочитаемости.



### Скобки в case

Синтаксис оператора `case` требует необязательную открывающую круглую скобку ( для каждой закрывающей скобки ). Скажем, в предыдущем примере мы могли бы написать ( "yes" ) вместо "yes" ) и так же оформить другие строки. Однако такой стиль редко используется на практике. В конце концов, почти никому не хочется набирать лишний символ.

Мощь оператора `case` и идиоматический способ применения обусловлены поддержкой сопоставления с шаблонами:

```
case "$var" in
    [Nn][Oo]* )
        echo "Fine. Leave then."
        exit
    ;;
    [Yy]?? | [Ss]ure | [Oo][Kk]* )
        echo "OK. Glad we agree."
    ;;
    * ) echo 'Try again.'
        continue
    ;;
esac
```

Давайте коротко рассмотрим принципы поддержки сопоставления с шаблонами в bash, которые многим из вас знакомы по *подстановочным знакам* командной строки, обычно используемым при передаче имен файлов. Есть три подстановочных знака:

- вопросительный знак ? соответствует точно одному символу;
- звездочка \* соответствует любому количеству (включая нулевое) символов;
- квадратные скобки [ ] соответствуют любому из символов, заключенных в них.

В нашем примере конструкция [Yy] интерпретируется как «либо прописная буква Y, либо строчная y». Конструкция [Nn][Oo]\* соответствует прописной или строчной N и последующей прописной или строчной O, за которыми следует любое количество любых других символов. Шаблон соответствует, например, следующим словам: *no, No, nO, NO, noway, Not Ever, por...* И он не совпадет со значением *never* в \$var.

Сможете ли вы определить возможные значения для «утвердительного» случая? Вертикальная черта разделяет разные шаблоны, соответствующие одному варианту (читайте их как логическое «ИЛИ», но не как ||.) С этим шаблоном совпадут, например, слова *Yes, yes, YES, yEs, yES, yup, Sure, sure, OK, ok, OKfine* и *OK why not*. Но не совпадут *ya, SURE, oook* и многие другие.

Вариант по умолчанию — шаблон, совпадающий с чем угодно. Если ни с одним из предыдущих шаблонов не было обнаружено совпадений, то с этим совпадение обнаружится всегда — он соответствует любому количеству любых символов. Поэтому разработчики сценариев на bash помещают его последним в списке, если хотят обработать случай по умолчанию.



### Шаблоны — это не регулярные выражения

Сопоставление с шаблоном в операторе `case` не является сопоставлением с регулярным выражением. В bash регулярные выражения поддерживаются только в операторе `if`, использующем оператор сравнения `=~`. Если вам потребуются регулярные выражения, то вместо `case` используйте последовательность операторов `if/then/else`.

## Применение на практике

Обычно оператор `case` используется для анализа параметров командной строки. Давайте рассмотрим простой и реалистичный пример сценария, демонстрирующий правильное применение этого оператора.

## Задача

Пользователям Linux или Unix часто приходится использовать команду `ls`, чтобы вывести список имен файлов и связанную с ними информацию. Команда `ls` поддерживает ряд параметров, позволяющих передать ей дополнительную информацию или отсортировать результаты определенным образом. Если вы привыкли использовать `ls`, то можете создать для нее несколько псевдонимов или даже сценариев. Эти сценарии будут иметь схожую функциональность, но действовать немного по-разному. Можно ли объединить их в один сценарий?

В качестве образца можно принять Git — популярное программное обеспечение для управления исходным кодом. У него есть несколько родственных, но разных функций, все они вызываются обращением к одной команде `git`, но отличаются вторым ключевым словом, например `git clone`, `git add`, `git commit`, `git push` и т. д.

## Наш сценарий

Подход на основе «подкоманд» можно применить и в нашей ситуации. С `ls` связаны несколько востребованных на практике функций: вывод файлов в порядке длины их имен (и размеров самих файлов); вывод только самых длинных имен файлов; вывод только последних недавно измененных файлов; вывод имен файлов с применением цветового кодирования, определяющего тип файла, — это стандартная функция `ls`, но для ее активации требуется помнить ряд параметров.

Наш сценарий мы назовем `list`, и он будет принимать второе слово, задающее одну из функций: `color`, `last`, `length` или `long`. В примере 3.1 показан исходный код сценария.

### Пример 3.1. Простой сценарий-обертка, использующий case

```
#!/usr/bin/env bash
# list.sh: сценарий-обертка для инструментов, связанных с ls,
#           и простая демонстрация оператора 'case..esac'
# Автор и дата: _bash Idioms_ 2022
# Имя файла в bash Idioms: examples/ch03/list.sh
# _____
```

```
VERSION='v1.2b'

function Usage_Exit {
    echo "$0 [color|last|len|long]"
    exit
}

# Перед каждым именем файла вывести длину имени и отсортировать по длине имен.
# Обратите внимание, что символ '-' можно использовать в именах функций,
# но не в именах переменных. Мы обычно не используем этот символ, но вы можете.
function Ls-Length {
    ls -1 "$@" | while read fn; do
        printf '%3d %s\n' ${#fn} ${fn}
    done | sort -n
}

(( $# < 1 )) && Usage_Exit ❶
sub=$1
shift

case $sub in
    color)                                # Использовать выделение цветом
        ls -N --color=tty -T 0 "$@"
        ;;
    last | latest)                         # Последние измененные файлы ❷
        ls -lrt | tail "-n${1:-5}" ❸
        ;;
    len*)                                  # Вывести имена файлов с их длинами ❹
        Ls-Length "$@"
        ;;
    long)                                  # Файлы с самыми длинными именами
        Ls-Length "$@" | tail -1
        ;;
    *)                                     # По умолчанию
        echo "unknown command: $sub"
        Usage_Exit
        ;;
esac
```

Мы не будем объяснять работу этого сценария, хотя к концу этой книги вы узнаете обо всех используемых в нем возможностях, а сосредоточимся на применении оператора `case`:

- ❶ Узнали логику проверки условия без `if`? Если нет, то перечитайте главу 1.
- ❷ Простой выбор «ИЛИ» между двумя словами.
- ❸ Используется команда `tail -n5`, если значение в `$1` не указано, — см. раздел «Значения по умолчанию» в главе 4.
- ❹ Шаблон соответствует любому слову, начинающемуся с «len», такому как «len», «length» «lenny» или «lens». Совпадающее значение вызовет функцию `ls-Length`, которая определена в начале сценария, и передаст ей полученные сценарием аргументы командной строки.

## Сценарии-обертки

Все мы занятые люди, и нам приходится многое запоминать, поэтому, когда появляется возможность написать сценарий, освобождающий нас от запоминания не самой важной информации, это всегда приятно. В примере 3.1 мы показали один из способов создания «сценария-обертки», но существует большое количество других интересных приемов, которые можно использовать в зависимости от сложности решаемой задачи или деталей, которые требуется «запомнить». В примере 3.1 мы вызывали функцию или просто вставляли фрагменты кода. Такой способ лучше подходит для очень коротких блоков кода, обычных для сценариев-оберток подобного типа.

Если ваша задача требует более сложного решения и вы используете существующие инструменты, то можете вызвать их или вспомогательные сценарии, хотя при этом вам придется настроить проверку ошибок и, возможно, передачу параметров. Этот подход можно комбинировать с подключаемыми каталогами (см. раздел «Подключаемые каталоги» в главе 9), чтобы получать необходимые модули, делегируя обслуживание блоков кода другим людям.

Следующий более объемный пример — это упрощенная и сокращенная версия сценария, который мы использовали в процессе редакционно-издательской подготовки этой книги. AsciiDoc — отличный инструмент, но нам приходится работать со многими языками разметки, и различия между ними стираются в памяти, поэтому мы решили написать инструмент, показанный в примере 3.2.

### Пример 3.2. Сложный сценарий-обертка, использующий case

```
#!/usr/bin/env bash
# wrapper.sh: простой демонстрационный "сценарий-обертка"
# Автор и дата: _bash Idioms_ 2022
# Имя файла в bash Idioms: examples/ch03/wrapper.sh
#
# Простейшие проверки ①
[ -n "$BOOK_ASC" ] || {
    echo "FATAL: export \$BOOK_ASC to the location of the Asciidoc files!"
    exit 1
}

\cd "$BOOK_ASC" || {
    echo "FATAL: can't cd to '$BOOK_ASC'!"
    exit 2
}

SELF="$0" ②

action="$1" ③
shift ④
[ -x /usr/bin/xsel -a $# -lt 1 ] && { ⑤
    # Чтение/запись в буфер обмена в Linux
    text=$(xsel -b)
    function Output {
        echo -en "$*" | xsel -bi
    }
} || {
    # Чтение/запись STDIN/STDOUT
    text=$*
    function Output {
        echo -en "$*"
    }
}

case "$action" in ⑥
```

```
#####
# Содержимое/разметка ⑦

### Заголовки ⑧
h1 )           # Заголовок уровня 1 (в AsciiDoc h3) ⑨
    Output "[[$($SELF id $text)]]\n==== $text" ⑩
;;
h2 )           # Заголовок уровня 2 (в AsciiDoc h4)
    Output "[[$($SELF id $text)]]\n===== $text"
;;
h3 )           # Заголовок уровня 3 (в AsciiDoc h5)
    Output "[[$($SELF id $text)]]\n===== $text"
;;

### Списки
bul|bullet )      # Маркированный список (** = уровень 2, + = многострочный)
    Output "* $text"
;;
nul|number|order* ) # Нумерованный список(.. = уровень 2, + = многострочный)
    Output ". $text"
;;
term )          # Термины
    Output "term_here:::\n $text"
;;

### Оформление внутри абзаца
bold )           # Полужирный
    Output "*$text*"
;;
i|italic*|itl )   # Курсив
    Output "_${text}_"
;;
c|constant|cons ) # Моноширинный (команды, код, ключевые слова и др.)
    Output "+$text+"
;;
type|constantbold ) # Полужирный моноширинный (ввод пользователя)
    Output "*+$text+*"
;;
var|constantitalic ) # Курсив моноширинный (пользовательские значения)
    Output "_+$text++_"
;;
sub|subscript )    # Нижний индекс
    Output "~$text~"
;;
sup|superscript )  # Верхний индекс
    Output "^$text^"
;;
```

```

foot )          # Сноска
    Output "footnote:[${text}]"
;;
url|link )      # URL с альтернативным текстом
    Output "link:$text$[$]" # URL [с отображаемым текстом]
;;
esc|escape )    # Экранирующий символ (например, *)
    Output "\$text\$" # $$*$$
;;
#####
# Инструменты ⑪

id )          ## Преобразовать имя рецепта в ID
    #us_text=${text// /_} # Пробелы в '_'
    #lc_text=${us_text,,} # Нижний регистр; только в bash 4+
    # Предварительно вызвать `tr -s '_' ''`, чтобы сохранить _ в случае
    # обработки ID дважды (например, в перекрестных ссылках "xref")
    # Длинные строки можно переносить, добавляя символ \ ⑫
    Output $(echo $text | tr -s '_' '' | tr '[[:upper:]]' '[[:lower:]]' \
        | tr -d '[:punct:]' | tr -s ' ' '_')
;;
index )         ## Создать 'index.txt' в каталоге AsciiDoc
    # Пример:
    # ch02.asciidoc:== утилиты для работы с текстом
    # ch02.asciidoc:== общие утилиты для работы с текстом и пр.
    # инструменты
    # ch02.asciidoc:== поиск данных
    egrep '^== ch*.asciidoc | egrep -v '^ch00.asciidoc' \
        > $BOOK_ASC/index.txt && {
        echo "Updated: $BOOK_ASC/index.txt"
        exit 0
    } || {
        echo "FAILED to update: $BOOK_ASC/index.txt"
        exit 1
    }
;;
rerun ) ## Запуск примеров для повторного создания (существующих!)
    ## выходных файлов
    # Запускать только для кода, для которого УЖЕ ИМЕЕТСЯ файл *.out...,,
    # но не для ВСЕГО кода *.sh
    for output in examples/*/*.out; do
        code=${output/out/sh}
        echo "Re-running code for: $code > $output"
        $code > $output
    done
;;

```

```
cleanup )          ## Очистка мусора XHTML/XML/PDF
    rm -fv {ch??,app?}.{pdf,xml,html} book.{xml,html} docbook-xsl.css
;;
* ) ❸
    \cd - # НЕУКЛЮЖИЙ способ отменить команду 'cd' выше...
    ( echo "Usage:" ❹
      egrep '\)[[:space:]]+\# ' $0 ❺
      echo ''
      egrep '\)[[:space:]]+\#\# ' $0 ❻
      echo ''
      egrep '\)[[:space:]]+\#\#\# ' $0 ) | grep "${1:-.}" | more ❾
;;
esac
```

В этом сценарии происходит много интересного, поэтому разберем его подробнее:

- ❶ Сценарий выполняет множество операций с исходным кодом формата AsciiDoc для этой книги, поэтому важно убедиться, что он запущен в нужном месте и имеется набор удобных переменных окружения.
- ❷ Обычно для хранения базового имени мы используем переменную `$PROGRAM`, но в данном случае сценарий будет вызываться рекурсивно, поэтому мы посчитали имя `$SELF` более понятным.
- ❸ Как мы обсудим подробнее в главе 11, использование осмысленных имен переменных вместо позиционных аргументов — хорошая практика, которой мы стараемся следовать.
- ❹ После сохранения операции, которую требуется выполнить, нам больше не нужен параметр `$1`, но в командной строке могут быть переданы другие параметры, поэтому сдвинем их командой `shift` на одну позицию влево.
- ❺ Если файл `/usr/bin/xsel` существует, имеет разрешение на выполнение и нет других параметров командной строки, то для чтения и записи используется буфер обмена X Window, иначе мы получаем текст из аргументов и отправляем результат в `STDOUT`. На практике мы копируем текст из редактора, переключаемся в командную строку, запускаем сценарий, переключаемся обратно и вставляем результат в редактор.

❶ Именно здесь начинается реальная работа. Прежде всего определяем, какое «действие» задано.

❷ Для удобочитаемости кода разделяем его на функциональные блоки (см. также п. ❻).

❸ Первый раздел — обработка разметки заголовков.

❹ Стока является одновременно и кодом, и документацией. Действие заключается в оформлении заголовка верхнего уровня `h1` (для кода книги). Позже мы увидим связь с документацией.

❺ Выполняем обработку. Сначала рекурсивно вызываем сценарий, чтобы получить идентификатор AsciiDoc для текста, затем выводим этот идентификатор в двойных квадратных скобках, следом — символ перевода строки `==`, соответствующий уровню заголовка, и, наконец, вызываем функцию `Output`. Надеюсь, остальной код вам понятен.

❻ Для удобочитаемости кода разделяем его на функциональные блоки (см. также п. ❷).

❼ Длинные строки можно переносить, добавляя символ `\` (см. также раздел «Форматирование» в главе 11).

❽ Далее следует еще одна порция интересного кода в варианте по умолчанию оператора `case`. Здесь подсказки о порядке использования при передаче неизвестных аргументов сочетаются с удобной функцией поиска в справке.

❾ Мы заключаем выходные данные в подоболочку, которую по конвейеру передаем команде `more` на случай, если вывод окажется слишком длинным.

❿ Стока кода как документация, о которой мы говорили в п. ❹. С помощью команды `egrep` мы извлекаем закрывающую скобку `)` из нашего оператора `case`, за которой следуют пробелы и один маркер начала комментария `#`. В результате мы получаем заголовок первого

функционального блока «Содержимое/разметка». Эта операция извлекает фактические строки кода, составляющие оператор `case`, и объясняет их действия благодаря комментариям.

❶ Описанное в п. ❸ действие выполняется для второго блока «Инструменты».

❷ Описанное в п. ❸ действие выполняется для третьего блока, обрабатывающего операции с репозиторием Git (мы опустили этот код для простоты). Здесь дополнительно используется команда `grep` с параметром  `${1:-.}`, чтобы показать либо подсказку по запросу, например `wrapper.sh help heading`, либо полный текст справки (`grep ".."`). В коротком сценарии это может показаться не особенно нужным, но, поскольку он со временем дополняется, такая возможность становится действительно удобной!

Результатом команд `grep` является отображение справочного сообщения, отсортированного и сгруппированного по разделам:

```
$ examples/ch03/wrapper.sh help
Usage:
h1 )                      # Заголовок уровня 1 (в AsciiDoc h3)
h2 )                      # Заголовок уровня 2 (в AsciiDoc h4)
h3 )                      # Заголовок уровня 3 (в AsciiDoc h5)
bul|bullet )               # Маркированный список (** = уровень 2, + =
                           # многострочный)
nul|number|order* )       # Нумерованный список (.. = уровень 2, + = многострочный)
term )                     # Термины
bold )                     # Полужирный
i|italic*|it1 )            # Курсив
c|constant|cons )          # Монодокументный (команды, код, ключевые слова и др.)
type|constantbold )        # Полужирный монодокументный (ввод пользователя)
var|constantitalic )       # Курсив монодокументный (пользовательские значения)
sub|subscript )             # Нижний индекс
sup|superscript )           # Верхний индекс
foot )                     # Сноска
url|link )                 # URL с альтернативным текстом
esc|escape )                # Экранирующий символ (например, *)
id )                       ## Преобразовать имя рецепта в ID
index )                    ## Создать 'index.txt' в каталоге AsciiDoc
rerun )                     ## Запуск примеров для повторного создания
                           ## (существующих!) выходных файлов
cleanup )                  ## Очистка мусора XHTML/XML/PDF
```

```
$ examples/ch03/wrapper.sh help heading
h1 )                      # Заголовок уровня 1 (в AsciiDoc h3)
h2 )                      # Заголовок уровня 2 (в AsciiDoc h4)
h3 )                      # Заголовок уровня 3 (в AsciiDoc h5)
```

## Еще один важный момент

Каждый вариант в операторе `case` мы завершаем двойной точкой с запятой. В первом примере в начале этой главы мы написали:

```
"yes") echo "glad you agreed" ;;
```

Символы `;;` означают, что никаких дальнейших действий предпринимать не нужно. Встретив эту пару символов, `bash` продолжит выполнение сценария с первой инструкции после ключевого слова `esac`.

Но такое поведение нежелательно, если требуется продолжить проверку других вариантов в операторе `case` или выполнить иные действия. Синтаксис `bash` позволяет это сделать с помощью комбинаций символов `;;&` и `&;`.

Вот пример такого кода для получения подробной информации о пути в `$filename`:

```
case $filename in
    ./) echo -n "local"          # Начинается с ./
        ;&                      # Спуститься к следующему варианту
    [^/]*) echo -n "relative"   # Начинается с любого символа, кроме слеша
        ;;&                      # Проверить совпадения с другими вариантами
    /*) echo -n "absolute"      # Начинается с символа слеша
        ;&                      # Спуститься к следующему варианту
    /*/) echo "pathname"        # В имени есть слеш
        ;;
    *) echo "filename"         # Все остальные случаи
        ;;
esac
```

Шаблоны в приведенных выше вариантах будут сравниваться по порядку со значением в \$filename. Первый шаблон состоит из двух буквенных символов — точки и слеша, за которыми следуют любые символы. Если обнаружится совпадение с этим шаблоном (например, если в \$filename передано значение ./this/file), то сценарий выведет «local», но без перевода строки в конце. Следующая строка в сценарии (;&) предписывает спуститься дальше и выполнить команды, перечисленные в следующем варианте (без проверки совпадения с шаблоном). В результате сценарий выведет слово «relative». В отличие от предыдущего шаблона, этот раздел кода заканчивается символами ;;&, требующими проверить совпадение с другими шаблонами (по порядку).

Поэтому далее будет проверено наличие косой черты в начале \$filename. Если значение не совпадет с этим шаблоном, то будет проверен следующий шаблон — символ косой черты в любом месте в строке (любое количество любых символов, затем косая черта, затем опять любое количество любых символов). Если совпадение с этим шаблоном обнаружится (а в нашем примере это так), сценарий выведет слово «pathname». Символы ;; в конце указывают, что в проверке последующих шаблонов нет необходимости, — и bash завершит выполнение оператора case.

## В заключение: стиль и удобочитаемость

В этой главе мы описали оператор case, поддерживающий возможность множественного ветвления. Возможность сопоставления с образцом делает его очень полезным для разработки сценариев, хотя чаще он используется для определения простого совпадения определенных слов.

Последовательности символов ;;, ;;& и ;& добавляют дополнительные возможности, но их применение может вызывать сложности. Возможно, конструкция if/then/else позволит неопытным пользователям лучше структурировать такую логику.

Символы `;;&` и `;&` имеют настолько тонкие различия, что есть опасность неправильно оценить последствия их использования. Поток управления после сопоставления с шаблоном может быть различным в каждом случае: «спускаться ниже» и выполнять дополнительный код, пытаться отыскать совпадение с другим шаблоном или завершить выполнение. Поэтому мы настоятельно рекомендуем снабжать такие строки сценария подробными комментариями, чтобы избежать путаницы или недопонимания.

## ГЛАВА 4

---

# Язык переменных

[https://t.me/it\\_boooks](https://t.me/it_boooks)

Нередко можно увидеть сообщение об ошибке или инструкцию присваивания с идиомой  `${0##*/}`, которая выглядит как ссылка на `$0`, но в действительности представляет собой нечто большее. Давайте рассмотрим подробнее ссылки на переменные и используемые в них дополнительные символы. Вы узнаете, что за этими несколькими специальными символами кроется целый набор операций со строками с весьма широкими возможностями.

## Ссылка на переменную

В большинстве языков программирования ссылка на значение переменной выглядит очень просто. Как правило, нужно просто указать имя переменной, но в некоторых языках к нему добавляется определенный символ, сообщающий, что требуется получить значение. К таким языкам относится и bash: если присваивание выполняется по имени переменной, `VAR=something`, то получение значения — по имени с префиксом в виде знака доллара, `$VAR`. Зачем нужен знак доллара? Так как bash в основном работает со строками, выражение

```
MSG="Error: FILE not found"
```

даст вам простую строку из четырех слов, тогда как

```
MSG="Error: $FILE not found"
```

заменит ссылку `$FILE` значением этой переменной (которая, как предполагается, содержит имя искомого файла).



### Интерпретация переменных

Если хотите, чтобы ссылки на переменные замещались их значениями, обязательно используйте двойные кавычки. При использовании одинарных кавычек все символы интерпретируются буквально и замена не производится.

Чтобы избежать путаницы с определением места, где заканчивается имя переменной (в рассмотренном выше примере пробелы упрощают задачу), следует использовать полный синтаксис — фигурные скобки вокруг имени переменной,  `${FILE}`.

Применение фигурных скобок служит основой для многих специальных синтаксических конструкций, связанных со ссылками на переменные. Например, можно поставить решетку перед именем переменной  `${#VAR}`, чтобы вернуть не значение, а его длину в символах.

<code> \${VAR}</code>	<code> \${#VAR}</code>
oneword	7
/usr/bin/longpath.txt	21
many words in one string	24
3	1
2356	4
1427685	7

---

Но bash может не только извлекать значение или выводить его длину.

## Дополнительные параметры

При извлечении значения переменной можно задать правила подстановки или правки, влияющие на возвращаемое значение, но не на значение в переменной (за исключением одного случая). Для этого используются специальные последовательности символов внутри фигурных скобок, ограничивающих имя переменной, например  `${VAR##*/}`. Рассмотрим несколько таких последовательностей, о которых следует знать.

### Сокращенный вариант команды `basename`

Чтобы запустить сценарий, можно использовать одно только имя его файла, но при этом требуется, чтобы файл имел разрешение на выполнение и находился в каталоге, включенном в список для поиска файлов в переменной окружения `PATH`. Сценарий можно запустить командой `./scriptname`, если он находится в текущем каталоге. Можно указать полный путь, например `/home smith utilities scriptname`, или относительный, если каталог со сценарием располагается недалеко от текущего рабочего каталога.

Независимо от способа вызова, `$0` будет хранить последовательность символов, использованную для запуска сценария, — относительный или абсолютный путь.

Для идентификации сценария в сообщении о порядке использования часто достаточно базового имени — имени самого файла без пути к нему:

```
echo "usage: ${0##*/} namesfile datafile"
```

Имя файла обычно указывают в сообщении, описывающем правильный синтаксис запуска сценария, а также могут использовать в операции присваивания переменной. Во втором случае переменная, как правило, имеет имя `PROGRAM` или `SCRIPT`, потому что выражение возвращает имя выполняемого сценария.

Давайте рассмотрим подробнее, как можно использовать дополнительные параметры переменной `$0` для получения только базового имени файла без других элементов пути.

## Удаление пути или префикса

Чтобы удалить символы в начале (слева) или конце (справа) строки, нужно добавить к ссылке на переменную символ `#` и шаблон, соответствующий удаляемым символам. Выражение  `${MYVAL#img_}` удалит символы `img_`, если с них начинается значение переменной `MYVAL`. Более сложный шаблон,  `${MYVAL#*_}`, удалит любую последовательность символов до подчёркивания включительно. Если же совпадения с шаблоном не обнаружится, то выражение вернет полное значение без изменений.

Если использовать один символ `#`, будет удалено *кратчайшее* из возможных совпадений. При использовании пары символов `##` будет удалено *самое длинное* совпадение.

Теперь, возможно, вы понимаете, что делает выражение  `${0##*/}`? Из значения в переменной `$0` — пути к файлу сценария — слева будет удалена самая длинная последовательность любых символов, заканчивающаяся косой чертой, т. е. все компоненты пути, и останется только имя самого сценария.

Ниже представлены несколько возможных значений `$0` и результаты применения двух видов шаблона, чтобы показать, какое влияние на результат оказывают требования удаления кратчайшего (`#`) и самого длинного (`##`) совпадений:

Значение \$0	Выражение	Возвращаемый результат
<code>./ascript</code>	<code> \${0#/}/</code>	<code>ascript</code>
<code>./ascript</code>	<code> \${0##*/}</code>	<code>ascript</code>
<code>../bin/ascript</code>	<code> \${0#/}/</code>	<code>bin/ascript</code>
<code>../bin/ascript</code>	<code> \${0##*/}</code>	<code>ascript</code>
<code>/home/guy/bin/ascript</code>	<code> \${0#/}/</code>	<code>home/guy/bin/ascript</code>
<code>/home/guy/bin/ascript</code>	<code> \${0##*/}</code>	<code>ascript</code>

Обратите внимание, что шаблон с кратчайшим совпадением для \*/ может соответствовать, в том числе, только косой черте.



### Шаблоны командной оболочки — не регулярные выражения

Шаблоны, используемые при извлечении значения переменной, *не являются* регулярными выражениями. В шаблонах командной оболочки \* соответствует нулевому или большему количеству символов, ? — одному символу, а [ символы ] — любому из символов внутри фигурных скобок.

## Сокращенный вариант команды `dirname` или удаление суффикса

Подобно тому как # удаляет префикс, то есть символы слева, знак % удаляет суффикс, то есть символы справа. Двойной знак процента удаляет самое длинное совпадение. Рассмотрим несколько примеров удаления суффикса. В первых примерах используется переменная \$FN, содержащая имя файла изображения. Это имя может заканчиваться расширением .jpg, .jpeg, .png или .gif. Сравните, как разные шаблоны удаляют различные части строки. В последних нескольких примерах показано, как получить нечто похожее на результат применения команды `dirname` к параметру \$0:

Значение переменной	Выражение	Возвращаемый результат
img.1231.jpg	\${FN%.*}	img.1231
img.1231.jpg	\${FN%%.*}	img
./ascript	\${0%/*}	.
./ascript	\${0%%/*}	.
/home/guy/bin/ascript	\${0%/*}	/home/guy/bin
/home/guy/bin/ascript	\${0%%/*}	

Это выражение не всегда возвращает тот же результат, что и команда `dirname`. Например, применительно к значению /file наше выражение

вернет пустую строку, а `dirname` — косую черту. При желании этот случай можно отдельно обработать в сценарии, добавив дополнительную логику. Также можно игнорировать его, если предполагается, что он никогда не встретится, или просто добавить косую черту в конец выражения, например  `${0%/*}/`, чтобы результат всегда заканчивался косой чертой.



### Удаление префикса и суффикса

Чтобы проще запомнить, что `#` удаляет совпадение слева, а `%` — справа, посмотрите на клавиатуру: символ `#` (`Shift+3`) находится слева от `%` (`Shift+5`).

## Другие модификаторы

Помимо `#` и `%`, есть еще несколько модификаторов, способных изменить значение, которое извлекается из переменной. С помощью `^` или `^^` можно преобразовать, соответственно, первый или все символы в строке в верхний регистр, а с помощью `,` или `,,` — в нижний регистр, как показано ниже:

Значение переменной TXT	Выражение	Возвращаемый результат
<code>message to send</code>	<code> \${TXT^}</code>	<code>Message to send</code>
<code>message to send</code>	<code> \${TXT^^}</code>	<code>MESSAGE TO SEND</code>
<code>Some Words</code>	<code> \${TXT,}</code>	<code>some Words</code>
<code>Do Not YELL</code>	<code> \${TXT,,}</code>	<code>do not yell</code>

Можно также использовать специальные объявления переменных: `declare -u UPPER` и `declare -l lower`. Содержимое переменных, объявленных таким образом, всегда будет преобразовываться в верхний или нижний регистр соответственно.

Но самым гибким является модификатор `/`. Он выполняет замену в любом месте строки, а не только в начале или конце. Подобно команде `sed`,

этот модификатор позволяет определить искомый шаблон, а с помощью дополнительной косой черты — строку замены. Одиночная косая черта означает одну замену (первое совпадение), а две косые черты — замену всех совпадений. Вот несколько примеров:

Значение переменной FN	Выражение	Возвращаемый результат
FN="my filename with spaces.txt"	\${FN/ /_}	my_filename with spaces.txt
FN="my filename with spaces.txt"	\${FN// /_}	my_filename_with_spaces.txt
FN="my filename with spaces.txt"	\${FN// /}	myfilenamewithspaces.txt
FN="/usr/bin/filename"	\${FN//\// }	usr bin filename
FN="/usr/bin/filename"	\${FN/\// }	usr/bin/filename



### Без завершающего слеша

Обратите внимание, что, в отличие от таких команд, как `sed` или `vi`, в конце выражений в примерах выше не ставится слеш. Строку завершает закрывающая фигурная скобка.

Почему бы не использовать этот гибкий механизм замены всегда? Зачем обременять себя изучением модификаторов # и %, удаляющих символы в начале и конце строки? Давайте рассмотрим такое имя файла: `frank.gifford.gif` и предположим, что нам нужно конвертировать его в формат `jpg` с помощью команды `convert` из пакета Image Magick. Однако замена с использованием / не позволяет привязать поиск к определенной части строки. Поэтому, если попытаться заменить `.gif` в имени файла на `.jpg`, то вы получите `frank.jpgford.gif`. В подобных ситуациях удаление суффикса с помощью % существенно облегчит задачу.

Еще один полезный модификатор извлекает подстроку из переменной. Добавьте двоеточие после имени переменной, укажите смещение вправо первого символа извлекаемой подстроки (отсчет смещений начинается с 0, то есть 0 соответствует первому символу строки), добавьте еще одно двоеточие и укажите длину извлекаемой подстроки. Если

опустить второе двоеточие и длину, то выражение вернет оставшуюся часть строки. Вот несколько примеров:

Значение переменной FN	Выражение	Возвращаемый результат
/home/bin/util.sh	`\${FN:0:1}`	/
/home/bin/util.sh	`\${FN:1:1}`	h
/home/bin/util.sh	`\${FN:3:2}`	me
/home/bin/util.sh	`\${FN:10:4}`	util
/home/bin/util.sh	`\${FN:10}`	util.sh

Пример 4.1 демонстрирует использование дополнительных параметров для анализа входных данных и обработки определенных полей, которые будут использоваться при автоматическом создании правил брандмауэра. Мы также включили в код большую справочную таблицу *дополнительных параметров bash*, так как заботимся об удобочитаемости кода. Результаты выполнения кода показаны в примере 4.2.

