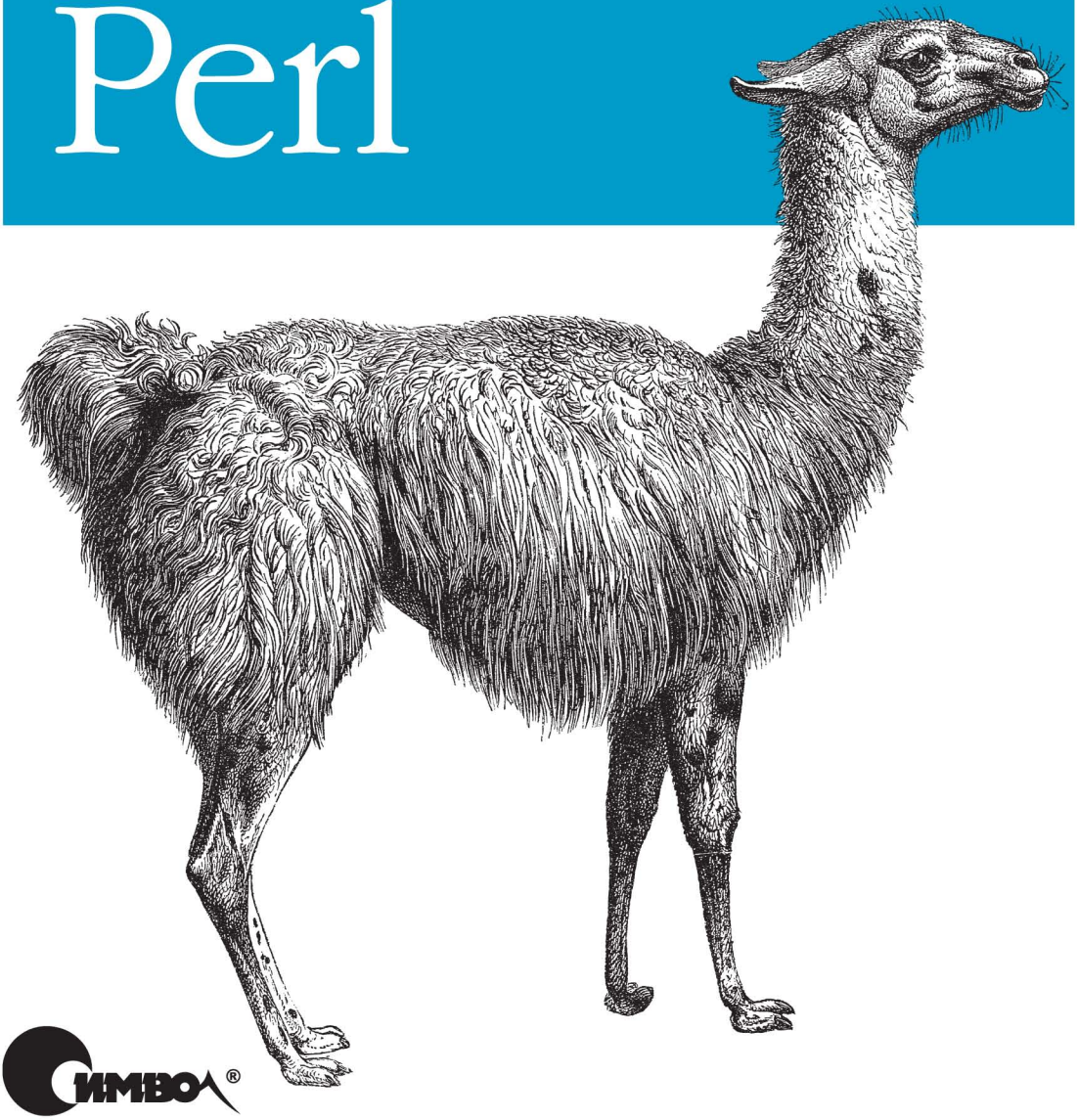
Как сделать сложное простым и невозможное возможным

5-е издание
Включает Perl 5.10

Изучаем Perl



Символ®
O'REILLY®

Рэндал Шварц,
Том Феникс и брайан д фой

Learning Perl

Fifth edition

Randal L. Schwartz,
Tom Phoenix & brian d foy

O'REILLY®

Изучаем Perl

Пятое издание

Рэндал Л. Шварц,
Том Феникс и брайан д фой

Санкт-Петербург — Москва
2009

Рэндал Л. Шварц, Том Феникс и брайан д фой

Изучаем Perl, 5-е издание

Перевод Е. Матвеева

Главный редактор	*А. Галунов*
Зав. редакцией	*Н. Макарова*
Выпускающий редактор	*А. Пасечник*
Редактор	*А. Петухов*
Корректор	*О. Макарова*
Верстка	*Д. Орлова*

Шварц Р., Феникс Т., брайан д фой
Изучаем Perl, 5-е издание. – Пер. с англ. – СПб: Символ-Плюс, 2009. – 384 с., ил.
ISBN: 978-5-93286-154-7

Известный как «книга с ламой», этот учебник, впервые изданный в 1993 году, выходит уже пятым изданием, в котором описываются последние изменения в языке вплоть до версии Perl 5.10.

В пятое издание вошли такие темы, как типы данных и переменные Perl, пользовательские функции, операции с файлами, регулярные выражения, операции со строками, списки и сортировка, управление процессами, умные сравнения, модули сторонних разработчиков и другие.

Perl – язык для тех, кто хочет быстро и эффективно выполнять свою работу. Некогда создававшийся как инструмент для сложной обработки текстов, предназначенный для системных администраторов, сейчас Perl является полнофункциональным языком программирования, подходящим для решения практически любых задач на почти любой платформе – от коротких служебных программ, умещающихся в командной строке, до задач веб-программирования, исследований в области биоинформатики, финансовых расчетов и многого другого.

Иные книги учат вас программировать на Perl, в то время как книга «Изучаем Perl» сделает из вас Perl-программиста.

ISBN: 978-5-93286-154-7
ISBN: 978-0-596-52010-6 (англ)

Издательство «Символ-Плюс». 199034, Санкт-Петербург, 16 линия, 7,
тел. (812) 324-5353, www.symbol.ru. Лицензия ЛП N 000054 от 25.12.98.

Подписано в печать 27.05.2009. Формат 70×100$^{1}/_{16}$. Печать офсетная.
Объем 24 печ. л. Тираж 1500 экз. Заказ №
Отпечатано с готовых диапозитивов в ГУП «Типография «Наука»
199034, Санкт-Петербург, 9 линия, 12.

Оглавление

Предисловие

Перед вами пятое издание книги «Learning Perl», обновленное для версии Perl 5.10 с учетом ее последних новшеств. Впрочем, книга пригодится и тем, кто продолжает использовать Perl 5.6 (хотя эта версия вышла уже давно; не думали об обновлении?).

Если вы желаете с пользой потратить первые 30–45 часов программирования на Perl, считайте, что вам повезло. Книга обстоятельно, без спешки знакомит читателя с языком, который является «рабочей лошадкой» Интернета, – языку Perl отдают предпочтение системные администраторы, веб-хакеры и рядовые программисты по всему миру.

