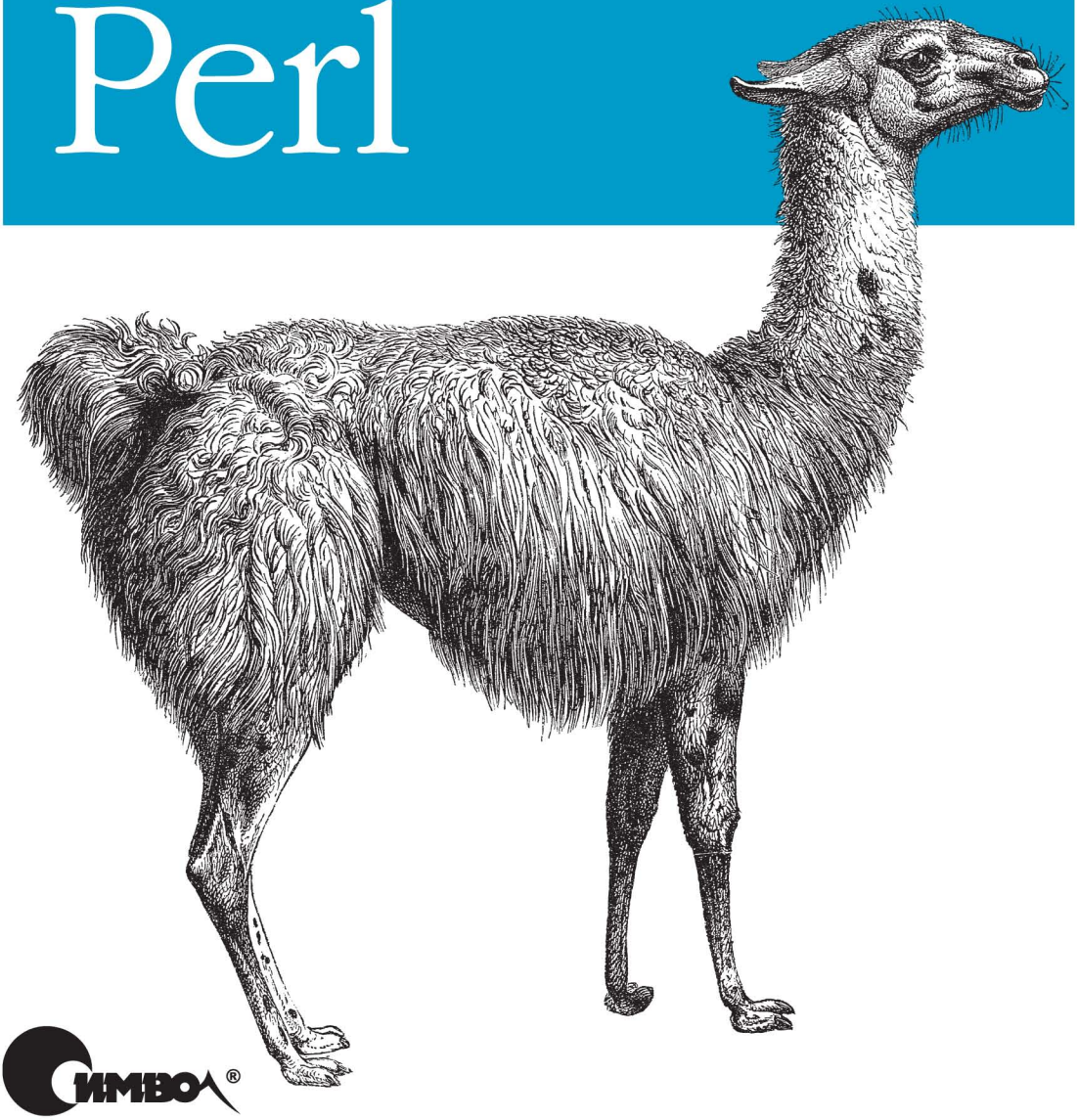
Как сделать сложное простым и невозможное возможным

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.

1

Введение

Приветствуем читателей «книги с ламой»!

Перед вами пятое издание книги, которую с 1993 года (выход первого издания) прочитало более полумиллиона человек. Надеемся, им понравилось. И уж можем точно сказать, что нам понравилось ее писать.[1]

Вопросы и ответы

Вероятно, у вас есть какие-нибудь вопросы о Perl, а может быть, и о книге, особенно если вы уже заглянули в середину и увидели, что вас ждет. В этой главе мы постараемся ответить на них.

Подходит ли вам эта книга?

Если вы хоть немного похожи на нас, то сейчас вы стоите в книжном магазине[2] и размышляете: то ли взять «книгу с ламой» и изучать Perl... а может, лучше взять вон ту книгу и изучить другой язык программи-

---

[1] На самом деле первое издание книги написал Рэндал Л. Шварц, второе – Рэндал и Том Кристиансен, затем было издание Рэндала и Тома Феникса, а теперь авторский коллектив составляют Рэндал, Том Феникс и брайан д фой (согласно *http://www252.pair.com/comdog/style.html*, именно такой вариант написания его имени следует считать правильным. – *Прим. перев.*). Следовательно, везде, где в этом издании мы говорим «мы», имеется в виду последняя группа. Вас интересует, почему мы говорим «нам понравилось» (в прошедшем времени), хотя это всего лишь первая страница? Все просто: мы начали с конца и постепенно продвигались к началу. Возможно, кому-то это покажется странным. Но, по правде говоря, когда мы закончили работу над алфавитным указателем, остальное было проще простого.

[2] Вообще-то если вы действительно похожи на нас, то вы стоите в *библиотеке*, а не в магазине. Мы – известные жмоты.

рования, названный в честь змеи[1], напитка или буквы латинского алфавита? У вас всего две минуты, пока подошедший продавец не напомнил, что здесь не библиотека.[2] Возможно, вы предпочитаете потратить эти две минуты на рассмотрение короткой программы, написанной на Perl; она даст представление о мощи Perl и о том, на что способен этот язык. В таком случае обращайтесь к разделу «Perl за две минуты».

Почему так много сносок?

Спасибо, что заметили. В книге действительно *много* сносок. Не обращайте внимания. Сноски нужны потому, что Perl полон исключений из правил. И это хорошо, ведь в самой жизни тоже полно исключений из правил.

Однако это означает, что мы не можем честно заявить: «Фигляционный оператор транглюкирует хвазистатические переменные» без сноски с исключениями.[3] Мы вообще честные парни, поэтому нам приходится писать сноски. Зато вы можете оставаться честным, не читая их (удивительно, но факт).

Многие исключения связаны с портируемостью. Perl создавался для систем семейства UNIX и по-прежнему глубоко связан с ними. Но там, где это возможно, мы постарались упомянуть о вероятных неожиданностях – то ли из-за особенностей других систем, то ли по иным причинам. Надеемся, что даже для читателей, которые ничего не знают о UNIX, эта книга все равно станет хорошим введением в Perl (а заодно они немного познакомятся с UNIX, и притом совершенно бесплатно).

Вдобавок многие исключения подчиняются старому правилу «80/20». Другими словами, 80% поведения Perl описывается в 20% документации, а оставшиеся 20% поведения занимают 80% документации. Чтобы книга не разрасталась сверх меры, самое типичное и доступное поведение обсуждается в основном тексте, а указатели на боковые дорожки приводятся в сносках (которые печатаются мелким шрифтом, что позволяет нам уместить больше информации в том же объеме[4]). Когда вы прочитаете всю книгу, не обращаясь к сноскам, возможно, вам захочется вернуться к некоторым разделам за дополнительными сведениями... Или если вас вдруг разберет любопытство во время чтения – читайте сноски. Большинство из них все равно содержит шутки для компьютерщиков.

---

1 Не торопитесь объяснять, что Python был назван по имени комедийной труппы, а не змеи. Мы имели в виду кобру... в смысле CORBA.

2 Если это, конечно, *не* библиотека.

3 Кроме вторников во время сбоев питания, если держать локти под особым углом, а на дворе полнолуние – или под действием `use integer` в цикле, вызываемом из прототипизованной процедуры в версиях до Perl 5.6.

4 Мы даже хотели напечатать весь текст в виде сноски для экономии бумаги, но сноски к сноскам – это уже слишком.

Зачем нужны упражнения и ответы?

Каждая глава завершается серией упражнений, потому что мы – все трое – преподавали наш учебный курс нескольким тысячам студентов.[1] Упражнения были тщательно построены так, чтобы дать вам возможность ошибиться.

Мы вовсе *не хотим*, чтобы вы ошибались, но предоставить такую *возможность* необходимо. Вы все равно совершите большинство этих ошибок во время своей карьеры Perl-программиста, так уж лучше сейчас. Совершив ошибку во время чтения книги, вы уже не повторите ее при написании программы за день до сдачи. А если где-то возникнут трудности, мы всегда придем вам на помощь – в приложении А приведены наши ответы на все упражнения с небольшим сопроводительным текстом, который объясняет совершенные (и несовершенные) ошибки. Закончив работу с упражнениями, сверьтесь с ответами.

Старайтесь не заглядывать в ответ до тех пор, пока как следует не потрудитесь над задачей. Вы лучше запомните то, что узнаете, а не то, о чем читаете. И даже если вам не удалось разобраться в каком-то решении, не стоит биться головой о стену: не огорчайтесь и переходите к следующей главе.

Даже если вы никогда не ошибаетесь, все равно загляните в ответы. В пояснительном тексте могут быть упомянуты некоторые подробности, не очевидные на первый взгляд.

Что означают цифры в начале упражнения?

Каждое упражнение начинается с цифры в квадратных скобках; это выглядит примерно так:

1. [2] Что означает цифра 2 в квадратных скобках перед текстом упражнения?

Это наша (весьма приблизительная) оценка того, сколько минут вы затратите на данное упражнение. Не удивляйтесь, если вся задача (с написанием кода, тестированием и отладкой) будет решена вдвое быстрее или займет вдвое больше времени. А если возникнут непреодолимые трудности, мы никому не скажем, что вы подсматривали в приложение А.

А если я веду курсы Perl?

Если вы преподаете Perl и хотите использовать книгу в качестве учебника (как это многие делали за прошедшие годы), знайте: мы постарались сделать каждую серию упражнений достаточно короткой, чтобы большинство студентов затратило на ее решение от 45 минут до часа и еще осталось немного времени на перерыв. В одних главах упражне-

[1] Не всем сразу, конечно.

ния решаются дольше, в других быстрее. Дело в том, что после расстановки цифр в квадратных скобках мы вдруг разучились суммировать (но хотя бы помним, как заставить компьютер делать это за нас).

Что означает «Perl»?

Иногда Perl расшифровывают как «Practical Extraction and Report Language» (практический язык выборки данных и построения отчетов), хотя встречались и другие варианты, например «Pathologically Eclectic Rubbish Lister» (атологически эклектичный генератор чепухи). На самом деле название является *бэкронимом*, а не акронимом, поскольку Ларри Уолл, создатель Perl, сначала придумал название, а потом расшифровку. Именно поэтому название «Perl» не состоит из одних прописных букв. Также бессмысленно спорить, какая расшифровка является правильной: Ларри одобряет обе.

Иногда «perl» записывают со строчной буквы «р». Как правило, «Perl» с прописной буквы обозначает язык, а «perl» со строчной буквы – интерпретатор, который компилирует и запускает ваши программы.

Зачем Ларри создал Perl?

Ларри создал Perl в середине 1980-х годов, когда он пытался построить сводку по новостям Usenet (что-то вроде иерархии файлов для системы анализа ошибок), а возможностей *awk* оказалось для этого недостаточно. Ларри, будучи весьма ленивым[1] программистом, решил действовать наверняка и создать универсальный инструмент, который можно было бы использовать хотя бы еще в одном месте. Так появилась нулевая версия Perl.

Почему Ларри не воспользовался другим языком?

Компьютерных языков хватает в избытке, не так ли? Но в то время Ларри не нашел ничего, что отвечало бы его потребностям. Если бы один из современных языков существовал в те времена, возможно, Ларри воспользовался бы им. Ему был нужен язык, сочетающий быстроту программирования командного процессора, или *awk*, с мощью более совершенных инструментов (таких как *grep*, *cut*, *sort* и *sed*[2]), и чтобы при этом не приходилось обращаться к таким языкам, как C.

---

[1] Мы вовсе не пытаемся обидеть Ларри; лень суть добродетель. Тачку изобрел тот, кому было лень носить в руках; письменность – тот, кому было лень запоминать; Perl – тот, кому было лень возиться с одной конкретной задачей без изобретения нового языка программирования.

[2] Не огорчайтесь, если вам незнакомы эти названия. Важно лишь то, что эти программы входили в UNIX-инструментарий Ларри, но не подходили для его задачи.

Perl пытается заполнить пробел между низкоуровневым (C, C++, ассемблер) и высокоуровневым программированием (командный процессор). Низкоуровневые программы трудны в написании и уродливы, но они работают быстро и обладают неограниченными возможностями; превзойти по скорости хорошо написанную низкоуровневую программу на конкретном компьютере довольно трудно. Для низкоуровневых программ нет почти ничего невозможного. С другой стороны, высокоуровневые программы медленно работают, они уродливы, а их возможности ограничены; существует множество задач, которые невозможно решить средствами командного процессора или пакетных файлов, если в системе нет команды, предоставляющей соответствующую функциональность. Perl прост, обладает почти неограниченными возможностями, работает в целом быстро... и в чем-то уродлив.

Разберемся подробнее с каждым из этих четырех утверждений.

Perl прост. Как вы вскоре увидите, речь идет о простоте *использования*, а в *изучении* Perl не так уж прост. Начинающий водитель много недель и месяцев учится водить машину, зато потом все становится легко. Когда вы проведете за программированием на Perl столько же времени, сколько учились водить машину, Perl тоже покажется вам простым.

Perl обладает почти неограниченными возможностями. Задачи, которые нельзя решить на Perl, встречаются очень редко. Скажем, на Perl не стоит писать драйвер устройства с обработкой прерываний на уровне микроядра (хотя и это было сделано), но Perl отлично справится с большинством задач, которые обычные люди решают в своей повседневной работе: от быстрых «одноразовых» программ до серьезных приложений корпоративного уровня.

Perl работает в целом быстро. Все программисты, пишущие на Perl, также используют его в своей работе, поэтому мы хотим, чтобы программы работали как можно быстрее. Если кто-то предложит добавить новую возможность, которая будет действительно классной, но замедлит работу других программ, Ларри почти наверняка откажет – если только мы не придумаем, как ускорить ее.

Perl в чем-то уродлив. Это правда. Символом Perl стал верблюд с обложки почтенной «книги с верблюдом» (Ларри Уолл, Том Кристиансен и Джон Орвант «Programming Perl»[1], O'Reilly), дальний родственник ламы (и ее сестры альпаки[2]). Верблюды тоже по-своему уродливы, зато они усердно работают даже в самых тяжелых условиях. Верблюд выполняет свою работу, несмотря на все трудности, хотя он плохо выглядит, еще хуже пахнет и иногда плюется. Perl на него немного похож.

---

[1] Ларри Уолл, Том Кристиансен и Джон Орвант «Программирование на Perl», 3-е издание, Символ-Плюс, 2002.

[2] «Книга с альпакой» – Рэндал Шварц, Том Феникс и брайан д фой «Intermediate Perl» (O'Reilly) – «Perl: изучем глубже», Символ-Плюс, 2007.

Perl прост или сложен?

Perl прост в использовании, но иногда бывает сложен в изучении. Конечно, это обобщение. При проектировании Perl Ларри приходилось принимать много компромиссных решений. Когда у него была возможность сделать что-то проще для программистов, но при этом усложнить изучение языка, он почти всегда отдавал предпочтение программисту. Дело в том, что вы изучаете Perl только раз, а потом используете его снова и снова.[1] Perl содержит массу вспомогательных средств, экономящих время программиста. Например, многие функции имеют значения по умолчанию, которые соответствуют наиболее типичному сценарию использования функции, поэтому в программах Perl часто встречаются фрагменты следующего вида:

```
while (<>) {
  chomp;
  print join("\t", (split /:/)[0, 2, 1, 5] ), "\n";
}
```

В полной записи без принятых в Perl сокращений и значений по умолчанию этот фрагмент станет в 10–12 раз длиннее; на его чтение и запись также потребуется намного больше времени. Дополнительные переменные затруднят сопровождение и отладку кода. Если вы уже немного знаете Perl и не видите в этом фрагменте ни одной переменной, то это особенность Perl. По умолчанию они не используются. Но за упрощение работы программиста приходится расплачиваться во время обучения; вам придется освоить все эти сокращения и значения по умолчанию.

Происходящее можно сравнить с сокращениями в нашей устной речи. Конечно, «полвторого» означает то же самое, что «тринадцать часов тридцать минут». И все же мы обычно предпочитаем первый вариант, потому что он короче, но при этом известен большинству людей и будет понятен им. Аналогично «сокращения» Perl позволяют быстрее «произносить» стандартные «фразы», которые воспринимаются программистом как одна идиома, а не как последовательность не связанных друг с другом шагов.

После освоения Perl вы будете тратить меньше времени на правильное экранирование символов в строках командного процессора (или на объявления C) и у вас останется больше времени на странствия в Интернете, так как Perl является отличным инструментом для повышения эффективности. Компактные конструкции Perl позволяют создавать (с минимальными усилиями) весьма впечатляющие программы –

[1] Если вы собираетесь программировать всего несколько минут в неделю или месяц, лучше выберите другой язык – в промежутках вы забудете почти все, что узнали. Perl создан для тех людей, которые программируют хотя бы 20 минут в день и в основном именно на Perl.

как одноразовые, так и универсальные. Вы также сможете использовать готовые наработки в новых проектах, потому что Perl отличается высокой портируемостью и повсеместным распространением, и у вас останется еще больше времени для Интернета.

Perl относится к языкам очень высокого уровня. Это означает, что программный код получается весьма компактным; программа, написанная на Perl, занимает от одной до трех четвертей объема аналогичной программы на C. Программы Perl быстрее пишутся, быстрее читаются, быстрее отлаживаются, а их сопровождение занимает меньше времени. Чтобы осознать этот факт, вам даже не понадобится большой опыт программирования: хорошо, когда функция помещается на экране и вам не приходится прокручивать ее туда-сюда, чтобы разобраться в происходящем. А поскольку количество ошибок в программе условно пропорционально длине исходного кода[1] (а не функциональности программы), короткий исходный код Perl оборачивается меньшим количеством ошибок в среднем.

На Perl, как и на любом языке, можно программировать в режиме «write-only» – иначе говоря, писать программы, которые никто не сможет прочитать. Но разумная осторожность позволит вам обойти это распространенное обвинение. Да, непосвященному программы Perl иногда кажутся случайной мешаниной символов, но опытной программист видит за этими малопонятными значками ноты величественной симфонии. Если вы будете следовать рекомендациям этой книги, у ваших программ не будет никаких проблем с чтением и сопровождением... и ваши творения не победят на конкурсе «Самой Запутанной Perl-программы».

Почему Perl стал таким популярным?

Ларри немного поэкспериментировал с Perl, слегка доработал его и опубликовал в Usenet. Пользователи этого разномастного, постоянно изменяющегося сообщества систем со всего мира (которых насчитывалjсь десятки тысяч) обеспечили обратную связь, попросили добавить то или это, причем многое из того, что в первоначальные планы Ларри совершенно не входило.

В результате Perl неуклонно разрастался. Росла его функциональность. Росло количество версий для других платформ. О том, что когда-то было маленьким языком для пары UNIX-систем, теперь написаны тысячи страниц электронной документации, десятки книг, созданы крупные группы в основном потоке Usenet (а также десятки альтернативных групп и списков рассылки для тех, кто не входит в основной поток) с бесчисленным множеством читателей и реализаций для практически

[1] А если размер любой секции программы превышает размер экрана, количество ошибок резко возрастает.

любой системы, существующей в наши дни. Да, и не забудьте прибавить к этому «книгу с ламой».

Что происходит с Perl сейчас?

Ларри Уолл сейчас сам не программирует, но по-прежнему руководит разработкой и принимает важнейшие решения. Поддержкой Perl в основном занимается группа энтузиастов, называемых «Perl 5 Porters». Вы можете подписаться на список рассылки по адресу *perl5-porters@perl.org*, чтобы быть в курсе их работы и обсуждений.

На момент написания книги (март 2008 года) в мире Perl происходит много важных событий. Последнюю пару лет многие люди работали над следующей принципиально новой версией: Perl 6.

Не торопитесь выбрасывать Perl 5 – эта версия все еще остается актуальной и стабильной. Выход стабильной версии Perl 6 в ближайшее время не ожидается. Perl 5 делает все, что он делал всегда, и так оно и будет дальше. Perl 5 не исчезнет с выходом Perl 6; вероятно, какое-то время будут параллельно использоваться обе версии. Perl 5 Porters сопровождают Perl 5, как это было всегда, и некоторые удачные идеи из Perl 6 были воплощены в Perl 5. Мы обновляем эту книгу, потому что недавно была опубликована версия Perl 5.10, и похоже, группа Perl 5 Porters работает над Perl 5.12.

В 2000 году Ларри впервые выпустил очередную версию Perl, написанную силами сообщества Perl. В следующие годы появился новый интерпретатор Parrot, но с точки зрения пользователей ничего принципиально нового не происходило. В 2005 году Отриус Тан (Autrijus Tang) начала эксперименты с Pugs (Perl User Golfing System) – облегченной реализацией Perl 6 на Haskell. Разработчики с обеих сторон (как Perl, так и Haskell) поспешили на помощь. Parrot – виртуальная машина, на которой выполняется Perl 6, – неплохо развивается. Сейчас вся основная работа идет именно в этом направлении. За дополнительной информацией о Perl 6 обращайтесь по адресам *http://perl-six.org* и *http://www.pugscode.org*. Впрочем, в этой книге мы не будем отвлекаться на Perl 6.

Для каких задач подходит Perl?

Perl хорошо подходит для написания программ «на скорую руку», которые создаются за три минуты. Кроме того, Perl хорошо подходит для длинных, крупномасштабных программ, которые пишутся десятком программистов за три года. На создание большинства ваших программ, скорее всего, будет уходить менее часа – от исходного плана до полного тестирования кода.

Perl оптимизирован для задач, которые на 90% состоят из работы с текстом и только на 10% из других операций. Похоже, такое описание соответствует большинству задач программирования, встречаю-

щихся в наши дни. В идеальном мире каждый программист знает все языки и выбирает лучший язык для своего проекта. Чаще всего вы бы выбирали Perl.[1] Хотя в те времена, когда Ларри создавал Perl, Тим Бернерс-Ли даже не помышлял о Web, в конечном итоге этот союз был неизбежен. Кое-кто считает, что распространение Perl в начале 1990-х годов позволило очень быстро перенести в формат HTML большие объемы контента, без которого само существование Web было бы невозможно. Конечно, Perl также является излюбленным языком для создания мелких сценариев CGI (программы, выполняемые веб-сервером) – доходит до того, что непосвященные часто спрашивают: «А разве CGI и Perl не одно и то же?» или «А почему вы используете Perl вместо CGI?». Выглядит довольно забавно.

Для каких задач Perl не подходит?

Если Perl годится почти для всего, то для чего он *не* годится? Не беритесь за Perl, если вам нужно создать *непрозрачный двоичный файл* – иначе говоря, если вы собираетесь передать или продать свою программу другим людям и не хотите, чтобы они увидели ваши тайные алгоритмы в исходном коде (с другой стороны, получатели не смогут помочь с сопровождением или отладкой кода). Программы Perl обычно передаются в виде исходного кода, а не двоичного файла.

Если вам непременно нужен непрозрачный двоичный файл, хотим предупредить, что полной непрозрачности не бывает. Если кто-то может установить и запустить вашу программу, он же сможет преобразовать ее в исходный код. Правда, этот код не всегда будет совпадать с тем, который был написан вами, но, по крайней мере, будет похож на него. Лучший способ сохранить секретность вашего тайного алгоритма – нанять команду адвокатов; они напишут лицензию, в которой говорится: «С этим кодом можно делать *то*, но нельзя делать *это*. А если вы нарушите наши правила, то мы, команда адвокатов, заставим вас пожалеть об этом».

Где взять Perl?

Вероятно, у вас он уже есть. Во всяком случае *мы* встречаем Perl повсюду. Он входит в поставку многих систем, а системные администраторы часто устанавливают его на всех компьютерах своей сети. Но даже если вы не найдете Perl в своей системе, то сможете установить его бесплатно.

---

[1] Впрочем, мы не призываем верить нам на слово. Если вы хотите знать, действительно ли Perl лучше языка X, изучите оба языка, опробуйте их и посмотрите, какой из них вы будете чаще использовать в своей работе. Этот язык и будет лучшим. В конечном итоге вы начнете лучше понимать Perl благодаря своему изучению X, и наоборот, так что время не пропадет зря.

Perl распространяется по двум разным лицензиям. Тем, кто будет только *пользоваться* Perl, подойдет любая лицензия. Но если вы собираетесь модифицировать Perl, прочитайте текст лицензий повнимательнее; в них указаны небольшие ограничения на распространение модифицированного кода. Для тех, кто не будет модифицировать Perl, в лицензии фактически сказано: «Распространяется бесплатно – развлекайтесь».

Perl не только свободно распространяется, но и отлично работает практически в любой системе, причисляющей себя к семейству UNIX и обладающей компилятором C. Загрузите пакет, введите пару команд – начинается автоматическая сборка и настройка Perl. А еще лучше, уговорите своего системного администратора ввести эту пару команд и установить Perl за вас.[1] Кроме семейства UNIX у приверженцев Perl также хватило энтузиазма на портирование Perl в другие системы: Mac OS X, VMS, OS/2, даже MS/DOS и во все современные разновидности Windows – и вероятно, когда вы будете читать эту книгу, список пополнится.[2] Многие портированные версии Perl поставляются с установочными программами, которые еще удобнее в работе, чем установка Perl в UNIX. Поищите по ссылкам в секции «ports» в CPAN.

Что такое CPAN?

CPAN (Comprehensive Perl Archive Network) – Сеть полных архивов Perl, главный сетевой ресурс всего, что относится к Perl. Здесь хранится исходный код Perl, готовые к установке версии Perl для всех систем, не входящих в семейство UNIX[3], примеры, документация, расширения Perl и архивы сообщений. Короче говоря, CPAN – действительно *полный* архив.

У CPAN имеются сотни зеркал по всему миру; просмотр или поиски стоит начать с сайта *http://search.cpan.org/* или *http://kobesearch.cpan.org/*. Если Интернет недоступен, попробуйте найти CD-ROM или DVD-ROM со всеми полезными компонентами CPAN, загляните в местный магазин технической литературы. Проследите за тем, чтобы ар-

---

[1] Если системный администратор не устанавливает программы, зачем он вообще нужен? А если не удастся уговорить, пообещайте купить пиццу. Мы еще не видели ни одного системного администратора, который отказался бы от бесплатной пиццы или, по крайней мере, не предложил заменить ее чем-нибудь таким же доступным.

[2] Нет, на момент написания книги Perl не помещался в Blackberry – даже в урезанном виде он остается слишком большим. Впрочем, ходят слухи, что его удалось запустить на WinCE.

[3] В системах UNIX всегда лучше компилировать Perl из исходного кода. В других системах может не оказаться компилятора C и других инструментов, необходимых для компиляции, поэтому в CPAN хранятся соответствующие двоичные файлы.

хив был относительно свежим. CPAN ежедневно изменяется, поэтому копия двухлетней давности безнадежно устарела. А еще лучше – попросите доброго друга с доступом к Сети записать вам сегодняшнюю копию CPAN.

Как получить техническую поддержку?

У вас есть полный исходный код – вот и исправляйте ошибки самостоятельно!

Вас не радует такая перспектива, верно? Но на самом деле это хорошо. Благодаря доступности исходного кода Perl, любой сможет исправить ошибку – более того, к моменту, когда вы найдете ошибку и убедитесь в ее наличии, она может уже оказаться исправленной. Тысячи людей по всему миру участвуют в сопровождении Perl.

Мы не хотим сказать, что Perl содержит много ошибок. Но это программа, а в любой программе присутствует хотя бы одна ошибка. Чтобы понять, почему так полезно иметь исходный код, представьте, что вместо Perl вы приобрели лицензионный язык программирования Forehead у гигантской могущественной корпорации, принадлежащей плохо подстриженному мультимиллиардеру. (Конечно, пример сугубо гипотетический. Всем известно, что языка программирования Forehead не существует.) Теперь подумайте, что делать, если вы обнаружили ошибку в Forehead. Сначала нужно сообщить об ошибке, затем можно надеяться... да, именно надеяться, что ошибку исправят, что ее исправят *быстро* и что с вас не сдерут слишком много денег за новую версию. А еще нужно надеяться, что в новой версии не появятся новые ошибки, а гигантская компания тем временем не разорится из-за антимонопольных исков.

При работе с Perl у вас есть исходный код. В редком и маловероятном случае, когда ошибку не удается исправить другим способом, вы нанимаете программиста (или 10 программистов) и решаете все проблемы. Купили новый компьютер, на котором Perl еще не работает? Портируйте его самостоятельно. Нужна возможность, которой пока нет в Perl? Вы знаете, что делать.

Существуют ли другие виды поддержки?

Конечно. Мы можем порекомендовать Perl Mongers – всемирную ассоциацию групп пользователей Perl (за дополнительной информацией обращайтесь на сайт *http://www.pm.org/*). Возможно, вблизи от вас найдется группа, в которую входит специалист или кто-то из ее участников знаком со специалистом. А если группы нет, вы можете легко создать ее.

Разумеется, не стоит пренебрегать документацией как первой линией поддержки. Помимо man-страниц[1] документацию также можно найти в CPAN (*http://www.cpan.org*) и на других сайтах. В частности, стоит посетить сайт *http://perldoc.perl.org*, где имеются HTML- и PDF-версии документации Perl, и *http://faq.perl.org/* с новейшей версией perlfaq.

Другой авторитетный источник информации – книга «Programming Perl», которую обычно называют «книгой с верблюдом» из-за изображения на обложке (подобно тому, как эту книгу называют «книгой с ламой»). «Книга с верблюдом» содержит полную справочную информацию, немного учебного материала и разнообразную информацию о Perl. Имеется также карманный справочник «Perl 5 Pocket Reference» (O'Reilly), написанный Йоханом Вромансом (Johan Vromans). Его удобно держать под рукой (или носить в кармане).

Для вопросов в Usenet существуют группы новостей и списки рассылки.[2] В любой час дня и ночи в каком-нибудь часовом поясе найдется специалист, отвечающий на вопросы в группах Usenet, посвященных Perl, – над империей Perl солнце никогда не заходит. А это значит, что ответ на заданный вопрос часто удается получить за несколько минут. Если же не заглянули в документацию или FAQ, то за те же несколько минут получите свою порцию флейма.

Официальные группы новостей Usenet входят в иерархию *comp.lang.perl.**. На момент написания книги таких групп было пять, но их состав меняется время от времени. Вам (или тому, кто отвечает за Perl на вашем сайте) обычно стоит подписаться хотя бы на *comp.lang.perl.announce* – группу с небольшим объемом трафика, публикующую важные объявления относительно Perl, особенно объявления из области безопасности. Если понадобится помощь с подключением к Usenet, обращайтесь к местным специалистам.

Имеются также веб-сообщества, сформированные на базе дискуссионных групп Perl. В очень популярном сообществе Perl Monastery (*http://www.perlmonks.org/*) участвуют многие авторы книг и рубрик Perl, в том числе двое авторов книги. Заслуживает внимания сайт *http://learn.pearl.org/* и его список рассылки *beginners@perl.org*. Новости Perl публикуются на сайте *http://use.perl.org/*. Многие известные программисты Perl ведут блоги по теме Perl; для их чтения можно воспользоваться *http://planet.perl.org*.

Наконец, если вам понадобятся платные услуги по поддержке Perl, существует множество компаний с разными расценками. И все же чаще всего необходимую поддержку удается получить бесплатно.

---

[1] Термином «man-страницы» в UNIX называется электронная документация. Если вы не работаете в UNIX, содержимое man-страниц Perl должно быть доступно через «родную» систему документации вашей системы.

[2] Многие списки перечислены на сайте *http://lists.perl.org*.

А если я найду ошибку в Perl?

Если вы обнаружили ошибку, прежде всего проверьте документацию[1] еще раз[2]. Perl содержит массу специальных возможностей и исключений из правил; возможно, то, с чем вы столкнулись, не ошибка, а специфическая особенность. Также убедитесь в том, что у вас установлена новая версия Perl; возможно, обнаруженная ошибка уже исправлена.

Если вы на 99% уверены, что нашли настоящую ошибку, расспросите окружающих. Спросите на работе, на встрече местной группы Perl Mongers, на конференции Perl. Возможно, это *все-таки* не ошибка.

Если вы на 100% уверены, что нашли настоящую ошибку, подготовьте тестовый сценарий. (Неужели вы еще не подготовили его?) Идеальный тестовый сценарий представляет собой маленькую автономную программу, запустив которую, любой пользователь Perl столкнется с тем же (неправильным) поведением. Когда тестовый сценарий, четко показывающий ошибку, будет создан, воспользуйтесь утилитой *perlbug* (входящей в поставку Perl) и сообщите об ошибке. Обычно уведомления об ошибках направляются разработчикам Perl по электронной почте, поэтому не используйте *perlbug* без готового тестового сценария.

Если все было сделано верно, ответ на сообщение об ошибке нередко приходит через несколько минут. Обычно проблема решается установкой простой «заплатки» (патча), после чего можно продолжать работу. Конечно, ответа может и не быть (худший случай); формально разработчики Perl не обязаны читать сообщения об ошибках. Но все мы любим Perl, и никто не захочет, чтобы в него прокралась незамеченная ошибка.

Как написать программу на Perl?

Хороший вопрос (даже если вы и не спрашивали). Программы Perl представляют собой обычные текстовые файлы; они создаются и редактируются в текстовых редакторах. (Никакие специальные средства разработки вам не понадобятся, хотя некоторые фирмы предлагают коммерческие продукты для программирования на Perl. Мы никогда ими не пользовались, так что ничего порекомендовать не можем.)

В общем случае следует использовать текстовый редактор для программистов вместо обычного текстового редактора. В чем различия? Текстовый редактор для программистов позволяет выполнять такие стандартные операции, как установка или удаление отступов в про-

---

[1] Даже Ларри признается, что время от времени ему приходится обращаться к документации.

[2] А лучше два или три раза. Очень часто мы обращались к документации, пытаясь объяснить неожиданное поведение Perl, и обнаруживали какой-то неизвестный нюанс.

граммных блоках или поиск пары для открывающей фигурной скобки. В системах UNIX программисты чаще всего используют редакторы *emacs* и *vi* (а также их многочисленные разновидности и клоны). В Mac OS X имеются хорошие редакторы BBEdit и TextMate, многие хорошо отзывались о редакторах UltraEdit и PFE (Programmer's Favorite Editor) для Windows. Другие редакторы перечислены в man-странице *perlfaq2*. Узнайте у местных специалистов, какие текстовые редакторы существуют для вашей системы.

Все программы, которые вы будете писать в упражнениях, достаточно просты. Ни одна из них не займет более 20 или 30 строк кода, поэтому для них подойдет любой редактор.

Некоторые новички пытаются работать в текстовых процессорах вместо текстовых редакторов. Не рекомендуем – в лучшем случае это неудобно, а в худшем невозможно. Впрочем, мы не пытаемся остановить вас. Не забудьте указать, чтобы текстовый процессор сохранял файлы в формате простого текста; собственный формат текстового процессора почти наверняка окажется непригодным. Многие текстовые процессоры также будут сообщать, что программа на Perl содержит слишком много орфографических ошибок и в ней слишком много символов «;».

Иногда программы приходится писать на одном компьютере, а потом пересылать их на другой компьютер для запуска. В подобных ситуациях следите за тем, чтобы данные передавались в режиме «text» или «ASCII», а не в режиме «binary». Это необходимо из-за различий в форматах текста на разных компьютерах. Нарушение этого условия приведет к непредсказуемым результатам – в некоторых версиях Perl непредвиденный формат перевода строки приводит к аварийному завершению.

Простая программа

Согласно давней традиции любая книга о языке программирования, происходящем из семейства UNIX, должна начинаться программой «Hello, world». Вот как выглядит эта программа на Perl:

```
#!/usr/bin/perl
print "Hello, world!\n";
```

Предположим, вы набрали этот текст в своем любимом редакторе. (Не беспокойтесь о том, что означают эти команды и как они работают, – вскоре вы все поймете.) Программа обычно сохраняется под любым именем по усмотрению программиста. Perl не предъявляет никаких требований к именам или расширениям файлов, и расширения лучше вообще не использовать.[1] Впрочем, некоторые системы требуют задавать расширение, например *.plx* (PerL eXecutable); за дополнительной информацией обращайтесь к сопроводительной документации своей версии.

Возможно, придется также позаботиться о том, чтобы система распознавала файл как исполняемую программу (т. е. команду). Как именно это сделать, зависит от системы; иногда бывает достаточно сохранить программу в определенном каталоге. В семействе UNIX файл помечается как исполняемый командой *chmod* следующего вида:

```
$ chmod a+x my_program
```

Знак доллара (и пробел) в начале строки обозначают приглашение командного процессора. Скорее всего, в вашей системе они будут выглядеть иначе. Если вы привыкли использовать *chmod* с числовыми кодами типа `755` вместо символических кодов типа `a+x`, это, конечно, тоже подходит. Так или иначе команда сообщает системе, что файл представляет собой исполняемую программу.

Теперь все готово к запуску:

```
$ ./my_program
```

Точка и косая черта в начале команды означают, что программа находится в текущем рабочем каталоге. Иногда без этих символов можно обойтись, но пока вы не будете в полной мере понимать их смысл, лучше включать их в начало команды.[1] Если все сразу заработало нормально, это настоящее чудо. Чаще выясняется, что программа содержит ошибку. Отредактируйте ее и попробуйте снова; вам не придется заново вводить команду *chmod*, так как нужный атрибут «закрепился» за файлом. (Конечно, если ошибка была допущена при вводе команды *chmod*, командный процессор выдаст сообщение «отказано в доступе».)

В Perl 5.10 эту простую программу можно написать другим способом; нам хотелось бы сразу упомянуть об этом. Вместо `print` используется команда `say`, которая делает почти то же самое, но с меньшим количеством символов. Поскольку `say` относится к новым возможностям языка, а Perl 5.10 может быть не установлен на вашем компьютере, в программу включается директива `use 5.010`, которая сообщает Perl об использовании новых возможностей:

---

[1] Почему, спросите вы? Представьте, что вы написали программу для вычисления счета в боулинге и сообщили всем своим друзьям, что она называется *bowling.plx*. Однажды вы решаете переписать ее на C. Сохраните ли вы прежнее имя файла, которое наводит на мысль, что программа написана на Perl? Или оповестите всех о том, что имя файла изменилось? Но тех, кто *использует* программу, не должно интересовать, на каком языке она написана. А значит, ее проще было с самого начала назвать *bowling*.

[1] В двух словах: они не позволяют командному процессору запустить другую программу (или встроенную команду) с тем же именем. Многие новички совершают распространенную ошибку, называя свою первую программу `test`. Во многих системах уже имеется программа (или встроенная команда) с таким именем; именно она и будет запущена вместо написанной программы.

```
#!/usr/bin/perl

use 5.010;
say "Hello World!";
```

Эта программа работает только в Perl 5.10. Представляя специфические возможности Perl 5.10 в книге, мы явно укажем на это в тексте и включим директиву `use 5.010`, чтобы напомнить лишний раз. Perl считает, что дополнительная версия задается трехзначным числом, поэтому используйте запись `use 5.010` вместо `use 5.10` (Perl решит, что это версия `5.100`, а ее определенно не существует!).

Из чего состоит программа?

Как и другие языки «свободного формата», Perl позволяет включать незначащие пропуски (пробелы, знаки табуляции и переводы строки), упрощающие чтение кода. Впрочем, большинство программ Perl пишется в относительно стандартном формате (наподобие того, который использовался в нашем примере). Мы настоятельно рекомендуем расставлять отступы в программах, так как с ними код проще читается; хороший текстовый редактор выполнит большую часть работы за вас. Хорошие комментарии также упрощают чтение программы. В Perl комментарий начинается от знака # и следует до конца строки (блочные комментарии в Perl отсутствуют[1]). Мы не будем подробно комментировать приводимые примеры, потому что принципы их работы подробно объясняются в сопроводительном тексте. Вы же используйте комментарии в своих программах настолько подробно, насколько потребуется.

Итак, ту же программу «Hello, world» можно записать другим способом (надо сказать, довольно странным):

```
#!/usr/bin/perl
    print    # Комментарий
"Hello, world!\n"
  ;    # Ваш код Perl не должен так выглядеть!
```

Первая строка в действительности содержит специальный комментарий. В семействе UNIX[2], если первая строка текстового файла начинается с символов `#!`, за ними следует имя программы, выполняющей остальную часть файла. В данном случае программа хранится в файле */usr/bin/perl*.

Строка `#!` в действительности является наименее портируемой частью Perl-программы, так как ее содержимое зависит от конкретного компьютера. К счастью, она почти всегда содержит имя */usr/bin/perl* или

---

[1] Хотя существуют разные способы их имитации; см. FAQ (*perldoc perlfaq* в большинстве установок).

[2] По крайней мере в современных системах этого семейства. «Стандартный» механизм появился где-то в середине 1980-х годов, а это довольно долгий срок, даже на относительно длинной временной шкале UNIX.

/usr/local/bin/perl. Если вам оно не подходит, узнайте, где в вашей системе хранится perl, и используйте этот путь. В системах семейства UNIX можно воспользоваться стандартной строкой, которая найдет perl за вас:

```
#!/usr/bin/env perl
```

Если системе не удастся найти perl в пути поиска, возможно, вам придется обратиться к системному администратору или другому пользователю той же системы.

В других системах первая строка традиционно состоит из символов `#!perl`. И эти символы даже приносят пользу – они хотя бы сообщают программисту, занимающемуся сопровождением, что перед ним программа Perl.

Если строка `#!` содержит неверные данные, командный процессор обычно выдает ошибку. Иногда она выглядит довольно неожиданно – скажем, сообщение «файл не найден». Однако это не значит, что системе не удалось найти вашу программу; это файл */usr/bin/perl* не был найден там, где ему следовало быть. Мы бы охотно сделали это сообщение более понятным, но оно поступает не от Perl; «жалуется» командный процессор. (Кстати, будьте внимательны и не напишите *user* вместо *usr*; создатели UNIX не любили вводить лишние символы и часто пропускали буквы.)

Еще одна возможная проблема: если ваша система вообще не поддерживает синтаксис `#!`. В таком случае командный процессор (или то, что использует ваша система), вероятно, попытается запустить вашу программу своими силами; результат вас разочарует или удивит. Если вы не можете понять смысл странного сообщения об ошибке, поищите его в man-странице *perldiag*.

«Основная» программа состоит из всех обычных команд Perl (за исключением кода функций, как вы увидите далее). В Perl нет функции `main`, в отличие от таких языков, как C и Java. Многие программы вообще не содержат внутренних функций.

Также не существует обязательной секции объявления переменных, как в некоторых языках. Тех, кто привык всегда объявлять переменные, это обстоятельство может встревожить. Однако оно позволяет писать программы «на скорую руку». Если программа состоит всего из двух строк, было бы нерационально тратить одну из них на объявление переменных. Хотя если вы действительно хотите объявлять свои переменные, в главе 4 будет показано, как это делается.

Большинство команд представляет собой выражение, завершаемое символом «точка с запятой» (;). Одна из команд, неоднократно встречающаяся в книге:

```
print "Hello, world!\n";
```

Вероятно, вы уже догадались, что эта строка выводит сообщение `Hello, world!`. Сообщение завершается комбинацией `\n`, хорошо знакомой всем программистам с опытом работы на C, C++ или Java; это символ новой строки. При выводе его после сообщения текущая позиция вывода перемещается к началу следующей строки, так что приглашение командного процессора выводится в отдельной строке, а не присоединяется к сообщению. Каждая выводимая строка должна завершаться символом новой строки. Символ новой строки и другие «комбинации с обратной косой чертой» более подробно описаны в следующей главе.

Как откомпилировать программу, написанную на Perl?

Просто запустите свою программу. Интерпретатор perl компилирует и запускает вашу программу за один шаг:

```
$ perl my_program
```

При запуске программы внутренний компилятор Perl сначала преобразует весь исходный код в *байт-код* (внутренняя структура данных, представляющая программу). Далее ядро обработки байт-кода Perl получает управление и запускает байт-код на выполнение. Если в строке 200 обнаружена синтаксическая ошибка, вы получите сообщение об этом еще до начала выполнения строки 2.[1] Если тело цикла выполняется 5000 раз, компилируется оно всего один раз; это позволяет выполнить цикл с максимальной скоростью. Сколько бы комментариев не использовалось в программе, сколько бы в ней не было пропусков, упрощающих понимание кода, это никак не отразится на скорости работы. Даже если в теле цикла выполняются операции с одними константами, константа-результат вычисляется в начале работы программы, а не при каждом выполнении тела цикла.

Разумеется, компиляция требует времени – было бы неэффективно писать объемистую программу Perl, которая решает одну небольшую операцию (скажем, одну из нескольких возможных операций), а затем завершается. Время выполнения программы окажется ничтожно малым по сравнению с временем компиляции. Однако компилятор работает очень быстро; в общем случае время компиляции составляет малый процент от времени выполнения.

Исключение составляют программы, выполняемые в качестве сценариев CGI; они могут запускаться сотни и тысячи раз в минуту. (Такая частота использования считается очень высокой. Если бы сценарии CGI запускались сотни и тысячи раз *в день*, как большинство программ в Сети, то и причин для беспокойства не было бы.) Многие из этих программ выполняются очень быстро, поэтому проблема повторной компиляции выходит на первый план. Если вы столкнетесь с этой

[1] Если только строка 2 не содержит операцию времени компиляции, например блок `BEGIN` или директиву `use`.

проблемой, найдите способ хранения программы в памяти между вызовами. Возможно, вам поможет расширение `mod_perl` для веб-сервера Apache (*http://perl.apache.org/*) или такие модули Perl, как `CGI::Fast`.

А нельзя ли сохранить откомпилированный байт-код, чтобы избежать затрат на компиляцию? Или еще лучше – преобразовать байт-код на другой язык (скажем, на C) и откомпилировать его? Оба варианта возможны при некоторых обстоятельствах, но, скорее всего, использование, сопровождение, отладка или установка программ от этого не упростится, а может быть, программа даже начнет работать медленнее. Perl 6 в этом отношении должен проявить себя гораздо лучше, хотя сейчас (на момент написания книги) говорить об этом слишком рано.

Perl за две минуты

Хотите увидеть настоящую содержательную программу Perl? (даже если не хотите, все равно придется). Пожалуйста:

```
#!/usr/bin/perl
@lines = `perldoc -u -f atan2`;
foreach (@lines) {
  s/\w<([^>]+)>/\U$1/g;
  print;
}
```

На первый взгляд этот код Perl выглядит довольно странно. (Потом, впрочем, тоже.) Но давайте разберем его строка за строкой, и вы поймете, что здесь происходит. Объяснения будут очень краткими, в противном случае мы не уложимся в две минуты. Все аспекты программы более подробно описаны в других местах книги. Даже если сейчас что-то останется непонятным, это не так важно.

Первая строка, `#!`, нам уже знакома. Возможно, ее придется изменить для вашей системы, как упоминалось ранее.

Вторая строка выполняет внешнюю команду, записанную в обратных апострофах (` `). (На полноразмерных клавиатурах клавиша обратного апострофа часто размещается рядом с цифрой 1. Не путайте обратный апостроф с обычным, '.) В данном случае выполняется команда *perldoc -u -f atan2*; попробуйте ввести ее в командной строке и посмотрите, что она делает. Команда *perldoc* в большинстве систем используется для чтения и вывода документации Perl, расширений и вспомогательных средств, поэтому обычно она присутствует в системе.[1] Эта команда выводит описание тригонометрической функции `atan2`; мы ис-

[1] Если команда *perldoc* недоступна, скорее всего, это означает, что в системе нет интерфейса командной строки, а Perl не может выполнять такие команды (например, *perldoc*) в обратных апострофах или с перенаправлением (piped open) – см. главу 16. В таком случае просто пропустите упражнения, в которых используется *perldoc*.

пользуем ее как пример внешней команды, выходные данные которой обрабатываются программой.

Выходные данные команды в обратных апострофах сохраняются в массиве `@lines`. В следующей строке кода начинается циклический перебор всех строк. Тело цикла выделено отступом. Perl этого не требует, но зато требует хороший стиль программирования.

Первая строка в теле цикла выглядит наиболее устрашающе. Она гласит: `s/\w<([^>]+)>/\U$1/g;`. Не углубляясь в подробности, скажем, что она изменяет любую строку, содержащую специальный маркер в виде угловых скобок (`< >`). В выходных данных *perldoc* должен присутствовать хотя бы один такой маркер.

Следующая строка, как ни удивительно, выводит каждую (возможно, измененную) строку. Итоговый вывод будет близким к тому, который выдается командой *perldoc -u -f atan2*, но все строки с вхождением маркеров будут изменены.

Таким образом, всего в нескольких строках кода мы запустили программу, сохранили ее вывод в памяти, обновили содержимое памяти и вывели его. Подобные программы, преобразующие данные к другому типу, часто встречаются в Perl.

Упражнения

Обычно каждая глава завершается серией упражнений, ответы на которые приводятся в приложении А. На этот раз вам не придется писать программы самостоятельно – они приведены в тексте главы.

Если вам не удастся заставить эти упражнения работать на вашем компьютере, перепроверьте свою работу и обратитесь к местному специалисту. Помните, что программы иногда приходится слегка подправлять так, как описано в тексте главы.

1. [7] Введите программу «Hello, world» и заставьте ее работать! (Имя программы выбирается произвольно, но для простоты можно задать имя `ex1-1` – упражнение 1 из главы 1.)
2. [5] Введите команду *perldoc -i -f atan2* в командной строке; просмотрите выходные данные. Если команда не работает, узнайте у системного администратора или из документации своей версии Perl, как вызвать команду *perldoc* или ее аналог. (Она все равно понадобится для следующего упражнения.)
3. [6] Введите второй пример программы (из предыдущего раздела) и посмотрите, что она выведет. (Подсказка: будьте внимательны и следите за тем, чтобы знаки препинания точно совпадали с приведенными в тексте!) Вы видите, как программа изменила выходные данные команды?

2

Скалярные данные

В разговорной речи мы привыкли различать единственное и множественное числа. Perl – язык программирования, созданный человеком-лингвистом, – ведет себя аналогично. Как правило, когда Perl имеет дело с одним экземпляром чего-либо, говорят, что это *скалярная величина* (или просто *скаляр*[1]). Скалярные данные составляют простейшую разновидность данных, с которыми работает Perl. Большинство скаляров представляет собой либо числа (например, 255 или 3.25e20), либо строки символов (скажем, `hello`[2] или полный текст гетисбергской речи). На наш взгляд, строки и числа – совершенно разные вещи, но Perl использует их как практически взаимозаменяемые объекты.

К скалярным значениям можно применять операторы (сложение, конкатенация и т. д.); как правило, результат также является скалярным значением. Скалярное значение может храниться в скалярной переменной. Скалярные данные могут читаться из файлов и устройств и могут записываться на них.

Числа

Скалярная переменная может содержать как число, так и строку, но на данный момент нам будет удобно рассматривать числа и строки по отдельности. Начнем с чисел, а затем перейдем к строкам.

---

[1] Скаляры Perl не имеют особого отношения к аналогичному термину из области математики или физики; «векторов» в Perl нет.

[2] Читатели с опытом программирования на других языках могут воспринимать `hello` как последовательность из пяти символов, а не как единое целое. Но в Perl вся строка является единым скалярным значением. Конечно, при необходимости можно обратиться к отдельным символам этой строки; о том, как это делается, будет рассказано в следующих главах.

Все числа хранятся в одном внутреннем формате

Как вы вскоре увидите, в программах Perl могут использоваться как целые (255, 2001 и т. д.), так и вещественные числа (в формате с десятичной точкой – 3.14159, 1.35×10^{25} и т. д.). Но во внутренней реализации Perl работает с вещественными числами двойной точности[1]. Это означает, что Perl в своих внутренних операциях не использует целые числа – целочисленная константа в программе интерпретируется как эквивалентное вещественное значение[2]. Скорее всего, вы не заметите эти преобразования (да они и не важны для вас), но не пытайтесь искать отдельные целочисленные операции (в отличие от *вещественных* операций) – их просто не существует.[3]

Вещественные литералы

Литералом называется представление значения в исходном коде программы Perl. Литерал не является результатом вычисления или операции ввода/вывода; это данные, непосредственно «вписанные» в исходный код программы.

Вещественные литералы Perl выглядят знакомо. Допускаются числа с дробной частью и без нее (а также с необязательным знаковым префиксом «плюс» или «минус») и экспоненциальная запись в формате «E – степень 10». Примеры:

```
1.25
255.000
255.0
7.25e45  # 7.25 умножить на 10 в 45 степени (большое число)
-6.5e24  # минус 6.5 умножить на 10 в 24 степени
         # (большое отрицательное число)
-12e-24  # минус 12 умножить на 10 в -24 степени
         # (очень маленькое отрицательное число)
-1.2E-23 # то же в другом формате - буква E может быть прописной
```

---

[1] Вещественное число двойной точности – формат, используемый компилятором C, которым компилируется Perl, для объявления `double`. Формально разрядность числа зависит от компьютера, но в большинстве современных систем используется формат IEEE-754 с 15-разрядной точностью и диапазоном представления как минимум от 1e-100 до 1e100.

[2] Вообще-то Perl иногда использует внутренние целые числа, но делает это незаметно для программиста. Внешне проявляется только одно отличие: программа работает быстрее. А кто станет на это жаловаться?

[3] Хорошо, директива `integer` существует, однако ее применение выходит за рамки книги. Да, некоторые операции вычисляют целое значение по заданному вещественному числу. Но сейчас речь идет о другом.

Целочисленные литералы

Целочисленные литералы тоже вполне тривиальны:

```
0
2001
-40
255
61298040283768
```

Последнее число трудно прочитать. Perl разрешает включать в целочисленные литералы знаки подчеркивания, так что число может быть записано в следующем виде:

```
61_298_040_283_768
```

Это то же значение; просто оно выглядит немного иначе для нас, людей. Возможно, вы бы предпочли использовать для разделения запятые, но они уже задействованы в Perl для более важных целей (см. следующую главу).

Недесятичные целочисленные литералы

Как и многие другие языки программирования, Perl позволяет задавать числа в других системах счисления кроме десятичной (по основанию 10). Восьмеричные литералы (основание 8) начинаются с `0`, шестнадцатеричные (основание 16) начинаются с префикса `0x`, а двоичные (основание 2) – с префикса `0b`.[1] Шестнадцатеричные цифры от `A` до `F` (или от `a` до `f`) представляют десятичные значения от 10 до 15. Пример:

```
0377         # 377 в восьмеричной записи; то же, что 255 десятичное
0xff         # FF в шестнадцатеричной записи; тоже 255 десятичное
0b11111111   # тоже 255 десятичное
```

Нам, людям, эти числа кажутся разными, но для Perl все три числа одинаковы. С точки зрения Perl абсолютно неважно, напишете ли вы `0xFF` или `255.000`, поэтому выберите то представление, которое будет наиболее логичным для вас и вашего специалиста по сопровождению (того несчастного, которому придется разбираться, что вы имели в виду при написании своего кода; чаще всего им оказываетесь вы сами, причем начисто не помните, почему три месяца назад вы поступили именно так!)

[1] «Начальный ноль» работает только в литералах – как вы узнаете позднее в этой главе, он не поддерживается в автоматических преобразованиях строк в числа. Строка данных с восьмеричным или шестнадцатеричным значением преобразуется в число функцией `oct()` или `hex()`. Аналогичной функции `bin()` для преобразования двоичных чисел не существует, но функция `oct()` выполняет такое преобразование для строк, начинающихся с `0b`.

Если недесятичный литерал содержит более четырех символов, его чтение также затрудняется. Perl разрешает для наглядности включать в такие литералы символы подчеркивания:

```
0x1377_0B77
0x50_65_72_7C
```

Числовые операторы

Perl поддерживает стандартный набор математических операторов: сложение, вычитание, умножение, деление и т. д. Пример:

```
2 + 3         # 2 плюс 3 = 5
5.1 - 2.4     # 5.1 минус 2.4 = 2.7
3 * 12        # 3 умножить на 12 = 36
14 / 2        # 14 разделить на 2 = 7
10.2 / 0.3    # 10.2 разделить на 0.3 = 34
10 / 3        # деление всегда в вещественном формате, поэтому 3.3333333...
```

Perl также поддерживает оператор *вычисления остатка* (%). Значение выражения `10 % 3` равно остатку от деления 10 на 3 нацело, то есть 1. Оба операнда сначала усекаются до целой части, так что выражение `10.5 % 3.2` вычисляется в формате `10 % 3`.[1] Также в Perl имеется оператор *возведения в степень* в стиле FORTRAN, который многим хотелось бы видеть в Pascal и C. Оператор изображается двойной звездочкой, например `2**3` соответствует двум в третьей степени, то есть 8.[2] Существуют и другие числовые операторы, которые будут представляться по мере надобности.

Строки

Строка представляет собой последовательность символов (например, `hello`). Строки могут содержать произвольную комбинацию произвольных символов.[3] Самая короткая строка не содержит ни одного символа, а самая длинная заполняет всю свободную память (хотя с такой строкой ничего полезного сделать не удастся). Это соответствует политике «отсутствия искусственных ограничений», которой Perl следует при любой возможности. Типичная строка представляет собой по-

---

[1] Результат оператора вычисления остатка, в котором задействован отрицательный операнд (или оба операнда), изменяется в зависимости от реализации. Будьте внимательны.

[2] В общем случае отрицательное число не может возводиться в неотрицательную степень. Математики знают, что результатом будет комплексное число. Чтобы такая операция стала возможной, необходимо прибегнуть к помощи модуля `Math::Complex`.

[3] В отличие от C или C++, символ NUL в Perl не выполняет специальных функций, так как конец строки в Perl определяется счетчиком длины, а не нуль-байтом.

следовательность печатных букв, цифр и знаков препинания в диапазоне от ASCII 32 до ASCII 126. Тем не менее возможность включения любых символов в строку означает, что вы можете создавать, сканировать и обрабатывать двоичные данные в строковом формате – в других языках и утилитах эта возможность реализуется с большим трудом. Например, вы можете прочитать графическое изображение или откомпилированную программу в строку Perl, внести изменения и записать результат обратно.

Строки, как и числа, имеют литеральное представление (то есть способ представления в программном коде). Строковые литералы делятся на две разновидности: *строковые литералы в апострофах* и *строковые литералы в кавычках*.

Строковые литералы в апострофах

Строковый литерал в апострофах представляет собой последовательность символов, заключенную в апострофы. Сами апострофы в строку не входят – они всего лишь помогают Perl определить позиции начала и конца строки. Любой символ, отличный от апострофа или обратной косой черты (включая символы новой строки, если строка продолжается в нескольких логических строках), означает «самого себя». Чтобы включить в строку символ \ (обратная косая черта), поставьте два символа \ подряд. Другими словами:

```
'fred'    # Четыре символа: f, r, e и d
'barney'  # Шесть символов
''        # Пустая строка (без символов)
'Don\'t let an apostrophe end this string prematurely!'
'the last character of this string is a backslash: \\'
'hello\n' # hello, затем обратная косая черта и n
'hello
there'    # hello, новая строка, there (всего 11 символов)
'\'\\'    # Апостроф, за которым следует обратная косая черта
```

Обратите внимание: `\n` в строке, заключенной в апострофы, интерпретируется не как новая строка, а как два символа: обратная косая черта и `n`. Обратная косая черта интерпретируется особым образом только в том случае, если за ней следует другая обратная косая черта или апостроф.

Строковые литералы в кавычках

Строковые литералы в кавычках напоминают строки, знакомые нам по другим языкам программирования. Они тоже представляют собой последовательности символов, но заключенные в кавычки вместо апострофов. Но на этот раз обратная косая черта в полной мере используется для определения управляющих символов и даже конкретных символов в восьмеричном или шестнадцатеричном коде. Примеры строк в кавычках:

```
"barney"        # То же, что 'barney'
"hello world\n" # hello world и новая строка
"The last character of this string is a quote mark: \""
"coke\tsprite"  # coke, символ табуляции и sprite
```

Строковый литерал в кавычках `"barney"` обозначает для Perl ту же строку из шести символов, что и строковый литерал в апострофах `'barney'`. Примерно то же мы видели при знакомстве с числовыми литералами, когда было показано, что литерал `0377` является другим способом записи `255.0`. Perl позволяет записать литерал тем способом, который будет для вас более удобным. Конечно, если вы хотите использовать управляющие комбинации с обратной косой чертой (например, `\n` для обозначения символа новой строки), необходимо использовать кавычки.

Обратная косая черта перед разными символами изменяет их смысл. Почти полный[1] список управляющих комбинаций с обратной косой чертой приведен в табл. 2.1.

Таблица 2.1. Управляющие комбинации для строковых литералов в кавычках

Конструкция	Описание
`\n`	Новая строка
`\r`	Возврат курсора
`\t`	Табуляция
`\f`	Подача страницы
`\b`	Backspace
`\a`	Звуковой сигнал
`\e`	Escape (символ Escape в ASCII)
`\007`	Любой восьмеричный код ASCII (007 = звуковой сигнал)
`\x7f`	Любой шестнадцатеричный код ASCII (7f = звуковой сигнал)
`\cC`	Символ «Control» (в данном случае Control-C)
`\\`	Обратная косая черта
`\"`	Кавычка
`\l`	Следующая буква преобразуется к нижнему регистру
`\L`	Все последующие буквы преобразуются к нижнему регистру до `\E`
`\u`	Следующая буква преобразуется к верхнему регистру
`\U`	Все последующие буквы преобразуются к верхнему регистру до `\E`

[1] В последних версиях Perl появились управляющие комбинации для задания символов Юникода, которые мы рассматривать не будем.

Конструкция	Описание
\Q	Все символы, не являющиеся символами слов, экранируются добавлением обратной косой черты до \E
\E	Завершает действие режимов \L, \U или \Q

У строк в кавычках есть и другая особенность: они поддерживают *интерполяцию переменных*, то есть имена переменных при использовании строки заменяются их текущими значениями. Впрочем, наше формальное знакомство с переменными еще не состоялось, так что мы вернемся к этой теме позже.

Строковые операторы

Конкатенация строк выполняется оператором . (да, просто точка). Естественно, сами строки при этом не изменяются – по аналогии с тем, как операция сложения 2+3 не изменяет операндов 2 и 3. Полученная (объединенная) строка используется в последующих вычислениях или присваивается переменной. Примеры:

```
"hello" . "world"       # То же, что "helloworld"
"hello" . ' ' . "world" # То же, что 'hello world'
'hello world' . "\n"    # То же, что "hello world\n"
```

Обратите внимание на необходимость присутствия оператора конкатенации (точки): в некоторых языках достаточно разместить два строковых значения рядом друг с другом.

К категории строковых операторов относится оператор *повторения строки*, который представляется в виде одной строчной буквы x. Оператор берет левый операнд (строку) и создает конкатенацию ее экземпляров, заданных правым операндом (числом). Пример:

```
"fred" x 3       # "fredfredfred"
"barney" x (4+1) # "barney" × 5, или "barneybarneybarneybarneybarney"
5 × 4            # на самом деле "5" × 4, то есть "5555"
```

На последнем примере стоит остановиться подробнее. Оператор повторения строки ожидает получить левый операнд в строковом формате, поэтому число 5 преобразуется в строку "5" (по правилам, которые подробно описываются далее). Созданная строка из одного символа копируется четыре раза, в результате чего создается строка из четырех символов 5555. Если переставить операнды 4×5, оператор создаст пять копий строки 4, то есть 44444. Пример наглядно показывает, что повторение строк некоммутативно.

Количество копий (правый операнд) перед использованием усекается до целой части (4.8 превращается в 4). При количестве копий меньше 1 создается пустая строка (строка нулевой длины).

Автоматическое преобразование между числами и строками

Как правило, Perl автоматически выполняет преобразования между числами и строками по мере надобности. Как он узнает, какой именно формат нужен? Все зависит от оператора, применяемого к скалярному значению. Если оператор получает число (как оператор +), Perl воспринимает значение как число. Если оператор получает строку (как оператор .), Perl воспринимает значение как строку. Таким образом, вам не придется отвлекаться на различия между строками и числами. Используйте нужные операторы, а Perl сделает все остальное.

Если строковое значение встречается в контексте, в котором оператор ожидает получить число (скажем, при умножении), Perl автоматически преобразует строку в эквивалентное числовое значение, как если бы оно было задано в десятичном вещественном формате.[1] Таким образом, выражение `"12" * "3"` дает результат `36`. Завершающие символы, не используемые при записи чисел, и начальные пропуски игнорируются, поэтому выражение `"12fred34" * " 3"` тоже даст результат `36` без каких-либо проблем.[2] Значение, которое не является числом, преобразуется в нуль. Например, это произойдет при использовании вместо числа строки `"fred"`.

Аналогичным образом при использовании числового значения там, где должна быть строка (скажем, при конкатенации), число преобразуется в свое строковое представление. Например, если вы хотите объединить строку `Z` с результатом умножения 5 на 7[3], это делается совсем просто:

```
"Z" . 5 * 7      # То же, что "Z" . 35 или "Z35"
```

Короче, вам не нужно беспокоиться о том, с чем вы работаете – с числом или со строкой (в большинстве случаев). Perl выполнит все преобразования за вас.[4]

Встроенные предупреждения Perl

Вы можете потребовать, чтобы Perl предупреждал вас о выполнении каждой сомнительной операции в вашей программе. Чтобы запустить

---

1 Помните, что трюк с начальным нулем, обозначающим недесятичные значения, работает для литералов, но никогда при автоматическом преобразовании. Для преобразования таких строк используются функции `hex()` и `oct()`.

2 Если только в программе не включен режим выдачи предупреждений, но об этом чуть позже.

3 Приоритеты операций и круглые скобки рассматриваются ниже.

4 Если вас беспокоит эффективность, не тревожьтесь. Perl обычно запоминает результат преобразования, так что оно выполняется только один раз.

программу с выводом предупреждений, вставьте в командную строку ключ `-w`:

```
$ perl -w my_program
```

А если режим предупреждений должен действовать при каждом запуске программы, добавьте его в строку `#!`:

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

Этот способ работает даже в системах, не входящих в семейство UNIX, где традиционно используется запись следующего вида (так как путь к Perl в общем случае не имеет значения):

```
#!perl -w
```

В Perl 5.6 и выше предупреждения могут включаться директивой `use` (но будьте осторожны – такое решение не будет работать в более ранних версиях Perl):[1]

```
#!/usr/bin/perl
use warnings;
```

Теперь Perl предупредит о попытке использования строки `'12fred34'` в числовом контексте:

```
Argument "12fred34" isn't numeric
```

Конечно, предупреждения обычно предназначены для программистов, а не для конечных пользователей. Если предупреждение останется незамеченным программистом, вероятно, оно окажется бесполезным. Кроме того, предупреждения не должны изменять поведение программы. Если вы получили предупреждение, смысл которого вам непонятен, получите более подробное описание проблемы при помощи прагмы[2] `diagnostics`. В man-странице *perldiag* приводится как короткое, так и длинное диагностическое описание:

```
#!/usr/bin/perl
use diagnostics;
```

При включении директивы `use diagnostics` запуск программы сопровождается небольшой паузой. Дело в том, что программе приходится выполнять большой объем дополнительной работы (а также выделять большой блок памяти) на случай, если вы захотите прочитать документацию сразу же после обнаружения ошибки. Это открывает возможность применения оптимизации, ускоряющей запуск программы (и расходы памяти), без отрицательных последствий для пользователя;

---

[1] Вообще говоря, директива `warnings` разрешает лексические предупреждения; о том, что это такое, можно узнать в *perllexwarn*.

[2] Называть pragma «директивой» некорректно; `use diagnostics` – директива, а `diagnostics` – прагма (особая разновидность модулей). – *Примеч. перев.*

когда надобность в чтении документации по предупреждениям отпадет, удалите из программы директиву `use diagnostics`. (А еще лучше – исправьте программу, чтобы она не выдавала предупреждения.)

Возможна и дальнейшая оптимизация: параметр командной строки Perl `-M` позволяет загружать директиву только в случае необходимости. С этим параметром вам не придется редактировать исходный код для включения и отключения режима `diagnostics`:

```
$ perl -Mdiagnostics ./my_program
Argument "12fred34" isn't numeric in addition (+) at ./my_program line 17 (#1)
    (W numeric) The indicated string was fed as an argument to
    an operator that expected a numeric value instead.  If you're
    fortunate the message will identify which operator was so unfortunate.
```

Ситуации, в которых Perl обычно может предупредить о возможной ошибке в коде, будут особо упоминаться в тексте. Однако не стоит рассчитывать на то, что тексты и поведение этих предупреждений останутся неизменными в будущих версиях Perl.

Скалярные переменные

Переменная представляет собой имя блока памяти для хранения одного или нескольких значений.[1] Имя переменной остается неизменным, но содержащиеся в ней данные обычно многократно изменяются на протяжении жизненного цикла программы.