#### Пример 4.1. Анализ входных данных с использованием дополнительных параметров: код

```
#!/usr/bin/env bash
# parameter-expansion.sh: применение дополнительных параметров для анализа
# Автор и дата: _bash Idioms_ 2022
# Имя файла в bash Idioms: examples/ch04/parameter-expansion.sh
#
# Не работает в Zsh 5.4.2!

customer_subnet_name='Acme Inc subnet 10.11.12.13/24'

echo ''
echo "Say we have this string: $customer_subnet_name"

customer_name=${customer_subnet_name%subnet*} # Удалить 'subnet' в конце
subnet=${customer_subnet_name##* }           # Удалить начальные пробелы
ipa=${subnet%/*}                           # Удалить '/' в конце
cidr=${subnet#/}                          # Удалить до '/'
fw_object_name=${customer_subnet_name// /_} # Заменить пробелы на '_'
fw_object_name=${fw_object_name///-}        # Заменить '/' на '-'
fw_object_name=${fw_object_name,,}          # В нижний регистр
```

```
echo ''  
echo 'When the code runs we get:'  
echo ''  
echo "Customer name: $customer_name"  
echo "Subnet:           $subnet"  
echo "IPA             $ipa"  
echo "CIDR mask:      $cidr"  
echo "FW Object:       $fw_object_name"  
  
# Дополнительные параметры в bash: https://oreil.ly/Af8lw  
  
# ${var#pattern}          Удалить кратчайшее совпадение с pattern в начале  
# ${var##pattern}         Удалить самое длинное совпадение с pattern в начале  
# ${var%pattern}          Удалить кратчайшее совпадение с pattern в конце  
# ${var%%pattern}         Удалить самое длинное совпадение с pattern в конце  
  
# ${var/pattern/replacement} Заменить первое совпадение с pattern на replacement  
# ${var//pattern/replacement} Заменить все совпадения с pattern на replacement  
  
# ${var^pattern}          Преобразовать первое совпадение с pattern в верхний  
#                           регистр  
# ${var^^pattern}         Преобразовать все совпадения с pattern в верхний  
#                           регистр  
# ${var,,pattern}         Преобразовать первое совпадение с pattern в нижний  
#                           регистр  
# ${var,,pattern}         Преобразовать все совпадения с pattern в нижний  
#                           регистр  
  
# ${var:offset}            Извлечь подстроку, начиная с offset  
# ${var:offset:length}    Извлечь подстроку, начиная с offset, длиной length  
  
# ${var:-default}          Вернуть значение var, если имеется, иначе default  
# ${var:=default}          Присвоить default переменной var, если она еще не  
#                           установлена  
# ${var:?error_message}    Вернуть error_message, если var не установлена  
# ${var:+replaced}         Вернуть replaced, если var установлена  
  
# ${#var}                  Вернуть длину var  
# ${!var[*]}               Вернуть индексы или ключи массива  
# ${!var[@]}               Вернуть индексы или ключи массива (поддерживаются  
#                           кавычки)  
# ${!prefix*}              Вернуть имена переменных, начинающиеся с +prefix+  
# ${!prefix@}              Вернуть имена переменных, начинающиеся с prefix,  
#                           (поддерживаются кавычки)  
# ${var@Q}                 Вернуть значение в кавычках  
# ${var@E}                 Вернуть развернутое значение (лучше, чем `eval`!)  
# ${var@P}                 Вернуть развернутое значение как приглашение к вводу  
# ${var@a}                 Вернуть оператор присваивания или объявления  
#                           переменной  
# ${var@a}                 Вернуть атрибуты
```

**Пример 4.2.** Анализ входных данных с использованием дополнительных параметров: вывод

Say we have this string: Acme Inc subnet 10.11.12.13/24

When the code runs we get:

```
Customer name: Acme Inc
Subnet: 10.11.12.13/24
IPA 10.11.12.13
CIDR mask: 24
FW Object: acme_inc_subnet_10.11.12.13-24
```

## Условные подстановки

Некоторые из подстановок, приведенных в примере 4.1, являются условными, то есть выполняются только при определенных условиях. Того же эффекта можно добиться, используя операторы `if`, но идиомы позволяют сократить код в некоторых распространенных случаях. Особенность условных подстановок — двоеточие, за которым следует другой специальный символ: минус (-), плюс (+) или знак равенства (=). Они проверяют, была ли создана переменная и имеет ли она значение. Во втором случае значением переменной является пустая строка. Несозданной (неустановленной) считается переменная, которой еще не было присвоено значение или которая была явно удалена с помощью команды `unset`. Позиционные параметры (`$1`, `$2` и т. д.) считаются несозданными, если пользователь не передал параметр в соответствующей позиции.

Если в условные подстановки не включить двоеточие, то они выполняются, только если переменная не создана; для созданных переменных возвращаются их фактические значения, даже если это пустая строка.

## Значения по умолчанию

Распространенным случаем использования значений по умолчанию является сценарий с одним необязательным параметром. Если параметр

не указан при вызове сценария, следует использовать значение по умолчанию. Например, в bash можно написать такой код:

```
LEN=${1:-5}
```

Он присвоит переменной LEN значение первого параметра (\$1), если он был указан, или значение 5. Вот пример сценария:

```
LEN="${1:-5}"
cut -d',' -f2-3 /tmp/megaraid.out | sort | uniq -c | sort -rn | head -n "$LEN"
```

Он извлекает второе и третье поля из записей в CSV-файле /tmp/megaraid.out, сортирует, подсчитывает количество вхождений каждой пары значений, а затем выводит первые пять пар из списка. Значение по умолчанию 5 можно переопределить и отобразить, например, первые три, 10 или сколько пожелаете пар, просто указав нужное количество в единственном параметре сценария.

## Списки значений, разделенных запятыми

Другая разновидность условной подстановки с использованием знака + проверяет, присвоено ли переменной какое-то (непустое) значение, и, если присвоено, возвращает заданное значение. Звучит странно: если переменная имеет значение, то зачем возвращать какое-то другое значение?

Однако эта, казалось бы, странная логика имеет полезное применение: создание списка значений, разделенных запятыми. Обычно такой список создается многократным добавлением значений. При этом возникает необходимость в операторе if, чтобы не добавить лишнюю запятую в начале или в конце списка, но этого не требуется при использовании идиомы соединения:

```
for fn in * ; do
    S=${LIST:+,}           # S -- разделитель
    LIST="$LIST${S}${fn}"
done
```

Также ознакомьтесь с примером 7.1.

## Изменение значения

Ни одна из описанных выше подстановок не изменяет значение самой переменной. Однако есть исключение. Выражение  `${VAR:=value}`, действует так же, как предыдущая идиома значения по умолчанию, но с одним важным исключением. Если переменная `VAR` имеет пустое значение или не создана, то ей будет присвоено значение `value` (на что намекает знак равенства) и оно же будет возвращено (если `VAR` уже имеет некоторое непустое значение, то выражение возвратит его). Но такой способ присваивания значения *не* работает с позиционными параметрами, такими как `$1`, поэтому он используется довольно редко.

## \$RANDOM

В bash есть очень удобная переменная `$RANDOM`. В справочном руководстве по bash<sup>1</sup> указано:

При каждом обращении к этой переменной генерируется случайное целое число от 0 до 32 767. Присвоение значения этой переменной запускает генератор случайных чисел.

Эта переменная не годится для криптографических функций, но вполне подойдет для моделирования игрового кубика или добавления шума в слишком предсказуемые операции. Мы используем ее в разделе «Простой пример подсчета слов» в главе 7.

Пример 4.3 показывает, как с помощью `$RANDOM` выбрать случайный элемент из списка.

### Пример 4.3. Выбор случайного элемента из списка

```
declare -a mylist
mylist=(foo bar baz one two "three four")

range=${#mylist[@]}
random=$(( $RANDOM % $range )) # от 0 до числа длины списка
```

---

<sup>1</sup> <https://oreil.ly/aQSXr>.

```
echo "range = $range, random = $random, choice = ${mylist[$random]}"  
# Более короткий способ, но его будет трудно понять через полгода после написания:  
# echo "choice = ${mylist[$(( $RANDOM % ${#mylist[@]} ))]}"
```

Иногда можно увидеть и такое применение \$RANDOM:

```
TEMP_DIR="$TMP/myscript.$RANDOM"  
[ -d "$TEMP_DIR" ] || mkdir "$TEMP_DIR"
```

Однако это решение чревато состоянием гонки и, очевидно, является шаблоном. Кроме того, иногда важно иметь представление о том, что захламляет \$TMP. Не забудьте поставить ловушку trap (см. раздел «Это ловушка!» в главе 9), чтобы прибрать за собой. Мы рекомендуем подумать об использовании mktemp, хотя обсуждение этой проблемы выходит за рамки идиом bash.



### \$RANDOM и dash

Переменная \$RANDOM недоступна в *dash* – интерпретаторе командной оболочки, на который указывает ссылка */bin/sh* в некоторых дистрибутивах Linux. Актуальные версии Debian и Ubuntu используют *dash*, так как он меньше по объему и быстрее, чем bash, что помогает им быстрее загружаться. Но некоторые возможности, доступные в bash, в них работать не будут. Однако в Zsh эта переменная имеется.

## Подстановка команд

Мы уже использовали *подстановку команд* в главе 2, но не говорили об этом. Старый способ такой подстановки в оболочке Bourne заключался в использовании обратных кавычек (обратных апострофов) ``. Мы рекомендуем использовать современный более читаемый и POSIX-совместимый синтаксис \$(). Вы можете встретить обе формы, потому что именно так вывод команд переносится в значения переменных:

```
unique_lines_in_file="$(sort -u "$my_file" | wc -l)"
```

Две следующих строки делают то же самое, но вторая использует внутренние механизмы командной оболочки и потому работает быстрее:

```
for arg in $(cat /some/file)
for arg in $(< /some/file) # Быстрее, чем с вызовом команды cat
```



### Подстановка команд

Подстановка команд имеет решающее значение в сфере DevOps, потому что позволяет собирать и использовать данные, существующие только во время выполнения кода. Например:

```
instance_id=$(aws ec2 run-instances --image $base_ami_id ... \
--output text --query 'Instances[*].InstanceId')

state=$(aws ec2 describe-instances --instance-ids $instance_id \
--output text --query 'Reservations[*].Instances[*].State.Name')
```



### Вложенная подстановка команд

Использование `` при вложенной подстановке команд выглядит неряшливо и чревато ошибками из-за сложного синтаксиса. Значительно проще использовать \$(), как показано ниже:

```
### Просто работает
$ echo $(echo $(echo $(echo inside)))
inside

### Ошибка
$ echo `echo `echo `echo inside```
echo inside

### Работает, но выглядит ужасно
$ echo `echo \`echo \\`echo inside\\``\``
```

Спасибо нашему рецензенту Яну Миеллу (Ian Miell), предоставившему код для примера.

## В заключение: стиль и удобочитаемость

Ссылаясь на переменную в bash, можно изменять извлекаемое или присваиваемое значение. Несколько специальных символов в конце ссылки на переменную могут удалять символы в начале или в конце строкового значения, менять их регистр, замещать символы или возвращать подстроку. Востребованность и удобство этих возможностей породили идиомы для значений по умолчанию, замены команд `basename` и `dirname`, а также создание списков значений, разделенных запятыми, без явного использования оператора `if`.

Подстановка переменных — замечательная особенность bash, и мы советуем использовать ее. Однако мы также настоятельно рекомендуем снабжать код комментариями, чтобы было ясно, какого рода подстановку вы выполняете. Тот, кому впоследствии придется разбирать ваш код, будет вам за это благодарен.

## ГЛАВА 5

---

# Выражения и арифметика

[https://t.me/it\\_boooks](https://t.me/it_boooks)

Командная оболочка bash, с одной стороны, позволяет сделать многие операции несколькими способами, с другой стороны, почти одинаковый синтаксис может приводить к совершенно разным действиям. Часто разница заключается лишь в нескольких специальных символах. Мы уже видели выражения  `${VAR}` и  `${#VAR}`, первое из которых возвращает значение переменной, второе — длину этого значения (см. раздел «Ссылка на переменную» в главе 4). Или  `${VAR[@]}` и  `${VAR[*]}`, отличающиеся поддержкой кавычек (см. раздел «Кавычки и пробелы» в главе 2).

Многие идиомы bash вызывают вопросы: следует ли использовать двойные или одинарные квадратные скобки и не лучше ли вообще отказаться от их применения, или в чем разница между `(( ... ))` и `$(( ... ))`? Обычно разные варианты использования символов имеют что-то общее, намекающее на сходство причин, лежащих в основе синтаксиса. Но в некоторых случаях синтаксис обусловлен исключительно традициями.

Давайте рассмотрим и попробуем объяснить некоторые из идиоматических шаблонов и арифметических выражений bash.



### Только целочисленные вычисления

Командная оболочка bash поддерживает только целочисленную арифметику, так как ее задачи в большинстве случаев связаны с подсчетом чего-либо: итераций, количества файлов, размеров в байтах и т. п. Но как быть, если потребуется выполнить вычисления с плавающей точкой? В конце концов,

современная версия команды `sleep` может принимать дробные значения. Например, `sleep 0.25` приостановит работу на четверть секунды. А если потребуется пауза на периоды, кратные четверти секунды? Вы могли бы написать `sleep $(( 6 * 0.25 ))`, но этот прием не сработает.

Самое простое решение — выполнить вычисления с помощью другой программы, такой как `bc` или `awk`. Ниже приведен сценарий `fp`, который можно поместить в каталог `~/bin` или другой, находящийся в списке в переменной `PATH` (не забудьте дать ему права на выполнение):

```
# /bin/bash -
# fp -- реализует операции с плавающей точкой через awk
# порядок использования: fp "выражение"
awk "BEGIN { print $* }"
```

При таком сценарии команда `sleep $(fp "6 * 0.25")` выполнит желаемые вычисления с плавающей точкой. Конечно, вычисления производят не сам интерпретатор `bash`, но он помогает выполнять расчеты.

## Арифметика

Язык `bash` в основном ориентирован на операции со строками, однако, встретив в сценарии двойные круглые скобки, знайте, что здесь выполняются арифметические вычисления — с целыми числами, а не со строками. По варианту цикла `for` с двойными круглыми скобками вам знакома следующая конструкция:

```
for ((i=0; i<size; i++))
```

Обратите внимание, что здесь не требуется использовать `$` перед именами переменных. Это верно для всех выражений с двойными круглыми скобками. Где еще их можно встретить?

Во-первых, двойные круглые скобки с предшествующим им знаком `$` можно использовать для арифметических вычислений в операции присваивания значения переменной, например:

```
max=$(( intro + body + outro - 1 ))
median_loc=$((len / 2))
```

Снова отметим, что знак \$ не требуется использовать для ссылки на переменные внутри двойных круглых скобок.

Во-вторых, взгляните на следующий пример:

```
if (( max - 3 > x * 4 )) ; then
    # Сделать что-то
fi
```

Почему на этот раз двойные круглые скобки используются без предшествующего им знака \$?

В первом случае в операции присваивания нам нужно получить значение выражения. Поэтому, как и в случае с переменными, мы добавляем \$, подсказывающий интерпретатору, что нам нужно значение. Но в операторе if знак доллара не нужен, так как для принятия решения достаточно логического значения истина/ложь. Если при вычислении выражения внутри двойных скобок (без знака \$) получается ненулевой результат, то возвращается статус 0, что в bash считается «истинным» значением. Иначе возвращается статус 1, что в bash означает «ложь».

Обратили внимание, что мы говорим «возвращается статус»? Это связано с тем, что двойные круглые скобки без знака \$ интерпретируются как выполнение одной или нескольких команд. Они не возвращают результат вычислений, который можно было бы присвоить переменной. Однако в некоторых случаях их можно использовать, чтобы присвоить некоторой переменной новое значение, потому что bash поддерживает некоторые операторы присваивания в стиле языка C. Вот пример трех полных операторов bash:

```
(( step++ ))
(( median_loc = len / 2 ))
(( dist *= 4 ))
```

Все операторы выполняют арифметические вычисления и присваивают результат переменной. Эти выражения возвращают не результат вычислений, а только статус (код возврата), который можно проверить в переменной \$?, устанавливаемой после выполнения каждого оператора.

Можно ли записать операторы из предыдущего примера, используя синтаксис со знаком доллара и двойными круглыми скобками? Да, причем такой вариант выглядит более привычным:

```
step=$(( step + 1 ))
median_loc=$(( len / 2 ))
dist=$(( dist * 4 ))
```

Не следует использовать выражение `$((step++))` как самостоятельную инструкцию в отдельной строке, потому что оно вернет числовое значение, которое интерпретатор воспримет как имя выполняемой команды. Если `step++` даст результат 3, то оболочка попытается отыскать команду с именем 3.



### Не забывайте о пробелах

В операции присваивания значения переменной пробелы вокруг знака равенства не допускаются. Синтаксически вся инструкция присваивания должна быть одним «словом». Однако внутри круглых скобок допускается использовать пробелы, потому что круглые скобки определяют границы этого «слова».

Теперь рассмотрим способ вычисления арифметических выражений, сохранившийся из прошлого. Чтобы получить эффект двойных круглых скобок без знака \$, можно использовать встроенную команду `let`. Взгляните на следующие пары эквивалентных инструкций:

```
(( step++ )) # То же, что и:
let "step++"

(( median_loc = len / 2 )) # То же, что и:
let "median_loc = len / 2"

(( dist *= 4 )) # То же, что и:
let "dist*=4"
```

Но будьте осторожны: если не заключить выражение `let` в кавычки (одинарные или двойные), то пробелы в нем лучше не использовать (первая инструкция `let` в примере не нуждается в кавычках, но всегда полезно их использовать). Пробелы разделят команду на отдельные слова, а `let`

принимает только одно слово. Вы получите синтаксическую ошибку, если выражение будет состоять из нескольких слов.

## Круглые скобки не нужны

Хотя bash в основном ориентирован на операции со строками, возможны исключения. Переменную можно объявить как целочисленную, например: `declare -i MYVAR`. Затем с ней можно выполнять арифметические операции и присваивать ей значение без использования двойных круглых скобок или знака `$`. Эти возможности демонстрирует сценарий `seesaw.sh`:

```
declare -i SEE
X=9
Y=3
SEE=X+Y      # Здесь будет выполнена арифметическая операция
SAW=X+Y      # Интерпретируется как строковый литерал
SUM=$X+$Y    # Интерпретируется как конкатенация строк
echo "SEE = $SEE"
echo "SAW = $SAW"
echo "SUM = $SUM"
```

Результат выполнения этого сценария наглядно демонстрирует ориентацию bash в основном на операции со строками. Значения `SAW` и `SUM` формируются строковыми операциями. Только `SEE` получает числовое значение — результат арифметического действия:

```
$ bash seesaw.sh
SEE = 12
SAW = X+Y
SUM = 9+3
$
```

Как видите, арифметические действия можно выполнять без двойных круглых скобок, но мы обычно избегаем такого стиля, так как он требует объявления переменной как целочисленной. Если забыть добавить оператор `declare`, то оболочка не сообщит об ошибке, а просто присвоит переменной иное значение, чем вы будете ожидать.

## Составные команды

Разумеется, вам хорошо знакома практика записывать одиночные команды в отдельных строках сценария. В условном выражении оператор `if` может оценивать успешность выполнения этой команды и инициировать некоторые действия в зависимости от результата. В главе 1 мы представили идиому оператора `if` «проверка условия без `if`». Теперь давайте взглянем на простой оператор `if` с единственной командой:

```
if cd $DIR ; then # Выполнить что-то...
```

Сравните его со следующими выражениями:

```
if [ $DIR ] ; then # Выполнить что-то...
if [[ $DIR ]] ; then # Выполнить что-то...
```

Почему в двух последних примерах используются квадратные скобки, а в первом нет, и есть ли разница? Почему во втором примере используются одинарные квадратные скобки, а в третьем — двойные?

Без квадратных скобок оболочка выполнит команду (в нашем примере `cd`), которая вернет признак успешного или неуспешного выполнения. Этот признак интерпретируется оператором `if` как истинное или ложное значение и используется для выбора между операторами `then` или `else` (если он есть). Но `bash` позволяет поместить в оператор `if` и целый конвейер команд (например, `cmd | sort | wc`). Статус выполнения последней команды в конвейере определяет значение условного выражения — истинное или ложное (такой подход может вызвать ошибки, которые очень трудно найти; см. описание `set -o pipefail` в разделе «Неофициальный строгий режим в `bash`» в главе 9).

Синтаксис с одинарными квадратными скобками запускает встроенную команду `test`. Открывающая квадратная скобка — этостроенная команда оболочки, аналог команды `test`, но отличающаяся обязательным конечным аргументом `]`. Двойные квадратные скобки технически являются ключевым словом, определяющим составную команду. По своим

свойствам это ключевое слово очень похоже на одинарные квадратные скобки и команду `test`, но имеет и некоторые отличия.

Синтаксис с одинарными или двойными квадратными скобками используется для выполнения некоторой логики и сравнений, то есть условных выражений, проверяющих состояние, например присутствие файла, наличие у него определенных разрешений, или имеет ли переменная непустое значение. Полный список проверок, которые можно выполнить в bash, вы найдете в странице справочного руководства `man bash` в разделе Conditional Expressions<sup>1</sup>, а для быстрой справки используйте команду `help test`.

Наш предыдущий пример проверяет, имеет ли переменная `DIR` непустое значение (ненулевую длину). Ту же проверку можно записать иначе:

```
if [[ -n "$DIR" ]]; then ...
```

Можно проверить и наоборот, имеет ли переменная пустое значение (с нулевой длиной) или вообще не была установлена:

```
if [[ -z "$DIR" ]]; then ...
```

Так чем же отличаются проверки с одинарными и двойными квадратными скобками? Таких отличий несколько, но все они незначительны.

Вероятно, самое важное отличие заключается в том, что синтаксис с двойными квадратными скобками поддерживает дополнительный оператор сравнения `=~`, позволяющий использовать регулярные выражения:

```
if [[ "$FILE_NAME" =~ .*xyz*.jpg ]]; then ...
```



### Регулярные выражения

Рассматриваемый случай — *единственный*, когда в bash могут использоваться регулярные выражения! И помните: не заключайте их в кавычки, иначе сопоставление будет выполняться буквально, а не как принято при их применении.

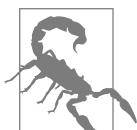
---

<sup>1</sup> <https://oreil.ly/Bn5gv> (перевод страницы на русский язык можно найти по адресу <https://www.opennet.ru/man.shtml?topic=bash&category=1>. — Примеч. пер.).

Еще одно отличие между одинарными и двойными квадратными скобками носит больше стилистический характер, но влияет на переносимость. Следующие две формы делают одно и то же:

```
if [[ $VAR == "literal" ]]; then ...  
if [ $VAR = "literal" ]; then ...
```

Программистам на C и Java использование одного знака равенства для сравнения может показаться ошибкой, но в условных выражениях bash = и == означают одно и то же. Одиночный знак равенства предпочтительнее в синтаксисе с одинарными квадратными скобками, так как соответствует стандарту POSIX (по крайней мере так утверждается на странице справочного руководства `man bash`).



### Тонкое отличие

Внутри двойных квадратных скобок операторы < и > выполняют «лексикографическое сравнение с использованием текущих региональных настроек», тогда как `test` и [ выполняют простое сравнение на основе ASCII.

Также в одинарных квадратных скобках < и > необходимо экранировать (например, `if [ $x \> $y ]`), иначе они будут интерпретироваться как операторы перенаправления. Почему? Дело в том, что одиночная квадратная скобка, как и `test`, является встроенной командой, а не ключевым словом, а ввод/вывод команд можно перенаправлять. Двойные квадратные скобки, напротив, являются ключевым словом, поэтому bash не рассматривает их как перенаправление. По этой причине из двух синтаксических форм мы предпочитаем синтаксис с двойными скобками.

Выражения с одинарными и двойными квадратными скобками позволяют использовать старый синтаксис числовых сравнений, напоминающий Fortran. Например, `-le` — это оператор «меньше или равно» (less-than-or-equal-to). Но здесь кроется еще одно отличие в применении квадратных скобок. Аргументы по обе стороны от этого оператора в одинарных квадратных скобках должны быть простыми целыми числами. В двойных квадратных скобках операнды могут быть целыми арифметическими

выражениями, хотя и без пробелов, если они не заключены в кавычки. Например:

```
if [[ $OTHERVAL*10 -le $VAL/5 ]] ; then ...
```

Однако для сравнения арифметических выражений лучше подходит синтаксис с двойными круглыми скобками. Он позволяет использовать более привычные операторы сравнения в стиле C/Java/Python и дает больше свободы в отношении расстановки пробелов:

```
if (( OTHERVAL * 10 <= VAL / 5 )) ; then ...
```

## В заключение: стиль и удобочитаемость

Какой стиль предпочесть при таком количестве вариантов? Выбирайте тот, который лучше соответствует конкретной ситуации.

Для записи математических выражений мы используем двойные круглые скобки. В bash они обычно указывают на выполнение арифметических действий. Знак \$ сообщает, что требуется получить результат вычисления выражения, иначе будет возвращен статус успешного/неудачного выполнения. Язык bash богат операторами и позволяет выполнять вычисления с использованием синтаксиса с двойными круглыми скобками или с помощью встроенной команды `let`. Так как внутри двойных круглых скобок для получения значений переменных не требуется использовать знак \$, мы стараемся его опускать.

Для вычисления арифметических выражений одни предпочитают использовать двойные круглые скобки, следуя за аналогией с оператором `if`, другие считают, что простая встроенная команда `let` выглядит понятнее. Вы можете выбрать и третий путь: не использовать двойные круглые скобки, объявляя свои переменные целочисленными, но мы этого не рекомендуем. Слишком легко перепутать переменные, объявленные и не объявленные как целочисленные. Использование двойных круглых скобок или команды `let` гарантирует, что выражение будет интерпретироваться как арифметическое.

Для сравнения строковых значений мы используем двойные квадратные скобки, в том числе потому что они поддерживают регулярные выражения.

Для условных выражений более новый синтаксис [[ намного предпочтительнее, чем [. Однако если условное выражение выполняет арифметическое сравнение, то лучше использовать () .

## ГЛАВА 6

---

# ФУНКЦИИ

В двух предыдущих главах были описаны идиомы bash для использования и объединения переменных в выражения. Следующий уровень — группировка выражений и инструкций в функции, которые можно вызывать из разных мест сценария. Язык bash поддерживает функции, но весьма своеобразно. Давайте посмотрим, чем отличаются функции в bash и других языках.

## Вызов функций

Ниже приведены три оператора, вызывающие функции (которые мы придумали для этого примера):

```
Do_Something
Find_File 25 $MPATH $ECODE
Show_All $*
```

Эти инструкции больше похожи на обычные вызовы команд из командной строки. Именно так! других языках вызовы функций записываются примерно так: `Find_File(25, MPATH, ECODE)`. В bash функция вызывается так же, как любая другая команда или сценарий оболочки. Поскольку для функции bash не запускает новый процесс, такой вызов более эффективен, чем вызов внешней команды или сценария.

Кроме того, вызовы функций в bash не возвращают значений, которые можно присвоить переменным. Но подробнее об этом мы расскажем в следующих разделах, а сейчас посмотрим, как определяются функции и их параметры.

## Определение функций

Синтаксис определения функций в bash допускает необязательные элементы. Предположим, например, что требуется объявить функцию с именем `helper`. Вот три варианта объявления:

```
function Helper ()  
function Helper  
Helper ()
```

Все они эквивалентны друг другу. Зарезервированное слово `function` необязательно, но, если оно используется, круглые скобки можно опустить. Мы предпочитаем использовать второй вариант из двух слов: `function` и имя функции. Такое объявление не только четко указывает, что мы делаем, но и напоминает, что, в отличие от других языков, в bash параметры функций не заключаются в круглые скобки. Кроме того, его довольно легко найти с помощью `grep`.

Тело функции следует за определением ее имени и обычно заключается в фигурные скобки:

```
function Say_So {  
    echo 'Here we are in the function named Say_So'  
}
```

## Параметры функций

Как определяются параметры функций в bash? Да никак. Вместо этого, вызывая функцию, можно указать столько параметров, сколько потребуется. Обычно функциям передаются определенные фиксированные

наборы параметров, поэтому при разработке функций с переменным числом аргументов не требуется использовать специальный синтаксис.

Аргументы, независимо от их количества, доступны в функции под именами \$1, \$2, \$3 и т. д., подобно параметрам сценария. Поскольку параметры не имеют имен, а только порядковый номер, всегда желательно добавлять в начало функции комментарий, описывающий, какие параметры она ожидает и в каком порядке. Также довольно часто в первых нескольких строках сценария можно увидеть инструкции, присваивающие значения позиционных параметров переменным с говорящими именами.

Поскольку параметры функции доступны под именами \$1 и т. д., может возникнуть вопрос, что происходит с параметрами сценария, для ссылки на которые используется тот же синтаксис. Вызов функции не изменяет параметры сценария, они просто не видны внутри функции (если, конечно, не передать их в функцию в аргументах).

Однако есть одно исключение. Параметр \$0 содержит имя сценария, в том числе внутри функций. Как тогда получить имя функции? Существует специальная переменная-массив `FUNCNAME`, которая содержит стек вызовов функций. Имя текущей функции всегда находится в элементе 0, поэтому для его получения можно просто обратиться к `$FUNCNAME` (указание имени массива без индекса всегда возвращает первый элемент, то есть элемент с нулевым индексом). Это может пригодиться для вывода отладочных сообщений, см. пример 10.3.

Следующая функция выводит значения двух переданных ей аргументов:

```
function See2it {  
    echo "First arg: $1"  
    echo "Second arg: $2"  
}
```

Если функции передать меньше аргументов, чем она ожидает, то соответствующие параметры будут содержать пустые значения.

Ниже приведена простая функция, отображающая аргументы, с которыми она была вызвана:

```
function Tell_All {  
    echo "You called $FUNCNAME"  
    echo "The $# args supplied are:"  
    for arg in "$@"; do  
        echo "$arg"  
    done  
}  
  
Tell_All 47 tree /tmp
```

Если запустить этот сценарий, он выведет:

```
You called Tell_All  
The 3 args supplied are:  
47  
tree  
/tmp
```

## Возвращаемые значения функций

Функции в bash больше похожи на сценарии, что отличает их от функций в других языках программирования. Возвращаемое значение функций в bash — это просто статус (код завершения). Его значение доступно в переменной `$?` после вызова функции. Так же, как после вызова сценария или выполняемого файла, после вызова функции можно прочитать значение `$?`, чтобы узнать, удачно ли она выполнилась. В действительности в `$?` помещается статус последней команды, выполненной в функции.

Каким образом можно использовать результаты выполнения функции в bash? Есть два основных подхода.

Во-первых, можно просто вывести результат, передав его следующей части сценария. Также можно сохранить этот вывод в переменной, использовав конструкцию `$( )`. Она запустит функцию в подоболочке, и по завершении вы сможете присвоить результат переменной, а затем использовать эту переменную в другой команде.

Во-вторых, можно использовать глобальные переменные, недоступные для внешнего сценария.

## Локальные переменные

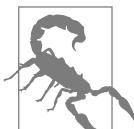
Рассмотрим функцию с циклом `for`, использующим переменную цикла `i`. А что, если эта функция будет вызвана из другого цикла `for`, который тоже использует переменную цикла `i`? Поскольку переменные в bash по своей природе являются глобальными, то внутренняя функция будет влиять на значение индекса внешнего цикла. Что делать в такой ситуации?