Невозможно в полной мере изложить весь Perl лишь за несколько часов. Книги, которые обещают нечто подобное, явно преувеличивают. Вместо этого мы тщательно отобрали важное подмножество Perl, которое следует изучить. Оно ориентировано на написание программ от 1 до 128 строк, а это примерно 90% реально используемых программ. А когда вы будете готовы продолжить изучение языка, переходите к книге «Intermediate Perl»[1], материал которой начинается с того места, на котором завершается данная книга. Также мы приводим некоторые рекомендации для дальнейшего изучения.

Объем каждой главы таков, чтобы ее можно было прочесть за один-два часа. Каждая глава завершается серией упражнений, которые помогут усвоить материал; ответы к упражнениям приведены в приложении А. Поэтому книга отлично подходит для учебных семинаров типа «Начальный курс Perl». Мы уверены в этом, потому что материал книги почти слово в слово позаимствован из нашего учебного курса, прочитанного тысячам студентов по всему миру. Тем не менее мы создавали эту книгу и для тех, кто будет учиться самостоятельно.

Perl позиционируется как «инструментарий для UNIX», но для чтения этой книги не обязательно быть знатоком UNIX (более того, не обязательно даже работать в UNIX). Если в тексте прямо не указано обратное, весь материал в равной степени относится к Windows ActivePerl от ActiveState, а также практически к любой современной реализации Perl.

---

[1] Рэндал Шварц, брайан д фой и Том Феникс «Perl: изучаем глубже». – Пер. с англ. – СПб.: Символ-Плюс, 2007.

В принципе читатель может вообще ничего не знать о Perl, но мы рекомендуем хотя бы в общих чертах познакомиться с базовыми концепциями программирования: переменными, циклами, подпрограммами и массивами, а также крайне важной концепцией «редактирования файла с исходным кодом в текстовом редакторе». Мы не будем тратить время на объяснение этих концепций. Многочисленные отклики людей, которые выбирали эту книгу и успешно осваивали Perl как свой первый язык программирования, конечно, радуют, но мы не можем гарантировать те же результаты всем читателям.

Условные обозначения

В книге используются следующие условные обозначения:

`Моноширинный шрифт`

Имена методов, функций, переменных и атрибутов. Также используется в примерах программного кода.

`Моноширинный полужирный шрифт`

Данные, вводимые пользователем.

`Моноширинный курсив`

Текст, который должен заменяться пользовательскими значениями (например, *`имя_файла`* заменяется фактическим именем файла).

Курсив

Имена файлов, URL-адреса, имена хостов, команды в тексте, важные термины при первом упоминании, смысловые выделения.

Сноски

Примечания, которые *не следует* читать при первом (а может быть, при втором и третьем) прочтении книги. Иногда мы попросту привираем в тексте ради простоты изложения, а сноски восстанавливают истину. Часто в сносках приводится нетривиальный материал, который в других местах книги даже не рассматривается.

Как с нами связаться

Мы сделали все возможное для того, чтобы проверить приведенную в книге информацию, однако не можем полностью исключить вероятность ошибок в тексте и постоянство Perl. Пожалуйста, сообщайте нам обо всех найденных ошибках, а также делитесь пожеланиями для будущих изданий по адресу:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (США или Канада)

707-829-0515 (международные или местные звонки)
707 829-0104 (факс)

С издательством также можно связаться по электронной почте. Чтобы подписаться на наш список рассылки или заказать каталог, напишите по адресу

info@oreilly.com

С комментариями и техническими вопросами по поводу книги обращайтесь по электронной почте:

bookquestions@oreilly.com

На сайте издательства имеется веб-страница книги со списками обнаруженных опечаток, примерами и всей дополнительной информацией. На ней также можно загрузить текстовые файлы (и пару программ Perl), которые будут полезны (хотя и не обязательны) при выполнении некоторых упражнений. Страница доступна по адресу:

http://www.oreilly.com/catalog/9780596520106

За дополнительной информацией о книгах и по другим вопросам обращайтесь на сайт O'Reilly:

http://www.oreilly.com

Использование примеров кода

Эта книга написана для того, чтобы помочь вам в решении конкретных задач. В общем случае вы можете использовать приводимые примеры кода в своих программах и документации. Связываться с авторами для получения разрешения не нужно, если только вы не воспроизводите значительный объем кода. Например, если ваша программа использует несколько фрагментов кода из книги, обращаться за разрешением не нужно. С другой стороны, для продажи или распространения дисков с примерами из книг O'Reilly потребуется разрешение. Если вы отвечаете на вопрос на форуме, приводя цитату из книги с примерами кода, обращаться за разрешением не нужно. Если же значительный объем кода из примеров книги включается в документацию по вашему продукту, разрешение необходимо. Мы будем признательны за ссылку на источник информации, хотя и не требуем этого. Обычно в ссылке указывается название, автор, издательство и ISBN, например: «*Learning Perl*, Fifth edition, by Randal L. Schwartz, Tom Phoenix, and brian d foy. Copyright 2008 O'Reilly Media, Inc., 978-0-596-52010-6». Если вы полагаете, что ваши потребности выходят за рамки оправданного использования примеров кода или разрешений, приведенных выше, свяжитесь с нами по адресу *permissions@oreilly.com*.

Safari® Enabled

Если на обложке технической книги есть пиктограмма «Safari® Enabled», это означает, что книга доступна в Сети через O'Reilly Network Safari Bookshelf.

Safari предлагает намного лучшее решение, чем электронные книги. Это виртуальная библиотека, позволяющая без труда находить тысячи лучших технических книг, вырезать и вставлять примеры кода, загружать главы и находить быстрые ответы, когда требуется наиболее верная и свежая информация. Safari свободно доступна по адресу *http://safari.oreilly.com*.

История появления книги

Для самых любознательных читателей Рэндал расскажет, как эта книга появилась на свет.

После того как я закончил первую книгу «Programming Perl»[1] с Ларри Уоллом (в 1991 году), компания Taos Mountain Software из Кремниевой долины предложила мне разработать учебный курс. Предложение включало проведение первого десятка занятий и обучение персонала для их продолжения. Я написал[2] и предоставил им обещанный курс.

Во время третьего или четвертого изложения курса (в конце 1991 года) ко мне обратился один слушатель, который сказал: «Знаете, мне нравится «Programming Perl», но материал этого курса гораздо лучше воспринимается – вам стоило бы написать книгу». Я решил, что это хорошая мысль, и начал действовать.

Я написал Тиму O'Рейли предложение. Общая структура материала походила на курс, который был создан для Taos, однако я переставил и изменил некоторые главы на основании своих наблюдений в учебных аудиториях. Мое предложение было принято в рекордные сроки – через 15 минут я получил от Тима сообщение, которое гласило: «А мы как раз хотели предложить вторую книгу – «Programming Perl» идет нарасхват». Работа продолжалась 18 месяцев и завершилась выходом первого издания «Learning Perl».

К этому времени возможности преподавания Perl появились и за пределами Кремниевой долины[3], поэтому я разработал новый учебный

---

[1] Ларри Уолл и др. «Программирование на Perl», 3-е издание. – Пер. с англ. – СПб.: Символ-Плюс, 2002.

[2] В контракте я сохранил за собой права на упражнения, надеясь когда-нибудь использовать их в своих целях, например в журнальных рубриках, которые я вел в то время. Упражнения – единственное, что перешло в эту книгу из курса Taos.