Скалярная переменная, как и следовало ожидать, содержит одно скалярное значение. Имена скалярных переменных начинаются со знака доллара (`$`), за ним следует так называемый *идентификатор Perl*: буква или символ подчеркивания, за которым следуют другие буквы, цифры или символы подчеркивания. Можно также сказать, что идентификатор состоит из алфавитно-цифровых знаков и подчеркиваний, но не может начинаться с цифры. Регистр символов в именах переменных учитывается; `$Fred` и `$fred` – две разные переменные. В имени переменной учитываются все символы, так что переменная

```
$a_very_long_variable_that_ends_in_1
```

отличается от переменной

```
$a_very_long_variable_that_ends_in_2
```

Имена скалярных переменных всегда начинаются с префикса `$`.[2] В командном процессоре префикс `$` используется для получения значения, но при присваивании он опускается. В *awk* и C знак `$` вообще не ис-

[1] Скалярная переменная может содержать только одно значение, но другие типы переменных, например массивы и хеши, позволяют хранить несколько значений.

[2] «Сигил» (sigil) на жаргоне Perl.

пользуется. При частых переходах вы будете периодически ошибаться с префиксами, это нормально. (Многие программисты Perl рекомендуют вообще отказаться от программирования для командного процессора, *awk* и C, но это не всегда возможно.)

Выбор имен переменных

Как правило, переменным рекомендуется присваивать имена, которые как-то описывают их назначение. Например, `$r` – не очень содержательное имя, а `$line_length` выглядит более понятно. Если переменная встречается только в двух-трех строках, расположенных вблизи друг от друга, можно ограничиться простым именем, вроде `$n`, но переменным, используемым в разных местах программы, лучше присвоить более содержательные имена.

Правильная расстановка символов подчеркивания также упростит чтение и понимание программы, особенно если сопровождением занимается программист, который говорит на другом языке. Например, имя `$super_bowl` лучше `$superbowl`, потому что последнее можно спутать с `$superb_owl`. А что означает имя `$stopid`: `$sto_pid` (хранилище для идентификатора процесса – STOrage for Process IDentifier), `$s_to_pid` (преобразование s в идентификатор процесса) или `$stop_id` (идентификатор некоего «стоп-объекта»)? А может, просто `$stupid` (глупый), написанное с ошибкой?

Имена переменных в программах Perl обычно записываются строчными буквами, как большинство имен в книге. В некоторых особых случаях используются прописные буквы. Запись имени переменной в верхнем регистре (например, `$ARGV`) указывает на особую роль переменной. Если имя переменной состоит из нескольких слов, одни программисты предпочитают запись `$underscores_are_cool`, а другие – `$giveMeInitialCaps`. Главное – будьте последовательны.

Конечно, удачный или неудачный выбор имен абсолютно неважен для Perl. Никто не запрещает вам назвать три главные переменные программы `$OOOoooOOO`, `$OOooOOoo` и `$0o0o0o0o0`; это нисколько не смутит Perl, но тогда не просите нас заниматься сопровождением вашего кода.

Скалярное присваивание

Самой частой операцией со скалярными переменными является *операция присваивания*, то есть задания значения переменной. Оператор присваивания в Perl записывается в виде знака = (как и во многих других языках), слева от которого указывается имя переменной, а справа – выражение, определяющее ее значение. Примеры:

```
$fred   = 17;          # Присвоить $fred значение 17
$barney = 'hello';     # Присвоить $barney строку из 5 символов 'hello'
$barney = $fred + 3;   # Присвоить $barney текущее значение $fred
                       # плюс 3 (20)
```

```
$barney = $barney * 2; # $barney заменяется своим значением,
                       # умноженным на 2 (40)
```

Обратите внимание на последнюю строку, в которой переменная `$barney` используется дважды: сначала для получения текущего значения (справа от знака равенства), а затем для определения места хранения вычисленного результата (слева от знака равенства). Такое использование вполне законно, безопасно и распространено. Более того, оно применяется настолько часто, что для него создана удобная сокращенная запись, представленная в следующем разделе.

Бинарные операторы присваивания

Такие выражения, как `$fred = $fred + 5` (когда одна переменная присутствует в обеих частях присваивания), настолько часто встречаются в Perl (а также в C и Java), что появилось удобное сокращение для операции изменения переменной — *бинарный оператор присваивания*. Почти все бинарные операторы, вычисляющие новое значение, имеют соответствующую форму бинарного присваивания с суффиксом `=`. Например, следующие две строки эквивалентны:

```
$fred  = $fred + 5; # Без бинарного оператора присваивания
$fred += 5;         # С бинарным оператором присваивания
```

Еще один пример эквивалентных команд:

```
$barney  = $barney * 3;
$barney *= 3;
```

В каждом случае оператор каким-то образом изменяет текущее значение переменной вместо простой его замены результатом нового выражения.

Другой распространенный оператор присваивания создан на базе оператора конкатенации строк (`.`); комбинация двух операций дает оператор присоединения (`.=`):

```
$str  = $str . " "; # Присоединить пробел к $str
$str .= " ";        # То же с оператором присваивания
```

Почти все бинарные операторы существуют в форме с присваиванием. Например, *оператор возведения в степень* записывается в виде `**=`. Таким образом, `fred **= 3` означает «возвести `$fred` в третью степень и поместить результат обратно в `$fred`».

Вывод командой print

Как правило, любая программа выводит данные; без них создается впечатление, что программа вообще ничего не делает. Оператор `print()` открывает такую возможность: он получает скалярный аргумент и по-

мещает его без какой-либо дополнительной обработки в стандартный вывод. Если вы не ошиблись с аргументом, данные будут выведены на терминале. Пример:

```
print "hello world\n"; # Строка hello world, за которой следует
                       # символ новой строки

print "The answer is ";
print 6 * 7;
print ".\n";
```

При вызове `print` можно передать серию значений, разделенных запятыми:

```
print "The answer is ", 6 * 7, ".\n";
```

Фактически `print` получает *список*, но поскольку мы еще не обсуждали списки, этот вопрос откладывается на будущее.

Интерполяция скалярных переменных в строках

Строковые литералы в кавычках поддерживают *интерполяцию переменных*[1] (наряду с поддержкой управляющих комбинаций с обратной косой чертой). Другими словами, имя любой скалярной переменной[2] в строке заменяется ее текущим значением. Пример:

```
$meal   = "brontosaurus steak";
$barney = "fred ate a $meal";    # Теперь $barney содержит
                                 # "fred ate a brontosaurus steak"
$barney = 'fred ate a ' . $meal; # Другой способ добиться того же результата
```

Как видно из последней строки этого фрагмента, того же результата можно добиться и без заключения строки в кавычки, однако такой синтаксис часто обеспечивает более удобную запись. Если скалярной переменной еще не было присвоено никакое значение[3], используется пустая строка:

```
$barney = "fred ate a $meat"; # Теперь $barney содержит "fred ate a"
```

Если выводится только значение одной переменной, не стоит возиться с интерполяцией:

```
print "$fred"; # Лишние кавычки
print $fred;   # То же, но с лучшим стилем
```

---

1 Не имеет никакого отношения к интерполяции в математике или статистике.

2 А также некоторых других типов переменных, но об этом позже.

3 На самом деле такая переменная имеет специальное неопределенное значение `undef`, о котором будет рассказано позднее в этой главе. При включении предупреждений Perl пожалуется на интерполяцию неопределенного значения.

Ничто не мешает заключить одинокую переменную в кавычки, но другие программисты будут смеяться у вас за спиной.[1] Интерполяция переменных также известна под названием «интерполяции в кавычках», потому что она происходит при использовании кавычек (но не апострофов). Интерполяция также выполняется в других разновидностях строк Perl, которые еще не рассматривались.

Чтобы включить «настоящий» знак доллара в строку, заключенную в кавычки, поставьте перед ним обратную косую черту, отменяющую особый смысл знака доллара:

```
$fred = 'hello';
print "The name is \$fred.\n";    # Выводит строку со знаком доллара
print 'The name is $fred' . "\n"; # То же самое
```

При интерполяции используется самое длинное имя переменной, имеющее смысл в данной части строки. Это может создать проблемы, если за подставленным значением должен следовать константный текст, начинающийся с буквы, цифры или символа подчеркивания.[2] В ходе сканирования имен переменных Perl сочтет эти символы продолжением имени, а это нежелательно. Просто *заключите* имя переменной в фигурные скобки. Строку можно также разбить надвое с использованием оператора конкатенации:

```
$what = "brontosaurus steak";
$n = 3;
print "fred ate $n $whats.\n";          # Не steaks, а значение $whats
print "fred ate $n ${what}s.\n";        # Теперь используется $what
print "fred ate $n $what" . "s.\n";     # Другой способ с тем же результатом
print 'fred ate ' . $n . ' ' . $what . "s.\n"; # Особенно неудобный способ
```

Приоритет операторов и ассоциативность

Приоритет операторов определяет порядок их выполнения в сложной последовательности. Например, какая операция должна выполняться сначала в выражении `2+3*4` — сложение или умножение? Если сначала выполняется сложение, получается `5*4`, то есть 20. Но если сначала выполняется умножение (как вас учили на уроках математики), получится `2+12`, то есть 14. К счастью, Perl выбирает стандартное математи-

---

[1] Разве что значение будет интерпретироваться как строка, а не как число. За исключением редких случаев, в которых это имеет смысл, это лишь увеличивает количество набираемых символов.

[2] Возможны проблемы и с другими символами. Если сразу же за именем скалярной переменной должна следовать левая квадратная или фигурная скобка, поставьте перед ней обратную косую черту. Этот прием помогает и в том случае, если за именем переменной следует апостроф или пара двоеточий; также можно воспользоваться фигурными скобками, как описано в тексте.

ческое определение и выполняет умножение первым. Говорят, что умножение обладает *более высоким приоритетом*, чем сложение.

Стандартный приоритет может переопределяться при помощи круглых скобок. Все, что находится в круглых скобках, полностью обрабатывается перед применением оператора за пределами скобок (опять же как вас учили на уроках математики). Итак, если вы хотите, чтобы сложение выполнялось перед умножением, используйте запись `(2+3)*4`; получается 20. А если вам вдруг захочется продемонстрировать, что умножение выполняется перед сложением, добавьте декоративную, хотя и ненужную пару круглых скобок, например `2+(3*4)`.

Для умножения и сложения разобраться с приоритетами несложно, но когда, скажем, конкатенация строк смешивается с возведением в степень, начинаются проблемы. Правильный путь решения таких проблем – обращение к официальной, не допускающей посторонних толкований таблице приоритетов операторов Perl (табл. 2.2).[1] (Некоторые операторы еще не упоминались в тексте, а может быть, и вовсе не появятся в этой книге, но это не помешает вам прочитать о них в man-странице *perlop*.)

Таблица 2.2. Ассоциативность и приоритет операторов (в порядке убывания)

Ассоциативность	Описание
левая	круглые скобки и аргументы списочных операторов
левая	`->`
	`++ --` (автоинкремент и автодекремент)
правая	`**`
правая	`\ ! ~ + -` (унарные операторы)
левая	`=~ !~`
левая	`* / % x`
левая	`+ - .` (бинарные операторы)
левая	`<< >>`
	именованные унарные операторы (`-X filetests, rand`)
	`< <= > >= lt le gt ge`
	`== != <=> eq ne cmp`
левая	`&`
левая	`| ^`
левая	`&&`

[1] Радуйтесь, программисты C! Операторы, присутствующие одновременно в Perl и C, обладают одинаковыми приоритетами и ассоциативностью.

Таблица 2.2 (продолжение)

Ассоциативность	Описание
левая	`||` `.. ...`
правая	`?:` (тернарный)
правая	`= += -= .=` (и другие аналогичные операторы присваивания)
левая	`, =>` списочные операторы (слева направо)
правая	`not`
левая	`and`
левая	`or xor`

В таблице каждый оператор обладает более высоким приоритетом, чем операторы, находящиеся под ним, и более низким приоритетом по отношению ко всем предшествующим операторам. Порядок выполнения операторов с одинаковым приоритетом определяется по правилам *ассоциативности*.

Ассоциативность, как и приоритет, определяет порядок выполнения, когда два оператора с одинаковым приоритетом «конкурируют» за три операнда:

```
4 ** 3 ** 2 # 4 ** (3 ** 2), или 4 ** 9 (правоассоциативные)
72 / 12 / 3 # (72 / 12) / 3, или 6/3, то есть 2 (левоассоциативные)
36 / 6 * 3  # (36/6)*3, или 18
```

В первом случае оператор `**` является правоассоциативным, поэтому круглые скобки подразумеваются справа. С другой стороны, операторы `*` и `/` являются левоассоциативными, поэтому и скобки подразумеваются слева.

Нужно ли заучивать таблицу приоритетов? Нет! Никто этого не делает. Если вы забыли порядок следования операций и вам некогда заглядывать в таблицу, используйте круглые скобки. В конце концов, если вы не можете вспомнить приоритет операторов, такие же трудности возникнут и у программиста, занимающегося сопровождением. Пожалейте его: когда-нибудь вы можете оказаться на его месте.

Операторы сравнения

Для сравнения чисел в Perl существуют логические операторы сравнения, напоминающие школьную алгебру: `< <= == >= > !=`. Каждый из этих операторов возвращает логическую истину (*true*) или логическую ложь (*false*). Эти значения более подробно описываются в следующем разделе. Некоторые из этих операторов отличаются от своих аналогов

в других языках. Например, для проверки равенства используется оператор == вместо знака =, задействованного для присваивания. А неравенство двух значений проверяется оператором !=, потому что привычная конструкция <> используется в Perl для других целей. По той же причине условие «больше либо равно» проверяется в Perl оператором >=, а не =>. Оказывается, практически любая комбинация знаков препинания используется в Perl для каких-нибудь целей. Так что если у вас наступит творческий кризис, пустите кошку прогуляться по клавиатуре и отладьте результат.

Для сравнения строк в Perl существует эквивалентный набор операторов, которые напоминают коротенькие слова: lt lg eq ge gt ne. Эти операторы сравнивают две строки символ за символом, проверяя, совпадают ли эти строки полностью или одна из них предшествует другой в стандартном порядке сортировки строк (в ASCII буквы верхнего регистра предшествуют буквам нижнего регистра, так что будьте внимательны).

Операторы сравнения (как для чисел, так и для строк) представлены в табл. 2.3.

Таблица 2.3. Ассоциативность и приоритет операторов (в порядке убывания)

Сравнение	Числовое	Строковое
Равно	==	eq
Не равно	!=	ne
Меньше	<	lt
Больше	>	gt
Меньше либо равно	<=	le
Больше либо равно	>=	ge

Примеры выражений с операторами сравнения:

```
35 != 30 + 5          # false
35 == 35.0            # true
'35' eq '35.0'        # false (сравниваются в строковом виде)
'fred' lt 'barney'    # false
'fred' lt 'free'      # true
'fred' eq "fred"      # true
'fred' eq 'Fred'      # false
' ' gt ''             # true
```

Управляющая конструкция if

Вероятно, после сравнения двух значений вы захотите выполнить некоторые действия по результатам проверки. В Perl, как и в других языках, существует управляющая конструкция if:

```
if ($name gt 'fred') {
  print "'$name' comes after 'fred' in sorted order.\n";
}
```

Как альтернатива предусмотрена разновидность `if` с ключевым словом `else`:

```
if ($name gt 'fred') {
  print "'$name' comes after 'fred' in sorted order.\n";
} else {
  print "'$name' does not come after 'fred'.\n";
  print "Maybe it's the same string, in fact.\n";
}
```

Условный код обязательно должен быть заключен в фигурные скобки, обозначающие границы блока (в отличие от языка C, знаете вы его или нет). В программах содержимое блоков рекомендуется выделять отступом, как это сделано в приведенной программе. Если вы используете текстовый редактор для программистов (см. главу 1), он сделает большую часть работы за вас.

Логические значения

В условии управляющей конструкции `if` может использоваться любое скалярное значение. В частности, это позволяет сохранить логический результат проверки (*true* или *false*) в переменной:

```
$is_bigger = $name gt 'fred';
if ($is_bigger) { ... }
```

Но как Perl определяет, какому логическому значению соответствует некоторое значение – *true* или *false*? В Perl не существует отдельного логического (булевского) типа данных, как в некоторых языках. Вместо этого Perl использует несколько простых правил:[1]

- Если значение является числом, 0 соответствует *false*; все остальные числа соответствуют *true*.
- Если значение является строкой, пустая строка (`''`) соответствует *false*; все остальные строки соответствуют *true*.
- В остальных случаях (то есть если значение является другой разновидностью скалярных данных, а не числом или строкой) значение преобразуется в число или строку, после чего интерпретируется повторно.[2]

---

[1] Конечно, Perl работает по несколько иной логике. Мы всего лишь привели правила, которые позволяют придти к тому же результату.

[2] Отсюда следует, что `undef` (см. далее) соответствует *false*, а все ссылки (которые подробно рассматриваются в «книге с альпакой») соответствуют *true*.

В этих правилах скрыта одна тонкость. Поскольку строка '0' соответствует такому же скалярному значению, что и число 0, Perl интерпретирует их абсолютно одинаково. Это означает, что строка '0' – единственная непустая строка, которая интерпретируется как *false*.

Если вам потребуется получить значение, обратное имеющемуся логическому значению, используйте унарный оператор отрицания !. Если переменная в следующем фрагменте истинна, оператор возвращает *false*; для ложной переменной возвращается *true*:

```
if (! $is_bigger) {
  # Действия для ложного значения $is_bigger
}
```

Получение данных от пользователя

Вероятно, на этой стадии вас уже заинтересовало, как ввести значение с клавиатуры в программу Perl. Вот самый простой способ: используйте оператор построчного ввода <STDIN>.[1]

Каждый раз, когда вы используете конструкцию <STDIN> в месте, где подразумевается скалярное значение, Perl читает из *стандартного ввода* следующую полную строку (до ближайшего символа новой строки) и использует ее в качестве значения <STDIN>. Термин «стандартный ввод» может иметь много разных интерпретаций, но в типичном случае он обозначает клавиатуру пользователя, запустившего программу. Если в <STDIN> нет данных, готовых к вводу (а так оно обычно и бывает, если только вы не ввели заранее полную строку), программа Perl останавливается и ждет появления символов, завершаемых символом новой строки.[2]

Строковое значение <STDIN> обычно завершается символом новой строки[3], поэтому *можно* поступить так:

```
$line = <STDIN>;
if ($line eq "\n") {
  print "That was just a blank line!\n";
```

---

[1] На самом деле это оператор построчного ввода, примененный к файловому дескриптору STDIN, но мы не имеем права сказать вам об этом, пока не доберемся до файловых дескрипторов (см. главу 5).

[2] Вернее, обычно ожидание ввода осуществляется на уровне системы; Perl ждет данные от системы. Хотя подробности зависят от системы и ее конфигурации, обычно вы можете исправить опечатки клавишей Backspace перед нажатием Return – эту возможность предоставляет система, а не Perl. Если вам необходимы дополнительные возможности управления вводом, загрузите модуль Term::ReadLine из CPAN.

[3] Возможны и исключения: если стандартный входной поток неожиданно завершается в середине строки. Разумеется, с нормальными текстовыми файлами такого не бывает!

```
} else {
  print "That line of input was: $line";
}
```

Но на практике символы новой строки обычно оказываются лишними. Для их устранения используется оператор chomp.

Оператор chomp

Когда вы впервые читаете описание оператора chomp, он кажется ужасно узкоспециализированным. Он работает с переменной. В переменной должна содержаться строка. Если эта строка завершается символом новой строки, chomp удаляет этот символ. Вот (почти) и все, что он делает. Пример:

```
$text = "a line of text\n"; # Или то же из <STDIN>
chomp($text);               # Удаление символа новой строки
```

Но в действительности этот оператор настолько полезен, что вы будете использовать его почти в каждой написанной программе. Даже в таком виде это самый простой способ удаления завершающего символа новой строки из переменной. Однако существует еще одна, более удобная форма использования chomp: каждый раз при введении новой переменной в Perl можно использовать оператор присваивания. Perl сначала выполняет присваивание, а затем использует переменную так, как вы потребуете. Таким образом, самый распространенный сценарий использования chomp выглядит так:

```
chomp($text = <STDIN>); # Прочитать текст без символа новой строки
$text = <STDIN>;        # То же самое...
chomp($text);           # ...но в два этапа
```

На первый взгляд комбинированное использование chomp не кажется таким уж простым – скорее оно выглядит более сложным! Если представить себе происходящее как две операции (прочитать строку, а затем «откусить» от нее символ новой строки), запись в виде двух команд выглядит более естественной. Но если воспринимать ее как одну операцию – прочитать только текст без символа новой строки – запись из одной команды становится более логичной. А поскольку большинство программистов Perl поступает именно так, лучше привыкайте сразу.

В действительности chomp является функцией, а не оператором. Как функция chomp имеет возвращаемое значение – количество удаленных символов. Впрочем, вам это число вряд ли когда-нибудь понадобится:

```
$food = <STDIN>;
$betty = chomp $food; # Получаем значение 1, но это и так известно!
```

Как вы вскоре увидите, вызов chomp может записываться как с круглыми скобками, так и без них. Это еще одно общее правило в Perl: круг-

лые скобки всегда можно убрать – кроме тех случаев, когда это изменяет смысл выражения.

Если строка завершается двумя или более символами новой строки[1], `chomp` удалит только один из них. Если символов новой строки нет, `chomp` не делает ничего и возвращает 0.

Управляющая конструкция while

Как и большинство алгоритмических языков программирования, Perl поддерживает несколько циклических конструкций.[2] Цикл `while` повторяет блок кода до тех пор, пока условие остается истинным:

```
$count = 0;
while ($count < 10) {
  $count += 2;
  print "count is now $count\n"; # Выводит 2 4 6 8 10
}
```

Истинность условия проверяется так же, как в условии конструкции `if`. Кроме того, как и в конструкции `if`, присутствие фигурных скобок обязательно. Выражение в условии вычисляется перед первой итерацией, и, если условие изначально ложно, цикл будет полностью пропущен.

Значение undef

Что произойдет, если попытаться использовать скалярную переменную до того, как ей будет присвоено значение? Ничего серьезного и уж точно ничего опасного. До первого присваивания переменные содержат специальное значение `undef`, которое на языке Perl выражает всего лишь следующее: «Не на что тут смотреть – проходите мимо». Если вы попытаетесь использовать это «ничто» как «числовое нечто», оно интерпретируется как нуль. Если вы попытаетесь использовать его как «строковое нечто», оно интерпретируется как пустая строка. Однако `undef` не является ни строкой, ни числом; это совершенно отдельный вид скалярного значения.

---

[1] При построчном чтении такая ситуация не возникнет, но она возможна: например, если вы выберете в качестве разделителя ввода (`$/`) что-то другое вместо символа новой строки, или воспользуетесь функцией `read`, или построите строку самостоятельно.

[2] Любой программист рано или поздно случайно создает бесконечный цикл. Если ваша программа упорно не желает останавливаться, ее можно прервать так же, как любую другую программу в вашей системе. Часто зациклившуюся программу удается остановить клавишами Control-C; за полной информацией обращайтесь к документации по своей системе.

Так как undef автоматически интерпретируется как нуль при использовании в числовом контексте, мы можем легко реализовать накопление суммы с пустым начальным значением:

```
# Суммирование нечетных чисел
$n = 1;
while ($n < 10) {
  $sum += $n;
  $n += 2; # Переход к следующему нечетному числу
}
print "The total was $sum.\n";
```

Такое решение нормально работает, если переменная $sum содержала undef до начала цикла. При первой итерации переменная $n равна 1; первая строка в теле цикла увеличивает $sum на 1. Это равносильно прибавлению 1 к переменной, содержащей нуль (значение undef здесь используется так, как если бы оно было числом). После суммирования переменная содержит значение 1. После этого переменная уже инициализирована, и дальнейшее суммирование работает традиционно.

Аналогичным образом реализуется накопление строк с пустой исходной строкой:

```
$string .= "more text\n";
```

Если $string содержит undef, переменная работает так, как если бы она содержала пустую строку; соответственно текст "more text\n" просто включается в переменную. Но если она уже содержит строку, новый текст присоединяется к концу строки.

Программисты Perl часто используют новые переменные подобным образом, заставляя приложение интерпретировать undef как нуль или пустую строку в зависимости от ситуации.

Многие операторы возвращают undef, если аргументы имеют недопустимые или бессмысленные значения. Если с переменной не делается ничего особенного, вы получите нуль или пустую строку, и ничего страшного не произойдет. На практике это обычно обходится без серьезных проблем; многие программисты используют такое поведение в своих программах. Но следует знать, что при включенных предупреждениях Perl в большинстве случаев сообщает о необычном использовании undef, так как это может свидетельствовать о возникновении проблем. Например, простое копирование undef из одной переменной в другую пройдет нормально, а при попытке передать его print обычно выводится предупреждение.

Функция defined

Оператор построчного ввода <STDIN> может возвращать undef. Обычно он возвращает строку текста, но когда входные данные закончатся (на-

пример, в конце файла), он сообщит об этом, возвращая undef.[1] Чтобы определить, что полученное значение является undef, а не пустой строкой, воспользуйтесь функцией undefined. Эта функция возвращает *false* для undef и *true* для всех остальных значений:

```
$madonna = <STDIN>;
if ( defined($madonna) ) {
  print "The input was $madonna";
} else {
  print "No input available!\n";
}
```

Если потребуется задать переменной значение undef, используйте оператор с «неожиданным» именем undef:

```
$madonna = undef; # Словно переменная никогда не использовалась
```

Упражнения

Ответы к упражнениям приведены в приложении А.

1. [5] Напишите программу для вычисления длины окружности радиусом 12.5. Длина окружности вычисляется как результат умножения 2π (примерно 2, умноженное на 3.141592654) на радиус. Результат должен быть равен примерно 78,5.
2. [4] Измените программу из предыдущего упражнения так, чтобы она запрашивала и получала значение радиуса от пользователя, запустившего программу. Если пользователь введет 12.5, программа должна выдать такое же значение, как в предыдущем упражнении.
3. [4] Измените программу из предыдущего упражнения так, чтобы при вводе отрицательного числа выдавался нуль (вместо отрицательной длины окружности).
4. [8] Напишите программу, которая запрашивает и получает два числа (в разных строках ввода), а затем выводит их произведение.
5. [8] Напишите программу, которая запрашивает и получает строку и число (в разных строках ввода), а затем выводит строку указанное число раз – в отдельных строках вывода. Скажем, если пользователь введет "fred" и "3", программа выводит три строки с текстом «fred». Если пользователь введет "fred" и "299792", вывод получится довольно длинным.

[1] При вводе с клавиатуры «конец файла» обычно не встречается, но в результате перенаправления входные данные могут поступать из файла. Пользователь также может нажать комбинацию клавиш, которая распознается системой как признак конца файла.

3

Списки и массивы

Если скалярные значения представляют «единственное число» в Perl, как упоминалось в начале главы 2, «множественное число» представляется списками и массивами.

Список содержит упорядоченную коллекцию скалярных значений. *Массив* представляет собой переменную для хранения списка. В Perl эти два термина часто используются как синонимы. Но если выражаться точнее, список – это данные, а массив – переменная. В программе можно создать списочное значение, которое не хранится в массиве, но любая переменная массива содержит список (хотя этот список может быть пустым). На рис. 3.1 изображен список, хранящийся в массиве или нет.

Каждый *элемент* массива или списка представляет собой отдельную скалярную переменную с независимым скалярным значением. Элементы упорядочены, то есть хранятся в определенной последовательности от первого до последнего элемента. Каждому элементу массива

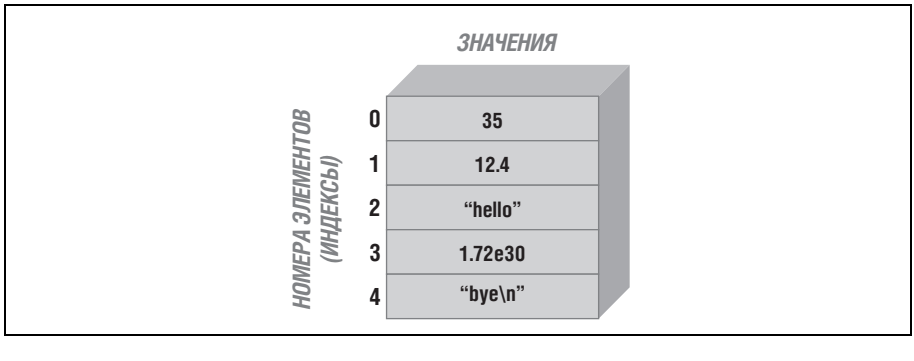


Рис. 3.1. *Список с пятью элементами*

или списка назначается *индекс* – число с единичным приращением, начиная с нуля[1]. Таким образом, индекс первого элемента массива или списка всегда равен нулю.

Так как элементы являются независимыми скалярными значениями, список или массив может содержать числа, строки или произвольные комбинации разных скаляров. Тем не менее на практике элементы обычно относятся к одному типу, скажем названия книг (только строки) или таблицы косинусов (только числа).

Массив или список может иметь произвольное количество элементов. Как минимум он не содержит ни одного элемента, как максимум – заполняет всю доступную память. Это тоже соответствует философии «отсутствия искусственных ограничений», принятой в Perl.

Обращение к элементам массива

Всем, кто уже работал с массивами в других языках, покажется вполне естественным, что в Perl предусмотрена возможность обращения к элементу массива по индексу.

Элементы массивов нумеруются последовательными целыми числами с шагом 1, начиная с нуля:

```
$fred[0] = "yabba";
$fred[1] = "dabba";
$fred[2] = "doo";
```

Имя массива (в данном случае `"fred"`) существует в отдельном пространстве имен, не зависимом от скаляров. В той же программе можно создать скалярную переменную с именем `$fred`; Perl будет различать эти переменные и не перепутает их.[2] (Зато перепутает программист, который будет заниматься сопровождением вашего кода, так что не стоит из прихоти называть все переменные одним именем!)

Элементы массива вида `$fred[2]` могут использоваться везде[3], где может использоваться скалярная переменная типа `$fred`. Например,

---

[1] Индексация массивов и списков в Perl всегда начинается с нуля, в отличие от некоторых других языков. В ранних версиях Perl начальный индекс массивов и списков можно было изменять (не для одного массива или списка, а для всех сразу!). Позднее Ларри осознал, что это нехорошо, и использовать эту возможность сейчас категорически не рекомендуется. Но если вы патологически любознательны, загляните в описание переменной `$[` в man-странице *perlvar*.

[2] Синтаксис всегда однозначен – иногда он бывает запутанным, но неоднозначным быть не может.

[3] Или почти везде. Самое заметное исключение: управляющая переменная цикла `foreach` (см. далее в этой главе) должна быть простым скаляром. Имеются и другие исключения, вроде «косвенных слотов объектов» и «косвенных слотов файловых дескрипторов» в вызовах `print` и `printf`.

можно получить значение элемента массива или изменить его с помощью тех же выражений, которые уже встречались нам в предыдущей главе:

```
print $fred[0];
$fred[2]  = "diddley";
$fred[1] .= "whatsis";
```

Конечно, вместо индекса может использоваться любое выражение с числовым результатом. Если оно не является целым числом, то будет автоматически усечено до ближайшего меньшего целого:

```
$number = 2.71828;
print $fred[$number - 1]; # Равносильно print $fred[1]
```

Если индекс обозначает элемент, выходящий за границы массива, соответствующее значение будет равно `undef`. Все происходит так же, как с обычными скалярными переменными; если в переменной никогда не сохранялось значение, она содержит `undef`:

```
$blank = $fred[ 142_857 ]; # Неиспользованный элемент массива содержит undef
$blanc = $mel;             # Неиспользованный скаляр $mel также содержит undef
```

Специальные индексы массивов

При записи в элемент с индексом, выходящим за границу массива, массив автоматически расширяется до необходимых пределов – его длина не ограничивается, пока в системе остается свободная память. Если Perl потребуется создать промежуточные элементы, они создаются со значениями `undef`:

```
$rocks[0]  = 'bedrock';      # Один элемент...
$rocks[1]  = 'slate';        # другой...
$rocks[2]  = 'lava';         # еще один...
$rocks[3]  = 'crushed rock'; # и еще...
$rocks[99] = 'schist';       # А теперь 95 элементов undef
```

Иногда требуется узнать индекс последнего элемента массива. Для только что показанного массива индекс последнего элемента определяется записью `$#rocks`.[1] Последний индекс не совпадает с количеством элементов, потому что массив начинается с элемента с нулевым индексом:

```
$end = $#rocks;                # 99, индекс последнего элемента
$number_of_rocks = $end + 1;   # Допустимо, но можно и лучше (см. далее)
$rocks[ $#rocks ] = 'hard rock'; # Последний элемент
```

Конструкция `$#имя` достаточно часто используется в качестве индекса, как в последнем примере, поэтому Ларри создал сокращенную запись:

---

[1] За эту уродливую запись следует винить командный процессор C. К счастью, на практике она используется довольно редко.

отрицательные индексы, отсчитываемые от конца массива. Если массив состоит из трех элементов, допустимыми являются отрицательные индексы -1 (последний элемент), -2 (средний элемент) и -3 (первый элемент). Впрочем, в реальных программах никто не использует другие отрицательные индексы, кроме -1.

```
$rocks[ -1 ]   = 'hard rock';   # Проще, чем в предыдущем примере
$dead_rock     = $rocks[-100];  # Возвращает 'bedrock'
$rocks[ -200 ] = 'crystal';     # Фатальная ошибка!
```

Списочные литералы

Массив (способ представления списочных значений в программе) инициализируется списком значений, разделенных запятыми и заключенных в круглые скобки. Эти значения образуют элементы списка. Примеры:

```
(1, 2, 3)      # Список из трех значений 1, 2 и 3
(1, 2, 3,)     # Те же три значения (последняя запятая игнорируется)
("fred", 4.5)  # Два значения, "fred" и 4.5
( )            # Пустой список - ноль элементов
(1..100)       # Список из 100 целых чисел
```

В последней строке используется *диапазонный оператор* `..`, с которым мы здесь встретились впервые. Этот оператор создает список от левого до правого скалярного значения с шагом 1. Примеры:

```
(1..5)            # То же, что (1, 2, 3, 4, 5)
(1.7..5.7)        # То  же самое - оба значения усекаются
(5..1)            # Пустой список - .. работает только "вверх"
(0, 2..6, 10, 12) # То же, что (0, 2, 3, 4, 5, 6, 10, 12)
($m..$n)          # Диапазон определяется текущими значениями $m и $n
(0..$#rocks)      # Индексы массива из предыдущего раздела
```

Как видно из двух последних примеров, элементы списочного литерала не обязаны быть константами – это могут быть выражения, которые вычисляются заново при каждом использовании литерала. Пример:

```
($m, 17)       # Два значения: текущее значение $m и 17
($m+$o, $p+$q) # Два значения
```

Конечно, список может состоять из одних скалярных значений, как этот типичный список строк:

```
("fred", "barney", "betty", "wilma", "dino")
```

Сокращение qw

Списки простых слов (как в последнем примере) достаточно часто используются в программах. Сокращение `qw` позволяет быстро создать такой список без множества лишних кавычек:

```
qw( fred barney betty wilma dino ) # То же, что прежде, но короче
```

Сокращение qw означает «Quoted Words» («квотирование по словам») или «Quoted by Whitespace» («квотирование по пропускам») в зависимости от того, кого вы спросите. Как бы то ни было, Perl интерпретирует данные в скобках по правилам строк в апострофах (соответственно в список qw, в отличие от строк в кавычках, нельзя включить \n или $fred). Пропуски (пробелы, табуляции, символы новой строки) удаляются, а все, что останется, преобразуется в список элементов. Из-за удаления пропусков тот же список можно записать другим (надо признать, довольно странным) способом:

```
qw(fred
  barney     betty
wilma dino )  # То же самое, но со странными пропусками
```

Список является разновидностью определения литералов, поэтому он не может содержать комментарии. В двух предыдущих примерах в качестве ограничителей использовались круглые скобки, но Perl позволяет выбрать любой знак препинания в качестве ограничителя. Несколько распространенных вариантов:

```
qw! fred barney betty wilma dino !
qw/ fred barney betty wilma dino /
qw# fred barney betty wilma dino #   # Как в комментариях!
qw( fred barney betty wilma dino )
qw{ fred barney betty wilma dino }
qw[ fred barney betty wilma dino ]
qw< fred barney betty wilma dino >
```

Как показывают четыре последних примера, возможно использование двух разных ограничителей. Если открывающий ограничитель принадлежит к числу «левых» символов, то закрывающим ограничителем будет соответствующий «правый» символ. В остальных случаях один и тот же символ используется для пометки как начала, так и конца списка.

Если закрывающий ограничитель должен присутствовать среди символов в списке, вероятно, вы неудачно выбрали ограничители. Но если сменить ограничитель почему-либо нежелательно или невозможно, символ все равно можно включить в список с префиксом :

```
qw! yahoo\! google ask msn ! # yahoo! включается как элемент списка
```

Как и в строках, заключенных в апострофы, два последовательных символа \\ обозначают один литеральный символ «обратная косая черта».

Девиз Perl гласит: «Это можно сделать несколькими способами», однако у вас может появиться вопрос: зачем нужны все эти разные ограничители? Вскоре вы увидите, что существуют и другие разновидности оформления строк, в которых они могут пригодиться. Но даже сейчас вы согласитесь с тем, что они будут удобны при построении списка имен файлов UNIX :

```
qw{
  /usr/dict/words
```

```
  /home/rootbeer/.ispell_english
}
```

Если бы косая черта была единственным возможным ограничителем, такой список было бы весьма неудобно читать, записывать и сопровождать.

Списочное присваивание

Вы уже знаете, как присвоить переменной скалярное значение. Аналогичным образом можно присвоить списочное значение списку из нескольких переменных:

```
($fred, $barney, $dino) = ("flintstone", "rubble", undef);
```

Все три переменные в списке слева получают новые значения, как если бы в программе были выполнены три присваивания. Список строится до начала присваивания, что позволяет легко поменять местами значения двух переменных в Perl:[1]

```
 ($fred, $barney) = ($barney, $fred);      # Значения меняются местами
($betty[0], $betty[1]) = ($betty[1], $betty[0]);
```

Но что произойдет, если количество переменных (слева от знака равенства) не совпадет с количеством присваиваемых значений (справа)? При списочном присваивании лишние значения игнорируются – Perl считает, что если бы вы хотели где-то сохранить эти значения, то указали бы, где именно их нужно сохранить. Если же в присваивании участвуют лишние переменные, им присваивается значение `undef`:[2]

```
($fred, $barney) = qw< flintstone rubble slate granite >; # Два значения
                                                          # проигнорированы
($wilma, $dino)  = qw[flintstone];             # $dino присваивается undef
```

Теперь, когда вы умеете присваивать значения спискам, строковый массив можно построить всего в одной строке следующего вида:[3]

```
($rocks[0], $rocks[1], $rocks[2], $rocks[3]) = qw/talc mica feldspar quartz/;
```

Но если речь идет о целом массиве, в Perl предусмотрена более простая запись. Просто поставьте знак `@` перед именем массива (без индекса в квадратных скобках после него), чтобы сослаться на весь массив сразу. Эту запись можно прочитать как «все элементы», то есть `@rocks`

---

[1] В отличие от языков вроде C, где эта задача не имеет простого общего решения. Программисты C вынуждены использовать вспомогательную переменную для временного хранения значения, возможно с применением макроса.

[2] Это справедливо для скалярных переменных. Переменным массивов, как вы вскоре увидите, присваивается пустой список.

[3] Здесь мы слегка смухлевали: предполагается, что массив `rocks` пуст перед выполнением этой команды. Если, скажем, в `$rocks[7]` хранится значение, присваивание его не изменит.

означает «все элементы rocks».[1] Она может использоваться с обеих сторон оператора присваивания:

```
@rocks  = qw/ bedrock slate lava /;
@tiny   = ( );                      # Пустой список
@giant  = 1..1e5;                   # Список из 100 000 элементов
@stuff  = (@giant, undef, @giant);  # Список из 200 001 элемента
$dino   = "granite";
@quarry = (@rocks, "crushed rock", @tiny, $dino);
```

Последнее присваивание заносит в `@quarry` список из пяти элементов `(bedrock, slate, lava, crushed rock, granite)`, так как массив `@tiny` добавляет в список нуль элементов. (В частности, он не добавляет в список значение `undef`, хотя это можно сделать вручную, как в предшествующем примере `@stuff`.) Следует также заметить, что имя массива заменяется содержащимся в нем списком. Массив не становится элементом списка, потому что массивы могут содержать только скаляры, но не другие массивы.[2] Переменная массива, которой еще не было присвоено значение, содержит пустой список `( )`. Подобно тому как новые скалярные переменные изначально содержат `undef`, массивы всегда изначально содержат пустой список.

Стоит заметить, что при копировании массива в другой массив все равно выполняется списочное присваивание. Списки просто хранятся в массивах. Пример:

```
@copy = @quarry; # Копирование списка из одного массива в другой
```

Операторы pop и push

Новые элементы *можно* добавлять в массив, просто сохраняя их в элементах с новыми индексами. Но настоящие программисты Perl не используют индексы.[3] По этой причине в нескольких ближайших разде-

---

[1] Ларри утверждает, что он выбрал знаки `$` и `@`, потому что они напоминают буквы «s» и «a» в словах Scalar (скаляр) и Array (массив). Если вам так не кажется или такое объяснение не поможет их запомнить – неважно.

[2] Однако в «книге с альпакой» представлена особая разновидность скаляров – так называемые *ссылки*. Они позволяют создавать то, что мы неформально называем «списками списков», а также другие интересные и полезные структуры данных. Но и в этом случае список реально содержит не другие списки, а ссылки на массивы.

[3] Конечно, мы шутим. Но в этой шутке есть доля истины. Индексирование массива не использует сильные стороны Perl. При использовании `pop`, `push` и других аналогичных операций, избегающих индексирования, программа обычно работает быстрее, к тому же вы избегаете ошибок «смещения на 1». Иногда начинающие программисты Perl, желающие сравнить Perl с C по скорости, берут алгоритм сортировки, оптимизированный для C (с многочисленными операциями индексирования), переписывают его на Perl «один в один» и удивляются, почему он так медленно работает. Оказывается, скрипка Страдивари – не лучший инструмент для забивания гвоздей.

лах мы представим приемы работы с массивами без использования индексов.

Массивы часто применяются для реализации стеков, в которых элементы добавляются и удаляются с конца списка (там, где у массива находится «последний» элемент с наибольшим индексом). Такие операции встречаются достаточно часто, поэтому для их выполнения были созданы специальные функции.

Оператор pop удаляет из массива последний элемент и возвращает его:

```
@array  = 5..9;
$fred   = pop(@array);  # $fred присваивается 9, в @array остаются
                        # элементы (5, 6, 7, 8)
$barney = pop @array;   # $barney присваивается 8, в @array остаются
                        # элементы (5, 6, 7)
pop @array;              # @array теперь содержит (5, 6). (Элемент 7 удален).
```

В последнем примере pop используется в «пустом контексте» – по сути это затейливый способ сказать, что возвращаемое выражение вообще не применяется. В подобном использовании нет ничего ошибочного, если вам требуется именно такое поведение.

Если массив пуст, pop не изменяет его (так как удаляемого элемента не существует) и возвращает undef.

Вероятно, вы заметили, что pop может использоваться как с круглыми скобками, так и без них. Это общее правило в Perl: круглые скобки всегда можно убрать – кроме тех случаев, когда это изменяет смысл выражения. Обратная операция push добавляет элемент (или список элементов) в конец массива:

```
push(@array, 0);       # @array теперь содержит (5, 6, 0)
push @array, 8;        # @array теперь содержит (5, 6, 0, 8)
push @array, 1..10;    # @array теперь содержит десять новых элементов
@others = qw/ 9 0 2 1 0 /;
push @array, @others; # @array теперь содержит пять новых элементов
                       # (итого 19)
```

Обратите внимание: в первом аргументе push или в единственном аргументе pop должна передаваться переменная массива. Для литеральных списков эти операции не имеют смысла.

Операторы shift и unshift

Операторы push и pop выполняют операции в конце массива (у правого края, среди элементов с большими индексами – в зависимости от того, как вам удобнее думать об этом). Похожие операторы unshift и shift выполняют соответствующие действия в «начале» массива (у левого края, среди элементов с наименьшими индексами). Несколько примеров:

```
@array = qw# dino fred barney #;
$m = shift(@array);      # $m присваивается "dino", @array теперь
```

```
                              # содержит ("fred", "barney")
$n = shift @array;            # $n присваивается "fred", @array теперь
                              # содержит ("barney")
shift @array;                 # @array остается пустым
$o = shift @array;            # $o присваивается undef, @array по-прежнему пуст
unshift(@array, 5);           # @array содержит список из одного элемента (5)
unshift @array, 4;            # @array теперь содержит (4, 5)
@others = 1..3;
unshift @array, @others; # @array теперь содержит (1, 2, 3, 4, 5)
```

По аналогии с `pop`, `shift` возвращает `undef` для пустой переменной массива.

Интерполяция массивов в строках

Массивы, как и скаляры, могут интерполироваться в строках, заключенных в кавычки. Элементы массивов при интерполяции автоматически разделяются пробелами:[1]

```
@rocks = qw{ flintstone slate rubble };
print "quartz @rocks limestone\n";  # печатает пять слов,
                                    # разделенных пробелами
```

Перед интерполируемым массивом и после него дополнительные пробелы не вставляются; если они вам нужны, вставьте их самостоятельно:

```
print "Three rocks are: @rocks.\n";
print "There's nothing in the parens (@empty) here.\n";
```

Если вы забудете, что массивы интерполируются подобным образом, при вставке адреса электронной почты в строку в кавычках вас может ждать сюрприз:

```
$email = "fred@bedrock.edu";  # ОШИБКА! Попытка интерполяции @bedrock
```

Хотя строка содержит обычный адрес электронной почты, Perl обнаруживает массив `@bedrock` и пытается интерполировать его. В некоторых версиях Perl выдается предупреждение:[2]

```
Possible unintended interpolation of @bedrock
```

Проблема решается либо экранированием `@` в строке в кавычках, либо использованием строки в апострофах:

```
$email = "fred\@bedrock.edu"; # Правильно
$email = 'fred@bedrock.edu';  # Другой способ с тем же результатом
```

---

[1] Вообще говоря, разделитель определяется значением специальной переменной `$"`, которая по умолчанию содержит пробел.

[2] В некоторых версиях Perl, предшествующих 5.6, происходила фатальная ошибка, но она была такой назойливой, что в конце концов была заменена предупреждением.

Один элемент массива при интерполяции заменяется своим текущим значением, как и следовало ожидать:

```
@fred = qw(hello dolly);
$y = 2;
$x = "This is $fred[1]'s place";    # "This is dolly's place"
$x = "This is $fred[$y-1]'s place"; # То же самое
```

Индексное выражение вычисляется по правилам обычных выражений, как если бы оно находилось вне строки. Предварительная интерполяция переменных не выполняется. Иначе говоря, если `$y` содержит строку `"2*4"`, то интерполироваться будет элемент 1, а не элемент 7, потому что `"2*4"` в числовом контексте (значение `$y`, используемое в числовом выражении) преобразуется в 2.[1] Если за простой скалярной переменной должна выводиться левая квадратная скобка, отделите ее, чтобы она не считалась частью ссылки на массив:

```
@fred = qw(eating rocks is wrong);
$fred = "right";                # Мы пытаемся вывести "this is right[3]"
print "this is $fred[3]\n";     # Выводит "wrong" из-за $fred[3]
print "this is ${fred}[3]\n";   # Выводит "right" (защита в виде
                                # фигурных скобок)
print "this is $fred"."[3]\n"; # Снова "right" (другая строка)
print "this is $fred\[3]\n";    # Снова "right" (экранирование обратной
                                # косой чертой)
```

Управляющая конструкция foreach

Последовательная обработка всех элементов массива или списка – весьма стандартная задача; специально для ее решения в Perl определена управляющая конструкция `foreach`. Цикл `foreach` перебирает список значений и выполняет одну итерацию (проход тела цикла) для каждого элемента списка:

```
foreach $rock (qw/ bedrock slate lava /) {
  print "One rock is $rock.\n";  # Выводит три слова
}
```

Управляющая переменная (`$rock` в данном примере) при каждой итерации получает очередное значение из списка. При первой итерации она равна `"bedrock"`, а при третьей – `"lava"`.

Управляющая переменная не является копией элемента списка – это сам элемент списка. Иначе говоря, при модификации управляющей переменной внутри цикла изменяется текущий элемент, как показывает следующий фрагмент. Это довольно удобно, но тем, кто об этом не знает, данная возможность может преподнести сюрприз:

---

[1] Конечно, при включенных предупреждениях Perl напомнит вам о том, что число `"2*4"` выглядит довольно странно.

```
@rocks = qw/ bedrock slate lava /;
foreach $rock (@rocks) {
  $rock = "\t$rock";        # Поставить символ табуляции перед каждым
                            # элементом @rocks
  $rock .= "\n";            # После каждого элемента ставится символ
                            # новой строки
}
print "The rocks are:\n", @rocks; # Элементы выводятся с отступами
                                 # и в разных строках
```

Какое значение будет иметь управляющая переменная после завершения цикла? То же, которое она имела до его начала. Perl автоматически сохраняет и восстанавливает значение управляющей переменной цикла `foreach`. Во время выполнения цикла невозможно обратиться к этому сохраненному значению или изменить его. Таким образом, после завершения цикла переменная обладает тем же значением, что и до начала цикла, или `undef`, если ей не было присвоено значение. Получается, что вы можете назвать управляющую переменную цикла `$rock`, не беспокоясь о том, что это имя уже могло быть использовано для другой переменной.

Главная служебная переменная: $_

Если управляющая переменная не указана в начале цикла `foreach`, Perl по умолчанию использует служебную переменную `$_`. Она (в целом) ведет себя как любая другая скалярная переменная, несмотря на необычное имя. Пример:

```
foreach (1..10) {  # По умолчанию используется $_
  print "I can count to $_!\n";
}
```

Хотя это далеко не единственная переменная по умолчанию в Perl, она используется чаще других. Позднее вы увидите много других ситуаций, в которых автоматически применяется `$_`, если в программе не указана другая переменная или значение; так программист избавляется от необходимости придумывать и вводить новое имя переменной. А чтобы не томить вас в ожидании, подскажем один из этих случаев: функция `print` использует `$_` при отсутствии других аргументов:

```
$_ = "Yabba dabba doo\n";
print;  # По умолчанию выводит $_
```

Оператор reverse

Оператор `reverse` получает список значений (например, из массива) и возвращает его переставленным в обратном порядке. Например, если вас огорчает, что диапазонный оператор `..` работает только «снизу вверх», проблема легко решается:

```
@fred   = 6..10;
@barney = reverse(@fred);  # Содержит 10, 9, 8, 7, 6
@wilma =  reverse 6..10;   # То же без использования другого массива
@fred   = reverse @fred;   # Результат снова возвращается
                           # к исходному состоянию
```

Обратите особое внимание на последнюю строку из-за двукратного использования `@fred`. Perl всегда вычисляет присваиваемое значение (справа) перед выполнением самого присваивания.

Помните, что `reverse` возвращает обращенный список; аргументы при этом не изменяются. Если возвращаемое значение ничему не присваивается, оно бесполезно:

```
reverse @fred;         # ОШИБКА - @fred не изменяется
@fred = reverse @fred; # Так лучше
```

Оператор sort

Оператор `sort` получает список значений (например, из массива) и сортирует их во внутреннем порядке символов. Для ASCII-строк это будет «ASCII-алфавитный» порядок. Конечно, ASCII-сортировка, где все буквы верхнего регистра предшествуют всем буквам нижнего регистра, а цифры предшествуют буквам, не лучший вариант. Однако ASCII-сортировка всего лишь используется *по умолчанию* (в главе 14 будет показано, как установить любой другой порядок сортировки по вашему усмотрению):

```
@rocks   = qw/ bedrock slate rubble granite /;
@sorted  = sort(@rocks);        # Получаем  bedrock, granite, rubble, slate
@back    = reverse sort @rocks; # Последовательность от slate до bedrock
@rocks   = sort @rocks;         # Отсортированный результат снова
                                # сохраняется в @rocks
@numbers = sort 97..102;        # Получаем 100, 101, 102, 97, 98, 99
```

Как видно из последнего примера, сортировка чисел так, как если бы они были строками, может не привести к желаемому результату. По стандартным правилам сортировки любая строка, начинающаяся с 1, предшествует любой строке, начинающейся с 9. Как и в случае с `reverse`, аргументы при вызове не изменяются. Чтобы отсортировать массив, сохраните результат в исходном массиве:

```
sort @rocks;          # ОШИБКА, @rocks не изменяется
@rocks = sort @rocks; # Теперь коллекция упорядочена
```

Скалярный и списочный контекст

Мы подошли к самому важному разделу этой главы. Более того, это самый важный раздел во всей книге. Можно без преувеличения сказать, что от того, насколько хорошо вы поймете этот раздел, будет зависеть

вся ваша карьера программиста Perl. Если до этого момента вы бегло просматривали текст, здесь пора переходить к внимательному чтению.

Но это вовсе не означает, что материал этого раздела сложен для понимания. Основная идея проста: смысл любого выражения зависит от того, где оно используется. В этом нет ничего принципиально нового; в естественных языках контекст используется сплошь и рядом. Допустим, кто-то спросил вас, что означает слово «узел». Однако это слово может иметь разный смысл в зависимости от того, где оно используется. Невозможно определить смысл, не зная *контекста*.

В ходе разбора выражений Perl всегда ожидает встретить либо скалярное, либо списочное значение.[1] Разновидность значения, которую ожидает встретить Perl, называется контекстом выражения.[2]

```
42 + нечто # нечто должно быть скаляром
sort нечто # нечто должно быть списком
```

Одна последовательность символов может интерпретироваться как скалярное значение в одном контексте и как списочное – в другом.[3] Выражения Perl всегда возвращают значение, соответствующее их контексту. Для примера возьмем «имя»[4] массива. В списочном контексте оно рассматривается как список элементов, а в скалярном – как количество элементов в массиве:

```
@people = qw( fred barney betty );
@sorted = sort @people; # Списочный контекст: barney, betty, fred
$number = 42 + @people;  # Скалярный контекст: 42 + 3 = 45
```

Даже обычное присваивание (скалярной переменной или списку) работает в разных контекстах:

```
@list = @people; # Список из трех элементов
$n = @people;    # Число 3
```

Но, пожалуйста, не торопитесь с выводом, что скалярный контекст всегда дает количество элементов, которые были бы возвращены в списоч-

---

1 Если, конечно, Perl не ожидает чего-то совершенно иного. Существуют и другие контексты, которые здесь не рассматриваются. Никто не знает, сколько контекстов используется в Perl; самые умные головы еще не пришли к единому мнению на этот счет.

2 В естественных языках происходит примерно то же. Допустив грамматическую ошибку, вы заметите ее немедленно, потому что ожидаете встретить определенные слова в определенных местах. Со временем вы начнете читать код Perl на таком же подсознательном уровне, но сначала придется обдумывать его.

3 Конечно, список может состоять из одного элемента. Он также может быть пустым или содержать произвольное количество элементов.

4 У массива `@people` «настоящим» именем считается `people`, а знак `@` – всего лишь квалификатор.

ном контексте. Большинство выражений, создающих списки[1], возвращают *намного* более интересную информацию.

Использование выражений, создающих списки, в скалярном контексте

Многие выражения обычно используются для создания списков. Что вы получите, используя такое выражение в скалярном контексте? Посмотрите, что скажет автор этой операции. Обычно автором является Ларри, а все сведения приводятся в документации. В сущности, изучение того, как Ларри думает[2], является важной частью изучения Perl. Научившись думать так, как думает Ларри, вы будете знать, что сделает Perl в той или иной ситуации. А пока будете учиться, вам, вероятно, придется заглядывать в документацию.

Некоторые выражения вообще не имеют значения в скалярном контексте. Например, что должен возвращать вызов `sort` в скалярном контексте? Для подсчета количества элементов сортировка списка не нужна, поэтому до тех пор пока кто-нибудь не создаст новую реализацию, `sort` в скалярном контексте всегда возвращает `undef`.

Или другой пример: оператор `reverse`. В списочном контексте он возвращает обращенный список. В скалярном контексте возвращается обращенная строка (или обращенный результат конкатенации всех строк списка, если он задан):

```
@backwards = reverse qw/ yabba dabba doo /;
   # Получаем doo, dabba, yabba
$backwards = reverse qw/ yabba dabba doo /;
   # Получаем oodabbadabbay
```

На первый взгляд не всегда можно сразу сказать, используется ли выражение в скалярном или списочном контексте. Но поверьте, со временем это станет вашей второй натурой.

Несколько типичных примеров контекста:

```
$fred = нечто;              # Скалярный контекст
@pebbles = нечто;           # Списочный контекст
($wilma, $betty) = нечто;   # Списочный контекст
($dino) = нечто;            # И в этом случае списочный контекст!
```

---

[1] В контексте этого раздела «выражение, создающее скаляр» не отличается от «выражения, создающего список»; любое выражение может создать список или скаляр в зависимости от контекста. Таким образом, говоря о «выражениях, создающих списки», мы имеем в виду выражения, обычно используемые в списочном контексте; появление таких выражений (например, `reverse` или `@fred`) в скалярном контексте может вас удивить.

[2] И это справедливо – ведь при создании Perl он старался думать как вы, чтобы предсказать, что вам понадобится!

Не ошибитесь со списком из одного элемента: он все равно имеет списочный, а не скалярный контекст. Круглые скобки здесь принципиальны; с ними четвертая строка отличается от первой. Если значение присваивается списку (независимо от количества элементов), всегда используется списочный контекст. Присваивание массиву также выполняется в списочном контексте.

Несколько уже знакомых выражений с указанием контекста. Для начала несколько выражений, в которых *нечто* имеет скалярный контекст:

```
$fred = нечто;
$fred[3] = нечто;
123 + нечто
нечто + 654
if (нечто) { ... }
while (нечто) { ... }
$fred[нечто] = нечто;
```

Несколько выражений со списочным контекстом:

```
@fred = нечто;
($fred, $barney) = нечто;
($fred) = нечто;
push @fred, нечто;
foreach $fred (нечто) { ... }
sort нечто
reverse нечто
print нечто
```

Использование выражений, создающих скаляры, в списочном контексте

В этом направлении все тривиально: если выражение обычно не имеет списочного значения, скалярное значение автоматически преобразуется в список из одного элемента:

```
@fred = 6 * 7; # Список из одного элемента (42)
@barney = "hello" . ' ' . "world";
```

Правда, в таком преобразовании кроется одна возможная ловушка:

```
@wilma = undef; # ОЙ! Получаем список из одного элемента (undef)
  # А это не эквивалентно:
@betty = ( );    # Правильный способ очистки массива
```

Так как `undef` является скалярным значением, присваивание `undef` массиву, естественно, не приводит к его очистке. Проблема лучше всего решается присваиванием пустого списка.[1]

[1] В реальных алгоритмах переменные, объявленные в правильной области видимости, обычно не нуждаются в явной очистке, поэтому такое присваивание редко встречается в правильно написанных программах Perl. Области видимости рассматриваются в следующей главе.

Принудительное использование скалярного контекста

Иногда бывает необходимо принудительно использовать скалярный контекст там, где Perl ожидает получить список. В таких ситуациях используется псевдофункция `scalar`. Она не является полноценной функцией и всего лишь сообщает Perl о необходимости использования скалярного контекста:

```
@rocks = qw( talc quartz jade obsidian );
print "How many rocks do you have?\n";
print "I have ", @rocks, " rocks!\n";        # НЕВЕРНО, выводятся строки
print "I have ", scalar @rocks, " rocks!\n"; # Верно, выводится число
```

Как ни странно, парной функции для принудительного использования списочного контекста не существует. Впрочем, она вам и не понадобится. Пока поверьте на слово.

<STDIN> в списочном контексте

Уже встречавшийся нам оператор `<STDIN>` изменяет свое поведение в списочном контексте. Как говорилось ранее, в скалярном контексте `<STDIN>` возвращает первую строку входных данных. В списочном контексте он возвращает *все* оставшиеся строки до конца файла. Каждая строка возвращается в отдельном элементе списка. Пример:

```
@lines = <STDIN>; # Чтение стандартного ввода в списочном контексте
```

Если данные поступают из файла, `<STDIN>` читает весь файл до конца. Но что происходит при вводе с клавиатуры? В UNIX и ряде других систем, включая Linux и Mac OS X, пользователь обычно нажимает клавиши Control-D[1]; этот символ не воспринимается Perl[2], хотя он и может отображаться на экране. В системах DOS/Windows используется комбинация Control-Z.[3] Если в вашей системе используются другие сочетания клавиш, обратитесь к документации или узнайте у местного специалиста.

Если пользователь, запустивший программу, ввел три строки, а затем нажал клавиши, обозначающие конец файла, полученный массив бу-

---

[1] Эта комбинация всего лишь используется по умолчанию; ее можно сменить командой `stty`. Однако на такое поведение можно положиться – мы еще ни разу не видели системы UNIX, в которой бы конец файла при вводе с клавиатуры обозначался другим сочетанием клавиш.

[2] ОС «видит» клавишу Control и сообщает приложению о «конце файла».

[3] В некоторых реализациях Perl для DOS/Windows присутствует ошибка, из-за которой первая строка вывода на терминал после нажатия Control-Z скрывается. В таких системах проблема решается простым выводом пустой строки (`"\n"`) после получения данных.

дет состоять из трех элементов. Каждый элемент будет содержать строку, завершающуюся символом новой строки.

А представьте, как удобно было бы устранить все эти символы новой строки одним вызовом chomp! Оказывается, если передать chomp массив со списком строк, функция удалит завершающий символ новой строки из каждого элемента списка. Пример:

```
@lines = <STDIN>; # Прочитать все строки
chomp(@lines);    # Удалить все завершающие символы новой строки
```

Но чаще эта задача решается конструкцией, похожей на приводившуюся ранее:

```
chomp(@lines = <STDIN>); # Прочитать только текст без символов новой строки
```

Находясь в своей уединенной рабочей секции, вы можете использовать любую из этих конструкций, но большинство программистов предпочитает вторую, более компактную запись.

Очевидно (хотя, как показывает опыт, не всем), что прочитанные строки входных данных невозможно прочитать заново.[1] После достижения конца файла дальнейших данных для чтения не остается.

Что произойдет, если данные поступают из 400-мегабайтного журнала? Оператор построчного ввода прочитает все строки, выделив для их хранения громадный блок памяти.[2] Perl обычно старается не ограничивать программиста, но другие пользователи системы (не говоря уже о системном администраторе) будут возражать. При большом объеме входных данных чаще всего приходится искать способ обработки данных без их одновременной загрузки в память.

Упражнения

Ответы к упражнениям приведены в приложении А.

1. [6] Напишите программу, читающую список строковых значений, каждое из которых находится в отдельной строке входных данных. Элементы списка выводятся в обратном порядке. Если данные вводятся с клавиатуры, вероятно, пользователь должен сообщить о конце ввода, нажав клавиши Control-D в UNIX или Control-Z в Windows.

---

[1] Вообще-то, если ввод поступает из источника с поддержкой seek, вы можете вернуться и прочитать данные снова. Но сейчас речь не об этом.

[2] Как правило, размер блока значительно превышает размер файла. 400-мегабайтный файл, прочитанный в массив, обычно занимает, по меньшей мере, гигабайт памяти. Дело в том, что Perl чаще всего предпочитает расходовать память для экономии времени, и это логично: если вам не хватает памяти, ее можно купить; если вам не хватает времени, сделать ничего нельзя.

2. [12] Напишите программу, которая читает список чисел (в разных строках) до конца входных данных, а затем выводит для каждого числа соответствующее имя из приведенного ниже списка. (Список имен жестко задается в исходном коде программы.) Например, если пользователь ввел `1`, `2`, `4` и `2`, программа должна вывести имена `fred`, `betty`, `dino` и `betty`:

```
fred betty barney dino wilma pebbles bamm-bamm
```

3. [8] Напишите программу, которая читает список строковых значений (из разных строк) до конца входных данных. Затем программа выводит прочитанные строки в «ASCII-алфавитном» порядке, то есть если ввести строки `fred`, `barney`, `wilma` и `betty`, они должны выводиться в порядке `barney betty fred wilma`. Как программа выводит данные – в одной строке или в разных строках? Как улучшить внешний вид выходных данных в обоих случаях?

4

Пользовательские функции

Мы уже видели и использовали некоторые встроенные системные функции — `chomp`, `reverse`, `print` и т. д. Но в Perl, как и в других языках, предусмотрена возможность создания пользовательских функций, также называемых подпрограммами (subroutines).[1] Пользовательские функции позволяют многократно применять написанный фрагмент кода в программе. Имя пользовательской функции представляет собой идентификатор Perl (последовательность букв, цифр и символов подчеркивания, которая не может начинаться с цифры) с необязательным в некоторых ситуациях префиксом `&`. Специальное правило определяет, когда знак `&` можно опустить, а когда этого делать нельзя; оно будет приведено в конце этой главы. А пока мы будем просто использовать `&` всюду, где это не запрещено (самый безопасный вариант). В тех ситуациях, где запрещено, мы, конечно, скажем об этом.

Имена пользовательских функций существуют в отдельном пространстве имен. Даже если в одной программе будет существовать пользовательская функция с именем `&fred` и скалярная переменная `$fred`, Perl их не перепутает, хотя обычно так поступать не рекомендуется.

Определение пользовательской функции

Определение пользовательской функции состоит из ключевого слова `sub`, имени функции (без знака `&`) и заключенного в фигурные скобки[2]

---

[1] В этой главе речь пойдет именно о функциях, определяемых пользователем.

[2] Ладно, формалисты, специально для вас признаем: фигурные скобки являются частью блока. И Perl не требует выделять блок отступом, но это пригодится программисту, который будет заниматься сопровождением вашего кода. Так что, пожалуйста, соблюдайте правила стиля.

блока программного кода, снабженного отступом. Блок содержит *тело* функции:

```
sub marine {
  $n += 1;  # Глобальная переменная $n
  print "Hello, sailor number $n!\n";
}
```

Определения пользовательских функций могут находиться где угодно в тексте программы, но программисты с опытом работы на C и Pascal обычно размещают их в начале файла. Другие предпочитают размещать их в конце, чтобы основная часть кода находилась в начале файла. Выбирайте сами. В любом случае опережающие объявления обычно не нужны.[1] Определения пользовательских функций глобальны; без особых ухищрений создать приватную пользовательскую функцию невозможно.[2] Если в программе присутствуют два определения функций с одинаковыми именами, первое определение заменяется более поздним.[3] Обычно это считается проявлением плохого стиля или признаком того, что программист по сопровождению кода окончательно запутался.

Как вы могли заметить из предыдущего примера, в теле функции могут использоваться любые глобальные переменные. Все переменные, встречавшиеся нам ранее, были глобальными; иначе говоря, они были доступны из любой части программы. Блюстителей чистоты языка глобальная доступность приводит в ужас, но группа разработчиков Perl вооружилась факелами и вилами и изгнала их из города много лет назад. О том, как создать приватную переменную, рассказано в разделе «Приватные переменные в пользовательских функциях» далее в этой главе.

Вызов пользовательской функции

Пользовательская функция может быть *вызвана* из любого выражения по имени (со знаком `&`):[4]

---

1 Если только функция не объявляет «прототип», который указывает, как компилятор должен разбирать и интерпретировать аргументы вызова. Но эта возможность используется редко; за дополнительной информацией обращайтесь к man-странице *perlsub*.

2 А если вам захочется особых ухищрений, обратитесь к описанию ссылок на программный код, хранящихся в приватных (лексических) переменных, в документации Perl.

3 По крайней мере, вы получите предупреждение.

4 Часто при вызове указывается пара круглых скобок, даже если они пусты. В этом варианте записи пользовательская функция наследует значение `@_` вызывающей стороны, которое мы обсудим чуть позже. Продолжайте читать, а то вызовы ваших функций будут приводить к непредвиденным побочным эффектам!

```
&marine;  # Hello, sailor number 1!
&marine;  # Hello, sailor number 2!
&marine;  # Hello, sailor number 3!
&marine;  # Hello, sailor number 4!
```

Возвращаемые значения

Вызов функции всегда является частью выражения, даже если результат вызова не используется напрямую. Вызвав `&marine` в предыдущем примере, мы вычисляли значение выражения, содержащего вызов, но игнорировали его результат.

Чаще при вызове функции ее непосредственный результат – иначе говоря, *возвращаемое значение* – используется в программе. Все пользовательские функции Perl имеют возвращаемое значение; не существует различий между функциями, которые возвращают значение, и теми, которые этого не делают. Впрочем, не все пользовательские функции Perl имеют *полезное* возвращаемое значение.

Так как любая пользовательская функция Perl может быть вызвана с применением возвращаемого значения, было бы немного расточительно определять специальный синтаксис для «возвращения» конкретного значения. Ларри поступил просто: в ходе выполнения пользовательской функции Perl совершает серию действий с вычислением различных значений. То значение, которое было вычислено последним, *автоматически* становится возвращаемым значением.

Допустим, в программе определяется следующая пользовательская функция:

```
sub sum_of_fred_and_barney {
  print "Hey, you called the sum_of_fred_and_barney subroutine!\n";
  $fred + $barney;  # Возвращаемое значение
}
```

Последним выражением, вычисляемым в теле функции, является сумма `$fred` и `$barney`; соответственно сумма `$fred` и `$barney` становится возвращаемым значением функции. Вот как это выглядит в действии:

```
$fred = 3;
$barney = 4;
$wilma = &sum_of_fred_and_barney;      # $wilma присваивается 7
print "\$wilma is $wilma.\n";
$betty = 3 * &sum_of_fred_and_barney;  # $betty присваивается 21
print "\$betty is $betty.\n";
```

Результат выполнения программы выглядит так:

```
Hey, you called the sum_of_fred_and_barney subroutine!
$wilma is 7.
Hey, you called the sum_of_fred_and_barney subroutine!
$betty is 21.
```

Команда print используется только для отладки, чтобы вы убедились в успешном вызове функции. Когда программа будет готова, отладочную печать можно будет удалить. Но допустим, в конец тела функции была добавлена еще одна строка отладочной печати:

```
sub sum_of_fred_and_barney {
  print "Hey, you called the sum_of_fred_and_barney subroutine!\n";
  $fred + $barney;   # А теперь это не возвращаемое значение!
  print "Hey, I'm returning a value now!\n";       # Ой!
}
```

В этом примере последним вычисляемым выражением становится не команда сложения, а команда print. Обычно возвращаемое значение print равно 1 (признак успешного вывода), но нам нужно совсем другое возвращаемое значение. Итак, будьте осторожны при включении дополнительного кода в функцию, потому что ее возвращаемое значение определяется значением последнего *вычисленного* выражения.