В bash есть возможность объявлять переменные локальными, чтобы они не были видны вне функции. Например, переменная `i` в цикле `for` должна объявляться как локальная:

```
function Summer {
    local i
    SUM=0
    for((i=0; i<$1; i++)) {
        let SUM+=i;
    }
}
```

Проблема решена. Наверное, лучше использовать объявление `local -i i` или `declare -i i`, чтобы объявить `i` целочисленной переменной и избежать лишних преобразований из строки в число и обратно. Объявление `declare` внутри функции действует так же, как `local`, скрывая одноименные глобальные переменные. Ключ `-i` указывает целочисленный тип переменной.

Переменная `SUM` в этом примере не объявлена локальной, потому что именно через нее результат функции возвращается вызывающей стороне. Должны ли все переменные в функции объявляться локальными, если они не нужны вне ее? Скорее да, но так поступают лишь немногие программисты. Если вы пишете функции, которые будут использоваться в разных сценариях и должны быть переносимыми, то стоит потратить немного сил и объявить переменные в этих функциях локальными, чтобы избежать неожиданностей.



### Динамическая область видимости

Если объявить локальную переменную в функции, а затем из нее вызвать другую функцию, то вторая функция увидит локальную переменную первой функции, а не глобальную переменную с тем же именем. Если вызвать эту вторую функцию из основного сценария, то она увидит (и будет использовать) глобальную переменную. Добро пожаловать в опасный мир bash с его динамической областью видимости! Это еще одна причина, почему сценарии должны быть простыми и хорошо документированными.

## Особые случаи

Запомните, что определение функции должно предшествовать ее вызову.

Если вы (в отличие от нас) используете круглые скобки в объявлении функции, то можете определить ее тело, применяя любой из видов синтаксиса составных операторов. Например, допустимы двойные круглые скобки, если функция выполняет арифметические вычисления, или двойные квадратные скобки, если функция оценивает условное выражение. Но это довольно редкие случаи.

В определение функции можно поместить оператор перенаправления ввода/вывода. Он будет выполняться в момент вызова функции. Вот пример перенаправления вывода функции в `STDERR` (через `1>&2`):

```
function Usage_Message {  
    echo "usage: $0 value pathname"  
    echo "where value must be positive"  
    echo "and pathname must be an existing file"  
    echo "for example: $0 25 /tmp/scratch.csv"  
} 1>&2
```

Сценарий может вызвать эту функцию, обнаружив, что получил недостаточное количество аргументов или файл с указанным именем не существует (подробнее об этом мы поговорим в разделе «HELP!» в главе 8). Важно отметить, что выходные данные перенаправляются

в `STDERR`, но перенаправление не обязательно помещать в каждую строку в функции — достаточно одного указания за закрывающей фигурной скобкой. Это не только сэкономит время, но и избавляет от необходимости помнить о добавлении переадресации к новым строкам при изменении сценария.

Обратите внимание, что для простоты и ясности мы нарушили некоторые из наших рекомендаций по стилю оформления функций; подробности см. в разделе «Функции» в главе 11.

## Функция `printf`

Знакомым с функцией `printf` в таких языках, как C и Java, материал этого раздела будет освоить несколько проще. Мы решили рассмотреть `printf` в этой главе, хотя она является встроенной функцией.

*Функция форматированного вывода `printf`* (`print formatted`) редко используется в сценариях bash, потому что обычно ей предпочитают команду `echo`. Однако встроенная функция имеет несколько идиоматических расширений, которые вы должны знать и уметь применять. Кроме того, она определена в стандарте POSIX, в отличие от встроенной в bash версии `echo`, и потому лучше переносима и более пригодна для сценариев, которые должны работать в системах, отличных от Linux.

Мы рассмотрим только встроенную версию `printf`, хотя у многих из вас наверняка имеется в системе внешний двоичный файл, который не связан с bash и использует другие параметры:

```
$ type -a printf
printf is a shell builtin
printf is /usr/bin/printf
```

Встроенная функция `printf` будет использоваться всегда, если не указать полный путь (`/usr/bin/printf`) или префикс `env`. Сравните `printf --help` и `env printf --help`, чтобы увидеть различия.

Мы не будем вдаваться в длинный список стандартных спецификаторов формата `printf`; они описаны во многих других источниках.

Получить дополнительную информацию о версии `printf` можно с помощью следующих команд:

- `help printf` или `printf --help`;
- `man 1 printf`;
- если имеется двоичный файл: `/usr/bin/printf --help` или `env printf --help`.

## Вывод POSIX

Если ваш сценарий должен работать в операционных системах, отличных от Linux, то предпочтительно использовать `printf` вместо `echo`. Это просто, но, в отличие от `echo`, функция `printf` не добавляет по умолчанию перевод строки:

```
printf '%s\n' # Перевод строки НЕ добавляется по умолчанию
printf '%b\n' # С переводом строки. Расширьте свои знания управляющих
             # последовательностей
```

Спецификаторы формата `printf` можно заключить как в одинарные, так и в двойные кавычки, следуя обычным правилам интерполяции переменных (в одинарных кавычках переменные интерполироваться не будут). Однако управляющие последовательности, такие как `\n`, будут интерпретироваться и в одинарных, и в двойных кавычках. Как уже отмечалось, мы предпочитаем одинарные кавычки, исключая интерполяцию. Хотя иногда она требуется, помогая понять, что мы делаем что-то неправильно.

Дополнительную информацию по этой теме можно найти по ссылкам:

- <https://unix.stackexchange.com/a/65819>;
- <https://www.in-ulm.de/~mascheck/various/echo+printf>.

## Получение и использование даты и времени

В bash 4.2 была добавлена поддержка спецификатора формата `printf %(%format_даты)T`, но по умолчанию выводилась дата начала эпохи Unix (1970-01-01 00:00:00 -0000). В bash 4.3 значение по умолчанию изменилось на более полезное — текущие дата и время. Спецификатор имеет два специальных аргумента: "-1", означающий «текущие дата и время», и "-2", означающий «дата и время вызова оболочки». Также вы можете задать время в секундах от начала эпохи с последующим преобразованием его в удобочитаемое представление. Но если вы имеете в виду «сейчас», то мы рекомендуем использовать аргумент "-1" для согласованности и ясности намерений:

```
### Установить переменную $today с помощью -v
$ printf -v today '%(%F)T' '-1'
$ echo $today
2021-08-13

### Простое журналирование (обе инструкции выводят одинаковое время)
$ printf '%(%Y-%m-%d %H:%M:%S %z)T: %s\n' '-1' 'Your log message here'
$ printf '%(%F %T %z)T: %s\n' '-1' 'Your log message here'
2021-08-13 12:48:33 -0400: Your log message here

### Какой дате и времени соответствует число секунд 1628873101?
$ printf '%(%F %T)T = %s\n' '1628873101' 'Epoch 1628873101 to human readable'
2021-08-13 12:45:01 = Epoch 1628873101 to human readable
```



### Форматированный вывод в переменную с помощью printf

В справке по `printf` вы могли обратить внимание на параметр `-v var`, позволяющий сохранить вывод в переменной вместо вывода на экран, подобно функции `sprintf` в C. В свое время мы рассмотрим такой способ применения `printf`.

Дополнительные сведения о выводе даты и времени см. в разделе «Журналирование в bash» в главе 10.

Мы предпочитаем использовать встроенную версию `printf`, но GNU-версия<sup>1</sup> `date` работает лучше, чем `printf %(%format_даты)T`, позволяя

<sup>1</sup> GNU (<https://oreil.ly/eG6nV>) — это рекурсивная аббревиатура, расшифровывающаяся как «GNU's Not Unix» (GNU — не Unix).

выполнять арифметические действия с датами, например `date -d '2 months ago' '+%B'` сообщает название позапрошлого месяца:

```
$ date -d '2 months ago' '+%B'  
August
```



### Устаревший bash в Mac

Обратите внимание, что для `printf %(%(формат_даты))T` требуется версия bash не ниже 4.2, а лучше — 4.3 или выше. В старой версии bash 3.2, которая устанавливается по умолчанию в Mac, этот спецификатор работать не будет. См. раздел «bash на Mac» во вступлении.

## printf для повторного использования или отладки

В справке bash говорится: «%q экранирует символы в аргументе, так чтобы его можно было повторно использовать в качестве ввода», — и мы используем эту особенность в главе 7, чтобы показать экранирование строк, но для нетерпеливых приведем простой пример:

```
$ printf '%q' "This example is from $today\n"  
This\ example\ is\ from\ 2021-08-13\n
```

Эта особенность может пригодиться для повторного использования вывода в другом месте, создания форматированного вывода (см. также «Списки значений, разделенных запятыми» в главе 4) и отладки, когда требуется видеть скрытые управляемые символы и поля (см. также раздел «Отладка в bash» в главе 10).

## В заключение: стиль и удобочитаемость

Функции в bash очень похожи на внутренние сценарии. Их вызов напоминает вызов команды, а их аргументы доступны, подобно параметрам сценария (как `$1`, `$2` и т. д.). Функции следует помещать в начало сценария, чтобы их определения располагались выше вызовов. Как и в любом

другом языке, функции должны быть короткими и ориентированными на решение одной задачи. Использование одного перенаправления ввода/вывода для всей функции поможет избавиться от необходимости перенаправлять вывод в каждой инструкции в теле функции.

Самая большая опасность, заключающаяся в функциях, — ссылки на переменные. Функции могут возвращать значения через переменные или путем вывода в `stdout`. В последнем случае вызывающий код сможет перенаправить вывод функции следующей команде. Для возврата значений из функций можно использовать глобальные переменные, но тогда есть риск изменить переменные, которые не должны изменяться. В таком случае может помочь использование объявления `local`, но не забудьте про динамическую область видимости — локальные переменные могут быть доступны другим функциям, вызываемым из основной. Обязательно описывайте такие вещи в комментариях.

Встроенная функция `printf` во многом похожа на одноименную функцию в других языках. Помимо стандартных спецификаторов формата, `printf` в `bash` имеет несколько полезных идиоматических расширений. Нам особенно нравится отсутствие необходимости создавать подоболочку для запуска команды `date`. Это удобно, если требуется только зафиксировать время, например при журналировании событий.

## ГЛАВА 7

---

# Списки и хеши

Компьютеры хорошо вычисляют и систематизируют данные. Мы можем использовать сценарии для вычислений и организации *структур данных*, а строительными блоками для них могут служить *массивы*. Массивы с самого начала поддерживались в bash, причем в версии 4.0 появилась поддержка *ассоциативных массивов*. Код, связанный с массивами, нередко трудно читать, отчасти потому что bash имеет богатую историю и важно сохранять обратную совместимость версий, но также и по вине некоторых разработчиков, склонных все усложнять. На самом деле в массивах нет ничего сложного, и в bash с ними вполне можно работать.

Напомним, что в информатике и программировании массивы — это переменные, содержащие несколько элементов, на которые можно ссылаться с использованием целочисленных индексов. Иначе говоря, массив — это переменная, содержащая список, а не скаляр или одиночное значение. Ассоциативный массив — это список, элементы которого индексируются строками, а не целыми числами. То есть это список пар «ключ – значение», образующий словарь или таблицу поиска, в котором ключи хешируются для формирования области памяти.

В документации bash используются термины *массив* и *ассоциативный массив*, но вы можете называть их *списками* и *хешами* или даже *словарями*, если вам так удобнее. Также в документации bash используется понятие *нижний индекс*, под которым можно понимать просто *индекс*. Обычно мы стараемся следовать документации bash для

согласованности, но в данном случае будем использовать более распространенные и понятные термины: *список*, *хеш* и *индекс*.

Хотя списки (массивы) появились раньше, хеши (ассоциативные массивы) немного проще в обращении. В хешах никогда не возникает вопроса о том, указывать ли индексы (нижние индексы) при ссылке на элементы, потому такое указание является обязательным. Целочисленные индексы списка могут только подразумеваться, причем некоторые операции над ними не имеют смысла для хешей.

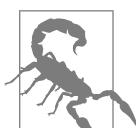
В руководстве указано, что «bash поддерживает переменные, способные хранить одномерные массивы с целочисленными индексами и ассоциативные массивы переменных». В действительности в bash можно создавать многомерные структуры, но они будут выглядеть уродливо, и, скорее всего, эта затея закончится плачевно. Если вам действительно понадобятся такие структуры, лучше реализуйте свою задумку на другом языке.



### Не все версии bash поддерживают хеши

Разбирая материалы этой главы, обратите внимание на вашу версию bash. Как отмечено выше, поддержка хешей (ассоциативных массивов) появилась только в версии 4.0, и потребовалось еще несколько релизов, чтобы отшлифовать некоторые детали. Например, только в версии 4.3 появилась возможность использовать `$list[-1]` для ссылки на последний элемент массива вместо жутковатой конструкции `$mylist[${#mylist[*]}-1]` (где `${#mylist[*]}` – количество элементов).

Как мы уже говорили в разделе «bash на Mac» во вступлении, не забывайте, что по умолчанию в Mac устанавливается очень старая версия bash. Новые версии можно найти на MacPorts, Homebrew или Fink.



### Не POSIX

Следует особо отметить, что массивы (как списки, так и хеши) не стандартизированы в POSIX. Поэтому, если вас беспокоит переносимость кода за пределы bash, проявляйте особую осторожность при их использовании. Например, синтаксис Zsh немного отличается, поэтому представленные далее примеры не будут работать на Mac с этой командной оболочкой.

## Сходные черты

Списки и хеши в bash очень похожи, поэтому мы начнем с обзора общих черт, а затем перейдем к различиям. На самом деле списки можно рассматривать как разновидность хешей, которые просто имеют упорядоченные целочисленные индексы. Конечно, использовать такую интерпретацию *необязательно*, но вполне возможно.

Списки по своей природе — упорядоченные коллекции, тогда как хеши — нет, а такие операции, как *сдвиг* (*shift*) или *добавление в конец* (*push*), имеют смысл только для упорядоченных наборов элементов. С другой стороны, вам никогда не понадобится сортировать ключи в списке, но эта операция имеет определенный смысл для хешей.



### Случайное присваивание

Присваивание без указания *нижнего индекса* изменит нулевой элемент, поэтому `myarray=foo` приведет к созданию или изменению `$myarray[0]`, даже если это хеш!

В документации bash указано следующее:

Если используется индекс @ или \*, ссылка на массив возвращает все его элементы. Эти индексы различаются, только когда ссылка заключается в двойные кавычки. Если "\${name[\*]}" разворачивается в одну строку, включающую значения всех элементов массива, разделенные первым символом из переменной `IFS` (см. раздел «*Приложение \$IFS ради забавы и практической выгоды при чтении файлов*» в главе 9), то "\${name[@]}" разворачивается в коллекцию строк, содержащих отдельные элементы массива.

Неочевидные правила, не так ли? Мы уже говорили в разделе «*Кавычки и пробелы*» в главе 2, что ошибки в этих нюансах могут привести к серьезным неприятностям. В примере 7.1 мы используем `printf "%q"` с вертикальной чертой (|), чтобы разделить части строк (отдельные «слова») при выводе результатов выполнения кода. Правила экранирования — те же, что были описаны в главе 2, только теперь они применяются в контексте списка или хеша.

## Списки

Как мы уже говорили, массивы, также известные как *списки*, — это переменные, содержащие несколько элементов, которые индексируются целыми числами.

В bash индексация начинается с нуля, а массивы могут объявляться как с помощью команд `declare -a`, `local -a`, `readonly -a`, так и простым присваиванием, например: `mylist[0]=foo` или `mylist=()` (пустой список). После объявления переменной списком простое присваивание, такое как `mylist+=(bar)`, будет действовать как операция добавления элемента в конец списка. Но обращайте внимание на знаки `+` и `()`, оба компонента играют важную роль. В табл. 7.1 представлен пример списка.

**Таблица 7.1.** Пример списка в bash

Элемент	Значение
<code>mylist[0]</code>	<code>foo</code>
<code>mylist[1]</code>	<code>bar</code>
<code>mylist[2]</code>	<code>baz</code>

Типичные операции со списками:

- объявление переменной списком;
- присваивание списку одного или нескольких значений;
- если список используется как стек (представьте очередь в столбовой FIFO — first in, first out, то есть первым пришел, первым вышел):
  - добавление в конец (`push`);
  - извлечение из начала (`pop`);
- отображение (вывод) всех значений для целей отладки или повторного использования;
- ссылка на одно или на все значения (`for` или `for each`);

- ссылка на подмножество (срез) значений;
- удаление одного или нескольких значений;
- удаление всего списка.

Давайте не будем рассуждать обо всех этих операциях, а просто рассмотрим их на примере, чтобы вы могли выбрать нужные идиомы, когда они вам понадобятся (пример 7.1).

### **Пример 7.1.** Примеры операций со списками в bash: код

```
#!/usr/bin/env bash
# lists.sh: примеры операций со списками в bash
# Автор и дата: _bash Idioms_ 2022
# Имя файла в bash Idioms: examples/ch07/lists.sh
#
# Не работает в Zsh 5.4.2!

# Книжные страницы не так широки, как экран компьютера!
FORMAT='fmt --width 70 --split-only'

# Объявление списка ❶
# declare -a mylist # Можно использовать такую конструкцию, или `local -a`,
#                   # или `readonly -a`, или:
mylist[0]='foo'      # Объявляет переменную списком и присваивает значение
# элементу mylist[0]

# Можно также одновременно объявить переменную списком
# и присвоить значения его элементам:
#mylist=(foo bar baz three four "five by five" six)

# Добавление элемента в список, обратите внимание на += и () ❷
###mylist=(bar)           # Изменит значение mylist[0]
mylist+=(bar)            # mylist[1]
mylist+=(baz)            # mylist[2]
mylist+=(three four)     # mylist[3] И mylist[4]
mylist+=("five by five") # mylist[5] Обратите внимание на пробелы и кавычки
mylist+=("six")          # mylist[6]

# Обратите внимание на знак "+": мы предполагаем,
# что нулевому элементу уже присвоено значение foo
#mylist+=(bar baz three four "five by five" six)

# Вывод содержимого списка ❸
echo -e "\nThe element count is: ${#mylist[@]} or ${#mylist[*]}"

echo -e "\nThe length of element [4] is: ${#mylist[4]}"
```



```

 myList=("${mylist[@]:$count}")      # Срез, начинающийся с элемента $count
 declare -p myList | $FORMAT        # Отображается после

 echo -e "\nPop LAST element (dumped before and after):"
 declare -p myList | $FORMAT
 unset -v 'myList[-1]'             # В bash версий 4.3 и выше
 #unset -v "myList[$#myList[*]-1]" # В более старых версиях
 declare -p myList

# Удаление срезов ⑦
echo -e "\nDelete element 2 using unset (dumped before and after):"
declare -p myList
unset -v 'myList[2]'
declare -p myList

# Удаление всего списка ⑧
unset -v myList

```

Необходимые пояснения:

- ① Объявление переменной массивом. Здесь мы называем переменную «массивом», а не «списком», потому что используем ключ `-a` (`array` — массив).
- ② Присваивание списку одного или нескольких значений.
- ③ Вывод всех значений для отладки или повторного использования; см. пример 4.1.
- ④ Две разные функции объединения; см. также раздел «Списки значений, разделенных запятыми» в главе 4.
- ⑤ Обход значений в списке; см. пример 4.1.
- ⑥ Операции со срезами (подмножествами) списка, сдвиг и выталкивание.
- ⑦ Удаление срезов.
- ⑧ Удаление всего списка. Используйте `unset` с осторожностью, потому что эта команда может иметь побочные эффекты. Если в файловой системе имеется файл с именем, совпадающим с именем переменной, то поддержка подстановки имен файлов в командной оболочке может привести к неожиданному для вас удалению данных. Чтобы избежать этого,

лучше заключить переменную в кавычки. Еще безопаснее использовать ключ `-v`, чтобы заставить `unset` рассматривать аргумент как переменную, например `unset -v 'list'`.

В примере 7.2 показан вывод сценария из примера 7.1.

**Пример 7.2.** Примеры операций со списками в bash: вывод

```
The element count is: 7 or 7
```

```
The length of element [4] is: 4
```

Dump or list:

```
declare -a mylist=([0]="foo" [1]="bar" [2]="baz" [3]="three"
[4]="four" [5]="five by five" [6]="six")
${mylist[@]} = foo|bar|baz|three|four|five|by|five|six|
${mylist[*]} = foo|bar|baz|three|four|five|by|five|six|
"${mylist[@]}" = foo|bar|baz|three|four|five\ by\ five|six|
"${mylist[*]}" = foo\ bar\ baz\ three\ four\ five\ by\ five\ six\ # Broken!
```

```
Join ',' ${mylist[@]} = foo,bar,baz,three,four,five by five,six
```

```
String_Join '<>' ${mylist[@]} = foo<>bar<>baz<>three<>four<>five by five<>six
foreach "${!mylist[@]}":
```

```
Element: 0; value: foo
Element: 1; value: bar
Element: 2; value: baz
Element: 3; value: three
Element: 4; value: four
Element: 5; value: five by five
Element: 6; value: six
```

But don't do this: \${mylist[\*]}

```
Element: foo; value: foo
Element: bar; value: foo
Element: baz; value: foo
Element: three; value: foo
Element: four; value: foo
Element: five; value: foo
Element: by; value: foo
Element: five; value: foo
Element: six; value: foo
```

```
Start from element 5 and show a slice of 2 elements:
five\ by\ five|six|
```

Shift FIRST element [0] (dumped before and after):

```
declare -a mylist=([0]="foo" [1]="bar" [2]="baz" [3]="three"
```

```
[4]="four" [5]="five by five" [6]="six")
declare -a mylist=([0]="bar" [1]="baz" [2]="three" [3]="four"
[4]="five by five" [5]="six")

Pop LAST element (dumped before and after):
declare -a mylist=([0]="bar" [1]="baz" [2]="three" [3]="four"
[4]="five by five" [5]="six")
declare -a mylist=([0]="bar" [1]="baz" [2]="three" [3]="four" [4]="five by five")

Delete element 2 using unset (dumped before and after):
declare -a mylist=([0]="bar" [1]="baz" [2]="three" [3]="four" [4]="five by five")
declare -a mylist=([0]="bar" [1]="baz" [3]="four" [4]="five by five")
```

## Хеши

Ассоциативные массивы, также известные как хеши или словари, — это списки, индексами в которых являются строки, а не целые числа. Кроме прочего, такие массивы очень удобны для подсчета или «уникализации» (то есть игнорирования или удаления дубликатов) строк.

В отличие от списков, хеши *обязательно* должны объявляться с помощью команд `declare -A`, `local -A` или `readonly -A`, а при обращении всегда необходимо указывать индекс. Пример хеша приведен в табл. 7.2.

**Таблица 7.2.** Пример хеша в bash

Элемент	Значение
<code>myhash[oof]</code>	<code>foo</code>
<code>myhash[rab]</code>	<code>bar</code>
<code>myhash[zab]</code>	<code>baz</code>

Типичные операции с хешами или словарями:

- объявление переменной ассоциативным массивом (здесь мы называем переменную «массивом», потому что используем ключ `-A`);
- присваивание переменной одного или нескольких значений;

- вывод всех значений для отладки или повторного использования;
- ссылка на одно или на все значения (`for` или `for each`);
- ссылка на конкретное значение (поиск);
- удаление одного или нескольких значений;
- удаление всего хеша.

И снова не будем рассуждать обо всех этих операциях, а просто покажем пример их использования, чтобы вы могли выбрать нужные идиомы, когда они вам понадобятся (пример 7.3).

### Пример 7.3. Примеры операций с хешами в bash: код

```
#!/usr/bin/env bash
# hashes.sh: примеры операций с хешами в bash
# Автор и дата: _bash Idioms_ 2022
# Имя файла в bash Idioms: examples/ch07/hashes.sh
#
# Не работает в Zsh 5.4.2!

# Книжные страницы не так широки, как экран компьютера!
FORMAT='fmt --width 70 --split-only'

# Объявление хеша ❶
declare -A myhash # Хеш ОБЯЗАТЕЛЬНО должен объявляться с помощью
# этой команды, `local -A` или `readonly -A`

# Присваивание значения, обратите внимание на "+" ❷
###myhash=(bar)           # Ошибка: чтобы присвоить значение элементу
# ассоциативного массива,
# необходимо использовать нижний индекс
myhash[a]='foo'          # Добавление первого (не нулевого) элемента
myhash[b]='bar'           # Добавление второго элемента
myhash[c]='baz'           # Добавление третьего элемента
myhash[d]='three'         # Четвертый элемент, отличающийся от значения четвертого
# элемента в примере со списками
myhash[e]='four'          # Добавление пятого элемента
myhash[f]='five by five'  # Шестой элемент. Обратите внимание на пробелы
myhash[g]='six'            # Добавление седьмого элемента
# ИЛИ
#myhash=([a]=foo [b]=bar [c]=baz [d]="three" [e]="four" [f]="five by five"
#[g]="six")

# Вывод некоторых деталей и содержимого ❸
echo -e "\nThe key count is: ${#myhash[@]} or ${#myhash[*]}"
```

```

echo -e "\nThe length of the value of key [e] is: ${#myhash[e]}"

echo -e "\nDump or list:"
declare -p myhash | $FORMAT
echo -n   "\${myhash[@]} = " ; printf "%q| " ${myhash[@]}
echo -en  "\n\$${myhash[*]} = " ; printf "%q| " ${myhash[*]}
echo -en  "\n\"${myhash[@]}\" = " ; printf "%q| \"${myhash[@]}\""
echo -en  "\n\"${myhash[*]}\" = " ; printf "%q| \"${myhash[*]}\""
echo -e " # Broken!" # Предыдущая строка -- ошибочная,
                     # она не выводит символ перевода строки
# См. `help printf` или раздел "printf для повторного использования или отладки"
# в главе 6. Мы использовали этот код, чтобы показать отдельные слова:
# %q экранирует аргумент так, что его можно повторно использовать
# в качестве входных данных в командной оболочке

# "Объединение" значений ④
function Join { local IFS="$1"; shift; echo "$*"; } # Односимвольный разделитель!
# Обратите внимание, что Join выше использует "$*", а не "$@"
echo -en "\nJoin ',' \$${myhash[@]} = " ; Join ',' "${myhash[@]}"
function String_Join {
    local delimiter="$1"
    local first_element="$2"
    shift 2
    printf '%s' "$first_element" "${@#/[$delimiter]}"
    # Выводит первый элемент, затем повторно использует формат '%s'
    # для отображения остальных элементов (из аргументов функции $@),
    # но добавляет префикс $delimiter, "замещая" пустой начальный шаблон (/#)
    # значением $delimiter
}
echo -n "String_Join '>' \$${myhash[@]} = " ; String_Join '>' "${myhash[@]}"

# Обход ключей и значений ⑤
echo -e "\nforeach \"\$!myhash[@]\":"
for key in "${!myhash[@]}"; do
    echo -e "\tKey: $key; value: ${myhash[$key]}"
done

echo -e "\nBut don't do this: \$${myhash[*]}"
for key in ${myhash[*]}; do
    echo -e "\tKey: $key; value: ${myhash[$key]}"
done

# Операции со срезами (подмножествами) хеша ⑥
echo -e "\nStart from hash insertion element 5 and show a slice of 2 elements:"
printf "%q| \"${myhash[@]:5:2}"
echo '' # Предыдущая инструкция не выводит символ перевода строки
echo -e "\nStart from hash insertion element 0 (huh?) and show a slice of 3
elements:"
printf "%q| \"${myhash[@]:0:3}"

```

```
echo '' # Предыдущая инструкция не выводит символ перевода строки
echo -e "\nStart from hash insertion element 1 and show a slice of 3 elements:"
printf "%q|" "${myhash[@]:1:3}"
echo '' # Предыдущая инструкция не выводит символ перевода строки

#echo -e "\nShift FIRST key [0]:" = не имеет смысла для хешей!
#echo -e "\nPop LAST key:" = не имеет смысла для хешей!

# Удаление ключей ⑦
echo -e "\nDelete key c using unset (dumped before and after):"
declare -p myhash | $FORMAT
unset -v 'myhash[c]'
declare -p myhash | $FORMAT

# Удаление всего хеша ⑧
unset -v myhash
```

Необходимые пояснения:

- ① Объявление переменной хешем.
- ② Присваивание значений элементам хеша.
- ③ Вывод некоторых деталей и значений; см. пример 4.1.
- ④ Две разных функции объединения; см. также раздел «Списки значений, разделенных запятыми» в главе 4.
- ⑤ Обход ключей и значений в списке; см. также раздел «Списки значений, разделенных запятыми» в главе 4.
- ⑥ Операции со срезами (подмножествами) хеша, которые кажутся довольно странными, потому что индексы не являются порядковыми целыми числами.
- ⑦ Удаление ключей.
- ⑧ Удаление всего хеша. Используйте `unset` с осторожностью, потому что эта команда может иметь побочные эффекты. Если в файловой системе имеется файл с именем, совпадающим с именем переменной, то поддержка подстановки имен файлов в командной оболочке может привести к неожиданному для вас удалению данных. Чтобы избежать этого, лучше заключить переменную в кавычки. Еще безопаснее ис-

пользовать ключ `-v`, чтобы заставить `unset` рассматривать аргумент как переменную, например, `unset -v 'list'`.

В примере 7.4 показан вывод сценария из примера 7.3.

**Пример 7.4.** Примеры операций с хешами в bash: вывод

```
The key count is: 7 or 7
```

```
The length of the value of key [e] is: 4
```

```
Dump or list:
```

```
declare -A myhash=([a]="foo" [b]="bar" [c]="baz" [d]="three"
[e]="four" [f]="five by five" [g]="six" )
${myhash[@]} = foo|bar|baz|three|four|five|by|five|six|
${myhash[*]} = foo|bar|baz|three|four|five|by|five|six|
"${myhash[@]}" = foo|bar|baz|three|four|five\ by\ five|six|
"${myhash[*]}" = foo\ bar\ baz\ three\ four\ five\ by\ five\ six| # Broken!
```

```
Join ',' ${myhash[@]} = foo,bar,baz,three,four,five by five,six
```

```
String_Join '<>' ${myhash[@]} = foo<>bar<>baz<>three<>four<>five by five<>six
foreach "${!myhash[@]}":
```

```
    Key: a; value: foo
    Key: b; value: bar
    Key: c; value: baz
    Key: d; value: three
    Key: e; value: four
    Key: f; value: five by five
    Key: g; value: six
```

```
But don't do this: ${myhash[*]}
```

```
    Key: foo; value:
    Key: bar; value:
    Key: baz; value:
    Key: three; value:
    Key: four; value:
    Key: five; value:
    Key: by; value:
    Key: five; value:
    Key: six; value:
```

```
Start from hash insertion element 5 and show a slice of 2 elements:
four|five\ by\ five|
```

```
Start from hash insertion element 0 (huh?) and show a slice of 3 elements:
foo|bar|baz|
```

```
Start from hash insertion element 1 and show a slice of 3 elements:
foo|bar|baz|
```

```
Delete key c using unset (dumped before and after):
declare -A myhash=([a]="foo" [b]="bar" [c]="baz" [d]="three"
[e]="four" [f]="five by five" [g]="six" )
declare -A myhash=([a]="foo" [b]="bar" [d]="three" [e]="four"
[f]="five by five" [g]="six" )
```

## Пример подсчета слов

Как мы уже говорили, одним из наиболее распространенных применений хешей является подсчет и/или «уникализация» элементов. Продемонстрируем это на задаче по подсчету слов (пример 7.5).