[3] Мой контракт с Taos включал пункт о недопустимости конкуренции, поэтому мне приходилось держаться подальше от Кремниевой долины с аналогичными курсами. Я соблюдал этот пункт в течение многих лет.

курс на основании текста, написанного для «Learning Perl». Я провел десяток семинаров для разных клиентов (включая моего основного работодателя Intel Oregon) и использовал полученную обратную связь для дальнейшего уточнения проекта книги.

Первое издание вышло в свет в первый день ноября[1] 1993 года и имело оглушительный успех. По темпам продаж книга обогнала даже «Programming Perl».

На обложке первой книги говорилось: «Написано известным преподавателем Perl». Фраза оказалась пророческой. В ближайшие месяцы я начал получать сообщения со всех уголков США, в которых мне предлагалось провести обучение «на месте». За следующие семь лет моя компания заняла ведущие позиции в области обучения Perl на территории заказчика, и я лично набрал (не преувеличиваю!) скидку на миллион миль по программе частых авиаперелетов. В то время начиналось становление Web и веб-мастера выбирали Perl для управления контентом, взаимодействия с CGI и сопровождения.

После двухлетнего тесного сотрудничества с Томом Фениксом, старшим преподавателем и менеджером по подбору материала для Stonehenge, я предложил ему поэкспериментировать с курсом «ламы», порядком изложения и структурой материала. Когда мы совместными усилиями создали то, что сочли лучшей обновленной версией курса, я обратился в O'Reilly и сказал: «Пора выпускать новую книгу!» Так появилось третье издание.

Через два года после выхода третьего издания «книги с ламой» мы с Томом решили, что пора оформить в виде книги наш «расширенный» курс для разработчиков программ «от 100 до 10 000 строк кода». Вместе мы создали первую «книгу с альпакой», выпущенную в 2003 году.

Но в это время с войны в Персидском заливе вернулся наш коллега – преподаватель брайан д фой. Он заметил, что часть материала в обеих книгах стоило бы переписать, потому что наш курс еще не в полной мере соответствовал изменяющимся потребностям типичного студента. Он предложил O'Reilly в последний раз (как мы надеемся) переписать «книгу с ламой» и «книгу с альпакой» перед выходом Perl 6. Внесенные изменения отражены в пятом издании книги. В действительности основную работу проделал брайан под моим общим руководством; он превосходно справился с непростой работой «пастуха кошек», с которой часто сравнивают коллективную работу над книгой.

18 декабря 2007 года группа perl5porters выпустила Perl 5.10 – очередную версию Perl с несколькими принципиально новыми возможностя-

[1] Я очень хорошо помню эту дату, потому что в этот же день меня арестовали дома за деятельность, связанную с применением компьютеров по поводу моего контракта с Intel. Мне была предъявлена серия обвинений, по которым я был позднее осужден.

ми. В предыдущей версии 5.8 центральное место занимало «доведение до ума» поддержки Юникода. Последняя версия дополнила стабильную основу 5.8 рядом новшеств, часть из которых была позаимствована из Perl 6 (еще не выпущенного). Некоторые усовершенствования (например, именованное сохранение в регулярных выражениях) значительно удобнее старых решений, а следовательно, идеально подходят для новичков. Тогда мы еще не думали о пятом издании книги, но Perl 5.10 был настолько хорош, что мы не устояли.

Некоторые отличия от предыдущих изданий:

- Текст изменен с учетом специфики последней версии Perl 5.10. Часть приводимого кода работает только в этой версии. На использование возможностей Perl 5.10 явно указано в тексте, и такие части кода помечаются специальной директивой `use`. Она гарантирует использование правильной версии:

  ```
  use 5.010; # Для выполнения сценария требуется Perl 5.10 и выше
  ```

  Если директива `use 5.010` отсутствует, значит, код должен работать и в Perl 5.6. Версию Perl можно узнать при помощи ключа командной строки `-v`:

  ```
  promt% perl -v
  ```

Далее перечислены некоторые нововведения Perl 5.10, представленные в книге. По возможности мы также опишем старые способы решения тех же задач:

- Главы, посвященные регулярным выражениям, дополнены новыми возможностями Perl 5.10: относительными обратными ссылками (глава 7), новыми символьными классами (глава 7) и именованным сохранением (глава 8).
- В Perl 5.10 появился аналог `switch` – конструкция `given-when`. Она рассматривается в главе 15 вместе с оператором умного сравнения.
- В пользовательских функциях теперь могут употребляться статические переменные (в Perl они обозначаются ключевым словом `state`). Такие переменные сохраняют свои значения между вызовами функции и обладают лексической областью видимости. Статические переменные рассматриваются в главе 4.

Благодарности

От Рэндала. Хочу поблагодарить всех преподавателей Stonehenge – как бывших, так и действующих. Джозеф Холл (Joseph Hall), Том Феникс (Tom Phoenix), Чип Салзенберг (Chip Salzenberg), брайан д фой (brian d foy) и Тэд Мак-Клеллан (Tad McClellan), спасибо за вашу готовность неделю за неделей приходить в классы и вести занятия. Ваши наблюдения за тем, что работает, а что нет, позволили нам лучше приспособить материал книги к потребностям читателя. Хочу особенно поблагода-

рить моего соавтора и делового партнера Тома Феникса за бесчисленные часы, потраченные на совершенствование курса «ламы», и за написание отличного текста для большей части книги. Спасибо брайану д фою за то, что он взял на себя основную работу над четвертым изданием и избавил меня от этого вечного пункта в списке незавершенных дел.

Я благодарен всему коллективу O'Reilly, особенно нашему терпеливому редактору и руководителю проекта предыдущего издания Эллисон Рэндал (Allison Randal) (не родственник, но фамилия хорошая), а также самому Тиму О'Рейли за то, что он когда-то доверил мне работу над «книгой с верблюдом» и «книгой с ламой».

Я также в неоплатном долгу перед тысячами людей, купивших предыдущие издания «книги с ламой», перед моими студентами, научившими меня лучше преподавать, и перед нашими многочисленными клиентами из списка «Fortune 1000», которые заказывали наши семинары в прошлом и намерены это делать в будущем.

Как всегда, я особенно благодарен Лайлу и Джеку – они научили меня практически всему, что относится к написанию книг. Парни, я вас никогда не забуду.

От Тома. Присоединяюсь к благодарностям Рэндала, адресованным персоналу O'Reilly. В третьем издании книги нашим редактором была Линда Муи (Linda Mui); спасибо ей за терпение, с которым она указывала на неуместные шуточки и сноски (за те, что остались, ее винить не следует). Вместе с Рэндалом она руководила моей работой над книгой, я благодарен им обоим. В четвертом издании должность редактора заняла Эллисон Рэндал; спасибо и ей.

Я также присоединяюсь к словам Рэндала, обращенным к другим преподавателям Stonehenge. Они почти не жаловались на неожиданные изменения материала курсов для опробования новой учебной методики. Они представили столько разных точек зрения на методологию обучения, которые мне и в голову не могли придти.

Долгие годы я работал в Орегонском музее науки и промышленности (OMSI). Благодарю всех работников музея за то, что они позволяли мне отточить мои преподавательские навыки.