Что же произошло во второй (ошибочной) функции с суммой $fred и $barney? Она не была присвоена никакой переменной, поэтому Perl проигнорировал ее. Если включить режим предупреждений, Perl заметит, что сложение двух переменных без присваивания не привело ни к какому полезному результату, и, вероятно, предупредит о «бесполезном сложении в пустом контексте». Термин «пустой контекст» всего лишь означает, что полученный ответ не сохраняется в переменной и не используется иным способом.

Под «последним вычисленным выражением» следует понимать именно *вычисленное* выражение, а не последнюю строку текста. Например, следующая функция возвращает большее из значений $fred и $barney:

```
sub larger_of_fred_or_barney {
  if ($fred > $barney) {
    $fred;
  } else {
    $barney;
  }
}
```

Последним вычисляется либо выражение $fred, либо $barney; значение одной из этих переменных становится возвращаемым значением. Невозможно заранее сказать, будет ли возвращено значение $fred или $barney, пока функция не сравнит значения этих переменных во время выполнения.

Все рассмотренные примеры весьма тривиальны. Ситуация становится интереснее, когда при разных вызовах функции могут передаваться разные входные значения (вместо использования глобальных переменных). Собственно, мы уже вплотную подошли к этой теме.

Аргументы

Функция larger_of_fred_or_barney станет намного удобнее, если пользователь не будет ограничиваться применением глобальных переменных fred и $barney. Например, чтобы определить большее из значений $wilma и $betty функцией larger_of_fred_or_barney, вам придется сначала скопировать их в $fred и $barney. Если в этих переменных хранятся какие-нибудь полезные данные, их придется временно сохранить в других переменных (скажем, $save_fred и $save_barney), а после вызова функции скопировать обратно в $fred и $barney.

К счастью, в Perl пользовательские функции могут вызываться с аргументами. Чтобы передать список аргументов функции, просто разместите списочное выражение в круглых скобках после имени функции:

```
$n = &max(10, 15);  # Функции передаются два аргумента
```

Список *передается* функции, то есть функция может использовать его так, как считает нужным. Конечно, этот список должен где-то храниться, поэтому Perl автоматически сохраняет список параметров (другое название списка аргументов) в специальной переменной массива с именем @_ на время всего вызова функции. Функция может обратиться к этой переменной, чтобы узнать как количество аргументов, так и их значения.

Таким образом, первый параметр функции хранится в элементе $_[0], второй – в $_[1] и т. д. Но (очень важный момент!) эти переменные не имеют никакого отношения к переменной $_– не более чем $dino[3] (элемент массива @dino) к переменной $dino (совершенно самостоятельная скалярная переменная). Просто список параметров должен храниться в каком-то массиве, чтобы с ним можно было работать в пользовательской функции, и Perl применяет для этой цели массив @_.

Теперь мы *можем* записать функцию &max, похожую на &larger_of_fred_or_barney. Но вместо $fred она использует первый параметр функции ($_[0]), а вместо $barney – второй параметр ($_[1]). И в результате *может* получиться следующий код:

```
sub max {
  # Сравните с &larger_of_fred_or_barney
  if ($_[0] > $_[1]) {
    $_[0];
  } else {
    $_[1];
  }
}
```

Как мы уже сказали, так действовать *можно*. Но из-за многочисленных индексов код выглядит уродливо, его трудно читать, писать, проверять и отлаживать. Вскоре вы увидите более элегантное решение.

К тому же у этой функции есть и другая проблема: ее компактное и симпатичное имя &max не напоминает нам о том, что функция правильно работает только для двух аргументов:

```
$n = &max(10, 15, 27);  # Лишний аргумент !
```

Лишние аргументы игнорируются; функция просто не обращается к $_[2]. Perl не интересует, есть какое-нибудь значение в списке аргументов или нет. Недостающие параметры тоже игнорируются – при попытке обратиться к элементу за границей массива @_ вы просто получите undef, как и для любого другого массива. Позднее в этой главе вы увидите, как написать более универсальную версию &max, работающую с произвольным количеством параметров.

Переменная @_ является приватной для этой пользовательской функции;[1] если @_ содержит глобальное значение, оно сохраняется перед вызовом функции, а затем восстанавливается после возврата управления.[2] Из этого также следует, что пользовательская функция может передать аргументы другой функции, не рискуя потерять свой массив @_ – вложенный вызов получит собственную копию @_. Даже если функция вызывает сама себя в результате рекурсии, при каждом вызове будет создаваться новая копия @_, поэтому @_ всегда содержит список параметров для *текущего* вызова.

Приватные переменные в пользовательских функциях

Но если Perl создает новую копию @_ для каждого вызова функции, не может ли он также создать переменные, которые будут использоваться только внутри этой функции? Конечно, может.

По умолчанию все переменные в Perl являются глобальными; иначе говоря, они доступны из любой точки программы. Но вы также можете в любой момент создать приватные переменные, называемые *лексическими переменными*, при помощи оператора my:

```
sub max {
  my($m, $n);        # Новые, приватные переменные для этого блока
  ($m, $n) = @_;     # Присваивание имен параметрам
  if ($m > $n) { $m } else { $n }
}
```

---

1 Если при вызове перед именем функции стоит знак &, а после имени нет круглых скобок (и аргументов), массив @_ наследуется из контекста вызова. Обычно такое наследование нежелательно, но в некоторых ситуациях оно может быть полезно.

2 Вспомните, что аналогичный механизм применяется к управляющей переменной цикла foreach (см. предыдущую главу). В обоих случаях Perl автоматически сохраняет и восстанавливает значение переменной.

Область видимости этих переменных ограничивается блоком, в котором они находятся; все внешние переменные с именами $m или $n совершенно не зависят от них. Независимость работает в обе стороны: внешний код не может обращаться к приватным переменным и изменять их, случайно или намеренно.[1] Таким образом, вы можете вставить эту функцию в любую программу Perl и применять ее, зная, что она не повредит переменные $m и $n этой программы (если они используются).[2] Стоит также заметить, что внутри блоков if точка с запятой после выражения, определяющего возвращаемое значение, не обязательна. Хотя Perl позволяет опустить последний символ ; в блоке, на практике это стоит делать только в очень простом коде, когда весь блок записывается в одну строку.

Код предыдущего примера можно сделать еще проще. Вы заметили, что список ($m, $n) встречается дважды? Оператор my также можно применить к списку переменных в круглых скобках, поэтому в пользовательских функциях эти две начальные команды часто объединяются:

```
my($m, $n) = @_;  # Присваивание имен параметрам
```

Эта команда создает приватные переменные и одновременно инициализирует их, так что первому параметру теперь присвоено более удобное имя $m, а второму – $n. Почти каждая пользовательская функция начинается с подобной строки. Она четко показывает, что функция должна получать два скалярных параметра, которым внутри функции присваиваются имена $m и $n.

Списки параметров переменной длины

В реальном коде Perl пользовательские функции часто получают списки параметров произвольной длины. Это обусловлено уже упоминавшейся философией «отсутствия искусственных ограничений», принятой в Perl. Конечно, в этом Perl отличается от многих традиционных языков программирования, требующих жесткой типизации каждой функции (т. е. возможности ее вызова только с заранее определенным числом параметров заранее определенных типов). Хорошо, конечно, что Perl настолько гибок, но, как было показано ранее в примере с функцией &max, вызов функции с другим количеством аргументов может создать проблемы.

Конечно, функция может легко проверить количество аргументов по массиву @_. Например, в функции &max проверка может выглядеть так:

---

[1] Опытный программист сообразит, что к лексической переменной можно обратиться за пределами области видимости по ссылке, но никогда – по имени.

[2] Конечно, если в этой программе уже есть пользовательская функция с именем &max, *ее* вы все же повредите.

```
sub max {
  if (@_ != 2) {
    print "WARNING! &max should get exactly two arguments!\n";
  }
  # Нормальное продолжение...
  .
  .
  .
}
```

Условие `if` использует «имя» массива в скалярном контексте для получения количества его элементов (см. главу 3).

Но в реальном программировании такие проверки встречаются относительно редко; лучше заставить функцию приспособиться к переданным параметрам.

Улучшенная версия &max

Давайте перепишем функцию `&max` так, чтобы она могла вызываться с произвольным количеством аргументов:

```
$maximum = &max(3, 5, 10, 4, 6);

sub max {
  my($max_so_far) = shift @_;  # Пока сохраняем как максимальный
                               # первый элемент
  foreach (@_) {               # Просмотреть все остальные аргументы
    if ($_ > $max_so_far) {    # Текущий аргумент больше сохраненного?
      $max_so_far = $_;
    }
  }
  $max_so_far;
}
```

В этом коде используется так называемый «алгоритм водяного горизонта»; после наводнения, когда волны накатят и отступят в последний раз, отметка наибольшего подъема воды покажет, до какого уровня дошло затопление. В этой функции переменная `@max_fo_far` хранит нашу «отметку наибольшего подъема воды» — максимальное число, обнаруженное при просмотре массива.

В первой строке `$max_so_far` присваивается значение 3 (первый параметр в примере); присваивание осуществляется вызовом `shift` для массива параметров `@_`. Теперь `@_` состоит из элементов `(5, 10, 4, 6)`, так как элемент 3 был «сдвинут» из массива. А наибольшим числом, обнаруженным на данный момент, остается *единственное* просмотренное число 3 (первый параметр).

Далее цикл `foreach` последовательно перебирает все остальные значения в списке параметров `@_`. По умолчанию в цикле используется управляющая переменная `$_` (не забудьте, что между `@_` и `$_` нет никакой автоматической связи; просто этим переменным случайно были

присвоены похожие имена). При первой итерации переменная $_ равна 5. Условие if видит, что это значение больше @max_so_far, поэтому @max_so_far присваивается 5 – новая «отметка наибольшего подъема».

При следующей итерации переменная $_ равна 10. Обнаружен новый максимум, который также сохраняется в $max_so_far.

Следующее значение $_ равно 4. Условие не выполняется, так как параметр меньше текущего значения $max_so_far (10); тело блока if пропускается.

При следующей итерации переменная $_ равна 6, тело if снова пропускается. Поскольку это была последняя итерация, цикл завершен.

В итоге функция возвращает значение $max_so_far. Это наибольшее значение среди просмотренных элементов, а раз мы просмотрели их все, это самое большое число в списке: 10.

Пустые списки параметров

Улучшенный алгоритм &max теперь работает нормально даже в том случае, если функции передается более двух параметров. Но что произойдет, если функция вызвана вообще без параметров?

На первый взгляд проблема кажется надуманной. Кому придет в голову вызывать &max без параметров? Но, возможно, кто-то написал строку следующего вида:

```
$maximum = &max(@numbers);
```

Может оказаться, что массив @numbers содержит пустой список; допустим, его содержимое было прочитано из пустого файла. А значит, необходимо решить, как функция &max должна работать в этой ситуации?

Первая строка функции инициализирует $max_so_far вызовом shift для @_, (пустого) списка параметров. Это безвредно; массив пуст, shift вернет undef, и это значение будет присвоено $max_so_far.

Теперь цикл foreach должен перебрать @_, но массив пуст, поэтому тело цикла будет выполнено 0 раз.

В итоге Perl использует значение $max_so_far – undef – как возвращаемое значение функции. В каком-то смысле это правильно, потому что в пустом списке наибольшего элемента не существует.

Конечно, тот, кто вызывает вашу функцию, должен знать, что она может вернуть undef, или просто следить за тем, чтобы список параметров никогда не был пустым.

О лексических переменных (my)

Вообще говоря, лексические переменные могут употребляться в любом блоке, не только в пользовательских функциях. Например, их можно определять в блоках `if`, `while` или `foreach`:

```
foreach (1..10) {
  my($square) = $_ * $_;  # Приватная переменная для этого цикла
  print "$_ squared is $square.\n";
}
```

Переменная `$square` является приватной для охватывающего блока; в данном случае это блок цикла `foreach`. Если охватывающего блока нет, переменная является приватной для всего файла с исходным кодом. Пока ваши программы будут размещаться в одном файле, так что это несущественно. Но важно понимать, что *область видимости* имени лексической переменной ограничивается наименьшим охватывающим блоком или файлом. *Весь* код, который может обратиться к `$square`, подразумевая именно эту переменную, находится в соответствующей текстовой области видимости. Это обстоятельство значительно упрощает сопровождение – если `$square` примет неверное значение, причину следует искать в ограниченном блоке исходного кода. Опытные программисты знают (часто по собственному горькому опыту), что ограничение области видимости переменной страницей кода или даже несколькими строками кода заметно ускоряет циклы разработки и тестирования.

Обратите также внимание на то, что оператор `my` не изменяет контекст присваивания:

```
my($num) = @_;  # Списочный контекст, эквивалентно ($num) = @_;
my $num  = @_;  # Скалярный контекст, эквивалентно $num = @_;
```

В первом случае переменная `$num` получает первый параметр в списочном контексте присваивания; во втором она получает количество параметров в скалярном контексте. Любая из этих строк кода может соответствовать намерениям программиста; по внешнему виду строки это определить невозможно, поэтому Perl не сможет предупредить об ошибке контекста (конечно, обе строки не должны присутствовать в одной функции, поскольку в одной области видимости невозможно определить две одноименные лексические переменные; это всего лишь пример). При чтении подобного кода можно всегда определить контекст присваивания, прикинув, какой контекст использовался бы без ключевого слова `my`.

Раз уж мы заговорили об использовании `my()` с круглыми скобками, стоит запомнить, что без круглых скобок `my` только объявляет одну лексическую переменную:[1]

```
my $fred, $barney;         # ОШИБКА! $barney не объявляется
my($fred, $barney);        # Объявляются обе переменные
```

Конечно, `my` можно использовать для создания приватных массивов:[2]

```
my @phone_number;
```

Любая новая переменная изначально содержит «пустое» значение – `undef` для скаляров, пустой список для массивов.

Директива use strict

Perl старается как можно меньше ограничивать программиста.[3] Но иногда бывает желательно установить более жесткие дисциплинарные рамки; эта задача решается при помощи директивы `use strict`.

Директива представляет собой инструкцию для компилятора, которая содержит какую-либо информацию о коде. В данном случае директива `use strict` сообщает внутреннему компилятору Perl, что он должен установить более жесткие требования к программированию оставшейся части блока или исходного файла.

Почему это может быть важно? Представьте, что при написании программы вы ввели команду следующего вида:

```
$bamm_bamm = 3;  # Perl создает переменную автоматически
```

Затем вы продолжаете набирать программу. Введенная команда уходит за верхний край экрана, а вы вводите команду увеличения переменной:

```
$bammbamm += 1;  # Неверное имя!
```

Обнаружив новое имя переменной (символ подчеркивания учитывается в именах переменных!), Perl создает новую переменную и увеличивает ее на 1. Если вам повезет и вы достаточно умны для того, чтобы включить предупреждения, Perl сообщит, что одно (или оба) из этих глобальных имен используется в программе всего один раз. А если просто умны, окажется, что имена используются многократно, и Perl не сможет вас предупредить.

---

[1] Как обычно, при включенных предупреждениях Perl сообщит о сомнительном использовании `my`; вы также можете сообщить о нем сами по бесплатному телефону 1-800-LEXICAL-ABUSE. Директива `strict`, с которой вы вскоре познакомитесь, полностью запрещает его.

[2] И хешей, как вы узнаете в главе 6.

[3] Неужели еще не заметили?

Чтобы Perl применял более жесткие ограничения, включите директиву `use strict` в начало программы (или любого блока или файла, в котором должны действовать эти правила):

```
use strict;  # Enforce some good programming rules
```

Теперь, наряду с другими ограничениями[1], Perl будет требовать объявления всех новых переменных – обычно с ключевым словом `my`:[2]

```
my $bamm_bamm = 3;  # Новая лексическая переменная
```

Если теперь вы допустите ошибку в имени переменной, Perl пожалуется, что переменная с именем `$bammbamm` не объявлена в программе, а ваша ошибка будет автоматически обнаружена во время компиляции.

```
$bammbamm += 1;  # Такой переменной не существует:
                 # Фатальная ошибка компиляции
```

Конечно, требования распространяются только на новые переменные; вам не придется объявлять встроенные переменные Perl, вроде `$_` или `@_`.[3] При включении `use strict` в уже написанную программу вы обычно получаете целый поток предупреждений, поэтому лучше использовать директиву сразу.

Часто говорят, что любая программа длиной более одного экрана текста должна содержать `use strict`. И мы согласны с этим.

В дальнейшем большинство наших примеров (хотя и не все) будет записываться так, как при действии `use strict`, даже если директива в программу не включается. В частности, мы обычно объявляем переменные с ключевым словом `my` там, где это уместно. Но даже несмотря на то, что в примерах книги директива `use strict` часто опускается, мы рекомендуем вам использовать ее в своих программах как можно чаще.

---

[1] За информацией о других ограничениях обращайтесь к описанию директивы `strict` в документации. Документация по каждой директиве приводится под ее именем, так что команда *perldoc strict* (или «родной» механизм документирования вашей системы) найдет нужный раздел. В двух словах, другие ограничения требуют более частого экранирования строк и использования истинных (жестких) ссылок. Ни одно из этих ограничений не повлияет на работу новичков.

[2] Существуют и другие способы объявления переменных.

[3] И по крайней мере в некоторых обстоятельствах не стоит объявлять переменные `$a` и `$b`, используемые во внутренней реализации `sort`. При тестировании этой функции лучше использовать другие имена переменных. О том факте, что `use strict` не запрещает объявление этих двух переменных, часто сообщают как об ошибке, хотя ошибкой он не является.

Оператор return

Оператор `return` немедленно передает управление из функции в точку вызова с возвратом значения:

```
my @names = qw/ fred barney betty dino wilma pebbles bamm-bamm /;
my $result = &which_element_is("dino", @names);

sub which_element_is {
  my($what, @array) = @_;
  foreach (0..$#array) {  # Индексы элементов @array
    if ($what eq $array[$_]) {
      return $_;          # Ранний возврат при успешном поиске
    }
  }
  -1;                   # Элемент не найден (включение return не обязательно)
}
```

Эта функция должна возвращать индекс заданного элемента (`"dino"`) в массиве `@names`. Сначала в объявлении `my` указываются параметры: `$what` – искомое значение и `@array` – массив, в котором ведется поиск. В данном случае он представляет собой копию массива `@names`. Цикл `foreach` перебирает индексы `@array` (первый индекс равен 0, а последний – `$#array`, как было показано в главе 3).

При каждой итерации цикла `foreach` мы проверяем, равна[1] ли строка `$what` элементу `@array` с текущим индексом. Если строки равны, функция немедленно возвращает текущий индекс. Это самое распространенное применение ключевого слова `return` в Perl – немедленный возврат значения без выполнения оставшейся части функции.

Но что произойдет, если элемент не найден? В этом случае автор функции решил вернуть `-1` как признак «значение не найдено». Вероятно, возврат `undef` лучше соответствует стилю Perl, но этот программист выбрал `-1`. Завершить функцию командой `return -1` было бы синтаксически правильно, но ключевое слово `return` здесь необязательно.

Некоторые программисты предпочитают использовать `return` для каждого возвращаемого значения. Например, это можно сделать, когда последнее вычисляемое выражение не находится в последней строке функции, как в приводившемся ранее примере `&larger_of_fred_or_barney`. Его присутствие необязательно, но безвредно. Однако, по мнению многих программистов Perl, тратить силы на ввод семи лишних символов неразумно.

---

[1] Не правда ли, вы заметили, что вместо числового сравнения `==` мы используем строковое сравнение `eq`?

Вызов функции без &

А теперь, как и было обещано, мы расскажем, в каких случаях можно опускать знак & при вызове пользовательской функции. Если компилятор видит определение функции перед вызовом или если Perl по синтаксису может определить, что перед ним вызов функции, пользовательские функции могут вызываться без знака & по аналогии с встроенными функциями. (В этом правиле имеется одна загвоздка, но об этом чуть позднее.)

Таким образом, если Perl по одному синтаксису может определить, что он имеет дело с вызовом функции, знак & можно опустить. Иначе говоря, если вы приводите список параметров в круглых скобках, значит, это вызов функции:[1]

```
my @cards = shuffle(@deck_of_cards);  # Знак & при вызове shuffle
                                      # не обязателен
```

Если внутренний компилятор Perl уже встречал определение пользовательской функции, знак & обычно тоже можно опустить; в этом случае можно даже опустить скобки в списке аргументов:

```
sub division {
  $_[0] / $_[1];                  # Первый параметр делится на второй
}

my $quotient = division 355, 113;  # Использует &division
```

Напомню, что это возможно благодаря правилу, по которому круглые скобки всегда можно опустить, если это не изменит смысл кода.

Однако объявление пользовательской функции не должно размещаться *после* вызова, иначе компилятор просто не поймет, о чем идет речь. Чтобы компилятор мог использовать вызовы пользовательских функций по аналогии с вызовами встроенных функций, он должен сначала увидеть определение.

Но загвоздка кроется в другом: если имя пользовательской функции совпадает с именем встроенной функции Perl, при ее вызове *обязательно* должен использоваться знак &. Без него пользовательская функция будет вызвана *только* при отсутствии одноименной встроенной функции:

```
sub chomp {
  print "Munch, munch!\n";
}

&chomp;  # Знак & обязателен!
```

[1] В данном случае это пользовательская функция &shuffle. Но, как вы вскоре увидите, это может быть и встроенная функция.

Без & будет вызвана встроенная функция chomp, несмотря на то, что в программе была определена пользовательская функция chomp. Итак, на практике следует руководствоваться следующим правилом: пока вы не будете знать имена *всех* встроенных функций Perl, *всегда* используйте & при вызовах пользовательских функций (вероятно, в вашей первой сотне программ или около того). Но когда вы видите, что другой программист опустил & в своем коде, это не обязательно является ошибкой; возможно, он просто знает, что в Perl нет встроенной функции с таким именем.[1] Когда программист собирается вызывать пользовательские функции по аналогии со встроенными функциями Perl, он обычно пишет *модули* и использует *прототипы* для передачи Perl информации об ожидаемых параметрах. Впрочем, построение модулей – тема нетривиальная; когда вы будете готовы, обратитесь к документации Perl (и особенно к документам *perlmod* и *perlsub*) за дополнительной информацией о прототипах пользовательских функций и построении модулей.

Нескалярные возвращаемые значения

Возвращаемое значение пользовательской функции не ограничивается скалярным типом. Пользовательская функция, вызываемая в списочном контексте[2], может вернуть список.

Допустим, вы хотите создавать диапазоны чисел, как при использовании оператора .., но так, чтобы диапазоны могли быть упорядочены как по возрастанию, так и по убыванию. Диапазонный оператор ограничивается только возрастающими диапазонами, но эта проблема легко решается:

```
sub list_from_fred_to_barney {
  if ($fred < $barney) {
    # Отсчет по возрастанию от $fred к $barney
    $fred..$barney;
  } else {
    # Отсчет по убыванию от $fred к $barney
    reverse $barney..$fred;
  }
}
$fred = 11;
$barney = 6;
```

[1] А может быть, он *все-таки* ошибся; чтобы узнать, существует ли встроенная функция с таким именем, проведите поиск в man-страницах *perlfunc* и *perlop*. Обычно Perl сообщает о таких конфликтах в режиме предупреждений.

[2] Чтобы определить, вызывается ли функция в скалярном или списочном контексте, используйте функцию wantarray. С ее помощью вы сможете легко писать функции для скалярного или списочного контекста.

```
@c = &list_from_fred_to_barney; # @c содержит (11, 10, 9, 8, 7, 6)
```

В этом примере диапазонный оператор создает список от 6 до 11, а затем функция `reverse` обращает список, чтобы он убывал от `$fred` (11) до `$barney` (6).

Наконец, функция может не возвращать ничего. Вызов `return` без аргументов вернет `undef` в скалярном контексте или пустой список в списочном контексте. В частности, так можно вернуть признак ошибки и сообщить вызывающей стороне, что вернуть более осмысленное значение невозможно.

Статические приватные переменные

Переменные, объявленные с ключевым словом `my`, являются приватными для пользовательской функции; при каждом последующем вызове они определяются заново. С ключевым словом `state` область видимости приватных переменных по-прежнему ограничивается пользовательской функцией, но Perl сохраняет их значения между вызовами.

Вернемся к первому примеру этой главы – в нем определялась функция `marine`, которая увеличивала значение переменной:

```
sub marine {
  $n += 1;  # Глобальная переменная $n
  print "Hello, sailor number $n!\n";
}
```

Но вы уже знаете о директиве `use strict`, можете включить ее в программу... и понять, что теперь наше использование глобальной переменной `$n` стало невозможным. Переменную `$n` нельзя объявить лексической с ключевым словом `my`, потому что она не будет сохранять свое значение между вызовами.

Проблема решается объявлением переменной с ключевым словом `state`. Такие переменные сохраняют свое значение между вызовами функции, а область их видимости ограничивается пользовательской функцией:

```
use 5.010;

sub marine {
  state $n = 0;  # private, persistent variable $n
  $n += 1;
  print "Hello, sailor number $n!\n";
}
```

Новая версия функция выдает тот же результат без использования глобальной переменной и без нарушения требований `strict`. При первом вызове функции Perl объявляет и инициализирует `$n`, а при всех последующих вызовах объявление игнорируется. Между вызовами Perl сохраняет текущее значение `$n`.

С ключевым словом `state` может быть объявлена переменная любого типа, не только скалярная. Следующая пользовательская функция запоминает свои аргументы и вычисляет накапливаемую сумму с применением массива `state`:

```
use 5.010;

running_sum( 5, 6 );
running_sum( 1..3 );
running_sum( 4 );

sub running_sum {
  state $sum = 0;
  state @numbers;

  foreach my $number ( @_ ) {
    push @numbers, $number;
    $sum += $number;
    }
  say "The sum of (@numbers) is $sum";
  }
```

При каждом вызове функция выводит новую сумму, прибавляя новые аргументы ко всем аргументам от предыдущих вызовов:

```
The sum of (5 6) is 11
The sum of (5 6 1 2 3) is 17
The sum of (5 6 1 2 3 4) is 21
```

Впрочем, для массивов и хешей устанавливается небольшое ограничение: в Perl 5.10 запрещена их инициализация в списочном контексте:

```
state @array = qw(a b c); # Ошибка!
```

Сообщение об ошибке наводит на мысль, что такая возможность появится в будущей версии Perl:

```
Initialization of state variables in list context currently forbidden ...
```

Упражнения

Ответы к упражнениям приведены в приложении А.

1. [12] Напишите пользовательскую функцию с именем `total`, которая возвращает сумму чисел в списке. (Подсказка: пользовательская функция *не должна* выполнять ввод/вывод; она только обрабатывает свои параметры и возвращает значение вызывающей стороне.) Проверьте ее в приведенной ниже тестовой программе. Сумма первой группы чисел должна быть равна 25:

```
my @fred = qw{ 1 3 5 7 9 };
my $fred_total = total(@fred);
print "The total of \@fred is $fred_total.\n";
print "Enter some numbers on separate lines: ";
```

```
my $user_total = total(<STDIN>);
print "The total of those numbers is $user_total.\n";
```

2. [5] Используя функцию из предыдущего упражнения, напишите программу для вычисления суммы чисел от 1 до 1000.

3. [18] Более сложная задача: напишите пользовательскую функцию `&above_average`, которая получает список чисел и возвращает те из них, которые превышают среднее арифметическое группы. (Подсказка: напишите отдельную функцию для вычисления среднего арифметического делением суммы на количество элементов.) Проверьте написанную функцию в следующей тестовой программе:

```
my @fred = above_average(1..10);
print "\@fred is @fred\n";
print "(Should be 6 7 8 9 10)\n";
my @barney = above_average(100, 1..10);
print "\@barney is @barney\n";
print "(Should be just 100)\n";
```

4. [10] Напишите пользовательскую функцию с именем `greet`. Функция приветствует человека по имени и сообщает ему имя, использованное при предыдущем приветствии. Например, для последовательности команд

```
greet( "Fred" );
greet( "Barney" );
```

должен выводиться следующий результат:

```
Hi Fred! You are the first one here!
Hi Barney! Fred is also here!
```

5. [10] Измените предыдущую программу так, чтобы она сообщала имена всех людей, которых она приветствовала ранее. Например, для последовательности команд

```
greet( "Fred" );
greet( "Barney" );
greet( "Wilma" );
greet( "Betty" );
```

должен выводиться следующий результат:

```
Hi Fred! You are the first one here!
Hi Barney! I've seen: Fred
Hi Wilma! I've seen: Fred Barney
Hi Betty! I've seen: Fred Barney Wilma
```

5

Ввод и вывод

Вы уже умеете выполнять простейшие операции ввода/вывода; они были необходимы для выполнения некоторых упражнений. Но в этой главе операции ввода/вывода рассматриваются более подробно, а представленного материала будет достаточно для выполнения 80% ввода/вывода во всех ваших программах. Если вы уже знакомы с концепциями стандартных потоков ввода, вывода и ошибок, вам будет проще читать эту главу. Если нет, вы освоите их к концу главы. Пока можно считать, что «стандартный ввод» – это клавиатура, а «стандартный вывод» – экран монитора.

Чтение данных из стандартного ввода

Получить данные из стандартного потока ввода несложно. Мы уже делали это при помощи оператора <STDIN>.[1] Выполнение этого оператора в скалярном контексте дает следующую строку входных данных:

```
$line = <STDIN>;                # Прочитать следующую строку
chomp($line);                   # Удалить завершитель

chomp($line = <STDIN>);         # То же самое, но более идиоматично
```

При достижении конца файла оператор построчного ввода возвращает undef; это обстоятельство часто используется для выхода из цикла:

```
while (defined($line = <STDIN>)) {
  print "I saw $line";
}
```

[1] То, что мы здесь называем оператором построчного ввода <STDIN>, в действительности является оператором построчного ввода (представленным угловыми скобками), примененным к *файловому дескриптору*. Файловые дескрипторы рассматриваются в этой главе.

В первой строке происходит много всего: мы читаем входные данные в переменную, проверяем ее на определенность, и если она определена (то есть если цикл еще не достиг конца входных данных), выполняется тело цикла `while`. Таким образом, в теле цикла переменная `$line`[1] по очереди содержит каждую строку цикла. Подобная операция выполняется очень часто; вполне естественно, что в Perl для нее была создана сокращенная форма записи, которая выглядит так:

```
while (<STDIN>) {
  print "I saw $_";
}
```

Под это сокращение Ларри задействовал заведомо бесполезную синтаксическую конструкцию. *Буквально* здесь говорится следующее: «Прочитать строку входных данных и проверить ее на истинность (обычно условие выполняется). Если строка истинна, войти в цикл `while` и *отбросить прочитанную строку!*» Ларри знал, что это бесполезная операция; никому не придет в голову выполнять ее в реальной программе Perl. Ларри взял этот бесполезный синтаксис и сделал его полезным.

По сути эта конструкция означает, что Perl должен сделать то, что мы видели в более раннем цикле: прочитать входные данные в переменную и (при условии, что результат определен, то есть мы не достигли конца файла) войти в цикл `while`. Но вместо того чтобы сохранять результат в `$line`, Perl использует переменную по умолчанию `$_`, как если бы запись выглядела так:

```
while (defined($_ = <STDIN>)) {
  print "I saw $_";
}
```

Прежде чем двигаться дальше, вы должны предельно четко понять: это сокращение работает *только* в такой записи. Если оператор построчного ввода находится в другом месте (например, в отдельной команде), строка не будет прочитана в переменную `$_` по умолчанию. Сокращение работает *только* тогда, когда условие цикла `while`[2] не содержит ничего, кроме оператора построчного ввода. Если в условии появится что-нибудь еще, сокращение работать не будет.

В остальном оператор построчного ввода (`<STDIN>`) никак не связан с «главной» переменной по умолчанию Perl `$_`. Но именно в данном конкретном случае ввод сохраняется в этой переменной.

---

[1] Вероятно, вы заметили, что для входных данных не вызывается функция `chomp`. В подобных циклах вызов `chomp` не удается включить в условное выражение, поэтому там, где эта команда нужна, она часто является первой командой цикла. Примеры приводятся в следующем разделе.

[2] Хорошо, признаем: условие цикла `for` представляет собой замаскированное условие `while`, так что оно тоже подойдет.

С другой стороны, при выполнении оператора построчного ввода в списочном контексте вы получаете все (оставшиеся) строки ввода в виде списка – каждый элемент соответствует одной строке:

```
foreach (<STDIN>) {
  print "I saw $_";
}
```

Еще раз подчеркнем: оператор построчного ввода никак не связан с «главной» переменной по умолчанию. Но в данном случае в цикле `foreach` по умолчанию используется управляющая переменная `$_`. Таким образом, в этом цикле в переменную `$_` последовательно заносятся все строки входных данных.

Звучит знакомо и не без оснований: это же поведение мы встречали в цикле `while`. Разве не так?

Различия скрыты внутри. В цикле `while` Perl читает строку входных данных, помещает ее в переменную и выполняет тело цикла, после чего возвращается за следующей строкой ввода. Но в цикле `foreach` оператор построчного ввода используется в списочном контексте (так как для выполнения цикла `foreach` необходим список). Таким образом, перед началом выполнения должны быть прочитаны все входные данные. Попробуйте прочитать 400-мегабайтный журнал веб-сервера и вы мгновенно поймете эти различия! Обычно подобный код рекомендуется использовать в сокращенной записи с циклом `while`, чтобы входные данные обрабатывались по строкам (если это возможно).

Ввод данных оператором <>

В другом способе получения входных данных применяется оператор <>. Он особенно полезен при создании программ, использующих аргументы вызова по аналогии со стандартными утилитами UNIX.[1] Если вы хотите написать программу Perl, которая может использоваться в сочетании с *cat*, *sed*, *awk*, *sort*, *grep*, *lpr* и другими программами, оператор <> будет вашим другом. Если нужно что-то другое, оператор <>, скорее всего, не поможет.

Аргументами вызова программы обычно называют «слова», следующие в командной строке после имени программы.[2] В следующей ко-

[1] Причем не только в семействе UNIX. Многие другие системы переняли этот способ использования аргументов вызова.

[2] При запуске программа получает список из нуля или более аргументов вызова, которые передаются запустившей программой. Часто этой программой является командный процессор, который строит список на основании данных, введенных вами в командной строке. Но позднее вы увидите, что при запуске программы можно передать практически любые строки. Так как аргументы вызова часто берутся из командной строки, они иногда называются «аргументами командной строки».

манде они содержат имена файлов, последовательно обрабатываемых вашей программой:

```
$ ./my_program fred barney betty
```

Команда запускает программу *my_program* (находящуюся в текущем каталоге) и приказывает ей обработать сначала файл *fred*, за ним файл *barney*, а после файл *betty*.

Если при вызове аргументы не передаются, программа должна обработать стандартный поток ввода. Или в особом случае, если в одном из аргументов передается дефис (-), он тоже означает стандартный ввод.[1] Таким образом, если при вызове программы передаются аргументы `fred - betty`, подразумевается, что программа должна сначала обработать файл *fred*, затем стандартный поток ввода, а после него файл *betty*.

Почему стоит соблюдать эти соглашения в программах? Потому что вы сможете выбрать, откуда программа должна получать входные данные, во время выполнения; например, вам не придется переписывать программу, чтобы использовать ее в конвейере (об этом чуть позже). Ларри включил эту возможность в Perl, потому что он хотел упростить написание программ, работающих в стиле стандартных утилит UNIX, даже в других системах. Вообще-то он сделал это для того, чтобы *его собственные* программы работали как стандартные утилиты UNIX. Так как реализации некоторых разработчиков не следуют общим правилам, Ларри может написать собственные утилиты, установить их на разных компьютерах и быть уверенным в том, что они будут работать одинаково. Конечно, для этого необходимо портировать Perl для всех компьютеров, с которыми он собирается работать.

Оператор <> в действительности является особой разновидностью оператора построчного ввода. Но вместо того чтобы получать входные данные с клавиатуры, он берет их из источников, выбранных пользователем:[2]

```
while (defined($line = <>)) {
  chomp($line);
  print "It was $line that I saw!\n";
}
```

Таким образом, если запустить эту программу с аргументами `fred`, `barney` и `betty`, она выдаст сообщение: «It was [строка из файла fred] that I saw!», «It was [следующая строка из файла fred] that I saw!» и т. д., пока не дойдет до конца файла *fred*. После этого программа автоматически перейдет к файлу *barney*, последовательно выведет его строки и проделает то же с файлом *betty*. Обратите внимание на «бесперебойные» переходы от одного файла к другому; при использовании оператора <>

---

[1] Малоизвестный факт: большинство стандартных утилит, включая *cat* и *sed*, использует те же правила, т. е. дефис обозначает стандартный поток ввода.

[2] Естественно, среди этих источников может быть и клавиатура.

все выглядит так, словно входные файлы объединены в один большой файл.[1] Оператор <> возвращает undef (что приводит к выходу из цикла while) только в конце всех входных данных.

В данном случае мы имеем дело с особой разновидностью оператора построчного ввода, поэтому можем использовать уже знакомое сокращение для чтения ввода в $_ по умолчанию:

```
while (<>) {
  chomp;
  print "It was $_ that I saw!\n";
}
```

Эта конструкция работает так же, как предыдущий цикл, но занимает меньше места. Также обратите внимание на вызов chomp без аргумента; раз аргумент не указан, по умолчанию chomp применяется к $_. Экономия вроде бы небольшая, а приятно!

Поскольку оператор <> чаще всего применяется для обработки всего ввода, обычно его не рекомендуется использовать более чем в одном месте программы. Если окажется, что вы размещаете два оператора <> в одной программе (а особенно если второй оператор находится внутри цикла while, обрабатывающего данные из первого), программа почти наверняка работает не так, как вы ожидаете.[2] По нашему опыту обычно оказывалось, что когда новички вставляют в программу второй оператор <>, они имеют в виду $_. Помните: оператор <> читает данные, но сами данные (по умолчанию) оказываются в $_.

Если оператор <> не может открыть один из файлов и прочитать данные из него, он выводит (вроде бы) полезное диагностическое сообщение вида

```
can't open wimla: No such file or directory
```

После этого оператор <> автоматически переходит к следующему файлу. Именно такое поведение обычно характерно для *cat* или другой стандартной утилиты.

Аргументы вызова

Формально оператор <> не просматривает аргументы вызова – он работает с массивом @ARGV. Это специальный массив, который интерпретатор Perl заполняет аргументами вызова. Иначе говоря, он во всем похож на обычные массивы (если не считать прописных букв в имени),

---

[1] Если вас интересует (и даже если не интересует), имя текущего файла хранится в специальной переменной Perl $ARGV. Если данные поступают из стандартного потока ввода, вместо имени файла переменная может содержать "-".

[2] Если @ARGV заново инициализируется перед вторым оператором <>, вы на правильном пути. Переменная @ARGV будет описана в следующем разделе.

но при запуске программы @ARGV уже содержит список всех аргументов вызова.[1]

С @ARGV можно выполнять те же операции, что и с любым другим массивом; например, вы можете извлекать из него элементы вызовом shift или перебирать их в цикле foreach. Вы даже можете проверить, не начинаются ли какие-либо аргументы с дефиса, чтобы обработать их как ключи вызова (как, например, поступает сам Perl с ключом вызова -w).[2]

Оператор <> обращается к @ARGV, чтобы определить, какие имена файлов он должен использовать. Обнаружив там пустой список, он использует стандартный поток ввода; в противном случае используется список найденных файлов. Это означает, что после запуска программы, но перед первым использованием @ARGV можно изменить содержимое @ARGV. Например, следующая программа всегда обрабатывает три конкретных файла независимо от того, что пользователь ввел в командной строке:

```
@ARGV = qw# larry moe curly #;  # Принудительное чтение трех файлов
while (<>) {
  chomp;
  print "It was $_ that I saw in some stooge-like file!\n";
}
```

Запись данных в стандартный вывод

Оператор print получает список значений и последовательно передает их (в виде строк, конечно) в стандартный вывод. Никакие дополнительные символы до, после или между элементами не выводятся.[3] Если вы хотите, чтобы выводимые элементы завершались символом новой строки и отделялись друг от друга пробелами, это придется сделать вручную:

---

[1] Программисты C обычно спрашивают о argc (в Perl такой переменной нет) и о том, что случилось с собственным именем программы (оно хранится в специальной переменной Perl $0, а не в @ARGV). В зависимости от способа запуска программы также могут выполняться дополнительные действия, которые здесь не упоминаются. За более полной информацией обращайтесь к man-странице *perlrun*.

[2] Если вы собираетесь использовать более одного-двух ключей, для их стандартной обработки почти наверняка лучше воспользоваться модулем. Обращайтесь к документации модулей Getopt::Long и Getopt::Std, входящих в стандартную поставку.

[3] По крайней мере, *по умолчанию*, но эту настройку (как и многие другие в Perl) можно изменить. Однако изменение настроек по умолчанию усложнит сопровождение программы, так что это рекомендуется делать только в маленьких программах, написанных «на скорую руку», или в небольших секциях нормальных программ. О том, как изменить настройки по умолчанию, рассказано в man-странице *perlvar*.

```
$name = "Larry Wall";
print "Hello there, $name, did you know that 3+4 is ", 3+4, "?\n";
```

Естественно, это означает, что вывод массива командой `print` отличается от его интерполяции:

```
print @array;     # Вывод списка элементов
print "@array";   # Вывод строки (содержащей интерполируемый массив)
```

Первая команда `print` выводит серию элементов, один за другим, не разделяя их пробелами. Вторая команда выводит ровно один элемент, полученный в результате интерполяции `@array` в пустую строку, то есть содержимое `@array`, разделенное пробелами.[1] Таким образом, если `@array` содержит `qw/ fred barney betty /`[2], в первом случае выводится строка `fredbarneybetty`, а во втором — `fred barney betty` с разделением пробелами.

Но пока вы не решили, что всегда будете использовать вторую форму, представьте, что `@array` содержит список входных строк, не обработанных функцией `chomp`. Иначе говоря, каждая строка в списке имеет завершающий символ новой строки. Теперь первая команда `print` выведет имена `fred`, `barney` и `betty` в трех разных строках. Но вывод второй команды будет выглядеть так:

```
fred
 barney
 betty
```

Видите, откуда взялись лишние пробелы? Perl интерполирует массив и вставляет пробелы между его элементами. Таким образом, выводится первый элемент массива (`fred` и символ новой строки), затем пробел, потом следующий элемент массива (`barney` и символ новой строки), затем пробел и последний элемент массива (`betty` и символ новой строки). В результате все строки, кроме первой, выводятся с дополнительным отступом. Каждую пару недель в списках рассылки или на форумах неизменно появляются сообщения с темой, которая выглядит примерно так:

> Perl выводит лишние пробелы во всех строках, кроме первой.

Даже не читая сообщение, можно сразу догадаться, что массив с завершающими символами новой строки был интерполирован в кавычках. Так оно всегда и оказывается.

В общем случае, если вводимые строки завершаются символом новой строки, их следует просто вывести командой

---

[1] Да, пробел тоже используется по умолчанию; за дополнительной информацией снова обращайтесь к *perlvar*.

[2] Вы же понимаете, что речь идет о списке из трех элементов, верно? Это всего лишь условная запись Perl.

```
print @array;
```

Если же завершающие символы новой строки отсутствуют, они обычно добавляются вручную:

```
print "@array\n";
```

Итак, при использовании `print` с кавычками (обычно) в конец строки добавляется `\n`. Надеемся, вы запомнили, чем различаются эти две формы `print`.

Вывод программы обычно *буферизуется*. Иначе говоря, вместо немедленной отправки каждого фрагмента выводимые данные сохраняются в буфере, пока их не наберется достаточно для выполнения операции. Скажем, при сохранении вывода на диске позиционирование пишущего механизма выполняется (относительно) медленно, поэтому было бы неэффективно выполнять эту операцию при каждой записи одного-двух символов в файл. Как правило, вывод накапливается в промежуточном буфере, а потом *сбрасывается* (то есть записывается на диск или в другой приемник) только при заполнении буфера или завершении вывода по другим причинам (например, выхода из программы). Обычно буферизация предпочтительнее прямого вывода.

Но если вы (или программа) с нетерпением ожидаете вывода, возможно, вы согласитесь на некоторое снижение быстродействия и предпочтете, чтобы содержимое буфера сбрасывалось при каждом выводе. За дополнительной информацией об управлении буферизацией обращайтесь к man-страницам Perl.

Поскольку `print` получает список строк для вывода, аргументы функции обрабатываются в списковом контексте. А раз оператор `<>` (как особая разновидность оператора построчного ввода) возвращает список строк в списочном контексте, они могут использоваться совместно:

```
print <>;          # Реализация 'cat'

print sort <>;     # Реализация 'sort'
```

Откровенно говоря, стандартные команды UNIX *cat* и *sort* обладают дополнительной функциональностью, отсутствующей в этих заменителях. Но уж превзойти их по компактности точно не удастся! Теперь вы можете переписать все свои стандартные утилиты UNIX на Perl и легко портировать их на любой компьютер с установленным языком Perl независимо от того, относится ли система к семейству UNIX или нет. И вы можете быть уверены, что на разнотипных компьютерах ваши программы будут работать одинаково.[1]

---

[1] Кстати, проект PPT (Perl Power Tools), целью которого была реализация всех классических утилит UNIX на Perl, завершил реализацию почти всех утилит (и большинства игр!), но забуксовал, когда дело дошло до реализации командного процессора. Благодаря проекту PPT стандартные утилиты UNIX стали доступны для большинства других систем.

Но есть еще одно, менее очевидное обстоятельство: `print` имеет необязательные круглые скобки, которые иногда приводят к недоразумениям. Вспомните, что круглые скобки в Perl всегда можно опустить, если только это не изменит смысла выражения. Значит, следующие две команды должны привести к одному результату:

```
print("Hello, world!\n");
print "Hello, world!\n";
```

Пока все хорошо. Но в Perl есть и другое правило, которое гласит: если вызов `print` *выглядит* как вызов функции, это *и есть* вызов функции. Правило простое, но как понимать «выглядит как вызов функции»?

Вызов функции состоит из ее имени, непосредственно[1] за которым следуют круглые скобки с аргументами функции:

```
print (2+3);
```

Команда выглядит как вызов функции, а следовательно, является вызовом функции. Она выводит 5, но затем, как и любая другая функция, возвращает значение. Функция `print` возвращает *true* или *false* в зависимости от того, успешно ли была выполнена печать. Она почти всегда выполняется успешно (разве что если произойдет ошибка ввода/вывода), поэтому переменной `$result` в следующей команде обычно присваивается значение 1:

```
$result = print("hello world!\n");
```

А если результат будет использоваться как-то иначе? Допустим, вы решили умножить возвращаемое значение на 4:

```
print (2+3)*4;  # Сюрприз!
```

Встретив эту строку, Perl выводит 5, как и положено. Затем он берет возвращаемое значение `print`, равное 1, умножает его на 4 и... отбрасывает, удивляясь, почему вы ничего не сделали с вычисленным произведением. Если кто-нибудь в этот момент заглянет вам через плечо, он скажет: «Да Perl не умеет считать! Здесь должно получиться 20, а не 5!»

В этом кроется проблема необязательных круглых скобок; иногда мы, люди, забываем, к чему они относятся. При отсутствии круглых скобок `print` является списочным оператором, который выводит все элементы из последующего списка; обычно программист рассчитывает именно на такое поведение. Но если сразу же после `print` следует открывающая круглая скобка, `print` становится вызовом функции и вы-

[1] Мы говорим «непосредственно», потому что Perl запрещает присутствие символа новой строки между именем функции и открывающей круглой скобкой. В этом случае Perl рассматривает фрагмент с круглой скобкой как списочный оператор, а не как вызов функции. Это мелкая техническая деталь, которую мы упоминаем только для полноты картины. Если вы патологически любознательны, поищите полное описание в man-страницах.

водит только то, что стоит в круглых скобках. При наличии круглых скобок для Perl эта команда равносильна следующей:

```
( print (2+3) ) * 4;  # Сюрприз!
```

К счастью, Perl почти всегда помогает находить подобные ситуации, если вы включите режим предупреждений. Поэтому используйте ключ `-w` или директиву `use warnings`, по крайней мере, во время разработки и отладки программы.

Вообще говоря, это правило – «Если что-то выглядит как вызов функции, то оно и является вызовом функции» – применимо ко всем списочным функциям[1] Perl, не только к `print`. Просто при вызове `print` вы заметите его с наибольшей вероятностью. Если за `print` (или именем другой функции) следует открывающая круглая скобка, проследите за тем, чтобы парная круглая скобка находилась *после всех* аргументов этой функции.

Форматирование вывода

Иногда стандартных возможностей `print` по управлению выводом оказывается недостаточно. Многие программисты привыкли форматировать вывод удобной функцией `printf` в языке C. Не бойтесь: в Perl существует похожая функция с тем же именем.

Оператор `printf` получает форматную строку, за которой следует список выводимых элементов. Форматная[2] строка представляет собой шаблон с описанием желаемой формы вывода:

```
printf "Hello, %s; your password expires in %d days!\n",
  $user, $days_to_die;
```

Форматная строка содержит так называемые *преобразования* (*conversions*). Каждое преобразование начинается со знака процента (`%`) и завершается буквой. (Как вы вскоре узнаете, между ними могут стоять другие значащие символы.) Количество элементов в списке должно совпадать с количеством преобразований; если они не совпадают, форматный вывод будет работать некорректно. В предыдущем примере задействованы два элемента и два преобразования, поэтому вывод может выглядеть примерно так:

```
Hello, merlyn; your password expires in 3 days!
```

---

[1] Функции, получающие 0 или 1 аргумент, избавлены от этой проблемы.

[2] Термин «формат» здесь используется в его общем значении. В Perl также существует механизм построения отчетов «formats», который лишь упоминается в приложении B (не считая этой сноски), но его рассмотрение не входит в задачи книги. Так что здесь вы предоставлены сами себе. Наше дело – предупредить, чтобы вы не запутались.

Для универсального вывода числа используйте преобразование %g, которое автоматически выбирает наиболее подходящую запись: вещественную, целочисленную или даже экспоненциальную:

```
printf "%g %g %g\n", 5/2, 51/17, 51 ** 17;  # 2.5 3 1.0683e+29
```

Формат %d обозначает десятичное целое число[1], усекаемое по мере надобности:

```
printf "in %d days!\n", 17.85;  # in 17 days!
```

Обратите внимание: число именно усекается, а не округляется. О том, как выполняется округление, вы узнаете чуть позже.

В Perl функция printf чаще всего используется для вывода столбцовых данных, так как большинство форматов позволяет задать ширину поля. Если данные не помещаются в поле, оно обычно расширяется настолько, насколько потребуется:

```
printf "%6d\n", 42;  # Вывод ````42 (символом ` обозначен пробел)
printf "%2d\n", 2e3 + 1.95;  # 2001
```

Преобразование %s означает строку; фактически заданное значение интерполируется как строка, но с заданной шириной поля:

```
printf "%10s\n", "wilma";  # `````wilma
```

При отрицательной ширине поля значение выравнивается по левому краю (во всех преобразованиях):

```
printf "%-15s\n", "flintstone";  # flintstone`````
```

Преобразование %f (вещественное) округляет вывод до требуемой точности и даже позволяет задать количество цифр в дробной части:

```
printf "%12f\n", 6 * 7 + 2/3;    # ```42.666667
printf "%12.3f\n", 6 * 7 + 2/3;  # ``````42.667
printf "%12.0f\n", 6 * 7 + 2/3;  # ``````````43
```

Чтобы включить знак процента в выводимые данные, используйте запись %%; в отличие от других, она не использует элементы из списка:[2]

```
printf "Monthly interest rate: %.2f%%\n",
  5.25/12;  # "0.44%"
```

---

[1] Существуют также преобразования %x для шестнадцатеричной и %o для восьмеричной записи, если они вам потребуются.

[2] Возможно, вы подумали, что перед % достаточно поставить знак \. Хорошая попытка, но она не работает. Дело в том, что форматная строка является *выражением*, а выражение "\%" обозначает строку из одного символа '%'. Даже если вы включите обратную косую черту в форматную строку, printf не будет знать, что с ней делать. Кроме того, программисты C уже привыкли к этому синтаксису в printf.

Массивы и printf

В обычной ситуации массив не может использоваться как аргумент `printf`. Это объясняется тем, что массив может содержать любое количество элементов, а форматная строка работает с фиксированным числом элементов. Если форматная строка содержит три преобразования, элементов тоже должно быть три.

Однако ничто не мешает вам построить форматную строку «на ходу», потому что она может быть любым выражением. Впрочем, сделать это правильно может быть нелегко, поэтому форматную строку удобно хранить в переменной (особенно во время отладки):

```
my @items = qw( wilma dino pebbles );
my $format = "The items are:\n" . ("%10s\n" x @items);
## print "the format is >>$format<<\n"; # Для отладки
printf $format, @items;
```

Оператор `x` (см. главу 2) дублирует заданную строку в `@items` экземплярах (переменная используется в скалярном контексте). В данном примере количество экземпляров равно 3, так как массив состоит из трех элементов. Построенная форматная строка получается такой, как если бы она с самого начала была записана в виде `The items are:\n%10s\n%10s\n%10s\n`. А в выходных данных каждый элемент печатается в отдельной строке и выравнивается по правому краю в столбце шириной 10 символов под строкой заголовка. Впечатляет, не правда ли? Но и это еще не все, поскольку строки можно объединить:

```
printf "The items are:\n".("%10s\n" x @items), @items;
```

Массив `@items` сначала используется в скалярном контексте для получения длины, а затем в списковом контексте для получения содержимого.

Файловые дескрипторы

Файловым дескриптором называется символическое имя в программе Perl, связывающее процесс Perl с внешним миром через канал ввода/вывода. Иначе говоря, файловый дескриптор является именем *канала ввода/вывода*, а не именем файла.

Имена файловых дескрипторов подчиняются тем же правилам, что и другие идентификаторы Perl (последовательность букв, цифр и подчеркиваний, которая не может начинаться с цифры). Из-за отсутствия префиксного символа их можно спутать с существующими или будущими зарезервированными словами или метками, которые будут рассматриваться в главе 10. Как и в случае с метками, Ларри рекомендует записывать имена дескрипторов в верхнем регистре – такая запись не только делает их более заметными, но и гарантирует, что в программе не произойдет сбой при появлении в будущем новых зарезервированных слов (записываемых в нижнем регистре).

Шесть специальных имен файловых дескрипторов Perl уже использует для своих целей: STDIN, STDOUT, STDERR, DATA, ARGV и ARGVOUT.[1] Хотя вы можете присвоить файловому дескриптору любое имя по своему усмотрению, не применяйте эти шесть имен, если только вы не собираетесь пользоваться их специфическими свойствами.[2]

Возможно, некоторые из этих имен вам уже знакомы. При запуске программы файловый дескриптор STDIN обозначает канал ввода/вывода между процессом Perl и тем местом, из которого программа должна получать входные данные (стандартный поток ввода). Обычно это клавиатура пользователя, если только пользователь не выберет другой источник, например файл или вывод другой программы.[3] Существует также стандартный поток вывода STDOUT. По умолчанию данные выводятся на экран монитора, но как вы вскоре увидите, пользователь может направить вывод в файл или передать его другой программе. Эти стандартные потоки пришли к нам из библиотеки «стандартного ввода/вывода» UNIX, но они практически так же работают в большинстве современных операционных систем.[4] Общий принцип состоит в том, что ваша программа должна «слепо» читать из STDIN и записывать в STDOUT, полагаясь на то, что они правильно заданы пользователем (или программой, запустившей вашу программу). Это позволяет ввести в командном процессоре команду следующего вида:

```
$ ./your_program <dino >wilma
```

Команда сообщает командному процессору, что входные данные программы должны читаться из файла *dino*, а вывод должен записываться в файл *wilma*. Если программа «слепо» читает данные из STDIN, обрабатывает их (так, как вы сочтете нужным) и «слепо» записывает в STDOUT, все будет работать нормально.

---

[1] Некоторые программисты ненавидят запись в верхнем регистре и пытаются записывать эти имена строчными буквами (например, stdin). Возможно, время от времени ваша программа даже будет работать, но не всегда. Подробности того, когда эта запись работает, а когда нет, выходят за рамки книги. Важно то, что программы, зависящие от этой мелочи, однажды работать перестанут, поэтому записи в нижнем регистре лучше избегать.

[2] В некоторых ситуациях эти имена можно применять повторно без всяких проблем. Но программист, занимающийся сопровождением, решит, что вы используете встроенный дескриптор Perl, и может запутаться.

[3] В этой главе рассматриваются три основных потока ввода/вывода, которые используются командным процессором UNIX по умолчанию. Но конечно, программы могут запускаться не только из командного процессора. В главе 16 будет показано, что происходит при запуске другой программы из Perl.

[4] Если вы еще незнакомы с тем, как организован стандартный ввод и вывод в вашей системе, не входящей в семейство UNIX, обратитесь к man-странице *perlport* и документации по аналогу командного процессора UNIX (программе, позволяющей запускать другие программы на основании ввода с клавиатуры) для вашей системы.

От вас даже не потребуется дополнительных усилий для того, чтобы программа могла работать в конвейерах (pipelines). Это еще одна концепция из мира UNIX, которая позволяет записывать командные строки следующим образом:

```
$ cat fred barney | sort | ./your_program | grep something | lpr
```

Если вы незнакомы с этими командами UNIX, ничего страшного. Здесь всего лишь говорится, что команда *cat* должна вывести все строки файла *fred*, за которыми следуют все строки файла *barney*. Выведенные данные подаются на вход команды *sort*, которая сортирует строки и передает их *your_program*. После завершения обработки в *your_program* данные передаются программе *grep*, которая отфильтровывает одни строки, а другие отправляет команде *lpr* для печати на принтере.

Подобные каналы часто используются в UNIX и других современных системах, потому что они позволяют создавать мощные, сложные команды из простых структурных блоков. Каждый структурный блок хорошо выполняет только одну специализированную операцию; ваша задача – правильно использовать их в сочетании друг с другом.

Наряду с описанными потоками ввода/вывода существует еще один стандартный поток. Если (в предыдущем примере) *your_program* выдаст какие-либо предупреждения или другие диагностические сообщения, они не должны переходить по конвейеру. Команда *grep* игнорирует все строки, в которых отсутствуют искомые данные, поэтому, скорее всего, она проигнорирует эти предупреждения. Но даже если предупреждения сохранятся, вероятно, они не должны передаваться вниз другим программам по конвейеру. По этой причине определяется *стандартный поток ошибок* `STDERR`. Даже если стандартный вывод передается другой программе или файлу, ошибки будут направлены туда, куда хочет пользователь. По умолчанию ошибки выводятся на экран[1], но пользователь также может направить их в файл командой вида

```
$ netstat | ./your_program 2>/tmp/my_errors
```

Открытие файлового дескриптора

Итак, Perl предоставляет три стандартных дескриптора `STDIN`, `STDOUT` и `STDERR`, автоматически открытых для файлов и устройств, заданных родительским процессом (вероятно, командным процессором). Если вам понадобятся другие файловые дескрипторы, воспользуйтесь опе-

[1] Причем в общем случае ошибки не буферизуются. Следовательно, если стандартный поток ошибок и стандартный поток вывода направляются в одно место (например, на экран), ошибки могут появиться раньше основного вывода. Скажем, если программа выводит строку обычного текста, а потом пытается делить на нуль, сначала может появиться сообщение о недопустимости деления, а затем обычный текст.

ратором open. Этот оператор выдает запрос к операционной системе на открытие канала ввода/вывода между вашей программой и внешним миром. Несколько примеров:

```
open CONFIG, "dino";
open CONFIG, "<dino";
open BEDROCK, ">fred";
open LOG, ">>logfile";
```

Первая команда открывает файловый дескриптор с именем CONFIG для файла *dino*. Иначе говоря, (существующий) файл dino открывается, а все хранящиеся в нем данные становятся доступными для программы через файловый дескриптор CONFIG. Происходящее напоминает обращение к данным файла через STDIN при включении в командную строку конструкций перенаправления командного процессора вроде <dino. Более того, во втором примере использована именно такая конструкция. Вторая команда делает то же, что и первая, но знак < явно указывает, что файл должен использоваться для ввода, хотя это и так делается по умолчанию.[1]

Использовать знак < при открытии файла для ввода необязательно, но мы включаем его, потому что, как показано в третьем примере, со знаком > файл открывается для вывода. Файловый манипулятор BEDROCK открывается для вывода в новый файл с именем *fred*. По аналогии с тем, как знак > используется для перенаправления вывода в командном процессоре, выходные данные направляются в новый файл с именем *fred*. Если файл с таким именем уже существует, он стирается и заменяется новым.

Четвертый пример показывает, как использовать два знака > (тоже по аналогии с командным процессором) при открытии файла в режиме присоединения. А именно, если файл уже существует, новые данные будут дописываться в конец. Если же файл не существует, он создается по аналогии с режимом >. Например, режим присоединения удобен при работе с журнальными файлами; программа записывает несколько строк в конец журнала при каждом запуске. Собственно, поэтому в четвертом примере файловый дескриптор назван LOG («журнал»), а файл – *logfile*.

---

[1] Это может быть важно по соображениям безопасности. Как вы вскоре увидите (и как более подробно объясняется в главе 14), в именах файлов могут использоваться некоторые «волшебные» символы. Если $name содержит имя файла, выбранное пользователем, простое открытие $name открывает возможность их использования. Это может быть удобно для пользователя, а может создать угрозу безопасности. Открытие "< $name" намного безопаснее, потому что в нем явно сказано, что файл должен открываться для ввода. И все же даже оно не устраняет всех потенциальных проблем. За дополнительной информацией о различных способах открытия файлов (особенно там, где возможны проблемы с безопасностью) обращайтесь к man-странице *perlopentut*.

Вместо имени файла может использоваться любое скалярное выражение, хотя направление обычно лучше задавать явно:

```
my $selected_output = "my_output";
open LOG, "> $selected_output";
```

Обратите внимание на пробел после знака >. Perl игнорирует его[1], но пробел предотвращает неприятные сюрпризы – например, если `$selected_output` будет содержать строку `">passwd"` (что приведет к открытию файла в режиме присоединения, а не в режиме записи).

В современных версиях Perl (начиная с Perl 5.6) стал возможен вызов `open` с тремя аргументами:

```
open CONFIG, "<", "dino";
open BEDROCK, ">", $file_name;
open LOG, ">>", &logfile_name();
```

Преимущество этой формы заключается в том, что Perl никогда не спутает режим (второй аргумент) с частью имени файла (третий аргумент), а это повышает уровень безопасности.[2] Но если ваш код должен быть совместим с предыдущими версиями (например, если вы собираетесь отправить его в CPAN), либо обходитесь без этих форм, либо явно пометьте свой код как совместимый только с новыми версиями Perl.[3]

Недопустимые файловые дескрипторы

В действительности Perl не может открыть файл своими силами. Он, как и другие языки программирования, всего лишь обращается к операционной системе с запросом на открытие файла. Конечно, операционная система может отказать из-за разрешений доступа, неверного имени файла или по другим причинам.

Попытавшись прочитать данные из недопустимого файлового дескриптора (т. е. из файлового дескриптора, который не был открыт положенным образом), вы немедленно получите признак конца файла. (Для механизмов ввода/вывода, представленных в этой главе, конец файла обозначается `undef` в скалярном контексте или пустым списком

---

[1] Да, это означает, что если имя файла начинается с пробела, он тоже будет проигнорирован Perl. Если вас это беспокоит, обращайтесь к *perlfunc* и *perlopentut*.

[2] С точки зрения безопасности описанный выше механизм обладает одним важным недостатком: он открывает путь внедрения вредоносного кода в вашу безобидную программу. Когда мы познакомимся с регулярными выражениями (начиная с главы 7), вы сможете применять их для проверки пользовательского ввода. А если среди пользователей вашей программы могут оказаться вредители, прочитайте о некоторых средствах безопасности Perl в «книге с альпакой» и/или в man-странице *perlsec*.

[3] Например, директивой `use 5.6`.

в списочном контексте.) Если вы попытаетесь записать данные в недопустимый файловый дескриптор, эти данные просто пропадут.

К счастью, всех этих неприятных последствий легко избежать. Прежде всего, при включенном режиме предупреждений ключом `-w` или директивой `use warnings` Perl подскажет, если мы используем недопустимый дескриптор. Но даже до этого `open` всегда сообщает, успешно ли завершился вызов, возвращая *true* в случае успеха или *false* в случае неудачи. Это позволяет применять конструкции следующего вида:

```
my $success = open LOG, ">>logfile";  # Сохранение возвращаемого значения
if ( ! $success) {
  # Вызов open завершился неудачей
  ...
}
```

Да, так поступить *можно*. Но существует и другой способ, который будет рассмотрен в следующем разделе.

Закрытие файлового дескриптора

После завершения работы с файловым дескриптором его можно закрыть оператором `close`:

```
close BEDROCK;
```

При закрытии файлового дескриптора Perl сообщает операционной системе, что работа с потоком данных завершена, а все выведенные данные следует записать на диск на тот случай, если кто-то ожидает их появления.[1] Perl автоматически закрывает файловый дескриптор при повторном открытии (то есть если имя дескриптора будет указано в новой команде `open`) или при выходе из программы.[2]

---

[1] Если вы знакомы с системой ввода/вывода, то знаете, что это описание упрощено. В общем случае при закрытии файлового дескриптора происходит следующее: если в файле остались входные данные, они игнорируются программой. Если входные данные остались в конвейере, записывающая программа может получить сигнал о закрытии конвейера. Если вывод направляется в файл или канал, буфер сбрасывается (т. е. незавершенный вывод передается по месту назначения). Если для файлового дескриптора была установлена блокировка, она снимается. За дополнительной информацией обращайтесь к документации по средствам ввода/вывода вашей системы.

[2] Выход из программы (независимо от причины) всегда приводит к закрытию всех файловых дескрипторов, но если «сломается» сам Perl, то ожидающие передачи данные в выходных буферах не будут записаны на диске. Скажем, если ваша программа аварийно завершится от случайного деления на нуль, Perl будет нормально работать и проследит за тем, чтобы выведенные данные действительно были записаны в приемник. Но если невозможна дальнейшая работа самого Perl (например, из-за нехватки памяти или получения непредвиденного сигнала), последние выведенные данные могут пропасть без записи на диск. Обычно это не создает особых проблем.

По этой причине многие простые программы Perl обходятся без вызова `close`. Но этот вызов существует, и поборники аккуратности завершают каждый вызов `open` парным вызовом `close`. В общем случае каждый файловый дескриптор рекомендуется закрывать сразу же после прекращения работы с ним, даже если программа должна скоро завершиться.[1]

Фатальные ошибки и функция die

Сейчас мы ненадолго отвлечемся. Нам понадобится материал, не связанный напрямую с темой ввода/вывода (и не ограничиваемый ею). Речь идет о том, как прервать работу программы до ее ожидаемого завершения.

Когда в Perl происходит фатальная ошибка (например, при делении на нуль, использовании некорректного регулярного выражения или вызове пользовательской функции, которая не была объявлена), программа прерывает свое выполнение и выдает сообщение с описанием причин ошибки.[2] Но эта функциональность доступна и для разработчика в виде функции `die`, что позволяет ему создавать свои собственные фатальные ошибки.

Функция `die` выводит сообщение, переданное при вызове (в стандартный поток ошибок, куда должны поступать такие сообщения), и обеспечивает выход из программы с ненулевым кодом завершения.

Возможно, вы и не знали об этом, но каждая программа, выполняемая в UNIX (и многих других современных операционных системах), обладает числовым кодом завершения, который сообщает, успешно была выполнена программа или нет. Программы, запускающие другие программы (например, утилита *make*), по коду завершения проверяют, правильно ли прошло выполнение. Код завершения состоит всего из одного байта, и в него трудно уместить сколько-нибудь подробное описание. Традиционно значение 0 считалось признаком успешного выполнения, а любое ненулевое значение – признаком ошибки. Допустим, код 1 может означать синтаксическую ошибку в аргументах команды, 2 – ошибку на стадии обработки, 3 – ошибку при поиске кон-

---

[1] Закрытие файлового дескриптора обычно приводит к сбросу всех выходных буферов на диск и освобождению всех блокировок файла. Если другая программа ожидает освобождения тех же ресурсов, каждый файловый дескриптор должен закрываться как можно скорее. Впрочем, для многочисленных программ, выполняемых в течение одной-двух секунд, это не столь существенно. При закрытии файлового дескриптора также освобождаются теоретически ограниченные ресурсы, так что в пользу подобной аккуратности есть, как минимум, несколько доводов.

[2] По крайней мере, так происходит по умолчанию; также возможен перехват ошибок в блоках `eval`, как будет показано в главе 17.

фигурационного файла; конкретные значения определяются для каждой команды. Но значение 0 всегда означает, что выполнение прошло нормально. Если код завершения указывает на возникшую ошибку, программа (например, *make*) знает, что к следующему этапу переходить не следует.