### Пример 7.5. Пример подсчета слов в bash: код

```
#!/usr/bin/env bash
# word-count-example.sh: Дополнительные примеры работы со списками, хешами
#                               и $RANDOM в bash
# Автор и дата: _bash Idioms_ 2022
# Имя файла в bash Idioms: examples/ch07/word-count-example.sh
#
# Не работает в Zsh 5.4.2!

# См. также: `man uniq`
WORD_FILE='/tmp/words.txt'
> $WORD_FILE ❶
trap "rm -f $WORD_FILE" ABRT EXIT HUP INT QUIT TERM

declare -A myhash ❷

echo "Creating & reading random word list in: $WORD_FILE"

# Создание списка слов для использования в примере
mylist=(foo bar baz one two three four)

# Выбор случайных элементов из списка в цикле ❸
range="${#mylist[@]}"
for ((i=0; i<35; i++)); do
    random_element="$(( $RANDOM % $range ))" ❹
    echo "${mylist[$random_element]}" >> $WORD_FILE ❺
done

# Запись слов из списка в хеш
while read line; do ❻
    (( myhash[$line]++ )) ❾
done < $WORD_FILE ❿
```

```
echo -e "\nUnique words from: $WORD_FILE" ❾
for key in "${!myhash[@]}"; do
    echo "$key"
done | sort

echo -e "\nWord counts, ordered by word, from: $WORD_FILE" ❿
for key in "${!myhash[@]}"; do
    printf "%s\t%d\n" $key ${myhash[$key]}
done | sort

echo -e "\nWord counts, ordered by count, from: $WORD_FILE" ⓫
for key in "${!myhash[@]}"; do
    printf "%s\t%d\n" $key ${myhash[$key]}
done | sort -k2,2n
```

Необходимые пояснения:

- ❶ Создание временного файла с установкой ловушки (см. раздел «Это ловушка!» в главе 9) для его удаления.
- ❷ Хеш обязательно должен объявляться с помощью команды `declare -A` как *ассоциативный массив* (мы снова называем хеш «массивом», потому что используется ключ `-A`).
- ❸ Получение количества элементов (диапазона случайных чисел).
- ❹ Выбор случайного элемента списка с помощью переменной `$RANDOM` (пример 4.3).
- ❺ Каждое случайно выбранное слово записывается во временный файл. Вывод случайно выбранных значений выполняется тремя строками кода (❻, ❼ и ❽), но то же самое можно реализовать в одной строке, например:  
`echo "${mylist[$RANDOM% ${#mylist[@]}]}" >> $WORD_FILE`, но понять такой код через шесть месяцев после его написания будет намного сложнее.
- ❻ Чтение только что созданного файла. Обратите внимание, что имя файла указано *после* ключевого слова `done` (см. ❸).
- ❼ Увеличение значения счетчика для уже встречавшегося слова.
- ❽ Обратите внимание, что имя файла указано *после* ключевого слова `done`, завершающего цикл ❻.

❾ Обход ключей для вывода списка слов без повторений и без использования внешней команды `uniq`. Обратите внимание на команду `sort` после ключевого слова `done`.

❿ Повторный обход ключей для отображения значений счетчиков слов.

⓫ Последний обход ключей для отображения счетчиков, но на этот раз с числовой сортировкой по второму полю (`sort -k2,2n`).

В примере 7.6 показан вывод сценария из примера 7.5.

**Пример 7.6.** Пример подсчета слов в bash: вывод

```
Creating & reading random word list in: /tmp/words.txt
```

```
Unique words from: /tmp/words.txt
bar
baz
foo
four
one
three
two
```

```
Word counts, ordered by word, from: /tmp/words.txt
bar 7
baz 6
foo 4
four 3
one 5
three 4
two 6
```

```
Word counts, ordered by count, from: /tmp/words.txt
four 3
foo 4
three 4
one 5
baz 6
two 6
bar 7
```

## В заключение: стиль и удобочитаемость

В этой главе мы познакомились с массивами bash (списками и хешами) и показали идиоматические приемы для распространенных случаев их использования. О списках и хешах в bash можно рассказывать долго, однако дальнейшее обсуждение этой темы выходит за рамки данной книги. За дополнительной информацией мы рекомендуем обратиться к следующим ресурсам:

- [https://www.gnu.org/software/bash/manual/html\\_node/Arrays.html#Arrays](https://www.gnu.org/software/bash/manual/html_node/Arrays.html#Arrays);
- <http://wiki.bash-hackers.org/syntax/arrays>;
- [http://tldp.org/LDP/Bash-Beginners-Guide/html/sect\\_10\\_02.html](http://tldp.org/LDP/Bash-Beginners-Guide/html/sect_10_02.html);
- [https://learning.oreilly.com/library/view/bash-cookbook-2nd/9781491975329](https://learning.oreilly.com/library/view/bash-cookbook-2nd/9781491975329/);
- `man uniq`;
- `man sort`.

После определения правильной структуры данных остальной код пишется практически сам собой. При ошибочном выборе структуры дальнейшая реализация становится проблемной. В bash есть готовые строительные блоки для создания простых структур данных, и, освоив дополнительные знаки препинания, вы с легкостью будете их применять и понимать. Просто помните, что почти во всех случаях следует использовать `[@]`, а не `[*]`, и, если засомневаетесь, возвращайтесь к нашим примерам в этой главе.

## ГЛАВА 8

---

# Аргументы

Некоторые сценарии предназначены для выполнения одной задачи, без каких-либо вариаций. Другие принимают аргументы: имена файлов или ключи, определяющие особенности выполнения. Если ваш сценарий имеет более одного параметра, то вам придется проанализировать аргументы, чтобы гарантировать правильную обработку всех возможных способов передачи аргументов пользователем. И если подумать, даже сценарию, выполняющему единственную задачу, может потребоваться обрабатывать параметр `-h` (или даже `--help`). Давайте посмотрим, как анализировать аргументы, не ухудшая удобочитаемости и простоты сопровождения сценария.

## Ваш первый аргумент

Если сценарий принимает только один параметр, его можно получить, сославшись на переменную `$1`. В сценарии могут быть такие операторы, как `echo $1` или `cat $1`. Но мы не рекомендуем использовать `$1` как универсальное имя, потому что оно ничего не говорит читателю о природе параметра. Для упрощения понимания лучше использовать переменные с говорящими именами. Например, если в параметре передается имя файла, выберите для переменной такое имя, как `in_file` или `pathname`, и присвойте ей значение сразу в начале сценария. Как было показано в главе 4, мы можем задавать значения по умолчанию:

```
filename=${1:-favorite.txt} # Или по умолчанию использовать /dev/null?
```

Если пользователь не задаст параметр при вызове сценария, то значение \$1 не будет установлено. В предыдущем примере, если сценарий будет вызван без параметра, переменная filename получит значение по умолчанию favorite.txt.

Что если сценарию нужны два, три или больше параметров? Как нетрудно догадаться, соответствующие аргументы будут храниться в переменных \$2, \$3 и т. д. И они не будут установлены, если требуемые параметры не были переданы при вызове сценария.

А что произойдет, если для параметра нет хорошего значения по умолчанию? Или, может быть, сценарий должен завершиться, если пользователь передал недостаточное количество аргументов? Число полученных аргументов в сценарии содержится в переменной \$#. Если значение \$# равно 0, значит пользователь запустил сценарий без аргументов. Если значение \$# равно 1, а сценарию нужны два аргумента, то он может вывести сообщение об ошибке и завершиться (см. раздел «Код выхода» в главе 9):

```
if ((#$ != 2)); then
    echo "usage: $0 file1 file2" # $0 -- имя сценария, с которым он был вызван
    exit
fi
```

Приведенный выше код просто проверяет количество полученных аргументов. Но приемы использования этих аргументов в сценарии весьма разнообразны. Некоторые из них были рассмотрены в главе 2.

Ранее для вывода списка всех аргументов было принято использовать переменную \$\*, например в таком варианте: echo \*. Но после появления поддержки пробелов в именах файлов предпочтительным стал другой синтаксис.

Имена файлов с пробелами необходимо заключать в кавычки, например "my file", иначе командная оболочка интерпретирует это имя как

два отдельных слова. Чтобы сослаться на все аргументы сценария и заключить каждый из них в кавычки, можно использовать конструкцию "\$@" (строка) или "\${@}" (список). Ссылка на "\$\*" даст одну большую строку в кавычках, содержащую все аргументы. Например, если вызвать сценарий следующим образом:

```
myscript file1.dat "alt data" "local data" summary.txt
```

ссылка "\$\*" вернет единственное значение "file1.dat alt data local data summary.txt", тогда как "\$@" вернет четыре отдельных слова: file1.dat "alt data" "local data" summary.txt.

Хотя мы уже говорили об этом в разделе «Кавычки и пробелы» в главе 2, а затем в главе 7, некоторые сложные особенности bash целесообразно напомнить еще раз.

## Поддержка ключей

Ключи (options) дают возможность изменить выполнение команды. Классический «идиоматический» способ представления ключей в Unix/Linux — одна буква с предшествующим дефисом, минусом или тире. Например, чтобы сообщить команде, что от нее ожидается развернутый вывод, можно передать ключ -l. А чтобы команда выдавала как можно меньше выходных данных, можно передать ключ -q.

Не все команды и сценарии поддерживают эти ключи, и не все интерпретируют их одинаково. Например, ключ -q одни команды интерпретируют как выбор «тихого» (от quiet) режима вывода, в других он может означать «быстро» (quick), а в третьих быть вообще недопустимым. Все эти особенности основаны, как правило, на традициях.

Традиции стоят того, чтобы их придерживаться, если нет веской причины для отказа. Следование традициям сокращает время обучения, поскольку позволяет использовать знания и опыт, полученные при работе с другими командами или сценариями. Кроме того, можно применять те же методы анализа ключей, что и в других командах и сценариях.

## Анализ ключей

Для анализа ключей сценария командной оболочки используйте встроенную команду `getopts`. Ее можно вызывать многократно (обычно в цикле `while`), пока не будут получены все ключи. Предполагается, что ключи предшествуют другим аргументам. `getopts` распознает ключи, указанные по отдельности (`-a -v`), а также сгруппированные вместе (`-av`). Также можно указать, что ключ должен сопровождаться дополнительным аргументом. Например, можно потребовать, чтобы ключ `-o` сопровождался именем выходного файла и пользователь вызывал сценарий с параметром `-o filename` или `-o filename` — `getopts` поддерживает оба варианта.

В примере 8.1 анализируются короткие ключи.

### Пример 8.1. Анализ аргументов с помощью `getopts`: короткие ключи

```
#!/usr/bin/env bash
# parseit.sh: использование getopts для анализа аргументов
# Автор и дата: _bash Idioms_ 2022
# Имя файла в bash Idioms: examples/ch08/parseit.sh
#_____
```

```
while getopts ':ao:v' VAL ; do ①
    case $VAL in
        a ) AMODE=1 ;;
        o ) OFILE="$OPTARG" ;;
        v ) VERBOSE=1 ;;
        : ) echo "error: no arg supplied to $OPTARG option" ;; ③
        * ) ④
            echo "error: unknown option $OPTARG"
            echo " valid options are: aov"
            ;;
    esac
done
shift $((OPTIND -1)) ⑤
```

Рассмотрим особенности сценария:

① Цикл `while getopts` используется, потому что требуется вызвать `getopts` несколько раз. Эта команда возвращает истинное значение, когда обнаруживает ключ (дефис, за которым следует любая буква, допустимая или нет), и ложное, когда дойдет до конца списка параметров. Встроенная

команда `getopts` требует передать ей два слова: список параметров и имя переменной, в которую она должна поместить очередной найденный ключ. При каждом вызове она будет находить только один параметр, поэтому мы вызываем ее многократно в цикле `while`. Строкой '`:ao:v`' мы сообщаем, что сценарий поддерживает ключи `a`, `o` и `v`. Двоеточие в начале этой строки указывает, что `getopts` не будет сообщать об ошибках при обнаружении неподдерживаемого параметра и оставит его обработку на усмотрение сценария. Двоеточие между `o` и `v` указывает, что ключ `o` должен сопровождаться дополнительным аргументом. `VAL` — имя переменной, куда будет записан очередной найденный ключ.

❷ После вызова `getopts` можно использовать оператор `case`, чтобы определить, какой ключ был найден (подробнее об операторе `case` рассказывается в главе 3).

❸ Поскольку мы потребовали от `getopts` не выводить сообщений об ошибках, мы должны предусмотреть два варианта обработки ошибок. Первый вариант обрабатывает случай, когда ключ `-o` не содержит аргумента. Мы сообщили команде `getopts`, что ожидаем аргумент, добавив двоеточие после `o` в строке '`:ao:v`'. Если, вызывая сценарий, пользователь не передаст аргумент, то переменная `$VAL` получит двоеточие в качестве значения, а `$OPTARG` — символ ключа, для которого не был указан обязательный аргумент (т. е. `o`).

❹ Второй вариант ошибки — пользователь указал не поддерживаемый ключ. В этом случае `$VAL` получит значение '?', а `$OPTARG` — символ нераспознанного ключа. Этот случай обрабатывается с помощью варианта (\*) в операторе `case`.

❺ Для контроля последовательности анализа командной строки в переменной `$OPTIND` сохраняется индекс следующего рассматриваемого аргумента. Когда все аргументы будут проанализированы, `getopts` вернет ложное значение — и цикл `while` завершится. После этого выполняется команда `shift $OPTIND -1`, чтобы исключить из дальнейшего рассмотрения все аргументы, связанные с ключами.

Независимо от того, каким образом вызывается сценарий, с помощью ли `myscript -a -o xout.txt -v file1` или просто `myscript file1`, после выполнения команды `shift` переменная `$1` будет хранить значение `file1`, потому что промежуточные ключи и их аргументы будут удалены. Аргументами сценария теперь будут считаться все оставшиеся аргументы без ключей.

## Длинные ключи

Иногда одной буквы недостаточно, требуются полные слова или даже фразы для описания ключа. Встроенная функция `getopts` поддерживает и их.

Длинные ключи должны как-то отличаться от нескольких однобуквенных ключей, объединенных вместе. Например, означает ли ключ `-last` объединение ключей `-l -a -s -t` или это длинный ключ — слово `last`? Поэтому длинные ключи начинаются с двух дефисов (т. е. в рассмотренном случае длинный ключ должен передаваться как `--last`).

Чтобы использовать `getopts` для анализа длинных ключей, нужно добавить в список параметров знак минус и двоеточие, а затем еще один оператор `case` для распознавания каждого из длинных ключей. Двоеточие должно включаться в список параметров, *даже если длинный ключ не принимает аргументов* (мы объясним эту особенность в следующих разделах).

В примере 8.2 приведен сценарий, созданный на основе предыдущего примера, но анализирующий два дополнительных длинных ключа: `--amode` и `--outfile`. Первый означает то же, что и короткий ключ `-a`; второй действует так же, как `-o`, принимающий аргумент. Имя длинного ключа и соответствующий ему аргумент могут передаваться сценарию одним из двух способов: как одно слово со знаком равенства в качестве разделителя или как два слова. Например, вызов с параметром `--outfile=file.txt` или `--outfile file.txt` сообщает сценарию имя выходного файла. Команда `getopts` и второй оператор `case` в примере 8.2 смогут обработать как короткие, так и длинные ключи.

**Пример 8.2.** Анализ аргументов с помощью getopt: длинные ключи

```
#!/usr/bin/env bash
# parseslong.sh: использование getopt для анализа аргументов, включая длинные
# Автор и дата: _bash Idioms_ 2022
# Имя файла в bash Idioms: examples/ch08/parselong.sh
#
# Длинные ключи: --amode
#                   и --outfile filename или --outfile=filename
#
VERBOSE=':' # По умолчанию выключен

while getopt ':--:ao:v' VAL ; do ①
    case $VAL in
        a ) AMODE=1 ;;
        o ) OFILE="$OPTARG" ;;
        v ) VERBOSE='echo' ;;
    #-----
    - ) # Этот раздел добавлен для поддержки длинных ключей ②
        case $OPTARG in
            amode      ) AMODE=1 ;;
            outfile=* ) OFILE="${OPTARG#*=}" ;; ③
            outfile   ) ④
                OFILE="${!OPTIND}" ⑤
                let OPTIND++ ⑥
            ;;
            verbose   ) VERBOSE='echo' ;;
            * )
                echo "unknown long argument: $OPTARG"
                exit
            ;;
        esac
    ;;
    : ) echo "error: no argument supplied" ;;
    * )
        echo "error: unknown option $OPTARG"
        echo " valid options are: aov"
    ;;
    esac
done
shift $((OPTIND -1))
```

Как работает поддержка длинных ключей? Давайте разберемся:

① Команда getopt разрабатывалась для поддержки односимвольных ключей (например, -a). Добавление знака минус в список ее параметров означает, что два дефиса (--) будут распознаваться как допустимый ключ.

- 
- ❷ Любой символы, следующие за двумя дефисами, будут считаться «аргументом» ключа и помещаться в переменную \$OPTARG. Для сопоставления значения \$OPTARG с длинными именами ключей используется вложенный оператор `case`.
  - ❸ Как обрабатываются длинные ключи, например `--outfile`, которые должны сопровождаться аргументами? Если аргумент передан со знаком равенства, например `--outfile=my.txt`, то `getopts` присвоит всю строку (после `--`) переменной \$OPTARG. Извлечь аргумент из строки можно с помощью механизма расширения параметров (см. раздел «Расширение параметров» в главе 4), удалив (`\#`) все символы (`\*`) до знака равенства (`=`) включительно. В результате останутся только символы, следующие за знаком равенства, т. е. собственно аргумент. Также для извлечения аргумента можно было бы явно указать удаляемую строку: `OFILE="${OPTARG##outfile=}"`.
  - ❹ Когда аргумент передается как отдельное слово, выполнится второй вариант `outfile` в операторе `case`. Здесь используется переменная \$OPTIND, в которой `getopts` сохраняет текущее местоположение в анализируемой строке параметров.
  - ❺ Аргумент с именем файла извлекается *косвенно* с помощью \${!OPTIND}. Это работает следующим образом. В переменной \$OPTIND хранится индекс следующего аргумента, который должен быть обработан `getopts`. Восклицательный знак (!) сообщает о косвенной ссылке, когда значение \$OPTIND используется как имя извлекаемой переменной. Например, если ключ `--output` был третьим аргументом, встреченным командой `getopts`, то в этот момент \$OPTIND будет иметь значение 4, и, следовательно, \${!OPTIND} вернет значение \${4}, т. е. следующий аргумент с именем файла.
  - ❻ В заключение цикла нужно снова «сдвинуть» \$OPTIND, чтобы перейти к еще не обработанным ключам.

Остальная часть сценария осталась неизменной.

## HELP!

В предыдущих примерах есть явное упущение — отсутствие справки или подсказок, описывающих ключи и аргументы. Действительно, ключей `-h` и/или `--help` в этих примерах нет. Мы настоятельно рекомендуем предусматривать вывод по запросу справочной информации. Соответствующие ключи легко согласуются с большинством других инструментов и реализовать их несложно.

Для простых сценариев подойдет решение, показанное в примере 8.3.

### Пример 8.3. Справка на скорую руку

```
PROGRAM=${0##*/} # Версия `basename` на языке bash ①
if [ $# -lt 2 -o "$1" = '-h' -o "$1" = '--help' ]; then ②
    # Отступы в строках, следующих за ЕоН (End-of-Help), оформлены табуляциями!
    cat <<-EoH
        This script does nothing but show help; a real script should be more
exciting. ③
        usage: $0 <foo> <bar> (<baz>)

    As you can see, there are two required arguments, +foo+ and +bar+, and
one optional argument, +baz+.
    e.g.
        usage: $PROGRAM foo bar
        usage: $PROGRAM foo bar baz

    You can put more detail here if you need to.
    EoH
    # Отступ в строке выше оформлен табуляциями! ④
    exit 1 ⑤
fi
```

Разберем код:

① Об этой идиоме мы рассказывали в разделе «Сокращенный вариант команды basename» в главе 4.

② Если количество аргументов (`$#`) меньше (`-lt`) двух, *или* первый аргумент `-h`, *или* первый аргумент `--help`, то для отображения справки следует использовать *встроенный документ* (here-document). Мы обсудим такие документы в разделе «Встроенные документы и строки» в главе 9.

❸ Символы << указывают, что далее начинается встроенный документ, который заканчивается меткой ЕоН (End-of-Help — конец справки). В качестве метки можно использовать и другую последовательность символов на ваш выбор. Дефис (-) нужен, чтобы из следующих ниже строк автоматически удалялись начальные табуляции (но не пробелы). Это позволяет использовать в коде абзацные отступы, которые будут игнорироваться при выводе.

❹ Отступ метки конца справки тоже оформлен табуляциями, как и в тексте справки.

❺ Можно спорить о том, какой код выхода (см. раздел «Код выхода» в главе 9) правильнее использовать в этом случае. Как правило, в случае успеха принято использовать `exit 0`, а `exit 1` (допустимо любое значение больше нуля) — в случае ошибки. В нашем случае мы попросили о помощи и получили ее, и, поскольку сценарий отработал правильно, логичнее было бы завершить его командой `exit 0`. Если бы мы не передали нужные аргументы, то можно было бы говорить об ошибке и выполнить `exit 1`. Однако наш сценарий не сделал ничего полезного, кроме вывода сообщения, поэтому завершение командой `exit 1` с признаком ошибки также можно считать корректным. Выберите для себя один из вариантов и придерживайтесь его, но `exit 1` является более безопасным.

В примере 8.3 мы поместили справочное сообщение в код сценария, но иногда имеет смысл предусмотреть для этого специальную функцию, чтобы при необходимости можно было вызвать справку из разных мест, как показано в примере 8.4.

#### **Пример 8.4.** Анализ аргументов с помощью getopt: длинные ключи и справка

```
#!/usr/bin/env bash
# parselonghelp.sh: использование getopt для анализа аргументов,
# включая длинные ключи и справку
# Автор и дата: _bash Idioms_ 2022
# Имя файла в bash Idioms: examples/ch08/parselonghelp.sh
#
# Длинные ключи: --amode и --help
#                   и --outfile filename или --outfile=filename
```

---

```
PROGRAM=${0##*/} # версия `basename` на языке bash
VERSION="$PROGRAM v1.2.3"

VERBOSE=':' # По умолчанию выключен
DEBUG=':' # По умолчанию выключен

function Display_Help {
    # Отступы в строках, следующих за ЕоН (End-of-Help), оформлены табуляциями!
    cat <<-EoN
        Этот сценарий просто выводит справку
        usage: $PROGRAM (options)

    Options:
        -a | --amode    = Enable "amode", default is off
        -d | --debug   = Include debug output, default is off
        -h | --help     = Show this help message and exit
        -o | --outfile = Send output to file instead of STDOUT
        -v | --verbose  = Include verbose output, default is off
        -V | --version  = Show the version and exit

    You can put more detail here if you need to.
    EoN
    # Отступ в строке выше оформлен табуляциями!
    # Если раскомментировать строку ниже, сценарий всегда будет завершаться
    # после вызова Display_Help. Нужно это или нет -- решайте сами
    # exit 1 # Если вы решите раскомментировать эту строку, то удалите все
    # операторы
    # exit, следующие за вызовом этой функции!
} # Конец функции Display_Help

while getopts '::-:adho:vV' VAL ; do
    case $VAL in
        # Если упорядочить ключи по алфавиту, их будет проще найти в будущем,
        # а также уменьшится риск конфликтов
        a ) AMODE=1 ;;
        d ) DEBUG='echo' ;;
        h ) Display_Help ; exit 1 ;; # Здесь мы нарушили наш стиль
        o ) OFILE="$OPTARG" ;;
        v ) VERBOSE='echo' ;;
        V ) echo "$VERSION" && exit 0 ;; # Здесь мы снова нарушили наш стиль
    -----
        -) # Этот раздел добавлен для поддержки длинных ключей
        case $OPTARG in
            amode ) AMODE=1 ;;
            debug ) DEBUG='echo' ;;
            help )
                Display_Help
                exit 1
```

```

;;
outfile=* ) OFILE="\${OPTARG#=}" ;;
outfile )
    OFILE="\${!OPTIND}"
    let OPTIND++
;;
verbose ) VERBOSE='echo' ;;
version )
    echo "$VERSION"
    exit 0
;;
* )
    echo "unknown long argument: \$OPTARG"
    exit
;;
esac
;;
#-----
: ) echo "error: no argument supplied" ;;
* )
    echo "error: unknown option \$OPTARG"
    echo " valid options are: aov"
;;
esac
done
shift $((OPTIND -1))

echo "Code for \$0 goes here."

$VERBOSE 'Example verbose message...'

$DEBUG 'Example DEBUG message...'

echo "End of $PROGRAM run."

```

Пример получился длинным, но большую часть этого кода вы уже видели. Мы добавили справку в пример с возможностью анализа длинных ключей, но, кроме того, предусмотрели поддержку ключей *debug* и *verbose* для отладочного и подробного режимов вывода. Также мы реализовали вывод справки с помощью функции *Display\_Help*, потому что теперь справка должна выводиться минимум в двух местах: при обработке коротких и длинных ключей. Это достаточно ясно.

Мы также нарушили наше соглашение по стилю *case..esac* об использовании односторонних инструкций в вариантах, потому что добавление

команды `exit` сделало блоки многострочными. Мы могли бы записать код в несколько строк (как было сделано в примере 8.2), но заменили четыре строки кода одной, благодаря чему сэкономили три строки экранного пространства (которые мы лучше потратим на более важный код), пусть и за счет некоторой потери ясности. Сравните:

```
h ) Display_Help ; exit 1 ;;
# и
h )
    Display_Help
    exit 1
;;
```

## Отладочный и подробный режимы вывода

В примере 8.4 мы присвоили переменной `VERBOSE` двоеточие в качестве значения по умолчанию. Это может показаться странным, но рассмотрим пример использования этой переменной далее в сценарии:

```
$VERBOSE 'Example verbose message...'
```

В случае по умолчанию — если пользователь не потребовал подробного режима вывода — при выполнении этой строки `$VERBOSE` заменяется на `:`, команду `no-op` (без операции) или `null`, которая ничего не делает, игнорирует свои аргументы и всегда возвращает истинное значение. В результате сообщение не будет выводиться. Но если пользователь вызывает сценарий с ключом `-v` или `--verbose`, то `$VERBOSE` получает значение `echo`, и тогда соответствующая строка примет вид:

```
echo 'Example verbose message...'
```

и выведет сообщение.

Эта идиома вывода сообщений по условию легко запоминается, читается и находится с помощью `grep` при написании документации. Тот же прием можно использовать для вывода отладочных сообщений.

## Версия

По аналогии с ключами `-h` и/или `--help` может пригодиться ключ, управляющий выводом номера версии сценария. Прежде всего нужно решить, что будет означать ключ `-v` — *версию* (version) или *подробный вывод* (verbose)? Мы предпочтаем использовать вариант с прописной буквой `-V` для вывода версии и со строчной `-v` — для включения режима подробного вывода. А вообще *нужна ли* информация о версии? Не так часто, как справка, но иногда она действительно требуется. Системы управления версиями 2-го поколения, такие как CVS и SVN, упростили работу с версиями — просто используйте расширение ключевого слова, например `VERSION=$Id$`. Но системы 3-го поколения, такие как Git, не поддерживают этой возможности, поэтому номер версии придется обновлять вручную или же использовать какой-то механизм контроля времени сборки или непрерывной интеграции. Для больших или находящихся в общем доступе сценариев вывод номера версии может быть важной функцией, а для небольших сценариев, предназначенных для личного использования, это менее важно. Нужна ли вам данная возможность — решайте сами.

Поддержка вывода *версии* реализуется с помощью присвоения `VERSION` некоторого значения, которое отображается по тому же принципу, что и *справка*. Выбор кода выхода (см. «Код выхода» в главе 9) в этом случае проще, поскольку версия выводится, только если она была запрошена, и `exit 0` выглядит логичнее.

Некоторые варианты форматов значений для присвоения переменной `$VERSION`:

- `VERSION=$Id$` для CVS или SVN;
- `VERSION=v1.2.3;`
- `VERSION="$PROGRAM v1.2.3";`
- `VERSION="$0 v1.2.3"` (возможно, не лучший способ, потому что `$0` может меняться);
- `VERSION=12.`

## В заключение: стиль и удобочитаемость

В этой главе мы обсудили ключи, параметры и аргументы командной строки. Мы показали простой способ анализа одного аргумента, а также как прервать выполнение сценария при получении недостаточного количества аргументов, проверив значение переменной `$#`. Далее мы рассмотрели встроенную функцию `getopts` — хороший инструмент анализа коротких и длинных ключей.

При разработке сценариев, поддерживающих большой набор аргументов, необходимо разделять код, анализирующий аргументы, и основные функции. Часто лучшим средством является установка переменных-флагов в операторах `case`, на которые можно сослаться в частях сценария, реализующих основную функциональность. Удобочитаемость кода сценария можно значительно улучшить, если анализ аргументов выполняет специальная функция. Оберните код из примера 8.4 в определение функции:

```
# Пример вызова: parseargs "${@}"
function parseargs {
    ...
}
```

затем вызовите ее как `parseargs "${@}"` — и остальной код в вашем сценарии сможет использовать установленные ею флаги.

Условный вывод большого количества отладочной или расширенной (подробной) информации может ухудшить удобочитаемость кода, если заключить соответствующие операторы в операторы `if`. Последние загромождают и усложняют логику кода. Мы показали идиому, которая позволяет добиться того же результата, но избавляет от необходимости использовать операторы `if`, что упрощает код и его чтение.

## ГЛАВА 9

---

# Файлы и не только

Что превращает обычные файлы в сценарии командной оболочки и как вернуть коды выхода? Как читать файлы? Об этом и многом другом мы поговорим в настоящей главе.

## Чтение файлов

Существуют три основных идиоматических способа чтения файлов в сценариях bash. Некоторые из них загружают файл в память целиком, другие читают его содержимое построчно.

### read

В главе 2 мы использовали команду `read` для обработки пар «ключ/значение»; еще раз мы встретимся с ней в разделе «Получение ввода пользователя» в главе 10. Другим важным применением этой команды является чтение файлов и анализ ввода сразу во всей строке:

```
$ grep '^nobody' /etc/passwd | read -d':' user shadow uid gid gecos home shell  
$ echo "$user | $shadow | $uid | $gid | $gecos | $home | $shell"  
| | | | | |
```

Что случилось, где наши данные? Проблема в том, что они попали в подоболочку, созданную конвейером (`|`), но так и не вышли оттуда. Давайте попробуем следующий вариант:

```
$ grep '^nobody' /etc/passwd | { \
    read -d':' user shadow uid gid gecos home shell; \
    echo "$user | $shadow | $uid | $gid | $gecos | $home | $shell" \
}
nobody | | | | | |
```

Немного лучше, но где остальные данные? Дело в том, что `-d` — это разделитель по концу строки, а не по полям (`$IFS`). Попробуем еще раз:

```
$ grep '^nobody' /etc/passwd | { \
    IFS=':' read user shadow uid gid gecos home shell; \
    echo "$user | $shadow | $uid | $gid | $gecos | $home | $shell"; \
}
nobody | x | 65534 | 65534 | nobody | /nonexistent | /usr/sbin/nologin
```

Мы еще поговорим об этом синтаксисе в разделе «Изменяем `$IFS` при чтении файлов» ниже в этой главе.



### **lastpipe**

В bash версий 4.0 и выше можно установить параметр `shopt -s lastpipe`, чтобы последняя команда конвейера выполнялась в текущей оболочке и сценарий мог видеть окружение. Обратите внимание, что этот прием работает, только если отключено *управление заданиями* (в сценариях оно отключено по умолчанию, но может быть включено в интерактивном сеансе). Отключить управление заданиями можно с помощью команды `set +m`, но при этом выключится реакция на комбинации клавиш `CTRL-C` и `CTRL-Z`, а также команды `fg` и `bg`, поэтому мы не рекомендуем ее использовать.