Спасибо всем коллегам по Usenet за поддержку и высокую оценку моего вклада. Надеюсь, мои старания оправдают ваши ожидания.

Благодарю своих многочисленных студентов, которые обращались ко мне с вопросами (и недоуменными взглядами), когда мне нужно было опробовать новый способ представления той или иной концепции. Надеюсь, это издание устранит все оставшиеся недоразумения.

Конечно, я особенно глубоко признателен своему соавтору Рэндалу за свободу в выборе представления материала – как в учебных классах, так и в книге (а также за саму идею преобразовать этот материал в книгу). И конечно, я должен сказать, что меня до сих пор вдохновляет

твоя неустанная работа, направленная на то, чтобы у других не возникали юридические неприятности, которые отняли у тебя столько времени и энергии; с тебя следует брать пример.

Спасибо моей жене Дженне за терпение и за все остальное.

От брайана. Прежде всего я должен поблагодарить Рэндала, потому что я изучал Perl по первому изданию этой книги, а затем мне пришлось изучать его снова, когда Рэндал предложил мне заняться преподаванием в Stonehenge в 1998 году. Преподавание нередко оказывается лучшим способом изучения. С того времени Рэндал учил меня не только Perl, но и другим вещам, которые, как он считал, мне необходимо знать – например, когда он решил, что для демонстрации на вебконференции нам стоит использовать Smalltalk вместо Perl. Меня всегда поражала широта его познаний. Именно Рэндал впервые предложил мне писать книги о Perl. А теперь я помогаю работать над книгой, с которой все начиналось. Это большая честь для меня, Рэндал.

Наверное, за все время работы в Stonehenge я видел Тома Феникса в общей сложности недели две, но его версию учебного курса я преподавал годами. Эта версия преобразовалась в третье издание книги. Объясняя ее своим студентам, я обнаружил новые способы представления почти всех концепций и открыл незнакомые мне уголки Perl.

Когда я убедил Рэндала, что смогу помочь в обновлении «книги с ламой», я получил от него благословение на составление предложения, отбор материала и контроль версий. Наш редактор четвертого издания, Эллисон Рэндал, помогла мне вжиться во все эти роли и стойко переносила мои многочисленные вопросы.

Спасибо Стейси, Бастеру, Мими, Роско, Амелии, Лили и всем остальным, кто пытался немного развлечь меня, когда я был занят, но продолжал разговаривать со мной, даже если я не выходил поиграть.

От всех нас. Спасибо нашим рецензентам: Дэвиду Адлеру (David H. Adler), Энди Армстронгу (Andy Armstrong), Дэйву Кроссу (Dave Cross), Крису Деверсу (Chris Devers), Полу Фенвику (Pau Fenwick), Стивену Дженкинсу (Stephen Jenkins), Мэттью Мусгрову (Matthew Musgrove), Стиву Питерсу (Steve Peters) и Уилу Уитону (Wil Wheaton) за комментарии по поводу предварительного проекта книги.

Также спасибо нашим многочисленным студентам, которые помогали нам понять, какие части учебного курса нуждаются в усовершенствовании. Благодаря вам мы сегодня можем гордиться этой книгой.

Спасибо участникам группы Perl Mongers – посещая их города, мы чувствовали себя как дома. Давайте как-нибудь повторим?

И наконец, мы искренне благодарны нашему другу Ларри Уоллу, который поделился своей классной, мощной «игрушкой» со всем миром. Это позволило нам выполнять свою работу чуть быстрее и проще... и чуть интереснее.

9

Обработка текста с использованием регулярных выражений

Регулярные выражения также могут использоваться для изменения текста. До настоящего момента мы рассматривали поиск по шаблону, а в этой главе вы увидите, как шаблоны применяются для обнаружения изменяемых частей строки.

Замена с использованием оператора s///

Если оператор поиска `m//` напоминает функцию поиска в текстовом редакторе, то оператор замены Perl `s///` может рассматриваться как аналог функции поиска с заменой. Оператор просто заменяет часть значения переменной[1], которая совпала с шаблоном, заданной строкой:

```
$_ = "He's out bowling with Barney tonight.";
s/Barney/Fred/;  # Заменить Barney на Fred
print "$_\n";
```

Если совпадение не найдено, ничего не происходит и содержимое переменной остается без изменений:

```
# Продолжение; $_ содержит "He's out bowling with Fred tonight."
s/Wilma/Betty/;  # Заменить Wilma на Betty (не получится)
```

[1] В отличие от оператора `m//`, который может применяться практически к любому строковому выражению, оператор `s///` модифицирует данные, поэтому он должен применяться к *левостороннему значению* (*lvalue*). Это почти всегда переменная, хотя формально допустима любая конструкция, которая может использоваться в левой части оператора присваивания.

Конечно, и шаблон, и замена могут быть более сложными. В данном примере строка замены использует первую нумерованную переменную $1, которая задается при поиске совпадения:

```
s/with (\w+)/against $1's team/;
print "$_\n";  # Содержит "He's out bowling against Fred's team tonight."
```

Еще несколько примеров замены (конечно, это всего лишь примеры – в реальных программах выполнение такого количества несвязанных замен подряд нехарактерно):

```
$_ = "green scaly dinosaur";
s/(\w+) (\w+)/$2, $1/;  # "scaly, green dinosaur"
s/^/huge, /;            # "huge, scaly, green dinosaur"
s/,.*een//;             # Пустая замена: "huge dinosaur"
s/green/red/;           # Неудачный поиск: все еще "huge dinosaur"
s/\w+$/($`!)$&/;        # "huge (huge !)dinosaur"
s/\s+(!\W+)/$1 /;       # "huge (huge!) dinosaur"
s/huge/gigantic/;       # "gigantic (huge!) dinosaur"
```

Оператор s/// возвращает полезный логический признак; если замена была выполнена успешно, возвращается *true*, а в случае неудачи возвращается *false*:

```
$_ = "fred flintstone";
if (s/fred/wilma/) {
  print "Successfully replaced fred with wilma!\n";
}
```

Глобальная замена (/g)

Как вы, вероятно, заметили в предыдущих примерах, s/// выполняет только одну замену, даже если в строке возможны и другие совпадения. Конечно, это всего лишь режим по умолчанию. Модификатор /g требует, чтобы оператор s/// выполнял все возможные неперекрывающиеся[1] замены:

```
$_ = "home, sweet home!";
s/home/cave/g;
print "$_\n";  # "cave, sweet cave!"
```

Глобальная замена часто используется для свертки пропусков, то есть замены всех серий разных пропусков одним пробелом:

```
$_ = "Input  data\t may have    extra whitespace.";
s/\s+/ /g;  # "Input data may have extra whitespace."
```

Удаление начальных и завершающих пропусков также выполняется достаточно легко:

---

[1] Замены должны выполняться без перекрытия, потому что поиск очередного совпадения начинается с позиции, следующей непосредственно за последней заменой.

```
s/^\s+//;  # Замена начальных пропусков пустой строкой
s/\s+$//;  # Замена завершающих пропусков пустой строкой
```

То же можно было бы выполнить одной командой с альтернативой и флагом /g, но такое решение работает чуть медленнее (по крайней мере, на момент написания книги). Впрочем, ядро регулярных выражений постоянно оптимизируется; информацию о том, из-за чего регулярные выражения работают быстро (или медленно), можно найти в книге Джеффри Фридла (Jeffrey Friedl) «Mastering Regular Expressions»[1] (O'Reilly).

```
s/^\s+|\s+$//g;  # Удаление начальных и завершающих пропусков
```

Другие ограничители

По аналогии с конструкциями m// и qw// для s/// также можно выбрать другой ограничитель. Но в операторе замены используются не два, а три ограничителя, поэтому ситуация слегка изменяется.