Таким образом, предыдущий пример можно переписать в следующем виде:

```
if ( ! open LOG, ">>logfile") {
  die "Cannot create logfile: $!";
}
```

Если вызов `open` завершается неудачей, `die` завершает программу и сообщает, что создать файл журнала не удалось. Но что делает `$!` в сообщении? Это системное описание ошибки в форме, понятной для человека. В общем случае, когда система отказывается выполнить запрос программы (например, открыть файл), в переменной `$!` указывается причина (например, «отказано в доступе» или «файл не найден»). Эту же строку можно получить при помощи `perror` в языке C или других языках. В Perl это сообщение содержится в специальной переменной `$!`.[1] Желательно включать эту переменную в сообщение; она поможет пользователю разобраться в том, что же было сделано неправильно. Но если `die` применяется для обозначения ошибок, происходящих не в результате ошибки по системному запросу, не включайте переменную `$!` в сообщение; обычно в ней будет содержаться постороннее сообщение, оставшееся от внутренних операций Perl. Полезная информация останется только после ошибок, происходящих при выполнении системных запросов. При удачном завершении запроса никакой полезной информации здесь не останется.

Есть еще одна операция, которую `die` автоматически выполнит за вас: к сообщению присоединяется имя программы Perl и номер строки[2], что позволит вам легко определить, какая команда `die` в программе ответственна за преждевременный выход. Сообщение об ошибке из предыдущего кода может выглядеть так (предполагается, что переменная `$!` содержит сообщение `permission denied`):

```
Cannot create logfile: permission denied at your_program line 1234.
```

---

[1] В некоторых системах, не входящих в семейство UNIX, переменная `$!` может содержать текст вида `error number 7`; предполагается, что пользователь должен сам найти описание ошибки в документации. В Windows и VMS дополнительная информация может содержаться в переменной `$^E`.

[2] Если ошибка произошла при чтении из файла, в сообщение об ошибке включается «номер фрагмента» (обычно номер строки) файла и имя файлового дескриптора; эти данные оказываются наиболее полезными при обработке ошибок.

Весьма полезные сведения – обычно нам хочется получать в сообщениях об ошибках больше информации, чем мы сами в них включаем. Если вы не хотите выдавать номер строки и имя файла, проследите за тем, чтобы «предсмертное сообщение» завершалось символом новой строки:

```
if (@ARGV < 2) {
  die "Not enough arguments\n";
}
```

Если при вызове передано менее двух аргументов командной строки, программа сообщит об этом и завершится. В сообщении не приводится ни имя программы, ни номер строки, так как для пользователя номер строки бесполезен (в конце концов, это именно его действия стали причиной ошибки). Запомните простое эмпирическое правило: символ новой строки используется в сообщениях, указывающих на ошибку пользователя, и опускается там, где ошибка может быть исправлена во время отладки.[1]

Всегда проверяйте возвращаемое значение `open`, так как оставшаяся часть программы зависит от его успеха.

Предупреждающие сообщения и функция warn

Подобно тому как функция `die` выдает фатальную ошибку, сходную со встроенными ошибками Perl (например, делением на нуль), функция `warn` выдает предупреждение, сходное с встроенными предупреждениями Perl (например, если `undef` используется так, как если бы это было определенное значение, при включенном режиме предупреждений).

Функция `warn` работает так же, как `die`, кроме последнего шага – она не прерывает работу программы. Но она, так же как `die`[2], выводит имя программы и номер строки и отправляет сообщение в стандартный вывод.

---

[1] Имя программы хранится в специальной переменной `$0`; его можно включить в строку: `$0:Not enough arguments\n`. Например, эта информация может пригодиться при конвейерной передаче или в сценариях командного процессора, когда на первый взгляд неочевидно, какая команда выдает сообщение об ошибке. Следует помнить, что значение `$0` может изменяться во время выполнения программы. Для получения информации, опущенной в результате добавления символа новой строки, можно использовать специальные лексемы `__FILE__` и `__LINE__` (или функцию `caller`) ; вы сможете вывести эту информацию в любом формате по своему усмотрению.

[2] Предупреждения, в отличие от фатальных ошибок, не могут перехватываться в блоках `eval`. Если вам все же потребуется перехватить предупреждение, обращайтесь к документации по псевдосигналу `__WARN__` (в man-странице *perlvar*).

Итак, от разговоров о «смерти» программ и суровых предупреждениях мы возвращаемся к обычному учебному материалу о вводе/выводе. Продолжайте читать.

Использование файловых дескрипторов

Когда файловый дескриптор будет открыт для чтения, чтение строк из него осуществляется практически так же, как чтение из стандартного ввода с дескриптором `STDIN`. Например, чтение строк из файла паролей UNIX происходит так:

```
if ( ! open PASSWD, "/etc/passwd") {
  die "How did you get logged in? ($!)";
}

while (<PASSWD>) {
  chomp;
  ...
}
```

В этом примере в сообщении `die` переменная `$!` заключена в круглые скобки. Это всего лишь литеральные символы, в которых выводится сообщение в выходных данных. (Иногда в Perl знак препинания – это просто знак препинания.) Как видите, то, что мы называли «оператором построчного вывода», в действительности состоит из двух компонентов: угловых скобок (собственно оператор построчного ввода) и заключенного в них файлового дескриптора.

Файловый дескриптор, открытый для записи или присоединения, может использоваться с функцией `print` или `printf`. Он указывается сразу же после ключевого слова, но перед списком аргументов:

```
print LOG "Captain's log, stardate 3.14159\n";  # Ввыод направляется в LOG
printf STDERR "%d percent complete.\n", $done/$total * 100;
```

Вы заметили, что между дескриптором и выводимыми элементами нет запятой?[1] С круглыми скобками это выглядит особенно странно. Обе следующие формы верны:

```
printf (STDERR "%d percent complete.\n", $done/$total * 100);
printf STDERR ("%d percent complete.\n", $done/$total * 100);
```

[1] В документации это называется «неявным синтаксисом объекта». Возможно, особо одаренный читатель скажет: «А, конечно! Теперь понятно, почему за файловым дескриптором нет запятой – это же неявный объект!» Мы к категории особо одаренных не принадлежим и не понимаем, почему здесь нет запятой... а опускаем ее только по одной причине: Ларри сказал, что запятая здесь не нужна.

Изменение файлового дескриптора вывода по умолчанию

По умолчанию, если при вызове `print` (или `printf` – все, что мы здесь говорим об одной функции, в равной степени относится к другой) не указан файловый дескриптор, данные направляются в `STDOUT`. Однако этот файловый дескриптор по умолчанию можно сменить оператором `select`. В следующем фрагменте несколько выходных строк отправляются в `BEDROCK`:

```
select BEDROCK;
print "I hope Mr. Slate doesn't find out about this.\n";
print "Wilma!\n";
```

Файловый дескриптор, выбранный по умолчанию для вывода, остается выбранным на будущее. Чтобы не создавать путаницу в программе, обычно рекомендуется возвращать `STDOUT` на место сразу же после завершения.[1] По умолчанию вывод в каждый файловый дескриптор буферизуется. Если задать специальной переменной `$|` значение 1, текущий файловый дескриптор (то есть выбранный на момент изменения переменной) всегда сбрасывает буфер после каждой операции вывода. Таким образом, если вы хотите быть уверены в том, что все записи сохраняются в журнале одновременно (например, если вы читаете данные из журнала, отслеживая ход продолжительного выполнения программы), используйте конструкцию следующего вида:

```
select LOG;
$| = 1;  # Не оставлять данные LOG в буфере
select STDOUT;
# ... Проходят годы, сдвигаются тектонические плиты, а потом...
print LOG "This gets written to the LOG at once!\n";
```

Повторное открытие стандартного файлового дескриптора

Ранее мы уже упоминали о том, что при повторном открытии файлового дескриптора (то есть если вы открываете файловый дескриптор `FRED`, хотя в программе уже имеется открытый файловый дескриптор `FRED`) старый дескриптор закрывается автоматически. И мы сказали, что использовать имена шести стандартных файловых дескрипторов

[1] В маловероятном случае, когда вместо `STDOUT` выбран какой-то другой дескриптор, вы можете сохранить и восстановить его способом, приведенным в описании `select` man-страницы *perlfunc*. И раз уж мы все равно переадресуем вас к этой man-странице, необходимо отметить, что в Perl существуют *две* встроенные функции с именем `select` и обе они описаны в man-странице *perlfunc*. Одна функция `select` всегда вызывается с четырьмя аргументами, поэтому ее иногда так и называют «`select` с четырьмя аргументами».

нежелательно, если только вы не собираетесь наделить их специальными возможностями. Также мы говорили, что сообщения `die` и `warn` вместе с внутренними сообщениями Perl автоматически направляются в `STDERR`. Если объединить все сказанное, становится ясно, как направить сообщения об ошибках в файл вместо стандартного потока ошибок вашей программы:[1]

```
# Отправлять ошибки в личный журнал
if ( ! open STDERR, ">>/home/barney/.error_log") {
  die "Can't open error log for append: $!";
}
```

После повторного открытия `STDERR` все сообщения об ошибках, поступающие от Perl, заносятся в новый файл. Но что произойдет при выполнении `die` – куда попадет *это* сообщение, если новый файл не удалось открыть для приема сообщений?

Если при повторном открытии одного из трех системных файловых дескрипторов – `STDIN`, `STDOUT` или `STDERR` – произойдет ошибка, Perl любезно восстанавливает исходное значение.[2] То есть Perl закрывает исходный дескриптор (любой из этих трех) только тогда, когда он видит, что открытие нового канала прошло успешно. Таким образом, этот прием может использоваться для перенаправления любых из трех системных файловых дескрипторов внутри вашей программы[3] – почти так же, как если бы программа изначально была запущена в командном процессоре с перенаправлением ввода/вывода.

Вывод функцией say

Perl 5.10 позаимствовал встроенную функцию `say` из текущей разработки Perl 6 (которая, возможно, взяла за образец функцию `println` из Pascal). Функция `say` делает то же, что `print`, но завершает вывод сим-

---

[1] Не поступайте так без веских причин. Почти всегда лучше разрешить пользователю задать перенаправление при запуске программы вместо того, чтобы жестко кодировать его. Однако данная возможность может быть удобна при автоматическом запуске вашей программы другой программой (скажем, веб-сервером или планировщиком типа `cron` или `at`). Возможны и другие причины: например, ваша программа запускает другой процесс (вероятно, командой `system` или `exec` – см. главу 16) и вам нужно, чтобы этот процесс имел другие каналы ввода/вывода.

[2] По крайней мере, если вы не изменяли специальную переменную Perl `$^F`, которая сообщает Perl, что подобное поведение должно использоваться только для этих файлов. Но эту переменную вообще никогда изменять не следует.

[3] Но не открывайте `STDIN` для вывода или другие дескрипторы для ввода. Только представьте, сколько проблем это создаст при сопровождении вашей программы.

волом новой строки. Все следующие команды выводят одинаковые строки:

```
use 5.010;

print "Hello!\n";
print "Hello!", "\n";
say "Hello!";
```

Чтобы вывести значение переменной с символом новой строки, не нужно создавать лишнюю строку или выводить список вызовом `print` — просто используйте `say`. Это особенно удобно в распространенной ситуации, когда все выводимые данные должны завершаться переводом строки:

```
use 5.010;

my $name = 'Fred';
print "$name\n";
print $name, "\n";
say $name;
```

Чтобы интерполировать массив, вам все равно придется заключить его имя в кавычки. Только в этом случае выводимые элементы будут разделяться пробелами:

```
use 5.010;

my @array = qw( a b c d );

say @array;   # "abcd\n"

say "@array"; # "a b c d\n";
```

По аналогии с `print` при вызове `say` можно задать файловый дескриптор:

```
use 5.010;

say BEDROCK "Hello!";
```

Так как функция относится к числу нововведений Perl 5.10, мы используем ее только в том случае, если в программе задействованы другие возможности Perl 5.10. Старая верная команда `print` работает ничуть не хуже, но мы подозреваем, что некоторые программисты Perl с радостью избавятся от ввода четырех лишних символов (два в имени и `\n`).

Упражнения

Ответы к упражнениям приведены в приложении А.

1. [7] Напишите программу, которая работает как *cat*, но выводит порядок выходных строк. (В некоторых системах существует такая утилита с именем *tac*.) Если вы запустите свою программу коман-

дой `./tac fred barney betty`, программа должна вывести все строки файла *betty* от последней к первой, затем все строки файла *barney* и только потом строки файла *fred* (тоже начиная с последней). Не забудьте использовать `./` при вызове своей программы, если назовете ее *tac*, чтобы не запустить вместо нее системную утилиту!

2. [8] Напишите программу, которая предлагает пользователю ввести список строковых значений в отдельных строках и выводит каждое значение выровненным по правому краю в столбце шириной 20 символов. Чтобы быть уверенным в том, что вывод находится в правильных столбцах, также выведите «линейку» из цифр (просто для упрощения отладки). Проследите за тем, чтобы по ошибке не использовался столбец из 19 символов! Например, при вводе текста `hello,good-bye` результат должен выглядеть примерно так:

```
12345678901234567890123456789012345678901234567890123456789012345678901234567890
                   hello
              good-bye
```

3. [8] Измените предыдущую программу так, чтобы пользователь мог выбрать ширину столбца. Например, при вводе значений `30,hello,good-bye` (в отдельных строках) текст должен печататься до 30 столбца. (Подсказка: о том, как управлять интерполяцией переменных, рассказано в главе 2.) Чтобы задание стало более интересным, увеличивайте длину линейки при больших введенных значениях.

6

Хеши

В этой главе представлена одна из особенностей Perl, благодаря которой он считается одним из выдающихся языков программирования: *хеши*.[1] При всей мощи и практической полезности хешей многие программисты годами работают на других известных языках, не имея с ними дела. Но вы будете использовать хеши едва ли не во всех программах Perl, которые вам предстоит написать, – настолько они полезны.

Что такое хеш?

Хеш представляет собой структуру данных, которая, как и массив, может содержать произвольное количество элементов и производить их выборку по мере надобности. Но вместо выборки по *числовым* индексам, как это делается с массивами, выборка из хешей производится по имени. Иначе говоря, *индексы* (здесь мы будем называть их *ключами*) представляют собой не числа, а произвольные уникальные строки (рис. 6.1).

Ключи являются *строками*. Соответственно вместо выборки из массива элемента с номером 3 мы обращаемся к элементу хеша по имени `wilma`.

Ключи являются произвольными строками – любое строковое выражение может использоваться в качестве ключа хеша. К тому же они уникальны: подобно тому как в массиве имеется только один элемент с номером 3, в хеше существует только один элемент с ключом `wilma`.

Хеш также можно представить как «бочку с данными» (рис. 6.2), в которой к каждому элементу прикреплен ярлык. Вы можете запустить

[1] Когда-то они назывались «ассоциативными массивами». Но около 1995 года сообщество Perl решило, что название слишком длинное, и перешло на название «хеши».

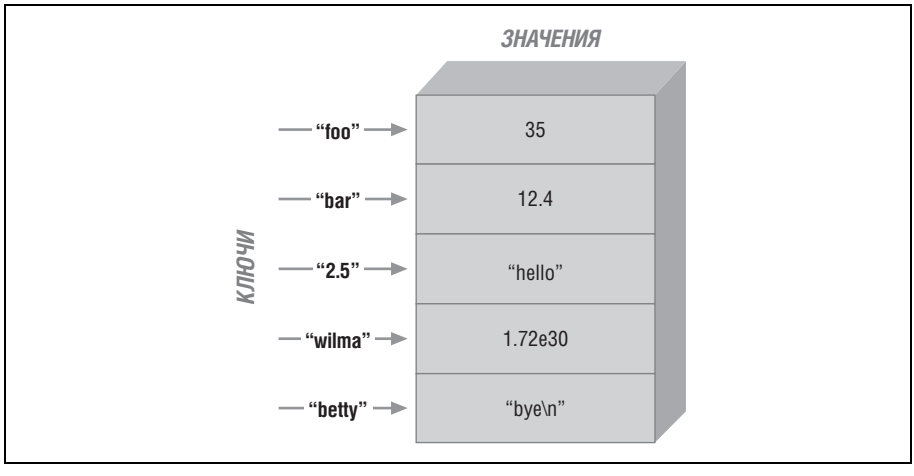


Рис. 6.1. *Ключи и значения хеша*

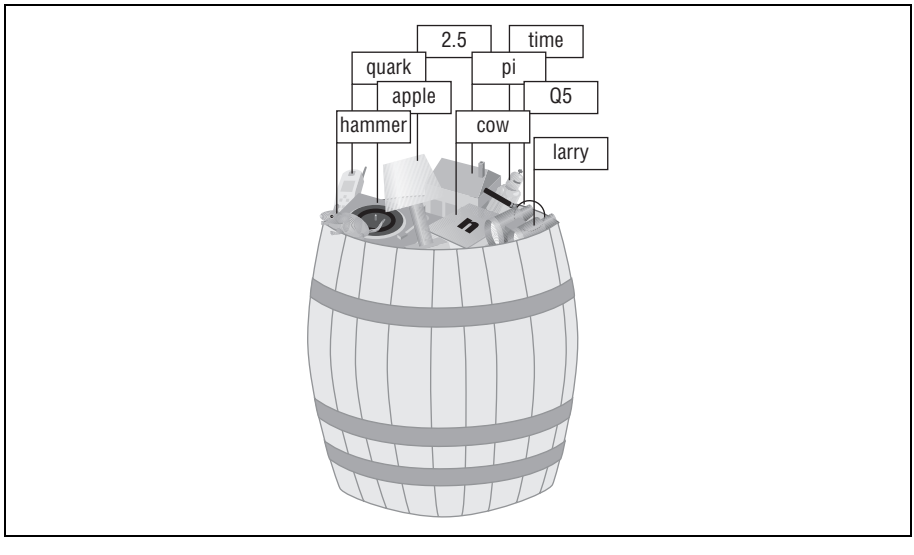


Рис. 6.2. *Хеш как «бочка с данными»*

руку в бочку, вытащить любой ярлык и посмотреть, какие данные на нем «висят». Однако в бочке не существует «первого» элемента, все элементы лежат вперемежку. В массиве перебор начинается с элемента 0, затем следует элемент 1, затем элемент 2 и т. д. В хеше нет ни фиксированного порядка, ни первого элемента. Его содержимое представляет собой множество пар «имя-значение».

Ключи и значения являются произвольными скалярными значениями, но ключи всегда преобразуются в строки. Следовательно, если использовать числовое выражение `50/20` в качестве ключа[1], оно преобразуется в строку из трех символов `"2.5"`, соответствующую одному из ключей на рис. 6.1.

Как обычно, действует принятая в Perl философия «отсутствия искусственных ограничений»: хеш может иметь произвольный размер – от пустого хеша с нулем пар «ключ-значение» до заполнения всей свободной памяти.

Некоторые реализации хешей (например, в языке *awk*, из которого Ларри позаимствовал идею) с увеличением хеша работают все медленнее и медленнее. В Perl это не так – в нем используется хороший эффективный масштабируемый алгоритм.[2] Таким образом, если хеш состоит всего из трех пар «ключ-значение», Perl очень быстро «запускает руку в бочку» и извлекает нужный элемент. Если хеш состоит из трех *миллионов* пар, выборка пройдет практически с такой же скоростью. Не бойтесь больших хешей.

Также стоит снова напомнить, что ключи всегда уникальны, тогда как значения могут повторяться. В хеше могут храниться числа, строки, значения `undef` в любых комбинациях.[3] При этом ключи должны быть произвольными, но уникальными строками.

Зачем использовать хеш?

Когда вы впервые слышите о хешах (особенно если за плечами у вас долгая продуктивная карьера программирования на языках, в которых хеши не поддерживаются), может возникнуть вопрос: для чего нужны эти странные создания? Вообще говоря, они применяются тогда, когда один набор данных как-то связан с другим набором данных. Несколько примеров хешей, встречающихся в типичных приложениях Perl:

Имя, фамилия

Имя (ключ) используется для выборки фамилии (значение). Например, по ключу `tom` хеш вернет значение `phoenix`. Конечно, для этого имена должны быть уникальными; если в данных присутствуют два человека с именем `randal`, хеш работать не будет.

---

[1] Числовое выражение, а не строку из пяти символов `"50/20"`. Если использовать строку из пяти символов в качестве ключа хеша, она, естественно, останется той же строкой из пяти символов.

[2] Perl заново строит хеш-таблицу так, как требуется для больших хешей. Более того, сам термин «хеш» обусловлен тем фактом, что для реализации этой структуры данных применяются хеш-таблицы.

[3] А точнее любые скалярные значения, включая скалярные типы, которые в этой книге не рассматриваются.

Имя хоста, *IP-адрес*

Вероятно, вы знаете, что каждый компьютер в Интернете обладает именем хоста (вида *http://www.stonehenge.com*) и числовым IP-адресом вида 123.45.67.89. Компьютерам удобнее работать с числами, а люди предпочитают запоминать имена. Имена хостов являются уникальными строками и могут использоваться в качестве ключей хеша. По имени хоста программа находит соответствующий IP-адрес.

IP-адрес, *имя хоста*

Возможно и обратное преобразование. Обычно мы рассматриваем IP-адрес как серию чисел, но его также можно преобразовать в уникальную строку, поэтому IP-адреса могут использоваться в качестве ключей хеша для выборки соответствующих имен хостов. Обратите внимание: этот хеш не эквивалентен описанному в предыдущем примере; выборка в хешах производится только в одном направлении – от ключа к значению. Невозможно передать значение и получить соответствующий ему ключ! Таким образом, эти два хеша составляют пару: в одном хранятся IP-адреса, а в другом имена хостов. Впрочем, как вы вскоре увидите, при наличии одного хеша можно легко построить другой.

Слово, *количество экземпляров*

Очень типичное использование хеша. Оно настолько типично, что даже войдет в одно из упражнений в конце главы!

Идея проста: вы хотите знать, сколько раз некоторое слово встречается в документе. Представьте, что вы индексируете группу документов, чтобы при поиске по строке `fred` узнать, что в одном документе строка `fred` встречается пять раз, в другом – семь, а в третьем не упоминается вовсе. Это позволит ранжировать документы по количеству вхождений. Когда программа индексации читает документ, при каждом обнаружении строки `fred` она увеличивает значение, связанное с ключом `fred`, на 1. Таким образом, если ранее строка `fred` дважды встречалась в документе, значение в хеше будет равно 2, а теперь оно увеличивается до 3. Если строка `fred` ранее не встречалась, значение изменяется с `undef` (значение по умолчанию) на 1.

Имя пользователя, *количество [неразумно] используемых блоков на диске*

Специально для системных администраторов: имена пользователей в системе уникальны, поэтому они могут использоваться в качестве ключей для получения информации о пользователе.

Номер водительских прав, *имя*

В мире существует великое множество людей по имени Джон Смит, но номера водительских прав у них разные. Номер образует уникальный ключ, а имя владельца – значение.

Хеш также можно рассматривать как *очень* простую базу данных, в которой с каждым ключом ассоциируется всего один блок данных. Если в описание задачи входят такие выражения, как «поиск дубликатов», «уникальность», «перекрестные ссылки» или «таблица преобразования», скорее всего, хеш будет полезен в ее реализации.

Обращение к элементам хеша

Чтобы обратиться к элементу хеша, используйте синтаксис следующего вида:

```
$hash{$some_key}
```

Он напоминает синтаксис обращения к элементам массива, но вместо индекса в квадратных скобках указывается ключ в фигурных скобках.[1] Вдобавок ключ задается строковым, а не числовым выражением:

```
$family_name{"fred"} = "flintstone";
$family_name{"barney"} = "rubble";
```

На рис. 6.3 показано, как выглядят созданные элементы хеша.

Это позволяет применять конструкции следующего вида:

```
foreach $person (qw< barney fred >) {
  print "I've heard of $person $family_name{$person}.\n";
}
```

Имена хешей подчиняются тем же правилам, что и остальные идентификаторы Perl (последовательность букв, цифр и символов подчеркивания, которая не может начинаться с цифры). Кроме того, они существуют в отдельном пространстве имен, т. е. элемент хеша `$family_name{"fred"}` никак не связан, например, с пользовательской функцией `&family_name`. Конечно, не стоит запутывать программу, присваивая разным сущностям одинаковые имена. Но Perl не станет возражать, если вы создадите скаляр с именем `$family_name` и элементы массива

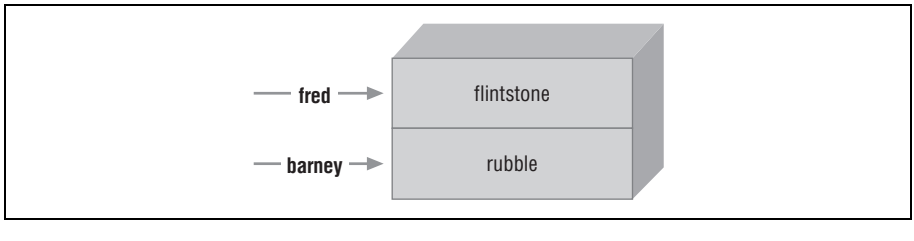


***Рис. 6.3.** Элементы хеша с назначенными ключами*

[1] Вот как это объясняет Ларри Уолл: мы используем фигурные скобки вместо квадратных, потому что для более экзотической операции, нежели простое обращение к элементу массива, нужен более экзотический синтаксис.

$family_name[5]. Мы, люди, должны поступать так же, как поступает Perl, – смотреть на знаки до и после идентификатора, чтобы понять, что он означает. Если перед именем стоит знак $, а после него – фигурные скобки, значит, происходит обращение к элементу хеша.

Разумеется, ключ хеша может задаваться любым выражением, не только строковыми литералами и простыми скалярными переменными:

```
$foo = "bar";
print $family_name{ $foo . "ney" };  # Выводит "rubble"
```

При сохранении данных в существующем элементе хеша они заменяют предыдущее значение:

```
$family_name{"fred"} = "astaire";  # Задает новое значение
                                   # для существующего элемента
$bedrock = $family_name{"fred"};   # Возвращает "astaire"; старое
                                   # значение теряется
```

По сути происходит то же самое, что при использовании массивов и скаляров: если вы сохраняете новое значение в $pebbles[17] или $dino, старое значение при этом заменяется. Если сохранить новое значение в $family_name{"fred"}, старое значение также будет заменено.

Элементы хешей создаются присваиванием:

```
$family_name{"wilma"} = "flintstone";             # Добавляет новый ключ
                                                  # (и значение)
$family_name{"betty"} .= $family_name{"barney"};  # Создает новый элемент
                                                 # в случае необходимости
```

И здесь происходит то же самое, что с массивами и скалярами: если ранее элемент $pebbles[17] или переменная $dino не существовали, они автоматически создаются присваиванием. Если ранее элемент $family_name{"betty"} не существовал, теперь он существует.

При обращении к несуществующему элементу хеша возвращается undef:

```
$granite = $family_name{"larry"};  # Ключа larry нет: undef
```

И снова здесь происходит то же, что с массивами и скалярами: если элемент $pebbles[17] или переменная $dino не были инициализированы, обращение к ним возвращает undef. Если элемент хеша $family_name{"larry"} не был инициализирован, обращение к нему также возвращает undef.

Хеш как единое целое

Чтобы сослаться на весь хеш (а не на его отдельный элемент), используйте префикс % (процент). Например, хеш, элементы которого мы использовали на нескольких предыдущих страницах, на самом деле называется %family_name.

Для удобства хеш можно преобразовать в список, и наоборот. Присваивание хешу (в данном случае показанному на рис. 6.1) выполняется в списочном контексте, где список состоит из пар «ключ-значение»:[1]

```
%some_hash = ("foo", 35, "bar", 12.4, 2.5, "hello",
      "wilma", 1.72e30, "betty", "bye\n");
```

Значение хеша (в списочном контексте) представляет собой простой список пар «ключ-значение»:

```
@any_array = %some_hash;
```

Мы называем эту конструкцию – преобразование хеша обратно в список пар «ключ-значение» – *раскруткой* хеша. Конечно, порядок следования пар может отличаться от исходного списка:

```
print "@any_array\n";
  # Примерный результат:
  #  betty bye (новая строка) wilma 1.72e+30 foo 35 2.5 hello bar 12.4
```

Порядок изменился, потому что Perl хранит пары «ключ-значение» в удобном для себя порядке, обеспечивающем максимальную скорость поиска. Хеш используется в тех случаях, когда порядок элементов неважен или имеется простой способ приведения их к желаемому порядку.

Конечно, несмотря на изменение порядка пар «ключ-значение», каждый ключ остается «прикрепленным» к соответствующему значению в итоговом списке. Таким образом, хотя мы не знаем, в какой позиции списка будет находиться ключ `foo`, можно быть уверенным в том, что соответствующее значение (38) будет находиться сразу же за ним.

Присваивание хешей

Хотя эта операция выполняется довольно редко, но хеши могут копироваться с использованием очевидного синтаксиса:

```
%new_hash = %old_hash;
```

В действительности при этом Perl выполняет больше работы, чем видно на первый взгляд. В отличие от языков типа Pascal или C, где такая операция сводилась бы к простому копированию блока памяти, структуры данных Perl более сложные. Следовательно, эта строка кода приказывает Perl раскрутить `%old_hash` в список пар «ключ/значение», а затем последовательно, пару за парой, присвоить его `%new_hash`.

---

[1] Хотя при присваивании может использоваться любое списочное выражение, оно должно иметь четное количество элементов, потому что хеш состоит из пар «ключ-значение». Нечетное количество элементов, скорее всего, приведет к всяким неожиданностям, хотя Perl может предупредить об этом нарушении.

Впрочем, чаще всего присваивание хешей сопровождается некоторым преобразованием. Например, хеш можно обратить, то есть переставить его элементы в обратном порядке:

```
%inverse_hash = reverse %any_hash;
```

Команда берет `%any_hash` и раскручивает его в список пар «ключ-значение»; образуется список вида (*ключ*, *значение*, *ключ*, *значение*...). Вызов `reverse` переставляет элементы списка в обратном порядке: (*значение*, *ключ*, *значение*, *ключ*...). Теперь ключи стоят там, где раньше были значения, и наоборот. При сохранении результата в `%inverse_hash` выборка производится по строкам, которые были значениями в `%any_hash`, – в `%inverse_hash` они стали ключами. А в результате выборки мы получаем значение, которое принадлежало в числу ключей `%any_hash`. Таким образом, выборка по «значению» (которое стало ключом) дает «ключ» (который стал значением).

Конечно, обращенный хеш будет нормально работать только в том случае, если значения исходного хеша были уникальными, в противном случае в новом хеше появятся дубликаты ключей, а ключи всегда должны быть уникальными. В подобных ситуациях в Perl используется простое правило: последнее добавление элемента побеждает. Иначе говоря, элементы, добавленные в список позднее, заменяют более ранние элементы. Конечно, мы не знаем, в каком порядке пары «ключ-значение» будут храниться в списке, поэтому невозможно заранее предсказать, какая пара «победит». Этот прием следует использовать только в том случае, если вы знаете, что среди исходных значений нет дубликатов.[1] Но в приводившемся ранее примере с IP-адресами и именами хостов дело обстоит именно так:

```
%ip_address = reverse %host_name;
```

Теперь можно одинаково легко осуществлять поиск как по IP-адресу, так и по имени хоста.

«Большая стрелка»

При присваивании списка хешу иногда бывает неочевидно, какие элементы списка являются ключами, а какие – значениями. Например, если мы захотим узнать, чем является 2.5 в следующей команде присваивания (которая уже встречалась нам ранее), нам придется вручную отсчитывать элементы списка – «ключ, значение, ключ, значение»:

```
%some_hash = ("foo", 35, "bar", 12.4, 2.5, "hello",
      "wilma", 1.72e30, "betty", "bye\n");
```

[1] Или если присутствие дубликатов вас не интересует. Например, обращение хеша `%family_name` (в котором ключами являются имена, а значениями – фамилии) позволит легко определить, присутствует ли в группе запись с заданной фамилией. Скажем, если в обращенном хеше нет ключа `slate`, мы знаем, что записи с такой фамилией в исходном хеше не было.

Разве не удобно было бы иметь механизм определения пар «ключ-значение» в таких списках, чтобы мы могли с первого взгляда определить, чем является тот или иной элемент? Ларри тоже так думал, поэтому он изобрел «большую стрелку» =>.[1] С точки зрения Perl «большая стрелка» является всего лишь альтернативным способом «изображения» запятой, поэтому иногда ее называют «жирной запятой». Иначе говоря, в грамматике Perl во всех ситуациях, где может использоваться запятая (,), ее можно заменить «большой стрелкой»; для Perl эти обозначения эквивалентны.[2] А следовательно, определение хеша с именами и фамилиями может выглядеть так:

```
my %last_name = (  # Хеш может быть лексической переменной
  "fred"   => "flintstone",
  "dino"   => undef,
  "barney" => "rubble",
  "betty"  => "rubble",
);
```

Этот синтаксис позволяет легко (или, по крайней мере, гораздо легче) увидеть, какая фамилия соответствует тому или иному имени, даже если в одной строке будут размещаться несколько пар. Обратите также внимание на завершающую запятую в конце списка. Как было показано ранее, она безвредна, но удобна. Если в список будут добавляться новые элементы, нам достаточно проследить за тем, чтобы каждая строка содержала пару «ключ-значение» и завершалась запятой. Perl увидит, что каждая пара отделяется от следующей пары запятой и что еще одна (безвредная) запятая завершает список.

Функции хешей

Естественно, в Perl существуют удобные вспомогательные функции для работы с хешами в целом.

Функции keys и values

Функция `keys` строит список всех ключей хеша, а функция `values` строит список соответствующих значений. Если хеш не содержит элементов, то обе функции возвращают пустой список:

---

[1] Да, существует и *маленькая* стрелка (->), применяемая при создании ссылок. В книге эта тема не рассматривается; когда вы будете готовы к ней, обращайтесь к man-страницам *perlreftut* и *perlref*.

[2] Вообще-то есть одно техническое различие: любое «тривиальное слово» (bareword) (последовательность букв, цифр и символов подчеркивания, не начинающаяся с цифры, но с возможным префиксом + или -) слева от «большой стрелки» неявно заключается в кавычки. Следовательно, слева от «большой стрелки» кавычки можно не указывать. Кавычки в фигурных скобках хеша также можно опускать, если в качестве ключа используется тривиальное слово.

```
my %hash = ("a" => 1, "b" => 2, "c" => 3);
my @k = keys %hash;
my @v = values %hash;
```

Список `@k` будет содержать элементы `"a"`, `"b"` и `"c"`, а `@v` – элементы 1, 2 и 3. Порядок следования элементов заранее неизвестен; напомним, что в хешах Perl элементы не упорядочиваются. Но в каком бы порядке ни следовали ключи, значения будут следовать в том же порядке: скажем, если `"b"` находится на последней позиции в списке ключей, то 2 будет на последней позиции в списке значений; если `"c"` – первый ключ, то 3 будет первым значением, и т. д. Это утверждение истинно при условии, что хеш не изменяется между получением списков ключей и значений. При добавлении элементов в хеш Perl сохраняет право переставить элементы так, как сочтет нужным, для обеспечения скорости выборки.[1] В скалярном контексте эти функции возвращают количество элементов в хеше (пар «ключ-значение»). Они делают это весьма эффективно, без перебора всех элементов хеша:

```
my $count = keys %hash;  # Возвращает 3 (три пары "ключ-значение")
```

Время от времени хеши встречаются в логических (*true/false*) выражениях следующего вида:

```
if (%hash) {
  print "That was a true value!\n";
}
```

Выражение истинно в том (и только в том) случае, если хеш содержит хотя бы одну пару «ключ-значение».[2] Таким образом, условие фактически означает: «Если хеш не пуст...» Но в целом такая конструкция встречается довольно редко.

Функция each

Если вы желаете последовательно перебрать все элементы хеша, это можно сделать несколькими способами. Один из них основан на применении функции `each`, возвращающей очередную пару «ключ-значение» в виде списка из двух элементов.[3] При каждом вызове этой функ-

---

[1] Конечно, если между вызовами `keys` и `values` в хеш будут добавляться новые элементы, то размер списка изменится и элементы второго списка будет трудно сопоставить с элементами первого списка. Ни один нормальный программист так поступать все равно не будет.

[2] Непосредственный результат представляет собой внутреннюю отладочную строку, предназначенную для коллектива разработчиков Perl. Сама строка имеет вид `"4/16"`, но ее значение заведомо истинно при наличии элементов в хеше и ложно при отсутствии элементов, поэтому мы тоже можем использовать ее с этой целью.

[3] Другой способ перебора хеша основан на использовании цикла `foreach` для списка ключей из хеша; он будет продемонстрирован в конце раздела.

ции для одного хеша возвращается следующая пара «ключ-значение», пока не будут получены все элементы. Когда пар больше не остается, each возвращает пустой список.

На практике each может нормально использоваться только в цикле while следующего вида:

```
while ( ($key, $value) = each %hash ) {
  print "$key => $value\n";
}
```

В этом фрагменте происходит довольно много всего. Сначала each %hash возвращает пару «ключ-значение» из хеша в виде списка с двумя элементами; допустим, это ключ "c" со значением 3, а список имеет вид ("c", 3). Он присваивается списку ($key, $value), так что переменная $key содержит "c", а переменная $value содержит 3.

Но это списочное присваивание выполняется в условном выражении цикла while, имеющем скалярный контекст. (А точнее логический контекст со значением *true/false* как конкретная разновидность скалярного контекста.) Значение списочного присваивания в скалярном контексте равно количеству элементов в исходном списке, то есть в данном случае 2. Так как 2 интерпретируется как логическая истина, программа входит в тело цикла и выводит сообщение c = >3.

При следующей итерации цикла each %hash дает новую пару «ключ-значение»; предположим, на этот раз была получена пара ("a", 1). (Функция each возвращает пару, отличную от предыдущей, потому что она отслеживает текущую позицию; говоря на техническом жаргоне, вместе с каждым хешем хранится итератор[1].) Эти два элемента сохраняются в ($key, $value). Количество элементов в исходном списке снова равно 2, то есть истинному значению; раз условие while истинно, тело цикла снова выполняется, и программа выводит a => 1.

Мы снова входим в цикл. Теперь уже логика происходящего понятна, и вполне понятно, почему программа выводит сообщение b => 2.

Но перебор не может продолжаться вечно. Когда Perl в следующий раз обрабатывает each %hash, пар «ключ-значение» больше не осталось, по-

[1] Поскольку каждый хеш имеет собственный приватный итератор, допускается использование вложенных циклов с each при условии, что они перебирают *разные* хеши. А раз уж мы создали сноску, скажем: вряд ли вам когда-нибудь понадобится это делать, но вы можете сбросить текущую позицию итератора, вызвав для хеша функцию keys или values. Итератор также автоматически сбрасывается при сохранении в хеше нового списка или при переборе всех элементов до «конца» хеша функцией each. С другой стороны, включать в хеш новые пары «ключ-значение» во время его перебора обычно не стоит, так как это не гарантирует сброса итератора. Скорее всего, вы лишь запутаетесь сами, запутаете того, кто будет заниматься сопровождением программы, да и функцию each тоже.

этому each возвращает пустой список.[1] Пустой список присваивается ($key, $value), поэтому обеим переменным $key и $value присваивается undef.

Но это не столь существенно, потому что все вычисления производятся в условном выражении цикла while. Значение списочного присваивания в скалярном контексте определяется как количество элементов в исходном списке; в данном случае оно равно 0. Так как значение 0 интерпретируется как *false*, цикл while завершается, а выполнение продолжается в оставшейся части программы.

Конечно, each возвращает пары «ключ-значение» в заранее неизвестном порядке. (Кстати, это тот же порядок, в котором возвращают результаты функции keys и values; «естественный» порядок хеша.) Если вам нужно перебрать элементы хеша в определенном порядке, отсортируйте ключи; это делается примерно так:

```
foreach $key (sort keys %hash) {
  $value = $hash{$key};
  print "$key => $value\n";
  # Или без использования лишней переменной $value:
  #  print "$key => $hash{$key}\n";
}
```

Сортировка хешей более подробно рассматривается в главе 14.

Типичные операции с хешами

А теперь будет полезно рассмотреть конкретные примеры.

В библиотеке используется написанная на Perl программа, в которой среди прочего задействован хеш с количеством книг, выданным каждому читателю:

```
$books{"fred"} = 3;
$books{"wilma"} = 1;
```

Вы можете легко узнать, выдана ли читателю хотя бы одна книга; это делается так:

```
if ($books{$someone}) {
  print "$someone has at least one book checked out.\n";
}
```

Некоторые элементы хеша имеют ложное значение:

```
$books{"barney"} = 0;       # Книги в данный момент не выданы
$books{"pebbles"} = undef;  # Книги НИКОГДА не выдавались -
                            # новая библиотечная карточка
```

[1] Так как функция используется в списочном контексте, она не может вернуть undef для обозначения неудачи; вместо пустого (содержащего нуль элементов) списка () получится список из одного элемента (undef).

Читатель pebbles никогда не брал ни одной книги, поэтому в хеше для него хранится значение undef вместо 0.

Хеш содержит ключи для всех читателей, которым были выписаны библиотечные карточки. С каждым ключом (т. е. читателем) связывается либо количество выданных книг, либо undef, если карточка этого читателя никогда не использовалась.

Функция exists

Чтобы узнать, присутствует ли ключ в хеше (т. е. существует ли библиотечная карточка для данного читателя или нет), используйте функцию exists. Функция возвращает *true*, если ключ присутствует в хеше – независимо от того, истинно или нет связанное с ним значение:

```
if (exists $books{"dino"}) {
  print "Hey, there's a library card for dino!\n";
}
```

Иначе говоря, exists $books{"dino"} вернет *true* в том (и только в том) случае, если в списке ключей keys %books присутствует элемент dino.

Функция delete

Функция delete удаляет заданный ключ (вместе с соответствующим значением) из хеша. (Если ключ не существует, ничего не происходит – ни ошибки, ни предупреждения не выдаются.)

```
my $person = "betty";
delete $books{$person};  # Лишить читателя $person библиотечной карты
```

Удаление элемента отнюдь не равносильно сохранению в нем значения undef – как раз наоборот! Проверка exists($books{"betty"}) в этих двух случаях даст обратные результаты; после удаления ключ не может существовать в хеше, а после сохранения undef присутствие ключа обязательно.

В нашем примере различия между delete и сохранением undef можно сравнить с лишением библиотечной карточки и выдачей карточки, которая ни разу не использовалась.

Интерполяция элементов хеша

Одиночный элемент хеша интерполируется в строку, заключенную в кавычки, ровно так, как следовало ожидать:

```
foreach $person (sort keys %books) {              # Перебираем всех читателей
  if ($books{$person}) {
    print "$person has $books{$person} items\n";   # fred has 3 items
  }
}
```

Однако интерполяция всего хеша не поддерживается: Perl воспринимает запись "%books" как литерал, состоящий из шести символов (%books[1]). Следовательно, вы уже знаете все «волшебные» символы, которые должны экранироваться символом \ в строках, заключенных в кавычки: это символы $ и @, потому что они задают интерполируемую переменную; символ ", потому что без экранирования он будет воспринят как завершение строки в кавычках; и \, сама обратная косая черта. Все остальные символы в строках, заключенных в кавычки, не имеют специальной интерпретации и обозначают сами себя.[2]

Хеш %ENV

Ваши программы Perl, как и любые другие программы, работают в определенной *среде* (*environment*) и могут получать информацию о ней. Perl хранит эту информацию в хеше %ENV. Например, в %ENV обычно присутствует ключ PATH:

```
print "PATH is $ENV{PATH}\n";
```

В зависимости от конкретной конфигурации и операционной системы результат выглядит примерно так:

```
PATH is /usr/local/bin:/usr/bin:/sbin:/usr/sbin
```

Большинство переменных среды задается автоматически, но вы можете дополнять переменные среды своими данными. Конкретный способ зависит от операционной системы и командного процессора:

Bourne

```
$ CHARACTER=Fred; export CHARACTER
$ export CHARACTER=Fred
```

csh

```
% setenv CHARACTER=Fred
```

command (DOS или Windows)

```
C:> set CHARACTER=Fred
```

---

[1] Ничего другого и быть не могло: если бы мы попытались вывести весь хеш в виде пар «ключ-значение», он был бы практически бесполезен. И как было показано в предыдущей главе, знак % часто используется в форматных строках printf; наделять его дополнительным смыслом было бы крайне неудобно.

[2] Но помните об апострофах (`), левой квадратной скобке ([), левой фигурной скобке ({), маленькой стрелке (->) и парном двоеточии (::) – за именем переменной в строке в кавычках они могут означать что-то такое, чего вы не ожидали.

После того как вы зададите переменные среды внешними командами, вы сможете обращаться к ним в программе Perl:

```
print "CHARACTER is $ENV{CHARACTER}\n";
```

Упражнения

Ответы к упражнениям приведены в приложении А.

1. [7] Напишите программу, которая запрашивает у пользователя имя и выводит соответствующую фамилию из хеша. Используйте имена знакомых вам людей или (если вы проводите за компьютером столько времени, что не знаете живых людей) воспользуйтесь следующей таблицей:

Ввод	Вывод
fred	flintstone
barney	rubble
wilma	flintstone

2. [15] Напишите программу, которая читает серию слов (по одному слову в строке[1]), а затем выводит сводку с количеством вхождений каждого слова. (Подсказка: помните, что при использовании `undef` как числа Perl автоматически преобразует его в 0. Если понадобится, вернитесь к более раннему упражнению с накоплением суммы.) Так, на вход поступают слова `fred`, `barney`, `fred`, `dino`, `wilma`, `fred` (каждое слово в отдельной строке); на выходе программа должна сообщить, что `fred` встречается 3 раза. Чтобы задание стало более интересным, отсортируйте сводку по ASCII-кодам.
3. [15] Напишите программу для вывода всех ключей и значений в `%ENV`. Выведите результаты в два столбца в ASCII-алфавитном порядке. Отформатируйте результат так, чтобы данные в обоих столбцах выравнивались по вертикали. Функция `length` поможет вычислить ширину первого столбца. Когда программа заработает, попробуйте задать новые переменные среды и убедитесь в том, что они присутствуют в выходных данных.

[1] Одно слово в строке, потому что мы еще не показали, как извлекать отдельные слова из введенной строки.

7

В мире регулярных выражений

Perl обладает многочисленными возможностями, выделяющими его на фоне других языков. Среди них одно из важнейших мест занимает мощная поддержка регулярных выражений – быстрого, гибкого и надежного механизма работы со строками.

Но за широту возможностей приходится расплачиваться. Регулярные выражения фактически представляют собой мини-программы, написанные на отдельном языке, встроенном в Perl. (Да, вам придется изучить *еще один* язык программирования![1] К счастью, он несложен.) В этой главе мы посетим мир регулярных выражений, в котором (отчасти) забудем о мире Perl. А в следующей главе вы увидите, как этот мир связан с миром Perl.

Регулярные выражения поддерживаются не только в Perl; их также можно найти в *sed* и *awk*, *procmail*, *grep*, во многих текстовых редакторах для программистов (например, *vi* и *emacs*) и даже в более экзотических местах. Если ранее вы уже встречались с ними, считайте, вам повезло. Присмотревшись, вы найдете много других инструментов, использующих или поддерживающих регулярные выражения: поисковые системы в Web (часто написанные на Perl), почтовые клиенты и т. д. К сожалению, в разных реализациях используются слегка различающиеся диалекты регулярных выражений, поэтому вам придется запомнить, когда следует включать (или наоборот, опускать) обратную косую черту.

---

[1] Кто-то скажет, что регулярные выражения не являются *полноценным* языком программирования. Мы не будем спорить.

Что такое регулярные выражения?

Регулярное выражение представляет собой *шаблон*, который либо совпадает, либо не совпадает в заданной строке. Другими словами, существует бесконечное множество возможных текстовых строк; заданный шаблон делит это бесконечное множество на две группы: тех строк, в которых это регулярное выражение совпадает, и тех, в которых оно не совпадает. Никаких почти-полных-или-очень-похожих совпадений не бывает; либо совпадение есть, либо его нет.

Шаблон может совпадать в одной возможной строке, в двух или трех, в десятке или сотне или в бесконечном множестве строк. А может совпадать во всех строках, *кроме* одной, или кроме некоторых, или кроме бесконечного числа строк.[1] Мы уже говорили о регулярных выражениях как о мини-программах, написанных на своем простом языке программирования. Простота языка объясняется тем, что написанные на нем программы решают всего одну задачу – они просматривают строку и решают: «Да, совпадение есть» или «Нет, совпадения нет».[2] Вот и все.

Вполне возможно, что вы уже встречались с регулярными выражениями в команде UNIX *grep*, отбирающей текстовые строки по заданному критерию. Например, если вы хотите при помощи команды UNIX *grep* узнать, в каких строках заданного файла встречается слово `flint`, за которым в этой же строке следует слово `stone`, это делается примерно так:

```
$ grep 'flint.*stone' chapter*.txt
chapter3.txt:a piece of flint, a stone which may be used to start a fire by
striking
chapter3.txt:found obsidian, flint, granite, and small stones of basaltic
rock,
 which
chapter9.txt:a flintlock rifle in poor condition. The sandstone mantle held
several
```

Не путайте регулярные выражения с *глобами* – шаблонами командного процессора, используемыми при поиске файлов. Например, вводя типичный глоб `*.pm` в командном процессоре UNIX, вы найдете все файлы с расширением `.pm`. В предыдущем примере используется глоб `chapter*.txt`. (Возможно, вы заметили, что шаблон заключен в апострофы; это сделано для того, чтобы командный процессор не интерпре-

---

[1] Как вы увидите, можно определить шаблон, который совпадает всегда или не совпадает никогда. В редких случаях даже такие шаблоны могут быть полезны. Впрочем, чаще они свидетельствуют об ошибке программиста.

[2] Кроме того, программы передают Perl служебную информацию для последующего использования. В частности, к этой информации относится «память регулярных выражений», о которой будет рассказано позже.

тировал его как глоб.) Хотя в глобах используются те же символы, что и в регулярных выражениях, смысл у них совершенно иной.[1] В главе 13 мы еще вернемся к глобам, но пока постарайтесь забыть о них.

Простые регулярные выражения

Чтобы проверить, совпадает ли регулярное выражение (шаблон) в содержимом $_, просто заключите шаблон между парой символов /, как в следующем фрагменте:

```
$_ = "yabba dabba doo";
if (/abba/) {
  print "It matched!\n";
}
```

Выражение /abba/ ищет эту подстроку из четырех символов в $_; если строка будет найдена, возвращается *true*. В данном случае $_ содержит несколько вхождений этой строки, но это несущественно. Если совпадение найдено, поиск удачен; если найти строку не удалось, значит, совпадения нет.

Поскольку поиск по шаблону обычно используется для получения значения *true* или *false*, он почти всегда выполняется в условиях if и while.

В шаблонах поддерживаются все стандартные служебные комбинации с обратной косой чертой, известные по строкам в кавычках. Например, шаблон /coke\tsprite/ может использоваться для поиска последовательности из 11 символов: coke, символа табуляции и sprite.

О метасимволах

Конечно, если бы шаблоны могли совпадать только со строковыми литералами, пользы от них было бы немного. Специальные символы, называемые *метасимволами*, имеют особый смысл в регулярных выражениях.

Например, точка (.) обозначает один произвольный символ, кроме символа новой строки (который представляется последовательностью "\n"). Таким образом, шаблон /bet.y/ совпадет с betty, а также с betsy, bet=y, bet.y и вообще любой строкой, которая начинается с bet, содержит любой один символ (кроме символа новой строки) с последующим символом y. Однако с bety или betsey он не совпадет, потому что t и y

[1] Глобы (увы!) тоже иногда называются шаблонами. Что еще хуже, некоторые плохие книги по UNIX для новичков (и возможно, *написанные* новичками) доходят до того, что глобы называются «регулярными выражениями», каковыми они, конечно, не являются. Все это только сбивает людей с толку в самом начале работы с UNIX.

в этих строках не разделяются ровно одним символом. Точка всегда совпадает с одним символом, не более и не менее.

Итак, при поиске совпадения для литеральной точки в строке *можно* использовать точку. Но в этом случае совпадение будет найдено для любого возможного символа (кроме новой строки), а это может оказаться излишним. Если вы хотите, чтобы точка в шаблоне совпадала *только* с литеральной точкой, поставьте перед ней обратную косую черту. Собственно, это правило справедливо для всех метасимволов регулярных выражений Perl: обратная косая черта перед любым метасимволом отменяет его специальную интерпретацию. Таким образом, шаблон `/3\.14159/` не содержит специальных символов.

Итак, обратная косая черта – наш второй метасимвол. Если вам нужно включить в шаблон литеральный символ \, просто поставьте два таких символа подряд – это универсальное правило в Perl.

Простые квантификаторы

Часто возникает необходимость в повторении некоторых частей шаблона. Звездочка (`*`) означает нуль или более повторений предшествующего элемента. Таким образом, шаблон `/fred\t*barney/` совпадет при любом количестве символов табуляции между `fred` и `barney`: и `"fred\tbarney"` с одним символом табуляции, и `"fred\t\tbarney"` с двумя табуляциями, и `"fred\t\t\tbarney"` с тремя символами табуляции, и даже с `"fredbarney"` без табуляции. Дело в том, что `*` означает «нуль и более» – хоть пятьсот символов табуляции, но ничего, кроме них. Возможно, вам будет удобнее запомнить смысл `*` в формулировке «предыдущий элемент, повторенный сколько угодно раз, даже нуль» (по аналогии с оператором умножения, который в программировании тоже записывается знаком `*`).

А если в совпадение должны включаться другие символы кроме табуляции? Точка совпадает с любым символом[1], поэтому `.*` совпадет с любым символом, повторенным сколько угодно раз. Таким образом, шаблон `/fred.*barney/` совпадет с `fred` и `barney`, разделенными любым «мусором». Любая строка, в которой присутствуют строки `fred` и (где-то потом) `barney`, совпадет с этим шаблоном. Мы часто называем шаблон `.*` «мусором», потому что он совпадает с любой последовательностью символов в строке.

Звездочка обычно называется *квантификатором*, то есть определителем количества повторений предыдущего элемента. Но это не единст-

[1] Кроме символа новой строки. Но мы больше не будем постоянно напоминать вам об этом, потому что вы уже, наверное, запомнили. В большинстве случаев это несущественно, потому что строки, с которыми вам предстоит работать, обычно не содержат внутренних символов новой строки. Но не забывайте об этой подробности – вдруг когда-нибудь эти символы все же проникнут в строковые данные, с которыми вы работаете?

венный квантификатор; есть и другой – плюс (+), обозначающий *одно* или более повторений предшествующего элемента: `/fred +barney/` совпадает, если `fred` отделяется от `barney` пробелами (и ничем, кроме пробелов!). Сам пробел метасимволом не является. Совпадение в строке `fredbarney` найдено не будет, потому что плюс требует присутствия одного и более пробелов, так что хотя бы один пробел обязателен. Возможно, вам будет удобнее запомнить смысл + в формулировке «последний элемент, *плюс* (не обязательно) повторенный сколько угодно раз».

Также существует третий квантификатор, схожий с * и +, но более специализированный. Вопросительный знак (?) означает, что предшествующий элемент не является обязательным. Другими словами, он может встречаться один раз или не встречаться вовсе. Как и два других квантификатора, вопросительный знак указывает на вхождение предшествующего элемента некоторое количество раз. Просто в данном случае элемент может совпадать один раз (он есть) или нуль раз (его нет). Других возможностей нет. Таким образом, `/bamm-?bamm/` совпадает с одним из двух вариантов написания: `bamm-bamm` или `bammbamm`. Запомнить его несложно; вопросительный знак фактически говорит: «Предшествующий элемент, он есть? Или его нет?»

Все три квантификатора должны следовать за каким-либо элементом, так как все они определяют количество повторений *предшествующего* элемента.

Группировка в шаблонах

Как и в математических операциях, круглые скобки `()` используются для группировки элементов выражения. Таким образом, круглые скобки также являются метасимволами. Например, шаблон `/fred+/` совпадает с такими строками, как `freddddd`, но подобные строки редко встречаются на практике. С другой стороны, шаблон `/(fred)+/` совпадает в строках вида `fredfredfred`, а это уже больше похоже на то, что вам может понадобиться. А как насчет шаблона `/(fred)*/`? Как ни странно, он совпадет даже в такой строке, как `hello, world`.[1]

Круглые скобки также дают возможность повторно использовать часть строки, непосредственно предшествующую совпадению. Мы можем применять *обратные ссылки* для обращения к тексту, совпавшему с элементами выражения в круглых скобках. Обратная ссылка обозначается обратной косой чертой, за которой следует цифра: `\1`, `\2` и т. д. Цифра обозначает номер пары скобок.

[1] Звездочка означает *нуль* и более повторений `fred`. Если вы сами согласились на нуль, вам не на что жаловаться! Приведенный шаблон совпадет в любой строке, даже в пустой.

Точка в круглых скобках совпадает с любым символом, кроме символа новой строки. В дальнейшем символ, совпавший с точкой в круглых скобках, обозначается обратной ссылкой \1:

```
$_ = "abba";
if (/(.)\1/) {  # Совпадает с 'bb'
  print "It matched same character next to itself!\n";
}
```

Запись (.)\1 говорит, что выражение совпадает с символом, стоящим справа от самого себя. Сначала (.) совпадает с a, но при переходе к обратной ссылке, согласно которой дальше должен совпасть символ a, следующая проверка завершается неудачей. Perl начинает проверку заново, проверяя (.) на совпадение со следующим символом b. Теперь обратная ссылка \1 говорит, что в следующей позиции шаблона стоит символ b, который успешно совпадает.

Обратная ссылка не обязана находиться сразу же за группой в круглых скобках. Следующий шаблон совпадает с любыми четырьмя символами, отличными от символа новой строки, после литерала y; далее обратная ссылка \1 показывает, что те же четыре символа должны находиться после d:

```
$_ = "yabba dabba doo";
if (/y(....) d\1/) {
  print "It matched the same after y and d!\n";
}
```

Выражение может содержать несколько групп круглых скобок, при этом каждой группе соответствует собственная обратная ссылка. В следующем примере первая пара скобок совпадает с любым символом, кроме символа новой строки, за которым следует вторая пара скобок, также совпадающая с любым символом, кроме символа новой строки. После этих двух групп следует обратная ссылка \2, за которой следует обратная ссылка \1. Фактически в строке ищутся палиндромы (например, abba):

```
$_ = "yabba dabba doo";
if (/y(.)(.)\2\1/) { # Совпадает с 'abba'
  print "It matched the same after y and d!\n";
}
```

Возникает резонный вопрос: «Как определить, какой номер соответствует той или иной группе?» К счастью, Ларри выбрал способ, наиболее понятный для нас, людей: просто посчитайте открывающие круглые скобки, не обращая внимание на вложение:

```
$_ = "yabba dabba doo";
if (/y((.)(.)\3\2) d\1/) {
  print "It matched!\n";
}
```

Регулярное выражение даже можно записать так, чтобы в нем были четко видны разные части (хотя следующий фрагмент не является действительным регулярным выражением[1]):

```
(       # Первая открывающая скобка
    (.) # Вторая открывающая скобка
    (.) # Третья открывающая скобка
    \3
    \2
)
```

В Perl 5.10 появился новый способ обозначения обратных ссылок. Комбинация из обратной косой черты и номера заменяется конструкцией `\g{N}`, где `N` – номер обратной ссылки. Такая конструкция позволяет более четко выразить предполагаемую цель шаблона.

Представьте, что будет, если обратная ссылка должна использоваться рядом с числовой частью шаблона. В регулярном выражении запись `\1` обозначает повторение символа, совпавшего с предыдущей парой круглых скобок, а за ней следует литеральная строка `11`:

```
$_ = "aa11bb";
if (/(.)\111/) {
  print "It matched!\n";
}
```

Perl приходится гадать, что имелось в виду: `\1`, `\11` или `\111`? Perl может создать столько обратных ссылок, сколько потребуется, поэтому он считает, что мы имели в виду `\111`. А поскольку выражение не содержит 111 (и даже 11) пар круглых скобок, Perl сообщает об этом при попытке откомпилировать программу.

Конструкция `\g{1}` позволяет однозначно отделить обратную ссылку от литеральной части шаблона:[2]

```
use 5.010;

$_ = "aa11bb";
if (/(.)\g{1}11/) {
  print "It matched!\n";
}
```

Запись `\g{N}` также позволяет использовать отрицательные числа. Вместо абсолютного номера группы в круглых скобках можно указать *относительную обратную ссылку*. Перепишем последний пример так, чтобы та же задача решалась с использованием номера `-1`:

---

[1] Режим «свободной записи» в регулярных выражениях включается флагом `/x`, но он будет описан лишь в следующей главе.

[2] В общем случае фигурные скобки в `\g{1}` необязательны; можно просто использовать запись `\g1`, но в данном случае скобки необходимы. Чтобы не думать лишний раз, мы просто используем их всегда.

```
use 5.010;

$_ = "aa11bb";
if (/(.)\g{-1}11/) {
  print "It matched!\n";
}
```

Если позднее в шаблон добавятся новые группы, при абсолютной нумерации нам пришлось бы изменять обратные ссылки. Но при относительной нумерации отсчет ведется от текущей позиции, а ссылка указывает на непосредственно предшествующую ей группу независимо от ее абсолютного номера, поэтому нумерация остается неизменной:

```
use 5.010;

$_ = "aa11bb";
if (/(.)(.)\g{-1}11/) {
  print "It matched!\n";
}
```

Альтернатива

Вертикальная черта (`|`) означает, что совпасть может либо выражение в левой части, либо выражение в правой части. Иначе говоря, если часть шаблона слева от вертикальной черты не совпадает, то в совпадение может быть включена часть справа. Таким образом, выражение `/fred|barney|betty/` совпадет в любой строке, в которой встречается слово `fred`, `barney` или `betty`.

Теперь вы можете создать шаблон типа `/fred( |\t)+barney/`, который совпадает, если `fred` и `barney` разделяются пробелами, символами табуляции или любой их комбинацией. Знак `+` означает повторение один или более раз; при каждом повторении подвыражению `( |\t)` предоставляется возможность совпасть с пробелом или символом табуляции.[1] Два имени должны быть разделены, как минимум, одним из этих символов.

Если вы хотите, чтобы все символы между `fred` и `barney` были одинаковы, запишите шаблон в виде `/fred( +|\t+)barney/`. В этом случае все разделители должны быть одинаковыми – либо только пробелы, либо только символы табуляции.

Шаблон `/fred (and|or) barney/` совпадает в любой строке, содержащей любую из двух возможных: `fred and barney` или `fred or barney`.[2] Те же две строки можно было искать по шаблону `/fred and barney|fred or barney/`, но тогда шаблон получается слишком длинным. Вероятно, он

[1] Эта конкретная задача обычно с большей эффективностью реализуется при помощи символьных классов (см. далее в этой главе).

[2] Обратите внимание: слова `and` и `or` не являются операторами в регулярных выражениях! Они выведены моноширинным шрифтом, потому что здесь эти слова являются частью строк.

к тому же будет менее эффективным – в зависимости от того, какие оптимизации встроены в ядро поддержки регулярных выражений.

Символьные классы

Символьный класс, перечень возможных символов в квадратных скобках ([]), совпадает с одним из символов, входящих в класс. Он совпадает только с одним символом, но это может быть любой из перечисленных символов.

Например, символьный класс `[abcwxyz]` может совпасть с любым из семи символов. Для удобства предусмотрена возможность определения диапазонов через дефис (-), поэтому этот символьный класс также можно записать в формате `[a-cw-z]`. В данном примере особой экономии не получается, но на практике чаще встречаются символьные классы вида `[a-zA-Z]`, совпадающие с любой буквой из 52.[1] В символьных классах можно использовать те же определения символов, что и в строках, заключенных в кавычки; так, класс `[\000-\177]` обозначает любой семиразрядный ASCII-символ.[2] Конечно, символьные классы обычно являются частью шаблонов; в Perl они почти никогда не используются сами по себе. Например, фрагмент кода с символьным классом может выглядеть примерно так:

```
$_ = "The HAL-9000 requires authorization to continue.";
if (/HAL-[0-9]+/) {
  print "The string mentions some model of HAL computer.\n";
}
```

Иногда вместо символов, входящих в символьный класс, бывает проще перечислить исключаемые символы. «Крышка» (^) в начале символьного класса обозначает его инверсию. Так, класс `[^def]` совпадет с любым одиночным символом, *кроме* трех перечисленных. А класс `[^n\-z]` совпадет с любым символом, кроме `n`, дефиса и `z`. (Обратите внимание: дефис здесь экранируется обратной косой чертой, потому что в символьных классах он имеет особый смысл. С другой стороны, первый дефис в `/HAL-[0-9]+/` экранировать не нужно, потому что за пределами символьных классов дефис особой интерпретации не имеет.)

Сокращенная запись символьных классов

Некоторые символьные классы встречаются так часто, что для них были созданы специальные сокращения. Например, символьный класс для обозначения любой цифры `[0-9]` может быть записан в виде `\d`. Та-

[1] В эту группу из 52 букв не входят такие буквы, как Å, É, Î, Ø и Ü. Но при включенной поддержке Юникода этот конкретный диапазон распознается и автоматически расширяется так, чтобы он включал все буквы.

[2] Конечно, если вы используете кодировку ASCII, а не EBCDIC.

ким образом, приведенный ранее шаблон /HAL-[0-9]+/ можно записать в виде /HAL-\d+/.

Сокращение \w обозначает так называемый «символ слова»: [A-Za-z0-9_]. Если ваши «слова» состоят исключительно из обычных букв, цифр и символов подчеркивания, вам этого будет достаточно. У большинства разработчиков слова состоят из обычных букв, дефисов и апострофов, поэтому мы бы предпочли изменить это определение «слова». На момент написания книги разработчики Perl работали над этой проблемой, но решение пока не готово.[1] Следовательно, этот символьный класс должен использоваться только тогда, когда вам достаточно обычных букв, цифр и символов подчеркивания.

Конечно, \w не совпадает со «словом», а только с одним символом «слова». Но для совпадения с целым словом можно воспользоваться удобным модификатором +. Шаблон вида /fred \w+barney/ совпадет со словом fred, за которым следует пробел, затем некоторое «слово», еще один пробел и barney. Иначе говоря, он совпадет в том случае, если между fred и barney находится еще одно слово, отделенное от них одиночными пробелами.

Как вы, возможно, заметили из предыдущего примера, для определения пропусков между словами тоже хотелось бы иметь более гибкие средства. Сокращенная запись \s хорошо подходит для идентификации пропусков; она эквивалентна [\f\t\n\r]. Иначе говоря, это то же самое, что символьный класс из пяти символов: подачи страницы, табуляции, новой строки, возврата курсора и собственно пробела. Все эти символы всего лишь смещают текущую позицию печати; чернила или тонер при их выводе не используются. Но по аналогии с уже рассмотренными сокращениями \s совпадает только с одним символом из класса, поэтому для описания пропусков произвольной длины (в том числе и нулевой) обычно используется запись \s*, а для пропусков из одного и более символов – запись \s+. (Более того, \s почти не встречается без одного из указанных квантификаторов.) Для человека все пропуски внешне ничем не отличаются, а эта сокращенная запись позволяет работать с ними на общем уровне.

В Perl 5.10 появились новые символьные классы для определения пропусков. Сокращение \h совпадает только с горизонтальными пропусками (табуляцией и пробелом); оно эквивалентно символьному классу [\t]. Сокращение \v совпадает только с вертикальными пропусками, или [\f\n\r]. Сокращение \R совпадает с любыми разрывами строк; другими словами, вам не нужно думать о том, в какой операционной системе работает ваша программа и что в ней считается разрывом строки; \R во всем разберется за вас.

---

[1] Если не считать ограниченной (но все равно полезной) поддержки в локальных контекстах; см. man-страницу *perllocale*.

Инвертированные сокращения

В некоторых ситуациях бывает нужно придать любому из этих трех сокращений прямо противоположный смысл. Иначе говоря, вы хотите использовать `[^\d]`, `[^\w]` или `[^\s]`, подразумевая символ, *не являющийся* цифрой, символом слова или пропуском соответственно. Задача легко решается при помощи их «антиподов» из верхнего регистра: `\D`, `\W` и `\S`. Они совпадают с любым символом, с которым не совпадает основное сокращение.

Любое из этих сокращений может использоваться вместо символьного класса (занимая собственную позицию в шаблоне) или в квадратных скобках внутри большего символьного класса. Это означает, что запись `/[\dA-Fa-f]+/` может использоваться для поиска шестнадцатеричных чисел, у которых буквы ABCDEF (или их аналоги в нижнем регистре) играют роль дополнительных цифр.

Еще один составной символьный класс, `[\d\D]`, обозначает любую цифру или любую не-цифру... иначе говоря, любой символ! Это стандартный способ обозначения любого символа, в том числе и символа новой строки (в отличие от точки, совпадающей с любым символом, *кроме* новой строки). Наконец, стоит упомянуть совершенно бесполезную конструкцию `[^\d\D]`, которая совпадает с любым символом, не являющимся цифрой или не-цифрой. Да, все верно – ни с чем!

Упражнения

Ответы к упражнениям приведены в приложении А. (Если вас удивят возможности регулярных выражений, это нормально. Кстати, это одна из причин, по которой упражнения этой главы даже важнее упражнений из других глав. Приготовьтесь к неожиданностям.)

1. [10] Напишите программу, выводящую каждую строку входных данных, в которой присутствует слово `fred` (с другими строками ничего делать не нужно). Будет ли найдено совпадение во входной строке `Fred`, `frederick` или `Alfred`? Создайте небольшой текстовый файл, в котором упоминаются `fred flintstone` и его друзья.[1] Используйте его для передачи входных данных этой и другим программам этого раздела.
2. [6] Измените предыдущую программу так, чтобы совпадение также находилось в строках со словом `Fred`. Будут ли теперь найдены совпадения во входных строках `Fred`, `frederick` и `Alfred`? (Включите строки с этими именами в текстовый файл.)
3. [6] Напишите программу, которая выводит каждую строку входных данных, содержащую точку (`.`); остальные входные строки при

[1] Персонажи мультипликационного сериала «The Flintstones» – *Примеч. перев.*

этом игнорируются. Проверьте ее на текстовом файле из предыдущего упражнения; будет ли найдено совпадение в строке `Mr. Slate`?