## **mapfile**

Команда `mapfile` или `readarray` читает файл в массив (список). Она была добавлена в bash, начиная с версии 4.0. Наиболее часто используемые параметры: `-n count`, ограничивающий количество читаемых строк; `-s count`, пропускающий указанное количество строк; `-c/-C`, включающий отображение индикатора хода выполнения операции. Эта команда намного проще в использовании, чем другие методы.

В самом простом случае она загружает весь файл в память, как показано в примере 9.1.

### Пример 9.1. Простое использование mapfile

```
mapfile -t nodes < /path/to/list/of/hosts # -t удаляет символы перевода строки  
  
# Цикл, обходящий узлы  
for node in ${nodes[@]}; do  
    ssh $node 'echo -e "$HOSTNAME:\t$(uptime)"'  
done
```

Ключ `-n` немного усложняет работу, потому что нужно проверять, действительно ли были прочитаны какие-то данные (значение `#{#nodes[@]}` отлично от нуля), иначе `while mapfile` будет выполняться вечно (пример 9.2).

### Пример 9.2. Пример использования mapfile для чтения файла порциями

```
BATCH=10  
# Прочитать данные...           && если данные доступны!  
while mapfile -t -n $BATCH nodes && ((#${#nodes[@]})); do  
    for node in ${nodes[@]}; do  
        ssh $node 'echo -e "$HOSTNAME:\t$(uptime)"'  
    done  
done < /path/to/list/of/hosts
```

Конечно, можно придумать что-то более изощренное и добавить обратную связь с пользователями. Также могут понадобиться ограничения процесса, как показано в примере 9.3.

### Пример 9.3. Более сложный пример использования mapfile: код

```
#!/usr/bin/env bash  
# fancy_mapfile.sh: Сложный пример использования mapfile  
# Автор и дата: _bash Idioms_ 2022  
# Имя файла в bash Idioms: examples/ch09/fancy_mapfile.sh  
#  
# Не работает в Zsh 5.4.2!  
  
HOSTS_FILE='/tmp/nodes.txt'  
  
# Создаем тестовый файл  
> $HOSTS_FILE  
for n in node{0..9}; do echo "$n" >> $HOSTS_FILE; done
```

```
### ПЕРЕМЕННЫЕ С НАСТРОЙКАМИ
#BATCH_SIZE=0 # Прочитает файл целиком (по умолчанию); следите за памятью
BATCH_SIZE=4
SLEEP_SECS_PER_NODE=1 # Можно присвоить 0
SLEEP_SECS_PER_BATCH=1 # Следует присвоить 0, если `BATCH_SIZE=0` !

# Вывод информации в STDERR,
# чтобы STDOUT можно было перенаправить в tee или в файл
node_count="$(cat $HOSTS_FILE | wc -l)" ①
batch_count="$(( node_count / BATCH_SIZE ))" ②
echo '' 1>&2
echo "Nodes to process:      $node_count" 1>&2
echo "Batch size and count:   $BATCH_SIZE / $batch_count" 1>&2 ③
echo "Sleep seconds per node: $SLEEP_SECS_PER_NODE" 1>&2
echo "Sleep seconds per batch: $SLEEP_SECS_PER_BATCH" 1>&2
echo '' 1>&2

node_counter=0
batch_counter=0
# Прочитать данные...          && если данные доступны в $HOSTS_FILE
while mapfile -t -n $BATCH_SIZE nodes && (( ${#nodes[@]} )); do
    for node in ${nodes[@]}; do ④
        echo "node $(( node_counter++ )): $node"
        sleep $SLEEP_SECS_PER_NODE
    done
    (( batch_counter++ ))
    # Чтобы не попасть сюда ПОСЛЕ чтения последнего (неполного) пакета...
    [ "$node_counter" -lt "$node_count" ] && {
        # Для вывода также можно использовать mapfile -C Call_Back -c $BATCH_SIZE,
        # но тогда обратный вызов выполняется заранее, поэтому возможна задержка
        # вывода
        echo "Completed $node_counter of $node_count nodes;" \
            "batch $batch_counter of $batch_count;" \
            "sleeping for $SLEEP_SECS_PER_BATCH seconds..." 1>&2
        sleep $SLEEP_SECS_PER_BATCH
    }
done < $HOSTS_FILE
```

Разберем код:

① Это не «бесполезное» использование `cat`. Обычно `wc -l` выводит <количество строк> <имя файла>, но нам нужно только <количество строк>, а получить его напрямую невозможно. Однако при чтении из стандартного ввода `wc` не отображает имя файла, то есть мы получаем именно то, что нужно.

❷ Вы наверняка помните, что bash поддерживает только целочисленную арифметику, поэтому при подсчете пакетов могут возникать ошибки усечения.

❸ Целочисленная арифметика; см. п. ❷.

❹ Мы не заключили в кавычки \${nodes[@]} в цикле `for`, но это допустимо в данном случае, потому что мы читаем имена хостов, в которых не может быть пробелов. Однако лучше выработать привычку всегда использовать кавычки.

Результат выполнения этого сценария приведен в примере 9.4.

**Пример 9.4.** Более сложный пример использования `mapfile`: вывод

```
Nodes to process:      10
Batch size and count:  4 / 2
Sleep seconds per node: 1
Sleep seconds per batch: 1

node 0: node0
node 1: node1
node 2: node2
node 3: node3
Completed 4 of 10 nodes; batch 1 of 2; sleeping for 1 seconds...
node 4: node4
node 5: node5
node 6: node6
node 7: node7
Completed 8 of 10 nodes; batch 2 of 2; sleeping for 1 seconds...
node 8: node8
node 9: node9
```

Источники дополнительной информации:

- `help mapfile`;
- `help readarray` (в этом случае вернется справка для `mapfile`);
- описание приемов эффективного использования `mapfile` для очистки корзины AWS S3: <https://oreil.ly/xie4t>.

## Метод «грубой силы»

Пример 9.5 демонстрирует, как прочитать файл в память целиком.

### Пример 9.5. Пример чтения файла методом «грубой силы»

```
for word in $(cat file); do
    echo "word: $word"
done

### Или, убрав "бесполезный" cat
for word in $(< file); do
    echo "word: $word"
done
```

## Изменяем \$IFS при чтении файлов

`IFS` — это аббревиатура от Internal Field Separator (внутренний разделитель полей). Переменная `$IFS` применима во всех случаях, когда требуется *разбить строку на слова*. По умолчанию используется конструкция `<пробел><табуляция><перевод строки>` или `IFS=$' \t\n'` с механизмом экранирования `' '`, соответствующим стандарту ANSI C<sup>1</sup>. Она служит основой для многих идиом bash. Если изменить `$IFS`, не понимая последствий, возможны неожиданности. В частности, замена первого символа в значении `$IFS`, который по умолчанию является пробелом и используется в расширениях слов, может привести к хаосу. Если вы уверены, что вам нужно изменить значение `$IFS`, сделайте это либо в функции, используя локальную переменную, либо локально по отношению к команде (раздел «Локальные переменные» в главе 10). Пример 9.6 иллюстрирует, как правильно изменять `$IFS`.

### Пример 9.6. Изменение IFS при использовании команды read: код

```
#!/usr/bin/env bash
# fiddle-ifs.sh: Манипуляции с $IFS при чтении файлов
# Автор и дата: _bash Idioms_ 2022
# Имя файла в bash Idioms: examples/ch09/fiddle-ifs.sh
#
```

---

<sup>1</sup> <https://oreil.ly/00I9N>.

```

# Создать тестовый файл (слово word сокращено, чтобы ширина вывода
# не превышала 80 символов)
IFS_TEST_FILE='/tmp/ifs-test.txt'
cat <<'EoF' > $IFS_TEST_FILE
line1 wd1 wd2 wd3
line2 wd1 wd2 wd3
line3 wd1 wd2 wd3
EoF

#-----
echo 'Normal $IFS and `read` operation; split into words:'
printf '$IFS before: %q\n' "$IFS"
while read line w1 w2 w3; do
    printf 'IFS during: %q\tline = %q, w1 = %q, w2 = %q, w3 = %q\n' \
        "$IFS" "$line" "$w1" "$w2" "$w3"
done < $IFS_TEST_FILE
printf 'IFS after: %q\n' "$IFS"

#-----
echo ''
echo 'Temporary $IFS change for `read` inline:'
echo 'Words are NOT split, yet $IFS appears unchanged, because only the read'
echo 'line has the changed $IFS. We also shortened "line" to "ln" to make'
echo 'it fit a book page.'
printf 'IFS before: %q\n' "$IFS"
while IFS=' ' read line w1 w2 w3; do
    printf 'IFS during: %q\tln = %q, w1 = %q, w2 = %q, w3 = %q\n' \
        "$IFS" "$line" "$w1" "$w2" "$w3"
done < $IFS_TEST_FILE
printf 'IFS after: %q\n' "$IFS"

#-----
function Read_A_File {
    local file="$1"
    local IFS=''
    while read line w1 w2 w3; do
        printf 'IFS during: %q\tline = %q, w1 = %q, w2 = %q, w3 = %q\n' \
            "$IFS" "$line" "$w1" "$w2" "$w3"
    done < $IFS_TEST_FILE
}

echo ''
echo 'Temporary $IFS change for `read` in a function; NOT split, $IFS changed:'
printf 'IFS before: %q\n' "$IFS"
Read_A_File
printf 'IFS after: %q\n' "$IFS"

#-----
echo ''
echo 'But you may not need to change $IFS at all... See `help read` and'

```

```

echo 'note the parts about:'
echo ' ...leftover words assigned to the last NAME'
echo ' ...[read line until] DELIM is read, rather than newline'
echo 'Normal $IFS and `read` operation using only 1 variable:'
printf 'IFS before: %q\n' "$IFS"
while read line; do
    printf 'IFS during: %q\tline = %q\n' "$IFS" "$line"
done < $IFS_TEST_FILE
printf 'IFS after: %q\n' "$IFS"

```

Вывод этого сценария показан в примере 9.7.

### Пример 9.7. Изменение IFS при использовании команды read: вывод

```

Normal $IFS and `read` operation; split into words:
$IFS before: $' \t\n'
IFS during: $' \t\n'      line = line1, w1 = wd1, w2 = wd2, w3 = wd3
IFS during: $' \t\n'      line = line2, w1 = wd1, w2 = wd2, w3 = wd3
IFS during: $' \t\n'      line = line3, w1 = wd1, w2 = wd2, w3 = wd3
IFS after: $' \t\n'

Temporary $IFS change for `read` inline:
Words are NOT split, yet $IFS appears unchanged, because only the read
line has the changed $IFS. We also shortened "line" to "ln" to make
it fit a book page.
IFS before: $' \t\n'
IFS during: $' \t\n'      ln = line1\ wd1\ wd2\ wd3, w1 = '', w2 = '', w3 = ''
IFS during: $' \t\n'      ln = line2\ wd1\ wd2\ wd3, w1 = '', w2 = '', w3 = ''
IFS during: $' \t\n'      ln = line3\ wd1\ wd2\ wd3, w1 = '', w2 = '', w3 = ''
IFS after: $' \t\n'

Temporary $IFS change for `read` in a function; NOT split, $IFS changed:
IFS before: $' \t\n'
IFS during: '' line = line1\ wd1\ wd2\ wd3, w1 = '', w2 = '', w3 = ''
IFS during: '' line = line2\ wd1\ wd2\ wd3, w1 = '', w2 = '', w3 = ''
IFS during: '' line = line3\ wd1\ wd2\ wd3, w1 = '', w2 = '', w3 = ''
IFS after: $' \t\n'

```

But you may not need to change \$IFS at all... See `help read` and note the parts about:

```

...leftover words assigned to the last NAME
...[read line until] DELIM is read, rather than newline
Normal $IFS and `read` operation using only 1 variable:
IFS before: $' \t\n'
IFS during: $' \t\n'      line = line1\ wd1\ wd2\ wd3
IFS during: $' \t\n'      line = line2\ wd1\ wd2\ wd3
IFS during: $' \t\n'      line = line3\ wd1\ wd2\ wd3
IFS after: $' \t\n'

```

Источники дополнительной информации по этому вопросу:

- раздел «Локальные переменные» в главе 10;
- раздел «Чтение файлов» в начале этой главы;
- раздел «ANSI-C Quoting» в справочном руководстве bash<sup>1</sup>.

## Имитации файлов

Мы знаем, что в UNIX, Linux и (конечно) bash ожидается, что все сущее будет файлом, верно? *Все есть файл. Это Путь Unix.* Но что делать, если нужно обработать только часть файла? Возможно, у вас есть несколько узлов, сообщающих статистику через определенные интервалы времени, и хотелось бы иметь возможность наблюдать за запуском и остановкой некоторых из них. Предположим, что поля в записях о состоянии каждого узла разделены табуляцией и каждый файл содержит множество записей с сообщениями, в которых имя узла указано в первом поле. Допустим, что вам требуется агрегировать эти данные за час, день или какой-то другой период. Для этого можно использовать временные файлы, хотя тогда придется потрудиться:

```
cut -f1 /path/to/previous-report.log | sort -u > /tmp/previous-report.log
cut -f1 /path/to/current-report.log | sort -u > /tmp/current-report.log
diff /tmp/previous-report.log /tmp/current-report.log
rm /tmp/previous-report.log /tmp/current-report.log
```

То же самое можно реализовать, используя так называемые «имитации файлов» (pretend files):

```
diff <(cut -f1 /path/to/previous-report.log | sort -u) \
      <(cut -f1 /path/to/current-report.log | sort -u)
```

Ладно, на самом деле этот прием называется подстановкой процессов, которая иллюстрируется примером 9.8, но термин «имитация файлов» показался нам более удачным.

---

<sup>1</sup> <https://oreil.ly/DFkqR>.

**Пример 9.8.** Пример простой подстановки процессов

```
$ head *report.log
==> current-report.log <==
always-talking
always-talking
always-talking
always-talking

==> previous-report.log <=
always-talking
going-away
always-talking
going-away
always-talking
going-away
always-talking
going-away

$ diff -u <(cut -f1 previous-report.log | sort -u) \
      <(cut -f1 current-report.log | sort -u)
--- /dev/fd/63 2022-01-09 20:01:37.857658587 -0500
+++ /dev/fd/62 2022-01-09 20:01:37.857658587 -0500
@@ -1,2 +1 @@
  always-talking
-going-away
```

## Настроочные каталоги

Каталог с возможностью добавления или удаления файла, содержащего настройки, для достижения определенного эффекта — идиома, востребованная во многих контекстах. Она необходима для дистрибутивов Linux, использующих системы управления пакетами, такие как RPM (Red Hat Package Manager) или Debian APT (Advanced Package Tool), потому что пакет может добавлять настроочные файлы при установке или стирать при удалении без необходимости их программного редактирования. Прекрасный пример — `/etc/cron.d/`; пакеты могут добавлять задания cron при установке и стирать их при удалении, не влияя ни на что другое.

Мы большие поклонники использования этого подхода для работы с файлами конфигурации. Например, представьте, что у вас есть файл

`/etc/syslog-ng/syslog-ng.conf`, содержащий инструкцию `@include "/etc/syslog-ng/syslog-ng.local.d"`. В таком случае вы сможете перетаскивать файлы в `/etc/syslogng/syslog-ng.local.d/`, чтобы настроить конфигурацию конкретного узла.

Реализация такого подхода на bash показана в примере 9.9.

### Пример 9.9. Подключение файлов с настройками

```
# Подключить локальные настройки, ЕСЛИ файл с ними существует
# И имеет разрешение на выполнение
for config in /etc/myscript.cfg.d/*; do
    # Файлы с настройками должны иметь разрешение на выполнение, иначе они
    # игнорируются, что позволяет легко отключать их
    [ -x "$config" ] && source "$config"
    # Или замените ` -x` на ` -r`, чтобы в качестве признака использовать
    # разрешение на чтение вместо разрешения на выполнение
done
```

Представленная идея основана на сценариях-обертках, которые обсуждались в главе 3. На ее основе можно создавать простые, расширяемые и в то же время мощные инструменты для себя или других пользователей. Суть такого подхода заключается в следующем: пишется простой скелетный сценарий с некоторой логикой для поиска в каталоге команд и дополнительным кодом, объединяющим эти команды и обеспечивающим возможность получения справки. Затем добавляется шаблон для новых модулей или действий, а пользователи просто копируют его и заполняют своими данными, чтобы добавить новую функцию.

## Организация библиотек

При наличии большого объема часто повторяющегося кода, такого как функция `Log`, использующая `printf %(%datefmt)T` (см. раздел «Получение и использование даты и времени» в главе 6), имеет смысл поместить его в библиотеку, чтобы уменьшить затраты на набор и обслуживание кода. Конечно, в этом случае вам придется решать типичные для библиотек проблемы развертывания и вызова. Если развертывание зависит от ваших процессов и практики, то с вызовом мы можем помочь.

Как ни странно, но для вызова вашей библиотеки мы предлагаем использовать повторяющийся «шаблонный» код. Его нельзя поместить в библиотеку, иначе процесс импорта зациклится. Но, поскольку у вас есть шаблон сценариев с типовым комментарием в заголовке и прочим (верно?), вы можете просто добавить в него код, показанный в примере 9.10.

#### **Пример 9.10.** Подключение библиотеки bash

```
# Подключить библиотеку
source /path/to/global/bash-library.lib || {
    echo "FATAL: $0 can't source '/path/to/global/bash-library.lib'"
    exit 1
}
```

Мы могли бы определить переменную `GLOBAL_LIBRARY=/path/to/global/bash-library.lib`, но для пары случаев использования в шаблонном коде в этом нет особого смысла. Если вы уверены в своих силах, то создайте глобальную конфигурацию переменных в этой библиотеке, а также разрешите переопределять переменные и, возможно, библиотечные функции, используя локальные настроочные каталоги (см. раздел «Настроочные каталоги» выше в этой главе).

## Shebang!

Что делает обычный файл сценарием командной оболочки? Стока shebang!

Shebang можно назвать заклинанием, которое сообщает ядру (не демонам), что делать с вашим кодом. Мы говорим о `#!/bin/bash` или, может быть, `#!/bin/sh` — первой строке в вашем сценарии. Пара символов `#!` интерпретируется одновременно и как комментарий, и как *магическое число*, сообщающее ядру, где искать интерпретатор, в данном случае `bash`.

В Linux наиболее распространена shebang-строка `#!/bin/bash`, но также можно встретить `#!/bin/sh`, `#!/bin/bash -`, `#!/usr/bin/env bash`, `#!/usr/bin/bash` и другие.

Мы рекомендуем `#!/bin/bash` или `#!/usr/bin/env bash` и настоятельно советуем не использовать `#!/bin/sh`, если только вы намеренно не ограничиваете свой код совместимостью с оболочкой Bourne. Раньше последний совет был неактуален для Linux, но теперь многие дистрибутивы переключились на использование командной оболочки *Debian Almquist shell*, и `/bin/sh` в них ссылается на `dash`, потому что эта оболочка намного компактнее, чем `bash`, и быстрее выполняет сценарии, в том числе загрузку системы. Но вообще использование `sh` — плохая идея, если в действительности имеется в виду `bash`, потому что такой шаг чреват ошибками и скрывает намерения программиста.

`#!/usr/bin/env bash` — значительно более переносимая строка shebang, потому что, пока `bash` находится в списке каталогов в `$PATH`, система найдет его. Это работает и в случаях, когда `bash` находится не в `/bin/`, как, например, в BSD, Solaris и других системах, отличных от Linux. Единственный минус — дополнительные расходы ресурсов на запуск `env`, поиск в `$PATH` и последующий запуск `bash`. В современных компьютерах это почти не имеет значения, но в системах с ограниченными ресурсами, которые запускают множество сценариев за короткое время, такие дополнительные расходы могут оказаться нежелательными (пример 9.11).

### Пример 9.11. Тестирование скорости разных строк shebang

```
$ head examples/ch09/shebang*
==> examples/ch09/shebang-bash.sh <==
#!/bin/bash -
:

==> examples/ch09/shebang-env.sh <==
#!/usr/bin/env bash
:

$ time for ((i=0; i<1000; i++)); do examples/ch09/shebang-bash.sh; done
real    0m3.279s
user    0m1.273s
sys     0m0.915s

$ time for ((i=0; i<1000; i++)); do examples/ch09/shebang-env.sh; done
real    0m4.291s
user    0m1.313s
sys     0m1.425s
```

Еще одна деталь, о которой следует упомянуть, — символ дефиса (-) в конце первого варианта строки. Он необходим, чтобы предотвратить подмену очень старым интерпретатором; подробности см. в разделе 4.7 сборника часто задаваемых вопросов по Unix<sup>1</sup>. Ядро Linux (но *не* ядра других систем!) обрабатывает все, что следует за первым «словом» (в данном случае `/bin/bash`), как один аргумент. Предотвратить это можно, добавив дефис — признак конца аргументов. Обратите внимание, что при использовании `#!/usr/bin/env bash` в дефисе нет необходимости, так как `bash` в этой строке является аргументом.

## Строгий режим bash

В Perl есть директива `use strict`, ограничивающая возможность использования «небезопасных конструкций». Она заставляет интерпретатор генерировать ошибку времени выполнения, если, например, переменные не были объявлены и инициализированы до их использования. В статье о неофициальном строгом режиме bash<sup>2</sup> утверждается, что то же самое (по сути) в bash обеспечивают следующие инструкции:

```
set -euo pipefail
IFS=$'\n\t'
```

Мы не вполне согласны с таким изменением `$IFS`, потому что оно делает неработоспособными многие идиомы bash (см. раздел «Изменяем `$IFS` при чтении файлов» выше в этой главе). Хотя автор статьи приводит достаточно убедительные аргументы, не следует забывать, что адаптация ранее написанного кода под новые требования может потребовать значительных усилий. А вот команда `set` великолепна:

- `-e` завершит сценарий при первой же ошибке;
- `-u` завершит сценарий, если обнаружится попытка обращения к неинициализированной переменной;
- `-o pipefail` вызовет сбой всего конвейера, если любая команда в нем, а не только последняя, завершится с ненулевым кодом.

---

<sup>1</sup> <http://bit.ly/2fdmYSl>.

<sup>2</sup> <https://oreil.ly/mZX7f>.

Ключ `-u` сломает многие сценарии, но исправить их будет довольно легко, к тому же этот параметр поможет найти опечатки в именах переменных, о которых вы не подозревали. Применение `-eo pipefail` будет реже приводить к поломке готовых сценариев. Если же такое случится, то, скорее всего, сценарий уже содержал ошибки, но они были замаскированы, а `-eo pipefail` лишь вскрыли проблему.

Подумайте о добавлении указанной инструкции в свои шаблоны и руководства по стилю, она определенно стоит того.

Источники дополнительной информации:

- <http://redsymbol.net/articles/unofficial-bash-strict-mode>;
- `help set`;
- <https://perldoc.perl.org/strict>.

## Код выхода

Если не указано иное, сценарий bash завершится с кодом выхода (или возврата) последней выполненной команды. Но иногда может потребоваться завершить сценарий раньше, например после неудачной проверки или по каким-то другим причинам.

В bash ноль означает  *успех*, а ненулевое значение — неудачу. Этот принцип отличен от многих других языков, но был принят, потому что есть только один оптимальный способ достижения успеха, а неудачу можно потерпеть множеством способов. Причем иногда желательно сообщить, по какой причине сценарий потерпел неудачу, например:

```
if grep --quiet "$HOSTNAME" /etc/hosts; then ...
```

Код выхода в bash состоит всего из восьми бит, поэтому максимальное значение равно 255, но вы можете использовать только коды не больше 125, потому что коды 126 и выше зарезервированы, как показано в табл. 9.1.

**Таблица 9.1.** Коды выхода/возврата в bash

Код завершения/возврата	Описание
0	Успех
1, 3–125	Коды ошибок, определяемые сценарием
2	Некорректное использование встроенных команд bash
126	Команда найдена, но файл не имеет разрешения на выполнение
127	Команда не найдена
128–255	Команда завершилась по сигналу N–128

Смотрите также описание кодов выхода в документации GNU<sup>1</sup>.

Команда, завершающая сценарий с кодом выхода (звучит барабанная дробь), — `exit n`. Команда возврата из функции (снова барабанная дробь) — `return n`, где `n` — необязательный аргумент. Не забывайте, что `return` возвращает код выхода, а не возвращаемое значение. `n` может быть любым выражением командной оболочки, таким как `5`, `$val` или более сложным. Если `n` не указан, то возвращается код завершения последней команды.

За дополнительной информацией обратитесь к нашему обсуждению `exit 0` и `exit 1` в разделе «HELP!» в главе 8.

## Это ловушка!

Встроенная команда `trap` (см. `help trap`) выполняет заданный код при получении сценарием *сигнала*, в том числе при выходе. Это позволяет «прибрать за собой», даже если кто-то нажмет `CTRL-C` (но не `kill -9`). Ловушки также можно использовать для запуска отладочного кода, но лучше просто добавить инструкцию `set -x` или запустить сценарий командой `bash -x`.

<sup>1</sup> <https://oreil.ly/jkAnE>.

Выполните команду `trap -l`, чтобы получить список всех поддерживаемых системой сигналов, или `trap -p <сигнал>`, чтобы показать команды `trap`, связанные с этим сигналом.

Как уже отмечалось, типичное использование `trap` — очистка и освобождение ресурсов после завершения сценария. Она имеет синтаксис: `trap [-lp] [[arg] signal_spec ...]`, где `arg` может быть автономным кодом, но обычно проще заранее определить функцию, а здесь лишь вызвать ее, как показано в примере 9.12 (вывод этого сценария показан в примере 9.13).

**Пример 9.12.** Пример простейшей ловушки: код

```
#!/usr/bin/env bash
# trivial_trap.sh: Пример простейшей ловушки в bash
# Автор и дата: _bash Idioms_ 2022
# Имя файла в bash Idioms: examples/ch09/trivial_trap.sh
#
# Не работает в Zsh 5.4.2!

function Cleanup {
    echo "$FUNCNAME was triggered! Cleaning up..."
}

echo "Starting script $0..."

echo 'Setting the trap...'
# Вызвать Cleanup по любому из этих шести сигналов
trap Cleanup ABRT EXIT HUP INT QUIT TERM

echo 'About to exit...'
```

**Пример 9.13.** Пример простейшей ловушки: вывод

```
Starting script examples/ch09/trivial_trap.sh...
Setting the trap...
About to exit...
Cleanup was triggered! Cleaning up...
```

Рекомендуем ознакомиться также с описанием `trap` в документации GNU<sup>1</sup> и разделом «Отладка в bash» в главе 10.

---

<sup>1</sup> <https://oreil.ly/KDuFJ>.

## Встроенные документы и строки

Встроенные конструкции поддерживаются в разных языках, например `/* ... */` в C и `''' ... '''` в Python, но в bash есть особые идиомы *встроенных документов* (*here-document*) и *встроенных строк* (*here-string*). Мы уже использовали встроенные документы в примере 8.3, однако есть и другие возможности их применения. Встроенные документы позволяют хранить и отображать блоки текста при использовании минимума разделителей и команд.

Синтаксис встроенного документа: `[fd]<<[-]["]word[""]`, где `fd` — необязательный и редко используемый идентификатор файла, дефис (`-`) означает удаление начальных табуляций (но не пробелов), а необязательные кавычки указывают, что не следует интерполировать содержимое. Слово `word` не подлежит расширению с помощью параметров, переменных и имен файлов, а так же подстановке команд и арифметическим операциям. Кроме того, `word` не может быть переменной. Интерполяция документа не выполняется, даже если какая-то часть `word` заключена в кавычки. Мы рекомендуем использовать одинарные кавычки, так как это исключает интерполяцию в других местах кода.

Добавление дефиса (`-`) позволит оформить встроенный документ в коде с отступом (но только табуляциями), причем такие отступы будут автоматически убраны при выводе, а при отсутствии кавычек в документе можно использовать переменные. Как было показано в примере 8.3, встроенные документы отлично подходят для отображения справочной информации, включающей имя сценария и путь к нему, избавляя от необходимости использования большого числа строк `echo` или `printf` и кавычек. Мы также применим их в разделе «Встроенная документация» в главе 10.

Синтаксис встроенных строк `[fd]<<<word` очень похож на синтаксис встроенных документов, но в слове `word` возможны расширения параметров и переменных, подстановка команд и арифметические операции. По сути, это более простой встроенный документ без разделителя с символом перевода строки в конце.

Дополнительная информация о встроенных документах и строках представлена в документации GNU:

- <https://oreil.ly/CsUWq>;
- <https://oreil.ly/ujM8B>.

## Код выполняется в интерактивном режиме?

Иногда необходимо знать, выполняется ли код в интерактивном режиме. Это может потребоваться, например, для изменения поведения при запросе ввода или для установки определенных конфигурационных параметров интерактивной оболочки.

В документации GNU по интерактивным оболочкам<sup>1</sup> рекомендуются следующие способы:

```
case "$-" in
  *i*) echo This shell is interactive ;;
  *) echo This shell is not interactive ;;
esac
```

ИЛИ:

```
if [ -z "$PS1" ]; then
  echo This shell is not interactive
else
  echo This shell is interactive
fi
```

Также можно использовать проверку `-t FD`, которая возвращает истинное значение, если дескриптор файла открыт в терминале. Если дескриптор файла не указан, то возвращается 0 (STDIN). Эту проверку можно объединить с другими тестами, например bash как оболочки:

```
# Только если bash выполняется в терминале!
if [ -t 1 -a -n "$BASH_VERSION" ]; then
  echo 'bash in a terminal'
else
  echo 'NOT bash in a terminal'
fi
```

---

<sup>1</sup> <https://oreil.ly/kbZE2>.

или:

```
# Только для интерактивного режима bash в терминале!
if [ -t 1 -a -n "$PS1" -a -n "$BASH_VERSION" ]; then
    echo 'Interactive bash in a terminal'
else
    echo 'NOT interactive bash in a terminal'
fi
```

Обратите внимание, что мы намеренно использовали [] (`test`) вместо [[]] (проверка условия в bash), чтобы этот код мог выполняться интерпретаторами, не поддерживающими [[]] (например, dash).

## В заключение

Идиомы, рассмотренные в этой главе, не всегда красивы, но их нужно знать. В bash поддерживается множество способов работы с файлами, часто с использованием перенаправления и/или конвейеров, поэтому мы показали только наиболее типичные шаблоны. Сценарии bash — это тоже файлы, поэтому мы обсудили волшебное заклинание, превращающее файл в выполняемый сценарий или библиотеку, способы завершения сценария и выполнение заключительных операций, а также другие полезные идиомы. Встроенные документы (`here-documents`) не являются полноценными файлами, но они часто заменяют отдельные текстовые файлы, поэтому также были рассмотрены в этой главе.

## ГЛАВА 10

---

# Помимо идиом: работа с bash

Программному коду свойственно развиваться. Сложность кода тоже имеет тенденцию к росту. Но нет ничего более постоянного, чем временное! Помните это, когда пишете код, еще только собираетесь его написать или говорите: «Готов поспорить, что я могу это автоматизировать»<sup>1</sup>. Планируйте заранее, будьте гибким, стремитесь к простоте и пишите код, который будет понятен и вам, и другим.

Хорошо, но... как?

В этой главе мы обсудим темы, которые не подошли ни к одному другому разделу. Здесь можно будет отдохнуть от обсуждения идиом, но все, о чем мы скажем, касается повседневной работы в командной строке bash.