Для обычных (непарных) символов, не имеющих «левой» и «правой» разновидности, просто используйте три одинаковых символа, как мы поступали с /. В следующей команде в качестве ограничителя выбран символ решетки (#):

```
s#^https://#http://#;
```

Для парных символов необходимо использовать две пары: в одну пару заключается шаблон, а в другую – строка замены. В этом случае строка и шаблон даже могут заключаться в разные ограничители. Более того, ограничители строки даже могут быть непарными! Следующие команды делают одно и то же:

```
s{fred}{barney};
s[fred](barney);
s<fred>#barney#;
```

Модификаторы режимов

Кроме уже упоминавшегося модификатора /g[2], при замене могут использоваться модификаторы /i, /x и /s, уже знакомые по обычному поиску совпадений (порядок перечисления модификаторов неважен):

```
s#wilma#Wilma#gi;  # Заменить все вхождения WiLmA или WILMA строкой Wilma
s{__END__.*}{}s;   # Удалить конечный маркер и все последующие строки
```

---

[1] Джеффри Фридл «Регулярные выражения», 3-е издание. – Пер. с англ. – СПб.: Символ-Плюс, 2008.

[2] Обычно мы говорим о модификаторах с именами вида /i, даже если в качестве ограничителя используется другой символ.

Оператор привязки

По аналогии с оператором m// для оператора s/// также можно выбрать другую целевую строку при помощи оператора привязки:

```
$file_name =~ s#^.*/##s;  # Удалить из $file_name путь "в стиле UNIX"
```

Переключение регистра

При замене часто бывает нужно обеспечить правильный регистр символов в заменяющем слове (нижний или верхний в зависимости от ситуации). Задача легко решается в Perl при помощи служебных последовательностей с обратной косой чертой. Комбинация \U преобразует все последующие символы к верхнему регистру:

```
$_ = "I saw Barney with Fred.";
s/(fred|barney)/\U$1/gi;  # $_ теперь содержит "I saw BARNEY with FRED."
```

Аналогичная комбинация \L обеспечивает преобразование к нижнему регистру. Продолжение предыдущего фрагмента:

```
s/(fred|barney)/\L$1/gi;  # $_ теперь содержит "I saw barney with fred."
```

По умолчанию преобразование распространяется на остаток (заменяющей) строки; также можно вручную отменить переключение регистра комбинацией \E:

```
s/(\w+) with (\w+)/\U$2\E with $1/i;  # $_ содержит "I saw FRED with barney."
```

При записи в нижнем регистре (\l и \u) эти комбинации влияют только на следующий символ:

```
s/(fred|barney)/\u$1/ig;  # $_ теперь содержит "I saw FRED with Barney."
```

Они даже могут использоваться в сочетании друг с другом. Например, объединение \u с \L означает: «все в нижнем регистре, но первая буква в верхнем регистре»:[1]

```
s/(fred|barney)/\u\L$1/ig;  # $_ теперь содержит "I saw Fred with Barney."
```

Хотя сейчас мы рассматриваем переключение регистра в связи с заменой, эти комбинации работают в любой строке, заключенной в кавычки:

```
print "Hello, \L\u$name\E, would you like to play a game?\n";
```

[1] \L и \u могут следовать друг за другом в любом порядке. Ларри предвидел, что некоторые люди все равно перепутают порядок; его стараниями Perl «догадывается», что в верхнем регистре должна записываться только первая буква, а все остальные буквы должны записываться в нижнем регистре. Весьма любезно с его стороны.

Оператор split

Регулярные выражения также используются в работе оператора `split`, который разбирает строку в соответствии с заданным шаблоном. Например, такая возможность пригодится при обработке данных, разделенных символами табуляции, двоеточиями, пропусками... и вообще *чем угодно*.[1] Оператор `split` может использоваться в любых ситуациях, когда возможно задать разделитель данных (вообще говоря, он представляет собой простое регулярное выражение). Синтаксис вызова выглядит так:

```
@fields = split /разделитель/, $строка;
```

Оператор `split` ищет совпадения шаблона в строке и возвращает список полей (подстрок), разделенных совпадениями. Каждое совпадение шаблона отмечает конец одного и начало следующего поля. Таким образом, текст, совпадающий с шаблоном, никогда не включается в возвращаемые поля. Типичный вызов `split` с разделением по двоеточиям:

```
@fields = split /:/, "abc:def:g:h";  # Получаем ("abc", "def", "g", "h")
```

Если строка содержит два смежных вхождения разделителя, список даже может содержать пустое поле:

```
@fields = split /:/, "abc:def::g:h";  # Получаем ("abc", "def", "", "g", "h")
```

Следующее правило на первый взгляд кажется странным, но редко создает проблемы: начальные пустые поля всегда возвращаются, а завершающие пустые поля отбрасываются. Пример:[2]

```
@fields = split /:/, ":::a:b:c:::";  # Получаем ("", "", "", "a", "b", "c")
```

Операция разбивки также часто выполняется по пропускам с использованием шаблона /\s+/. При таком шаблоне все серии пропусков считаются эквивалентными одному пробелу:

```
my $some_input = "This  is a \t        test.\n";
my @args = split /\s+/, $some_input;  # ("This", "is", "a", "test.")
```

По умолчанию `split` разбивает `$_` по пропускам:

```
my @fields = split;  # Эквивалентно split /\s+/, $_;
```

Это почти то же самое, что разбиение по шаблону /\s+/, если не считать того, что в этом особом случае начальные пустые поля подавляются.

---

[1] Кроме данных, разделенных запятыми (часто называемых CSV – Comma Separated Values). Работать с ними оператором `split` весьма неудобно; лучше воспользоваться модулем `Text::CSV` из архива CPAN.

[2] Впрочем, это всего лишь действие по умолчанию, применяемое для повышения эффективности. Если вы не хотите терять завершающие пустые поля, передайте при вызове `split` в третьем аргументе `-1`; см. man-страницу *perlfunc*.

Таким образом, если строка начинается с пропуска, вы не увидите пустое поле в начале списка. (Если вы захотите реализовать аналогичное поведение при разбиении другой строки по пропускам, замените шаблон одним пробелом: `split ' ', $other_string`. Использование пробела вместо шаблона является особым случаем вызова `split`.)

Обычно для разбиения используются простые шаблоны вроде приведенных выше. Если шаблон усложняется, постарайтесь не включать в него сохраняющие круглые скобки, потому что это приводит к активизации «режима включения разделителей» (за подробностями обращайтесь к man-странице *perlfunc*). Если вам потребуется сгруппировать элементы выражения в шаблоне `split`, используйте несохраняющие круглые скобки `(?:)`.