4. [8] Напишите программу, которая выводит каждую строку со словом, содержащим буквы верхнего регистра (но не состоящим из них ПОЛНОСТЬЮ!) Будет ли она находить совпадение в строке `Fred`, но не в строках `fred` и `FRED`?
5. [8] Напишите программу, которая выводит каждую строку с двумя смежными одинаковыми символами, не являющимися символами пропусков. Программа должна находить совпадение в строках, содержащих слова вида `Mississippi`, `Bamm-Bamm` и `llama`.
6. [8] Упражнение «на повышенную оценку»: напишите программу для вывода входных строк, в которых присутствуют *оба* слова `wilma` и `fred`.

8

Поиск совпадений с использованием регулярных выражений

В предыдущей главе вы познакомились с регулярными выражениями. Теперь вы увидите, как они существуют в мире Perl.

Поиск совпадения оператором m//

В предыдущей главе шаблоны заключались в пару символов /, например `/fred/`. Но в действительности эта запись всего лишь является сокращенной формой оператора `m//`. Как было показано ранее для оператора `qw//`, для строкового квотирования содержимого могут использоваться и другие ограничители. Следовательно, то же выражение можно записать в виде `m(fred)`, `m<fred>`, `m{fred}` и `m[fred]` с парными ограничителями, или же в виде `m,fred,`, `m!fred!`, `m^fred^`, или с множеством других непарных ограничителей.[1]

Если в качестве ограничителя используется косая черта, начальное `m` можно не указывать. Большинство программистов Perl предпочитает обходиться без лишних символов, поэтому подавляющая часть операций поиска совпадения записывается в ограничителях `/.../` – как в примере `/fred/`.

[1] Непарными называются те ограничители, которые не имеют разных «левосторонней» и «правосторонней» формы; на обоих концах ставится один и тот же знак.

Конечно, ограничитель следует выбирать так, чтобы он не встречался в вашем шаблоне.[1] Например, шаблон, который совпадает с префиксом обычного URL-адреса, может выглядеть так: `/http:\/\//`. Но разумный выбор ограничителя упростит чтение, написание, сопровождение и отладку шаблона: `m%http://%`.[2] Также в качестве ограничителя часто используются фигурные скобки. В текстовых редакторах для программистов обычно предусмотрена функция перехода от открывающей фигурной скобки к парной закрывающей; она может быть удобна при сопровождении кода.

Модификаторы

Справа от завершающего ограничителя регулярного выражения могут добавляться буквенные модификаторы, также называемые *флагами*. Они изменяют поведение оператора поиска.

Поиск без учета регистра символов (/i)

Чтобы при поиске игнорировался регистр символов, а по одному условию легко находились строки `FRED`, `fred` и `Fred`, используйте модификатор `/i`:

```
print "Would you like to play a game? ";
chomp($_ = <STDIN>);
if (/yes/i) {  # Игнорировать регистр символов
  print "In that case, I recommend that you go bowling.\n";
}
```

Совпадение точки с любым символом (/s)

По умолчанию точка (`.`) не совпадает с символом новой строки; это вполне логично для большинства шаблонов, рассчитанных на поиск «в пределах одной логической строки». Если строковые данные могут содержать внутренние символы новой строки и вы хотите, чтобы точ-

---

[1] При использовании парных ограничителей обычно не приходится беспокоиться о возможном присутствии ограничителя в шаблоне, так как для него обычно найдется пара внутри шаблона. Скажем, шаблоны `m(fred(.*)barney)`, `m{\w{2,}}` и `m[wilma[\n\t]+betty]` нормально работают, несмотря на то, что шаблон содержит ограничитель; ведь для каждого «левостороннего» символа имеется «правосторонняя» пара. Но угловые скобки (`<` и `>`) не являются метасимволами регулярных выражений, поэтому они не могут использоваться парами. Скажем, если шаблон `m{(\d+)\s*>=?\s*(\d+)}` будет заключен в угловые скобки, внутренний знак `>` придется экранировать обратной косой чертой, чтобы он не был принят за преждевременное окончание шаблона.

[2] Не забывайте: косая черта не является метасимволом, поэтому ее не нужно экранировать, если она не используется в качестве ограничителя.

ка совпадала с этими символами, воспользуйтесь модификатором /s. С ним каждая точка[1] в шаблоне действует как символьный класс [\d\D], то есть совпадает с любым символом, включая символы новой строки. Конечно, поведение программы изменяется только в том случае, если данные содержат внутренние символы новой строки:

```
$_ = "I saw Barney\ndown at the bowling alley\nwith Fred\nlast night.\n";
if (/Barney.*Fred/s) {
  print "That string mentions Fred after Barney!\n";
}
```

Без модификатора /s поиск завершится неудачей, потому что два имени находятся в разных логических строках.

Добавление пропусков (/x)

Третий модификатор позволяет включать в шаблон произвольные пропуски, упрощающие его чтение:

```
/-?\d+\.?\d*/         # Что здесь творится?
/ -? \d+ \.? \d* /x   # Немного лучше
```

Так как модификатор /x позволяет включать пропуски в шаблон, литеральные пробелы или символы табуляции в шаблоне игнорируются. Для их идентификации можно использовать экранированный пробел или \t (среди прочего), но чаще для этого применяется сокращение \s (или \s*, или \s+).

Напомним, что в Perl комментарии могут включаться в пропуски. Любому, кто будет читать ваш шаблон, сразу станет ясно, что делает та или иная его часть:

```
/
  -?      # Необязательный минус
  \d+     # Одна или несколько цифр перед точкой
  \.?     # Необязательная точка
  \d*     # Необязательные цифры после точки
/x        # Конец строки
```

Поскольку знак # обозначает начало комментария, в тех редких случаях, когда вам нужно идентифицировать его в строке, следует использовать \# или [#]. Будьте внимательны и следите за тем, чтобы завершающий ограничитель не встречался в строке, так как это приведет к преждевременному завершению шаблона.

Объединение модификаторов

Если в одном шаблоне действует сразу несколько модификаторов, их можно перечислить друг за другом (в произвольном порядке):

---

[1] Если вы хотите ограничиться лишь небольшой частью точек в строке, вероятно, вам стоит заменить нужные точки комбинацией [\d\D].

```
if (/barney.*fred/is) {  # both /i and /s
  print "That string mentions Fred after Barney!\n";
}
```

Расширенная версия с комментариями:

```
if (m{
  barney # Имя
  .*     # Все промежуточные символы
  fred   # Второе имя
}six) {  # Три модификатора: /s, /i и /x
  print "That string mentions Fred after Barney!\n";
}
```

Обратите внимание на использование фигурных скобок в качестве ограничителей; они позволяют текстовым редакторам для программистов быстро перейти от начала регулярного выражения к его концу.

Другие модификаторы

Наряду с уже рассмотренными модификаторами существует ряд других модификаторов режимов. Мы рассмотрим их по мере изложения материала. Вы можете найти информацию в man-странице *perlop*, а также в описаниях `m//` и других операторов регулярных выражений, которые встретятся вам в оставшейся части этой главы.

Якоря

Если шаблон не совпадает в начале строки, по умолчанию ядро регулярных выражений Perl «перемещает» его по строке, пытаясь найти совпадение в другом месте. Однако выражение может содержать «якорные» символы, привязывающие шаблон к определенной позиции строки.

Якорь `^`[1] (циркумфлекс, «крышка») обозначает начало строки, а знак `$` совпадает только в конце строки[2]. Таким образом, шаблон `/^fred/` совпадет только от начала строки; в строке `manfred  mann` совпадение найдено не будет. А шаблон `/rock$/` совпадает только с символами `rock` в конце строки; в строке `knute rockne` совпадения не будет.

---

[1] Да, этот символ используется в шаблонах и для других целей. На первой позиции в символьном классе он инвертирует класс. Но *за пределами* символьного класса он интерпретируется иным образом – как якорь начала строки. Количество символов ограничено, поэтому некоторые приходится использовать дважды.

[2] Вообще говоря, он совпадает либо с концом строки, либо с символом новой строки в конце строки. Это позволяет успешно обнаруживать совпадение в конце строки как с завершающим символом новой строки, так и без него. Многие программисты не обращают внимания на это различие, но время от времени бывает полезно вспомнить, что `/^fred$/` совпадет как с `"fred"`, так и с `"fred\n"`.

В некоторых ситуациях используются оба якоря; они гарантируют, что шаблон займет всю строку. Типичный пример – шаблон `/^\s*$/`, совпадающий с пустой строкой. Впрочем, эта пустая строка может содержать пропуски (например, табуляции и пробелы), невидимые для нас с вами. Все строки, состоящие из таких символов, внешне не отличимы друг от друга, поэтому шаблон рассматривает их как эквивалентные. Без якорей он бы также совпадал и в непустых строках.

Границы слов

Якорная привязка не ограничивается концами строки. Якорь границы слова, `\b`, совпадает на любой границе слова.[1] Таким образом, `/\bfred\b/` совпадет со словом `fred`, но не совпадает с `frederick`, `alfred` или `manfred mann`. Происходящее немного напоминает режим поиска целых слов[2] в команде поиска текстовых редакторов.

Впрочем, это не те осмысленные слова, к которым мы с вами привыкли; здесь имеются в виду последовательности символов, состоящие из обычных букв, цифр и символов подчеркивания. Якорь `\b` совпадает с началом или концом группы символов `\w`.

На рис. 8.1 каждое «слово» подчеркнуто серой линией, а стрелки обозначают позиции, в которых совпадет `\b`. Количество границ слов в заданной строке всегда четно, поскольку у каждого начала слова имеется парный конец слова.

«Словами» считаются серии букв, цифр и символов подчеркивания; иначе говоря, словом считается последовательность, совпадающая с `/\w+/`. Показанное предложение содержит пять слов: `That`, `s`, `a`, `word` и `boundary`.[3] Обратите внимание: кавычки вокруг `word` не изменяют границ слов, так как последние состоят из символов `\w`.

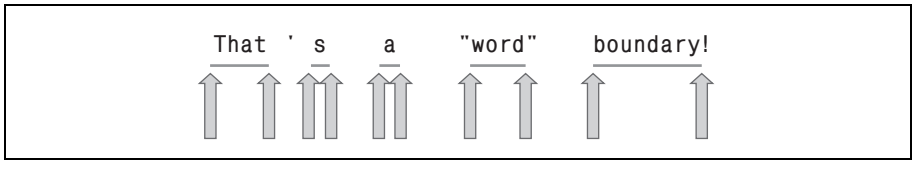


***Рис. 8.1.** Совпадения \b на границах слов*

---

[1] В некоторых реализациях регулярных выражений существуют разные якоря для начала и конца слова, но в Perl в обоих случаях используется `\b`.

[2] «Только слово целиком» в локализованной версии Microsoft Word. – *Примеч. перев.*

[3] Теперь вы видите, почему нам хотелось бы изменить определение «слова»: `That's` должно быть одним словом, а не двумя словами, разделенными апострофом. Даже в тексте, написанном на обычном английском языке, время от времени встречаются слова типа soupçon или другие необычные символы.

Каждая стрелка указывает либо в начало, либо в конец серой линии, так как якорь границы слова \b совпадает только в начале или в конце группы символов слова.

Якорь границы слова предотвратит случайное нахождение cat в delicatessen, dog в boondoggle или fish в selfishness. Иногда используется только один якорь границы слова, например шаблон /\bhunt/ найдет такие слова, как hunt, hunting или hunter, но не shunt, а шаблон /stone\b/ — такие слова, как sandstone или flintstone, но не capstones.

Инвертированный якорь границы слова \B совпадает в любой точке, где не совпадет \b. Таким образом, шаблон /\bsearch\B/ совпадет в строках searches, searching и searched, но не в search или researching.

Оператор привязки =~

Переменная $_ всего лишь используется по умолчанию; *оператор привязки* =~ приказывает Perl применить шаблон в правой части к строке в левой части (вместо переменной $_[1]). Пример:

```
my $some_other = "I dream of betty rubble.";
if ($some_other =~ /\brub/) {
  print "Aye, there's the rub.\n";
}
```

На первый взгляд оператор привязки может показаться некой разновидностью оператора присваивания. Ничего подобного! Он просто говорит: «Операция поиска совпадения, которая по умолчанию должна быть связана с $_, должна применяться к строке в левой части». Если оператор привязки отсутствует, по умолчанию выражение использует $_.

В (несколько необычном) примере, приведенном ниже, переменной $likes_perl присваивается логическое значение в зависимости от того, какие данные были введены пользователем. Программа немного отдает стилем программирования «на скорую руку», потому что сама строка ввода при этом теряется. Программа читает строку ввода, проверяет ее по шаблону и отбрасывает ее.[2] Переменная $_ не используется и не изменяется.

```
print "Do you like Perl? ";
my $likes_perl = (<STDIN> =~ /\byes\b/i);
...  # Проходит время...
if ($likes_perl) {
```

[1] Как вы узнаете позже, оператор привязки используется не только с операциями поиска совпадения по шаблону.

[2] Напомним, что строка ввода не сохраняется в $_ автоматически, если только все условие цикла while не состоит из оператора построчного ввода (<STDIN>).

```
    print "You said earlier that you like Perl, so...\n";
    ...
}
```

Так как оператор привязки обладает довольно высоким приоритетом, круглые скобки вокруг выражения проверки не обязательны, а следующая строка делает то же, что и приведенная выше, – сохраняет результат проверки (но не строку ввода) в переменной:

```
my $likes_perl = <STDIN> =~ /\byes\b/i;
```

Интерполяция в шаблонах

Регулярные выражения поддерживают интерполяцию по правилам строк в кавычках (как если бы они были заключены в кавычки). Это позволяет писать короткие программы «в стиле *grep*» следующим образом:

```
#!/usr/bin/perl -w
my $what = "larry";

while (<>) {
  if (/^($what)/) {  # Шаблон закрепляется в начале строки
    print "We saw $what in beginning of $_";
  }
}
```

При выполнении поиска шаблон будет построен из содержимого `$what`. В данном случае все происходит так же, как если бы мы воспользовались шаблоном `/^(larry)/` для поиска последовательности `larry` в начале каждой строки.

Но значение `$what` необязательно брать из строкового литерала; его можно было бы прочитать из аргументов командной строки в `@ARGV`:

```
my $what = shift @ARGV;
```

Если в первом аргументе командной строки передается текст `fred|barney`, шаблон превращается в `/^(fred|barney)/` и ищет текст `fred` или `barney` в начале каждой строки.[1] Круглые скобки (которые были лишними при поиске `larry`) здесь необходимы; без них шаблон будет искать `fred` в начале строки или `barney` в любой позиции.

Итак, шаблон получает данные из `@ARGV`, а сама программа начинает напоминать команду UNIX *grep*. Но мы должны принять особые меры предосторожности для метасимволов в строке. Если `$what` содержит

[1] Проницательный читатель знает, что текст `fred|barney` обычно нельзя ввести в аргументе командной строки, потому что вертикальная черта является метасимволом командного процессора. За информацией об экранировании аргументов командной строки обращайтесь к документации вашего командного процессора.

текст `'fred(barney'`, то в шаблоне появляется непарная скобка `/^(fred(barney)/`, а программа аварийно завершится с ошибкой регулярного выражения.

Используя сложные технические приемы[1], можно перехватить ошибки такого рода (или изначально предотвратить особую интерпретацию метасимволов) и устранить опасность сбоя программы. А пока запомните: если вы наделяете своих пользователей силой регулярных выражений, они несут ответственность за их правильное применение.

Переменные совпадения

До настоящего момента круглые скобки в шаблонах использовались только для группировки частей выражения. Однако в действительности круглые скобки также активизируют память ядра регулярных выражений. В памяти хранятся части строки, совпавшие с частью шаблона в круглых скобках. Если выражение содержит несколько пар скобок, будет сохранено несколько блоков. Каждая переменная в памяти регулярного выражения содержит часть *исходной строки*, а не часть шаблона.

Так как эти переменные предназначены для хранения строк, они являются скалярными переменными; в Perl им присваиваются имена вида `$1` и `$2`. Количество таких переменных определяется количеством пар сохраняющих круглых скобок в шаблоне. Как нетрудно догадаться, `$4` означает строку, совпавшую с четвертой парой круглых скобок.[2]

Эти переменные являются немаловажным аспектом мощи регулярных выражений, потому что они позволяют выделять отдельные части из совпавшей строки:

```
$_ = "Hello there, neighbor";
if (/\s(\w+),/) {             # Запомнить слово между пробелом и запятой
  print "the word was $1\n";  # the word was there
}
```

В одном выражении можно использовать сразу несколько переменных:

```
$_ = "Hello there, neighbor";
if (/(\S+) (\S+), (\S+)/) {
  print "words were $1 $2 $3\n";
}
```

---

[1] Либо посредством перехвата ошибки в блоке `eval`, либо экранированием интерполируемого текста функцией `quotemeta` (или ее эквивалентной формой `\Q`).

[2] И эта же строка будет задаваться обратной ссылкой `\4` при поиске совпадения. Но не стоит полагать, что один объект данных называется двумя разными именами; `\4` относится к содержимому памяти во время поиска, а `$4` – к содержимому памяти *уже завершенного* поиска. За дополнительной информацией об обратных ссылках обращайтесь к man-странице *perlre*.

Программа выводит перечень слов в исходной строке: `Hello there neighbor`. Обратите внимание на отсутствие запятой в выводе. Так как запятая не входит в сохраняющие круглые скобки в этом шаблоне, она не включается во вторую переменную. При составлении шаблона можно точно указать, что должно быть сохранено в переменных, а что следует оставить.

Переменные частичных совпадений могут быть пустыми[1], если для соответствующей части шаблона не нашлось совпадения. Иначе говоря, переменная может содержать пустую строку:

```
my $dino = "I fear that I'll be extinct after 1000 years.";
if ($dino =~ /(\d*) years/) {
  print "That said '$1' years.\n";  # 1000
}

$dino = "I fear that I'll be extinct after a few million years.";
if ($dino =~ /(\d*) years/) {
  print "That said '$1' years.\n";  # Пустая строка
}
```

Жизненный цикл переменных частичных совпадений

Переменные частичных совпадений обычно продолжают существовать до следующего *успешного* совпадения.[2] Другими словами, неудачный поиск сохраняет предыдущие значения переменных, а при успешном поиске они сбрасываются. Это подразумевает, что переменные частичных совпадений не могут использоваться без успешного поиска совпадения; в противном случае в них будет храниться информация, относящаяся к предыдущему шаблону. Следующий (неудачный) пример вроде бы должен выводить совпавшее слово из строки `$wilma`. Но если поиск завершится неудачей, переменная сохранит свое предыдущее значение:

```
$wilma =~ /(\w+)/;  # BAD! Untested match result
print "Wilma's word was $1... or was it?\n";
```

Это еще одна причина, по которой поиск по шаблону почти всегда размещается в условии `if` или `while`:

```
if ($wilma =~ /(\w+)/) {
  print "Wilma's word was $1.\n";
} else {
```

---

[1] В отличие от переменной с неопределенным значением `undef`. Если шаблон содержит три или более пар круглых скобок, переменная `$4` будет равна `undef`.

[2] Реальные правила видимости переменных более сложны (при необходимости обращайтесь к документации), но если вы не будете рассчитывать, что переменные сохранят свои значения в течение долгого срока, проблем обычно не будет.

```
  print "Wilma doesn't have a word.\n";
}
```

Так как переменные частичных совпадений не сохраняют свое значение навсегда, их не рекомендуется использовать далее чем в нескольких строках от команды поиска. Если во время сопровождения вашей программы между регулярным выражением и использованием `$1` будет случайно вставлено новое регулярное выражение, то вы получите значение `$1` для второго совпадения вместо первого. Если же текст частичного совпадения может понадобиться далее чем в нескольких строках от команды поиска, чаще всего он копируется в обычную переменную. Одновременно такое копирование сделает программу более понятной:

```
if ($wilma =~ /(\w+)/) {
  my $wilma_word = $1;
  ...
}
```

В главе 9 вы увидите, как сохранить текст частичного совпадения *прямо* в переменной при нахождении совпадения без явного использования переменной `$1`.

Несохраняющие круглые скобки

До настоящего момента мы рассматривали круглые скобки, которые «захватывали» части совпадения и сохраняли их в переменных в памяти. Но что, если круглые скобки должны использоваться только для группировки? Возьмем регулярное выражение с необязательной частью, в котором должна сохраняться другая часть. В следующем примере строка «bronto» является необязательной частью выражения, но чтобы сделать ее таковой, мы должны сгруппировать эту последовательность символов с круглых скобках. Позднее в этом шаблоне конструкция альтернативы выбирает между `steak` и `burger`, и мы хотим знать, какая из этих строк была обнаружена.

```
if (/(bronto)?saurus (steak|burger)/) {
    print "Fred wants a $2\n";
}
```

Даже если `bronto` отсутствует в строке, эта часть шаблона соответствует переменной `$1`. Напомним: чтобы решить, какой переменной соответствует та или иная часть шаблона, Perl просто считает открывающие круглые скобки. Нужная нам часть оказывается в переменной `$2`. В более сложных шаблонах ситуация оказывается еще более запутанной.

К счастью, в регулярных выражениях Perl предусмотрена возможность использования круглых скобок исключительно для группировки, без сохранения данных в памяти. Такие круглые скобки называются *несохраняющими*, а при их записи используется особый синтаксис.

После открывающей скобки ставится вопросительный знак и двоеточие (?:)[1]; тем самым вы сообщаете Perl, что круглые скобки будут использоваться только для группировки.

Изменим наше регулярное выражение и заключим «bronto» в несохраняющие круглые скобки; та часть совпадения, которую мы хотим сохранить, теперь будет находиться в $1:

```
if (/(?:bronto)?saurus (steak|burger)/) {
    print "Fred wants a $1\n";
}
```

Предположим, позднее регулярное выражение изменится, например в него добавится новая необязательная часть BBQ (с пробелом после Q). Этот элемент можно будет сделать необязательным и несохраняющим, а та часть совпадения, которую необходимо запомнить, по-прежнему будет храниться в $1. В противном случае нам пришлось бы сдвигать нумерацию всех переменных каждый раз, когда в регулярное выражение добавляются новые группирующие скобки:

```
if (/(?:bronto)?saurus (?:BBQ )?(steak|burger)/) {
    print "Fred wants a $1\n";
}
```

Регулярные выражения Perl поддерживают и другие специальные последовательности с круглыми скобками для выполнения сложных, нетривиальных операций: опережающей и ретроспективной проверки, встроенных комментариев и даже выполнения кода в середине шаблона. За подробностями обращайтесь к man-странице *perlre*.

Именованное сохранение

Вы можете сохранить части совпавшей строки при помощи круглых скобок, а затем использовать переменные $1, $2 и т. д. для работы с частичными совпадениями. Следить за этими нумерованными переменными и их содержимым непросто даже для элементарных шаблонов. Следующее регулярное выражение пытается найти два имени в строке $names:

```
use 5.010;

my $names = 'Fred or Barney';
if( $names =~ m/(\w+) and (\w+)/ ) { # Нет совпадения
    say "I saw $1 and $2";
    }
```

[1] Это четвертая разновидность ?, которую вы увидите в регулярных выражениях: литеральный вопросительный знак (экранированный), квантификатор 0 или 1, модификатор минимального поиска (см. далее), а теперь еще и начало несохраняющих круглых скобок.

Мы не видим сообщения от say, потому что в строке вместо and стоит or. Чтобы поиск мог производиться в обоих вариантах, мы изменяем регулярное выражение и включаем альтернативу для and и or; для группировки альтернативы используются круглые скобки:

```
use 5.010;

my $names = 'Fred or Barney';
if( $names =~ m/(\w+) (and|or) (\w+)/ ) { # Теперь совпадение есть
    say "I saw $1 and $2";
    }
```

Ой! Теперь сообщение выводится, но в нем нет второго имени, потому что в выражении появилась новая пара сохраняющих круглых скобок. В $2 хранится совпадение альтернативы, а второе имя оказалось в переменной $3 (которая не выводится):

```
I saw Fred and or
```

Проблему можно было бы обойти при помощи несохраняющих круглых скобок, но настоящая проблема в другом: мы должны помнить, какой номер круглых скобок соответствует каждым сохраняемым данным. Представьте, насколько усложнится задача при большом количестве сохранений.

Вместо того чтобы запоминать числа вида $1, Perl 5.10 позволяет нам задать имя сохраняемого частичного совпадения прямо в регулярном выражении. Совпавший текст сохраняется в хеше с именем %+: ключом является метка (то есть имя совпадения), а значением – совпавшая часть строки. Метка назначается конструкцией вида (?<МЕТКА>ШАБЛОН), где МЕТКА заменяется нужным именем.[1] Первому частичному совпадению присваивается метка name1, второму – name2, а для получения их значений используются конструкции $+{name1} и $+{name2}:

```
use 5.010;

my $names = 'Fred or Barney';
if( $names =~ m/(?<name1>\w+) (?:and|or) (?<name2>\w+)/ ) {
    say "I saw $+{name1} and $+{name2}";
    }
```

Теперь программа выводит правильное сообщение:

```
I saw Fred and Barney
```

Назначив метки сохраненным совпадениям, мы можем свободно перемещать их и добавлять сохраняющие круглые скобки без нарушения порядка:

---

[1] Perl также позволяет применять для этих целей синтаксис Python (?P<МЕТКА>...).

```
use 5.010;

my $names = 'Fred or Barney';
if( $names =~ m/((?<name2>\w+) (and|or) (?<name1>\w+))/ ) {
    say "I saw $+{name1} and $+{name2}";
    }
```

Итак, частичные совпадения помечены; теперь необходим способ обращения к ним в обратных ссылках. Ранее мы уже встречали обратные ссылки вида \1 или \g{1}. Для помеченных групп используется синтаксис вида \g{*метка*}:

```
use 5.010;

my $names = 'Fred Flinstone and Wilma Flinstone';

if( $names =~ m/(?<last_name>\w+) and \w+ \g{last_name}/ ) {
    say "I saw $+{last_name}";
    }
```

То же самое можно сделать и в другом синтаксисе. Вместо записи \g{*метка*} используется запись \k<*метка*>:[1]

```
use 5.010;

my $names = 'Fred Flinstone and Wilma Flinstone';

if( $names =~ m/(?<last_name>\w+) and \w+ \k<last_name>/ ) {

    say "I saw $+{last_name}";
    }
```

Автоматические переменные совпадения

Существуют еще три переменные поиска совпадений, которые «бесплатно»[2] предоставляются в ваше распоряжение независимо от того, содержит шаблон сохраняющие круглые скобки или нет. К сожалению, эти переменные имеют довольно странные имена.

Возможно, Ларри с радостью присвоил бы им какие-нибудь более привычные имена, скажем $gazoo или $ozmodiar. Но кто знает, не захочется ли какому-нибудь программисту использовать их в своем коде? Чтобы рядовому программисту Perl не приходилось запоминать имена *всех* специальных переменных Perl перед выбором имени первой перемен-

---

[1] Синтаксис \k<*метка*> немного отличается от \g{*метка*}. В шаблонах, содержащих две и более помеченные группы с одинаковыми метками, \k<*метка*> и \g{*метка*} всегда соответствуют крайней левой группе, а \\g{*N*} может быть относительной обратной ссылкой. Поклонники Python также могут использовать синтаксис (?P=*метка*).

[2] Да, конечно – ничего бесплатного в этой жизни не бывает. Эти переменные «бесплатны» только в том отношении, что они не требуют включения круглых скобок в шаблон. Позднее мы поговорим об их настоящей «цене».

ной в первой программе[1], Ларри присвоил многим встроенным переменным Perl странные имена, которые «нарушают правила». В данном случае имена состоят из знаков препинания: $&, $` и $'. Странно, уродливо, непривычно, но это их имена.[2] Часть строки, фактически совпавшая с шаблоном, автоматически сохраняется в $&:

```
if ("Hello there, neighbor" =~ /\s(\w+),/) {
  print "That actually matched '$&'.\n";
}
```

Программа сообщает, что шаблон совпал с текстом " there," (пробел, слово и запятая). Переменная $1 содержит только слово, а в переменной $& хранится все совпадение.

Часть исходной строки, предшествующая совпавшей части, хранится в переменной $`, а часть, которая идет после нее, в $'. Сформулировать можно и иначе: $` содержит текст, пропущенный ядром регулярных выражений перед обнаруженным совпадением, а $' — остаток строки, до которого шаблон так и не добрался. «Склеив» эти три строки, вы всегда получите исходную строку:

```
if ("Hello there, neighbor" =~ /\s(\w+),/) {
  print "That was ($`)($&)($').\n";
}
```

Формат строки в сообщении, (Hello)(there,)(neighbor), демонстрирует все три автоматические переменные поиска в действии.

Конечно, каждая из этих трех переменных может быть пустой (по аналогии с нумерованными переменными поиска). Все они обладают такой же областью видимости, что и нумерованные переменные. В общем случае это означает, что они сохраняют свои значения до следующего успешного совпадения.

Ранее мы назвали эти три переменные «бесплатными». Что ж, «бесплатное» тоже имеет свою цену. В данном случае цена такова: если хотя бы одна из автоматических переменных используется где-либо в программе, другие регулярные выражения будут работать чуть медленнее.[3] Замедление не катастрофическое, но даже его оказывается достаточно для того, чтобы многие программисты Perl никогда не поль-

---

[1] Некоторые «классические» имена переменных все равно необходимо избегать (например, $ARGV), но имена этих переменных записываются в верхнем регистре. Все встроенные переменные документированы в man-странице *perlvar*.

[2] Если вы решительно не переносите эти имена, обратитесь к модулю English, который пытается присвоить всем «странным» переменным почти нормальные имена. Однако этот модуль так и не прижился; программисты Perl привыкли к именам переменных из знаков препинания, как бы странно они ни выглядели.

[3] При каждом входе и выходе из блока, то есть практически везде.

зовались автоматическими переменными.[1] Вместо этого они отыскивают всевозможные обходные решения. Например, если вас интересует только значение `$&`, заключите весь шаблон в круглые скобки и используйте `$1` (конечно, это может потребовать перенумерации переменных).

Переменные поиска (как автоматические, так и нумерованные) чаще всего используются в операциях замены. Впрочем, об этом будет рассказано в следующей главе.

Общие квантификаторы

Квантификатор в шаблоне обозначает некоторое количество повторений предшествующего элемента. Мы уже видели три квантификатора `*`, `+` и `?`. Если ни один из них не подходит для ваших целей, используйте разделенные запятыми пары чисел в фигурных скобках (`{}`), определяющие минимальное и максимальное количество повторений.

Скажем, шаблон `/a{5,15}/` обозначает от 5 до 15 повторений буквы `a`. Если `a` повторяется три раза, этого недостаточно, и совпадение не обнаруживается. Если `a` повторяется пять раз, шаблон совпадет. При 10 повторениях оно по-прежнему остается. Если буква `a` повторяется 20 раз, совпадут только первые 15 вхождений, то есть верхний предел.

Если опустить второе число (но указать запятую), максимальное количество совпадений элемента не ограничивается. Таким образом, `/(fred){3,}/` совпадет с тремя и более повторениями `fred` подряд (не разделенными дополнительными символами вроде пробелов). Верхнего предела не существует; если строка содержит 88 повторений `fred`, шаблон совпадет с ними всеми.

Если опустить не только верхнюю границу, но и запятую, число в фигурных скобках будет определять точное количество повторений: `/\w{8}/` совпадет ровно с восемью символами слова (например, являющимися частью большей строки)... А `/,{5}chameleon/` совпадет с «запятая запятая запятая запятая запятая chameleon».

Вообще говоря, три квантификатора, которые мы рассматривали ранее, представляют собой обычные сокращения. Звездочка означает то же, что `{0,}` (т. е. нуль и более); плюс эквивалентен `{1,}` (один и более);

---

[1] Впрочем, большинство из них никогда не пыталось проводить хронометраж в своих программах, чтобы узнать, действительно ли их обходные решения экономят время; они просто избегают этих переменных, как отравы. Мы не виним их за это – многие программы, которые могли бы выиграть от применения этих трех переменных, занимают лишь несколько минут процессорного времени в неделю, так что их хронометраж и оптимизация были бы непроизводительной тратой времени. Но в таком случае стоит ли экономить несколько лишних миллисекунд? Кстати говоря, разработчики Perl работают над этой проблемой, но до выхода Perl 6 решение, вероятно, не появится.

наконец, вопросительный знак можно записать в виде {0,1}. На практике квантификаторы в фигурных скобках используются редко, поскольку трех сокращенных квантификаторов оказывается достаточно.

Приоритеты

Мы рассмотрели целый ряд метасимволов и служебных конструкций; но в каком порядке они должны обрабатываться в регулярных выражениях? Ниже приведена таблица приоритетов, которая показывает, какие части шаблонов наиболее плотно «прилегают» друг к другу. В отличие от таблицы приоритета операторов, таблица приоритетов элементов регулярных выражений проста; она состоит всего из четырех уровней. Приоритеты показаны в табл. 8.1.

Таблица 8.1. Приоритеты элементов регулярных выражений

Элемент	Пример
Круглые скобки (сохраняющие или группирующие)	(...), (?:...), (?<МЕТКА>...)
Квантификаторы	a* a+ a? a{n,m}
Якоря и последовательности	abc ^a a$
Альтернатива	a|b|c
Атомы	a [abc] \d \1

1. На верхнем уровне иерархии приоритетов находятся круглые скобки (()), используемые для группировки и сохранения. Все содержимое круглых скобок «прилегает друг к другу» плотнее, чем какие-либо внешние элементы.
2. На втором уровне находятся квантификаторы, то есть операторы повторения: звездочка (*), плюс (+) и вопросительный знак (?), а также квантификаторы в фигурных скобках вида {5,15}, {3,} и {5}. Они всегда «прилегают» к предшествующему элементу.
3. На третьем уровне иерархии находятся якоря и последовательности. К первой категории относятся: якорь начала строки (^), якорь конца строки ($), якорь границы слова \b и якорь «не-границы слова» \B. Последовательность (размещение одного элемента за другим) в действительности тоже является оператором, хотя и не использует специальные метасимволы. Это означает, что буквы в слове «прилегают» друг к другу так же плотно, как и якоря прилегают к буквам.
4. На нижнем уровне иерархии находится вертикальная черта (|), обозначающая альтернативу. Фактически альтернатива «нарезает» шаблон на части. Она обладает низшим приоритетом, потому что буквы в словах выражения /fred|barney/ «прилегают» друг к другу

плотнее, чем альтернативные части. Если бы альтернатива обладала более высоким приоритетом, этот шаблон означал бы поиск совпадения для `fre`, за которым следует буква `d` или `b`, с продолжением `arney`. Соответственно альтернатива находится в самом низу иерархии, а буквы имен рассматриваются как единое целое.

5. Далее идут так называемые атомы, составляющие самые элементарные части шаблонов: отдельные символы, символьные классы и обратные ссылки.

Примеры определения приоритетов

Если вам потребуется расшифровать сложное регулярное выражение, действуйте так, как действует Perl, и используйте таблицу приоритетов, чтобы разобраться в происходящем.

Например, выражение `/^fred|barney$/` вряд ли соответствует намерениям программиста. Дело в том, что вертикальная черта альтернативы находится на очень низком уровне приоритета; она разбивает шаблон надвое. Шаблон совпадает либо с `fred` в начале строки, либо с `barney` в конце строки. Скорее всего, программист имел в виду выражение `/^(fred|barney)$/`, которое совпадает со всей строкой, содержащей только слово `fred` или только слово `barney`.[1] А с чем совпадет шаблон `/(wilma|pebbles?)/` ? Вопросительный знак относится к предыдущему символу[2], так что шаблон совпадет с `wilma`, `pebbles` или `pebble` – вероятно, в составе большей строки (так как якоря отсутствуют).

Шаблон `/^(\w+)\s+(\w+)$/` совпадает со строкой, содержащей «слово», некоторые обязательные пропуски, а затем другое «слово» – и никаких дополнительных символов до или после. Например, с его помощью можно находить строки вида `fred  flintstone`. Круглые скобки, в которые заключены слова, не нужны для группировки; вероятно, они предназначаются для сохранения частичных совпадений в нумерованных переменных.

Пытаясь разобраться в сложном шаблоне, попробуйте добавить круглые скобки, чтобы прояснить приоритет элементов. Не забывайте, что группирующие скобки автоматически становятся сохраняющими; если вам нужно только сгруппировать элементы, используйте несохраняющие скобки.

Это не все

В этой главе рассмотрены все основные элементы регулярных выражений, обычно применяемые в повседневном программировании, но су-

---

[1] И возможно, завершающий символ новой строки, как упоминалось ранее при описании якоря `$`.

[2] Потому что связь квантификатора с буквой `s` сильнее связи `s` с другими буквами в `pebbles`.

ществуют и другие возможности. Некоторые из них описаны в «книге с альпакой»; дополнительную информацию о том, на что способны шаблоны Perl, можно найти в man-страницах *perlre*, *perlrequick* и *perlretut*.[1]

Тестовая программа

Когда в ходе программирования на Perl программисту требуется написать регулярное выражение, часто бывает трудно определить, что именно делает шаблон. Часто выясняется, что шаблон совпадает с большим или меньшим количеством символов, чем предполагалось. Также совпадение может найтись ранее, чем вы ожидали, или позднее, или не найтись вообще.

Следующая программа позволяет протестировать шаблон на нескольких строках и узнать, что и где совпадает:

```
#!/usr/bin/perl
while (<>) {                        # Чтение данных по одной строке
  chomp;
  if (/ВАШ ШАБЛОН/) {
    print "Matched: |$`<$&>$'|\n";  # Специальные переменные
  } else {
    print "No match: |$_|\n";
  }
}
```

Тестовая программа написана для программистов, а не для конечного пользователя; об этом легко догадаться хотя бы потому, что она не выводит никаких подсказок или информации об использовании. Программа получает любое количество входных строк и проверяет каждую строку по шаблону, который подставляется на место строки *ВАШ ШАБЛОН*. Для каждой строки, в которой будет найдено совпадение, три специальные переменные (`$``, `$&` и `$'`) показывают, где именно оно произошло. Так, для шаблона `/match/` с входными данными `beforematchafter` результат будет выглядеть так: «`|before<match>after|`». Угловые скобки показывают, с какой частью строки совпал шаблон. Если в совпадение включаются лишние символы, вы это немедленно увидите.

Упражнения

В некоторых упражнениях предлагается использовать тестовую программу из этой главы. Конечно, вы *можете* набрать ее вручную, следя

[1] Также ознакомьтесь с `YAPE::Regexp::Explain` — «переводчиком» регулярных выражений на английский язык.

за тем, чтобы все знаки препинания были введены без ошибок.[1] Но вероятно, программу вместе с другими примерами кода будет проще загрузить с сайта O'Reilly, упоминавшегося в предисловии. Эта программа хранится там под именем *pattern_test*.[2]

Ответы к упражнениям приведены в приложении А.

1. [8] Создайте в тестовой программе шаблон для строки `match`. Запустите программу для входной строки `beforematchafter`. Выводятся ли в результатах все три части строки в правильном порядке?
2. [7] Создайте в тестовой программе шаблон, совпадающий с любым словом (в смысле `\w`), завершающимся буквой `a`. Будет ли он совпадать с `wilma`, но не с `barney`? Совпадет ли он в строке `Mrs. Wilma Flintstone`? А как насчет строки `wilna&fred`? Опробуйте его на текстовом файле из упражнений предыдущей главы (и добавьте тестовые строки, если они не были добавлены ранее).
3. [5] Измените программу из предыдущего упражнения так, чтобы слово, завершающееся буквой `a`, сохранялось в переменной `$1`. Значение переменной должно выводиться в апострофах, например `$1 contains 'Wilma'`.
4. [5] Измените программу из предыдущего упражнения так, чтобы вместо `$1` в ней использовались именованные сохранения. Обновите программу, чтобы метка переменной выводилась в итоговом сообщении, например `'word' contains 'Wilma'`.
5. [5] Упражнение «на повышенную оценку»: измените программу из предыдущего упражнения так, чтобы сразу же за словом, завершающимся буквой `a`, до пяти следующих символов (если они есть в строке, конечно) сохранялись в отдельной переменной. Обновите приложение, чтобы оно отображало обе переменные. Например, если во входной строке говорится `I saw Wilma yesterday`, в переменной должны сохраняться символы « yest». Для входной строки `I, Wilma!` в дополнительной переменной сохраняется всего один символ. Будет ли обновленный шаблон совпадать с простой строкой `wilma`?
6. [5] Напишите новую программу (*не* тестовую!), которая выводит все входные строки, завершающиеся пропуском (кроме символов новой строки). Чтобы пропуск был виден при выводе, завершите выходную строку маркером.

---

[1] А если вы действительно будете вводить ее вручную, помните, что обратный апостроф (`) – совсем не то же самое, что обычный (').

[2] Не удивляйтесь, если загруженная вами программа будет немного отличаться от версии, приведенной в книге.

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`.)

10

Другие управляющие конструкции

В этой главе мы рассмотрим другие способы структурирования кода Perl. В основном эти средства не расширяют возможности языка, но упрощают вашу работу или помогают выполнить ее более удобно. Вы не обязаны использовать их в своем коде, но не поддавайтесь искушению пропустить эту главу – рано или поздно эти управляющие конструкции встретятся вам в коде, написанном другими программистами (более того, вы совершенно точно увидите их к тому моменту, когда закончите читать книгу).

Управляющая конструкция unless

В управляющей конструкции `if` блок кода выполняется только в том случае, если условие истинно. Если блок кода должен выполняться только при ложном условии, замените `if` на `unless`:

```
unless ($fred =~ /^[A-Z_]\w*$/i) {
  print "The value of \$fred doesn't look like a Perl identifier name.\n";
}
```

В сущности, конструкция `unless` эквивалентна проверке `if` с противоположным условием. Также можно сказать, что `unless` напоминает самостоятельную секцию `else`. Если вы встретили в программе конструкцию `unless`, смысл которой вам не понятен, перепишите ее (мысленно или реально) в виде проверки `if`:

```
if ($fred =~ /^[A-Z_]\w*$/i) {
  # Ничего не делать
} else {
  print "The value of \$fred doesn't look like a Perl identifier name.\n";
}
```

Такое действие не изменяет эффективности программы, которая должна компилировать в тот же внутренний байт-код. Также можно переписать условное выражение, используя оператор отрицания (!):

```
if ( ! ($fred =~ /^[A-Z_]\w*$/i) ) {
  print "The value of \$fred doesn't look like a Perl identifier name.\n";
}
```

Выберите тот стиль программирования, который покажется вам наиболее логичным, потому что он же, скорее всего, будет выглядеть наиболее логично при сопровождении вашего кода. Если вы считаете вариант с отрицанием более логичным, используйте его. И все же использование `unless` выглядит более естественно.

Секция else в unless

Конструкция `unless` даже может содержать необязательную секцию `else`. Этот синтаксис поддерживается, но он может привести к потенциальным недоразумениям:

```
unless ($mon =~ /^Feb/) {
  print "This month has at least thirty days.\n";
} else {
  print "Do you see what's going on here?\n";
}
```

Некоторые программисты используют этот синтаксис, особенно если первая секция очень коротка (например, состоит всего из одной строки), а вторая занимает несколько строк кода. Но мы предпочитаем либо использовать `if` с отрицанием условия, либо просто переставить секции для получения обычной конструкции `if`:

```
if ($mon =~ /^Feb/) {
  print "Do you see what's going on here?\n";
} else {
  print "This month has at least thirty days.\n";
}
```

Не забывайте, что код всегда пишется для двух читателей: компьютера, который будет его выполнять, и человека, который должен обеспечить его работоспособность. Если человек не поймет, что вы написали, вскоре ваша программа перестанет правильно работать.

Управляющая конструкция until

Иногда бывает удобнее выполнить цикл `while` с обратным условием. Для этого воспользуйтесь конструкцией `until`:

```
until ($j > $i) {
  $j *= 2;
}
```

Цикл выполняется до тех пор, пока выражение в условии не вернет *true*. Но в действительности перед нами всего лишь «замаскированный» цикл `while`; просто в цикле `until` цикл выполняется, пока условие остается ложным (вместо истинного, как в цикле `while`). Условие вычисляется перед первой итерацией, поэтому цикл `until`, как и `while`, выполняется нуль и более раз.[1] Как и в случаях с `if` и `unless`, любой цикл `until` можно преобразовать в эквивалентный цикл `while` простым отрицанием условия. Однако в общем случае проще и естественнее использовать `until` там, где это уместно.

Модификаторы выражений

Чтобы сделать запись более компактной, после выражения может следовать управляющий модификатор. Например, модификатор `if` работает по аналогии с блоком `if`:

```
print "$n is a negative number.\n" if $n < 0;
```

Результат получается точно таким же, как при использовании следующего кода (если не считать того, что мы сэкономили на вводе круглых и фигурных скобок):[2]

```
if ($n < 0) {
  print "$n is a negative number.\n";
}
```

Как мы уже говорили, программисты Perl не любят вводить лишние символы. К тому же более короткая форма хорошо читается на естественном языке: вывести сообщение, если значение `$n` меньше нуля.

Учтите, что условное выражение по-прежнему обрабатывается первым, хотя и записывается в конце. Здесь стандартный порядок обработки слева направо заменяется противоположным порядком. Чтобы понять код Perl, необходимо действовать так, как действует внутренний компилятор Perl, и прочитать команду до конца; только после этого вы поймете, что же в действительности происходит в программе.

Существуют и другие модификаторы:

```
&error("Invalid input") unless &valid($input);
$i *= 2 until $i > $j;
print " ", ($n += 2) while $n < 10;
&greet($_) foreach @person;
```

---

[1] Программисты Pascal, будьте внимательны: в Pascal цикл repeat-until всегда выполняется хотя бы один раз, а в Perl цикл `until` может не выполняться ни разу, если условное выражение истинно перед началом цикла.

[2] А также символа новой строки. Однако следует упомянуть, что форма с фигурными скобками создает новую область видимости. В тех редких случаях, когда вам потребуется полная информация по этой теме, обращайтесь к документации.

Вероятно, вы и сами уже догадались, как они работают. Каждый модификатор можно переписать по аналогии с модификатором `if` из предыдущего примера. Пример:

```
while ($n < 10) {
  print " ", ($n += 2);
}
```

Обратите внимание на выражение в круглых скобках в списке аргументов `print`: оно увеличивает `$n` на 2 и сохраняет результат в `$n`. Затем новое значение возвращается для вывода командой `print`.

Сокращенные формы читаются почти так же, как фразы на естественном языке: вызвать функцию `&greet` для каждого элемента `@person` в списке. Удваивать переменную `$i`, пока она не станет больше `$j`.[1] В одном из стандартных способов применения этих модификаторов используются команды следующего вида:

```
print "fred is '$fred', barney is '$barney'\n"          if $I_am_curious;
```

При подобной «инвертированной» записи наиболее важная часть команды записывается в начале. Главной целью команды является отслеживание значений переменных, а не проверка условия.[2] Некоторые программисты предпочитают записывать всю команду в одну строку, иногда с включением символов табуляции перед `if` для смещения к правому краю, как в нашем примере; другие размещают модификатор `if` в отдельной строке:

```
print "fred is '$fred', barney is '$barney'\n"
    if $I_am_curious;
```

Хотя любое выражение с модификатором можно переписать в виде блока («традиционная» запись), обратное не всегда справедливо. С каждой стороны модификатора может находиться только одно выражение. Следовательно, вы не можете использовать запись «что-то `if` что-то `while` что-то `until` что-то `unless` что-то `foreach` что-то»; все равно получится слишком сложно. Кроме того, несколько команд нельзя разместить слева от модификатора. Если с каждой стороны требуется нечто большее, чем простое выражение, запишите код в «традиционной» форме – с круглыми и фигурными скобками.

Как упоминалось ранее в связи с модификатором `if`, первым всегда вычисляется управляющее выражение (справа), как в традиционной форме. Модификатор `foreach` не позволяет выбрать другую управляющую переменную – всегда используется переменная `$_`. Обычно это не

---

[1] Во всяком случае *нам* удобнее так думать о них.

[2] Конечно, мы придумали имя `$I_am_curious`; эта переменная не является встроенной переменной Perl. Программисты, использующие этот прием, обычно называют свою переменную `$TRACING` или выбирают константу, объявленную директивой `constant`.

создает проблем, но если вы хотите использовать другую переменную, команду придется переписать в виде традиционного цикла foreach.

Простейший блок

Так называемый *простейший блок* (*naked block*) записывается без ключевых слов или условий. Допустим, в программе присутствует цикл while, который выглядит примерно так:

```
while (условие) {
  тело;
  тело;
  тело;
}
```

Убрав из него ключевое слово while и условное выражение, мы получаем простейший блок:

```
{
  тело;
  тело;
  тело;
}
```

Простейший блок напоминает цикл while или foreach, но без повторений; тело «цикла» выполняется ровно один раз, и на этом все заканчивается.

Одна из специфических особенностей простейшего блока заключается в том, что он создает область видимости (scope) для временных лексических переменных (впрочем в будущем вы увидите другие применения простейших блоков):

```
{
  print "Please enter a number: ";
  chomp(my $n = <STDIN>);
  my $root = sqrt $n;  # Вычисление квадратного корня
  print "The square root of $n is $root.\n";
}
```

Переменные $n и $root в этом блоке являются временными переменными, область видимости которых ограничивается блоком. В общем случае все переменные должны объявляться с минимально возможной областью видимости. Если вам нужна переменная для нескольких строк кода, поместите эти строки в простейший блок и объявите в нем переменную. Конечно, если значение $n или $root будет использоваться позднее, эти переменные необходимо объявить в большей области видимости.

Возможно, вы заметили функцию sqrt и поинтересовались, что это такое, – да, это функция, о которой мы раньше не упоминали. Perl содержит много встроенных функций, выходящих за рамки книги. За

дополнительной информацией об этих функциях обращайтесь к man-странице *perlfunc*.

Секция elsif

Время от времени требуется последовательно проверить несколько условий и узнать, какое из них истинно. Для этого можно воспользоваться секцией `elsif` конструкции `if`, как в следующем примере:

```
if ( ! defined $dino) {
  print "The value is undef.\n";
} elsif ($dino =~ /^-?\d+\.?$/) {
  print "The value is an integer.\n";
} elsif ($dino =~ /^-?\d*\.\d+$/) {
  print "The value is a _simple_ floating-point number.\n";
} elsif ($dino eq '') {
  print "The value is the empty string.\n";
} else {
  print "The value is the string '$dino'.\n";
}
```

Perl проверяет условные выражения одно за другим. Когда очередная проверка даст положительный результат, выполняется соответствующий блок кода, обработка всей управляющей конструкции на этом завершается[1] и управление передается остальной программе. Если все условия ложны, выполняется завершающий блок `else`. (Конечно, секция `else` не обязательна, но в данном случае ее стоит включить.)

Количество секций `elsif` не ограничено, но следует помнить, что Perl должен проверить первые 99 условий, прежде чем переходить к 100-му. Если конструкция содержит больше шести-семи `elsif`, подумайте, нельзя ли сделать то же самое более эффективно. В Perl FAQ (см. man-страницу *perlfaq*) приведены рекомендации по имитации команд `switch/case` других языков программирования. Пользователи Perl 5.10 и более поздних версий также могут использовать конструкцию `given-when`, описанную в главе 15.

Возможно, вы заметили, что ключевое слово `elsif` пишется только с одной буквой «e». Если записать его в виде `elseif`, Perl сообщит о неправильном написании. Почему? Потому что Ларри так решил.[2]

---

[1] В отличие от конструкций `switch` в языках семейства C, «сквозной переход» к следующему блоку не выполняется.

[2] Более того, он сопротивляется всем попыткам хотя бы разрешить альтернативное написание. «Если вы хотите писать `elsif` с двумя «e», это нетрудно. Шаг 1 – создайте собственный язык. Шаг 2 – сделайте его популярным». В своем языке программирования вы можете записывать ключевые слова так, как считаете нужным. Надеемся, что до создания `elseunless` дело все же не дойдет.

Автоинкремент и автодекремент

Значения скалярных переменных часто требуется увеличить или уменьшить на 1. Эти операции являются весьма типичными, поэтому для них были созданы сокращения (как и для многих других часто выполняемых операций). Оператор автоинкремента (++) увеличивает скалярную переменную на 1 по аналогии с тем же оператором в C и других языках:

```
my $bedrock = 42;
$bedrock++;  # Увеличить $bedrock на 1; теперь 43
```

Как и при других способах увеличения переменной на единицу, скалярная переменная создается в случае необходимости:

```
my @people = qw{ fred barney fred wilma dino barney fred pebbles };
my %count;                      # новый пустой хеш
$count{$_}++ foreach @people;   # новые ключи и значения создаются
                                # по мере необходимости
```

При первой итерации цикла `foreach` значение `$count{$_}` увеличивается. Таким образом, элемент `$count{"fred"}` переходит из состояния `undef` (так как он ранее не существовал в хеше) в состояние 1. При следующей итерации `$count{"barney"}` становится равным 1; затем `$count{"fred"}` увеличивается до 2. При каждой итерации один элемент `%count` увеличивается, а при необходимости создается. После завершения цикла значение `$count{"fred"}` равно 3. Это позволяет легко и быстро узнать, какие элементы входят в список и сколько раз повторяется каждый из них.

Аналогичный оператор автодекремента (`--`) уменьшает скалярную переменную на 1:

```
$bedrock--;  # уменьшить $bedrock на 1; снова получается 42
```

Значение автоинкремента

Выборка значения может быть совмещена с изменением переменной. Если поставить оператор ++ перед именем переменной, то переменная сначала увеличивается, а затем происходит выборка ее значения. Эта форма называется *префиксным инкрементом*:

```
my $m = 5;
my $n = ++$m;  # увеличить $m до 6, сохранить полученное значение в $n
```

Если поставить перед переменной оператор --, то переменная сначала уменьшится, а затем ее значение будет использовано в программе (*префиксный декремент*):

```
my $c = --$m;  # Уменьшить $m до 5, сохранить полученное значение в $c
```

А теперь хитрый момент: в другой разновидности синтаксиса имя переменной ставится в начале выражения (выборка), а за ним идет опе-

ратор инкремента или декремента. Эта форма называется *постфиксным инкрементом* или *постфиксным декрементом*:

```
my $d = $m++;  # В $d сохраняется старое значение (5), затем $m
               # увеличивается до 6
my $e = $m--;  # В $e сохраняется старое значение (6), затем $m
               # уменьшается до 5
```

Хитрость в том, что мы выполняем две операции одновременно. Выборка значения совмещается с его изменением в том же выражении. Если на первом месте стоит оператор, переменная сначала увеличивается (или уменьшается), а затем используется ее новое значение. Если на первом месте стоит переменная, выражение сначала возвращает ее (старое) значение, а потом увеличивает или уменьшает его. Также можно сказать, что эти операторы возвращают значение, но имеют побочный эффект в виде изменения значения переменной.

Если выражение записано само по себе[1], а его значение не используется (только побочный эффект), от размещения оператора до или после переменной ничего не изменится:[2]

```
$bedrock++;  # Увеличить $bedrock на 1
++$bedrock;  # То же самое; увеличить $bedrock на 1
```

Эти операторы часто используются при работе с хешами для идентификации встречавшихся ранее значений:

```
my @people = qw{ fred barney bamm-bamm wilma dino barney betty pebbles };
my %seen;

foreach (@people) {
  print "I've seen you somewhere before, $_!\n"
    if $seen{$_}++;
}
```

Когда `barney` встречается впервые, выражение `$seen{$_}++` ложно, так как оно равно `$seen{$_}`, то есть `$seen{"barney"}`, а значение этого элемента хеша равно `undef`. Однако у выражения имеется побочный эффект в виде увеличения `$seen{"barney"}`. Когда `barney` встретится в следующий раз, значение `$seen{"barney"}` будет истинным, поэтому сообщение будет выведено.

---

[1] То есть в пустом контексте.

[2] Программисты, разбирающиеся во внутренней реализации языков, могут предположить, что постфиксный инкремент и декремент уступают своим аналогам по эффективности, но в Perl это не так. Perl автоматически оптимизирует постфиксные формы в пустом контексте.

Управляющая конструкция for

Управляющая конструкция Perl `for` напоминает стандартные конструкции циклов из других языков, таких как C. Она выглядит так:

```
for (инициализация; проверка; приращение) {
  тело;
  тело;
}
```

Но Perl рассматривает эту конструкцию как замаскированный цикл `while` следующего вида:[1]

```
инициализация;
while (проверка) {
  тело;
  тело;
  приращение;
}
```

Несомненно, самым типичным применением циклов `for` являются счетные итеративные вычисления:

```
for ($i = 1; $i <= 10; $i++) {  # Отсчет от 1 до 10
  print "I can count to $i!\n";
}
```

Встретив такой заголовок, вы поймете суть происходящего в первой строке и без чтения комментария. Перед началом цикла управляющей переменной `$i` задается значение 1. Далее замаскированный цикл `while` выполняется, пока переменная `$i` остается меньшей либо равной 10. Между итерациями выполняется приращение управляющей переменной, которое в данном примере имеет форму инкремента; управляющая переменная (`$i`) увеличивается на 1.

Итак, при первом проходе цикла переменная `$i` равна 1. Так как она остается меньшей либо равной 10, программа выводит сообщение. Хотя приращение указано в начале цикла, логически оно выполняется в конце итерации, после вывода сообщения. Таким образом, переменная `$i` становится равной 2; это значение меньше либо равно 10, поэтому сообщение выводится снова, а переменная `$i` увеличивается до 3. Новое значение также меньше либо равно 10, и т. д.

Вскоре программа выведет сообщение о том, что она умеет считать до 9. Переменная `$i` увеличивается до 10. Новое значение по-прежнему меньше либо равно 10, поэтому цикл выполняется в последний раз, а программа сообщает, что она умеет считать до 10. Наконец, переменная `$i` увеличивается в последний раз до 11; это значение больше 10.

---

[1] На самом деле приращение выполняется в блоке `continue`, но эта тема выходит за рамки книги. За правдой обращайтесь к man-странице *perlsyn*.

Управление передается за пределы цикла, и выполнение основной программы продолжается.

Увидев заголовок цикла, любой опытный программист сразу скажет: «Ага, в этом цикле `$i` увеличивается от 1 до 10».

Обратите внимание: после завершения цикла управляющая переменная сохраняет «последнее» значение. В нашем примере она добралась до 11. Цикл `for` чрезвычайно гибок; с его помощью можно организовать весьма разнообразные вычисления. Например, переменная может уменьшаться от 10 до 1:

```
for ($i = 10; $i >= 1; $i--) {
  print "I can count down to $i\n";
}
```

А этот цикл считает от –150 до 1000 с приращением 3:[1]

```
for ($i = -150; $i <= 1000; $i += 3) {
  print "$i\n";
}
```

Более того, любая из трех управляющих частей (инициализация, проверка и приращение) может отсутствовать; впрочем заголовок все равно должен содержать две точки с запятой. В следующем (довольно необычном) примере проверка представляет собой операцию замены, а секция приращения пуста:

```
for ($_ = "bedrock"; s/(.)//; ) {      # Продолжать, пока s///
                                       # применяется успешно
  print "One character is: $1\n";
}
```

На месте условия (в подразумеваемом цикле `while`) находится операция замены, которая возвращает *true* в случае успеха. При первой итерации замена удалит букву `b` из `bedrock`. Каждая последующая итерация будет сопровождаться удалением очередного символа. Когда в строке не останется ни одного символа, очередная попытка замены завершится неудачей и цикл завершится.

Если выражение проверки (между двумя точками с запятой) пусто, оно автоматически считается истинным, что приводит к бесконечному циклу. Не создавайте бесконечные циклы, не предусмотрев возможности выхода из них. О том, как это делается, будет рассказано далее в этой главе.

```
for (;;) {
  print "It's an infinite loop!\n";
}
```

[1] Конечно, ровно до 1000 он никогда не досчитает. В последней итерации используется значение 999, так как значения `$i` кратны 3.

Если вам действительно понадобится бесконечный цикл[1], запишите его в форме `while`; этот способ больше соответствует стилю программирования на Perl:

```
while (1) {
  print "It's another infinite loop!\n";
}
```

Хотя программисту C первый способ покажется более знакомым, даже начинающему программисту Perl известно, что значение `1` всегда истинно. Он сразу поймет, что в программе намеренно создается бесконечный цикл, поэтому вторая запись предпочтительнее. Perl достаточно умен, чтобы распознать константное выражение и исключить его из цикла в ходе оптимизации, так что эффективность не пострадает.

Тайная связь между foreach и for

Оказывается, для парсера Perl ключевое слово `foreach` в точности эквивалентно ключевому слову `for`. Иначе говоря, каждый раз, когда Perl встречает одно из этих слов, он не отличает одно от другого. Чтобы узнать, что вы имели в виду, Perl заглядывает в круглые скобки. Если там находятся две точки с запятой, значит, это счетный цикл `for` (вроде тех, которые мы только что рассматривали). Если в скобках нет двух точек с запятой, значит, Perl имеет дело с циклом `foreach`:

```
for (1..10) {  # На самом деле цикл foreach от 1 до 10
  print "I can count to $_!\n";
}
```

В действительности это цикл `foreach`, но в записи использовано слово `for`. Во всех остальных примерах книги мы будем полностью записывать ключевое слово `foreach` там, где оно встречается. Но неужели вы думаете, что в реальном мире программисты Perl будут вводить четыре лишних символа?[2] Во всех программах, кроме кода новичков, все всегда пишут `for`. Чтобы понять, какой цикл используется в программе, вам придется действовать по примеру Perl и поискать символы «`;`».

В программах Perl «настоящий» цикл `foreach` почти всегда является предпочтительным. В цикле `foreach` (записанном с ключевым словом `for`) из предыдущего примера мы сразу видим, что цикл считает от 1 до 10. Но видите ли вы, в чем проблема счетного цикла, который пыта-

---

[1] Если в программе случайно образовался бесконечный цикл, попробуйте прервать ее выполнение клавишами Control-C. Возможно, вывод будет какое-то время продолжаться и после нажатия Control-C; это зависит от системы ввода/вывода и других факторов. Мы же вас предупреждали!

[2] Если вы так подумали, то вы невнимательно читали книгу. В среде программистов (и особенно программистов Perl) лень считается одной из классических добродетелей. Если не верите, спросите кого-нибудь на следующей встрече Perl Mongers.

ется сделать то же самое? Не заглядывайте в сноску, пока не будете уверены в том, что знаете правильный ответ:[1]

```
for ($i = 1; $i < 10; $i++) {  # Ошибка! Что-то здесь не так!
  print "I can count to $_!\n";
}
```

Управление циклом

Несомненно, вы уже поняли, что Perl относится к числу так называемых структурных языков программирования. В частности, в любой блок кода можно войти лишь через одну точку в начале блока. Но в некоторых случаях необходимы более широкие возможности контроля или большая гибкость, чем в уже рассмотренных нами примерах. Например, представьте, что вам нужно создать цикл, аналогичный `while`, но всегда выполняемый хотя бы один раз. А может быть, в какой-то ситуации нужно преждевременно выйти из блока. В Perl существуют три оператора управления циклами, которые используются в блоках циклов для выполнения подобных фокусов.

Оператор last

Оператор `last` немедленно прекращает выполнение цикла. (Если вы использовали оператор `break` в C или похожем языке, это его аналог.) В сущности, `last` открывает «аварийный выход» из блока цикла. Его выполнение завершает цикл. Пример:

```
# Вывести все входные строки с упоминанием fred вплоть до маркера __END__
while (<STDIN>) {
  if (/__END__/) {
    # После маркера входные данные отсутствуют
    last;
  } elsif (/fred/) {
    print;
  }
}
## Сюда last передает управление #
```

Когда во входных данных обнаруживается маркер `__END__`, цикл немедленно завершается. Конечно, комментарий в конце – это всего лишь

[1] Пример содержит две с половиной ошибки. Во-первых, в условии используется знак «меньше», поэтому цикл будет выполнен 9 раз вместо 10. В циклах такого рода часто встречается «ошибка смещения на 1»: представьте, что вам нужно построить 30-метровый забор со столбами через каждые 3 метра. Сколько столбов для этого потребуется? (10 – неправильный ответ). Во-вторых, в заголовке указана управляющая переменная `$i`, а в теле цикла используется `$_`. Наконец, такие циклы гораздо труднее читать, писать, сопровождать и отлаживать – это еще половина ошибки. Вот почему мы говорим, что в Perl обычно предпочтителен «истинный» цикл `foreach`.

комментарий; он ни в коем случае не является обязательным. Мы вставили его для того, чтобы вы лучше поняли суть происходящего.

В Perl существуют пять разновидностей блоков циклов: это блоки `for`, `foreach`, `while`, `until` и простейший блок.[1] Фигурные скобки блоков `if` и пользовательских функций[2] к ним не относятся.

Оператор `last` применяется к внутреннему блоку цикла, то есть блоку с максимальной вложенностью. Хотите знать, как выйти из внешних блоков? Оставайтесь с нами; вскоре мы расскажем об этом.

Оператор next

Иногда текущая итерация завершена, но прекращать выполнение всего цикла пока рано. Для таких случаев существует оператор `next`, осуществляющий переход к нижнему концу текущего блока цикла.[3] После `next` выполнение продолжается со следующей итерацией цикла (по аналогии с оператором `continue` в C или другом похожем языке):

```
# Анализ слов во входном файле или файлах
while (<>) {
  foreach (split) {  # Разбить $_ на слова, последовательно присвоить
                     # каждое слово $_
    $total++;
    next if /\W/;    # Для непонятных слов остаток итерации пропускается
    $valid++;
    $count{$_}++;    # Увеличить счетчик для каждого слова
    ## Сюда next передает управление ##
  }
}

print "total things = $total, valid words = $valid\n";
foreach $word (sort keys %count) {
  print "$word was seen $count{$word} times.\n";
}
```

Этот пример немного сложнее большинства примеров, встречавшихся ранее; давайте разберем его шаг за шагом. Цикл `while` последовательно

---

[1] Да, `last` может использоваться для выхода из простейшего блока.

[2] Вероятно, это не лучшая мысль, но операторы управления циклами могут использоваться внутри функций для управления циклами, *внешними* по отношению к функции. Иначе говоря, если функция вызывается в блоке цикла и внутри нее выполняется `last`, предполагается, что сама функция не содержит своего цикла, управление передается за пределы блока цикла *в основном коде*. Возможность управления циклами из функций может исчезнуть в будущих версиях Perl, и никто о ней не пожалеет.

[3] Здесь мы в очередной раз соврали. На самом деле `next` переходит к началу (обычно опускаемого) блока `continue`. За полной информацией обращайтесь к man-странице *perlsyn*.

читает входные строки из оператора <> в $_; мы уже видели это раньше. При каждой итерации в $_ оказывается очередная строка ввода.

Внутри цикла вложенный цикл foreach перебирает возвращаемое значение split. Помните используемую по умолчанию версию split без аргументов?[1] Она разбивает $_ по пропускам, то есть $_ фактически разбивается на список слов. Так как в цикле foreach управляющая переменная не указана, будет использоваться $_. Итак, в $_ последовательно заносится одно слово за другим.

Но разве мы не говорили, что в $_ последовательно заносятся строки входных данных? Да, во внешнем цикле это именно так. Но во вложенном цикле foreach в $_ последовательно заносятся слова. Повторное использование $_ для других целей не создает никаких проблем; такое происходит постоянно.

Итак, в цикле foreach в переменную $_ последовательно заносятся слова текущей строки входных данных. Переменная $total увеличивается; в ней должно храниться общее количество слов. Но следующая строка (главная в этом примере) проверяет, содержит ли слово «посторонние» символы – любые, кроме букв, цифр и символов подчеркивания. Таким образом, для слов вида Tom's или full-sized, а также слов с прилегающими запятыми, вопросительными знаками или другими нестандартными символами шаблон совпадает, оставшаяся часть итерации пропускается, а программа переходит к следующему слову.

Но, допустим, в переменную попало обычное слово, например fred. В этом случае $valid увеличивается на 1; то же происходит с $count{$_}, счетчиком вхождений слова. Таким образом, после завершения двух циклов будет подсчитано каждое слово в каждой строке входных данных из каждого файла, указанного пользователем.

Мы не будем объяснять последние несколько строк. Надеемся, к этому моменту вы уже разобрались, как это происходит.

Как и last, оператор next может использоваться в любых из пяти видов блоков циклов: for, foreach, while, until и простейшем блоке. Во вложенных блоках циклов next завершает итерацию внутреннего блока. О том, как изменить это поведение, вы узнаете в конце раздела.

Оператор redo

Триаду операторов управления циклами завершает оператор redo. Он возвращает управление в начало текущего блока цикла без проверки условия или перехода к следующей итерации. (Если вы программировали на C или другом похожем языке, не ищите аналоги. В этих языках похожего оператора нет.) Пример:

---

[1] Если не помните, не огорчайтесь. Не тратьте клетки мозга на запоминание того, что всегда можно посмотреть в *perldoc*.

```
# Проверка навыков печати
my @words = qw{ fred barney pebbles dino wilma betty };
my $errors = 0;

foreach (@words) {
  ## Сюда redo передает управление ##
  print "Type the word '$_': ";
  chomp(my $try = <STDIN>);
  if ($try ne $_) {
    print "Sorry - That's not right.\n\n";
    $errors++;
    redo;  # Возврат в начало цикла
  }
}
print "You've completed the test, with $errors errors.\n";
```

Как и два других оператора, `redo` работает с любой из пяти разновидностей блоков циклов, а во вложенных циклах его действие распространяется только на внутренний блок цикла.