Даже если вы пишете что-то «одноразовое», не поленитесь придумать хорошие говорящие имена и добавить комментарии. Закончив, потратьте еще несколько минут, чтобы прикинуть, как среда и код могут развиваться и какие изменения или исключения могут потребоваться. На начальном этапе разработки, пока все детали еще не забыты, намного проще увеличить гибкость кода и добавить новые возможности. Постарайтесь оценить, действительно ли этот код универсален (вряд ли) или могут обнаружиться исключения? Сможет ли оператор `case` обрабатывать некоторые настройки, основанные на `$HOSTNAME`, или придется спросить пользователя? Нужны ли сценарию аргументы или временные файлы?

---

<sup>1</sup> Рекомендуем комикс Automation (<https://xkcd.com/1319>).

Может быть, ему нужно обратиться к `$RANDOM`? Далее мы рассмотрим эти и другие вопросы.

Жизнь современных людей весьма насыщена, а работа — тем более. Как упростить ее? Можно ли структурировать репозиторий с исходным кодом и привести его в соответствие с целями развертывания? Как написать новые команды или сценарии для сохранения параметров и других данных? Возможно ли автоматизировать рутинную задачу? Надеемся, что в предыдущих главах вы получили достаточно инструментов для решения перечисленных задач. Однако у нас есть еще несколько идей, как повысить гибкость при работе и качество жизни. Обсудим их, прежде чем закончить эту книгу.

## Приглашения к вводу

На самом деле в работе с приглашениями к вводу команд (`prompt`) нет идиом. Приглашение по умолчанию зависит от интерпретатора `bash` и настроек, определенных в конфигурационных файлах. Но есть кое-что, о чем вы должны знать.

Мы не будем вдаваться в подробности управления приглашениями к вводу, так как они описаны в документации `bash` и множестве статей, находящихся в свободном доступе, например:

- [https://www.gnu.org/software/bash/manual/html\\_node/Controlling-the-Prompt.html](https://www.gnu.org/software/bash/manual/html_node/Controlling-the-Prompt.html);
- <http://www.bashcookbook.com/bashinfo/source/bash-4.2/examples/scripts.noah/prompt.bash>;
- <https://www.tldp.org/HOWTO/Bash-Prompt-HOWTO/index.html>;
- <https://sourceforge.net/projects/bashish>.

Давайте коротко рассмотрим различные виды приглашений к вводу, уделив основное внимание новым и малоизвестным возможностям (табл. 10.1).

**Таблица 10.1.** Приглашения к вводу в bash

Приглашение	Использование	По умолчанию
PS0	Отображение информации перед выполнением команды в bash версий 4.4 и выше	Нет
PS1	Основное приглашение к вводу в Bourne/bash	'\s-\v\\$ '
PS2	Дополнительное приглашение к вводу в Bourne/bash	
PS3	Приглашение к вводу для оператора select	'#? '
PS4	Отладочный параметр bash	'+ '
PROMPT_COMMAND	Команда перед \$PS1	Нет

**PS0**

Отображение информации после нажатия клавиши Enter в командной строке, но до запуска команды. Может пригодиться, например, для вывода времени начала выполнения команды (пример 10.1).

**Пример 10.1.** Вывод времени перед выполнением команды в bash

```
PS0='Start: \D{\%Y-\%m-\%d \%H:\%M:\%S_%Z}\n'

### Обратите внимание на разницу во времени в первом приглашении к вводу
### и в строке, которая выводится перед началом выполнения команды
[user@hostname:T2:L1:C5289:J0:2021-08-27_17:46:04_EDT]
/home/user/bash-idioms$ ls -1 ch*
Start: 2021-08-27_18:47:37_EDT
ch01.asciidoc
...
```

**PS1**

Основное приглашение к вводу в bash, унаследованное от командной оболочки Bourne. Именно с этим приглашением чаще всего приходится иметь дело в интерактивном сеансе bash, поэтому его надо уметь настраивать. Приглашение, показанное в примере 10.2, кого-то может озадачить, но оно содержит полную информацию, которую удобно копировать и вставлять в документацию или отчеты об ошибках. Обратите внимание, что дата и время соответствуют

моменту отображения приглашения, с которого могли пройти несколько часов или дней. Этую проблему можно обойти, просто нажимая `Enter` перед запуском новой команды или сеанса. А если у вас установлена версия `bash v4.4+`, то можете использовать `PS0`, как показано в примере 10.1. Также обратите внимание на номер `C5275` в истории команд. Если по какой-то причине в истории команд `bash` отобразится пароль, то просто выполните команду `history -d 5275` (см. `help history` и дополнительные указания, как предотвращать добавление команд в историю с помощью начального пробела, — это показано в примере 10.13).

### **Пример 10.2.** Пример основного приглашения в `bash`

```
export PS1='`n[\u@\h:T\l:$SHLVL:C\!:J\j:\D{\%Y-%m-%d_%H:%M:%S_%Z}`]\n$PWD\$ '  
[user@hostname:T2:L1:C5275:J0:2021-08-27_16:51:20_EDT]  
/home/user/bash-idioms$
```

**PS2**

Приглашение появляется после нажатия клавиши `Enter` при вводе текста в кавычках или встроенного документа.

**PS3**

Приглашение, используемое встроенной командой `select`, см. одинаковый подраздел в этой главе.

**PS4**

Отладочный префикс, который выводится, когда активна настройка `set -x`. Обратите внимание, что первый символ дублируется по мере необходимости, чтобы показать уровень вложенности оболочки, поэтому для большей ясности лучше использовать символ `+`, см. также раздел «Отладка в `bash`» в этой главе. Префикс в примере 10.3 очень информативен.

### **Пример 10.3.** Пример отладочного префикса

```
export PS4='+xtrace $BASH_SOURCE:$LINENO:$FUNCNAME: '
```

### PROMPT\_COMMAND

В примере 10.4 показана команда, которая будет выполнена непосредственно перед отображением \$PS1. Эта возможность используется для самых разных вещей, таких как обновление заголовка окна для терминалов с графическим интерфейсом, отображение динамических сведений о вашей среде (например, в какой ветке Git вы находитесь) или даже для очень примитивной и небезопасной регистрации.

#### **Пример 10.4.** Пример настройки PROMPT\_COMMAND для журналирования

```
### Этот код должен набираться в одну строку; мы разбили его,  
### только чтобы уместить по ширине книжной страницы  
export PROMPT_COMMAND='logger -p local1.notice -t "bashlog[$$];" \  
"SSH=$SSH_CONNECTION; USER=$USER; PATH=$PWD; COMMAND=$(fc -ln -1)"'
```

## Часовой пояс в приглашении

На самом деле это не идиома bash, а лишь настройка приглашения. По умолчанию приглашение в bash отображает дату и время в системном часовом поясе, но вы можете выполнить команду `export TZ=UTC` и указать часовой пояс по своему выбору. Это удобно, если требуется, чтобы время в графическом интерфейсе отображалось в местном часовом поясе, а приглашения в bash сообщали бы время UTC для документирования, или чтобы отображение времени в bash на локальном компьютере соответствовало серверному времени. Обратите внимание, что приглашение `TZ` не изменится, пока вы не выполните внешнюю команду: простого нажатия `Enter` или запуска встроенной команды `bash` недостаточно.

Чтобы узнать больше об обработке даты и времени, см. также раздел «Функция `printf`» в главе 6.

# Получение ввода пользователя

В большинстве случаев сценарии bash получают ввод пользователя через аргументы командной строки, но иногда требуется запросить ввод

прямо из сценария. Для этого можно использовать встроенные команды `read` и `select`.

Дополнительная информация:

- `help read;`
- `help select.`

## read

`read` читает строку из STDIN или файлового дескриптора, записывает слова в список переменных или в переменную-массив (список) и имеет несколько полезных параметров (пример 10.5).

### Пример 10.5. Справка для команды `read` в bash

```
$ bash --version
GNU bash, version 4.4.20(1)-release (x86_64-pc-linux-gnu)
...
$ help read
read: read [-ers] [-a имя_массива] [-d разделитель] [-i текст] [-n количество]
      [-р приглашение] [-t тайм-аут] [-u fd] [имя ...]
      Читает строку из стандартного ввода и разбивает ее на поля.
```

Читает одну строку из стандартного ввода или из дескриптора `fd`, если задан ключ `-u`. Стока разбивается на поля по границам слов, первое слово присваивается первому имени, второе слово -- второму, и т. д., последнему имени присваиваются все остальные слова

Если имена не заданы, то прочитанная строка сохраняется в переменной `REPLY`

Параметры:

<code>-a имя_массива</code>	слова присваиваются последовательным элементам указанного массива, начиная с 0
<code>-d разделитель</code>	чтение продолжается, пока не будет встречен первый символ в строке-разделителе, а не символ перевода строки
<code>-e</code>	использовать библиотеку Readline для получения строки в интерактивной оболочке
<code>-i текст</code>	использовать указанный текст как начальный для Readline
<code>-n количество</code>	завершить чтение после получения указанного количества символов, не дожидаясь, когда встретится перевод строки, но учитывать символ-разделитель, если перед ним было прочитано символов меньше, чем задано
<code>-р приглашение</code>	выводить указанную строку приглашения без завершающего

-r	символа перевода строки перед попыткой чтения
-s	не интерпретировать обратные слеши как символ экранирования
-t тайм-аут	отключить эхо-вывод символов, получаемых в терминале если за указанное время в потоке ввода строка появилась не вся, то операция чтения завершается с ошибкой. По умолчанию величину тайм-аута определяет переменная TMOOUT. Аргумент 'тайм-аут' может быть дробным числом. Если передать число 0, то read завершится немедленно без чтения данных. Успешное завершение возможно, только если входная строка доступна в файловом дескрипторе fd. При завершении по тайм-ауту возвращает значение больше 128
-u fd	чтение будет выполняться из файлового дескриптора fd

Код выхода:

Возвращает нуль, если не встретился конец файла, не истекло время тайм-аута (в этом случае возвращается число больше 128), не произошла ошибка присваивания переменной и был указан верный дескриптор файла при вызове с ключом -u.

Из перечисленных ключей очень полезны -eip: -p выводит приглашение к вводу; -i поставляет значение по умолчанию; -e позволяет использовать возможности библиотеки *Readline* (например, навигацию по строке клавишами со стрелками, см. пример 10.13) для редактирования значения по умолчанию перед его подтверждением (пример 10.6).

#### **Пример 10.6.** Использование редактируемого значения по умолчанию

```
$ read -ei 'default value' -p 'Enter something: '
Enter something: default value
### Для правки значения по умолчанию можно использовать возможности
### библиотеки Readline, такие как навигация по строке клавишами со стрелками
Enter something: changed: default value

$ echo $REPLY
changed: default value
```

Параметр -s позволяет запрашивать пароли без отображения вводимых символов на экране (пример 10.7).

#### **Пример 10.7.** Запрос ввода пароля

```
$ read -sp 'Enter the secret password: ' ; \
echo -e "\n\nShhhh, the password is: ~$REPLY~"
Enter the secret password:
```

Shhhh, the password is: ~super secret~

В сценариях, написанных на скорую руку, можно использовать нечто подобное:

```
read -n1 -p 'CTRL-C to abort or any other key to continue...'
```

но помните, что нет ничего более постоянного, чем «временное».

Поддержка тайм-аута отлично подходит для запроса ввода дополнительных данных, позволяя не «подвесить» сценарий, если пользователю нечего ввести (пример 10.8).

#### **Пример 10.8.** Запрос ввода с тайм-аутом

```
$ time read -t 4 -p "Are you there?"  
Are you there?  
real    0m4.000s  
user    0m0.000s  
sys     0m0.000s
```

`read` также можно использовать для чтения файлов и анализа ввода, как рассказывалось в начале главы 9.

## **pause**

Возможно, кто-то еще помнит старую команду `pause` в DOS/Windows? В bash ее можно сымитировать следующим образом:

```
$ read -n1 -p 'Press any key to continue...'  
Press any key to continue...
```

## **select**

Как следует из названия, команда `select` создает простое меню и позволяет сделать выбор. При необходимости вы можете добавить в меню пункты для выхода или отмены. Похожий инструмент со значительно большими возможностями — `dialog`, но это не идиома bash, поэтому мы не будем его рассматривать.

Вот простой пример:

```

#!/usr/bin/env bash
# select-ssh.sh: создание меню из ~/.ssh/config для выбора целевого
# хоста ssh_target и подключения к нему
# Автор и дата: _bash Idioms_ 2022
# Имя файла в bash Idioms: examples/ch10/select-ssh.sh
#
#ssh_config="$HOME/.ssh/config"
# Заменить завершающий фрагмент 'select-ssh.sh' на 'ssh_config'
ssh_config="${{0%/*}}/ssh_config" # Идиома проверки файла

PS3='SSH to> '
select ssh_target in Exit \
$(egrep -i '^Host \w+' "$ssh_config" | cut -d' ' -f2-); do
    case $REPLY in
        1|q|x|quit|exit) exit 0 ;;
        *) break ;;
    esac
done

# Выводится выбранный пункт меню
echo ssh $ssh_target

```

Этот сценарий выведет следующее:

```

$ examples/ch10/select-ssh.sh
1) Exit                  3) gitlab.com          5) mythtv-be01
2) git.atlas.oreilly.com 4) github.com         6) kitchen
SSH to> 1

$ examples/ch10/select-ssh.sh
1) Exit                  3) gitlab.com          5) mythtv-be01
2) git.atlas.oreilly.com 4) github.com         6) kitchen
SSH to> 6
ssh kitchen

```

## Псевдонимы

Коль скоро вы читаете эту книгу, вполне вероятно, что вы уже знакомы с командами `alias` и `unalias`. Тем не менее нам хотелось бы осветить несколько важных моментов.

Во-первых, из соображений безопасности для удаления псевдонима следует использовать команду `\unalias`. Обратный слеш (`\`) запрещает расширение псевдонима, исключая возможность применения злоумышленниками псевдонима `unalias`.

Во-вторых, некоторые дистрибутивы Linux любят настраивать «полезные» псевдонимы для `root` и других пользователей, поэтому рекомендуем изучить значения по умолчанию в вашем дистрибутиве. В частности, вы увидите псевдоним `rm=rm -i` и аналогичные ему псевдонимы для `cp` и `mv`.

В примерах 10.9 и 10.10 показаны два набора псевдонимов, которые можно включить в свой файл `~/.bashrc`.

#### **Пример 10.9.** Обеспечение простейшей «совместимости» с DOS/Windows

```
alias cls='clear'          # Аналог команды clear в DOS
alias copy='cp'            # Аналог команды cp в DOS
alias del='rm'             # Аналог команды rm в DOS
alias dir='ls'             # Аналог команды ls в DOS
alias ipconfig='ifconfig'  # Аналог команды ifconfig в DOS
alias md='mkdir'           # Аналог команды mkdir в DOS
alias move='mv'             # Аналог команды mv в DOS
alias rd='rmdir'           # Аналог команды rmdir в DOS
alias ren='mv'              # Аналог команды mv/rename в DOS
alias tracert='traceroute' # Аналог команды traceroute в DOS
```

#### **Пример 10.10.** Примеры псевдонимов

```
# Установите `xclip` или `xsel` для выполнения копирования и вставки в Linux
alias gc='xsel -b' # "GetClip" извлекает содержимое буфера обмена
alias pc='xsel -bi' # "PutClip" помещает содержимое в буфер обмена
# Для Mac: pbcopy/pbpaste
# Для Windows: gclip.exe/pclip.exe или getclip.exe/putclip.exe

# `df` с сокращенным объемом выводимой информации
alias df='df --print-type --exclude-type=tmpfs --exclude-type=devtmpfs'
alias diff='diff -u'          # Унификация различий по умолчанию
alias locate='locate -i'      # Поиск без учета регистра символов
alias ping='ping -c4'         # По умолчанию выполнять только четыре попытки
alias vzip='unzip -lvM'        # Просмотр содержимого ZIP-файлов
alias lst='ls -lrt | tail -5' # Показать пять последних измененных файлов

# Немного измениенная версия с сайта
# https://oreil.ly/1SUg7
```

```
alias randomwords="shuf -n102 /usr/share/dict/words | perl -ne 'print qq(\u\$_);' \
| column"
```

Иногда полезно, чтобы одна и та же команда работала по-разному на разных хостах. Это легко реализовать с помощью оператора `case` (пример 10.11).

#### **Пример 10.11.** Разные псевдонимы для разных хостов

```
case "$HOSTNAME" in
    host1* ) # Linux, если установлен пакет `xclip`
        alias gc='xclip -out' # Отправить выделенное в STDOUT
        alias pc='xclip -in' # Вставить из STDIN на место выделенного
    ;;
    host2* ) # Mac
        alias gc='pbpaste' # Отправить данные для вставки в STDOUT
        alias pc='pbcopy' # Отправить данные STDIN в буфер обмена для вставки
    ;;
    * ) # По умолчанию Linux
        alias gc='xsel -b' # Отправить буфер обмена в STDOUT
        alias pc='xsel -bi' # Отправить STDIN в буфер обмена
    ;;
esac
```

## ФУНКЦИИ

Если в псевдонимы требуется передать аргументы, то надо прибегнуть к помощи функций, которые мы рассмотрели в главе 6. Здесь мы показываем не идиомы, а рецепты, но все же продемонстрируем маленькую и удобную функцию (пример 10.12).

#### **Пример 10.12.** Функция `mcd`

```
function mcd {
    \mkdir -p "\$1"
    \cd "\$1"
}
```

Да, она простая, не выполняет проверок и не позволяет задавать разрешения (-`m`), но даже в таком виде помогает сэкономить массу времени,

когда требуется создать множество каталогов. Обратите внимание на начальный обратный слеш (\), запрещающий расширение псевдонимов. Такое решение гарантирует вызов только встроенных команд bash. Доработку этой функции мы оставляем вам в качестве самостоятельного упражнения.

## Локальные переменные

Мы уже затрагивали эту тему в главе 6, но хотели бы напомнить, что команду `local` можно использовать только внутри функций для объявления переменных, которые будут локальными для этих функций. Это распространенная практика в программировании, но если вам нужна «локальная» область видимости вне функции, то такая возможность имеется. Как пример рассмотрим переменную `$IFS`, хранящую символы — разделители полей и используемую командной оболочкой для разделения строк на слова. Обычно изменение `$IFS` нежелательно, но иногда оно может быть целесообразным для определенного фрагмента кода (вне функции). В этом случае просто добавьте присваивание нового значения переменной перед командой, для которой требуется это изменение. Такой командой может быть `read`, о которой мы поговорим ниже:

```
IFS=':' read ...
```

См. также раздел «Изменяем `$IFS` при чтении файлов» в главе 9.

## Возможности Readline

*Readline* — библиотека для чтения и редактирования командной строки. Доступ к ней открывает команда `read -e`. Команда `read` поддерживает множество ключей и параметров для настройки *Readline*.

Вы можете попробовать создать файл `~/.inputrc` (см. описание команды `bind -f`), но для простых экспериментов достаточно добавить связи в файл `~/.bashrc` (пример 10.13).

**Пример 10.13.** Настройки Readline

```

bind "'\e[A': history-search-backward' # Обратный поиск в командной строке ①
bind "'\e[B": history-search-forward' # Прямой поиск в командной строке ②
bind "'\C-i": menu-complete'          # Перебор возможных завершений ③
bind 'set completion-ignore-case on' # Игнорировать регистр символов ④
                                         # в завершениях

# Команды bash, связанные с настройками интерфейса командной строки
export HISTCONTROL='erasedups:ignoredups:ignorespace'+ ⑤
export HISTIGNORE='&:[ ]*' # bash v3+, игнорировать дубликаты и строки, ⑥
                           # начинающиеся с пробела

```

Разберем код:

**①** Связывает «стрелку вверх» с командой `history-search-backward` поиска в истории от конца к началу, т. е., если вы начнете вводить команду, а затем нажмете «стрелку вверх», bash попробует отыскать эту строку в истории. Примерно ту же функцию выполняет комбинация `CTRL-R`, но «стрелка вверх» быстрее и понятнее. Однако комбинация `CTRL-R` способна искать строки в середине команды, тогда как `history-search-backward` ограничена началом строки.

**②** Связывает «стрелку вниз» с командой `history-search-forward` поиска в истории от начала к концу.

**③** При нескольких вариантах завершения вместо подачи звукового сигнала можно заставить оболочку после нажатия клавиши `TAB` перебирать эти варианты.

**④** Игнорировать регистр символов в именах файлов и команд при завершениях.

**⑤** При сохранении истории `erasedups` удалит все предыдущие строки, совпадающие с текущей строкой; `ignoredups` предотвратит сохранение повторяющихся строк; `ignorespace` предотвратит сохранение команд, начинающихся с пробела. Благодаря этим настройкам ваша история команд будет короче, но из-за удаления повторяющихся команд исказится последовательность. Добавляя пробел перед командой, можно предотвратить сохранение в истории команд, например, содержащих

конфиденциальные данные, но учтите, что они все равно будут видны в списке процессов.

❶ То же, что и в п. ❸, но совместимое со старыми версиями bash.

## Журналирование в bash

Для вывода записей в системный журнал отлично подходит инструмент `logger` (`man logger`), но он имеет ошибку: аргумент `-t`, или `--tag`, не является обязательным и по умолчанию принимает имя пользователя. Мы рекомендуем использовать этот инструмент, как показано в примере 10.14.

### Пример 10.14. Теги в вызове `logger`

```
logger --tag "$0[$$]" <msg> ❶
logger --tag "$PROGRAM[$$]" <msg> ❷
logger --tag "${0##*/}[$$]" <msg> ❸
```

Комментарии:

❶ В простейшем случае используются две переменные: `$0` — имя сценария; `$$` — идентификатор процесса. Проблема с `$0` состоит в том, что ее содержимое зависит от способа вызова сценария. Мы рекомендуем использовать команду `basename`.

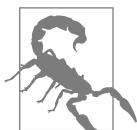
❷ Если у вас уже есть переменная для хранения имени файла сценария, например `$PROGRAM` (см. пример 8.3), используйте ее.

❸ Если у вас нет переменной для хранения имени сценария, получить его можно прямо в команде `logger`. Этот код выглядит не очень ясным, но работает. Попробуйте `echo "${0##*/}[$$]"`, чтобы оценить результат.

Если вы предполагаете создать свой механизм журналирования, то ознакомьтесь с описанием команды `printf` и приемов работы с датой и временем в разделе «Получение и использование даты и времени» в главе 6.

## Обработка JSON с помощью jq

jq — это не встроенная команда bash, но мы расскажем о ней, потому что JSON — распространенный формат, особенно в облачных системах. Нам приходилось видеть код для обработки данных в формате JSON, использующий такие инструменты, как `awk`, `grep` и `sed`. Он выглядел ужасно. По возможности старайтесь использовать jq.



### Команда jq доступна не везде

К сожалению, несмотря на широкое распространение формата JSON, не все дистрибутивы устанавливают команду jq по умолчанию. Она отсутствует и в репозитории Git Bash для Windows, включающем `awk`, `sed`, `cut`, `grep`, `tr` и т. д.

Иногда эту проблему можно обойти. Например, в AWS можно использовать `--output=text` и проанализировать вывод с помощью обычных инструментов. Также можно установить jq в свои сборки или найти обходные пути с использованием Python и других инструментов, обсуждение которых, впрочем, выходит за рамки этой книги.

В число файлов с исходным кодом для этой книги входил `atlas.json`, который мы будем использовать для демонстрации некоторых приемов. Вот как представлено название этой книги:

```
$ jq '.title' atlas.json
"bash Idioms"
$ jq -r '.title' atlas.json # Режим без интерполяции (вывод без кавычек)
bash Idioms

### Также можно использовать `grep`
$ grep 'title' atlas.json
  "titlepage.html",
  "title": "bash Idioms"
```

А что, если мы хотим знать, включена ли подсветка синтаксиса в выходных форматах? С помощью grep это сделать сложно, а с jq — легко:

```
### Какие форматы поддерживаются?  
$ jq '.formats | keys' atlas.json  
[  
    "epub",  
    "html",  
    "mobi",  
    "pdf"  
]  
  
### Этот вариант намного хуже; он не дает контекста и потому бесполезен  
$ grep 'syntaxhighlighting' atlas.json  
    "syntaxhighlighting": true,  
    "syntaxhighlighting": true,  
    "syntaxhighlighting": true,  
    "syntaxhighlighting": true,  
  
### Хороший вариант, сообщающий, что вывод в PDF поддерживает  
### подсветку синтаксиса  
$ jq '.formats.pdf.syntaxhighlighting' atlas.json  
true  
  
### Лучший вариант, но команда сложна для понимания,  
### она выводит форматы и признак поддержки подсветки синтаксиса,  
### см. документацию к jq  
$ jq -r '.formats | keys[] as $k | "\($k), \(.[$k] \\"  
    .syntaxhighlighting)"' atlas.json  
epub, true  
html, true  
mobi, true  
pdf, true
```



### Просмотр JSON в Firefox

В Firefox встроен *потрясающий* инструмент просмотра JSON, который помогает понять, что и как получить с помощью `jq`. Просто сохраните документ в формате JSON в файле `myfile.json`, а затем откройте его в Firefox как `file:///path/to/myfile.json`.

## Поиск в списке процессов

При поиске в списке процессов с помощью `grep` одна из возвращаемых записей — это ваша команда `grep`, то есть совсем не то, что вы ищете.

В примере 10.15 показана идиома, помогающая удалить строку с grep из вывода.

**Пример 10.15.** Удаление строки с grep: плохой вариант

```
ps auwx | grep 'proggy' | grep -v 'grep'
```

Мы считаем это решение некрасивым и неэффективным. Пример 10.16 нравится нам больше.

**Пример 10.16.** Удаление строки с grep: рекомендуемый вариант

```
ps auwx | grep '[p]roggy'
```

Этот прием работает потому, что *строка* [p]roggy в списке процессов не соответствует регулярному выражению /[p]roggy/, которое в действительности означает proggy. Иначе говоря, [p]roggy != proggy.

Также можно использовать pgrep -f1, или в крайнем случае старую команду pidof.

## Ротация старых файлов

Журналирование и архивирование журналов — хорошая и правильная практика, но в какой-то момент возникает необходимость «ротации», или удаления, старых файлов. Существует множество идиоматических способов сделать это, поэтому мы расскажем лишь о некоторых из них.

Многие подходы к ротации файлов основаны на применении эффективной утилиты find, но есть и использующие параметры -exec и/или xargs, которые в современных версиях find не требуются. Вот старые идиомы:

```
find /path/to/files -name 'some-pattern' -a -mtime 5 -exec rm -f \{\}\; ①  
find /path/to/files -name 'some-pattern' -a -mtime 5 | xargs rm -f ②  
find /path/to/files -name 'some-pattern' -a -mtime 5 -print0 | xargs -0 rm -f ③
```

Комментарии:

- ❶ Неэффективно, потому что для каждого файла вызывается новый экземпляр `rm`.
- ❷ Более эффективный вариант, так как файлы группируются в пакеты (`xargs`), но возможны ошибки, если имена файлов содержат пробелы.
- ❸ Этот вариант способен обрабатывать пробелы в именах файлов, но выглядит довольно сложным.

Если у вас установлена утилита GNU `find` и нет проблем с переносимостью, то рекомендуем использовать код из примера 10.17.

#### Пример 10.17. Удаление старых файлов с помощью `find`

```
find /path/to/files \( -type f -a -name 'some-pattern' -a -mtime 5 \) -delete
```

Для работы с `find` следует знать следующее:

- В `man find` есть описание параметров `atime` и `mtime`. Например, `-mtime +5` означает, что последний раз файл изменялся не менее пяти дней назад. Подробности смотрите в справочном руководстве.
- Для тестирования используйте `-ls` или `-print` вместо `-delete!` Но не забудьте вернуть изменения, когда закончите.
- `find` рекурсивно обрабатывает все подкаталоги пути, если только не ограничить глубину поиска с помощью `-maxdepth`.
- Используйте `( )` для группировки (обрабатывать только *те* элементы, которые соответствуют *всем* критериям), но скобки нужно экранировать, чтобы избежать интерпретации командной оболочки.
- Предполагается, что между параметрами в выражении используется оператор `-a` (and — и); однако при необходимости можно также использовать `-o` (or — или). Мы предпочитаем добавлять в код `-a`, потому что явное лучше неявного.

- Параметр `-type f` означает «тип — обычный файл», поэтому каталоги, ярлыки и др. будут игнорироваться.
- Параметр `-name` не требует объяснения, но имейте в виду, что он чувствителен к регистру символов. Чтобы поиск выполнялся без учета регистра, используйте `-iname`.

## Встроенная документация

Программисты имеют репутацию людей, не любящих писать или обновлять документацию. Иногда это соответствует действительности, но встречаются и такие программисты, как мы. Очевидно, что чем больше сложностей возникает при написании или обновлении документации, тем ниже вероятность ее наличия. Способы решения этой проблемы могут различаться, но одна из возможностей — встроить документацию в код. В разных языках используются различные способы встраивания документации, и bash также имеет свои особенности.

Вы можете добавлять комментарии в код, и мы показали несколько способов использования комментариев как части справки (см. раздел «HELP!» в главе 8). Документацию можно разместить, например, в файле `myscript.asciidoc` рядом с файлом сценария `myscript.sh`. Но также можно встроить полноценную документацию на языке разметки прямо в файл с кодом, как показано в примере 10.18.

Обратите внимание, что в этом примере использован как язык Perl POD (Plain Old Documentation — простая старая документация), так и оригинальная разметка bash, но вообще желательно использовать какой-то один стиль. Формат POD кажется архаичным по сравнению с Markdown, Textile и др., но для него имеются хорошие инструменты преобразования, особенно для `man`-страниц. Поэтому POD все еще целесообразно применять, особенно если вы используете Perl. Еще один полезный инструмент — конвертер разметки Pandoc, см., например: `pandoc README.md > /tmp/README.html && firefox file:///tmp/README.html`.

Обязательно прочтайте код ниже или его вывод, потому что они содержат дополнительные советы. Обратите внимание, что сноски выглядят в исходном коде необычно и приобретают смысл только в выводе. Просто продолжайте читать:

**Пример 10.18.** Пример встроенной документации в bash: код

```
#!/usr/bin/env bash
# embedded-docs.sh: пример кода на bash с разными видами встроенной документации
# Автор и дата: _bash Idioms_ 2022
# Имя файла в bash Idioms: examples/ch10/embedded-docs.sh
#
# _____
# Не работает в Zsh 5.4.2!

[[ "$1" == '--emit-docs' ]] && {
    # Оператор диапазона в Perl выведет только строки МЕЖДУ маркерами
    # DOCS во встроенных документах, за исключением случаев, когда
    # маркеры встречаются в обсуждении этого кода в самой документации.
    # См. вывод, чтобы понять, о чем речь.
    perl -ne "s/^t+//; print if m/DOCS'?\$/ .. m/^s*?DOCS'?\$/ \n
        and not m/DOCS'?\$/;" $0
    exit 0
}

echo 'Code, code, code... <2>'
echo 'Code, code, code...'
: << 'DOCS'
==== Документация для моего сценария ⑦
```

Не обращайте внимания на сноски в этом заголовке, она имеет смысл только в выводе.

Документы могут оформляться на языках разметки Markdown, AsciiDoc, Textile и т. д. Этот блок оформлен с применением универсальной разметки.

Мы создали его с помощью оператора ':' (не выполняющего никаких операций) и встроенного документа, но вы должны помнить, что этот встроенный документ обрабатывается не на bash. Поэтому использование <<-DOC, чтобы получить возможность оформления отступов, не даст желаемого эффекта. Кавычки вокруг маркера запрещают интерполяцию переменных, но, даже если бы она была разрешена, это не повлияло бы на вывод этой документации. Всегда заключайте в кавычки маркер встроенного документа, чтобы ваши документы не мешали вашему сценарию (например, используйте обратные кавычки, как мы ниже).