Функция join

Функция `join` не использует шаблоны, но выполняет операцию, обратную по отношению к `split`. Если `split` разбивает строку на фрагменты, `join` «склеивает» набор фрагментов в одну строку. Вызов `join` выглядит так:

```
my $result = join $glue, @pieces;
```

Первый аргумент `join` определяет произвольную соединительную строку. Остальные аргументы определяют список фрагментов. Функция `join` вставляет соединительную строку между парами фрагментов и возвращает итоговую строку:

```
my $x = join ":", 4, 6, 8, 10, 12;  # $x содержит "4:6:8:10:12"
```

В этом примере объединяются пять фрагментов, поэтому между ними вставляются четыре соединительные строки (двоеточия). Соединительная строка добавляется только между элементами, но не в начале или в конце списка. Это означает, что количество соединительных строк всегда на 1 меньше количества элементов в списке.

Если список не содержит хотя бы двух элементов, соединительных строк вообще не будет:

```
my $y = join "foo", "bar";       # Получается только "bar",
                                 # соединитель foo лишний
my @empty;                       # Пустой массив
my $empty = join "baz", @empty;  # Элементов нет, пустая строка
```

Используя переменную `$x` из предыдущего примера, мы можем разбить строку и собрать ее заново с другим разделителем:

```
my @values = split /:/, $x;  # @values содержит (4, 6, 8, 10, 12)
my $z = join "-", @values;   # $z содержит "4-6-8-10-12"
```

Хотя `split` и `join` хорошо работают в сочетании друг с другом, не забывайте, что первый аргумент `join` всегда содержит строку, а не шаблон.

m// в списочном контексте

При использовании split шаблон задает разделитель (части строки, не содержащие полезных данных). Иногда бывает проще задать тот текст, который вы хотите сохранить.

При использовании оператора поиска по шаблону (m//) в списочном контексте возвращаемое значение представляет собой список переменных, созданных для совпадений, или пустой список (если поиск завершился неудачей):

```
$_ = "Hello there, neighbor!";
my($first, $second, $third) = /(\S+) (\S+), (\S+)/;
print "$second is my $third\n";
```

Это позволяет легко присвоить переменным понятные удобные имена, которые будут действовать и после следующего поиска по шаблону. (Также обратите внимание на то, что в коде отсутствует оператор =~, поэтому шаблон по умолчанию применяется к $_.)

Модификатор /g, впервые встретившийся нам при описании s///, работает и с m//; он позволяет найти совпадение в нескольких местах строки. В этом случае шаблон с парой круглых скобок будет возвращать элемент списка для каждого найденного совпадения:

```
my $text = "Fred dropped a 5 ton granite block on Mr. Slate";
my @words = ($text =~ /([a-z]+)/ig);
print "Result: @words\n";
# Result: Fred dropped a ton granite block on Mr Slate
```

Происходящее напоминает «split наоборот»: вместо того что необходимо удалить, мы указываем, что необходимо оставить.

Если шаблон содержит несколько пар круглых скобок, для каждого совпадения может возвращаться более одной строки. Предположим, у нас имеется строка, которую необходимо прочитать в хеш:

```
my $data = "Barney Rubble Fred Flintstone Wilma Flintstone";
my %last_name = ($data =~ /(\w+)\s+(\w+)/g);
```

При каждом совпадении шаблона возвращаются два частичных совпадения. Эти два совпадения образуют пары «ключ-значение» в создаваемом хеше.

Другие возможности регулярных выражений

После чтения трех (почти!) глав, посвященных регулярным выражениям, становится ясно, что этот мощный механизм принадлежит к числу важнейших аспектов Perl. Но ядро регулярных выражений Perl обладает и другими возможностями, добавленными в него создателями Perl; самые важные из них представлены в этом разделе. Заодно вы кое-что узнаете о том, как работает ядро регулярных выражений.

Минимальные квантификаторы

Четыре квантификатора, встречавшиеся нам ранее (в главах 7 и 8), являются *максимальными* (*greedy*). Это означает, что они находят совпадение максимально возможной длины и неохотно «уступают» символы только в том случае, если это необходимо для общего совпадения шаблона. Пример: допустим, шаблон `/fred.+barney/` применяется к строке `fred and barney went bowling last night`. Понятно, что совпадение будет найдено, но давайте посмотрим, как это происходит.[1] Сначала, конечно, элемент шаблона `fred` совпадает с идентичной литеральной строкой. Далее следует элемент `.+`, который совпадает с любым символом, кроме символа новой строки, не менее одного раза. Но квантификатор `+` «жаден»: он пытается захватить как можно больше символов. Поэтому он немедленно пытается найти совпадение для всей оставшейся строки, включая слово `night`. (Как ни странно, поиск совпадения на этом не закончен.)

Теперь ядру хотелось бы найти совпадение для элемента `barney`, но сделать это не удается – мы находимся в конце строки. Но т. к. совпадение `.+` будет считаться успешным даже в случае, если оно станет на один символ короче, `+` неохотно уступает букву `t` в конце строки. (Несмотря на «жадность», квантификатор стремится к совпадению всего выражения даже больше, чем к захвату максимального числа символов.)

Снова ищется совпадение для элемента `barney`, и снова поиск оказывается неудачным. Элемент `.+` уступает букву `h` и делает очередную попытку. Так, символ за символом `.+` отдает весь захваченный фрагмент, пока в какой-то момент не уступит все буквы `barney`. Теперь для элемента `barney` находится совпадение и общий поиск завершается удачей.

Ядро регулярных выражений постоянно «отступает» подобным образом, перебирая все возможные варианты размещения шаблона, пока один из них не завершится удачей или станет ясно, что совпадение невозможно.[2] Но как видно из рассмотренного примера, поиск может потребовать большого количества «отступлений», потому что квантификатор захватывает слишком большую часть строки, а ядро регулярных выражений заставляет его вернуть часть символов.

---

[1] Ядро регулярных выражений применяет некоторые оптимизации, из-за которых процесс поиска немного отличается от описанного в тексте, причем эти оптимизации изменяются в зависимости от версии Perl. Впрочем, на функциональности они не отражаются. Если вы хотите досконально выяснить, как именно все работает, читайте исходный код последней версии. И не забудьте исправить все найденные ошибки!

[2] Более того, некоторые ядра регулярных выражений продолжают поиск даже *после* обнаружения совпадения! Но ядро регулярных выражений Perl интересует только сам факт существования совпадения; обнаружив совпадение, оно считает свою работу законченной. Подробности см. в книге Фридла «Регулярные выражения», 3-е издание, Символ-Плюс, 2008.

Однако у каждого максимального («жадного») квантификатора существует парный минимальный квантификатор. Вместо плюса (+) мы можем использовать минимальный квантификатор +?. Как и +, он совпадает один или несколько раз, но при этом довольствуется минимальным количеством символов (вместо максимально возможного). Давайте посмотрим, как работает новый квантификатор в шаблоне /fred.+?barney/.

Совпадение для fred снова находится в самом начале строки. Но на этот раз следующий элемент шаблона .+? старается совпасть с одним символом, поэтому на совпадение с ним проверяется только пробел после fred. Следующий элемент шаблона barney в этой позиции совпасть не может (так как строка в текущей позиции начинается с and barney...). Элемент .+? неохотно добавляет к совпадению a и повторяет попытку. И снова для barney не находится совпадения, поэтому .+? добавляет букву n и т. д. Когда .+? совпадет с пятью символами, для barney будет найдено совпадение, и применение шаблона завершится успешно.