Главное отличие между операторами `next` и `redo` заключается в том, что `next` перемещается к следующей итерации, а `redo` повторяет текущую итерацию. Следующий пример дает некоторое представление о том, как работают эти три оператора:[1]

```
foreach (1..10) {
  print "Iteration number $_.\n\n";
  print "Please choose: last, next, redo, or none of the above? ";
  chomp(my $choice = <STDIN>);
  print "\n";
  last if $choice =~ /last/i;
  next if $choice =~ /next/i;
  redo if $choice =~ /redo/i;
  print "That wasn't any of the choices... onward!\n\n";
}
print "That's all, folks!\n";
```

Если просто нажать Return без ввода каких-либо символов (попробуйте это сделать два-три раза), цикл просто продвигается к следующей итерации. Если выбрать `last` при достижении числа 4, цикл завершается, и число 5 уже не выводится. Если выбрать `next` на том же числе 4, переход к числу 5 происходит без вывода завершающего сообщения «...onward!». А если выбрать `redo` в той же фазе, итерация 4 будет выполнена заново.

[1] Если вы загрузили файлы примеров с сайта O'Reilly (так, как было описано в предисловии), эта программа называется *lnr-example*.

Метки блоков

Чтобы выполнить операцию не с внутренним, а каким-то другим блоком цикла, воспользуйтесь меткой. Метки в Perl напоминают другие идентификаторы – они тоже состоят из букв, цифр и символов подчеркивания, но не могут начинаться с цифр. С другой стороны, поскольку метки не имеют префиксного символа, их можно спутать с именами встроенных функций и даже с именами пользовательских функций. Скажем, называть метку `print` или `if` явно не стоит. По этой причине Ларри рекомендует записывать имена меток в верхнем регистре. Это не только гарантирует, что метка не будет конфликтовать с другими идентификаторами, но и упрощает ее поиск в коде. В любом случае метки создаются редко и встречаются лишь в небольшой части программ Perl.

Чтобы назначить метку блоку цикла, поставьте ее с двоеточием перед началом цикла. Затем внутри цикла используйте метку в операторах `last`, `next` или `redo` по мере необходимости:

```
LINE: while (<>) {
  foreach (split) {
    last LINE if /__END__/;  # Выход из цикла LINE
    ...
  }
}
```

Для удобства чтения метку обычно рекомендуется ставить у левого поля, даже если текущий код обладает отступом более высокого уровня. Обратите внимание: метка обозначает целый блок, а не целевую точку в коде.[1] В предыдущем фрагменте специальный маркер `__END__` отмечает конец всех входных данных. Встретив этот маркер, программа игнорирует все остальные строки данных (даже из других файлов).

Часто в качестве имени выбирается существительное.[2] Скажем, внешний цикл последовательно обрабатывает строки, поэтому мы назвали его `LINE`. Если бы имя присваивалось внутреннему циклу, его можно было бы назвать `WORD`, потому что в нем обрабатываются слова. Это позволяет создавать удобные и понятные конструкции вида `next WORD` или `redo LINE`.

---

[1] Это все же не `goto`.

[2] По крайней мере, такой выбор смотрится более логично, хотя Perl он совершенно неважен, даже если вы присвоите меткам имена типа XYZZY или PLUGH. (XYZZY и PLUGH – команды телепортации в культовой игре Colossal Cave. – *Примеч. перев.*) А впрочем, если вы не увлекались Colossal Cave в 70-е годы, вы вряд ли поймете, о чем идет речь.

Тернарный оператор ?:

Когда Ларри выбирал, какие операторы должны быть доступны в Perl, он не захотел обижать бывших программистов C, поэтому в Perl были перенесены все операторы C.[1] А это означало, что в Perl переносится и самый запутанный оператор C – тернарный оператор `?:`. Впрочем при всей запутанности он может быть весьма полезен.

Тернарный оператор напоминает проверку *if-then-else*, упакованную в одно выражение. Оператор называется «тернарным», потому что он получает три операнда. Оператор выглядит примерно так:

```
выражение ? выражение_для_true : выражение_для_false
```

Сначала Perl вычисляет первое выражение и определяет, истинно оно или ложно. При истинном результате используется второе подвыражение, а при ложном – третье. Каждый раз вычисляется только одно из двух «правых» выражений, а другое игнорируется. Иначе говоря, если первое выражение истинно, то вычисляется второе подвыражение, а третье игнорируется. Если первое выражение ложно, то второе выражение игнорируется, а третье вычисляется как общий результат.

В этом примере результат вызова функции `&is_weekend` определяет, какое выражение будет присвоено переменной:

```
my $location = &is_weekend($day) ? "home" : "work";
```

А здесь мы вычисляем и выводим среднее значение или строку-заполнитель из дефисов, если данные отсутствуют:

```
my $average = $n ? ($total/$n) : "-----";
print "Average: $average\n";
```

Любое использование оператора `?:` можно переписать в виде структуры `if`, но часто с потерей удобства и компактности:

```
my $average;
if ($n) {
  $average = $total / $n;
} else {
  $average = "-----";
}
print "Average: $average\n";
```

А вот полезный прием, который может использоваться для кодирования удобного разветвленного выбора:

```
my $size =
```

[1] Кроме операторов, бесполезных в Perl, например оператора, преобразующего число в адрес переменной в памяти. Также Ларри добавил несколько новых операторов (скажем, оператор конкатенации строки), чтобы программисты C завидовали.

```
($width < 10) ? "small"  :
($width < 20) ? "medium" :
($width < 50) ? "large"  :
                "extra-large"; # По умолчанию
```

В действительности конструкция состоит из трех вложенных операторов `?:`. Когда вы поймете суть происходящего, все становится вполне очевидно.

Конечно, вы не обязаны использовать тернарный оператор. Например, новичкам стоит держаться от него подальше. Но рано или поздно вы встретите тернарный оператор в чужом коде, и мы надеемся, что у вас появятся веские причины для того, чтобы использовать его в своих программах.

Логические операторы

Как и следовало ожидать, Perl содержит все необходимые логические операторы для работы с логическими (булевскими) значениями *true*/*false*. Например, логические условия часто объединяются логическими операторами AND (`&&`) и OR (`||`):

```
if ($dessert{'cake'} && $dessert{'ice cream'}) {
  # Оба условия истинны
  print "Hooray! Cake and ice cream!\n";
} elsif ($dessert{'cake'} || $dessert{'ice cream'}) {
  # По крайней мере, одно условие истинно
  print "That's still good...\n";
} else {
  # Оба условия ложны - ничего не делать
}
```

В некоторых ситуациях применяется ускоренное вычисление выражения. Если левая сторона логической операции AND ложна, то все выражение заведомо ложно, поскольку логический оператор AND возвращает истинное значение только в случае истинности обоих операндов. В нашем случае проверять правую сторону не нужно, поэтому она даже не обрабатывается. Что произойдет, если в приведенном примере переменная `$hour` равна 3:

```
if ( (9 <= $hour) && ($hour < 17) ) {
  print "Aren't you supposed to be at work...?\n";
}
```

Аналогично при истинности левой стороны логической операции OR правая сторона также не вычисляется. Допустим, в следующем выражении переменная `$name` содержит строку `fred`:

```
if ( ($name eq 'fred') || ($name eq 'barney') ) {
  print "You're my kind of guy!\n";
}
```

Из-за этой особенности такие операторы называют «ускоренными» (short-circuit). Там, где это возможно, они стараются придти к результату самым быстрым путем. Ускоренное вычисление результата логическими операторами часто используется в программах. Возьмем пример с вычислением среднего значения:

```
if ( ($n != 0) && ($total/$n < 5) ) {
  print "The average is below five.\n";
}
```

В этом примере правая сторона вычисляется только в том случае, если левая сторона истинна. Это предотвращает возможность случайного деления на нуль с аварийным завершением программы.

Значение ускоренного логического оператора

В отличие от C (и других похожих языков), значением ускоренного логического оператора является значение, полученное при обработке последней части, а не логическая величина. Формально результат получается эквивалентным: последняя вычисленная часть всегда истинна, если истинно все выражение, и всегда ложна, если ложно все выражение.

Однако такое возвращаемое значение намного полезнее. В частности, логический оператор OR весьма удобен для выбора значения по умолчанию:

```
my $last_name = $last_name{$someone} || '(No last name)';
```

Если значение `$someone` отсутствует в хеше, левая сторона равна `undef`, что интерпретируется как *false*. Значит, логическому оператору OR придется вычислять правую сторону, а полученный результат будет использоваться как значение по умолчанию. Однако в этой идиоме значение по умолчанию не просто заменяет `undef`; оно с таким же успехом заменит любое ложное значение. Проблема решается при помощи тернарного оператора:

```
my $last_name = defined $last_name{$someone} ?
    $last_name{$someone} : '(No last name)';
```

Однако запись получается слишком громоздкой, а `$last_name{$someone}` в нее включается дважды. В Perl 5.10 появился более удобный синтаксис для выполнения подобных операций; он описан в следующем разделе.

Оператор //

В предыдущем разделе мы показали, как присваивать значение по умолчанию при помощи оператора `||`. Но при этом игнорировался особый случай – ложные, но определенные (а следовательно, абсолютно

законные) значения. Далее было представлено более уродливое решение с тернарным оператором.

В Perl 5.10 подобные проблемы решаются при помощи оператора //, использующего ускоренное вычисление при обнаружении определенного значения независимо от истинности или ложности значения в левой части. Даже если значение `$last_name{$someone}` равно 0, эта версия все равно работает:

```
use 5.010;

my $last_name = $last_name{$someone} // '(No last name)';
```

Иногда требуется присвоить переменной значение, если она еще не была инициализирована, и оставить ее без изменений в противном случае. Допустим, программа должна выводить сообщения только при заданной переменной среды `VERBOSE`. Мы проверяем значение, связанное с ключом `VERBOSE` в хеше `%ENV`. Если значение отсутствует, оно задается программой:

```
use 5.010;

my $Verbose = $ENV{VERBOSE} // 1;
print "I can talk to you!\n" if $Verbose;
```

Следующая программа проверяет, как работает оператор //: она перебирает несколько значений и смотрит, какие из них будут заменены значением по умолчанию `default`:

```
use 5.010;

foreach my $try ( 0, undef, '0', 1, 25 ) {
    print "Trying [$try] ---> ";
    my $value = $try // 'default';
    say "\tgot [$value]";
    }
```

Результат показывает, что строка `default` используется только в одном случае: если переменная `$try` содержит `undef`:

```
Trying [0] --->     got [0]
Trying [] --->      got [default]
Trying [0] --->     got [0]
Trying [1] --->     got [1]
Trying [25] --->    got [25]
```

В некоторых ситуациях требуется задать значение, если оно не было задано ранее. Например, попытавшись вывести неопределенное значение при включенных предупреждениях, вы получите раздражающую ошибку:

```
use warnings;

my $name;  # Значение не указано, undef!
printf "%s", $name; # Use of uninitialized value in printf ...
```

Иногда эта ошибка безобидна, и ее можно просто проигнорировать. Но если вы ожидаете, что выводимое значение может оказаться неопределенным, его можно заменить пустой строкой:

```
use 5.010;
use warnings;

my $name;  # Значение отсутствует, undef!
printf "%s", $name // '';
```

Управляющие конструкции с операторами неполного вычисления

Все четыре оператора, представленные в предыдущем разделе, `&&`, `||`, `//` и `?:`, обладают одной особенностью: вычисление выражения зависит от значения в левой части. Иногда выражение вычисляется, иногда нет. По этой причине они называются *операторами неполного вычисления*. Все операторы неполного вычисления автоматически являются управляющими конструкциями.[1] Дело вовсе не в том, что Ларри горел желанием добавить в Perl побольше управляющих конструкций. Просто когда он решил включить эти операторы неполного вычисления, они автоматически стали управляющими конструкциями. В конце концов, все, что может привести к активизации и деактивизации фрагментов кода, по своей сути является управляющей конструкцией.

К счастью, это обстоятельство проявляется только при наличии у управляющего выражения побочных эффектов, например изменения значения переменной или вывода каких-либо данных. Допустим, вы встретили следующую строку кода:

```
($m < $n) && ($m = $n);
```

Мы сразу видим, что результат логической операции AND ничему не присваивается.[2] Почему?

Если значение `$m` действительно меньше `$n`, левая сторона истинна, поэтому будет вычислена правая сторона с выполнением присваивания. Но если переменная `$m` больше или равна `$n`, левая сторона будет ложной, а правая сторона игнорируется. Таким образом, эта строка кода фактически эквивалентна следующей, более понятной:

```
if ($m < $n) { $m = $n }
```

А может быть, вы занимаетесь сопровождением программы и видите в ней строку следующего вида:

```
($m > 10) || print "why is it not greater?\n";
```

---

[1] А вас не удивило, что логические операторы рассматриваются в этой главе?

[2] Конечно, оно может использоваться как возвращаемое значение (последнее выражение в пользовательской функции).

Если $m действительно больше 10, то левая сторона истинна и выполнение логического оператора OR завершено. В противном случае левая сторона ложна и программа переходит к выводу сообщения. Эту конструкцию тоже можно (и нужно) записать традиционным способом – вероятно, с if или unless.

Как правило, подобная запись управляющих конструкций чаще всего применяется бывшими программистами C или программистами Perl старой школы. Почему? Одни ошибочно полагают, что она эффективнее. Другим кажется, что от этого их код смотрится «круче». Третьи просто копируют то, что они увидели где-то еще.

Аналогичным образом тернарный оператор может использоваться для управления. В следующем примере $x присваивается меньшей из двух переменных:

```
($m < $n) ? ($m = $x) : ($n = $x);
```

Если $m меньше, $x присваивается ей, в противном случае $x присваивается переменной $n.

Существует еще одна разновидность записи логических операторов: их можно записывать словами and и or.[1] Эти операторы ведут себя точно так же, как записываемые знаками, но «словарные» формы находятся в нижней части таблицы приоритетов. Так как слова не «прилипают» к ближним частям выражения, с ними иногда удается обойтись меньшим количеством круглых скобок:

```
$m < $n and $m = $n;  # Но лучше записывать в эквивалентной форме if
```

А может оказаться и наоборот: понадобится *больше* круглых скобок. С приоритетами всякое бывает. Используйте круглые скобки, чтобы точно задать порядок выполнения операций, если вы не уверены в приоритетах. Но поскольку «словарные» формы обладают очень низким приоритетом, обычно бывает видно, что выражение делится на большие фрагменты. Сначала выполняется все, что находится слева, а затем (если понадобится) все, что находится справа.

Хотя использование логических операторов в качестве управляющих конструкций запутывает программу, иногда они задействуются в стандартных идиомах. Общепринятый способ открытия файла в Perl выглядит так:

```
open CHAPTER, $filename
  or die "Can't open '$filename': $!";
```

Используя низкоприоритетный оператор or с ускоренным вычислением, мы приказываем Perl: «Открой этот файл... или умри!» Если вызов open завершается успешно с возвращением истинного значения, вы-

[1] Также существует низкоприоритетный оператор not (аналог оператора логического отрицания !) и редкий оператор xor.

полнение `or` на этом заканчивается. Но в случае неудачи ложное значение заставляет `or` вычислить правую часть, что приводит к аварийному завершению программы с выдачей сообщения.

Таким образом, использование этих операторов в качестве управляющих структур является частью идиоматики Perl. Их правильное применение расширит возможности вашего кода, а неправильное затруднит его сопровождение. Не злоупотребляйте ими.[1]

Упражнения

Ответы к упражнениям приведены в приложении А.

1. [25] Напишите программу, которая многократно предлагает пользователю угадать число от 1 до 100, пока тот не введет правильное число. Число должно выбираться случайным образом по волшебной формуле `int(1+rand 100)`.[2] Если пользователь вводит неправильное число, программа должна выдавать подсказки типа «Too high» или «Too low». Если пользователь вводит слово `quit` или `exit`, или же пустую строку, программа завершает работу. Конечно, если пользователь угадал правильно, программа тоже должна завершаться!
2. [10] Измените программу из предыдущего упражнения так, чтобы во время работы она выводила отладочную информацию (например, загаданное число). Внесите изменения так, чтобы режим расширенного вывода можно было отключить, но при этом не выдавались предупреждения. Если вы работаете в Perl 5.10, используйте оператор //. В более ранних версиях используйте тернарный оператор.
3. [10] Измените программу из упражнения 3 в главе 6 (вывод переменных среды) так, чтобы для переменных, которым не задано значение, выводилась строка `(undefined value)`. Новые переменные среды можно устанавливать прямо в программе. Убедитесь в том, что программа выводит правильную информацию для переменных с ложным значением. Если вы работаете в Perl 5.10, используйте оператор //. В более ранних версиях используйте тернарный оператор.

---

[1] Использование этих экзотических форм (что угодно `or die`) чаще одного раза в месяц считается злоупотреблением.

[2] Если вас интересуют функции `int` и `rand`, обращайтесь к их описаниям в man-странице *perlfunc*.

11

Модули Perl

Perl обладает огромными возможностями, полное описание которых не уместить в одной книге. Множество людей делает на нем массу интересных вещей. Если вы решаете какую-то задачу, вполне возможно, что кто-то уже решил ее и опубликовал свое решение в архиве CPAN (Comprehensive Perl Archive Network) – всемирной сети из основных и зеркальных серверов, содержащей тысячи модулей с кодом Perl, пригодных для повторного использования.

Мы не собираемся учить вас писать модули; о том, как это делается, рассказано в «книге с альпакой». В этой главе мы только покажем, как использовать уже существующие модули.

Поиск модулей

Модули делятся на два типа: поставляемые вместе с Perl (а следовательно, уже доступные для вас) и те, которые приходится самостоятельно загружать из CPAN для установки. Если в тексте не указано обратное, все рассматриваемые ниже модули входят в поставку Perl.

Поиск модулей, не поставляемых с Perl, следует начинать с CPAN Search (*http://search.cpan.org*) или Kobes' Search (*http://kobesearch.cpan.org/*).[1] Просмотрите списки модулей по категориям или переходите к прямому поиску.

Оба ресурса чрезвычайно полезны, потому что они позволяют ознакомиться с описанием модуля без загрузки всего пакета. Также можно просмотреть поставку и заглянуть в файлы без хлопот с установкой модулей.

---

[1] Да, в URL-адресе должно быть две буквы «s», но никто не обращает на это внимания.

Прежде чем искать модуль, для начала проверьте, не установлен ли он в вашей системе. Например, для этого можно просто попытаться прочитать документацию командой *perldoc*. Модуль *CGI.pm* входит в поставку Perl (мы поговорим о нем позднее в этой главе), поэтому вы сможете прочитать его документацию:

```
$ perldoc CGI
```

При попытке сделать то же для отсутствующего модуля выдается сообщение об ошибке:

```
$ perldoc Llamas
$ No documentation found for "Llamas".
```

В вашей системе документация также может быть доступна в других форматах (например, в формате HTML). Если она, конечно, находится в положенном месте.[1]

Установка модулей

Чтобы установить модуль, отсутствующий в системе, иногда бывает достаточно загрузить дистрибутив, распаковать его и выполнить серию команд в командном процессоре. Поищите файл *README* или *INSTALL* с дополнительной информацией. Если модуль использует MakeMaker[2], команды будут выглядеть примерно так:

```
$ perl Makefile.PL
$ make install
```

Если модули не могут быть установлены в каталоги системного уровня, задайте другой каталог при помощи аргумента `PREFIX` файла *Makefile.PL*:

```
$ perl Makefile.PL PREFIX=/Users/fred/lib
```

Некоторые авторы модулей Perl используют для построения и установки своих творений модуль `Module::Build`. В этом случае последовательность выглядит примерно так:

```
$ perl Build.PL
$ ./Build install
```

Однако некоторые модули зависят от присутствия других модулей и не работают без их установки. Вместо того чтобы выполнять всю ра-

---

[1] Документация Perl рассматривается в книге «Intermediate Perl» (O'Reilly) («Perl: изучаем глубже», Символ-Плюс, 2007), а пока просто поверьте, что большая часть документации модулей находится в одном файле с исходным кодом.

[2] А на самом деле модуль Perl `ExtUtils::MakeMaker`, входящий в поставку Perl. Модуль делает все необходимое для создания файла с инструкциями по установке, соответствующими вашей системе и установке Perl.

боту самостоятельно, можно воспользоваться одним из модулей, входящих в поставку Perl, `CPAN.pm`.[1] Пользователь запускает модуль `CPAN.pm` в командной строке и вводит в нем необходимые команды:

```
$ perl -MCPAN -e shell
```

Но даже этот способ получается немного запутанным, поэтому недавно один из наших авторов написал маленький сценарий с именем «cpan», который тоже входит в поставку Perl и обычно устанавливается вместе с *perl* и его инструментарием. Запустите сценарий и передайте ему список устанавливаемых модулей:

```
$ cpan Module::CoreList LWP CGI::Prototype
```

Возможно, кто-то скажет: «Но у меня нет командной строки!» Если вы работаете с версией ActiveState (для Windows, Linux или Solaris), используйте PPM[2] (Perl Package Manager), который установит модули за вас. Версии ActiveState даже можно найти на CD и DVD.[3] Помимо того, что мы уже видели, в вашей операционной системе могут существовать свои способы установки программного обеспечения, включая модули Perl.

Использование простых модулей

Допустим, в вашей программе используется полное имя файла вида */usr/local/bin/perl*, а вы хотите выделить из него базовое имя. К счастью, это делается просто, потому что базовое имя состоит из всех символов за последним знаком / (в данном примере только *perl*):

```
my $name = "/usr/local/bin/perl";
(my $basename = $name) =~ s#.*/##;  # А вот и нет!
```

Как было показано ранее, сначала Perl выполняет присваивание в круглых скобках, а затем переходит к замене. Предполагается, что любая строка, завершаемая косой чертой (т. е. часть с именем каталога), заменяется пустой строкой, а в исходной строке остается только имя файла.

Такое решение вроде бы работает. Но только «вроде бы» – в действительности здесь скрываются три проблемы.

---

[1] Расширение «.pm» означает «Perl Module». Некоторые популярные модули снабжаются суффиксом «.pm», отличающим их от чего-то другого. В данном случае архив CPAN отличается от модуля CPAN, поэтому последний записывается в форме «`CPAN.pm`».

[2] *http://aspn.activestate.com/ASPN/docs/ActivePerl/faq/ActivePerl-faq2.html*

[3] Вы можете создать собственный диск CD или DVD, организовав локальный репозиторий. Хотя CPAN сейчас занимает почти 4 Гбайт, в версии «minicpan» (опять-таки написанной одним из авторов) архив сокращается до последних версий всех модулей суммарным объемом около 800 Мбайт. См. описание модуля `CPAN::Mini`.

Во-первых, имя файла или каталога в UNIX может содержать символ новой строки (случайно такие вещи обычно не происходят, но это не запрещено). Так как точка (.) в регулярных выражениях не совпадает с символом новой строки, для имени файла вида "/home/fred/flintstone\n/brontosaurus" такое решение работать не будет – программа решит, что базовым именем является строка "flintstone\n/brontosaurus". Проблему можно решить включением флага /s (если вы не забудете об этом неочевидном и редком особом случае); тогда подстановка приобретает следующий вид: s#.*/##s. Вторая проблема заключается в том, что это решение привязано к специфике UNIX. Оно предполагает, что в качестве разделителя каталога всегда используется косая черта, как в UNIX, а не обратная косая и не двоеточие, как в некоторых системах.

Но третья (и самая серьезная) проблема заключается в том, что вы пытаетесь решить уже решенную задачу. Perl поставляется с несколькими модулями, которые расширяют возможности Perl и дополняют его функциональность. А если их окажется недостаточно, в CPAN имеется множество других полезных модулей, причем каждую неделю их число растет. Вы (а еще лучше ваш системный администратор) можете самостоятельно установить эти модули, если вам потребуется их функциональность.

В оставшейся части этого раздела мы рассмотрим примеры использования некоторых функций пары простых модулей, входящих в поставку Perl. (Эти модули способны на большее; наш обзор всего лишь демонстрирует общие принципы использования простых модулей.)

К сожалению, мы не сможем рассказать все, что необходимо знать об использовании модулей вообще; для работы с некоторыми модулями необходимо разбираться в таких нетривиальных темах, как ссылки и объекты.[1] Эти темы, как и способы создания модулей, подробно рассматриваются в «книге с альпакой». Впрочем, знакомства с этим разделом должно быть достаточно для того, чтобы вы смогли использовать многие простые модули. Дополнительная информация о некоторых интересных и полезных модулях приведена в приложении B.

Модуль File::Basename

В предыдущем примере базовое имя файла определялось способом, который не был портируемым. Мы показали, что даже тривиальное решение может быть подвержено неочевидным ошибочным предположениям (в данном случае предположение об отсутствии символов новой строки в именах файлов и каталогов). И вообще не стоило «изобретать велосипед заново», решая проблему, которая уже была многократно решена (и отлажена) до нас.

---

[1] Впрочем, как вы вскоре увидите, модули с объектами и ссылками иногда удается использовать и без понимания этих нетривиальных тем.

Итак, извлечь базовое имя из полного имени файла можно другим, более правильным способом. В поставку Perl включен модуль `File::Basename`. О том, что он делает, можно узнать при помощи команды *perldoc File::Basename* или прочитать в документации к вашей системе. Это первый шаг по использованию нового модуля (а нередко также третий и пятый).

Когда вы будете готовы использовать модуль в своей программе, объявите его директивой `use` в начале программы:[1]

```
use File::Basename
```

В процессе компиляции Perl обнаруживает эту строку и загружает модуль. Теперь с точки зрения Perl все выглядит так, словно в программе появились новые функции, которые могут в ней использоваться.[2] В нашем примере потребуется основная функция модуля `basename`:

```
my $name = "/usr/local/bin/perl";
my $basename = basename $name;       # Получаем 'perl'
```

В UNIX такое решение работает. А если программа будет запущена в MacPerl, в Windows, в VMS или в другой системе? Нет проблем – модуль определяет, на каком компьютере он выполняется, и использует стандартные правила для имен файлов этого компьютера. (Конечно, в переменной `$name` должно храниться значение, соответствующее файловой системе этого компьютера.)

Модуль также предоставляет ряд сопутствующих функций. Одна из них – функция `dirname` – извлекает из полного имени часть с именем каталога. Модуль также позволяет отделить имя файла от расширения или изменить стандартный набор правил для имен файлов.[3]

Использование отдельных функций модуля

Предположим, при включении в существующую программу модуля `File::Basename` обнаружилось, что программа уже содержит пользовательскую функцию с именем `&dirname` (то есть имя существующей функции совпадает с именем одной из функций модуля). И здесь начи-

---

[1] Модули традиционно объявляются в начале файла, потому что программисту, занимающемуся сопровождением программы, будет проще узнать, какие модули в ней используются. Например, это значительно упрощает установку программы на новый компьютер.

[2] Да, вы угадали: на самом деле существуют другие проблемы, относящиеся к пакетам и полностью уточненным именам. Если объем основного кода вашей программы превысит несколько сотен строк (не считая кода модулей), что довольно много для программ Perl, вам, вероятно, стоит ознакомиться с этими расширенными возможностями. Начните с man-страницы *perlmod*.

[3] Когда может потребоваться эта возможность? Например, при работе с файлами из UNIX на компьютере с системой Windows – скажем, при передаче команд через подключение FTP.

наются проблемы, потому что новая функция dirname *тоже* реализована в виде пользовательской функции Perl (внутри модуля). Что делать?

Включите в объявлении use модуля File::Basename *список импорта*, в котором точно перечислены имена импортируемых функций. Директива импортирует только эти имена и никакие другие. Например, при следующем объявлении в программу из модуля включается только функция basename:

```
use File::Basename qw/ basename /;
```

А здесь никакие новые функции вообще не запрашиваются:

```
use File::Basename qw/ /;
```

Эта форма также часто записывается в следующем виде:

```
use File::Basename ();
```

Зачем это нужно? Эта директива приказывает Perl загрузить модуль File::Basename, как и прежде, но без импортирования имен функций. Импортирование позволяет нам использовать короткие, простые имена типа basename и dirname. Даже если не импортировать имена, вы все равно сможете использовать функции, но для этого придется вызывать их с указанием полных имен:

```
use File::Basename qw/ /;              # Не импортировать имена функций

my $betty = &dirname($wilma);          # Используем собственную
                                       # функцию &dirname (код не показан)

my $name = "/usr/local/bin/perl";
my $dirname = File::Basename::dirname $name;  # dirname из модуля
```

В последней строке функция dirname из модуля File::Basename вызывается по полному имени File::Basename::dirname. Полное имя функции можно использовать всегда (после загрузки модуля, естественно) независимо от того, было импортировано короткое имя dirname или нет.

Чаще всего используется список импорта по умолчанию, однако вы всегда можете заменить его своим собственным списком, чтобы отменить часть импорта по умолчанию. Другая возможная причина для определения собственного списка импорта – импортирование функций, не входящих в список по умолчанию. Некоторые модули определяют (редко используемые) функции, не включаемые в список импорта по умолчанию.

Как нетрудно догадаться, некоторые модули по умолчанию импортируют больше символических имен по сравнению с другими. В каждой документации модуля должно быть четко указано, какие символические имена он импортирует (если они есть), но вы всегда можете переопределить список импорта по умолчанию, указав свой собственный список, как это было сделано для File::Basename. При передаче пустого списка символические имена не импортируются.

Модуль File::Spec

Теперь вы умеете определять базовое имя файла. Это полезно, но довольно часто возникает обратная задача: объединить его с именем каталога для получения полного имени файла. Например, в следующем фрагменте требуется прибавить префикс к базовому имени файла в полном имени вида /home/rootbeer/ice-2.1.txt:

```
use File::Basename;

print "Please enter a filename: ";
chomp(my $old_name = <STDIN>);

my $dirname = dirname $old_name;
my $basename = basename $old_name;

$basename =~ s/^/not/;    # Добавление префикса к базовому имени
my $new_name = "$dirname/$basename";

rename($old_name, $new_name)
  or warn "Can't rename '$old_name' to '$new_name': $!";
```

Видите, в чем проблема? Мы снова неявно предполагаем, что имена файлов соответствуют правилам UNIX, и используем косую черту для отделения имени каталога от базового имени. К счастью, Perl содержит готовый модуль для выполнения подобных операций.

Модуль File::Spec предназначен для манипуляций с файловыми спецификациями, которые определяют имена файлов, каталогов и других объектов файловой системы. Как и File::Basename, этот модуль распознает файловую систему, в которой он работает, и автоматически выбирает нужный набор правил. Но, в отличие от File::Basename, модуль File::Spec является объектно-ориентированным (часто сокращается до OO) модулем.

Если вы еще не заразились лихорадкой объектно-ориентированного программирования, пусть вас это не беспокоит. Разбираетесь в объектах – хорошо; вы сможете использовать OO-модули. Не разбираетесь в объектах – тоже не страшно. Просто введите символические имена так, как мы вам покажем, и все будет прекрасно работать.

Из документации File::Spec мы узнаем, что для решения нашей задачи необходимо использовать *метод* с именем catfile. Что такое метод? Просто разновидность функции (в том, что касается наших практических целей). Отличительная особенность методов заключается в том, что методы File::Spec всегда вызываются по полным именам:

```
use File::Spec;

.
.  # Получить значения $dirname и $basename, как было сделано выше
.

my $new_name = File::Spec->catfile($dirname, $basename);
```

```
rename($old_name, $new_name)
  or warn "Can't rename '$old_name' to '$new_name': $!";
```

Полное имя метода образуется из имени модуля (в этом контексте он называется *классом*), маленькой стрелки (->) и короткого имени метода. Очень важно использовать в записи именно маленькую стрелку, а не двойное двоеточие.

Так как метод вызывается по полному имени, какие символические имена импортирует этот модуль? Никаких. Для ОО-модулей это нормально, поэтому вам не придется беспокоиться о возможных конфликтах между именами пользовательской функции и многочисленных методов `File::Spec`.

Стоит ли использовать такие модули? Как обычно, решайте сами. Если вы уверены, что ваша программа всегда будет запускаться только на компьютерах с системой UNIX, а также в том, что хорошо знаете правила выбора имен файлов в UNIX[1], возможно, вы предпочтете жестко закодировать свои предположения в программах. Однако модули предоставляют простой способ повысить надежность ваших программ, а также улучшить их портируемость без всяких дополнительных затрат.

CGI.pm

Если вы занимаетесь созданием CGI-программ (которые в этой книге не рассматриваются), используйте модуль `CGI.pm`.[2] Если только вы не уверены твердо в том, что делаете (а иногда даже в этом случае), вам не придется иметь дело с интерфейсом и разбором входного кода, с которыми у пользователей возникает столько проблем. Автор `CGI.pm` Линкольн Стейн (Lincoln Stein) потратил много времени на то, чтобы обеспечить работу своего модуля с большинством серверов и операционных систем. Просто используйте модуль и займитесь более интересными частями своего сценария.

Модуль CGI существует в двух разновидностях: с традиционным функциональным и с объектно-ориентированным интерфейсом. Мы будем использовать первую разновидность. Как и прежде, вы можете изучить примеры в документации `CGI.pm`. Наш простой сценарий CGI разбирает входные данные CGI и выводит имена и значения в виде текстового документа. В списке импорта используется обозначение `:all` –

[1] А если вы не знали, что имена файлов и каталогов в UNIX могут содержать символы новой строки, как упоминалось ранее в этом разделе – вы *не знаете* всех правил, верно?

[2] Как и в случае с модулем `CPAN.pm`, мы добавляем суффикс «.pm» в `CGI.pm`, чтобы отличить модуль от самого протокола.

экспортный тег, определяющий группу функций вместо одной функции (как в предыдущих примерах).[1]

```
#!/usr/bin/perl

use CGI qw(:all);

print header("text/plain");

foreach my $param ( param() )
    {
    print "$param: " . param($param) . "\n";
    }
```

Мы хотим выводить код HTML, а модуль `CGI.pm` содержит великое множество вспомогательных функций для этой цели. Он обеспечивает вывод заголовка CGI, начальных конструкций HTML функцией `start_html()`, а многие теги HTML выводятся функциями с соответствующими именами, например функция `h1()` для тега `H1`:

```
#!/usr/bin/perl

use CGI qw(:all);

print header(),
    start_html("This is the page title"),
    h1( "Input parameters" );

my $list_items;
foreach my $param ( param() )
    {
    $list_items .= li( "$param: " . param($param) );
    }

print ul( $list_items );

print end_html();
```

Просто, не правда ли? Вам необязательно знать, как `CGI.pm` все это делает; просто поверьте, что он делает это правильно. Поручив всю «черную работу» `CGI.pm`, вы можете сосредоточиться на интересных частях программы.

Модуль `CGI.pm` способен решать множество других задач: он обеспечивает работу с cookies, перенаправление, создание многостраничных форм и т. д. У нас нет возможности подробно описывать его в книге, однако вы сможете найти дополнительную информацию в документации модуля.

---

[1] Модуль содержит несколько других экспортных тегов для выбора разных групп функций. Например, если вам нужны только функции, относящиеся к CGI, используйте тег `:cgi`; если нужны функции, генерирующие код HTML, используйте тег `:html4`. За подробностями обращайтесь к документации `CGI.pm`.

Базы данных и DBI

Модуль DBI (Database Interface) не поставляется вместе с Perl, но принадлежит к числу самых популярных модулей, так как многие программы работают с теми или иными базами данных. Изящество DBI проявляется в том, что единый интерфейс может использоваться практически для любой стандартной базы данных: от простых файлов, разделенных запятыми, до больших серверов баз данных типа Oracle. Модуль включает драйверы ODBC, причем некоторые из его драйверов даже поддерживаются фирмами-разработчиками. Если вас заинтересуют подробности, найдите книгу «Programming the Perl DBI»[1] Аллигатора Декарта (Alligator Descartes) и Тима Банса (Tim Bunce) (O'Reilly). Также стоит заглянуть на сайт DBI по адресу *http://dbi.perl.org/*.

После установки DBI также необходимо установить DBD (драйвер базы данных, Database Driver). Длинный список всевозможных DBD можно найти при помощи CPAN Search. Установите правильный драйвер для своего сервера баз данных и проследите за тем, чтобы установленная версия драйвера соответствовала версии сервера.

DBI – объектно-ориентированный модуль, но для его использования вам не нужно ничего знать об объектно-ориентированном программировании. Просто следуйте примерам, приведенным в документации. Чтобы подключиться к базе данных, включите модуль директивой `use` и вызовите метод `connect`:

```
use DBI;

$dbh = DBI->connect($data_source, $username, $password);
```

Аргумент `$data_source` содержит информацию, специфическую для используемого драйвера; ее следует получить от DBD. Например, для PostgreSQL используется драйвер `DBD::Pg`, а `$data_source` выглядит примерно так:

```
my $data_source = "dbi:Pg:dbname=name_of_database";
```

После подключения к базе данных начинается цикл подготовки, выполнения и чтения запросов.

```
$sth = $dbh->prepare("SELECT * FROM foo WHERE bla");
$sth->execute();
@row_ary  = $sth->fetchrow_array;
$sth->finish;
```

После завершения работы с базой следует отключиться от нее:

```
$dbh->disconnect();
```

Модуль DBI также обладает другими интересными возможностями. За подробностями обращайтесь к документации.

---

[1] Аллигатор Декарт и Тим Банс «Программирование на Perl DBI». – Пер. с англ. – СПб.: Символ-Плюс, 2000 (в продаже только электронная версия).

Упражнения

Ответы к упражнению приведены в приложении А.

1. **[15] Установите модуль `Module::CoreList` из CPAN. Выведите список всех модулей, поставляемых с Perl 5.008. Хеш, ключами которого являются имена модулей для заданной версии Perl, строится следующей командой:**

```
my %modules = %{ $Module::CoreList::version{5.008} };
```

12

Получение информации о файлах

Ранее мы показывали, как открыть файловый дескриптор для вывода. Обычно при этом создается новый файл, а старый файл с тем же именем стирается. Возможно, вы захотите убедиться в том, что файл с указанным именем не существует. А может, вас интересует, давно ли был создан файл. Или вы хотите перебрать файлы по списку и определить, у каких файлов размер превышает заданное пороговое значение, а с момента последнего обращения прошел заданный промежуток времени. Perl содержит полный набор средств для получения информации о файлах.

Операторы проверки файлов

Прежде чем создавать новый файл в программе, следует проверить, что файл с таким именем не существует. Это поможет предотвратить случайное уничтожение важного файла с электронной таблицей или личным календарем. Проверка существования файла осуществляется конструкцией `-e`:

```
die "Oops! A file called '$filename' already exists.\n"
  if -e $filename;
```

Обратите внимание: содержимое `$!` не включается в сообщение `die`, потому что система в данном случае не отклонила наш запрос. В следующем примере программа проверяет актуальность файла. В данном случае вместо строкового имени проверяется уже открытый файловый дескриптор. Допустим, конфигурационный файл программы должен обновляться каждую неделю или две (предположим, в результате проверки компьютера на заражение вирусами). Если файл не изменялся за последние 28 дней, значит, что-то не так:

```
warn "Config file is looking pretty old!\n"
  if -M CONFIG > 28;
```

Третий пример более сложен. Допустим, дисковое пространство постепенно заполняется и вместо покупки новых дисков было решено переместить большие, редко используемые файлы на архивные ленты. Мы перебираем список файлов[1] и смотрим, у каких файлов размер превышает 100 Кбайт. Но даже если файл имеет больший размер, он перемещается на ленту только в том случае, если к нему не было ни одного обращения за последние 90 дней (так мы узнаем о том, что файл не используется слишком часто):[2]

```
my @original_files = qw/ fred barney betty wilma pebbles dino bamm-bamm /;
my @big_old_files;  # Файлы для перенесения на архивную ленту
foreach my $filename (@original_files) {
  push @big_old_files, $filename
    if -s $filename > 100_000 and -A $filename > 90;
}
```

Обратите внимание: управляющая переменная в цикле `foreach` объявлена с ключевым словом `my`. Это означает, что область видимости переменной ограничивается самим циклом, поэтому пример будет работать с директивой `use strict`. Без ключевого слова `my` будет использоваться глобальная переменная `$filename`.

Все проверки файлов состоят из дефиса и буквы (имени проверки), за которыми следует имя файла или проверяемый дескриптор. Многие проверки возвращают логический признак *true/false*, но некоторые предоставляют более интересную информацию. В табл. 12.1 приведен полный список проверок, а затем описываются особые случаи.

Проверки `-r`, `-w`, `-x` и `-o` сообщают, истинен ли атрибут для действующего идентификатора пользователя или группы[3]; по сути, речь идет о пользователе, «ответственном» за запущенную программу[4]. Эти проверки определяют разрешения по «битам разрешений», установленным для файла в системе. Если в вашей системе применяются списки

---

1 Более вероятно, что вместо готового списка файлов в массиве, как в приведенном примере, вы будете читать данные прямо из файловой системы с использованием глоба или дескриптора каталога (см. главу 13). Поскольку эта тема еще не рассматривалась, мы начнем со списка.

2 Ближе к концу главы будет показано, как сделать этот пример более эффективным.

3 Проверки `-o` и `-O` относятся только к идентификатору пользователя, но не к идентификатору группы.

4 Примечание для квалифицированных читателей: соответствующие проверки `-R`, `-W`, `-X` и `-O` работают с реальным идентификатором пользователя или группы, что может быть важно при использовании программой set-ID. О том, что такое set-ID, можно узнать в хорошей книге о UNIX-программировании для специалистов.

ACL (Access Control List), они также будут использоваться проверками. Фактически эти проверки сообщают, будет ли система хотя бы *пытаться* выполнить некоторое действие; если да, то это еще не означает, что оно будет действительно возможно. Например, проверка `-w` может оказаться истинной для файла на диске CD-ROM, хотя запись на него невозможна, а проверка `-x` может быть истинной для пустого файла, который невозможно выполнить.

Таблица 12.1. Операторы проверки файлов

Проверка	Смысл
`-r`	(Действующему) пользователю или группе разрешено чтение файла
`-w`	(Действующему) пользователю или группе разрешена запись файла
`-x`	(Действующему) пользователю или группе разрешено исполнение файла
`-o`	(Действующий) пользователь или группа является владельцем файла или каталога
`-R`	(Реальному) пользователю или группе разрешено чтение файла
`-W`	(Реальному) пользователю или группе разрешена запись файла
`-X`	(Реальному) пользователю или группе разрешено исполнение файла
`-O`	(Реальный) пользователь или группа является владельцем файла или каталога
`-e`	Имя файла или каталога существует
`-z`	Файл существует и имеет нулевой размер (всегда *false* для каталогов)
`-s`	Файл или каталог существует и имеет ненулевой размер (возвращает размер в байтах)
`-f`	Простой файл
`-d`	Каталог
`-l`	Символическая ссылка
`-S`	Сокет
`-p`	Именованный канал («fifo»)
`-b`	Блочный специальный файл (например, монтируемый диск)
`-c`	Символьный специальный файл (например, устройство ввода/вывода)
`-u`	Для файла или каталога используется `setuid`
`-g`	Для файла или каталога используется `setgid`
`-k`	Для файла или каталога установлен бит закрепления (sticky bit)

Проверка	Смысл
`-t`	Файловый дескриптор является TTY (по данным системной функции `isatty()`; к именам файлов эта проверка неприменима)
`-T`	Файл выглядит как «текстовый»
`-B`	Файл выглядит как «двоичный»
`-M`	Срок последней модификации (в днях)
`-A`	Срок последнего обращения (в днях)
`-C`	Срок модификации индексного узла (в днях)

Проверка `-s` возвращает *true*, если файл не пуст, но это особая разновидность *true*: в действительности возвращается длина файла в байтах, которая для ненулевого числа интерпретируется как *true*.

В файловой системе UNIX[1] существуют семь видов элементов (узлов) файловой системы, представленные семью проверками: `-f`, `-d`, `-l`, `-S`, `-p`, `-b` и `-c`. Каждый элемент должен относиться к одному из этих типов. Однако для символической ссылки, указывающей на файл, истинный результат возвращается как для `-f`, так и для `-l`. Следовательно, если вы хотите знать, является ли элемент символической ссылкой, обычно нужно начинать с `-l`. (О символических ссылках подробнее рассказано в главе 13.)

Проверки `-M`, `-A` и `-C` (обратите внимание на верхний регистр) возвращают количество дней, прошедших с момента последней модификации, обращения к элементу или изменения его индексного узла.[2] (Индексный узел содержит всю информацию о файле кроме его содержимого; за подробностями обращайтесь к man-странице системной функции `stat` или к хорошей книге по внутреннему устройству UNIX.) Срок задается в виде вещественного числа, поэтому если файл был изменен два дня и одну секунду назад, вы можете получить значение `2.00001`. («Дни» в этом случае не всегда соответствуют привычной нам системе отсчета; например, если в 1:30 ночи проверить файл, модифицированный в 23:00, значение `-M` для этого файла составит около `0.1`, хотя по нашим представлениям этот файл был изменен «вчера».)

---

[1] Это описание относится и ко многим системам, не входящим в семейство UNIX, но не все проверки файлов всегда дают осмысленные результаты. Например, вы вряд ли найдете блочные специальные файлы в других системах.

[2] В системах, не входящих в семейство UNIX, эта информация может отличаться, потому что эти системы отслеживают другие временные атрибуты. Например, в некоторых системах поле `ctime` (проверяемое при помощи `-C`) содержит время создания файла (которое в UNIX не отслеживается) вместо времени изменения индексного узла; см. man-страницу *perlport*.

При проверке временных атрибутов файла вы даже можете получить отрицательное значение вида -1.2, которое означает, что метка последнего обращения к файлу сдвинута почти на 30 часов в будущее! Нулевой точкой временной шкалы считается момент запуска программы[1], поэтому отрицательное значение может означать, что давно запущенная программа проверяет файл, последнее обращение к которому произошло совсем недавно. А может быть, временная метка была смещена (случайно или намеренно) на будущее.

Проверки -T и -B пытаются определить, является ли файл текстовым или двоичным. Однако люди, разбирающиеся в файловых системах, знают, что бита, определяющего тип файла, не существует (по крайней мере в UNIX-подобных операционных системах) – как же Perl это делает? Оказывается, Perl жульничает: он открывает файл, просматривает несколько тысяч начальных байт и делает обоснованное предположение. Если среди них обнаруживается много нуль-байтов, необычных управляющих символов и байтов с установленным старшим битом, скорее всего, файл является двоичным. Если же файл не содержит подобных «странностей», он больше похож на текст. Как нетрудно предположить, догадка иногда оказывается ошибочной. Если текстовый файл содержит много шведских или французских слов (с символами, в представлении которых устанавливается старший бит, как в некоторых разновидностях ISO-885 и даже в Юникоде), Perl может решить, что он имеет дело с двоичным файлом. Таким образом, система идентификации неидеальна, но если вам нужно только отделить исходный код от откомпилированных файлов или файлы HTML от графики PNG, этих проверок будет достаточно.

Казалось бы, -T и -B всегда должны давать взаимоисключающие результаты, поскольку текстовый файл не является двоичным, и наоборот, однако в двух особых случаях результаты этих проверок совпадают. Если файл не существует или из него не удается прочитать данные, обе проверки дают ложный результат, так как недоступный файл не является ни текстовым, ни двоичным. Кроме того, пустой файл одновременно может считаться как текстовым, так и двоичным, поэтому обе проверки дают истинный результат.

Проверка -t возвращает *true*, если заданный дескриптор является TTY – проще говоря, если он интерактивен, то есть не является простым файлом или каналом. Если проверка -t STDIN возвращает *true*, это обычно означает, что программа может задавать пользователю вопросы. Если возвращается *false*, программа, скорее всего, получает данные из файла или канала, а не с клавиатуры.

---

[1] Сохраненный в переменной $^T, значение которой можно изменять (командами вида $^T = time;), если вам потребуется получить временные атрибуты по отношению к другому моменту времени.

Не огорчайтесь, если смысл некоторых проверок остался непонятным – если вы никогда не слышали о них, то, скорее всего, они вам не понадобятся. Но если вам интересно, найдите хорошую книгу по программированию для UNIX.[1] (В других системах эти проверки стараются выдать результат, *аналогичный* результату для UNIX, или `undef` для недоступных возможностей. Обычно вы сможете правильно понять, что они делают.)

Если при проверке файла не указано имя файла или дескриптор (то есть при простом вызове вида `-r` или `-s`), по умолчанию в качестве операнда используется файл, имя которого содержится в `$_`.[2] Таким образом, чтобы проверить список имен файлов и узнать, какие из них доступны для чтения, достаточно ввести следующий фрагмент:

```
foreach (@lots_of_filenames) {
  print "$_ is readable\n" if -r;  # same as -r $_
}
```

Но если параметр не указан, будьте внимательны и следите за тем, чтобы то, что следует за проверкой файла, не *выглядело* как параметр. Например, если потребуется узнать размер файла в килобайтах, а не в байтах, возникает искушение разделить результат `-s` на 1000 (или 1024):

```
# Имя файла в $_
my $size_in_K = -s / 1000;       # Сюрприз!
```

Когда парсер Perl встречает косую черту, он вовсе не думает о делении. Поскольку парсер ищет необязательный операнд для `-s`, он видит то, что принимает за начало регулярного выражения, в косых чертах. Впрочем, подобные недоразумения легко предотвращаются: достаточно заключить проверку файла в круглые скобки:

```
my $size_in_k = (-s) / 1024;     # По умолчанию используется $_
```

Конечно, явная передача параметра исключает любые недоразумения при проверке.

Проверка нескольких атрибутов одного файла

Объединение нескольких файловых проверок позволяет создавать сложные логические условия. Предположим, программа должна выполнить некую операцию только с файлами, доступными как для чтения, так и для записи. Проверки атрибутов объединяются оператором `and`:

---

[1] Уильям Ричард Стивенс и Стивен Раго «UNIX. Профессиональное программирование». – Пер. с англ. – СПб.: Символ-Плюс, 2007.

[2] Единственным исключением является проверка `-t`, бесполезная для имен файлов (они никогда не соответствуют TTY). По умолчанию она проверяет дескриптор `STDIN`.

```
if( -r $file and -w $file ) {
    ... }
```

Однако объединение проверок обойдется весьма недешево. При каждой проверке Perl обращается к файловой системе с запросом всей информации о файле (фактически Perl каждый раз выполняет системную функцию *stat*, о которой будет рассказано в следующем разделе). Хотя информация уже была получена при проверке `-r`, Perl снова запрашивает ее для проверки `-w`. Как неэффективно! При проверке атрибутов большого количества файлов могут возникнуть серьезные проблемы с быстродействием.

В Perl существует специальная сокращенная запись, которая позволит обойтись без напрасной работы. Виртуальный файловый дескриптор _ (просто символ подчеркивания) использует информацию, полученную в результате выполнения последнего оператора проверки файла. Теперь Perl достаточно запросить информацию о файле всего один раз:

```
if( -r $file and -w _ ) {
    ... }
```

Для использования _ проверки файлы необязательно располагать рядом друг с другом. Здесь они размещаются в разных условиях `if`:

```
if( -r $file ) {
    print "The file is readable!\n";
    }

if( -w _ ) {
    print "The file is writable!\n";
    }
```

Однако будьте внимательны и следите за тем, чтобы содержимое _ относилось именно к тому файлу, который вам нужен. Если между проверками происходит что-то еще, например вызов пользовательской функции, информация о последнем файле может измениться. Например, в следующем примере вызывается функция `lookup`, в которой также выполняется своя проверка. При возврате и выполнении другой проверки файловый дескриптор _ содержит не данные `$file`, как мы ожидаем, а данные `$other_file`:

```
if( -r $file ) {
    print "The file is readable!\n";
    }

lookup( $other_file );

if( -w _ ) {
    print "The file is writable!\n";
    }

sub lookup {
    return -w $_[0];
    }
```

Сгруппированная проверка файлов

До выхода Perl 5.10, если вы хотели одновременно проверить несколько атрибутов файла, это приходилось делать по отдельности (даже при том, что дескриптор _ избавлял вас от части работы).

Допустим, мы хотим узнать, доступен ли файл для чтения и записи одновременно. Для этого необходимо сначала выполнить проверку на чтение, а затем проверку на запись:

```
if( -r $file and -w _ ) {
    print "The file is both readable and writable!\n";
    }
```

Конечно, проверки было бы удобнее выполнить одновременно. Perl 5.10 позволяет последовательно сгруппировать операторы проверки перед именем файла:

```
use 5.010;

if( -w -r $file ) {
    print "The file is both readable and writable!\n";
    }
```

Пример почти не отличается от предыдущего, изменился только синтаксис. Хотя на первый взгляд кажется, что порядок проверки изменился, в действительности Perl начинает с оператора, находящегося ближе к имени. Обычно это несущественно.

Сгруппированные проверки файлов особенно удобны в сложных ситуациях. Допустим, требуется найти все каталоги, доступные для чтения, записи и исполнения, владельцем которых является текущий пользователь. Для этого понадобится лишь правильный набор файловых проверок:

```
use 5.010;

if( -r -w -x -o -d $file ) {
    print "My directory is readable, writable, and executable!\n";
    }
```

Группировка не подходит для проверок с возвращаемыми значениями, отличными от *true* и *false*, которые нам хотелось бы использовать в сравнениях. Казалось бы, следующий фрагмент кода сначала проверяет, что элемент является каталогом, а затем – что его размер менее 512 байт, но на самом деле это не так:

```
use 5.010;

if( -s -d $file < 512) { # WRONG! DON'T DO THIS
    print "The directory is less than 512 bytes!\n";
    }
```

Чтобы понять, что происходит, достаточно записать сгруппированные проверки файлов в предыдущем варианте записи. Результат комбинации проверок становится аргументом для сравнения:

```
if( ( -d $file and -s _ ) < 512 ) {
    print "The directory is less than 512 bytes!\n";
    }
```

Когда `-d` возвращает *false*, Perl сравнивает полученное значение *false* с 512. Результат сравнения оказывается истинным, потому что *false* интерпретируется как 0, а это меньше 512. Чтобы не создавать путаницы и помочь программистам сопровождения, которые придут после нас, достаточно разделить проверку на две части:

```
if( -d $file and -s _ < 512 ) {
    print "The directory is less than 512 bytes!\n";
    }
```

Функции stat и lstat

Операторы проверки файлов хорошо подходят для получения информации об атрибутах, относящихся к конкретному файлу или дескриптору, но они не дают полной картины. Например, ни одна проверка не возвращает количество ссылок на файл или идентификатор пользователя (UID) его владельца. Чтобы получить остальную информацию о файле, вызовите функцию `stat`, которая дает более или менее всю информацию, возвращаемую системной функцией UNIX `stat` (вероятно, она расскажет гораздо больше, чем вам захочется знать[1]). В операнде `stat` передается файловый дескриптор (в том числе и виртуальный дескриптор_) или выражение, вычисляемое как имя файла. Возвращаемое значение содержит либо пустой список, означающий, что вызов `stat` завершился неудачей (обычно из-за того, что файл не существует), либо список из 13 чисел, который проще всего описывается следующим списком скалярных переменных:

[1] В системах, не входящих в семейство UNIX, функции `stat` и `lstat`, а также операторы проверки файлов должны возвращать «по возможности близкую информацию». Например, система, в которой нет идентификаторов пользователей (то есть система с одним «пользователем» с точки зрения UNIX), должна возвращать нули вместо идентификаторов пользователя и группы, так как единственным пользователем является системный администратор. Если вызов `stat` или `lstat` завершается неудачей, функция возвращает пустой список. Если при вызове системной функции, обеспечивающей проверку файла, происходит сбой (или эта функция недоступна в системе), проверка обычно возвращает `undef`. За последней информацией о том, чего ожидать в разных системах, обращайтесь к man-странице *perlport*.

```
my($dev, $ino, $mode, $nlink, $uid, $gid, $rdev,
  $size, $atime, $mtime, $ctime, $blksize, $blocks)
    = stat($filename);
```

Имена относятся к полям структуры `stat`, которая подробно описана в man-странице *stat(2)*. Однако с кратким списком самых важных переменных вы можете ознакомиться прямо сейчас:

`$dev` и `$ino`

Номер устройства и номер индексного узла, соответствующие файлу. Вместе они образуют «удостоверение личности» файла. Файл может иметь более одного имени (жесткие ссылки), но комбинация номеров устройства и индексного узла всегда уникальна.

`$mode`

Набор битов разрешений доступа и еще некоторых битов, соответствующих файлу. Если вы когда-либо использовали команду UNIX `ls -l` для получения подробного (длинного) списка файлов, то заметили, что каждая строка начинается с последовательности вида `-rwxr-xr-x`. Девять букв и дефисов в разрешениях файла[1] соответствуют девяти младшим битам `$mode`, которые в данном примере складываются в восьмеричное число 0755. Другие биты, помимо 9 младших, описывают другие аспекты информации о файле. При работе с данными `$mode` обычно используются поразрядные операторы, описанные далее в этой главе.

`$nlink`

Количество (жестких) ссылок на файл или каталог; определяет количество «истинных» имен данного элемента. Для каталогов значение всегда равно 2 и более, а для файлов оно (чаще всего) равно 1. Мы вернемся к этой теме в главе 13, когда речь пойдет о создании ссылок на файлы. В результатах `ls -l` значение `$nlink` соответствует числу после строки разрешений.

`$uid` и `$gid`

Числовые идентификаторы пользователя и группы, определяющие принадлежность файла.

`$size`

Размер в байтах, возвращаемый оператором проверки файла `-s`.

`$atime`, `$mtime` и `$ctime`

Три временные метки, представленные в системном формате: 32-разрядные числа, определяющие количество секунд, прошедших с на-

[1] Первый символ строки не является битом разрешения, а описывает тип элемента: дефис – для обычного файла, `d` – для каталога, `l` – для символической ссылки и т. д. Команда `ls` определяет его по другим битам, кроме 9 младших.

чала *эпохи*, произвольной отправной точки для измерения системного времени. В UNIX и ряде других систем эпоха отсчитывается от полуночи начала 1970-года по единому мировому времени, но на некоторых компьютерах используется другая эпоха. Позднее в этой главе приводится более подробная информация о том, как преобразовать временную метку в более удобное значение.

Вызов `stat` для имени символической ссылки возвращает информацию об объекте, на который указывает ссылка, а не о самой ссылке (если только она не указывает на недоступный объект). Если вам необходима (в целом бесполезная) информация о самой символической ссылке, используйте функцию `lstat` вместо `stat`; эта функция возвращает ту же информацию в том же порядке. Если операнд не является символической ссылкой, `lstat` возвращает ту же информацию, что и `stat`.

По умолчанию в качестве операнда `stat` и `lstat` используется переменная `$_`; это означает, что нижележащий вызов системной функции `stat` будет выполнен для файла, имя которого содержится в скалярной переменной `$_`.

Функция localtime

Метки времени (например, полученные при вызове `stat`) обычно выглядят как длинные и непонятные числа, например 1180630098. Для большинства программистов такие метки неудобны, разве что вы захотите сравнить две метки посредством вычитания. Возможно, вам потребуется преобразовать метку в строку вида `Thu May 31 09:48:18 2007`. В Perl эта задача решается функцией `localtime` в скалярном контексте:

```
my $timestamp = 1180630098;
my $date = localtime $timestamp;
```

В списочном контексте `localtime` возвращает список чисел, причем некоторые его элементы оказываются довольно неожиданными:

```
my($sec, $min, $hour, $day, $mon, $year, $wday, $yday, $isdst)
  = localtime $timestamp;
```

`$mon` – номер месяца от 0 до 11 – может использоваться в качестве индекса в массиве названий месяцев. `$year` – количество лет, прошедших с 1900 года; чтобы получить «настоящий» год, достаточно прибавить 1900. Значение `$wday` лежит в интервале от 0 (воскресенье) до 6 (суббота), а `$yday` – номер дня года, от 0 (1 января) до 364 или 365 (31 декабря).

Есть еще две функции, которые пригодятся при работе с временными метками. Функция `gmtime` аналогична `localtime`, но результат она возвращает в формате единого мирового времени (когда-то называвшемся «гринвичским»). Если вам понадобится текущая временная метка от системных часов, воспользуйтесь функцией `time`. И `localtime`, и `gmtime`

при вызове без параметра по умолчанию используют текущее значение `time`:

```
my $now = gmtime;  # Получить текущее мировое время в виде строки
```

За дополнительной информацией о работе с датой и временем обращайтесь к перечню модулей в приложении B.

Поразрядные операторы

Для манипуляций с числами на уровне отдельных битов (как при работе с битами разрешений, возвращаемыми `stat`) потребуются поразрядные операторы. Эти операторы выполняют двоичные математические операции. Поразрядный оператор конъюнкции (`&`) сообщает, какие биты установлены в левом *и* в правом аргументе. Например, значение выражения `10 & 12` равно 8. Двоичная единица в разряде результата устанавливается только в том случае, если соответствующий разряд обоих операндов тоже содержит 1. Таким образом, поразрядная конъюнкция 10 (`1010` в двоичном представлении) и 12 (`1100` в двоичном представлении) дает 8 (`1000` – единственная единица находится в первом разряде, который равен 1 у левого *и* у правого операндов); см. рис. 12.1.

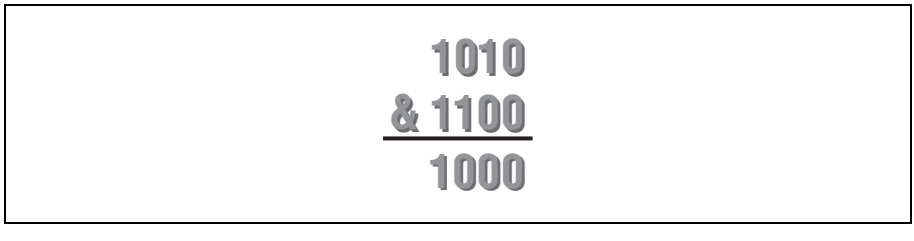


Рис. 12.1. *Поразрядная конъюнкция*

Основные поразрядные операторы перечислены в табл. 12.2.

Рассмотрим пример обработки данных `$mode`, возвращаемых функцией `stat`. Результаты этих операций с битами также могут пригодиться при использовании `chmod` (см. главу 13):

```
# $mode - значение, полученное от stat для CONFIG
warn "Hey, the configuration file is world-writable!\n"
  if $mode & 0002;                          # Проблема с безопасностью
my $classical_mode = 0777 & $mode;          # Отсечение лишних
                                            # старших битов
my $u_plus_x = $classical_mode | 0100;      # Установка одного бита
my $go_minus_r = $classical_mode & (~ 0044); # Сброс двух битов
```

Таблица 12.2. Поразрядные операторы

Проверка	Смысл
`10 & 12`	Поразрядная конъюнкция – определяет, какие биты истинны в обоих операндах (получается 8)
`10 | 12`	Поразрядная дизъюнкция – определяет, какие биты истинны хотя бы в одном из двух операндов (получается 14)
`10 ^ 12`	Поразрядная исключающая дизъюнкция – определяет, какие биты истинны только в одном из двух операндов (получается 6)
`6 << 2`	Поразрядный сдвиг влево – левый операнд сдвигается на количество битов, определяемое правым операндом, с заполнением младших разрядов нулевыми битами (получается 24)
`25 >> 2`	Поразрядный сдвиг вправо – левый операнд сдвигается на количество битов, определяемое правым операндом, с потерей младших байтов (получается 6)
`~ 10`	Поразрядное отрицание, также называемое унарным дополнением – возвращает число, содержащее инвертированные биты в каждом разряде (получается **0xFFFFFFF5**, см. текст)

Работа с битовыми строками

Все поразрядные операторы могут работать как с целыми числами, так и с битовыми строками. Если операнды являются целыми числами, то результатом будет целое число (как минимум 32-разрядное, но может быть и больше, если поддерживается вашим компьютером. Таким образом, на 64-разрядном компьютере для операции `~10` будет получен 64-разрядный результат `0xFFFFFFFFFFFFFFF5` вместо 32-разрядного результата `0xFFFFFFF5`).

Но если любой операнд поразрядного оператора является строкой, Perl выполнит операцию с соответствующей цепочкой битов. Таким образом, для выражения `"\xAA" | "\x55"` будет получена строка `"\xFF"`. Обратите внимание: значения являются однобайтовыми строками; результат представляет собой байт, в котором установлены все 8 бит. Длина битовых строк не ограничивается.

Это один из очень немногочисленных аспектов, в которых Perl различает строки и числа. За дополнительной информацией об использовании поразрядных операторов со строками обращайтесь к man-странице *perlop*.

Упражнения

Ответы к упражнениям приведены в приложении А.

1. [15] Напишите программу, которая получает в командной строке список файлов и для каждого файла сообщает, доступен ли он для

чтения, записи, исполнения или не существует. (Подсказка: напишите функцию, которая будет при вызове выполнять все проверки для одного файла.) Что сообщит функция о файле, которому командой *chmod* было присвоено значение 0? (Если вы работаете в системе UNIX, используйте команду *chmod 0 some_file*, чтобы запретить чтение, запись и исполнение файла.) Во многих командных процессорах звездочка в аргументах запуска обозначает все обычные файлы в текущем каталоге. Иначе говоря, команда вида `./ex12-2 *` запрашивает у программы атрибуты сразу нескольких файлов.

2. [10] Напишите программу, которая определяет самый старый файл из списка, переданного в командной строке, и сообщает его «возраст» в днях. Что сделает ваша программа, если список пуст (то есть командная строка не содержит ни одного файла)?
3. [10] Напишите программу, которая использует сгруппированные операторы проверки файлов для перечисления всех файлов, указанных в командной строке, принадлежащих вам и доступных для чтения и записи.

13

Операции с каталогами

Файлы, с которыми мы работали в предыдущей главе, обычно размещаются в одном каталоге с программой. Но современные операционные системы позволяют распределять файлы по каталогам, чтобы мы могли хранить любимые записи Beatles в формате MP3 отдельно от исходного кода для «книги с ламой» и не отправили файл MP3 в издательство. Perl позволяет работать с каталогами напрямую, причем выполняемые операции в целом портируемы между разными операционными системами.

Перемещение по дереву каталогов

Программа имеет определенный «рабочий каталог», который является отправной точкой для построения относительных путей. Иначе говоря, когда вы ссылаетесь на файл `fred`, имеется в виду файл `fred` в текущем рабочем каталоге.

Рабочий каталог меняется оператором `chdir`. Команда работает практически так же, как команда `cd` командного процессора UNIX:

```
chdir "/etc" or die "cannot chdir to /etc: $!";
```

Так как смена каталога осуществляется системной функцией, при возникновении ошибки будет задано значение `$!`. Обычно `$!` проверяется при получении от `chdir` значения *false*, потому что это указывает на возникновение каких-то сбоев.

Рабочий каталог наследуется всеми процессами, запущенными Perl (эта тема более подробно рассматривается в главе 16). С другой стороны, изменение рабочего каталога не влияет на процесс, запустивший

Perl, например командный процессор.[1] Это означает, что вы не сможете написать программу Perl, заменяющую команду *cd* командного процессора.

Если параметр не указан, Perl определяет домашний каталог, насколько у него это получится, и пытается назначить его рабочим каталогом (по аналогии с командой *cd*, введенной в командном процессоре без параметров). Это одно из немногих мест, в которых по умолчанию не используется переменная `$_`.

Некоторые командные процессоры позволяют включать в команду *cd* путь с префиксом `~`, чтобы использовать в качестве отправной точки домашний каталог другого пользователя (например, `cd ~merlyn`). Это функция командного процессора, а не операционной системы, а Perl вызывает функции операционной системы напрямую, поэтому префикс `~` не работает с `chdir`.

Глобы

Обычно командный процессор расширяет файловые спецификации в командной строке до имен конкретных файлов, совпадающих с ними. Эти спецификации называются *глобами* (*glob*). Например, если передать команде *echo* файловую спецификацию `*.pm`, командный процессор расширит ее до списка подходящих имен:

```
$ echo *.pm
barney.pm dino.pm fred.pm wilma.pm
$
```

Команде *echo* ничего не нужно знать о замене `*.pm`, потому что командный процессор уже выполнил ее. Глобы работают даже в программах Perl:

```
$ cat >show-args
foreach $arg (@ARGV) {
  print "one arg is $arg\n";
}
^D
$ perl show-args *.pm
one arg is barney.pm
one arg is dino.pm
one arg is fred.pm
one arg is wilma.pm
$
```

[1] Это не недостаток Perl, а специфическая особенность UNIX, Windows и других систем. Если вы действительно хотите сменить рабочий каталог командного процессора, обращайтесь к документации.

Обратите внимание: *show-args* не нужно ничего знать о глобах – имена уже расширены и занесены в массив @ARGS.

Но иногда в программах Perl появляются шаблоны типа *.pm. Можно ли расширить их до подходящих имен файлов без особых усилий? Конечно – достаточно воспользоваться оператором glob:

```
my @all_files = glob "*";
my @pm_files = glob "*.pm";
```

Здесь в @all_files заносятся все файлы текущего каталога, отсортированные по алфавиту и не начинающиеся с точки (по аналогии с командным процессором). В массив @pm_files заносится тот же список, который был получен ранее с передачей шаблона *.pm в командной строке.

Практически все шаблоны, выводимые в командной строке, также можно разместить в (единственном) аргументе glob, в том числе и список шаблонов, разделенных запятыми:

```
my @all_files_including_dot = glob ".* *";
```

Здесь мы передаем дополнительный параметр .* для получения имен всех файлов – как начинающихся с точки, так и обычных. Обратите внимание: пробел между этими двумя шаблонами в строке, заключенной в кавычки, важен, так как он разделяет два разных шаблона.[1] Почему эта конструкция в точности повторяет функциональность командного процессора? Дело в том, что до выхода версии Perl 5.6 оператор glob просто вызывал /bin/csh[2] для выполнения расширения. По этой причине глобы занимали много времени, а их обработка для больших каталогов (и в ряде других случаев) могла приводить к сбоям. Сознательные программисты Perl старались избегать глобов, отдавая предпочтение дескрипторам каталогов, которые мы рассмотрим позже в этой главе. Впрочем, если вы используете современную версию Perl, об этих проблемах можно забыть.