Всю документацию можно сгруппировать в начале файла для удобства поиска или, наоборот, в конце, чтобы она не мешала читать код. Также код и документация могут чередоваться, чтобы описание находилось рядом с соответствующим кодом. Поступайте так, как удобнее вам и вашей команде.

```
DOCS
echo 'More code, code, code... <3>'
echo 'More code, code, code...'
: << 'DOCS'
=head1 Пример POD ⑧
```

Не обращайте внимания на сноска в этом заголовке, она имеет смысл только в выводе.

Этот блок оформлен в формате Perl POD.

Если вы решите оформить документацию в формате POD, то у вас появится возможность использовать для ее обработки `perldoc` и различные инструменты `pod2\*`, такие как `pod2html`. Но вы не сможете делать отступы, иначе Perl не увидит разметку, если только не обработать эти отступы перед передачей документации инструментам.

И не забудьте строку `=cut`!

```
=cut
```

```
DOCS
echo 'Still more code, code, code... <4>'
echo 'Still more code, code, code...'
: << 'DOCS'
    Вывод документации ⑨
-----
```

Не обращайте внимания на сноска в этом заголовке; она имеет смысл только в выводе.

Этот блок мог бы быть оформлен в формате POD или с применением любой другой разметки.

В этом разделе отступы в тексте оформлены с помощью табуляций, но и этот блок обрабатывается с помощью Perl, а не bash. :-/

для вывода документации вы должны добавить "обработчик" в свои параметры обработки/помощи. Для вывода формата POD используйте соответствующие инструменты, но убедитесь, что они установлены!  
Вывод других видов разметки вы должны обеспечить самостоятельно.

Мы знаем, что эта книга о bash, но следующая строка на Perl с регулярными выражениями и оператором диапазона действительно удобна:

```
perl -ne "s/^t//; print if m/DOCS'?\$/ .. m/^s*?DOCS'?\$/ \n
        and not m/DOCS'?\$/;" $0
```

Этот код выводит строки, соответствующие регулярному выражению `m/DOCS'?\$/`, и прекращает вывод, когда встречается соответствие с `m/^s*?DOCS'?\$/`.

Кроме того, он не выведет строку, содержащую `\$DOCS/`.

Используйте этот код, когда требуется вывод документации.

DOCS

```
echo 'End of code... <5>'

exit 0 # Если выход с кодом >0 не произошел выше

: << 'DOCS'
h2. Дополнительная документация ПОСЛЕ кода ⑩
```

Не обращайте внимания на сноски в этом заголовке, она имеет смысл только в выводе.

Этот блок снова оформлен с применением универсальной разметки. Мы не рекомендуем смешивать и сочетать разные стили оформления, как в этом примере! Выберите определенный стиль, несколько инструментов и придерживайтесь их. Если вы сомневаетесь в выборе, используйте разметку Markdown, очень популярную на GitHub.

Документация может размещаться после кода. Есть аргументы в пользу объединения всей документации в одном месте, в начале или конце сценария. Так же есть аргументы в пользу размещения документации ближе к соответствующему коду, особенно к функциям. Выбор за вами.

Но если раздел размещен ниже `exit 0` и не оформлен как встроенный документ, то некоторые редакторы с подсветкой синтаксиса будут сбиваться, а инструкция на Perl, реализующая вывод документации, пропустит его.

DOCS

В примере 10.19 показан вывод этого сценария.

### Пример 10.19. Пример встроенной документации в bash: вывод

```
### Вот как выглядит вывод сценария, если запустить его без аргументов
### (`./embedded-docs.sh`): ①
Code, code, code... ②
Code, code, code...
More code, code, code... ③
More code, code, code...
Still more code, code, code... ④
Still more code, code, code...
End of code... ⑤

### А так выглядит вывод документации из сценария
### (`./embedded-docs.sh --emit-docs`): ⑥
==== Документация для моего сценария ⑦
```

Не обращайте внимания на сноски в этом заголовке, она имеет смысл только в выводе.

Документы могут оформляться на языках разметки Markdown, AsciiDoc, Textile и т. д. Этот блок оформлен с применением универсальной разметки.

Мы создали его с помощью оператора `:' (не выполняющего никаких операций) и встроенного документа, но вы должны помнить, что этот встроенный документ обрабатывается не на bash. Поэтому использование <<-DOC, чтобы получить возможность оформления отступов, не даст желаемого эффекта. Кавычки вокруг маркера запрещают интерполяцию переменных, но, даже если бы она была разрешена, это не повлияло бы на вывод этой документации. Всегда заключайте в кавычки маркер встроенного документа, чтобы ваши документы не мешали вашему сценарию (например, используйте обратные кавычки, как мы ниже).

Всю документацию можно сгруппировать в начале файла для удобства поиска или, наоборот, в конце, чтобы она не мешала читать код. Также код и документация могут чередоваться, чтобы описание находилось рядом с соответствующим кодом. Поступайте так, как удобнее вам и вашей команде.

=head1 Пример POD ⑧

Не обращайте внимания на сноски в этом заголовке, она имеет смысл только в выводе.

Этот блок оформлен в формате Perl POD.

Если вы решите оформить документацию в формате POD, то у вас появится возможность использовать для ее обработки `perldoc` и различные инструменты `pod2\*`, такие как `pod2html`. Но вы не сможете делать отступы, иначе Perl не увидит разметку, если только не обработать эти отступы перед передачей документации инструментам.

И не забудьте строку `=cut`!

=cut

Вывод документации ⑨

---

Не обращайте внимания на сноски в этом заголовке; она имеет смысл только в выводе.

Этот блок мог бы быть оформлен в формате POD или с применением любой другой разметки.

В этом разделе отступы в тексте оформлены с помощью табуляций, но и этот блок обрабатывается с помощью Perl, а не bash. :-/

Для вывода документации вы должны добавить "обработчик" в свои параметры обработки/помощи. Для вывода формата POD используйте соответствующие инструменты, но убедитесь, что они установлены! Вывод других видов разметки вы должны обеспечить самостоятельно.

Мы знаем, что эта книга о bash, но следующая строка на Perl с регулярными выражениями и оператором диапазона действительно удобна:

```
perl -ne "s/^\t+//; print if m/DOCS'?\$/ .. m/^{\s*'?DOCS'?\$/ \n and not m/DOCS'?\$/;" $0
```

Этот код выводит строки, соответствующие регулярному выражению `m/DOCS'?\$/`, и прекращает вывод, когда встречается соответствие с `m/^{\s*'?DOCS'?\$/`. Кроме того, он не выведет строку, содержащую `m/DOCS/`.

Используйте этот код, когда требуется вывод документации.

## h2. Дополнительная документация ПОСЛЕ кода ①

Не обращайте внимания на сноски в этом заголовке, она имеет смысл только в выводе.

Этот блок снова оформлен с применением универсальной разметки. Мы не рекомендуем смешивать и сочетать разные стили оформления, как в этом примере! Выберите определенный стиль, несколько инструментов и придерживайтесь их. Если вы сомневаетесь в выборе, используйте разметку Markdown, очень популярную на GitHub.

Документация может размещаться после кода. Есть аргументы в пользу объединения всей документации в одном месте, в начале или конце сценария. Также есть аргументы и в пользу размещения документации ближе к соответствующему коду, особенно к функциям. Выбор за вами.

Но если раздел размещен ниже `exit 0` и не оформлен как встроенный документ, то некоторые редакторы с подсветкой синтаксиса будут сбиваться, а инструкция на Perl, реализующая вывод документации, пропустит его.

Необходимые пояснения:

- ① Если запустить сценарий как обычно, то он выполнит свои действия и завершится.
- ② Вывод блока кода, выполняемого в обычном случае. Обратите внимание, что встроенная документация, следующая за блоком, игнорируется.
- ③ Результат еще одного блока кода. Встроенная документация, предшествующая этому блоку и следующая за ним, игнорируется.
- ④ Догадаетесь сами?
- ⑤ Конец кода, предшествующая встроенная документация игнорируется, как было отмечено выше. В этой точке производится выход из сценария, поэтому следующая ниже документация технически не нуждается

в оформлении в виде встроенного документа, но все же лучше оформить ее так для согласованности и корректной работы механизмов подсветки синтаксиса в редакторах.

❶ Теперь можно вывести документацию! Для этого предусмотрен ключ `--emit-docs`, но вы можете использовать и другие инструменты для извлечения и обработки документов.

❷ Универсальная разметка. Вместо нее может использоваться Markdown, Textile и любая другая.

❸ Формат Perl POD, который обрабатывается с помощью различных инструментов pod2\*.

❹ Блок, который может быть приведен в формате POD или содержать универсальную разметку. Дело в том, что мы добавили отступ, который нужно учесть в инструменте обработки.

❺ Как уже отмечалось в п. ❸, этот блок расположен после конца кода.

Дополнительные источники информации:

- <https://en.wikipedia.org/wiki/Pandoc>;
- [https://en.wikipedia.org/wiki/Plain\\_Old\\_Documentation](https://en.wikipedia.org/wiki/Plain_Old_Documentation);
- <https://en.wikipedia.org/wiki/Markdown>;
- [https://en.wikipedia.org/wiki/Textile\\_\(markup\\_language\)](https://en.wikipedia.org/wiki/Textile_(markup_language));
- <https://en.wikipedia.org/wiki/Asciidoc>;
- [https://en.wikipedia.org/wiki/Comparison\\_of\\_document\\_markup\\_languages](https://en.wikipedia.org/wiki/Comparison_of_document_markup_languages);
- [https://en.wikipedia.org/wiki/Lightweight\\_markup\\_language](https://en.wikipedia.org/wiki/Lightweight_markup_language);

## Отладка в bash

Мы уже говорили о добавлении операторов отладки в разделе «Режимы отладочного и подробного вывода» в главе 8, о выводе значений для

отладки в разделе «printf для повторного использования или отладки» в главе 6 и в примере 10.3, но до сих пор мы не рассматривали отладку на уровне интерпретатора.

Прежде всего, можно выполнить полную проверку синтаксиса командой `bash -n /path/to/script`. Технически `bash -n` означает «читать команды, но не выполнять» (см. `help set`), что аналогично `perl -c`. В этом режиме интерпретатор не сообщит вам об ошибках времени выполнения, неправильной логике, неверных параметрах внешних команд и т. п. Однако он укажет на непарные кавычки и скобки, недопустимый синтаксис встроенных функций `bash` и т. д. Это отличный прием для регулярного применения при разработке или редактировании сценариев.

Команды `bash -v` и `set -v` выводят строки по мере их чтения оболочкой, но на практике они не так полезны, как кажется. Дело в том, что строки отображаются перед их интерпретацией, поэтому вы просто получите обратно свой исходный код.

`bash -x (xtrace)` полезна, поскольку выводит команды и их аргументы по мере выполнения, и это следующий шаг в отладке. Запустите сценарий командой `bash -x /path/to/script` и получите массу отладочной информации. В примере 10.3 показано, как сделать вывод более полезным. Трассировку `xtrace` можно включить и в середине сценария командой `set -x`, а затем выключить командой `set +x` (да, выглядит немного странно: включение с помощью `-` и выключение с помощью `+`, но, попрактиковавшись, вы запомните эту особенность). Пример 10.20 иллюстрирует процесс отладки.

#### Пример 10.20. Простой пример отладки

```
### Без отладки
$ examples/ch10/select-ssh.sh
1) Exit                      3) gitlab.com          5) mythtv-be01
2) git.atlas.oreilly.com    4) github.com          6) kitchen
SSH to> 1
```

```
### С отладкой (две одиночные строки были разделены,
### чтобы уместить код по ширине страницы)
$ bash -x examples/ch10/select-ssh.sh
+trace examples/ch10/select-ssh.sh:9:: ssh_config=examples/ch10/ssh_config
+trace examples/ch10/select-ssh.sh:11:: PS3='SSH to> '
+trace examples/ch10/select-ssh.sh:12:: select ssh_target in Exit $(egrep \
-i '^Host \w+' "$ssh_config" | cut -d' ' -f2-)
++trace examples/ch10/select-ssh.sh:12:: egrep -i '^Host \w+' \
examples/ch10/ssh_config
++trace examples/ch10/select-ssh.sh:12:: cut '-d ' -f2-
1) Exit 3) gitlab.com 5) mythtv-be01
2) git.atlas.oreilly.com 4) github.com 6) kitchen
SSH to> 1
+trace examples/ch10/select-ssh.sh:13:: case $REPLY in
+trace examples/ch10/select-ssh.sh:14:: exit 0
```

Возможно, вы захотите вернуться к разделу «Строгий режим bash» в главе 9 и прочитать о режиме, помогающем выявлять малозаметные проблемы и предотвращать их в будущем.



### Вывод содержимого окружения

Содержимое окружения полезно журналировать при запуске сценария из cron, в системе непрерывной интеграции и в других случаях, когда окружение может измениться или отличаться от ожидаемого. Для этого часто используются внешние команды `env` и `printenv`, однакостроенная команда `set` может дать значительно больше информации, включая сведения о функциях. С другой стороны, `set` выводит много лишних подробностей, поэтому решайте сами, какой способ подходит для вас.



### Опасность утечки информации об окружении

Если вы регистрируете содержимое своего окружения, помните, что оно часто содержит конфиденциальные сведения, такие как имена пользователей, пароли, ключи API и т. д.! Чтобы предотвратить вывод такой информации, используйте однострочники `sed` или Perl:

```
set | perl -pe 's/^($SECRET=.*/\1<REDACTED>/g;'
```

Начиная с версии 3.0 bash поддерживает флаг `--debugger` и возможность расширенной отладки `shopts`. Но мы их никогда не использовали, поскольку нам всегда хватало команды `set -x`. Запуск `bash --debugger - /path/to/script` может привести к следующему результату:

```
/path/to/script: /usr/share/bashdb/bashdb-main.inc: No such file or directory  
/path/to/script: warning: cannot start debugger; debugging mode disabled
```

Если вы увидите это сообщение, то загляните на сайт проекта Bash Debugger Project<sup>1</sup>, в рамках которого разрабатывается «отладчик исходного кода для bash, следующий синтаксису команды `gdb`». Проблема может быть обусловлена несовместимостью с версией bash, поэтому убедитесь, что ваша система соответствует требованиям.

Смотрите также:

- <http://bashdb.sourceforge.net/>;
- пример 10.3;
- раздел «Инструмент проверки оформления кода на bash» в главе 11;
- раздел «Отладочный и подробный режимы вывода» в главе 8;
- раздел «`printf` для повторного использования или отладки» в главе 6;
- раздел «Строгий режим bash» в главе 9.

## Модульное тестирование в bash

В репозитории GitHub<sup>2</sup> есть фреймворк модульного тестирования для сценариев командной оболочки Bourne, аналогичный фреймворкам JUnit, PyUnit и т. д., и linter – инструмент проверки оформления кода на bash, о котором мы поговорим в главе 11. Вам стоит познакомиться и с тем, и с другим.

---

<sup>1</sup> <http://bashdb.sourceforge.net/>.

<sup>2</sup> <https://github.com/kward/shunit2>.

## В заключение

Надеемся, что приведенные в этой главе советы и примеры понятны и вместе с тем достаточно гибки, чтобы помочь в вашей повседневной работе с bash и создать основу для дальнейшего развития. Если вам понравилась эта глава, то, вероятно, понравится и наша книга «*bash Cookbook*», которая содержит несколько сотен страниц полезных «рецептов», подобных представленным выше.

## ГЛАВА 11

---

# Разработка своего руководства по стилю

Главная тема этой книги — знакомство с идиоматическими приемами программирования на bash и правилами оформления кода. Надеемся, что нам удалось представить все необходимые для этого инструменты. Стиль — важный способ пояснить, как *мы* пишем код. Выберите рекомендации по стилю на свой вкус, запишите и придерживайтесь их.

В этой книге мы рассмотрели несколько важных аспектов оформления кода и другие рекомендации, о которых следует помнить при проектировании и разработке систем. Вы можете использовать эту главу в качестве отправной точки для создания собственного руководства по стилю или просто принять предложенное как есть, если оно вам нравится.

В приложении представлены дополнительные сведения без их обсуждения. Его можно использовать как справочник по стилю. Кроме того, материалы приложения с разметкой Markdown и HTML можно найти на странице этой книги в GitHub<sup>1</sup>.

---

<sup>1</sup> <https://github.com/vossenjp/bashidioms-examples/tree/master/appa>.

Запомните следующие основные принципы:

- Прежде всего — KISS (Keep It Simple, Stupid! — не будь глупцом, упрощай!). Сложность — враг безопасности<sup>1</sup>, но она также затрудняет чтение и понимание кода. Конечно, при современных требованиях системы не могут быть простыми, но постарайтесь не делать их сложнее, чем требуется.
- Одно из следствий нарушения принципа KISS — усложнение отладки. Как сказал Брайан Керниган (Brian Kernighan): «Отладка в два раза сложнее, чем написание кода, поэтому, если вы пишете настолько сложный код, насколько способны, то по определению вы недостаточно умны, чтобы отладить его».
- Страйтесь не изобретать велосипед. Все, что вы задумали, скорее всего, уже было сделано раньше, и, вероятно, имеется подходящий готовый инструмент или библиотека. Если такой инструмент уже установлен, просто используйте его. Как бы вы ни старались, вы не сможете превзойти качество и надежность инструмента `rsync`, поэтому просто используйте его. Если вы нашли подходящий код в интернете... что ж, можно подумать и о его использовании.
- Заранее планируйте особые случаи или переопределения, поскольку без них не обойтись. Возьмите из дистрибутива Linux файл `/etc/thing/global.cfg` с настройками по умолчанию, а затем реализуйте возможность переопределения настроек в каталоге `/etc/thing/config.d/` или подобном ему. См. раздел «Настроочные каталоги» в главе 9.
- Код не существует вне системы управления версиями! Рано или поздно он будет потерян, и тогда его *действительно* не станет.
- Документируйте все (но не нужно писать книгу о своем сценарии). Пишите свой код, комментарии и документацию в расчете на то, что их будет читать человек, который присоединится

---

<sup>1</sup> Доказательства вы найдете в статье «Complexity is the worst enemy of security» (<https://oreil.ly/zMJLF>).

к команде через год, когда вы забудете, *почему* вы сделали *именно так*. Документируйте приемы, которые *не* сработали, и почему, и особенно приемы, которые потенциально могут навредить (`rm -rf /$undefined_variable` оказалось *по-настоящему плохой идеей!*).

- Держите код и документацию «сухими»<sup>1</sup>: не повторяйтесь. Несколько копий одного и того же кода обязательно рассинхронизируются — вопрос лишь в том, когда это произойдет. Используйте функции и библиотеки; избегайте «сырости»<sup>2</sup>.

Положения «Дзен Python» применимы и к bash. Попробуйте выполнить команду `python -c "import this"` или загляните в документацию Python<sup>3</sup>.



#### Руководство по стилю bash не переносится на другие командные оболочки

Это руководство по стилю предназначено исключительно для bash, и его нельзя перенести на POSIX, Bourne, Dash и другие командные оболочки. Если вам придется писать сценарии для них, переработайте и адаптируйте рекомендации из этого руководства, чтобы учесть особенности синтаксиса и возможности этих оболочек.

Будьте особенно осторожны в Docker и других контейнеризаторах, где `/bin/sh` не ссылается на bash, а `/bin/bash` может вообще отсутствовать! Это относится и к ограниченным окружениям, таким как системы интернета вещей и промышленные контроллеры. См. разделы «bash в контейнерах» в предисловии и «Shebang!» в главе 9.

А если конкретно, что должно быть отражено в руководстве по стилю? В следующих разделах мы рассмотрим некоторые рекомендации.

---

<sup>1</sup> Сухой — англ. dry. DRY, или Don't Repeat Yourself (не повторяйся), — один из принципов разработки. — *Примеч. пер.*

<sup>2</sup> Сырой — англ. wet. WET, или We Enjoy Typing (нам нравится печатать), — один из антипринципов разработки. — *Примеч. пер.*

<sup>3</sup> <https://oreil.ly/O2nYx>.

## Удобочитаемость

Удобочитаемость кода важна! Или, как говорят программисты на Python, *читаемость имеет значение*. Код пишется один раз, но вы (и другие), скорее всего, будете читать его много раз. Потратьте лишние секунды или минуты, подумайте о тех, кому придется читать этот код в следующем году... вполне вероятно, что это будете вы сами. Стремитесь к балансу между абстракцией (DRY – не повторяйся) и удобочитаемостью:

- KISS (Keep It Simple, Stupid! – не будь глупцом, упрощай!).
- Удобочитаемость: не старайся быть «умным», старайся быть ясным.
- Понятные имена имеют решающее значение!
- Всегда используйте заголовки.
- Если возможно, предусмотрите вывод полезной информации при получении `-h`, `--help` и неверных аргументов!
  - Используйте встроенные документы (с отступами) вместо последовательностей инструкций `echo` со строками, потому что встроенный документ проще обновить и переформатировать.
- Для включения файлов с настройками, которые должны иметь расширение `.cfg` (или `.conf`, или любое другое, соответствующее вашим стандартам), используйте `source` вместо точки `(.)`. Точку легко не заметить и сложнее найти.
- Если возможно, используйте даты в формате ISO-8601.
- Страйтесь упорядочивать элементы списков, это уменьшит вероятность дублирования, а также упростит добавление и удаление элементов. Примерами могут служить IP-адреса (используйте GNU-версию `sort -v`), имена хостов, пакеты для установки, операторы `case` и содержимое переменных или массивов/списков.

- Если возможно, используйте для облегчения понимания длинные ключи команд, например `diff --quiet` вместо `diff -q`, но следите за переносимостью на системы, отличные от GNU/Linux.
  - Если какие-то ключи имеют только короткий или неочевидный вариант, добавьте поясняющий комментарий.
  - Страйтесь документировать свой выбор: почему взят именно этот ключ и даже почему не подходят другие ключи.
  - Обязательно документируйте ключи, применение которых заманчиво, но на самом деле вызывает проблемы, особенно если вы их часто используете.

Выбирая имена для переменных, страйтесь избегать чересчур общих понятий, таких как:

```
`${GREP} ${pattern} ${file}`
```

Эти имена слишком абстрактны, они могут использоваться повторно и в то же время практически не зависят от контекста. Давайте заменим обратные кавычки `` более современной, читаемой (как нам кажется) и определенно более пригодной к вложению конструкцией `$()`. Но самое важное — уберем лишние фигурные скобки `${}` и используем осмысленные имена:

```
$(grep "$re_process_errors" $critical_process_daily_log_file)
```

Через некоторое время вы обнаружите, что нашли общие с коллегами (осмелимся сказать) идиоматические способы выражения понятий и операций, с которыми вы часто сталкиваетесь. Значит, настал момент написать руководство по стилю, если у вас его еще нет.

Если же вы занимаетесь доработкой или сопровождением кода, то лучше следовать соглашениям, принятым в этом коде, или придется изменять стиль и, возможно, реорганизовывать его целиком.

## Комментарии

О комментариях написано много: что, где, когда и т. д. Тем не менее, приведем некоторые рекомендации по оформлению комментариев:

- *Всегда используйте заголовки.*
- Пишите комментарии так, будто они предназначены для нового члена вашей команды, который придет через год.
- Комментируйте свои функции.
- Описывайте не что вы делаете или не делаете, а почему.
  - Исключение: комментируйте, что вы делаете, когда код на bash труден для понимания.
- Добавляйте комментарии, описывающие ключи внешних программ, особенно если они короткие или неочевидные.
- Начинайте комментарий с заглавной буквы, но опускайте знаки пунктуации в конце, если только комментарий не состоит из нескольких предложений.

Добавляйте *полезные* комментарии, объясняющие, *почему* вы поступили именно так и каковы ваши намерения! Например:

```
continue # К следующему файлу в цикле for
```

В теории нет необходимости объяснять, *что* делается, если код написан достаточно понятно. Но иногда код на bash слишком запутан. Вот, например, прием с подстановкой переменных (см. «Удаление пути или префикса» в главе 4):

```
PROGRAM=${0##*/} # Аналог basename на bash
```

Логические блоки кода полезно выделять разделителями. Но не добавляйте закрывающий разделитель внизу — он лишь внесет ненужный «шум» и уменьшит количество полезных строк кода на экране. Что бы

вы ни делали, *не* стройте рамки из символов со всех четырех сторон! Это совершенно не нужно. Вы будете тратить слишком много времени на «правильное оформление». Что еще хуже, впоследствии это будет мешать исправлять или обновлять комментарии, потому что тогда придется снова исправлять рамки.

Не делайте так:

```
#####
# Не стройте такие рамки, как эта!
#
# В будущем, когда понадобится исправить комментарий, они
# потребуют дополнительных усилий, чтобы сохранить форму рамки.
# И это не самый плохой пример
#####
```

Делайте так:

```
#####
# Оформляйте рамки, как здесь!
#
# Такие комментарии проще редактировать, потому что не приходится
# заботиться о выравнивании правой рамки. Если потребуется внести
# что-то новое, достаточно просто добавить начальный символ "#"
```

## Имена

Важно выбирать говорящие имена. На момент написания кода, когда все детали еще свежи в памяти, разница между `$file` и `$critical_process_daily_log_file` не кажется существенной, кроме необходимости вводить лишние символы. Но мы гарантируем, что дополнительное время, потраченное на выбор и набор говорящих имен, окупится сторицей за счет снижения вероятности ошибок и уменьшения времени, которое потребуется в будущем, чтобы понять и улучшить код. Некоторые рекомендации по стилю, связанные с именами:

- Понятные имена имеют решающее значение!
- Имена глобальных переменных и констант записывайте ЗА-ГЛАВНЫМИ буквами.

- Страйтесь не изменять глобальные переменные, хотя иногда это упрощает код (**KISS**).
- Объявляйте константы с помощью `readonly` или `declare -r`.
- Имена других переменных записывайте строчными буквами.
- Для имен функций используйте Смешанный\_Регистр.
- Не используйте ВерблюжийРегистр, заменяйте пробелы символом подчеркивания (`_`) и не забывайте, что дефис (`-`) нельзя использовать в именах переменных.
- Минимизируйте использование массивов `bash`, поскольку они часто сложны для чтения (см. главу 7). Во многих случаях хорошо работает конструкция `for var in $regular_var`.
- С начала сценария вместо `$1`, `$2`, ... `$N` используйте переменные с говорящими именами. Такой код не только проще читать и отлаживать, он также более удобен для внедрения значений по умолчанию, добавления или реорганизации аргументов.
- Различайте типы ссылок, например `$input_file` и `$input_dir`.
- Используйте метки «**FIXME**» и «**TODO**» с именами и номерами заявок, если это уместно.

Подумайте, насколько легко перепутать или допустить опечатку и использовать неправильно следующие имена:

```
file1='/path/to/input'  
file2='/path/to/output'
```

Намного проще, понятнее и менее подвержен ошибкам такой код:

```
input_file='/path/to/input'  
output_file='/path/to/output'
```

Кроме того, не экономьте силы на ввод символов и не сокращайте имена: набирая имя `$filename`, сложнее ошибиться. А при использовании сокращений нередко потом трудно будет вспомнить, какое из них требуется: `$filenm`, `$f1name` или `$f1nm`?

## ФУНКЦИИ

Перейдем к рекомендациям по оформлению функций:

- *Всегда используйте заголовки.*
- Хорошие имена имеют решающее значение!
- Функции должны определяться до их использования.
  - Группируйте функции в начале сценария и отделяйте их друг от друга двумя пустыми строками и разделяющими комментариями.
  - Не размещайте между функциями блоки другого кода!
- Используйте комбинацию Верблюжьей\_И\_Змеиной\_Нотации, чтобы имена функций отличались от имен переменных.
- Используйте `function My_Func_Name {` вместо `My_Func_Name()` `{`, так как такое объявление понятнее и его проще отыскать с помощью команды `grep -P '^\\s*function '`.
- Каждая функция должна содержать комментарий, описывающий, что она делает, какие данные принимает (включая глобальные) и какие данные выводит.
- Если в сценарии имеются полезные обособленные фрагменты кода или фрагменты, используемые многократно (а также похожие друг на друга), превратите их в функции. Если они востребованы в разных сценариях, как, например, журналирование или отправка электронной почты, подумайте о создании библиотеки — файла, который можно подключать к сценариям.
- Начинайте имена «библиотечных» функций с одинакового префикса, например подчеркивания (`_`) — `_Log`.
- Подумайте об использовании слов-заполнителей для удобства чтения, если это имеет смысл. Определите их как `local junk1="$2"` # Неиспользуемый заполнитель. Например: `_Create_File_If_Needed "/path/to/$file" #` содержит важное значение.
- Используйте встроенную команду `local` для определения локальных переменных в функциях. Но имейте в виду, что такие переменные маскируют неудачный код возврата, поэтому

объявляйте и инициализируйте их в отдельных строках, используя подстановку команд. Например, вначале `local my_input`, затем `my_input=$(some-command)`.

- Функции, насчитывающие более 25 строк кода, закрывайте комментарием `} # Конец функции MyFuncName`, чтобы упростить поиск конца функции на экране. Для функций короче 25 строк это не обязательно, но не возбраняется, если такие комментарии не загромождают код.
- Обявление функции `main` в большинстве случаев не требуется.
  - Использование `main` имеет смысл для программистов на Python и C, или если код используется также в качестве библиотеки и такая функция нужна для модульного тестирования.
- Отделяйте основной код от блока определения функций двумя пустыми строками и разделительным комментарием, особенно если функций много.

Определите в своей библиотеке одну функцию для журналирования (например, `_Log`) и используйте ее! В противном случае есть риск получитьдишую смесь из функций журналирования, стилей и ссылок. В идеале, как мы уже говорили, используйте журналирование в `syslog` и позвольте операционной системе самой позаботиться о месте назначения, ротации журналов и т. д.

## Кавычки

Кавычки не вызывают сложностей до определенного момента. Мы знаем многие их особенности, но все равно иногда действуем методом проб и ошибок, особенно когда пытаемся создать односторонний интерфейс для запуска команды от имени другого пользователя через `sudo` или на другом узле через `ssh`. Добавьте несколько инструкций `echo` и будьте внимательны. Ниже перечислены рекомендации, связанные с кавычками:

- Заключайте в кавычки переменные и строки, потому что это выделяет их и поясняет ваши намерения, но не используйте кавычки, если они загромождают код или мешают расширению.

- Не заключайте в кавычки целые числа.
- Используйте одинарные кавычки, если не требуется интерполяция.
- Не используйте конструкцию  `${var}` без необходимости, чтобы не загромождать код. Необходима она, например, в таких случаях, как  `${variable}_suffix` или  `${being_lower_cased}, , }`.
- Заключайте в кавычки подстановку команд, например `var="$(command)"`.
- Всегда заключайте в кавычки обе части выражения любого оператора проверки, например  `[[ "$foo" == 'bar' ]]`. Исключения:
  - Если одна часть выражения является целым числом.
  - Если используется `~`, потому что регулярные выражения нельзя заключать в кавычки!
- Заключайте в одинарные кавычки переменные внутри инструкций `echo`, например `echo "cd to '$DIR' failed"`, потому что это поможет определить переменные, которые оказались неинициализированными или пустыми.
  - Другой вариант — `echo "cd to [$DIR] failed"`, если так вам больше нравится.
  - Если переменная не определена, то при использовании `set -u` вы получите сообщение об ошибке. Но не в случае, если она определена и имеет пустое значение.
- Страйтесь использовать одинарные кавычки в строках формата команды `printf` (см. подраздел «Вывод POSIX» и остальную часть раздела «Функция `printf`» в главе 6).