И в этом случае без неудачных попыток не обошлось, но ядру пришлось возвращаться всего несколько раз, что должно обеспечить значительный выигрыш по скорости. Вообще говоря, выигрыш присутствует только в том случае, если barney обычно находится вблизи от fred. Если в ваших данных fred чаще всего находится в начале строки, а barney – в конце, максимальный квантификатор будет работать быстрее. В конечном итоге скорость работы регулярных выражений зависит от данных.

Но достоинства минимальных квантификаторов не ограничиваются эффективностью. Хотя они всегда совпадают (или не совпадают) в тех же строках, что и их максимальные версии, квантификаторы могут совпасть с разными частями строки. Допустим, у нас имеется HTML-подобный[1] текст, из которого необходимо удалить все теги <BOLD> и </BOLD>, оставив их содержимое без изменений. Текст выглядит так:

```
I'm talking about the cartoon with Fred and <BOLD>Wilma</BOLD>!
```

А вот оператор замены для удаления этих тегов. Почему он не подходит?

```
s#(.*)#$1#g;
```

Проблема в максимальности квантификатора *.[2] А если бы текст выглядел так:

---

[1] И снова мы не используем реальный HTML, потому что его невозможно корректно разобрать при помощи простых регулярных выражений. Если вы работаете с HTML или другим языком разметки, воспользуйтесь модулем, который позаботится обо всех сложностях.

[2] Еще одна возможная проблема: нам также следовало бы использовать модификатор /s, потому что начальный и конечный теги могут находиться в разных строках. Хорошо, что это всего лишь пример: в реальной работе мы бы воспользовались собственным советом и взяли хорошо написанный модуль.

```
I thought you said Fred and  <BOLD>Velma</BOLD>, not  <BOLD>Wilma</BOLD>
```

Шаблон совпадет с текстом от первого тега <BOLD> до последнего тега </BOLD>, а все промежуточные теги останутся в строке. Какая неприятность! Вместо максимальных квантификаторов здесь необходимо использовать минимальные квантификаторы. Минимальная форма * имеет вид *?, так что замена принимает вид

```
s#<BOLD>(.*?)</BOLD>#$1#g;
```

И она работает правильно.

Итак, в минимальной версии + превращается в +?, а * в *?. Вероятно, вы уже догадались, как будут выглядеть два других квантификатора. Минимальная форма квантификатора в фигурных скобках выглядит аналогично, но вопросительный знак ставится после закрывающей скобки, например {5,10}? или {8,}?.[1] Даже квантификатор ? существует в минимальной форме: ??. Она совпадает только один раз или не совпадает вовсе, причем второй вариант является предпочтительным.

Многострочный поиск

Классические регулярные выражения использовались для поиска совпадений в пределах одной строки текста. Но поскольку Perl может работать со строками произвольной длины, шаблоны Perl так же легко совпадают в строках, разбитых на несколько логических строк. Например, следующая строка состоит из четырех логических строк:

```
$_ = "I'm much better\nthan Barney is\nat bowling,\nWilma.\n";
```

Якоря ^ и $ обычно привязываются к началу и концу всей строки (см. главу 8). Но флаг /m также позволяет им совпадать в позициях внутренних новых строк. Таким образом, они превращаются в якоря логических, а не физических строк. В приведенном примере совпадение будет найдено:

```
print "Found 'wilma' at start of line\n" if /^wilma\b/im;
```

Многострочная замена выполняется аналогично. В следующем примере весь файл читается в одну переменную[2], после чего имя файла выводится в начале каждой строки:

```
open FILE, $filename
  or die "Can't open '$filename': $!";
my $lines = join '', <FILE>;
$lines =~ s/^/$filename: /gm;
```

---

[1] Теоретически также существует форма минимального квантификатора, определяющая точное количество повторений, вроде {3}?. Но так как она в любом случае совпадет ровно с тремя вхождениями предшествующего элемента, минимальность или максимальность квантификатора ничего не изменит.

[2] Будем надеяться, что файл – а вместе с ним и переменная – относительно невелик.

Обновление нескольких файлов

Программное обновление текстового файла чаще всего реализуется как запись нового файла, похожего на старый, но содержащего все необходимые изменения. Как вы увидите, этот прием приводит почти к такому же результату, как обновление самого файла, но имеет ряд побочных положительных эффектов.

В следующем примере используются сотни файлов, имеющих похожий формат. Один из них, *fred03.dat*, содержит строки следующего вида:

```
Program name: granite
Author: Gilbert Bates
Company: RockSoft
Department: R&D
Phone: +1 503 555-0095
Date: Tues March 9, 2004
Version: 2.1
Size: 21k
Status: Final beta
```

Требуется обновить файл и включить в него другую информацию. После обновления файл должен выглядеть примерно так:

```
Program name: granite
Author: Randal L. Schwartz
Company: RockSoft
Department: R&D
Date: June 12, 2008 6:38 pm
Version: 2.1
Size: 21k
Status: Final beta
```

Коротко говоря, в файл вносятся три изменения. Имя автора (`Author`) заменяется другим именем; дата (`Date`) заменяется текущей датой; телефон (`Phone`) удаляется совсем. Все эти изменения повторяются в сотнях аналогичных файлов.

Perl поддерживает механизм редактирования файлов «на месте» с небольшой дополнительной помощью со стороны оператора `<>`. Далее приводится программа, выполняющая нужные операции, хотя на первый взгляд непонятно, как она работает. В ней появился только один незнакомый элемент – специальная переменная `$^I`. Пока не обращайте на нее внимания, мы вернемся к ней позже.

```
#!/usr/bin/perl -w

use strict;

chomp(my $date = `date`);
$^I = ".bak";

while (<>) {
```

```
    s/^Author:.*/Author: Randal L. Schwartz/;
    s/^Phone:.*\n//;
    s/^Date:.*/Date: $date/;
    print;
}
```

Так как нам нужна сегодняшняя дата, программа начинается с выполнения системной команды *date*. Пожалуй, для получения даты (в слегка ином формате) было бы лучше воспользоваться функцией Perl `localtime` в скалярном контексте:

```
my $date = localtime;
```

В следующей строке задается переменная `$^I`; пока не обращайте на нее внимания.

Список файлов для оператора <> берется из командной строки. Основной цикл читает, обновляет и выводит данные по одной строке. (Если руководствоваться тем, что вам уже известно, все измененное содержимое будет выведено на терминал и моментально промчится мимо ваших глаз, а файлы не изменятся... Но продолжайте читать.) Обратите внимание: второй вызов `s///` заменяет всю строку с телефоном пустой строкой, не оставляя даже символа новой строки. Соответственно в выходных данных эта строка присутствовать не будет, словно она никогда не существовала. В большинстве выходных строк ни один из трех шаблонов не совпадет, и эти строки будут выведены в неизменном виде.

Итак, результат близок к тому, что мы хотим получить, но пока мы не показали вам, как обновленная информация вернется на диск. Секрет кроется в переменной `$^I`. По умолчанию она равна `undef`, а программа работает обычным образом. Но если задать ей какую-либо строку, оператор <> начинает творить еще большие чудеса.