Альтернативный синтаксис глобов

Говоря о глобах, мы имеем в виду оператор glob, однако ключевое слово glob не так уж часто встречается в программах, использующих глобы. Почему? Большая часть старого кода была написана до того, как оператор glob получил свое имя. Вместо этого он использовался в синтаксисе с угловыми скобками, напоминающем синтаксис чтения из файлового дескриптора:

```
my @all_files = <*>; ## Полностью эквивалентно my @all_files = glob "*";
```

---

[1] Пользователи Windows привыкли к тому, что глоб *.* означает «все файлы». Но в действительности он означает «все файлы, в именах которых присутствует точка» – даже в Perl для Windows.

[2] Или возможную замену, если командный процессор *csh* был недоступен.

Значение в угловых скобках интерполируется по аналогии со строками в кавычках; это означает, что переменные Perl перед обработкой глоба заменяются своими текущими значениями:

```
my $dir = "/etc";
my @dir_files = <$dir/* $dir/.*>;
```

Этот фрагмент получает список всех файлов (с точкой и без) из заданного каталога, так как переменная `$dir` заменяется своим текущим значением.

Итак, если угловыми скобками обозначается как чтение из файлового дескриптора, так и операции с глобами, как же Perl определяет, какой из двух операторов должен использоваться в каждом конкретном случае? По содержимому угловых скобок. Файловый дескриптор должен быть идентификатором Perl. Таким образом, если содержимое угловых скобок является идентификатором Perl, имеется в виду операция чтения; в противном случае используется операция с глобами. Пример:

```
my @files = <FRED/*>;  ## Глоб
my @lines = <FRED>;    ## Чтение из файлового дескриптора
my $name = "FRED";
my @files = <$name/*>; ## Глоб
```

У этого правила существует единственное исключение: если в угловых скобках находится простая скалярная переменная (не элемент хеша или массива) с именем файлового дескриптора, то конструкция интерпретируется как *опосредованное чтение из файлового дескриптора*:[1]

```
my $name = "FRED";
my @lines = <$name>; ## Опосредованное чтение из дескриптора FRED
```

Выбор между операцией с глобом и файловым дескриптором осуществляется во время компиляции, и поэтому он не зависит от содержимого переменной.

При желании функциональность опосредованного чтения из файловых дескрипторов можно представить оператором `readline`[2]; такая запись более понятна:

```
my $name = "FRED";
my @lines = readline FRED;  ## Чтение из FRED
my @lines = readline $name; ## Чтение из FRED
```

---

[1] Если косвенный дескриптор является текстовой строкой, он подвергается проверке «символических ссылок», запрещенной под действием директивы `use strict`. Однако косвенный дескриптор также может представлять собой typeglob или ссылку на объект ввода/вывода; тогда он будет работать даже в режиме `use strict`.

[2] Только в Perl 5.005 и более поздних версиях.

Однако и оператор `readline` используется редко, опосредованное чтение тоже относительно нетипично и выполняется обычно с простой скалярной переменной.

Дескрипторы каталогов

Для получения списка имен из заданного каталога также можно воспользоваться *дескриптором каталога*. Дескрипторы каталога очень похожи на файловые дескрипторы – как по внешнему виду, так и по поведению. Их тоже необходимо открыть (`opendir` вместо `open`), прочитать из них данные (`readdir` вместо `readline`), а затем закрыть (`closedir` вместо `close`). Но вместо чтения *содержимого* файлов читаются *имена* файлов (и прочая информация о них). Пример:

```
my $dir_to_process = "/etc";
opendir DH, $dir_to_process or die "Cannot open $dir_to_process: $!";
foreach $file (readdir DH) {
  print "one file in $dir_to_process is $file\n";
}
closedir DH;
```

Дескрипторы каталогов, как и файловые дескрипторы, автоматически закрываются в конце работы программы или при их повторном открытии для другого каталога.

В отличие от глобов, для которых в старых версиях Perl порождался отдельный процесс, дескриптор каталога никогда не создает новый процесс. Это повышает их эффективность в приложениях, выжимающих из компьютера все вычислительные ресурсы. Однако операции с ними выполняются на более низком уровне, а это означает, что программисту придется выполнять большую часть работы самостоятельно.

Например, имена файлов возвращаются без упорядочения[1], а в список включаются все файлы, а не только те, которые соответствуют заданному шаблону (например, `*.pm`, как в примере с глобами). В частности, в него включаются файлы с точкой и специальные записи `.` и `..`.[2] Таким образом, если нас интересуют только файлы с расширением *pm*, в цикл включается функция пропуска файлов:

```
while ($name = readdir DIR) {
  next unless $name =~ /\.pm$/;
```

[1] В действительности используется порядок следования записей каталога, аналогичный тому, который используется командой *ls -l* или *find*.

[2] Не допускайте ошибку, встречающуюся во многих старых программах для UNIX: нельзя предполагать, что `.` и `..` всегда возвращаются в первых двух элементах (с сортировкой или без нее). А если вам это и в голову не приходило, считайте, что мы об этом не говорили, потому что это ошибочное предположение. Пожалуй, нам вообще не стоило упоминать об этом.

```
  ... Продолжение обработки ...
}
```

Обратите внимание: в данном случае используется синтаксис регулярного выражения, а не глоба. В этом примере отбираются все файлы, не начинающиеся с точки:

```
next if $name =~ /^\./;
```

А если, скажем, нас интересуют все файлы, кроме специальных записей `.` (текущий каталог) и `..` (родительский каталог), это можно выразить так:

```
next if $name eq ".." or $name eq "..";
```

Мы переходим к той части, в которой многие программисты путаются, так что будьте внимательны. Имена файлов, возвращаемые оператором `readdir`, не содержат пути. Возвращаются просто имена файлов в каталоге. Таким образом, вы получаете не `/etc/passwd`, а `passwd`. (В этом проявляется еще одно отличие от операций с глобами, отсюда и возникают недоразумения.)

Итак, для получения полных имен необходимо выполнить дополнительную обработку результатов:

```
opendir SOMEDIR, $dirname or die "Cannot open $dirname: $!";
while (my $name = readdir SOMEDIR) {
  next if $name =~ /^\./;           # Пропустить файлы, начинающиеся с точки
  $name = "$dirname/$name";         # Добавить путь
  next unless -f $name and -r $name;   # Только файлы с доступом для чтения
  ...
}
```

Без обработки операторы проверки файлов будут проверять файлы в текущем каталоге, а не в том каталоге, имя которого хранится в `$dirname`. Это самая распространенная ошибка при использовании дескрипторов каталогов.

Рекурсивное чтение каталогов

Вряд ли вам понадобится возиться с рекурсивным чтением каталогов в самом начале вашей карьеры Perl-программиста. А раз так, вместо того чтобы отвлекать вас описанием возможностей замены уродливых сценариев *find*, мы просто скажем, что Perl поставляется с библиотекой `File::Find`, обеспечивающей удобный механизм рекурсивной обработки каталогов. Мы упоминаем об этом еще и для того, чтобы вы не начинали писать собственные функции (за это обычно берутся все новички через десяток-другой часов программирования), а потом мучились с вопросами: «Что такое локальные дескрипторы каталогов?» или «Как мне вернуться в старый каталог?»

Операции с файлами и каталогами

Perl очень часто применяется для обработки файлов и каталогов. Так как язык Perl вырос в среде UNIX и до сих пор активно используется в ней, может показаться, что материал этой главы ориентирован на UNIX. Но к счастью, Perl старается работать аналогичным образом (настолько, насколько это возможно) и в других системах, не входящих в семейство UNIX.

Удаление файлов

Файлы обычно создаются для хранения данных. Однако со временем данные устаревают и файл приходится удалять из системы. На уровне командного процессора UNIX удаление файла или файлов выполняется командой *rm*:

```
$ rm slate bedrock lava
```

В Perl для этой цели используется оператор `unlink`:

```
unlink "slate", "bedrock", "lava";
```

Три файла с заданными именами бесследно исчезают из системы.

Так как функция `unlink` получает список, а функция `glob` возвращает список, эти две функции можно объединить для удаления множества файлов по маске:

```
unlink glob "*.o";
```

Результат аналогичен выполнению команды `rm *.o` в командном процессоре, если не считать того, что нам не пришлось порождать отдельный процесс *rm*. Теперь ваши файлы с важными данными будут уничтожаться гораздо быстрее!

Возвращаемое значение `unlink` сообщает количество успешно удаленных файлов. Возвращаясь к первому примеру, мы можем проверить, успешно ли прошло удаление:

```
my $successful = unlink "slate", "bedrock", "lava";
print "I deleted $successful file(s) just now\n";
```

Конечно, если результат равен 3, мы знаем, что все файлы были удалены, а если 0 – то ни один файл не был удален. А если функция возвращает 1 или 2? По этому значению невозможно определить, какие файлы были удалены. Если вас интересует эта информация, удаляйте файлы по одному в цикле:

```
foreach my $file (qw(slate bedrock lava)) {
  unlink $file or warn "failed on $file: $!\n";
}
```

Здесь каждая операция удаляет только один файл, поэтому возвращаемое значение равно либо 0 (неудача), либо 1 (успешное удаление); мы получаем удобный логический признак для управления выполнением `warn`. Конструкция `or warn` аналогична `or die` – конечно, не считая отсутствия фатальной ошибки (см. главу 5). В данном случае сообщение `warn` завершается символом новой строки, потому что сообщение было выдано не из-за ошибки в *нашей* программе.

Если попытка выполнения `unlink` завершается неудачей, в переменную `$!` заносится информация, связанная с ошибкой операционной системы, которую мы включаем в сообщение. Это имеет смысл только при последовательном удалении файлов, потому что следующий неудачный запрос к операционной системе сбросит переменную. Вызов `unlink` не может использоваться для удаления каталога (подобно тому, как каталог нельзя удалить простым вызовом *rm*). О том, как удалить каталог, рассказано ниже в описании функции `rmdir`.

А теперь один малоизвестный факт из области UNIX. Оказывается, даже если файл недоступен для чтения, записи и исполнения (а может быть, он вообще принадлежит другому пользователю), его все равно можно удалить. Дело в том, что разрешения на удаление файла не зависят от битов разрешений самого файла; они определяются битами разрешений каталога, содержащего этот файл.

Мы упоминаем об этом, потому что начинающие программисты Perl экспериментируют с `unlink`: они создают файл, устанавливают командой *chmod* разрешение 0 (запрет как чтения, так и записи) и проверяют, что вызов `unlink` завершается неудачей. Но вместо этого файл бесследно исчезает.[1] Если вы хотите увидеть неудачную попытку `unlink`, попробуйте удалить */etc/passwd* или другой системный файл. Этот файл находится под контролем системного администратора, и вы не сможете удалить его.[2]

Переименование файлов

Функция `rename` присваивает новое имя существующему файлу:

```
rename "old", "new";
```

Как и аналогичная команда UNIX *mv*, она берет файл с именем *old* и присваивает ему имя *new* в том же каталоге. Переименование даже может сопровождаться перемещением файла:

```
rename "over_there/some/place/some_file", "some_file";
```

---

[1] Некоторые из них даже знают, что *rm* обычно просит подтвердить удаление файла. Но *rm* – команда, а *unlink* – системная функция. Системные функции никогда не просят разрешения, да и потом не извиняются.

[2] Конечно, если вам хватит ума проделать это с правами системного администратора, вы сами виноваты.

Команда перемещает файл *some_file* из другого каталога в текущий каталог при условии, что пользователь, запустивший программу, обладает соответствующими разрешениями.[1] Как и большинство функций, обращающихся с запросами к операционной системе, `rename` возвращает *false* в случае неудачи и устанавливает `$!` при ошибке операционной системы, о чем можно (и нужно) сообщить пользователю при помощи `or die` или `or warn`.

В группах новостей, посвященных использованию командных интерпретаторов UNIX, часто[2] встречается один вопрос: можно ли переименовать все файлы с расширением *.old* так, чтобы они сохранили прежние имена, но получили расширение *.new*? Вот как это делается в Perl:

```
foreach my $file (glob "*.old") {
  my $newfile = $file;
  $newfile =~ s/\.old$/.new/;
  if (-e $newfile) {
    warn "can't rename $file to $newfile: $newfile exists\n";
  } elsif (rename $file, $newfile) {
    ## Успешное переименование, ничего не делать
  } else {
    warn "rename $file to $newfile failed: $!\n";
  }
}
```

Проверка существования `$newfile` необходима из-за того, что `rename` преспокойно запишет переименованный файл на место существующего (если пользователь обладает разрешениями на удаление соответствующего файла). Мы вставили эту проверку, чтобы свести к минимуму возможную потерю информации. Конечно, если вы *хотите* заменять существующие файлы вида *wilma.new*, проверка `-e` не нужна.

Первые две строки цикла обычно объединяются в следующей записи:

```
(my $newfile = $file) =~ s/\.old$/.new/;
```

Программа объявляет переменную `$newfile`, копирует ее исходное значение из `$file`, а затем преобразовывает `$newfile` по результатам замены. Команду можно прочитать в виде «преобразовать `$file` в `$newfile` с использованием замены в правой части». Круглые скобки необходимы из-за относительных приоритетов операций.

Впервые увидев эту замену, некоторые программисты интересуются, почему обратная косая черта необходима слева, а не справа. Две сторо-

---

[1] При этом файлы должны находиться в одной файловой системе. Позднее в этой главе вы узнаете, почему было установлено это правило.

[2] Это не просто старый, привычный вопрос; вопрос о переименовании группы файлов является *самым* часто задаваемым вопросом в этих группах. Именно по этой причине он стоит на первом месте в списках FAQ этих групп. И при этом все равно остается самым частым вопросом. Хм.

ны не симметричны: в левой части стоит регулярное выражение, а в правой – строка в кавычках. Соответственно мы используем шаблон `/\.old$/`, означающий «`.old` с привязкой к концу строки» (привязка нужна, чтобы предотвратить замену первого вхождения `.old` в файле `betty.old.old`), а в правой части можно просто записать заменяющую строку `.new`.

Ссылки и файлы

Чтобы лучше усвоить суть некоторых операций с файлами и каталогами, желательно понимать модель файлов и каталогов в системе UNIX, даже если вы работаете в другой системе, которая ведет себя несколько иначе. Как обычно, наше краткое изложение не дает полной картины; за подробностями обращайтесь к любой хорошей книге по внутреннему устройству UNIX.

Смонтированный том представляет собой дисковое устройство (или нечто иное, что работает более или менее похожим образом, – раздел жесткого диска, флоппи-диск, CD-ROM или DVD-ROM). Том может содержать любое количество файлов и каталогов. Каждый файл хранится в пронумерованном индексном узле, который можно рассматривать как конкретный фрагмент дискового пространства. Один файл может храниться в индексном узле 613, другой – в узле 7033 и т. д.

Чтобы найти некоторый файл, необходимо провести поиск по каталогу. Каталог является особой разновидностью файла, находящейся под управлением системы. Фактически в каталоге хранится таблица с именами файлов и соответствующими им индексными узлами.[1] Наряду с другими объектами каждый каталог всегда содержит две специальные записи. Одна из них (`.`) представляет сам каталог, а другая (`..`) – родительский каталог[2], то есть каталог более высокого уровня в иерархии. На рис. 13.1 изображены два индексных узла. В одном из них хранится файл с именем *chicken*, а в другом – каталог */home/barney/poems*, содержащий этот файл. Файл хранится в индексном узле 613, а каталог – в индексном узле 919. (Имя самого каталога *poems* на рисунке не показано, потому что оно хранится в другом каталоге.) Каталог содержит записи трех файлов (включая *chicken*) и двух каталогов (один из которых содержит ссылку на сам каталог с индексным узлом 919);

---

[1] В семействе UNIX (в других системах обычно нет индексных узлов, жестких ссылок и т. д.) для просмотра номеров индексных узлов файлов можно воспользоваться ключом `-i` команды *ls*. Попробуйте ввести команду *ls -ail*. Если два и более объектов файловой системы имеют одинаковый номер индексного узла, на самом деле объекты определяют разные имена для одного фрагмента дискового пространства.

[2] Каталог *root* в системе UNIX не имеет родителя. В этом каталоге элемент `..` соответствует тому же каталогу, что и элемент `.`, а именно самому каталогу *root*.

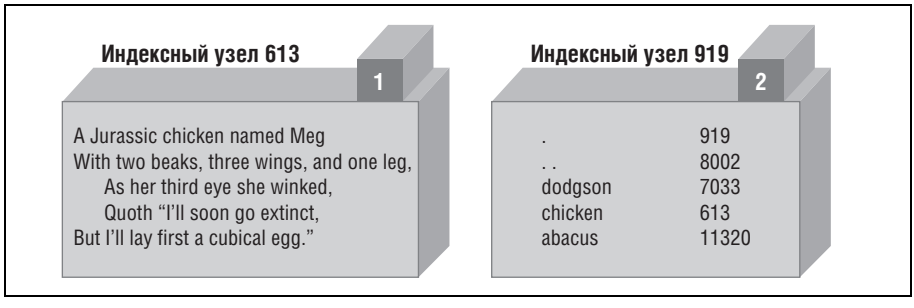


Рис. 13.1. *Индексные узлы в файловой системе UNIX*

для каждой записи в каталоге хранится номер соответствующего индексного узла.

Когда наступает момент создания нового файла в каталоге, система добавляет запись с именем файла и номером нового индексного узла. Но как система определяет, свободен ли тот или иной индексный узел? В каждом индексном узле хранится число, называемое *счетчиком ссылок*. Счетчик ссылок всегда равен 0, если узел не присутствует ни в одном каталоге, поэтому любой индексный узел с нулевым счетчиком ссылок свободен для размещения нового файла. При добавлении индексного узла в каталог счетчик ссылок увеличивается, а при удалении сведений о нем уменьшается. Для файла *chicken* из предыдущего примера счетчик 1 показан над данными индексного узла.

Но некоторые индексные узлы упоминаются в более чем одной записи каталогов. Например, мы уже видели, что каждая запись каталога включает `.`, ссылку на индексный узел самого каталога. Таким образом, счетчик ссылок каталога всегда содержит значение не менее 2: для его записи в родительском каталоге и записи в нем самом. Если каталог содержит подкаталоги, каждый из них тоже увеличивает счетчик ссылок, потому что каждый подкаталог содержит элемент `..`.[1] На рис. 13.1 значение счетчика ссылок 2 показано над данными индексного узла. Счетчик ссылок определяет количество полноценных имен индексного узла.[2] Могут ли ссылки на обычный файл присутствовать в нескольких записях каталогов? Конечно, могут. Допустим, в показанном выше каталоге функция Perl `link` используется для создания новой ссылки:

---

[1] Напрашивается предположение, что счетчик ссылок каталога всегда равен 2 + количество содержащихся в нем каталогов. В некоторых системах это действительно так, но другие системы работают по иным принципам.

[2] В выходных данных *ls -l* количество жестких ссылок на объект файловой системы отображается справа от флагов разрешений (строка вида `-rwxr-xr-x`). Теперь вы знаете, почему это число для каталогов всегда больше 1, а для обычных файлов почти всегда равно 1.

```
link "chicken", "egg"
  or warn "can't link chicken to egg: $!";
```

Результат будет таким же, как если бы вы ввели `"ln chicken egg"` в приглашении командного процессора UNIX. Если вызов `link` завершится успешно, функция возвращает *true*. В случае неудачи функция возвращает *false* и устанавливает переменную `$!`, которая включается в сообщение об ошибке. После выполнения фрагмента имя `egg` становится вторым именем файла *chicken*, и наоборот; ни одно имя не является более «настоящим», чем другое, и чтобы узнать, какое из них появилось первым, придется основательно постараться. На рис. 13.2 изображено новое состояние файловой системы с двумя ссылками на узел 613.

Итак, эти два имени ссылаются на одну область диска. Если файл *chicken* содержит 200 байт данных, то *egg* содержит те же 200 байт (а их суммарный объем составляет 200 байт, потому что это всего лишь один файл с двумя именами). Если добавить новую строку текста в файл *egg*, эта строка появится в конце *chicken*.[1] Если теперь пользователь случайно (или намеренно) удалит *chicken*, данные не потеряются – они все еще будут доступны под именем *egg*. И наоборот: после удаления *egg* у нас останется файл *chicken*. Конечно, при удалении обоих файлов данные пропадут.[2] Также для ссылок в содержимом каталогов установлено еще одно правило: номера индексных узлов

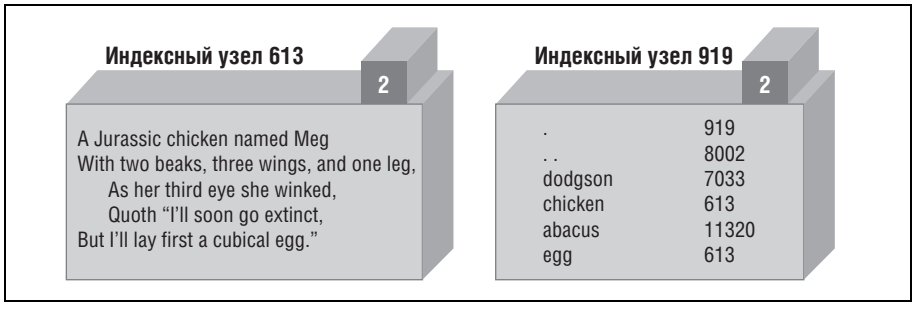


Рис. 13.2. *Создание ссылки egg на файл chicken*

[1] Если вам захочется поэкспериментировать с созданием ссылок и изменением текстовых файлов, учтите, что текстовые редакторы обычно не редактируют файл «на месте», а сохраняют измененную копию. Если пользователь изменит *egg* в текстовом редакторе, скорее всего, на диске появится новый файл *egg* и старый файл *chicken* – два разных файла вместо двух ссылок на один файл.

[2] Хотя система не всегда перезаписывает индексный узел немедленно, простых общих способов восстановления данных после обнуления счетчика ссылок не существует. А вы не забыли заархивировать свои данные?

в каталоге всегда относятся к одному смонтированному тому.[1] Это правило гарантирует, что при перенесении физического носителя (скажем, дискеты) на другой компьютер все каталоги по-прежнему будут связаны со своими файлами. Вот почему функция `rename` может использоваться для перемещения файла из одного каталога в другой, но только в том случае, если оба каталога находятся в одной файловой системе (смонтированном томе). Если бы файлы могли находиться на разных дисках, системе пришлось бы перемещать содержимое индексных узлов, а это слишком сложная операция для простого вызова системной функции.

Для ссылок устанавливается еще одно ограничение: они не могут использоваться для создания новых имен каталогов. Это объясняется тем, что каталоги объединены в иерархию. Если нарушить эту иерархию, служебные программы, вроде *find* или *pwd*, потеряются в своих блужданиях по файловой системе.

Итак, ссылки не могут создаваться для каталогов и не могут вести с одного смонтированного тома на другой. К счастью, эти ограничения обходятся при помощи другой разновидности: *символических ссылок*.[2] Символические ссылки (также называемые *мягкими ссылками*, в отличие от *жестких* ссылок, о которых мы говорили ранее) представляют собой специальные записи в каталоге, которые перенаправляют систему к другому месту. Допустим, пользователь создает в каталоге *poems* из предыдущего примера символическую ссылку; для этого он использует функцию Perl `symlink`:

```
symlink "dodgson", "carroll"
  or warn "can't symlink dodgson to carroll: $!";
```

Происходит практически то же, что произошло бы при выполнении команды *ln-s dodgson carroll* в командном процессоре. На рис. 13.3 показано новое состояние файловой системы с файлом в индексном узле 7033. Если пользователь захочет прочитать файл */home/barney/poems/carroll*, он получит те же данные, как если бы он открыл */home/barney/poems/dodgson* напрямую, потому что система автоматически переходит по символическим ссылкам. Однако новое имя не является «полноценным» именем файла, потому что (как видно из рисунка) счетчик ссылок для узла 7033 по-прежнему содержит 1. Символическая ссылка просто сообщает системе: «Если вы заглянули сюда в поисках *carroll*, теперь переходите к поискам *dodgson*».

В отличие от жестких ссылок, символические ссылки могут свободно выходить за границы смонтированных файловых систем или опреде-

---

[1] Единственное исключение – специальная запись `..` в каталоге *root* тома относится к каталогу, в котором смонтирован этот том.

[2] Очень старые системы UNIX не поддерживают символические ссылки, но в наши дни они встречаются крайне редко.

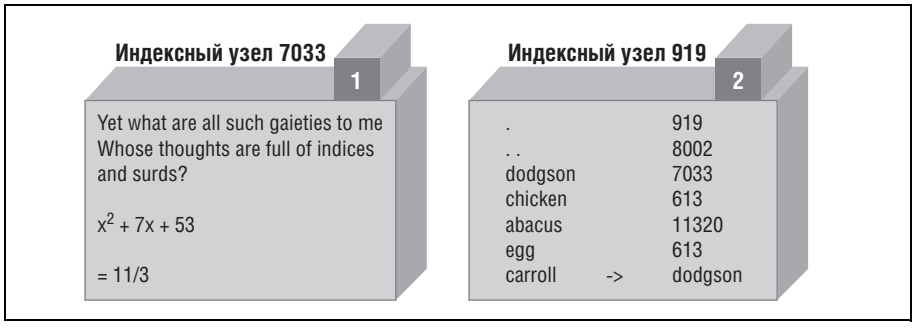


Рис. 13.3. *Символическая ссылка на индексный узел 7033*

лять новые имена для каталогов. Более того, символическая ссылка может указывать на любое имя в текущем или другом каталоге и даже на имя файла, который еще не существует! Но это также означает, что мягкие ссылки (в отличие от жестких) не могут предотвратить потерю данных, потому что они не учитываются в значении счетчика ссылок. Если пользователь удалит файл *dodgson*, система не сможет переходить по мягкой ссылке.[1] Запись *carroll* останется в каталоге, но попытка прочитать данные из нее приведет к ошибке вида «файл не найден». Проверка *-l 'carroll'* вернет *true*, а проверка *-e 'carroll'* вернет *false*: это символическая ссылка, но она указывает на несуществующий файл.

Так как мягкая ссылка может указывать на файл, который еще не существует, она также может использоваться при создании файла. Пользователь Барни хранит большинство своих файлов в домашнем каталоге */home/barney*, но ему также приходится часто обращаться к каталогу с длинным именем, которое так неудобно вводить: */usr/local/opt/system/httpd/root-dev/users/staging/barney/cgi-bin*. Он создает символическую ссылку */home/barney/my_stuff*, которая указывает на длинное имя, и теперь попасть в нужный каталог становится совсем легко. Если Барни создаст (из своего домашнего каталога) файл *my_stuff/bowling*, то настоящим именем файла будет */usr/local/opt/system/httpd/root-dev/users/staging/barney/cgi-bin/bowling*. На следующей неделе системный администратор перемещает все файлы в каталог */usr/local/opt/internal/httpd/www-dev/users/staging/barney/cgi-bin*, но Барни достаточно перенастроить одну символическую ссылку, и все его программы по-прежнему будут легко находить свои файлы.

Во многих системах */usr/bin/perl* или */usr/local/bin/perl* (или оба имени) являются символическими ссылками на «настоящий» двоичный файл Perl в вашей системе. Это упрощает переход на новую версию Perl.

---

[1] Конечно, удаление *carroll* приведет к потере только символической ссылки, но не оригинала.

Допустим, вы, будучи системным администратором, построили новую версию Perl. Конечно, старая версия все еще работает, и вы не хотите ничего портить. Когда все будет готово к переходу, достаточно переназначить одну-две символических ссылки; каждая программа, начинающаяся со строки `#!/usr/bin/perl`, будет автоматически использовать новую версию. Если с новой версией вдруг возникнут проблемы, восстановите старые ссылки, и старая версия Perl снова заработает, как положено. (Конечно, вы, как любой хороший администратор, заранее оповестили своих пользователей о необходимости тестирования кода с новой версией */usr/bin/perl-7.2*, а также о том, что во время «переходного периода» при необходимости они могут использовать старую версию, заменив первые строки в своих программах на `#!/usr/bin/perl-6.1`.)

Как ни странно, обе разновидности ссылок – и жесткие, и мягкие – чрезвычайно полезны. Во многих операционных системах, не входящих в семейство UNIX, ссылки вообще не поддерживаются, и их там весьма не хватает. В некоторых системах символические ссылки реализуются в виде «ярлыков» (shortcuts) или «псевдонимов» (aliases) – за дополнительной информацией обращайтесь к man-странице *perlport*.

Чтобы узнать, на какой объект указывает символическая ссылка, воспользуйтесь функцией `readlink`. Функция вернет либо информацию о целевом объекте, либо `undef`, если аргумент не является символической ссылкой:

```
my $where = readlink "carroll";             # Получаем "dodgson"

my $perl = readlink "/usr/local/bin/perl";  # Вероятно, сообщит,
                                            # где находится perl
```

Обе разновидности ссылок удаляются функцией `unlink`. Функция просто удаляет запись каталога, связанную с именем файла, с уменьшением счетчика ссылок и, возможно, с освобождением индексного узла.

Создание и удаление каталогов

Создать новый каталог в существующем каталоге несложно. Просто вызовите функцию `mkdir`:

```
mkdir "fred", 0755 or warn "Cannot make fred directory: $!";
```

И снова при успешном создании возвращается *true*, а при неудаче задается переменная `$!`.

Но что означает второй параметр, 0755? Он определяет начальные разрешения[1] для созданного каталога (разрешения всегда можно изменить позднее). Значение задается в восьмеричном формате, потому что оно будет интерпретироваться в стандартном формате UNIX с группа-

[1] Фактические разрешения, как обычно, зависят от значения `umask`. За дополнительной информацией обращайтесь к man-странице `umask(2)`.

ми из 3 бит, а восьмеричные значения удобны для представления такой группировки. Да, даже в Windows и MacPerl для использования функции mkdir необходимо кое-что знать о разрешениях UNIX. Режим 0755 предоставляет вам полный набор разрешений, а всем остальным пользователям – доступ только для чтения (без возможности что-либо изменить).

Функция mkdir не требует, чтобы значение задавалось в восьмеричном формате – она просто берет передаваемое число (литерал или результат вычисления). Но если вы не сможете быстро прикинуть, что восьмеричное число 0755 соответствует десятичному 493, вычисления лучше поручить Perl. А если нечаянно опустить начальный нуль, вы получите десятичное число 755, то есть 1363 в восьмеричной записи – довольно странная комбинация разрешений.

Как было показано в главе 2, строковое значение, используемое как число, никогда не интерпретируется в восьмеричной системе счисления, даже если оно начинается с 0. Следовательно, такая запись работать не будет:

```
my $name = "fred";
my $permissions = "0755";  # Не получится
mkdir $name, $permissions;
```

Кажется, мы только что создали каталог со странными разрешениями 01363 из-за того, что число 0755 было случайно интерпретировано как десятичное. Чтобы избежать подобных неприятностей, используйте функцию oct, которая обеспечивает восьмеричную интерпретацию строки независимо от присутствия начального нуля:

```
mkdir $name, oct($permissions);
```

Конечно, если значение задается прямо в программе, используйте число вместо строки. Необходимость в дополнительной функции oct чаще всего возникает при получении данных от пользователя. Предположим, аргументы передаются в командной строке:

```
my ($name, $perm) = @ARGV;  # Первые два аргумента - имя и разрешения
mkdir $name, oct($perm) or die "cannot create $name: $!";
```

Значение $perm изначально задается как строка, поэтому функция oct обеспечивает правильную восьмеричную интерпретацию.

Пустые каталоги удаляются функцией rmdir. Функция отчасти напоминает unlink, но может удалять только один каталог при каждом вызове:

```
foreach my $dir (qw(fred barney betty)) {
  rmdir $dir or warn "cannot rmdir $dir: $!\n";
}
```

Если каталог не пуст, попытка вызова rmdir завершается неудачей. Сначала следует попытаться удалить содержимое каталога функцией unlink, а затем попробуйте удалить каталог, который к этому моменту

должен быть пуст. Предположим, нам необходимо место для записи временных файлов в ходе выполнения программы:

```
my $temp_dir = "/tmp/scratch_$$";       # Определяется по идентификатору
                                        # процесса; см. в тексте
mkdir $temp_dir, 0700 or die "cannot create $temp_dir: $!";
...
# Каталог $temp_dir используется для хранения всех временных файлов
...
unlink glob "$temp_dir/* $temp_dir/.*"; # Удалить содержимое $temp_dir
rmdir $temp_dir;                        # Удалить пустой каталог
```

В имя каталога для временных файлов включается уникальный идентификатор текущего процесса, для получения которого используется переменная `$$` (по аналогии с командным процессором). Мы делаем это для предотвращения конфликтов с другими процессами при условии, что их идентификаторы тоже будут включаться в имя каталога. (На практике наряду с идентификатором процесса часто используется имя программы; если программа называется `quarry`, каталог принимает вид `/tmp/quarry_$$`.)

При завершении работы программы последний вызов `unlink` должен удалить все файлы из временного каталога, после чего функция `rmdir` удалит пустой каталог. Но если в каталоге были созданы подкаталоги, для них вызов `unlink` завершится неудачей, и общий вызов `rmdir` тоже не сработает. Если вам потребуется более надежное решение, поинтересуйтесь функцией `rmtree` из модуля `File::Path` из стандартной поставки Perl.

Изменение разрешений

Команда UNIX *chmod* изменяет разрешения доступа для файла или каталога. В Perl эта задача решается функцией `chmod`:

```
chmod 0755, "fred", "barney";
```

Как и многие функции операционной системы, `chmod` возвращает количество успешно измененных файлов, а при вызове с одним аргументом задает переменной `$!` значение, удобное для вывода сообщений об ошибках. Первый параметр определяет разрешения UNIX (даже в версиях Perl, предназначенных для других систем). По причинам, изложенным ранее при описании `mkdir`, значение обычно задается в восьмеричной системе.

Символические обозначения разрешений (`+x` или `go=u-w`), поддерживаемые командой UNIX *chmod*, для функции `chmod` недействительны.[1]

[1] Если только вы не установите и не будете использовать модуль `File::chmod` из CPAN, который расширяет оператор `chmod` поддержкой символических обозначений.

Смена владельца

Если операционная система предоставляет такую возможность, вы можете сменить владельца (пользователя и группу) для списка файлов (или файловых дескрипторов) при помощи функции chown. Пользователь и группа изменяются одновременно, и в обоих параметрах должны передаваться соответствующие числовые значения. Пример:

```
my $user = 1004;
my $group = 100;
chown $user, $group, glob "*.o";
```

А если вы предпочитаете использовать вместо числа имя пользователя (например, merlyn)? Легко. Вызовите функцию getpwnam, чтобы преобразовать имя пользователя в числовой код, а затем функцию getgrnam[1], чтобы преобразовать название группы в число:

```
defined(my $user = getpwnam "merlyn") or die "bad user";
defined(my $group = getgrnam "users") or die "bad group";
chown $user, $group, glob "/home/merlyn/*";
```

Функция defined проверяет, что возвращаемое значение отлично от undef (возвращается в том случае, если код пользователя или группы недействителен).

Функция chown возвращает количество файлов и задает переменную $! при обнаружении ошибки.

Изменение временных меток

В тех редких случаях, когда вам нужно «соврать» другой программе относительно времени последнего обращения или модификации файла, функция utime придет вам на помощь. В первых двух аргументах передается новое время последнего обращения и модификации, а остальные аргументы содержат списки имен файлов, которым назначаются эти временные метки. Время задается во внутреннем формате (значения, возвращаемые функцией stat – см. главу 12).

Одно из характерных значений временных меток – «прямо сейчас» (то есть текущий момент времени) – возвращается в нужном формате функцией time. Допустим, вы хотите, чтобы все файлы в текущем каталоге выглядели так, словно они были созданы сутки назад, а первое обращение к ним произошло только сейчас. Это делается так:

```
my $now = time;
my $ago = $now - 24 * 60 * 60;  # Количество секунд в сутках
```

[1] Это едва ли не самые уродливые имена функций, известные человечеству. Но не стоит винить Ларри; он всего лишь присвоил функциям имена, придуманные парнями из Беркли.

```
utime $now, $ago, glob "*";      # Время последнего обращения - сейчас,
                                 # модификация - сутки назад
```

Разумеется, ничто не мешает создать файл с временной меткой, обозначающей произвольный момент в будущем или прошлом (в пределах диапазона временных меток UNIX от 1970 до 2038 года или диапазона вашей системы, если только в ней не используются 64-разрядные метки). Например, вы можете использовать эту возможность для создания каталога, в котором будут храниться черновики вашего романа о путешествиях во времени.

Третьей временной метке (`ctime`) при любых изменениях файла всегда задается текущее время, и изменить ее функцией `ctime` невозможно (она вернется к текущему времени сразу же после изменения). Дело в том, что метка `ctime` предназначена для инкрементной архивации данных: если атрибут `ctime` файла новее времени на ленте с архивом, файл необходимо заархивировать снова.

Упражнения

Эти программы потенциально опасны! Будьте осторожны и тестируйте их в пустых каталогах, чтобы предотвратить случайное удаление полезных данных.

Ответы к упражнениям приведены в приложении А.

1. [12] Напишите программу, которая запрашивает у пользователя имя каталога и делает его текущим каталогом. Если пользователь вводит строку из одних пропусков, по умолчанию программа должна перейти в домашний каталог пользователя. После смены каталога выведите его стандартное содержимое (за исключением элементов, имена которых начинаются с точки) в алфавитном порядке. (Подсказка: как это проще сделать – с дескриптором каталога или с глобом?) Если попытка перехода в другой каталог завершилась неудачей, сообщите об этом пользователю, но не пытайтесь выводить содержимое.
2. [4] Измените программу так, чтобы она выводила информацию обо всех файлах (в том числе и тех, имена которых начинаются с точки).
3. [5] Если вы использовали в предыдущем упражнении дескриптор каталога, перепишите программу с использованием глоба. Если программа была написана для глоба, перепишите ее для дескриптора каталога.
4. [6] Напишите программу, имитирующую команду *rm*; программа должна удалять файлы, имена которых вводятся в командной строке. (Поддерживать различные ключи *rm* не нужно.)
5. [10] Напишите программу, имитирующую команду *mv*; программа должна переименовывать источник, заданный первым аргументом

командной строки, в приемник, заданный вторым аргументом. (Поддерживать различные ключи *mv* или дополнительные аргументы не нужно.) Помните, что в качестве приемника может быть задан каталог; в этом случае в новом каталоге создается файл с исходным базовым именем.

6. [7] Если ваша операционная система поддерживает жесткие ссылки, напишите программу, имитирующую команду *ln*; программа должна создавать жесткую ссылку из первого аргумента командной строки на второй. (Поддерживать различные ключи *ln* или дополнительные аргументы не нужно.) Если в системе не поддерживаются жесткие ссылки, просто выведите сообщение о том, какая операция должна быть выполнена. (Подсказка: эта программа имеет нечто общее с предыдущей – если вы поймете, что именно, это сэкономит ваше время при кодировании.)
7. [7] Если ваша операционная система поддерживает жесткие ссылки, дополните программу из предыдущего упражнения так, чтобы в аргументах мог передаваться необязательный ключ *-s*; с этим ключом вместо жесткой ссылки должна создаваться мягкая ссылка. (Даже если жесткие ссылки в системе не поддерживаются, попробуйте создать хотя бы мягкую ссылку.)
8. [7] Если ваша операционная система поддерживает такую возможность, напишите программу для поиска символических ссылок в текущем каталоге и вывода их значений (по аналогии с тем, как это делает *ls -l*: `имя -> значение`).

14

Строки и сортировка

В начале книги мы уже упоминали о том, что Perl на 90% проектировался для работы с текстом и только на 10% для решения других задач. Поэтому вполне понятно, что Perl обладает мощными средствами обработки текста, включая уже рассмотренные нами регулярные выражения. Но в некоторых ситуациях возможности регулярных выражений оказываются чрезмерными, и для работы со строками желательно использовать более простые средства, рассмотренные в этой главе.

Поиск подстроки по индексу

Как найти подстроку? Зависит от того, где вы ее потеряли. Если подстрока затерялась в большей строке, считайте, вам повезло, потому что функция `index` поможет вам в поисках. Вот как это делается:

```
$where = index($big, $small);
```

Perl находит первое вхождение подстроки и возвращает целочисленный индекс первого символа. Нумерация индексов начинается с нуля – если подстрока находится в самом начале строки, функция `index` возвращает 0. Если подстрока смещена на один символ, функция возвращает 1, и т. д. Если найти подстроку не удалось, функция сообщает об этом, возвращая -1.[1] В следующем примере `$where` возвращает 6:

```
my $stuff = "Howdy world!";
my $where = index($stuff, "wor");
```

[1] Бывшие программисты C узнают функцию `index` из языка C. Программисты, продолжающие работать на C, тоже узнают ее, но к этой странице они уже должны перейти в разряд *бывших* программистов C.

При желании можно рассматривать возвращаемое значение index как количество символов, которые необходимо пропустить перед началом подстроки. Если переменная $where равна 6, значит, мы должны пропустить первые шесть символов $stuff, чтобы добраться до wor.

Функция index всегда возвращает позицию первого вхождения подстроки. Однако ей можно приказать начать поиск не от начала строки, а с более поздней позиции – для этого функции передается необязательный третий параметр:

```
my $stuff  = "Howdy world!";
my $where1 = index($stuff, "w");                # $where1 присваивается 2
my $where2 = index($stuff, "w", $where1 + 1);   # $where2 присваивается 6
my $where3 = index($stuff, "w", $where2 + 1);   # $where3 присваивается -1
                                                # (подстрока не найдена)
```

(Конечно, обычно многократный поиск подстроки осуществляется в цикле.) Третий параметр фактически определяет минимальное возвращаемое значение; если найти подстроку в этой позиции и далее не удается, функция возвращает -1.

В некоторых ситуациях требуется найти не первое, а *последнее* вхождение подстроки.[1] Эта задача решается функцией rindex. В следующем примере функция ищет последнее вхождение /, которое обнаруживается в позиции 4:

```
my $last_slash = rindex("/etc/passwd", "/");  # Значение равно 4
```

Функция rindex также имеет необязательный третий параметр, но в этом случае он определяет *максимальное* допустимое возвращаемое значение:

```
my $fred = "Yabba dabba doo!";
my $where1 = rindex($fred, "abba");  # $where1 gets 7
my $where2 = rindex($fred, "abba", $where1 - 1);  # $where2 присваивается 1
my $where3 = rindex($fred, "abba", $where2 - 1);  # $where3 присваивается -1
```

Операции с подстроками и функция substr

Функция substr работает с подмножеством символов большей строки. Синтаксис вызова выглядит так:

```
$part = substr($string, $initial_position, $length);
```

Функция получает три аргумента: строку, отсчитываемую от нуля начальную позицию (по аналогии с возвращаемым значением index)

[1] В действительности Perl не перебирает все предыдущие вхождения – функция начинает поиск с другого конца строки и возвращает первую найденную позицию, что приводит к тому же результату. Конечно, функция возвращает тот же индекс, отсчитываемый от нуля, который используется для описания позиции подстроки.

и длину подстроки. Возвращаемое значение представляет собой заданную подстроку:

```
my $mineral = substr("Fred J. Flintstone", 8, 5);  # "Flint"
my $rock = substr "Fred J. Flintstone", 13, 1000;  # "stone"
```

Как видно из предыдущего примера, при выходе длины за пределы строки (в этом случае 1000 символов) Perl не жалуется, а просто возвращает строку максимально возможной длины. Впрочем, если вы хотите, чтобы подстрока завершалась в конце основной строки (независимо от ее фактической длины), опустите при вызове третий параметр (длину):

```
my $pebble = substr "Fred J. Flintstone", 13;  # "stone"
```

Начальная позиция подстроки может задаваться отрицательным числом; в этом случае она отсчитывается от конца строки (то есть позиция -1 соответствует последнему символу).[1] В следующем примере позиция -3 соответствует трем символам от конца строки, то есть букве `i`:

```
my $out = substr("some very long string", -3, 2);  # $out присваивается "in"
```

Как и следовало ожидать, `index` и `substr` удобно использовать совместно. В этом примере из строки извлекается подстрока, начинающаяся с позиции буквы `l`:

```
my $long = "some very very long string";
my $right = substr($long, index($long, "l") );
```

А теперь самое интересное: если строка хранится в переменной[2], выбранную часть строки можно изменить:

```
my $string = "Hello, world!";
substr($string, 0, 5) = "Goodbye";  # $string содержит "Goodbye, world!"
```

Как видите, присваиваемая (под)строка не обязана совпадать по длине с заменяемой подстрокой. Длина полученной строки изменяется в соответствии с длиной замены. А если и это не произвело на вас впечатление, оператор привязки (`=~`) позволяет ограничить операцию частью строки. В следующем примере подстрока `fred` заменяется подстрокой `barney` в пределах 20 последних символов:

---

[1] По аналогии с индексами массивов в главе 3. Как вы уже знаете, массивы могут индексироваться либо от 0 (первый элемент) по возрастанию, либо от -1 (последний элемент) по убыванию. Точно так же вхождения подстрок индексируются от позиции 0 (первый символ) по возрастанию либо от позиции -1 (последний символ) по убыванию.

[2] Формально она может представлять собой любое *левостороннее значение* (lvalue). Точный смысл этого термина выходит за рамки книги; считайте, что это любое выражение, которое может стоять слева от знака `=` при скалярном присваивании. Обычно это переменная, но как видно из примера, левосторонним значением может быть даже вызов оператора `substr`.

```
substr($string, -20) =~ s/fred/barney/g;
```

По правде говоря, нам никогда не приходилось использовать эти возможности в своем коде. Скорее всего, вам они тоже не понадобятся. Но всегда приятно сознавать, что Perl может сделать больше, чем вам когда-либо понадобится, не так ли?

Почте все, что могут сделать `substr` и `index`, также можно сделать с использованием регулярных выражений. Используйте регулярные выражения там, где это уместно. Но `substr` и `index` часто работают быстрее, потому что они требуют меньших затрат ресурсов, чем полноценное ядро регулярных выражений: они не поддерживают поиск без учета регистра символов, не имеют метасимволов, о которых нужно беспокоиться, и не заполняют переменные частичными совпадениями.

Кроме присваивания функции `substr` (которое на первый взгляд смотрится немного необычно), вы также можете использовать `substr` более традиционным[1] способом – с четырьмя аргументами, в которых четвертый аргумент задает подстроку замены:

```
my $previous_value = substr($string, 0, 5, "Goodbye");
```

Функция возвращает предыдущее значение, хотя вы, как обычно, можете использовать эту функцию в пустом контексте, если результат вас не интересует.

Форматирование данных функцией sprintf

Функция `sprintf` получает те же аргументы, что и `printf` (кроме необязательного файлового дескриптора, конечно), но не выводит, а возвращает полученную строку. Например, это позволяет сохранить отформатированную строку в переменной для использования в будущем, либо вам недостаточно возможностей управления выводом, предоставляемых функцией `printf`:

```
my $date_tag = sprintf
  "%4d/%02d/%02d %2d:%02d:%02d",
  $yr, $mo, $da, $h, $m, $s;
```

В переменной `$date_tag` сохраняется значение вида `"2038/01/19 3:00:08"`. Форматная строка (первый аргумент `sprintf`) содержит начальные нули в некоторых кодах форматов; при описании форматов `printf` в главе 5 об этой возможности ничего не говорилось. Начальный нуль в коде формата означает, что число должно дополняться начальными нулями до заданной ширины. Без начальных нулей полученная строка даты/времени содержала бы нежелательные начальные пробелы вместо нулей: `"2038/ 1/19 3: 0: 8"`.

---

[1] Под «традиционностью» мы имеем в виду способ вызова функции, а не поддержку в Perl – эта возможность появилась относительно недавно.

Использование функции sprintf для вывода денежных сумм

Функция `sprintf` часто применяется для форматирования чисел с определенным количеством знаков в дробной части. Например, денежные суммы должны выводиться в виде 2.50, но не 2.5 – и конечно, не 2.49997! Задача легко решается при помощи формата `"%.2f"`:

```
my $money = sprintf "%.2f", 2.49997;
```

При округлении чисел необходимо учитывать множество тонкостей, но в большинстве случаев числа могут храниться в памяти с максимальной точностью, а округляется только вывод.

Если выводимое число достаточно велико, для разделения разрядов запятыми (например, если это денежная сумма) можно создать удобную пользовательскую функцию:[1]

```
sub big_money {
 my $number = sprintf "%.2f", shift @_;
 # При каждой итерации цикла добавляется одна запятая
 1 while $number =~ s/^(-?\d+)(\d\d\d)/$1,$2/;
 # Добавляем знак доллара в нужную позицию
 $number =~ s/^(-?)/$1\$/;
 $number;
}
```

В этой функции используются некоторые возможности, которые мы еще не рассматривали, но они логически следуют из того, что было показано ранее. Первая строка функции форматирует первый (и единственный) параметр так, чтобы он содержал ровно две цифры в дробной части. Иначе говоря, если параметр содержит число 12345678.9, в `$number` сохраняется строка `"12345678.90"`.

В следующей строке кода используется модификатор `while`. Как упоминалось ранее при описании модификаторов в главе 10, эту строку всегда можно переписать в виде традиционного цикла `while`:

```
while ($number =~ s/^(-?\d+)(\d\d\d)/$1,$2/) {
  1;
}
```

Что здесь происходит? Тело цикла выполняется, пока замена возвращает истинное значение (признак успешного выполнения). Но тело цикла не делает ничего! Для Perl это вполне допустимо, а нам эта запись сообщает, что целью этой команды является выполнение условия (замена), а не бесполезное тело цикла. Значение 1 традиционно используется как условный заполнитель, хотя подойдет и любое другое

[1] Да, мы знаем, что не во всех странах группы разрядов разделяются запятыми, разряды не всегда группируются по три, а знак денежной единицы порой отличается от $. Все равно это хороший пример!

значение.[1] Следующая запись работает точно так же, как и приведенный ранее цикл:

```
'keep looping' while $number =~ s/^(-?\d+)(\d\d\d)/$1,$2/;
```

Итак, теперь мы знаем, что цикл создан для выполнения замены. Но что делает замена? Напоминаем, что `$number` на этой стадии содержит строку вида `"12345678.90"`. Шаблон совпадает с первой частью строки, но не может пройти дальше точки. (А вы видите, почему не может?) В переменную `$1` заносится значение `"12345"`, а в переменную `$2` – значение `"678"`, так что замена преобразует `$number` в `"12345,678.90"` (еще раз: совпадение для точки не находится, поэтому завершающая часть строки остается без изменений).

А вы видите, что делает дефис в начале шаблона? (Подсказка: дефис разрешен только в одном месте строки.) Если не догадаетесь сами, мы расскажем в конце раздела.

Однако мы еще не закончили с командой замены. Так как замена завершилась успешно, фиктивный цикл возвращается для новой итерации. На этот раз возможные совпадения шаблона должны находиться до запятой, поэтому `$number` преобразуется в `"12,345,678.90"`. Таким образом, замена включает запятую в число при каждой итерации цикла.

Но цикл еще не завершен. Предыдущая попытка замены была успешной, поэтому происходит очередная итерация цикла. Но на этот раз шаблон не совпадает, потому что в начале строки должно быть не менее четырех цифр; на этом цикл завершается.

Почему мы не могли воспользоваться модификатором `/g`, чтобы выполнить глобальный поиск с заменой и избавиться от хлопот с `1 while`? Это невозможно, так как мы перемещаемся назад от точки, а не вперед от начала строки. Расставить запятые в таких числах одной лишь подстановкой `s///g` невозможно.[2] Кстати, вы поняли, зачем нужен дефис? Знак «-» в начале строки – это необязательный префикс. Он присутствует и в следующей строке, где знак `$` выводится в нужной позиции, так что `$number` принимает вид `"$12,345,678.90"` (или, возможно, `"-$12,345,678.90"`, если число отрицательное). Обратите внимание: знак доллара не всегда является первым символом в строке (хотя это бы значительно упростило ее). Наконец, последняя строка возвращает отформатированную «денежную сумму» для вывода в ежегодном отчете.

---

[1] Хотя никакой реальной пользы от него не будет. Кстати, если вам интересно, Perl исключает константу из цикла в ходе оптимизации, поэтому никакие дополнительные ресурсы на ее обработку не потребуются.

[2] По крайней мере, без использования дополнительных возможностей регулярных выражений, которые мы вам еще не показывали. Из-за разработчиков Perl становится все труднее писать книги о Perl, содержащие слово «невозможно».

Расширенная сортировка

В главе 3 мы показали, что список можно отсортировать в ASCII-алфавитном порядке по возрастанию встроенным оператором `sort`. А если вам нужна числовая сортировка? Или сортировка без учета регистра символов? А может, вы хотите отсортировать данные на основании информации, хранящейся в хеше? Perl предоставляет возможность отсортировать список в любом желательном порядке; до конца главы вы увидите немало примеров.

Информация о порядке сортировки передается Perl в виде *функции сортировки*. Возможно, после начального курса информатики слова «функция сортировки» вызывают у вас кошмарные видения пузырьковой сортировки, сортировки по Шеллу и быстрой сортировки; вы качаете головой и говорите: «Нет, ни за что!» Не беспокойтесь, все не так страшно. Perl уже умеет сортировать списки; он просто не знает, какой порядок вам нужен. Функция сортировки всего лишь укажет ему, как это делать.

Почему это необходимо? Если подумать, сортировка сводится к размещению элементов в определенном порядке, основанном на результате их сравнения. Сравнить все элементы сразу невозможно, поэтому сравнение должно осуществляться попарно. В конечном итоге информация, полученная об относительном порядке для каждой пары, используется для определения общего порядка в отсортированном списке. Perl умеет выполнять все эти действия, но не знает, как бы вы хотели сравнивать элементы. Вам остается лишь написать соответствующую функцию.

Функция сортировки не должна заниматься сортировкой многих элементов. От нее требуется лишь уметь сравнивать два элемента. Если вы расположите два элемента в нужном порядке, Perl сможет определить (многократно вызывая функцию сортировки) конечный порядок сортируемых данных.

Функция сортировки определяется как (почти) обычная пользовательская функция. Она вызывается многократно для разных пар элементов сортируемого списка.

Представьте, что вы пишете функцию, которая получает два сортируемых параметра. Вероятно, первый вариант этой функции будет выглядеть примерно так:

```
sub any_sort_sub {  # На самом деле этого не происходит
  my($a, $b) = @_;  # Получить параметры и присвоить их переменным
  # Начинаем сравнивать $a и $b
  ...
}
```

Но функция сортировки будет вызываться снова и снова, нередко по несколько тысяч раз. Объявление переменных `$a` и `$b` занимает немно-

го времени, как и присваивание им значений в начале функции, но умножьте на тысячи вызовов функции, и они окажут заметное влияние на общую скорость выполнения.

Конечно, нам бы этого не хотелось. (Но даже если вы поступите подобным образом, такое решение работать не будет.) Все выглядит так, словно Perl уже выполнил присваивание за вас еще до начала выполнения кода функции. Вы пишете функцию сортировки без первой строки; переменные $a и $b уже содержат нужные значения. В начале функции сортировки в них хранятся два элемента исходного списка.

Функция возвращает код, определяющий результат сравнения элементов (по аналогии с функцией C `qsort(3)`, но здесь используется собственная внутренняя реализация сортировки Perl). Если значение $a должно предшествовать $b в итоговом списке, функция сортировки сообщает об этом, возвращая значение -1. Если значение $b должно предшествовать $a, функция возвращает 1.

Если порядок $a и $b неважен, функция возвращает 0. Как такое может быть? Допустим, сортировка выполняется без учета регистра символов и сравниваются две строки `fred` и `Fred`... Или при выполнении числовой сортировки сравниваются два одинаковых числа.

Простейшая функция числовой сортировки выглядит так:

```
sub by_number {
  # Функция сортировки, переменные $a и $b уже подготовлены
  if ($a < $b) { -1 } elsif ($a > $b) { 1 } else { 0 }
}
```

Чтобы использовать эту функцию при сортировке, укажите ее имя (без префикса &) между ключевым словом `sort` и сортируемым списком. В следующем примере отсортированный список чисел помещается в `@result`:

```
my @result = sort by_number @some_numbers;
```

Обратите внимание: в функции сортировки мы не пытаемся объявлять переменные $a and $b или задавать их значения – если это сделать, функция будет работать неправильно. Мы просто поручаем Perl задать $a и $b, а нам остается лишь написать функцию сравнения.

В действительности функцию можно сделать еще проще (и еще эффективнее). Поскольку такие трехсторонние сравнения встречаются достаточно часто, в Perl предусмотрена удобная сокращенная форма для их записи: оператор <=>. Этот оператор сравнивает два числа и возвращает -1, 0 или 1, необходимые для их числовой сортировки. Таким образом, ту же функцию сортировки можно записать в упрощенном виде:

```
sub by_number { $a <=> $b }
```

Если оператор <=> сравнивает числа, нетрудно предположить, что и для строк существует свой оператор трехстороннего сравнения: `cmp`. Эти

два оператора легко запоминаются и просты в использовании. Оператор <=> напоминает числовые операторы вида >=, но состоит из трех символов вместо двух, потому что может возвращать три возможных значения вместо двух. А оператор cmp похож на операторы сравнения строк вроде ge, но состоит из трех символов вместо двух по тем же причинам.[1] Конечно, cmp сам по себе обеспечивает тот же порядок, что и сортировка по умолчанию. Функцию, которая всего лишь обеспечивает стандартный порядок сортировки, даже не нужно писать:[2]

```
sub ASCIIbetically { $a cmp $b }

my @strings = sort ASCIIbetically @any_strings;
```

Однако cmp может использоваться для определения более сложного порядка сортировки – скажем, сортировки без учета регистра символов:

```
sub case_insensitive { "\L$a" cmp "\L$b" }
```

В этом случае строка из $a (приведенная к нижнему регистру) сравнивается со строкой из $b (приведенной к нижнему регистру); в результате мы получаем сортировку без учета регистра.

Обратите внимание: сами элементы при этом не изменяются, мы всего лишь используем их значения. Это очень важно: по соображениям эффективности $a и $b не являются копиями элементов данных. Они представляют собой временные псевдонимы для элементов исходного списка, а их изменение приведет к повреждению исходных данных. Никогда так не делайте.

Если ваша функция сортировки не сложнее, чем функции из наших примеров (а обычно так оно и есть), программу можно дополнительно упростить: замените имя функции сортировки «встроенным» фрагментом кода:

```
my @numbers = sort { $a <=> $b } @some_numbers;
```

В современных Perl-программах отдельные функции сортировки почти не встречаются. Обычно они записываются во встроенном виде, как в нашем примере.

Допустим, список необходимо отсортировать в числовом порядке по убыванию. Задача легко решается при помощи функции reverse:

```
my @descending = reverse sort { $a <=> $b } @some_numbers;
```

Но мы покажем еще один красивый трюк. Операторы сравнения (<=> и cmp) «близоруки»; они смотрят не на то, какой операнд хранится в $a

---

[1] И это не случайно. Ларри намеренно принимает подобные решения, чтобы Perl было проще изучать и запоминать. Не забывайте, что в душе он лингвист, поэтому ему известно, что люди думают об изучении языков.

[2] Если, конечно, вы не пишете учебник Perl для начинающих, в котором она приводится в качестве примера.

или в $b, а на то, где он стоит – слева или справа. Таким образом, если переставить $a и $b местами, оператор сравнения будет каждый раз получать обратный результат. Следовательно, числовую сортировку по убыванию можно выполнить и таким способом:

```
my @descending = sort { $b <=> $a } @some_numbers;
```

После небольшой тренировки такие конструкции легко читаются «с листа». Элементы упорядочиваются по убыванию (потому что $b предшествует $a), при этом используется числовое сравнение (оператор <=> вместо cmp). Получается, что встроенная функция сортирует числа в обратном порядке. (В современных версиях Perl выбор варианта не влияет на эффективность; reverse распознается как модификатор sort. Компилятор выбирает «короткий путь», чтобы список не пришлось сортировать в одну сторону только для того, чтобы немедленно развернуть его в обратном направлении.)

Сортировка хеша по значениям

Вскоре вы освоите сортировку списков и окажетесь в ситуации, когда хеш требуется отсортировать по значению. Допустим, трое уже знакомых персонажей отправились в боулинг, а их результаты хранятся в следующем хеше. Требуется вывести элементы списка в правильном порядке, начиная с победителя, так чтобы хеш был отсортирован по набранным очкам:

```
my %score = ("barney" => 195, "fred" => 205, "dino" => 30);
my @winners = sort by_score keys %score;
```

Конечно, мы вовсе не собираемся сортировать хеш по значениям; это всего лишь фигура речи. Хеш вообще нельзя отсортировать! Но в предыдущих примерах использования sort с хешами фактически выполнялась сортировка ключей хеша (в ASCII-алфавитном порядке). Сейчас мы опять собираемся отсортировать ключи хеша, но порядок сортировки определяется соответствующими значениями. Результат должен выглядеть как список имен, упорядоченных по набранным очкам.

Написать функцию сортировки несложно. Все, что для этого нужно, – использовать числовое сравнение значений вместо строкового сравнения имен. Иначе говоря, вместо $a и $b (имена игроков) следует сравнивать $score{$a} и $score{$b} (набранные очки). Если взглянуть на задачу с этой точки зрения, программа пишется сама собой:

```
sub by_score { $score{$b} <=> $score{$a} }
```

Давайте разберемся, как она работает. Допустим, при первом вызове $a содержит barney, а $b – fred. Значит, функция выполняет сравнение $score{"fred"} <=> $score{"barney"}, то есть (как видно из хеша) 205 <=> 195. Вспомните о «близорукости» оператора <=>: увидев 205 перед 195, он фактически говорит: «Нет, это неправильный числовой порядок;

значение $b должно идти перед $a». В результате он сообщает Perl, что строка fred должна предшествовать barney.

При следующем вызове функции $a снова содержит barney, а $b — dino. На этот раз числовое сравнение видит выражение 30 <=> 195 и сообщает, что элементы следуют в правильном порядке; $a действительно предшествует $b (то есть barney предшествует dino). На этой стадии Perl располагает достаточной информацией для упорядочения списка: fred — победитель, на втором месте barney и на третьем — dino.

Почему в сравнении $score{$b} предшествует $score{$a}, а не наоборот? Потому что мы хотим упорядочить результаты игры по убыванию, от максимума к минимуму. Вскоре (опять же после небольшой тренировки) вы тоже научитесь читать такие конструкции «с листа»: $score{$b} <=> $score{$a} означает, что сортировка будет выполняться по набранным очкам, в числовом порядке и по убыванию.

Сортировка по нескольким ключам

Мы забыли о четвертом игроке, который отправился в боулинг с первыми тремя. На самом деле содержимое хеша выглядит так:

```
my %score = (
  "barney" => 195, "fred" => 205,
  "dino" => 30, "bamm-bamm" => 195,
);
```

Как видите, bamm-bamm набрал одинаковое количество очков с barney. Кто же из них должен стоять первым в отсортированном списке игроков? Неизвестно, потому что оператор сравнения (обнаружив одинаковые значения с обеих сторон) вернет нуль при проверке этой пары.

Возможно, это несущественно, и все же желательно, чтобы сортировка осуществлялась в четко определенном, предсказуемом порядке. Если несколько игроков имеют одинаковое количество очков, они должны находиться вместе в списке. Однако внутри этой группы имена должны быть упорядочены в ASCII-алфавитном порядке. Но как выразить это в функции сортировки? И снова задача решается относительно просто:

```
my @winners = sort by_score_and_name keys %score;

sub by_score_and_name {
  $score{$b} <=> $score{$a}  # Числовая сортировка по убыванию
                             # набранных очков
    or
  $a cmp $b                  # ASCII-алфавитная сортировка по именам
}
```

Как работает это решение? Когда оператор <=> видит два разных счета, использоваться должен именно этот критерий сравнения. Оператор возвращает -1 или 1 (истинное значение), низкоприоритетная ускорен-

ная обработка or пропускает оставшуюся часть выражения, и функция выдает нужный результат (напомним, что ускоренная обработка or возвращает последнее вычисленное выражение). Но если оператор <=> видит два одинаковых счета, он возвращает 0 (ложное значение). Управление передается оператору cmp, который возвращает нужный результат со сравнением ключей в строковом виде. Иначе говоря, в случае совпадения счетов «ничья» разрешается посредством сравнения строк.

Мы знаем, что при таком использовании функция сортировки by_score_and_name никогда не вернет 0. (А вы видите, почему? Ответ в сноске[1].) Итак, мы знаем, что порядок сортировки всегда четко определен; сегодняшний результат полностью идентичен тому, который мы получим завтра с теми же данными.

Конечно, ваша функция сортировки не ограничивается всего двумя уровнями. Ниже приведена программа для библиотеки, упорядочивающая список читателей с сортировкой по пяти уровням. Данные сортируются по величине штрафов за просрочку (вычисляемых функцией %fine, которая здесь не приводится), количеству взятых книг (из хеша %items), полным именам (сначала фамилия, затем имя – и то, и другое берется из хешей) и, наконец, по идентификаторам, если все остальные атрибуты совпадают:

```
@patron_IDs = sort {
  &fines($b) <=> &fines($a) or
  $items{$b} <=> $items{$a} or
  $family_name{$a} cmp $family_name{$a} or
  $personal_name{$a} cmp $family_name{$b} or
  $a <=> $b
} @patron_IDs;
```

Упражнения

Ответы к упражнениям приведены в приложении А.

1. [10] Напишите программу, которая читает список чисел и сортирует их по возрастанию. Полученный список выводится в столбец с выравниванием по правому краю. Опробуйте программу на следующих тестовых данных или воспользуйтесь файлом *numbers* с сайта O'Reilly (см. предисловие):

```
17 1000 04 1.50 3.14159 -10 1.5 4 2001 90210 666
```

2. [15] Напишите программу, которая выводит данные следующего хеша отсортированными по фамилиям (значения хеша) в алфавит-

[1] Функция может вернуть 0 только в одном случае: если две строки будут идентичны. Но так как строки являются ключами хеша, они заведомо различны. Конечно, если передать sort список с дубликатами, она вернет 0 при их сравнении, но мы передаем список ключей хеша.

ном порядке без учета регистра символов. Если фамилии совпадают, проведите дополнительную сортировку по именам (ключи хеша) – также без учета регистра символов. На первом месте в выходных данных должен стоять элемент с ключом `Fred`, а на последнем – элемент с ключом `Betty`. Все люди с одинаковыми фамилиями должны быть сгруппированы. Не изменяйте данные; они должны выводиться в том же регистре, что и в примере. (Исходный код создания подобного хеша содержится в файле *sortable_hash*, также доступном для загрузки.)

```
my %last_name = qw{
  fred flintstone Wilma Flintstone Barney Rubble
  betty rubble Bamm-Bamm Rubble PEBBLES FLINTSTONE
};
```

3. [15] Напишите программу, которая ищет в заданной строке все вхождения заданной подстроки и выводит их позиции. Например, для входной строки `"This is a test."` и подстроки `"is"` программа должна вывести позиции 2 и 5, а для подстроки `"a"` – позицию 8. А что выведет программа для подстроки `"t"`?

15

Умные сравнения и given-when

Как бы нам всем хотелось, чтобы компьютеры всегда понимали, чего мы хотим, и делали это за нас! Perl и так старается использовать числа, строки, одиночные значения и списки там, где это подразумевается контекстом. Но с появлением в Perl 5.10 оператора умного сравнения и управляющей конструкции `given-when` все становится еще удобнее.

Оператор умного сравнения

Оператор умного сравнения `~~` проверяет оба операнда и самостоятельно решает, как он их будет сравнивать. Если операнды выглядят как числа, выполняется числовое сравнение. Если операнды выглядят как строки, оператор сравнивает их как строки. Если один операнд содержит регулярное выражение, выполняется поиск по шаблону. Оператор даже способен выполнять сложные задачи, решение которых потребует большого объема кода, избавляя вас от ввода лишних символов.

Оператор `~~` внешне напоминает оператор привязки `=~`, представленный в главе 8, но оператор `~~` способен на большее. Более того, он даже может заменить оператор привязки. Ранее мы использовали оператор привязки для того, чтобы связать `$name` с оператором поиска совпадения по регулярному выражению:

```
print "I found Fred in the name!\n" if $name =~ /Fred/;
```

Если заменить оператор привязки оператором умного сравнения, программа будет делать абсолютно то же самое:

```
use 5.010;
say "I found Fred in the name!" if $name ~~ /Fred/;
```

Оператор умного сравнения видит, что в левой части находится скалярное значение, а в правой – оператор поиска совпадения, и само-

стоятельно определяет, что от него требуется. Впрочем, это только начало. Далее все будет намного интереснее.

Возможности оператора умного сравнения в полной мере проявляются при более сложных операциях. Допустим, вы хотите вывести сообщение, если в одном из ключей хеша `%names` совпадает шаблон `Fred`. Использовать `exists` не удастся, эта функция проверяет только точное значение ключа. Можно создать цикл `foreach`, который проверяет каждый ключ по оператору регулярного выражения и пропускает ключи, в которых совпадение не обнаружено. Обнаружив ключ с совпадением, мы изменяем значение переменной `$flag` и пропускаем остальные итерации командой `last`:

```
my $flag = 0;
foreach my $key ( keys %names ) {
    next unless $key =~ /Fred/;
    $flag = $key;
    last;
    }

print "I found a key matching 'Fred'. It was $flag\n" if $flag;
```

Ого! Даже объяснение получается слишком длинным, хотя такое решение работает в любой версии Perl 5. Но с оператором умного сравнения вы просто ставите в левой части хеш, а в правой части – оператор регулярного выражения:

```
use 5.010;

say "I found a key matching 'Fred'" if %names ~~ /Fred/;
```

Оператор умного совпадения знает, что делать, потому что он видит хеш и регулярное выражение. Он знает, что с такими операндами нужно взять ключи `%names` и применить регулярное выражение к каждому из них. Если оператор найдет совпадение, он останавливается и возвращает *true*. При получении скалярного значения и регулярного выражения оператор действовал бы иначе. Именно поэтому оператор `~~` называется «умным»; он делает то, что лучше подходит для текущей ситуации. Оператор остается прежним, но выполняемые им действия изменяются.