## Форматирование

Рекомендации, связанные с форматированием кода:

- Не пренебрегайте отступами! Несколько лишних пробелов, как правило, не имеют значения в `bash` (за исключением пробелов

вокруг =), но отступы в командах, составляющих один блок, облегчают понимание и позволяют увидеть сходства и различия.

- *Не оставляйте пробелов в конце строк!* Такие пробелы вызовут лишний шум в системе управления версиями.
- Оформляйте отступы четырьмя пробелами, а во встроенных документах — табуляцией.
- Переносите длинные строки примерно на 78-м знаке (включая пробелы). В строке, следующей за переносом, сделайте дополнительный отступ на два пробела. Разрывайте строки непосредственно перед символом | или >, чтобы они были сразу заметны при просмотре кода.
- Код, открывающий блок, размещайте в одной строке, например:

```
if expression; then
    for expression; do
```
- В элементах списка case..esac оформляйте дополнительный отступ в четыре пробела и закрывайте их ;; с тем же отступом. Блоки кода в каждом элементе должны иметь дополнительный отступ в четыре пробела.
  - Однострочные элементы следует закрывать символами ;;; в той же строке.
  - Предпочтительнее выравнивать отступы по ) в каждом элементе, если это не загромождает код.
  - Правильное форматирование иллюстрирует пример 8.4 на с. 119.

Многие считают ненужным разбивать строки шириной более 70–80 знаков (с пробелами), предполагая, что все используют широкие дисплеи и интегрированные среды разработки. Во-первых, в зависимости от команды и личных предпочтений человека это не всегда соответствует действительности. Во-вторых, даже если это верно, когда *действительно* что-то случится и вам придется заняться отладкой в vi и консоли 80 × 24, вы не оцените строки кода длиной 200+.

Переносите строки.

Переносите их непосредственно *перед* важной частью, чтобы при беглом просмотре, когда взгляд скользит вдоль левого края, продолжения были сразу же заметны. Мы добавляем половину обычного отступа после переноса. В книге сложно привести хороший (плохой?) пример длинной строки, потому что она будет перенесена при меньшей длине, чем 70–80 знаков. Но есть простой и, на наш взгляд, читаемый пример:

```
... Много разного кода, с отступами...
/long/path/to/some/interesting/command \
| grep "$re_stuff_we_care_about" \
| grep -v "$re_stuff_to_ignore" \
| /path/to/email_or_log_command ...
...еще масса кода
```

## Синтаксис

Рекомендации, связанные с синтаксисом:

- В начале сценария укажите `#!/bin/bash` – или `#!/usr/bin/env bash`, но не `#!/bin/sh`.
- Используйте `$@`, если вы не уверены, что вам *действительно* нужна `$*`.
- Для проверки равенства используйте `==` вместо `=`, чтобы избежать путаницы с присваиванием.
- Вместо обратных кавычек и обратных апострофов используйте `$()`.
- Используйте `[[` вместо `[`, если только вам не нужна `[` для переносимости, например в `dash`.
- Для целочисленной арифметики применяйте `(( ))` и `$(( ))` и избегайте `let` и `expr`.
- Используйте `[[ expression ]] && block` или `[[ expression ]] || block`, если такой код легко читается. Не применяйте `[[ expression ]] && block || block`, потому что этот код может сделать не то, что вы думаете; используйте для этого `if .. then .. (elif ..) else .. fi`.

- Попробуйте использовать строгий режим (см. раздел «Строгий режим bash» в главе 9).
  - `set -euo pipefail` предотвратит или поможет обнаружить многие простые ошибки.
  - `IFS=$'\n\t'` применяйте с осторожностью (а лучше вообще избегайте этой команды).

## Другие рекомендации

Дополнительные рекомендации:

- В «системных» сценариях используйте журналирование в `syslog` и позвольте операционной системе самой позаботиться о пункте назначения, ротации журналов и т. д.
- Сообщения об ошибках должны выводиться в `STDERR`, например `echo 'A Bad Thing happened' 1>&2`.
- Проверяйте доступность внешних инструментов с помощью `[ -x /path/to/tool ] || { ...блок обработки ошибки... }`.
- Когда что-то не получается, должны выводиться информативные сообщения.
- Определите коды выхода в операторах `exit`, особенно в случае выхода по ошибке.

## Шаблон сценария

Ниже приводится шаблон сценария, который можно использовать полностью или частично (он действительно довольно длинный) для создания новых сценариев:

```
#!/bin/bash -
# Или может быть: #!/usr/bin/env bash
# <Имя>: <описание>
# Автор и дата:
# Текущий сопровождающий?
```

```
# Авторские права/лицензия?  
# Местонахождение кода? (Хост, путь и т. п.)  
# Проект/репозиторий?  
# Предупреждения/недостатки?  
# Порядок использования? (лучше оформить с ключами ` -h` и/или ` --help` !)  
# $URL$ # Если используется SVN  
ID='' # Если используется SVN  
#  
PROGRAM=${0##*/} # Версия `basename` на bash  
  
# Неофициальный строгий режим?  
#set -euo pipefail  
### БУДЬТЕ ВНИМАТЕЛЬНЫ при использовании IFS=$'\n\t'  
  
# Глобальные переменные и константы записываются заглавными буквами  
LOG_DIR='/path/to/log/dir'  
  
### Подумайте о добавлении в шаблон обработки аргументов, см.:  
# examples/ch08/parseit.sh  
# examples/ch08/parselong.sh  
# examples/ch08/parselonghelp.sh  
  
# Имена функций следует записывать в Смешанном_Регистре  
#####
# Определения функций  
#-----  
# Пример функции  
# Глобальные переменные: нет  
# Входные параметры: нет  
# Выходные значения: нет  
function Foo {  
    local var1="$1"  
    ...  
} # Конец функции foo  
  
#-----  
# Пример другой функции  
# Глобальные переменные: нет  
# Входные параметры: нет  
# Выходные значения: нет  
function Bar {  
    local var1="$1"  
    ...  
} # Конец функции bar  
  
#####
# Основной  
# Код...
```

## Другие руководства по стилю

Мы настоятельно рекомендуем вам использовать руководство по стилю! Если вам не нравится наше и вы не хотите составлять свое, то можете посмотреть другие руководства, возможно, они понравятся больше:

- Руководство по стилю оформления кода на языке командной оболочки от Google<sup>1</sup>.

Мы не во всем согласны с этим руководством, но все же считаем его очень хорошим и довольно подробным, к тому же многие проекты приняли его на вооружение... Можно следовать ему или использовать как отправную точку для разработки собственного стиля.

Это руководство вызывает следующие возражения:

- Ограничение объема кода в 100 строк имеет смысл для окружений, формируемых Google. Но многие наши сценарии превышают это ограничение, попадая, согласно руководству, в группу «других утилит», которые «сравнительно мало манипулируют данными».
- Отступы в два пробела нам кажутся недостаточными. Мы используем отступы в два пробела только для продолжения строк, а в остальных случаях — в четыре пробела.
- "\${var}" слишком загромождает код. Мы предпочитаем "\$var", когда это возможно.
- `function cleanup() {?` Ну уж нет!
- `; ;` закрывает блок `case..esac` и должен иметь тот же отступ, что и сам блок.
- Еще несколько руководств по стилю, с которыми имеет смысл ознакомиться:
  - [https://linuxcommand.org/lc3\\_adv\\_standards.php](https://linuxcommand.org/lc3_adv_standards.php);
  - <https://www.ovirt.org/develop/infra/infra-bash-style-guide.html>;

---

<sup>1</sup> <https://oreil.ly/HWaSV>.

- <https://wiki.bash-hackers.org/scripting/style>;
- <http://mywiki.woledge.org/BashGuide/Practices>.
- Отличный сборник дополнительной информации: <http://mywiki.woledge.org/BashPitfalls>.
- Также можно поискать в интернете по фразам:
  - «Shell Style Guide» (руководство по стилю оформления кода командной оболочки);
  - «Bash Style Guide» (руководство по стилю оформления кода bash);
  - «Shell script coding standards» (стандарты программирования на языке командной оболочки).

## Инструмент проверки оформления кода на bash

Вместо руководства по стилю или в дополнение к нему можно использовать дополнительный *инструмент проверки оформления кода (linter)*. Мы не пользуемся такими инструментами, но посчитали должным упомянуть об их существовании. Собственно говоря, нам известен только один такой инструмент — `shellcheck` (<https://www.shellcheck.net>). Он особенно требователен к кавычкам и экранированию, но мы не всегда согласны с его предложениями, хотя эти параметры можно настраивать в определенных пределах. Тем не менее это очень достойный инструмент, и вам стоит взглянуть на него.

- Исходные файлы: <https://github.com/koalaman/shellcheck>.
- История развития: <https://www.vidarholen.net/contents/blog/?p=859>.
- Известные проблемы: <https://github.com/koalaman/shellcheck/wiki/Checks>.
- Настройки: <https://github.com/koalaman/shellcheck/wiki/Directive>.
- Игнорирование ошибок: <https://github.com/koalaman/shellcheck/wiki/Ignore>.

- Замечания по установке в CentOS-6: <https://github.com/koalaman/shellcheck/wiki/CentOS6>.

Дистрибутивы могут содержать старые версии shellcheck. Новые версии для Linux распространяются в виде архивов tar с единственным скомпилированным двоичным файлом, который можно поместить в один из каталогов, перечисленных в \$PATH.

## В заключение

Надеемся, что эта книга помогла вам разобраться, как читать и писать идиоматический код. В заключительной главе мы собрали советы, на основе которых можно разработать собственное руководство по стилю написания кода.

Успехов в программировании на bash!

## ПРИЛОЖЕНИЕ

---

# Руководство по стилю

В этом приложении приводятся рекомендации из главы 11, но без комментариев и примеров. В каталоге с примерами имеется файл в формате Markdown<sup>1</sup>, который можно загрузить и отредактировать по своему усмотрению, а затем распечатать или просматривать с помощью `pandoc` или другого инструмента.



### Руководство по стилю `bash` не переносится на другие командные оболочки

Это руководство по стилю предназначено исключительно для `bash`, и его нельзя перенести на POSIX, Bourne, Dash и другие командные оболочки. Если вам придется писать сценарии для них, переработайте и адаптируйте рекомендации из этого руководства, чтобы учесть особенности синтаксиса и возможности этих оболочек.

Будьте особенно осторожны в Docker и других контейнеризаторах, где `/bin/sh` не ссылается на `bash`, а `/bin/bash` может вообще отсутствовать! Это относится и к ограниченным окружениям, таким как системы интернета вещей и промышленные контроллеры. См. разделы «`bash` в контейнерах» в предисловии и «Shebang!» в главе 9.

---

<sup>1</sup> <https://github.com/vossenjp/bashidioms-examples/tree/master/appa>.

## Удобочитаемость

Удобочитаемость кода важна! Или, как говорят программисты на Python, *читаемость имеет значение*. Код пишется один раз, но вы (и другие), скорее всего, будете читать его много раз. Потратьте лишние секунды или минуты, подумайте о тех, кому придется читать этот код в следующем году... вполне вероятно, что это будете вы сами. Стремитесь к балансу между абстракцией (DRY – не повторяйся) и удобочитаемостью:

- KISS (Keep It Simple, Stupid! – не будь глупцом, упрощай!).
- *Удобочитаемость*: не старайся быть «умным», старайся быть ясным.
- Понятные имена имеют решающее значение!
- *Всегда используйте заголовки*.
- Если возможно, предусмотрите вывод полезной информации при получении `-h`, `--help` и неверных аргументов!
  - Используйте встроенные документы (с отступами) вместо последовательностей инструкций `echo` со строками, потому что встроенный документ проще обновить и переформатировать.
- Для включения файлов с настройками, которые должны иметь расширение `.cfg` (или `.conf`, или любое другое, соответствующее вашим стандартам), используйте `source` вместо точки `(.)`. Точки легко не заметить и сложнее найти.
- Если возможно, используйте даты в формате ISO-8601.
- Страйтесь упорядочивать элементы списков, это уменьшит вероятность дублирования, а также упростит добавление и удаление элементов. Примерами могут служить IP-адреса (используйте GNU-версию `sort -v`), имена хостов, пакеты для установки, операторы `case` и содержимое переменных или массивов/списков.
- Если возможно, используйте для облегчения понимания длинные ключи команд, например `diff --quiet` вместо `diff -q`, но следите за переносимостью на системы, отличные от GNU/Linux.

- Если какие-то ключи имеют только короткий или неочевидный вариант, добавьте поясняющий комментарий.
- Страйтесь документировать свой выбор: почему выбран именно этот ключ и даже почему не подходят другие ключи.
- Обязательно документируйте ключи, применение которых заманчиво, но на самом деле вызывает проблемы, особенно если вы их часто используете.

## Комментарии

- *Всегда используйте заголовки.*
- Пишите комментарии так, будто они предназначены для нового члена вашей команды, который придет через год.
- Комментируйте свои функции.
- Описывайте не что вы делаете или не делаете, а почему.
  - Исключение: комментируйте, что вы делаете, когда код на bash труден для понимания.
- Добавляйте комментарии, описывающие ключи внешних программ, особенно если они короткие или неочевидные.
- Начинайте комментарий с заглавной буквы, но опускайте знаки пунктуации в конце, если только комментарий не состоит из нескольких предложений.

## Имена

- Понятные имена имеют решающее значение!
- Имена глобальных переменных и констант записывайте ЗА-ГЛАВНЫМИ буквами.
  - Страйтесь не изменять глобальные переменные, хотя иногда это упрощает код (KISS).
  - Объявляйте константы с помощью `readonly` или `declare -r`.

- Имена других переменных записывайте строчными буквами.
- Для имен функций используйте Смешанный\_Регистр.
- Заменяйте пробелы символом подчеркивания (\_) и не забывайте, что дефис (-) нельзя использовать в именах переменных.
- Минимизируйте использование массивов bash, поскольку они часто сложны для чтения (см. главу 7). Во многих случаях хорошо работает конструкция `for var in $regular_var`.
- С начала сценария вместо `$1`, `$2`, ... `$N` используйте переменные с говорящими именами. Такой код не только проще читать и отлаживать, он также более удобен для внедрения значений по умолчанию, добавления или реорганизации аргументов.
- Различайте типы ссылок, например `$input_file` и `$input_dir`.
- Используйте метки «FIXME» и «TODO» с именами и номерами заявок, если это уместно.

## ФУНКЦИИ

- Всегда используйте заголовки.
- Хорошие имена имеют решающее значение!
- Функции должны определяться до их использования.
  - Группируйте функции в начале сценария и отделяйте их друг от друга двумя пустыми строками и разделяющими комментариями.
  - Не размещайте между функциями блоки другого кода!
- Используйте комбинацию Верблюжьей\_И\_Змеиной\_Нотаций, чтобы имена функций отличались от имен переменных.
- Используйте `function My_Func_Name {` вместо `My_Func_Name()` {}, так как такое объявление понятнее и его проще отыскать с помощью команды `grep -P '^\\s*function '`.
- Каждая функция должна содержать комментарий, описывающий, что она делает, какие данные принимает (включая глобальные) и какие данные выводит.

- Если в сценарии имеются полезные обособленные фрагменты кода или фрагменты, используемые многократно (а также похожие друг на друга), превратите их в функции. Если они востребованы в разных сценариях, как, например, журналирование или отправка электронной почты, подумайте о создании библиотеки — файла, который можно подключать к сценариям.
- Начинайте имена «библиотечных» функций с одинакового префикса, например подчеркивания (\_) — `_Log`.
- Подумайте об использовании слов-заполнителей для удобства чтения, если это имеет смысл. Определите их как `local junk1="$2"` # Неиспользуемый заполнитель. Например: `_Create_File_If_Needed "/path/to/$file"` # содержит важное значение.
- Используйте встроенную команду `local` для определения локальных переменных в функциях. Но имейте в виду, что такие переменные маскируют неудачный код возврата, поэтому объявляйте и инициализируйте их в отдельных строках, используя подстановку команд. Например, вначале `local my_input`, затем `my_input="$(some-command)"`.
- Функции, насчитывающие более 25 строк кода, закрывайте комментарием `} # Конец функции MyFuncName`, чтобы упростить поиск конца функции на экране. Для функций короче 25 строк это необязательно, но не возбраняется, если такие комментарии не загромождают код.
- Объявление функции `main` в большинстве случаев не требуется.
  - Использование `main` имеет смысл для программистов на Python и C, а также если код используется в качестве библиотеки и такая функция нужна для модульного тестирования.
- Отделяйте основной код от блока определения функций двумя пустыми строками и разделительным комментарием, особенно если функций много.

## Кавычки

- Заключайте в кавычки переменные и строки, потому что это выделяет их и поясняет ваши намерения, но не используйте кавычки, если они загромождают код или мешают расширению.
- Не заключайте в кавычки целые числа.
- Используйте одинарные кавычки, если не требуется интерполяция.
- Не используйте конструкцию  `${var}` без необходимости, чтобы не загромождать код. Необходима она, например, в таких случаях, как  `${variable}_suffix` или  `${being_lower_cased,,}`.
- Заключайте в кавычки подстановку команд, например `var="$(command)"`.
- *Всегда* заключайте в кавычки обе части выражения любого оператора проверки, например  `[[ "$foo" == 'bar' ]]`. Исключения:
  - Если одна часть выражения является целым числом.
  - Если используется `~`, потому что регулярные выражения нельзя заключать в кавычки!
- Заключайте в одинарные кавычки переменные внутри инструкций `echo`, например `echo "cd to '$DIR' failed"`, потому что это поможет определить переменные, которые оказались неинициализированными или пустыми.
  - Другой вариант — `echo "cd to [$DIR] failed"`, если так вам больше нравится.
  - Если переменная не определена, то при использовании `set -u` вы получите сообщение об ошибке. Но не в случае, если она определена и имеет пустое значение.
- Страйтесь использовать одинарные кавычки в строках формата команды `printf` (см. подраздел «Вывод POSIX» и остальную часть раздела «Функция `printf`» в главе 6).

## Форматирование

- Не пренебрегайте отступами! Несколько лишних пробелов, как правило, не имеют значения в bash (за исключением пробелов вокруг =), но отступы в командах, составляющих один блок, облегчают понимание и позволяют увидеть сходства и различия.
- *Не оставляйте пробелов в конце строк!* Такие пробелы вызовут лишний шум в системе управления версиями.
- Оформляйте отступы четырьмя пробелами, а во встроенных документах — табуляцией.
- Переносите длинные строки примерно на 78-м знаке (включая пробелы). В строке, следующей за переносом, сделайте дополнительный отступ на два пробела. Разрывайте строки непосредственно перед символом | или >, чтобы они были сразу заметны при просмотре кода.
- Код, открывающий блок, размещайте в одной строке, например:

```
if expression; then
    for expression; do
```
- В элементах списка case..esac оформляйте дополнительный отступ в четыре пробела и закрывайте их ; ; с тем же отступом. Блоки кода в каждом элементе должны иметь дополнительный отступ в четыре пробела.
  - Однострочные элементы следует закрывать символами ; ; в той же строке.
  - Предпочтительнее выравнивать отступы по ) в каждом элементе, если это не загромождает код.
  - Правильное форматирование иллюстрирует пример 8.4 на с. 119.

## Синтаксис

- В начале сценария укажите #!/bin/bash - или #!/usr/bin/env bash, но не #!/bin/sh.

- Используйте `$@`, если вы не уверены, что вам *действительно* нужна `$*`.
- Для проверки равенства используйте `==` вместо `=`, чтобы избежать путаницы с присваиванием.
- Вместо обратных кавычек и обратных апострофов используйте `$()`.
- Используйте `[[` вместо `[`, если только вам не нужна `[` для переносимости, например в `dash`.
- Для целочисленной арифметики применяйте `(( ))` и `$(( ))` и избегайте `let` и `expr`.
- Используйте `[[ expression ]] && block` или `[[ expression ]] || block`, если такой код легко читается. Не применяйте `[[ expression ]] && block || block`, потому что этот код может сделать не то, что вы думаете; используйте для этого `if .. then .. (elif ..) else .. fi`.
- Попробуйте использовать строгий режим (см. раздел «Строгий режим `bash`» в главе 9).
  - `set -euo pipefail` предотвратит или поможет обнаружить многие простые ошибки.
  - `IFS=$'\n\t'` применяйте с осторожностью (а лучше вообще избегайте этой команды).

## Другие рекомендации

- В «системных» сценариях используйте журналирование в `syslog` и позвольте операционной системе самой позаботиться о пункте назначения, ротации журналов и т. д.
- Сообщения об ошибках должны выводиться в `STDERR`, например `echo 'A Bad Thing happened' 1>&2`.
- Проверяйте доступность внешних инструментов с помощью `[ -x /path/to/tool ] || { ...блок обработки ошибки... }`.
- Когда что-то не получается, должны выводиться информативные сообщения.
- Определите коды выхода в операторах `exit`, особенно в случае выхода по ошибке.

## Шаблон сценария

```
#!/bin/bash -
# Или может быть: #!/usr/bin/env bash
# <Имя>: <описание>
# Автор и дата:
# Текущий сопровождающий?
# Авторские права/лицензия?
# Местонахождение кода? (Хост, путь и т. п.)
# Проект/репозиторий?
# Предупреждения/недостатки?
# Порядок использования? (лучше оформить с ключами ` -h` и/или ` --help` !)
# $URL$ # Если используется SVN
ID='' # Если используется SVN
#
PROGRAM=${0##*/} # Версия `basename` на bash

# Неофициальный строгий режим?
#set -euo pipefail
### БУДЬТЕ ВНИМАТЕЛЬНЫ при использовании IFS=$'\n\t'

# Глобальные переменные и константы записываются заглавными буквами
LOG_DIR='/path/to/log/dir'

### Подумайте о добавлении в шаблон обработки аргументов, см.:
# examples/ch08/parseit.sh
# examples/ch08/parselong.sh
# examples/ch08/parselonghelp.sh

# Имена функций следует записывать в Смешанном_Регистре
#####
# Определения функций
#-----
# Пример функции
# Глобальные переменные: нет
# Входные параметры: нет
# Выходные значения: нет
function Foo {
    local var1="$1"
    ...
}
# Конец функции foo
#-----
# Пример другой функции
```

```
# Глобальные переменные: нет
# Входные параметры: нет
# Выходные значения: нет
function Bar {
    local var1="$1"
    ...
} # Конец функции bar

#####
# Основной
# Код...
```

# 06 авторах

**Карл Олбинг (Carl Albing)** – профессор, исследователь и программист. Соавтор книг «bash Cookbook»<sup>1</sup> и «Cybersecurity Ops with bash»<sup>2</sup> (O'Reilly), а также автор видео *Great bash*<sup>3</sup>, опубликованного на сайте издательства O'Reilly. Занимался разработками в области программного обеспечения для компаний из разных стран мира и различных отраслей. Имеет степени бакалавра математики, магистра международного менеджмента и доктора информатики.

**Джей Пи Фоссен (JP Vossen)** работает в ИТ-индустрии с начала 1990-х годов. В конце 1990-х стал специализироваться на информационной безопасности. Проникся идеями написания сценариев и автоматизации с тех пор как освоил autoexec.bat. Был рад открыть для себя мощь и гибкость bash и GNU в Linux в середине 1990-х. Автор статей для журналов «Information Security Magazine», «SearchSecurity» и других изданий.

---

<sup>1</sup> <https://learning.oreilly.com/library/view/bash-cookbook-2nd/9781491975329>.

<sup>2</sup> Тронкон П., Олбинг К. «Bash и кибербезопасность. Атака, защита и анализ из командной строки Linux». СПб., издательство «Питер».

<sup>3</sup> <https://learning.oreilly.com/videos/great-bash/9781449307769>.

# Иллюстрация на обложке

На обложке книги «Идиомы bash» изображена раковина улитки-арфы (*Harpa articulatis*) — разновидности морских моллюсков из семейства Harpidae, насчитывающего около 55 видов. Большинство из них обитает в Индо-Тихоокеанском регионе, и только два вида — в прибрежных водах полуострова Баха (мексиканская часть Калифорнии). Улитка *Harpa articulatis* встречается на мелководье в Индийском и южной части Тихого океанов, вплоть до островов Фиджи и побережья Австралии. Этот вид отличается замысловатыми зубчатыми узорами и цветом раковин, размер которых составляет от 50 до 110 мм.

Семейство Harpidae характерно гладкой блестящей поверхностью раковин, по всей длине которых проходят мощные ребра. Расширяющееся отверстие имеет выемку внизу. Раковина не имеет крышечки — листового отростка, который играет роль дверцы и предотвращает высыхание моллюска. Вместо этого большая нога моллюска выступает далеко за край раковины.

Арфы — ночные хищники. Днем они закапываются в песок, а ночью выбираются наружу, чтобы поохотиться на крабов и креветок. Обнаружив добычу, арфа накрывает ее ногой и обволакивает слизью. При этом улитки сами могут стать жертвами более крупных крабов, рыб и хищных моллюсков. Чтобы защитить себя, арфы используют очень интересный прием: они отторгают заднюю часть ноги, которая продолжает извиваться и отвлекать хищника, в то время как улитка уползает.

Многие животные, изображаемые на обложках книг издательства O'Reilly, находятся под угрозой исчезновения; все они важны для нашего мира.

Иллюстрацию для обложки нарисовала Карен Монтгомери (Karen Montgomery) на основе древней гравюры из книги Pictorial Museum of Animated Nature.

# *Карл Олбинг, Джей Пи Фоссен*

## **Идиомы bash**

*Перевела с английского Л. Киселева*

Руководитель дивизиона

*Ю. Сергиенко*

Руководитель проекта

*А. Питиримов*

Ведущий редактор

*Е. Строганова*

Литературный редактор

*Д. Гудилин*

Художественный редактор

*В. Мостицан*

Корректоры

*Л. Галаганова, М. Молчанова*

Верстка

*Е. Цыцен*

Изготовлено в России. Изготовитель: ООО «Прогресс книга».

Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,  
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 04.2023. Наименование: книжная продукция.

Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции  
ОК 034-2014, 58.11.12 — Книги печатные  
профессиональные, технические и научные.

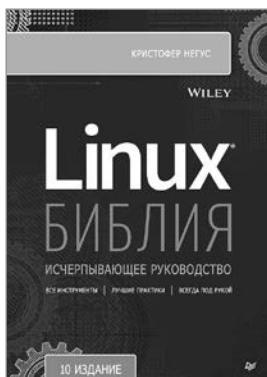
Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 02.02.23. Формат 70×100/16. Бумага офсетная. Усл. п. л. 16,700. Тираж 1000. Заказ 0000.

*Кристофер Негус*

## **БИБЛИЯ LINUX**

**10-е издание**



Полностью обновленное 10-е издание «Библии Linux» поможет как начинающим, так и опытным пользователям приобрести знания и навыки, которые выведут на новый уровень владения Linux. Известный эксперт и автор бестселлеров Кристофер Негус делает акцент на инструментах командной строки и новейших версиях Red Hat Enterprise Linux, Fedora и Ubuntu. Шаг за шагом на подробных примерах и упражнениях вы досконально поймете операционную систему Linux и пустите знания в дело. Кроме того, в 10-м издании содержатся материалы для подготовки к экзаменам на различные сертификаты по Linux.

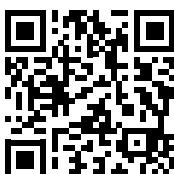


*Пол Тронкон, Карл Олбинг*

## **BASH И КИБЕРБЕЗОПАСНОСТЬ: АТАКА, ЗАЩИТА И АНАЛИЗ ИЗ КОМАНДНОЙ СТРОКИ LINUX**



Командная строка может стать идеальным инструментом для обеспечения кибербезопасности. Невероятная гибкость и абсолютная доступность превращают стандартный интерфейс командной строки (CLI) в фундаментальное решение, если у вас есть соответствующий опыт. Авторы Пол Тронкон и Карл Олбинг рассказывают об инструментах и хитростях командной строки, помогающих собирать данные при упреждающей защите, анализировать логи и отслеживать состояние сетей. Пентестеры узнают, как проводить атаки, используя колossalный функционал, встроенный практически в любую версию Linux.





**ИЗДАТЕЛЬСКИЙ ДОМ «ПИТЕР»**  
предлагает профессиональную, популярную  
и детскую развивающую литературу

Заказать книги оптом можно в наших представительствах

**РОССИЯ**

**Санкт-Петербург**

м. «Выборгская», Б. Сампсониевский пр., д. 29а;  
тел. (812) 703-73-73, доб. 6282; e-mail: dudina@piter.com

**Москва**

м. «Электрозводская», Семеновская наб., д. 2/1, стр. 1, 6 этаж;  
тел./факс (495) 234-38-15; e-mail: reception@piter.com

**БЕЛАРУСЬ**

**Минск**

ул. Харьковская, д. 90, пом. 18  
тел./факс: +37 (517)348-60-01, 374-43-25, 272-76-56  
e-mail: dudik@piter.com

**Издательский дом «Питер» приглашает к сотрудничеству авторов:**  
тел./факс (812) 703-73-72, (495) 234-38-15; e-mail: ivanova@piter.com  
**Подробная информация здесь:** <http://www.piter.com/page/avtoru>

---

**Издательский дом «Питер» приглашает к сотрудничеству зарубежных  
торговых партнеров или посредников, имеющих выход на зарубежный  
рынок:** тел./факс (812) 703-73-73, доб. 6282; e-mail: sales@piter.com

---

**Заказ книг для вузов и библиотек:**  
тел./факс (812) 703-73-73, доб. 6243; e-mail: uchebnik@piter.com

---

**Заказ книг в интернет-магазине:** на сайте [www.piter.com](http://www.piter.com);  
тел. (812) 703-73-74, доб. 6216; e-mail: books@piter.com

---

**Вопросы по продаже электронных книг:** тел. (812) 703-73-74, доб. 6217;  
e-mail: kuznetsov@piter.com



## ВАША УНИКАЛЬНАЯ КНИГА

Хотите издать свою книгу?

Книга может стать идеальным подарком для партнеров и друзей или отличным инструментом продвижения личного бренда. Мы поможем осуществить любые, даже самые смелые и сложные, идеи и проекты!

### МЫ ПРЕДЛАГАЕМ

- издание вашей книги
- издание корпоративной библиотеки
- издание книги в качестве корпоративного подарка
- издание электронной книги (формат ePub или PDF)
- размещение рекламы в книгах

### ПОЧЕМУ НАДО ВЫБРАТЬ ИМЕННО НАС

Более 30 лет издательство «Питер» выпускает полезные и интересные книги. Наш опыт — гарантия высокого качества. Мы печатаем книги, которыми могли бы гордиться и мы, и наши авторы.

### ВЫ ПОЛУЧИТЕ

- услуги по обработке и доработке вашего текста
- современный дизайн от профессионалов
- высокий уровень полиграфического исполнения
- продажи книги в крупнейших книжных магазинах страны
- продвижение книги (реклама в профильных изданиях и местах продаж; рецензии в ведущих СМИ; интернет-продвижение)

Мы имеем собственную сеть дистрибуции по всей России и в Белоруссии, сотрудничаем с крупнейшими книжными магазинами страны и ближнего зарубежья. Издательство «Питер» — постоянный участник многих конференций и семинаров, которые предоставляют широкие возможности реализации книг. Мы обязательно проследим, чтобы ваша книга имелась в наличии в магазинах и была выложена на самых видных местах. А также разработаем индивидуальную программу продвижения книги с учетом ее тематики, особенностей и личных пожеланий автора.

**Свяжитесь с нами прямо сейчас:**

Санкт-Петербург — Анна Титова, (812) 703-73-73, titova@piter.com