Вы уже знакомы с волшебством оператора <>: он автоматически открывает и закрывает серию файлов или читает данные из стандартного входного потока, если имена файлов не заданы. Но если `$^I` содержит строку, эта строка используется в качестве расширения резервной копии файла. Давайте посмотрим, как это работает на практике.

Допустим, оператор <> открывает наш файл *fred03.dat*. Он открывает файл так же, как прежде, но переименовывает его в *fred03.dat.bak*.[1] Открытым остается тот же файл, но теперь он хранится на диске под другим именем. Затем <> создает новый файл и присваивает ему имя *fred03.dat*. Никаких проблем при этом не возникает; это имя файла уже не используется. Далее <> по умолчанию выбирает новый файл

[1] В системах, не входящих в семейство UNIX, некоторые детали изменяются, но конечный результат будет практически тем же. Обращайтесь к документации своей версии Perl.

для вывода, так что все выводимые данные попадут в этот файл.[1] Цикл `while` читает строку из старого файла, обновляет ее и выводит в новый файл. На среднем компьютере программа способна обновить тысячи файлов за считанные секунды. Неплохо, верно?

Что же увидит пользователь, когда программа завершит работу? Пользователь скажет: «Ага, я вижу! Perl изменил мой файл *fred03.dat*, внес все необходимые изменения и любезно сохранил копию в резервном файле *fred03.dat.bak*!» Но мы-то знаем правду: Perl никакие файлы не обновлял. Он создал измененную копию, сказал «Абракадабра!» и поменял файлы местами, пока мы следили за волшебной палочкой. Хитро.

Некоторые программисты задают `$^I` значение `~` (тильда), потому что *emacs* создает резервные копии именно с таким расширением. Другое возможное значение `$^I` – пустая строка. Оно активизирует режим редактирования «на месте», но исходные данные не сохраняются в резервном файле. Но поскольку даже небольшая опечатка в шаблоне может стереть все старые данные, использовать пустую строку рекомендуется только в том случае, когда вы хотите проверить надежность своих архивов на магнитных лентах. Резервные копии файлов легко удалить после завершения. А если что-то пойдет не так и вам потребуется заменить обновленные файлы резервными копиями, Perl и в этом вам поможет (см. пример переименования нескольких файлов в главе 14).

Редактирование «на месте» в командной строке

Программу, приведенную в предыдущем разделе, написать несложно. Но Ларри решил, что и этого недостаточно.

Представьте, что вам потребовалось обновить сотни файлов, в которых имя `Randal` ошибочно записано с двумя `l` (`Randall`). Конечно, можно написать программу наподобие приведенной в предыдущем разделе. А можно ввести короткую программу прямо в командной строке:

```
$ perl -p -i.bak -w -e 's/Randall/Randal/g' fred*.dat
```

Perl поддерживает многочисленные ключи командной строки, позволяющие строить сложные программы при минимальном объеме кода.[2] Посмотрим, что происходит в этом примере.

Начало команды, `perl`, делает примерно то же, что строка `#!/usr/bin/perl` в начале файла: оно указывает, что программа *perl* должна обработать последующие данные.

---

[1] Оператор <> также пытается по возможности скопировать атрибуты разрешений доступа и владельца исходного файла. Например, если старый файл был доступен для чтения всем пользователям, этот же режим будет действовать и для нового файла.

[2] За полным списком обращайтесь к man-странице *perlrun*.

Ключ -p приказывает Perl автоматически сгенерировать программу. Впрочем, программа получается совсем маленькая; она выглядит примерно так:[1]

```
while (<>) {
  print;
}
```

Если вам и этого слишком много, используйте ключ -n; с ним команда print не генерируется, так что вы можете вывести только то, что нужно. (Ключи -p и -n знакомы поклонникам *awk*.) Программа, конечно, минимальная, но для нескольких нажатий клавиш неплохо.

Как нетрудно догадаться, следующий ключ -i.bak задает $^I значение ".bak" перед запуском программы. Если резервная копия не нужна, укажите ключ -i без расширения. (Если запасной парашют не нужен, прыгайте только с основным... может, и не понадобится.)

Ключ -w нам уже знаком – он включает предупреждения.

Ключ -e означает: «далее следует исполняемый код». Другими словами, строка s/Randall/Randal/g должна интерпретироваться как код Perl. Так как программа уже содержит цикл while (от ключа -p), код включается в цикл перед командой print. По техническим причинам последняя точка с запятой в коде -e не является обязательной. Но если команда содержит несколько ключей -e (а следовательно, несколько фрагментов кода), без риска можно опустить только точку с запятой в конце последнего фрагмента.

Последний параметр командной строки fred*.dat говорит, что массив @ARGV должен содержать список имен файлов, соответствующих этому шаблону. Сложим все вместе: командная строка работает так, как если бы мы написали следующую программу и запустили ее для файлов fred*.dat:

```
#!/usr/bin/perl -w

$^I = ".bak";

while (<>) {
  s/Randall/Randal/g;
  print;
}
```

Сравните эту программу с приведенной в предыдущем разделе. Эти две программы очень похожи. Ключи командной строки удобны, не правда ли?

---

[1] Вообще говоря, print находится в блоке continue. За дополнительной информацией обращайтесь к man-страницам *perlsyn* и *perlrun*.

Упражнения

Ответы к упражнениям приведены в приложении А.

1. [7] Создайте шаблон, который совпадает с тремя последовательными копиями текущего содержимого `$what`. Иначе говоря, если `$what` содержит `fred`, шаблон должен совпасть с `fredfredfred`. Если `$what` содержит `fred|barney`, шаблон должен совпадать с `fredfredbarney`, `barneyfredfred`, `barneybarneybarney` и множеством других вариантов. (Подсказка: значение `$what` должно задаваться в начале тестовой программы командой вида `$what = 'fred|barney';`.)
2. [12] Напишите программу, которая создает измененную копию текстового файла. В копии каждое вхождение строки `Fred` (с любым регистром символов) должно заменяться строкой `Larry` (таким образом, строка `Manfred Mann` должна превратиться в `ManLarry Mann`). Имя входного файла должно задаваться в командной строке (не запрашивайте его у пользователя!), а имя выходного файла образуется из того же имени и расширения `.out`.
3. [8] Измените предыдущую программу так, чтобы каждое вхождение `Fred` заменялось строкой `Wilma`, а каждое вхождение Wilma – строкой `Fred`. Входная строка `fred&wilma` в выходных данных должна принимать вид `Wilma&Fred`.
4. [10] Упражнение «на повышенную оценку»: напишите программу, которая включает в любую программу из упражнений строку с информацией об авторских правах следующего вида:

   ```
   ## Copyright (C) 20XX by Yours Truly
   ```

   Строка должна размещаться сразу же за строкой с «решеткой». Файл следует изменять «на месте» с сохранением резервной копии. Считайте, что при запуске программы имена изменяемых файлов передаются в командной строке.
5. [15] Упражнение на «совсем повышенную оценку»: измените предыдущую программу так, чтобы она не изменяла файлы, уже содержащие строку с информацией об авторских правах. (Подсказка: имя файла, из которого оператор `<>` читает данные, хранится в `$ARGV`.)