Допустим, вам понадобилось сравнить два массива (с одинаковым количеством элементов). Можно перебрать индексы одного массива и в каждой итерации сравнить соответствующие элементы двух массивов. Каждый раз, когда элементы совпадают, в программе увеличивается счетчик `$equal`. Если после завершения цикла значение `$equal` совпадает с количеством элементов `@names1`, массивы совпадают:

```
my $equal = 0;
foreach my $index ( 0 .. $#names1 ) {
    last unless $names1[$index] eq $names2[$index];
    $equal++;
```

```
    }

print "The arrays have the same elements!\n"
    if $equal == @names1;
```

И снова получается слишком много работы. Нельзя ли то же самое сделать как-то проще? Постойте-ка! А как насчет оператора умного сравнения? Передайте два массива оператору `~~`. Следующий небольшой фрагмент делает то же, что и предыдущий пример, но с минимумом программного кода:

```
use 5.010;

say "The arrays have the same elements!"
    if @names1 ~~ @names2;
```

Еще один пример. Допустим, мы вызываем функцию и хотим убедиться в том, что ее возвращаемое значение входит в множество возможных или ожидаемых значений. Вспомните функцию `max()` из главы 4: она всегда должна возвращать одно из переданных ей значений. Чтобы убедиться в этом, можно проверить возвращаемое значение `max` по списку аргументов «вручную», как это делалось в предыдущих трудных решениях:

```
my @nums   = qw( 1 2 3 27 42 );
my $result = max( @nums );

my $flag = 0;
foreach my $num ( @nums ) {
    next unless $result == $num;
    $flag = 1;
    last;
    }

print "The result is one of the input values\n" if $flag;
```

Вы уже знаете, что мы скажем: слишком много работы! Оператор `~~` позволяет исключить из этого кода весь средний фрагмент. Решение заметно упрощается:

```
use 5.010;

my @nums   = qw( 1 2 3 27 42 );
my $result = max( @nums );

say "The result [$result] is one of the input values (@nums)"
    if @nums ~~ $result;
```

Даже если записать операнды в другом порядке, результат от этого не изменится. Оператор умного сравнения не обращает внимания, с какой стороны записан тот или иной операнд:

```
use 5.010;

my @nums   = qw( 1 2 3 27 42 );
my $result = max( @nums );
```

```
say "The result [$result] is one of the input values (@nums)"
    if $result ~~ @nums;
```

Оператор умного сравнения обладает свойством коммутативности. Возможно, вы помните из школьного курса алгебры, что этот ученый термин означает лишь то, что порядок операндов неважен. Оператор умного сравнения в этом отношении напоминает операции сложения или умножения; перестановка операндов не влияет ни на способ сравнения, ни на ответ. Следующие две команды эквивалентны:

```
use 5.010;

say "I found a name matching 'Fred'" if $name ~~ /Fred/;
say "I found a name matching 'Fred'" if /Fred/ ~~ $name;
```

Приоритеты умного сравнения

Итак, вы видите, что оператор умного сравнения экономит немало времени и усилий. Остается узнать, какие сравнения выполняются в тех или иных ситуациях. Для этого необходимо обратиться к таблице из раздела «Smart matching in detail» документации *perlsyn*. В табл. 15.1 представлены некоторые виды сравнений, выполняемых оператором `~~`.

Таблица 15.1. Операции умного сравнения, выполняемые для разных пар операндов

Пример	Тип сравнения
`%a ~~ %b`	Ключи хешей идентичны
`%a ~~ @b`	Хотя бы один ключ `%a` содержится в `@b`
`%a ~~ /Fred/`	Шаблон совпадает хотя бы в одном ключе
`%a ~~ 'Fred'`	Существование ключа хеша: `exists $a{Fred}`
`@a ~~ @b`	Массивы совпадают
`@a ~~ /Fred/`	Шаблон совпадает хотя бы в одном элементе
`@a ~~ 123`	Содержит хотя бы один элемент 123 (в числовом формате)
`@a ~~ 'Fred'`	Содержит хотя бы один элемент `'Fred'` (в строковом формате)
`$name ~~ undef`	Переменная `$name` имеет неопределенное значение
`$name ~~ /Fred/`	Поиск совпадения по шаблону
`123 ~~ '123.0'`	Проверка числового равенства с «числовыми строками»
`'Fred' ~~ 'Fred'`	Равенство строк
`123 ~~ 456`	Числовое равенство

Встретив оператор умного сравнения, Perl переходит в начало таблицы и начинает искать тип, соответствующий двум операндам. Сравнение осуществляется по первому найденному типу. Операнды могут

следовать в произвольном порядке. Допустим, в операндах передается массив и хеш:

```
use 5.010;

if( @array ~~ %hash ) { ... }
```

Perl находит для хеша и массива сравнение, которое проверяет, что хотя бы один из элементов `@array` является ключом в `%hash`. Здесь все просто, потому что для этих двух операндов возможен только один тип сравнения. А если указаны два скаляра?

```
use 5.010;

if( $fred ~~ $barney ) { ... }
```

Пока невозможно сказать, какое сравнение будет выполнено; чтобы выбрать тип сравнения, необходимо заглянуть в содержимое этих двух скалярных переменных. Perl не может принять решения без анализа данных, содержащихся в этих переменных. В каком виде сравнивать переменные – в числовом или в строковом?

Чтобы определить, как следует сравнивать `$fred` и `$barney`, Perl проверяет значения по уже изложенным правилам. Он перебирает строки таблицы от начала к концу, пока не найдет подходящее описание, а затем использует соответствующую операцию. При этом необходимо учитывать одну тонкость: Perl распознает некоторые строки как числа (так называемые «числовые строки» – numish strings). Речь идет о строках вида `'123'`, `'3.14149'` и т. д. Содержимое этих строк заключено в апострофы, поэтому они фактически являются последовательностями символов, однако Perl может преобразовать их в числа без предупреждений. Если Perl обнаруживает в обеих частях оператора умного сравнения числа или числовые строки, он выполняет числовое сравнение. В противном случае выполняется строковое сравнение.

Команда given

Управляющая конструкция `given` позволяет выполнить блок кода, если аргумент удовлетворяет указанному условию. В сущности, это Perl-эквивалент команды `switch` языка C; но, как и многие конструкции Perl, `given` обладает оригинальными возможностями, поэтому ей присваивается оригинальное имя.

Следующий фрагмент кода берет первый аргумент из командной строки `$ARGV[0]`, перебирает условия `when` и смотрит, где в аргументе удастся найти `Fred`. Каждый блок `when` сообщает об одном способе вхождения `Fred`, начиная с самого общего и заканчивая самым конкретным:

```
use 5.010;

given( $ARGV[0] ) {
    when( /fred/i ) { say 'Name has fred in it' }
```

```
    when( /^Fred/ ) { say 'Name starts with Fred' }
    when( 'Fred'  ) { say 'Name is Fred' }
    default         { say "I don't see a Fred" }
    }
```

В конструкции `given` аргумент представлен псевдонимом `$_`, а каждое условие пытается применить умное сравнение к `$_`. Предыдущий пример можно переписать с явным применением умного сравнения, чтобы понять, что происходит:

```
use 5.010;

given( $ARGV[0] ) {
    when( $_ ~~ /fred/i ) { say 'Name has fred in it' }
    when( $_ ~~ /^Fred/ ) { say 'Name starts with Fred' }
    when( $_ ~~ 'Fred'  ) { say 'Name is Fred' }
    default               { say "I don't see a Fred" }
    }
```

Если `$_` не удовлетворяет ни одному условию `when`, выполняется блок `default`. Вот как выглядят результаты нескольких пробных запусков:

```
$ perl5.10.0 switch.pl Fred
Name has fred in it
$ perl5.10.0 switch.pl Frederick
Name has fred in it
$ perl5.10.0 switch.pl Barney
I don't see a Fred
$ perl5.10.0 switch.pl Alfred
Name has fred in it
```

«Подумаешь», – скажете вы. – «Я бы мог записать этот пример с `if-elsif-else`». В следующем примере именно это и делается с использованием переменной `$_`, объявленной с ключевым словом `my`. На эту переменную распространяются все правила лексической видимости `my`, еще одной новой возможности Perl 5.10:

```
use 5.010;

{
my $_ = $ARGV[0]; # lexical $_ as of 5.10!
   if( $_ ~~ /fred/i ) { say 'Name has fred in it' }
elsif( $_ ~~ /^Fred/ ) { say 'Name starts with Fred' }
elsif( $_ ~~ 'Fred'  ) { say 'Name is Fred' }
else                   { say "I don't see a Fred" }
}
```

Если бы конструкция `given` делала абсолютно то же, что `if-elsif-else`, никакого интереса она бы не представляла. Но, в отличие от `if-elsif-else`, конструкция `given-when` после выполнения одного условия может продолжить проверку остальных условий. С другой стороны, `if-elsif-else` при выполнении одного условия всегда выполняет ровно один блок кода.

Прежде чем следовать далее, необходимо указать на пару важных моментов. Если не принять специальные меры, в конец каждого блока включается неявная команда `break`, которая приказывает Perl прервать обработку `given-when` и продолжить выполнение программы. Таким образом, предыдущий пример содержал команды `break`, хотя вам и не пришлось вводить их самостоятельно:

```
use 5.010;

given( $ARGV[0] ) {
    when( $_ ~~ /fred/i ) { say 'Name has fred in it'; break }
    when( $_ ~~ /^Fred/ ) { say 'Name starts with Fred'; break }
    when( $_ ~~ 'Fred'  ) { say 'Name is Fred'; break }
    default               { say "I don't see a Fred"; break }
    }
```

Но для нашей задачи такой способ обработки не подходит. Так как наш пример переходит от общих условий к более конкретным, в случае совпадения `/fred/i` Perl не проверяет другие условия `when`. До проверки аргумента на полное совпадение с `Fred` дело уже не доходит, потому что первый же блок `when` прерывает дальнейшую обработку конструкции.

Но если завершить блок `when` ключевым словом `continue`, Perl даже в случае выполнения условия перейдет к следующему условию, и весь процесс повторится заново. Конструкция `if-elsif-else` на такое не способна. Если другое условие `when` окажется истинным, Perl выполняет его блок (снова неявно завершаемый командой `break`, если в программе явно не указано обратное). Если добавить `continue` в конец каждого блока, Perl опробует все условия:

```
use 5.010;

given( $ARGV[0] ) {
    when( $_ ~~ /fred/i ) { say 'Name has fred in it'; continue }
    when( $_ ~~ /^Fred/ ) { say 'Name starts with Fred'; continue }
    when( $_ ~~ 'Fred'  ) { say 'Name is Fred'; continue } # OOPS!
    default               { say "I don't see a Fred" }
    }
```

Однако с этим кодом возникает небольшая проблема. Запустив его, вы увидите, что блок `default` выполняется вместе со всеми предшествующими блоками:

```
$ perl5.10.0 switch.pl Alfred
Name has fred in it
I don't see a Fred
```

Блок `default` в действительности представляет собой `when` с условием, которое всегда истинно. Если блок `when` перед `default` содержит `continue`, Perl переходит к `default`. С его точки зрения все выглядит так, словно `default` является очередным блоком `when`:

```
use 5.010;

given( $ARGV[0] ) {
    when( $_ ~~ /fred/i ) { say 'Name has fred in it'; continue }
    when( $_ ~~ /^Fred/ ) { say 'Name starts with Fred'; continue }
    when( $_ ~~ 'Fred'  ) { say 'Name is Fred'; continue } # OOPS!
    when( 1 == 1        ) { say "I don't see a Fred" } # default
    }
```

Чтобы этого не происходило, достаточно убрать последнюю команду `continue`. В этом случае последний блок `when` завершает процесс:

```
use 5.010;

given( $ARGV[0] ) {
    when( $_ ~~ /fred/i ) { say 'Name has fred in it'; continue }
    when( $_ ~~ /^Fred/ ) { say 'Name starts with Fred'; continue }
    when( $_ ~~ 'Fred'  ) { say 'Name is Fred'; break } # OK now!
    when( 1 == 1        ) { say "I don't see a Fred" }
    }
```

Итак, теперь вы знаете, как работает этот код, и мы можем переписать его в идиоматической форме, которую и следует применять в ваших программах:

```
use 5.010;

given( $ARGV[0] ) {
    when( /fred/i ) { say 'Name has fred in it'; continue }
    when( /^Fred/ ) { say 'Name starts with Fred'; continue }
    when( 'Fred'  ) { say 'Name is Fred'; }
    default         { say "I don't see a Fred" }
    }
```

Обычное сравнение

Наряду с умным сравнением в `given-when` также можно использовать обычные, «глупые» сравнения. Конечно, ничего плохого в них нет, просто это самые обычные поиски совпадений по регулярным выражениям, которые вам уже известны. Обнаружив в блоке `when` явно заданный оператор сравнения (любого типа) или оператор привязки, Perl делает только то, что делают эти операторы:

```
use 5.010;

given( $ARGV[0] ) {
    when( $_ =~ /fred/i ) { say 'Name has fred in it'; continue }
    when( $_ =~ /^Fred/ ) { say 'Name starts with Fred'; continue }
    when( $_ eq 'Fred'  ) { say 'Name is Fred' }
    default               { say "I don't see a Fred" }
    }
```

Допускается даже смешанное использование умных и обычных сравнений; для каждого отдельного блока `when` сравнения вычисляются отдельно от других блоков:

```
use 5.010;

given( $ARGV[0] ) {
    when( /fred/i )       { #smart
        say 'Name has fred in it'; continue }
    when( $_ =~ /^Fred/ ) { #dumb
        say 'Name starts with Fred'; continue }
    when( 'Fred'  )       { #smart
        say 'Name is Fred' }
    default               { say "I don't see a Fred" }
    }
```

Для поиска по шаблону умное сравнение не отличается от обычного, поскольку операторы регулярных выражений по умолчанию привязываются к $_.

Оператор умного сравнения ищет совпадения (полные или частичные), поэтому он не работает для сравнений типа «больше» или «меньше». В таких случаях необходимо использовать традиционные операторы сравнения:

```
use 5.010;

given( $ARGV[0] ) {
    when( /^-?\d+\.\d+$/ ) { # Умное сравнение
        say 'Not a number!' }
    when( $_ > 10 )        { # Обычное сравнение
        say 'Number is greater than 10'  }
    when( $_ < 10 )        { # Обычное сравнение
        say 'Number is less than 10' }
    default                { say 'Number is 10' }
    }
```

В некоторых ситуациях Perl автоматически использует обычные сравнения. Например, если в when включается результат вызова пользовательской функции[1], Perl использует логическое значение (*true*/*false*) возвращаемого значения:

```
use 5.010;

given( $ARGV[0] ) {
    when( name_has_fred( $_ ) ) { # Обычное сравнение
        say 'Name has fred in it'; continue }
    }
```

Правило о вызове функции распространяется и на встроенные функции Perl defined, exists и eof, возвращающие логический результат.

Умное сравнение также не используется для инвертированных выражений, включая инвертированные регулярные выражения. Эти случаи

[1] Perl также не применяет умное сравнение для вызовов методов, но ОО-программирование здесь не рассматривается. Обращайтесь к книге «Intermediate Perl» (O'Reilly) – «Perl: изучаем глубже», Символ-Плюс, 2007.

ничем не отличаются от условий управляющих конструкций, представленных в предыдущих главах:

```
use 5.010;

given( $ARGV[0] ) {
    when( ! $boolean )  { # Обычное сравнение
        say 'Name has fred in it' }
    when( ! /fred/i   } { # Обычное сравнение
        say 'Does not match Fred' }
    }
```

Условия when с несколькими элементами

Иногда требуется выполнить однотипную обработку нескольких элементов, но given каждый раз получает только один элемент. Конечно, можно заключить given в цикл foreach. Скажем, если вы хотите перебрать массив @names, присвойте текущий элемент переменной $name и используйте ее в given:

```
use 5.010;

foreach my $name ( @names ) {
    given( $name ) {
        ...
        }
    }
```

Что скажете? Да, все верно: слишком много работы. (Вы еще не устали от этой фразы?) На этот раз мы определяем псевдоним для текущего элемента @names только для того, чтобы конструкция given могла определить для него другой псевдоним! Не может быть, чтобы в Perl не существовало более эффективного решения! Не беспокойтесь – конечно, оно существует.

Для перебора нескольких элементов ключевое слово given вообще не потребуется. Пусть цикл foreach сам поместит текущий элемент в $_. Если вы хотите использовать умные сравнения, текущий элемент должен храниться в $_.

```
use 5.010;

foreach ( @names ) { # Не используем именованную переменную!
    when( /fred/i ) { say 'Name has fred in it'; continue }
    when( /^Fred/ ) { say 'Name starts with Fred'; continue }
    when( 'Fred'  ) { say 'Name is Fred'; }
    default         { say "I don't see a Fred" }
    }
```

Возможно, при переборе вам потребуется узнать, с каким именно элементом вы работаете в настоящий момент. В блок foreach можно включать и другие команды, в том числе команду say:

```
use 5.010;

foreach ( @names ) { # Не используем именованную переменную!
    say "\nProcessing $_";

    when( /fred/i ) { say 'Name has fred in it'; continue }
    when( /^Fred/ ) { say 'Name starts with Fred'; continue }
    when( 'Fred'  ) { say 'Name is Fred'; }
    default         { say "I don't see a Fred" }
    }
```

Дополнительные команды даже могут располагаться между блоками `when`, например команда вывода отладочной информации при входе в `default` (то же самое можно сделать и с `given`):

```
use 5.010;

foreach ( @names ) { # Не используем именованную переменную!
    say "\nProcessing $_";

    when( /fred/i ) { say 'Name has fred in it'; continue }
    when( /^Fred/ ) { say 'Name starts with Fred'; continue }
    when( 'Fred'  ) { say 'Name is Fred'; }
    say "Moving on to default...";
    default         { say "I don't see a Fred" }
    }
```

Упражнения

Ответы к упражнениям приведены в приложении А.

1. [15] Перепишите программу из упражнения 1 главы 10 так, чтобы в ней использовалась конструкция `given`. Как обеспечить обработку нечисловых входных данных? Использовать умные сравнения необязательно.
2. [15] Напишите программу с `given-what`, которая получает число во входных данных, а затем выводит строку «Fizz», если число делится на 3, «Bin», если оно делится на 5, и «Sausage», если оно делится на 7. Например, для числа 15 программа должна выводить «Fizz» и «Bin», потому что 15 делится как на 3, так и на 5. Для какого числа ваша программа выведет «Fizz Bin Sausage»?
3. [15] Используя конструкцию `for-when`, напишите программу, которая перебирает список файлов в командной строке и сообщает, доступен ли каждый файл для чтения, записи или исполнения. Использовать умные сравнения необязательно.
4. [20] Используя `given` и умные сравнения, напишите программу для вывода всех делителей числа, переданного в командной строке (кроме 1 и самого числа). Например, для числа 99 программа должна выдать делители 3, 9, 11 и 33. Если число является простым (т. е. не имеет таких делителей), программа должна сообщить об этом. Если аргумент командной строки не является числом, про-

грамма выдает сообщение об ошибке и не пытается вычислять делители. Хотя задачу можно решить при помощи конструкций `if` с обычными сравнениями, используйте только умные сравнения.

Для начала приведем пользовательскую функцию, возвращающую список делителей. Функция последовательно проверяет все числа от 1 до половины `$number`:

```
sub divisors {
    my $number = shift;

    my @divisors = ();
    foreach my $divisor ( 2 .. $number/2 ) ) {
        push @divisors, $divisor unless $_ % $divisor;
        }

    return @divisors;
    }
```

5. [20] Измените программу из предыдущего упражнения так, чтобы она дополнительно сообщала, является ли число четным или нечетным, является ли оно простым (если для него не найдены другие делители, кроме 1 и самого числа), делится ли оно на ваше любимое число. В этой программе также должны использоваться только умные сравнения.

16

Управление процессами

Одна из самых приятных сторон работы программиста – возможность запускать чужой код, чтобы вам не пришлось писать его своими силами. Пора узнать, как управлять дочерними процессами, запуская другие программы прямо из Perl.

Как и во всем, что происходит в Perl, здесь действует известный девиз: «Это можно сделать несколькими способами», с различными вариациями и специальными возможностями. Если вам не понравился первый способ, продолжайте читать, и через пару страниц найдется решение, которое придется вам по вкусу.

Perl отличается хорошей портируемостью; в предыдущих главах почти нет сносок, в которых бы говорилось, что некая функция работает в UNIX так, в Windows – иначе, а в VMS – еще как-нибудь. Но в том, что касается запуска других программ, ситуация изменяется: на Macintosh доступны одни программы, на Cray – другие. Примеры этой главы в основном ориентированы на UNIX; в других системах возможны некоторые отличия.

Функция system

Для запуска дочернего процесса в Perl проще всего воспользоваться функцией `system`. Например, выполнение команды UNIX *date* в Perl выглядит так:

```
system "date";
```

Дочерний процесс выполняет команду *date*, которая наследует от Perl стандартные потоки ввода, вывода и ошибок. Это означает, что стандартная строка с датой и временем в коротком формате, выдаваемая

date, попадает в тот приемник, с которым в Perl уже связан дескриптор STDOUT.

В параметре функции system передается строка, которая обычно вводится в командном процессоре для выполнения команды. Таким образом, для более сложных команд (скажем, ls -l $HOME) весь текст придется включить в параметр:

```
system 'ls -l $HOME';
```

Обратите внимание: нам пришлось перейти от кавычек к апострофам, так как $HOME является переменной командного процессора. Без апострофов командный процессор не «увидит» знак $, потому что он также является признаком интерполяции переменной для Perl. Конечно, нужный знак можно экранировать в строке:

```
system "ls -l \$HOME";
```

Но такая запись быстро становится слишком громоздкой.

Команда *date* только выводит данные. Но предположим, что она также общается с пользователем, спрашивая его: «Для какого часового пояса вы хотите получить время?»[1] Строка попадет в стандартный вывод, а программа будет ожидать поступления данных из стандартного ввода (унаследованного от дескриптора STDIN). Пользователь видит вопрос, вводит ответ, а *date* выполняет свою работу.

Пока дочерний процесс работает, Perl терпеливо ждет его завершения. Если выполнение команды *date* занимает 37 секунд, Perl приостановит выполнение программы на эти 37 секунд. Впрочем, вы можете воспользоваться функциональностью командного процессора для запуска процесса в фоновом режиме:[2]

```
system "long_running_command with parameters &";
```

Командный процессор получает управление, замечает & в конце командной строки и запускает *long_running_command* в фоновом режиме. Затем он довольно быстро возвращает управление, Perl замечает это и продолжает выполнение программы. Процесс *long_running_command* становится «внуком» процесса Perl; последний не имеет ни прямого доступа к нему, ни информации о его существовании.

Если команда «достаточно проста», командный процессор вообще не задействуется. Так, упоминавшиеся ранее команды *date* и *ls* Perl запускает напрямую, для чего он ищет команду по унаследованному

---

[1] Насколько нам известно, никто еще не написал команду *date*, работающую подобным образом.

[2] Теперь видите, что мы имели в виду, говоря о зависимости от системы? Командный процессор UNIX (*/bin/sh*) позволяет включать знак & в такие команды для выполнения в фоновом режиме. Если ваша система не поддерживает этот способ запуска фоновых процессов, он не сработает, вот и все.

значению PATH.[1] Но если в строке есть что-то необычное (например, метасимволы командного процессора: $, ; или |), для ее обработки активизируется стандартный командный процессор Bourne Shell (*/bin/sh*[2]). В этом случае дочерним процессом является командный процессор, а запрашиваемая команда становится «внуком» (или потомком следующего уровня). Например, вы можете записать в аргументе целый мини-сценарий командного процессора:

```
system 'for i in *; do echo == $i ==; cat $i; done';
```

Мы снова используем апострофы, потому что знак $ предназначается для командного процессора, а не для Perl. С кавычками Perl заменит $i текущим значением переменной и не позволит командному процессору использовать свое значение.[3] Кстати говоря, этот мини-сценарий перебирает все обычные файлы текущего каталога, выводя их имена и значения; если не верите, убедитесь сами.

Выполнение команд в обход командного процессора

Оператор system также может вызываться с несколькими аргументами.[4] В этом случае командный процессор не используется, какой бы сложной ни была команда:

```
my $tarfile = "something*wicked.tar";
my @dirs = qw(fred|flintstone <barney&rubble> betty );
system "tar", "cvf", $tarfile, @dirs;
```

Первый параметр ("tar" в приведенном примере) определяет имя команды, находимой стандартными средствами поиска в PATH, а остальные аргументы один за одним передаются команде. Даже если аргументы содержат символы, интерпретируемые командным процессором (например, имя файла в $tarfile или имена каталогов в @dirs), командный процессор не сможет обработать строку. Таким образом, команда *tar* получит ровно пять параметров. Сравните со следующим вызовом:

---

1 Значение PATH можно в любой момент изменить через $ENV{'PATH'}. Изначально это переменная среды, унаследованная от родительского процесса (чаще всего командного процессора). Ее изменение влияет на создаваемые дочерние процессы, но не отражается на родительских процессах. В переменной PATH хранится список каталогов, в которых ищутся исполняемые программы (команды), – даже в системах, не входящих в семейство UNIX.

2 Или другой командный процессор, определяемый на стадии построения Perl. В UNIX-подобных системах почти всегда используется */bin/sh*.

3 Конечно, если использовать присваивание $i = '$i', программа будет работать, пока в ходе сопровождения кто-нибудь не «исправит» эту строку.

4 Или с параметром в позиции косвенного объекта, как в system { 'fred' }'barney'; эта команда запускает программу barney, но убеждает ее в том, что она называется 'fred'. См. man-страницу *perlfunc*.

```
system "tar cvf $tarfile @dirs";  # Ошибка!
```

Обратите внимание: вызов `system` с одним аргументом практически эквивалентен следующему вызову с несколькими аргументами:

```
system $command_line;
system "/bin/sh", "-c", $command_line;
```

Но последний вариант записи почти не используется, если только вы не хотите, чтобы обработка выполнялась другим командным процессором, например C Shell:

```
system "/bin/csh", "-fc", $command_line;
```

Но и такая ситуация встречается редко, потому что Единственно Верный Командный Процессор[1] обладает большей гибкостью.

Возвращаемое значение оператора `system` зависит от кода завершения дочерней команды.[2] В UNIX код 0 означает, что все прошло нормально, а ненулевой код завершения обычно свидетельствует о возникших проблемах:

```
unless (system "date") {
  # Возвращаемое значение равно нулю - признак успеха
  print "We gave you a date, OK!\n";
}
```

В данном случае ситуация отличается от стандартной стратегии «*true* – хорошо, *false* – плохо», используемой большинством операторов, поэтому для применения типичной идиомы «сделай `or die`» придется поменять значения *true* и *false*. Проще всего снабдить оператор `system` префиксом `!` (оператор логического отрицания):

```
!system "rm -rf files_to_delete" or die "something went wrong";
```

Здесь включать `$!` в сообщение об ошибке не следует. Любые возможные сбои почти наверняка произойдут из-за команды *rm*, а не из-за ошибки, связанной с вызовом `system`, о которой может сообщить переменная `$!`.

---

[1] Это */bin/sh* или ближайший к Bourne аналог, установленный в вашей системе UNIX. Если у вас нет Единственно Верного Командного процессора, Perl вычисляет, как вызвать другой интерпретатор командной строки с соответствующими последствиями (см. документацию для вашей портированной версии Perl).

[2] Вообще говоря, от «статуса ожидания», который представляет собой код завершения дочерней команды плюс 128 в случае аварийного завершения, плюс номер сигнала, инициировавшего завершение (если он был). Но конкретное значение нас обычно не интересует, и простого логического признака *true*/*false* достаточно практически для всех приложений.

Функция exec

Практически все, что было сказано о синтаксисе и семантике system, также относится к функции exec – кроме одного (очень важного) обстоятельства. Функция system создает дочерний процесс, который берется за свою работу, пока Perl терпеливо ждет. Функция exec заставляет *сам процесс Perl* выполнить запрашиваемое действие. Происходящее больше напоминает переход goto, нежели вызов функции.

Предположим, мы хотим выполнить команду *bedrock* из каталога */tmp*. Команда вызывается с аргументами *-o args1*, а за ними должны следовать аргументы, с которыми была запущена наша программа. Это будет выглядеть примерно так:

```
chdir "/tmp" or die "Cannot chdir /tmp: $!";
exec "bedrock", "-o", "args1", @ARGV;
```

При достижении операции exec Perl находит команду *bedrock* и «передает ей управление». После передачи процесс Perl исчезает[1], а остается процесс, в котором выполняется команда *bedrock*. С завершением *bedrock* процесса Perl, которому можно было бы вернуть управление, не остается, и мы возвращаемся к приглашению командной строки (так, словно программа была запущена из командной строки).

Зачем это нужно? Скажем, если единственной целью программы Perl является подготовка среды исполнения для другой программы. Как только эта цель будет выполнена, запускается другая программа. Если бы мы использовали system вместо exec, программе Perl пришлось бы терпеливо дожидаться завершения system – только для того, чтобы немедленно завершиться после возврата управления, а это весьма неэффективно.

При этом exec на практике используется относительно редко, разве что в сочетании с fork (см. далее). Если вы ломаете голову над выбором «system или exec», выбирайте system, и вы почти всегда окажетесь правы.

Так как Perl перестает управлять выполнением программы после запуска указанной команды, любой код Perl после вызова exec не имеет смысла, кроме обработки ошибок в том случае, если запрос на запуск команды завершился неудачей:

```
exec "date";
die "date couldn't run: $!";
```

Более того, если включен режим предупреждений, а за exec следует любой другой код вместо die[2], Perl оповестит вас об этом.

---

[1] На самом деле процесс остается прежним, но выполняет системную функцию UNIX exec(2) (или ее эквивалент). Идентификатор процесса при этом не изменяется.

[2] Или exit. Или конец блока. И вообще в будущих версиях Perl все может измениться.

Переменные среды

При запуске другого процесса (любым из описанных способов) может возникнуть необходимость в подготовке среды выполнения. Как упоминалось ранее, процесс можно запустить с определенным рабочим каталогом, который наследуется от текущего процесса. Другой стандартный аспект конфигурации – переменные среды.

Из всех переменных среды наибольшей известностью пользуется PATH. (Если вы никогда не слышали о ней, вероятно, в вашей системе переменные среды не поддерживаются.) В UNIX и других аналогичных системах PATH содержит разделенный двоеточиями список каталогов, в которых могут храниться программы. Когда вы вводите команду (например, *rm fred*), система последовательно ищет файл *rm* во всех перечисленных каталогах. Perl (или ваша система) использует PATH всегда, когда потребуется найти запускаемую программу. Если программа в свою очередь запускает другие программы, они тоже ищутся среди каталогов PATH. (Конечно, если команда запускается по полному имени, например */bin/echo*, поиск в PATH оказывается лишним, но обычно запуск по полному имени слишком неудобен.)

В Perl для работы с переменными среды используется специальный хеш %ENV; каждый ключ хеша соответствует одной переменной. В начале выполнения программы %ENV содержит значения, унаследованные от родительского процесса (чаще всего командного процессора). Модификация хеша изменяет переменные среды, которые наследуются новыми процессами и могут использоваться самим Perl. Допустим, вы хотите запустить системную утилиту *make* (которая обычно запускает другие программы) так, чтобы поиск команд (включая саму команду *make*) начинался с приватного каталога. Также будем считать, что при запуске команды переменная среды IFS не должна устанавливаться, потому что это может нарушить работу *make* или другой подкоманды. Вот как это делается:

```
$ENV{'PATH'} = "/home/rootbeer/bin:$ENV{'PATH'}";
delete $ENV{'IFS'};
my $make_result = system "make";
```

Создаваемые процессы обычно наследуют от своих родителей переменные среды, стандартный рабочий каталог, стандартные потоки ввода, вывода и ошибок, а также ряд других эзотерических параметров. За дополнительной информацией обращайтесь к документации по программированию для вашей системы. (Учтите, что в большинстве систем программа не может изменять среду командного процессора или другого родительского процесса, запустившего ее.)

Обратные апострофы и сохранение вывода

При использовании обеих функций system и exec выходные данные запущенной команды направляются в стандартный поток вывода Perl. Иногда бывает нужно сохранить этот вывод в строковом виде для дальнейшей обработки. Задача решается просто: замените апострофы или кавычки при создании переменной обратными апострофами ` `:

```
my $now = `date`;              # Сохранить вывод date
print "The time is now $now"; # Символ новой строки уже присутствует
```

Обычно команда *date* выдает в стандартный вывод строку длиной приблизительно 30 символов. Строка содержит текущую дату и время и завершается символом новой строки. Когда мы заключаем вызов *date* в обратные апострофы, Perl выполняет команду *date*, сохраняет ее вывод в виде строкового значения и (в данном случае) присваивает ее переменной $now.

Этот синтаксис очень близок к использованию обратных апострофов в командном процессоре UNIX. Однако командный процессор также удаляет завершающий символ новой строки, чтобы упростить дальнейшее использование значения. Perl действует честнее: он выдает настоящие выходные данные. Чтобы получить тот же результат в Perl, нам пришлось бы обработать результат дополнительной операцией *chomp*:

```
chomp(my $no_newline_now = `date`);
print "A moment ago, it was $no_newline_now, I think.\n";
```

Значение в обратных апострофах интерпретируется как форма system с одним аргументом по правилам строк в кавычках (то есть с интерпретацией служебных последовательностей с символом \ и расширением переменных).[1] Например, для получения документации по функциям Perl мы могли бы многократно вызвать *perldoc* с разными аргументами:

```
my @functions = qw{ int rand sleep length hex eof not exit sqrt umask };
my %about;

foreach (@functions) {
  $about{$_} = `perldoc -t -f $_`;
}
```

Переменная $_ содержит разные значения при каждом вызове, что позволяет нам получать результаты разных вызовов, отличающихся только одним из параметров. Если эти функции вам еще незнакомы, загляните в документацию и посмотрите, что они делают.

---

[1] Таким образом, если вы хотите передать командному процессору литерал \, его необходимо удвоить. Если потребуется передать последовательность \\ (как это часто бывает в Windows), используйте четыре исходных символа.

В синтаксисе обратных апострофов нет простого аналога режима «обычных» апострофов[1]: ссылки на переменные и комбинации с \ расширяются всегда. Также не существует простого аналога версии system с несколькими аргументами (выполняемой без участия командного процессора). Если команда в обратных апострофах достаточно сложна, для ее интерпретации автоматически активизируется UNIX Bourne Shell (или другой командный процессор, используемый в вашей системе).

Постарайтесь обходиться без обратных апострофов в тех местах, где вывод не сохраняется.[2] Пример:

```
print "Starting the frobnitzigator:\n";
`frobnitz -enable`; # Не делайте этого!
print "Done!\n";
```

Дело в том, что Perl приходится выполнять ряд дополнительных действий для сохранения вывода команды (который немедленно теряется), к тому же вы теряете возможность использования system с несколькими аргументами для более точного управления списком аргументов. Итак, и с точки зрения эффективности, и с точки зрения безопасности system оказывается предпочтительнее.

Стандартный поток ошибок наследуется командой в обратных апострофах от Perl. Если команда выводит сообщения об ошибках в стандартный поток ошибок, скорее всего, они будут выведены на терминал; это собьет с толку пользователя, который не запускал команду *frobnitz*. Если вы предпочитаете сохранять сообщения об ошибках в стандартном выводе, воспользуйтесь стандартным механизмом «слияния стандартного потока ошибок со стандартным потоком вывода» командного процессора; на языке UNIX это называется записью 2>&1:

```
my $output_with_errors = `frobnitz -enable 2>&1`;
```

Но в этом случае стандартный поток ошибок смешивается со стандартным выводом по аналогии с тем, как это происходит при выводе на терминал (хотя, возможно, в несколько иной последовательности из-за буферизации). Если вы предпочитаете разделить поток вывода и поток ошибок, существует пара более сложных решений.[3] Стандартный поток ввода также наследуется от текущего стандартного ввода Perl. Как правило, команды, заключаемые в обратные апострофы, все равно не читают данные из стандартного ввода, так что это не создает особых проблем. Но допустим, что команда *date* спрашивает, какой часо-

---

[1] Хотя непростые существуют: заключите свою строку в ограничители qx'...' или сохраните ее в переменной, используя строку в апострофах, а затем интерполируйте *эту* переменную в обратных апострофах (интерполяция распространяется только на один уровень).

[2] То есть в пустом контексте.

[3] А именно IPC::Open3 из стандартной библиотеки Perl и самостоятельная реализация ветвления (см. далее).

вой пояс вам нужен (см. ранее). Строка с запросом направляется в стандартный вывод, сохраняется как часть результата, после чего команда *date* пытается получить данные из стандартного ввода. Но пользователь не видел запрос и не знает, что ему нужно вводить! Вскоре он позвонит вам и скажет, что ваша программа «зависла».

Итак, держитесь подальше от команд, читающих данные из стандартного ввода. Если вы не уверены в том, читаются данные из стандартного ввода или нет, добавьте перенаправление из */dev/null*:

```
my $result = `some_questionable_command arg arg argh </dev/null`;
```

В этом случае дочерний процесс командного процессора перенаправит ввод из */dev/null*, а «внук» *some_questionable_command* в худшем случае попытается прочитать данные и немедленно получит признак конца файла.

Обратные апострофы в списочном контексте

Если результат выполнения команды состоит из нескольких строк, в скалярном контексте обратные апострофы возвращают одну длинную строку с внутренними символами новой строки. Однако при использовании той же строки в списочном контексте создается список, один элемент которого соответствует одной строке вывода. Например, команда UNIX *who* обычно выдает строку текста для каждого текущего пользователя в системе:

```
merlyn     tty/42     Dec 7  19:41
rootbeer   console    Dec 2  14:15
rootbeer   tty/12     Dec 6  23:00
```

В левом столбце указано имя пользователя, в среднем – имя tty (т. е. имя подключения пользователя к компьютеру), а в оставшейся части строки выводится дата и время входа (и возможно, дополнительная информация, но не в этом примере). В скалярном контексте вся информация возвращается в одной строке, и нам придется разбивать ее самостоятельно:

```
my $who_text = `who`;
```

Но в списочном контексте данные сразу возвращаются с разбивкой по строкам:

```
my @who_lines = `who`;
```

`@who_lines` содержит элементы, каждый из которых завершается символом новой строки. Конечно, можно удалить все эти символы функцией `chomp`, но давайте пойдем в другом направлении. Если разместить вызов в обратных апострофах в заголовке `foreach`, цикл автоматически переберет все строки, последовательно присваивая каждую из них переменной `$_`:

```
foreach (`who`) {
  my($user, $tty, $date) = /(\S+)\s+(\S+)\s+(.*)/;
  $ttys{$user} .= "$tty at $date\n";
}
```

Для приведенных выше данных цикл выполняется три раза. (Вероятно, в вашей системе количество активных входов будет больше трех.) Обратите внимание на поиск по регулярному выражению; в отсутствие оператора привязки (=~) он применяется к переменной $_, и это хорошо, потому что именно в этой переменной хранятся выходные данные команды.

Регулярное выражение ищет шаблон вида «непустое слово, пропуски, непустое слово, пропуски, а затем весь остаток строки до символа новой строки, но не включая его (так как точка по умолчанию не совпадает с символом новой строки)».[1] Таким образом, при первой итерации в $1 сохраняется строка "merlyn", в $2 – строка "tty/42", а в $3 – строка "Dec 7 19:41". Однако регулярное выражение применяется в списочном контексте, поэтому вместо логического значения «совпало или нет» мы получим список заполненных переменных (см. главу 8). Переменная $user заполняется строкой "merlyn" и т. д.

Вторая команда в этом цикле просто сохраняет tty и дату, присоединяя их к текущему значению в хеше (возможно, undef), потому что пользователь может присутствовать в списке несколько раз (как пользователь "rootbeer" в нашем примере).

Процессы как файловые дескрипторы

До настоящего момента рассматривался исключительно синхронный запуск процессов, когда Perl заправляет всем происходящим, запускает команду, (обычно) дожидается ее завершения и получает вывод. Но Perl также может запускать дочерние процессы, которые существуют самостоятельно и взаимодействуют[2] с Perl на долгосрочной основе вплоть до завершения задачи.

В синтаксисе запуска параллельного дочернего процесса команда задается в качестве «имени файла» при вызове open, а перед или после нее ставится вертикальная черта. Этот синтаксис часто называется *открытием канала*:

```
open DATE, "date|" or die "cannot pipe from date: $!";
open MAIL, "|mail merlyn" or die "cannot pipe to mail: $!";
```

---

[1] Теперь вы видите, *почему* точка не совпадает с символом новой строки по умолчанию: в подобных шаблонах, применяемых довольно часто, не нужно беспокоиться о завершающем символе новой строки.

[2] Через каналы (pipes) или другой механизм простых межпроцессных коммуникаций, поддерживаемый вашей системой.

В первом примере, когда символ | стоит справа, стандартный вывод запущенной команды связывается с файловым дескриптором DATE, открытым для чтения, по аналогии с выполнением *date | your_program* в командном процессоре. Во втором примере, когда символ | стоит слева, стандартный ввод команды соединяется с дескриптором MAIL, открытым для записи, по аналогии с командой *your_program | mail merlyn*. В обоих случаях команда продолжает работать независимо от процесса Perl.[1] Если создать дочерний процесс не удалось, вызов open завершается неудачей. Если команда не существует или некорректна, это (обычно) не воспринимается как ошибка при открытии, но приводит к ошибке при закрытии. Вскоре мы вернемся к этому вопросу.

Остальная часть программы не знает, что файловый дескриптор открыт для процесса, а не для файла – более того, это неважно, и чтобы узнать об этом, придется основательно потрудиться. Таким образом, чтобы получить данные из файлового дескриптора, открытого для чтения, достаточно выполнить обычную операцию чтения:

```
my $now = <DATE>;
```

А чтобы отправить данные процессу *mail* (ожидающему получить тело сообщения для *merlyn* в стандартном вводе), хватит простой команды print с файловым дескриптором:

```
print MAIL "The time is now $now"; # Предполагается, что $now
                                   # завершается символом новой строки
```

Короче говоря, вы можете считать, что эти дескрипторы связаны с «волшебными файлами»: один файл содержит вывод команды *date*, а другой автоматически передает данные команде *mail*.

Если подключение связывается с дескриптором, открытым для чтения, а потом завершается, дескриптор возвращает признак конца файла, как при попытке чтения после конца обычного файла. При закрытии файлового дескриптора, открытого для передачи данных процессу, процесс получает признак конца файла. Таким образом, для завершения отправки электронной почты достаточно закрыть дескриптор:

```
close MAIL;
die "mail: non-zero exit of $?" if $?;
```

Закрытие дескриптора, связанного с процессом, ожидает завершения процесса, чтобы Perl мог узнать его код завершения. Код завершения сохраняется в переменной $? (по аналогии с соответствующей переменной Bourne Shell). Он представляет собой то же числовое значение, которое возвращается функцией system: нуль обозначает успех, все остальные числа – неудачу. Каждый новый завершенный процесс заме-

[1] Если процесс Perl завершит работу до завершения команды, по умолчанию читающая команда получит признак конца файла, а записывающая команда при следующей попытке записи получит сигнал сбоя канала.

няет предыдущее значение, так что сохраните его побыстрее, если собираетесь использовать в будущем. (Переменная $? также содержит код завершения последней команды system или ` `, если вас это интересует.)

Процессы синхронизируются точно так же, как цепочки конвейерных команд. Если вы пытаетесь прочитать данные, а данные недоступны, процесс приостанавливается (без потребления дополнительного процессорного времени) до тех пор, пока программа-отправитель не «заговорит» снова. Аналогично, если записывающий процесс «опередит» читающий процесс, он приостанавливается до тех пор, пока последний не «догонит» его. Между процессами создается промежуточный буфер обмена данными (обычно 8 Кбайт или около того), так что абсолютно точная синхронизация не требуется.

Зачем связывать процессы с файловыми дескрипторами? Прежде всего, это единственный простой способ передачи данных процессу на основании результатов вычислений. При чтении данных обратные апострофы обычно гораздо удобнее, если только данные не должны обрабатываться сразу же после записи.

Например, команда UNIX *find* ищет файлы по атрибутам и при относительно большом количестве файлов (например, при рекурсивном поиске от корневого каталога) выполняется сравнительно долго. Команду *find* можно выполнить в обратных апострофах, но часто бывает удобнее получать результаты по мере их поступления:

```
open F, "find / -atime +90 -size +1000 -print|" or die "fork: $!";
while (<F>) {
  chomp;
  printf "%s size %dK last accessed on %s\n",
    $_, (1023 + -s $_)/1024, -A $_;
}
```

Команда *find* находит файлы, к которым не было обращений за последние 90 дней, превышающие размером 1000 блоков. (Эти файлы являются хорошими кандидатами для перемещения на архивные носители.) В процессе поиска Perl ждет. При обнаружении очередного файла Perl реагирует на входящее имя и выводит информацию о файле для дальнейшего анализа. Если бы команда поиска выполнялась в обратных апострофах, мы бы не получили никаких данных до завершения *find*. Всегда полезно знать, что команда по крайней мере работает.

Ветвление

Кроме высокоуровневых интерфейсов, описанных ранее, Perl предоставляет практически прямой доступ к низкоуровневым системным функциям UNIX и других систем. Если ранее вы никогда не имели де-

ла с этой областью[1], вероятно, этот раздел можно пропустить. Привести подробное описание в этой главе не удастся, но, по крайней мере, разберем в общих чертах реализацию следующего вызова:

```
system "date";
```

При использовании низкоуровневых системных вызовов эта задача реализуется так:

```
defined(my $pid = fork) or die "Cannot fork: $!";
unless ($pid) {
  # Дочерний процесс
  exec "date";
  die "cannot exec date: $!";
}
# Родительский процесс
waitpid($pid, 0);
```

Мы проверяем возвращаемое значение `fork`, которое будет равно `undef` в случае сбоя. Обычно вызов завершается удачно, а к следующей строке переходят уже два разных процесса, но только родительский процесс содержит ненулевое значение в `$pid`, поэтому только дочерний процесс выполнит функцию `exec`. Родительский процесс пропускает этот вызов и выполняет функцию `waitpid`, ожидающую завершения этого конкретного дочернего процесса. Если это описание покажется полной абракадаброй, просто запомните, что вы можете пользоваться функцией `system`, и над вами никто не будет смеяться.

За все дополнительные хлопоты вы получаете полный контроль за созданием каналов, переназначением файловых дескрипторов и информацию об идентификаторах процесса и его родителя (если он известен). Но как уже говорилось ранее, тема ветвления немного сложна для этой главы. За дополнительной информацией обращайтесь к man-странице *perlipc* (или любой хорошей книге по прикладному программированию для вашей системы).

Отправка и прием сигналов

Сигнал UNIX представляет собой крошечное сообщение, отправленное процессу. Он не может содержать подробной информации; представьте себе автомобильный сигнал – что он может означать? «Осторожно, здесь обвалился мост», или «Светофор переключился, можно ехать», или «Остановись, у тебя на крыше ребенок», или «Привет всем»? К счастью, понять смысл сигнала UNIX несколько проще, потому что

[1] Или если в вашей системе *fork* не поддерживается. Но разработчики Perl усердно трудятся над поддержкой ветвления даже в тех системах, модель процессов которых значительно отличается от модели UNIX.

в каждой из описанных ситуаций используется свой сигнал.[1] Сигналы идентифицируются по именам (например, SIGINT – «сигнал прерывания») и коротким числовым кодам (в диапазоне от 1 до 16, от 1 до 32 или от 1 до 63 в зависимости от вашей разновидности UNIX). Сигнал обычно отправляется при обнаружении важного события. Например, при нажатии клавиш прерывания программы (чаще всего Control-C) на терминале всем процессам, присоединенным к этому терминалу, отправляется сигнал SIGINT.[2] Некоторые сигналы автоматически отправляются системой, но они также могут отправляться другими процессами.

Вы можете отправить из процесса Perl сигнал другому процессу, но для этого необходимо знать идентификатор целевого процесса. Определить его не так просто[3], но предположим, вы хотите отправить сигнал SIGINT процессу 4201. Делается это достаточно просто:

```
kill 2, 4201 or die "Cannot signal 4201 with SIGINT: $!";
```

Функция называется kill, потому что сигналы очень часто применяются для остановки процесса, выполнение которого занимает слишком много времени. Вместо 2 также можно использовать строку 'INT', так как числовой код 2 соответствует сигналу SIGINT. Если процесс не существует[4], вы получите значение *false*, поэтому этот способ позволяет проверить, жив ли процесс. Специальный сигнал с номером 0 означает примерно следующее: «Просто проверить, смогу ли я отправить сигнал, если захочу... Но я пока не хочу, так что и отправлять ничего не нужно». Таким образом, зондирование процесса может выглядеть примерно так:

```
unless (kill 0, $pid) {
  warn "$pid has gone away!";
}
```

---

1 Пусть не точно в таких, но в аналогичных ситуациях. Речь идет о сигналах SIGHUP, SIGCONT, SIGINT и фиктивном сигнале SIGZERO.

2 А вы думали, что нажатие Control-C останавливает вашу программу? На самом деле это просто приводит к отправке сигнала SIGINT, который останавливает программу по умолчанию. Как будет показано позднее, вы можете написать программу, которая вместо аварийного завершения будет выполнять другие действия при получении SIGINT.

3 Чаще всего идентификатор целевого процесса известен, потому что это дочерний процесс, созданный вызовом fork; также возможно получение идентификатора из файла или от внешней программы. Использование внешней программы создает массу сложностей и проблем, поэтому многие программы сохраняют идентификатор своего текущего процесса в файле, о чем обычно говорится в документации программы.

4 Отправка сигнала также завершится неудачей, если вы не являетесь суперпользователем, а процесс принадлежит кому-то другому. Кстати говоря, отправлять SIGINT чужим программам весьма невежливо.

Пожалуй, принимать сигналы немного интереснее, чем отправлять их. Зачем это может быть нужно? Допустим, ваша программа создает временные файлы в каталоге */tmp*, которые удаляются в конце работы программы. Если кто-то нажмет Control-C во время выполнения, при аварийном завершении в */tmp* останется «мусор», а это крайне невежливо. Проблема решается обработчиком сигнала, который позаботится об очистке:

```
my $temp_directory = "/tmp/myprog.$$"; # Каталог для временных файлов
mkdir $temp_directory, 0700 or die "Cannot create $temp_directory: $!";

sub clean_up {
  unlink glob "$temp_directory/*";
  rmdir $temp_directory;
}

sub my_int_handler {
  &clean_up;
  die "interrupted, exiting...\n";
}

$SIG{'INT'} = 'my_int_handler';
.
.   # Время идет, программа работает, в каталоге создаются
.   # временные файлы. Потом кто-то нажимает Control-C.
.
# Конец нормального выполнения
&clean_up;
```

Присваивание элементу специального хеша `%SIG` активизирует обработчик сигнала (пока он не будет снова отменен). Ключом является имя сигнала (без префикса `SIG`), а значением – строка[1] с именем функции (без знака `&`). В дальнейшем при получении сигнала `SIGINT` Perl немедленно прерывает свою текущую работу и передает управление заданной функции. Функция удаляет временные файлы, а затем завершает работу программы. Если никто не нажал Control-C, функция `&clean_up` все равно будет вызвана в конце нормального выполнения программы.

Если функция возвращает управление вместо вызова `die`, выполнение продолжается с того места, на котором оно было прервано. Это может быть полезно, если сигнал должен прервать какую-то операцию без завершения всей программы. Допустим, обработка каждой строки файла занимает несколько секунд, и вы хотите отменить общую обработку при получении сигнала прерывания, но только не на середине обработки строки. Установите флаг в обработчике прерывания и проверяйте его состояние в конце обработки каждой строки:

[1] Также может использоваться ссылка на пользовательскую функцию, но здесь эта тема не рассматривается.

```
my $int_count;
sub my_int_handler { $int_count++ }
$SIG{'INT'} = 'my_int_handler';
...
$int_count = 0;
while (<SOMEFILE>) {
  ... Обработка в течение нескольких секунд ...
  if ($int_count) {
    # Получен сигнал прерывания !
    print "[processing interrupted...]\n";
    last;
  }
}
```

Во время обработки каждой строки значение `$int_count` равно 0. Если никто не нажал Control-C, цикл переходит к следующей строке. Но при поступлении сигнала прерывания обработчик сигнала увеличивает флаг `$int_count`, а это приводит к выходу из цикла при завершающей проверке.

Итак, при получении сигнала можно либо установить флаг, либо завершить работу программы; собственно, на этом возможности обработки сигналов заканчиваются. Впрочем, текущая реализация обработчиков сигналов неидеальна[1]; постарайтесь свести объем выполняемых действий к минимуму, или ваша программа может «упасть» в самый неподходящий момент.

Упражнения

Ответы к упражнениям приведены в приложении А.

1. [6] Напишите программу, которая переходит в некоторый жестко заданный каталог (например, системный корневой каталог) и выполняет команду *ls -l* для получения списка файлов в длинном формате. (Если вы не работаете в UNIX, используйте команду своей системы для получения расширенной информации о содержимом каталога.)

[1] Это один из самых важных пунктов в списке задач разработчиков Perl. Ожидается, что надежная обработка сигналов станет одной из главных особенностей Perl 6. Проблема в том, что сигнал может поступить в любой момент – даже тогда, когда Perl к этому не готов. Если Perl, например, в момент поступления сигнала выделяет блок памяти, а обработчик тоже попытается выделить свой блок памяти, программа умрет. Вы не можете управлять тем, когда код Perl выделяет память, но код XSUB (обычно написанный на C) способен обеспечить безопасную обработку сигналов. За дополнительной информацией по этой теме обращайтесь к документации Perl.

2. [10] Измените предыдущую программу так, чтобы выходные данные команды сохранялись в файле *ls.out* текущего каталога. Информация об ошибках должна сохраняться в файле *ls.err*. (Любой из этих файлов может быть пустым, никакие специальные действия по этому поводу не требуются.)
3. [8] Напишите программу, которая разбирает результат команды *date* для определения текущего дня недели. Для рабочих дней программа выводит сообщение *get to work*, для выходных – сообщение *go play*. Вывод команды *date* начинается с `Mon` для понедельника (Monday[1]). Если в вашей системе команда *date* отсутствует, напишите фиктивную программу, которая просто выдает строку в формате *date*. Вы даже можете воспользоваться нашей программой из двух строк – только не спрашивайте нас, как она работает:

```
#!/usr/bin/perl
print localtime( ) . "\n";
```

[1] Если названия дней недели выводятся на английском языке. При использовании иной записи в вашей системе внесите соответствующие изменения.

17

Расширенные возможности Perl

Материал, который мы рассматривали ранее, составлял основу Perl – то, что должен знать каждый пользователь Perl. Однако существуют и другие возможности. Они не являются строго обязательными, но могут стать весьма ценным дополнением к вашему арсеналу. Самые важные из них были собраны в этой главе.

Пусть вас не смущает название главы; она не сложнее для понимания, чем предыдущие главы. Все описанные возможности являются «расширенными» лишь в том смысле, что они не предназначены для новичков. При первом чтении книги вы можете пропустить (или пролистать) эту главу, чтобы перейти непосредственно к использованию Perl. Вернитесь к ней через месяц-два, когда захотите освоить новые возможности Perl. Считайте эту главу одной большой сноской.[1]

Перехват ошибок в блоках eval

Иногда даже самый обычный, повседневный код приводит к фатальным ошибкам в программе. Любая из следующих типичных команд может стать причиной аварийного завершения:

```
$barney = $fred / $dino;          # Деление на нуль?

print "match\n" if /^($wilma)/;   # Некорректное регулярное выражение?

open CAVEMAN, $fred               # Ошибка пользователя приводит к сбою die?
  or die "Can't open file '$fred' for input: $!";
```

Некоторые из этих ошибок можно выявить заранее, но обнаружить их все невозможно. (Как проверить строку `$wilma` из этого примера и убедиться в том, что она содержит корректное регулярное выражение?)

[1] Мы даже хотели это сделать, но редакторы из O'Reilly отказались наотрез.

К счастью, в Perl имеется простой способ перехвата фатальных ошибок – проверяемый код заключается в блок eval:

```
eval { $barney = $fred / $dino } ;
```

Даже если переменная $dino равна нулю, эта строка не приведет к сбою программы. В действительности eval является выражением (а не управляющей конструкцией, как while или foreach), поэтому точка с запятой в конце блока обязательна.

Если во время выполнения блока eval происходит фатальная (в обычных условиях) ошибка, блок продолжает работать, но программа аварийно не завершается. Таким образом, сразу же после завершения eval желательно проверить, завершился ли блок нормально или произошла фатальная ошибка. Ответ содержится в специальной переменной $@. Если в блоке была перехвачена фатальная ошибка, $@ будет содержать «последние слова» программы – сообщение вида «Недопустимое деление на нуль в my_program строка 12». Если ошибки не было, переменная $@ пуста. Конечно, это означает, что $@ содержит удобный логический признак для проверки (*true* в случае ошибки), поэтому после блоков eval часто встречается команда следующего вида:

```
print "An error occurred: $@" if $@;
```

Блок eval является полноценным блоком, поэтому он определяет новую область видимости для лексических (my) переменных. В следующем фрагменте показан блок eval в действии:

```
foreach my $person (qw/ fred wilma betty barney dino pebbles /) {
  eval {
    open FILE, "<$person"
      or die "Can't open file '$person': $!";

    my($total, $count);

    while (<FILE>) {
      $total += $_;
      $count++;
    }

    my $average = $total/$count;
    print "Average for file $person was $average\n";

    &do_something($person, $average);
  };

  if ($@) {
    print "An error occurred ($@), continuing\n";
  }
}
```

Сколько возможных фатальных ошибок перехватывает этот блок? Если произойдет ошибка при открытии файла, она будет перехвачена. Вычисление среднего арифметического может привести к делению на

нуль, и эта ошибка тоже перехватывается. Даже вызов загадочной функции &do_something защищается от фатальных ошибок, потому что блок eval перехватывает все фатальные ошибки, происходящие во время его активности. (В частности, это будет удобно, если вы вызываете пользовательскую функцию, написанную другим программистом, но не знаете, насколько надежно она написана.)

Если ошибка происходит в ходе обработки одного из файлов, мы получим сообщение об ошибке, но программа спокойно перейдет к следующему файлу.

Блоки eval даже могут вкладываться в другие блоки eval. Внутренний блок перехватывает ошибки во время выполнения, не позволяя им добраться до внешних блоков. (Конечно, после завершения внутреннего блока eval можно «перезапустить» ошибку при помощи die, чтобы она была перехвачена внешним блоком.) Блок eval перехватывает любые ошибки, происходящие во время выполнения, в том числе и ошибки при вызове функций (как показано в предыдущем примере).

Ранее мы уже упоминали о том, что eval является выражением, и поэтому после завершающей фигурной скобки необходима точка с запятой. Но как и любое выражение, eval имеет некоторое возвращаемое значение. При отсутствии ошибок значение вычисляется по аналогии с функциями: как результат последнего вычисленного выражения или как значение, возвращаемое на более ранней стадии необязательным ключевым словом return. А вот другой способ выполнения вычислений, при котором вам не нужно беспокоиться о делении на нуль:

```
my $barney = eval { $fred / $dino };
```

Если eval перехватывает фатальную ошибку, возвращаемое значение представляет собой либо undef, либо пустой список (в зависимости от контекста). Таким образом, в предыдущем примере $barney либо содержит правильный результат деления, либо undef; проверять $@ необязательно (хотя, вероятно, перед дальнейшим использованием $barney стоит включить проверку defined($barney)).

Существуют четыре вида проблем, которые eval перехватить не может. Первую группу составляют очень серьезные ошибки, нарушающие работу Perl, – нехватка памяти или получение необработанного сигнала. Поскольку Perl при этом перестает работать, перехватить эти ошибки он не сможет.[1] Конечно, синтаксические ошибки в блоке eval перехватываются на стадии компиляции – они никогда не возвращаются в $@.

Оператор exit завершает программу немедленно, даже если он вызывается в пользовательской функции в блоке eval. (Отсюда следует, что

[1] Если вам интересно, некоторые из этих ошибок перечислены с кодом (X) в man-странице *perldiag*.

при написании пользовательской функции в случае возникновения проблем следует использовать `die` вместо `exit`.)

Четвертую и последнюю разновидность проблем, не перехватываемых блоком `eval`, составляют предупреждения – пользовательские (из `warn`) или внутренние (включаемые ключом командной строки *-w* или директивой `use warnings`). Для перехвата предупреждений существует специальный механизм, не связанный с `eval`; за информацией обращайтесь к описанию псевдосигнала `__WARN__` в документации Perl.

Стоит также упомянуть еще об одной форме `eval`, которая может быть опасна при неправильном использовании. Более того, некоторые люди считают, что `eval` вообще не следует использовать в программах по соображениям безопасности. Действительно, `eval` следует использовать с осторожностью, но здесь имеется в виду другая форма `eval` – «`eval` для строк». Если же за ключевым словом `eval` сразу же следует блок программного кода в фигурных скобках, беспокоиться не о чем – эта разновидность `eval` безопасна.

Отбор элементов списка

Иногда вас интересует лишь некоторое подмножество элементов из списка. Допустим, из списка чисел необходимо отобрать только нечетные числа или из текстового файла отбираются только те строки, в которых присутствует подстрока `Fred`. Как будет показано в этом разделе, задача отбора элементов списка легко решается при помощи оператора `grep`.

Давайте начнем с первой задачи и выделим нечетные числа из большого списка. Ничего нового для этого нам не понадобится:

```
my @odd_numbers;

foreach (1..1000) {
  push @odd_numbers, $_ if $_ % 2;
}
```

В этом фрагменте используется оператор вычисления остатка (`%`), который был представлен в главе 2. Для четного числа остаток от деления на 2 равен 0, а проверяемое условие ложно. Для нечетного числа остаток равен 1; значение истинно, поэтому в массив заносятся только нечетные числа.

В этом коде нет ничего плохого, разве что он пишется и выполняется немного медленнее, чем следует, потому что в Perl имеется оператор `grep`:

```
my @odd_numbers = grep { $_ % 2 } 1..1000;
```

Этот фрагмент строит список из 500 нечетных чисел всего в одной строке кода. Как он работает? Первый аргумент `grep` содержит блок, в котором переменная `$_` представляет текущий элемент списка. Блок

возвращает логическое значение (*true*/*false*). Остальные аргументы определяют список элементов, в котором выполняется поиск. Оператор grep вычисляет выражение для каждого элемента списка по аналогии с циклом foreach. Элементы, для которых последнее выражение в блоке возвращает истинное значение, включаются в итоговый список grep.

Во время работы grep переменная $_ последовательно представляет один элемент списка за другим. Аналогичное поведение уже встречалось нам в цикле foreach. Как правило, изменять $_ в выражении grep не рекомендуется, потому что это приведет к повреждению исходных данных.

Оператор grep унаследовал свое имя от классической утилиты UNIX, которая отбирает строки из файла по регулярным выражениям. Оператор Perl grep решает ту же задачу, но обладает гораздо большими возможностями. В следующем примере из файла извлекаются только те строки, в которых присутствует подстрока fred:

```
my @matching_lines = grep { /\bfred\b/i } <FILE>;
```

У grep также существует упрощенный вариант синтаксиса. Если в качестве селектора используется простое выражение (вместо целого блока), замените блок этим выражением, за которым следует запятая. В упрощенной записи предыдущий пример выглядит так:

```
my @matching_lines = grep /\bfred\b/i, <FILE>;
```

Преобразование элементов списка

Другая распространенная задача – преобразование элементов списка. Предположим, имеется список чисел, которые необходимо перевести в «денежный формат» для вывода, как в функции &big_money (см. главу 14). Однако исходные данные изменяться не должны; нам нужна измененная копия списка, используемая только для вывода. Одно из возможных решений:

```
my @data = (4.75, 1.5, 2, 1234, 6.9456, 12345678.9, 29.95);
my @formatted_data;

foreach (@data) {
  push @formatted_data, &big_money($_);
}
```

Немного напоминает пример кода, приведенный в начале описания grep, не правда ли? Вероятно, вас не удивит, что альтернативное решение напоминает первый пример с grep:

```
my @data = (4.75, 1.5, 2, 1234, 6.9456, 12345678.9, 29.95);

my @formatted_data = map { &big_money($_) } @data;
```

Оператор `map` похож на `grep`, потому что он получает те же аргументы: блок, в котором используется `$_`, и список элементов для обработки. И функционирует он сходным образом: блок последовательно выполняется для каждого элемента списка, а `$_` при каждой итерации представляет новый элемент списка. Но последнее выражение блока для `map` используется иначе: вместо логического признака оно определяет значение, включаемое в итоговый список.[1] Любую конструкцию с `grep` или `map` можно переписать в виде эквивалентного цикла `foreach` с занесением элементов во временный массив, но короткая запись обычно и эффективнее, и удобнее. Результат `map` и `grep` представляет собой список, что позволяет напрямую передать его другой функции. В следующем примере отформатированные «денежные величины» выводятся в виде списка с отступами под заголовком:

```
print "The money numbers are:\n",
  map { sprintf("%25s\n", $_) } @formatted_data;
```

Конечно, всю обработку можно выполнить «на месте» без использования временного массива `@formatted_data`:

```
my @data = (4.75, 1.5, 2, 1234, 6.9456, 12345678.9, 29.95);
print "The money numbers are:\n",
  map { sprintf("%25s\n", &big_money($_) ) } @data;
```

У `map`, как и у `grep`, существует упрощенная форма синтаксиса. Если в качестве селектора используется простое выражение (вместо полноценного блока), поставьте это выражение, за которым следует запятая, на место блока:

```
print "Some powers of two are:\n",
  map "\t" . ( 2 ** $_ ) . "\n", 0..15;
```

Упрощенная запись ключей хешей

В Perl предусмотрено множество способов сокращенной записи, упрощающих работу программиста. Вот один из них, весьма удобный: некоторые ключи хешей необязательно заключать в кавычки.[2]

Конечно, это возможно не для всех ключей, потому что ключ хеша может представлять собой произвольную строку. Однако ключи часто относительно просты. Если ключ хеша не содержит ничего кроме букв, цифр и символов подчеркивания и не начинается с цифры, кавычки можно опустить. Подобные простые строки без кавычек называются *тривиальными словами* (*barewords*).

---

[1] Еще одно важное отличие заключается в том, что выражение, используемое `map`, вычисляется в списочном контексте и может возвращать любое количество элементов (не обязательно один элемент за одну итерацию).

[2] В этом разделе, кроме собственно кавычек, подразумеваются все остальные способы оформления строк (квотирования). – *Примеч. перев.*

Эта сокращенная запись чаще всего применяется в самом распространенном месте записи ключей хеша: в фигурных скобках ссылки на элемент хеша. Например, вместо `$score{"fred"}` можно написать просто `$score{fred}`. Так как многие ключи хешей достаточно просты, отказ от кавычек действительно удобен. Но помните: если содержимое фигурных скобок не является тривиальным словом, Perl интерпретирует его как выражение.

Ключи хешей также часто встречаются при заполнении всего хеша по списку пар «ключ-значение». Большая стрелка (`=>`) между ключом и значением в этом случае особенно полезна, потому что она автоматически оформляет ключ как строку (и снова только если ключ является тривиальным словом):

```
# Хеш с результатами партий в боулинг
my %score = (
  barney   => 195,
  fred     => 205,
  dino     => 30,
);
```

Здесь проявляется еще одно важное отличие между «большой стрелкой» и запятой; тривиальное слово слева от большой стрелки неявно оформляется как строка (хотя все, что находится справа, остается без изменений). Данная особенность «большой стрелки» может использоваться не только при работе с хешами, однако этот вариант использования является самым распространенным.

Срезы

Часто оказывается, что из всего списка требуется обработать лишь часть элементов. Допустим, библиотека хранит информацию о своих читателях в большом файле. Каждая строка файла описывает одного читателя в виде шести полей, разделенных символом «`:`»: имя, номер библиотечной карточки, домашний адрес, домашний телефон, рабочий телефон и количество выданных книг. Небольшой фрагмент файла выглядит примерно так:

```
fred flintstone:2168:301 Cobblestone Way:555-1212:555-2121:3
barney rubble:709918:3128 Granite Blvd:555-3333:555-3438:0
```

Одному из библиотечных приложений нужен только номер библиотечной карточки и количество выданных книг; остальные данные в его работе не используются. Конечно, необходимые поля можно получить следующим образом:

```
while (<FILE>) {
  chomp;
  my @items = split /:/;
  my($card_num, $count) = ($items[1], $items[5]);
```

```
    ...  # Теперь работаем с этими двумя переменными
}
```

Но массив `@items` больше ни для чего не нужен; он выглядит совершенно лишним.[1] Может быть, результат `split` лучше будет присвоить списку скаляров:

```
my($name, $card_num, $addr, $home, $work, $count) = split /:/;
```

Нам удалось обойтись без ненужного массива `@items`, но теперь в программе появились четыре ненужные скалярные переменные. В таких ситуациях некоторые программисты используют фиктивные имена вида `$dummy_1`, показывающие, что после разбивки строки значение этого элемента не используется. Но Ларри решил, что это слишком хлопотно, поэтому в Perl был добавлен специальный способ использования `undef`. «Присваивание» элемента списка значению `undef` говорит о том, что соответствующий элемент исходного списка попросту игнорируется:

```
my(undef, $card_num, undef, undef, undef, $count) = split /:/;
```

Удобнее? Да, из программы исчезают ненужные переменные. Впрочем, есть и недостаток: теперь нам придется считать `undef`, чтобы определить, какому элементу соответствует `$count`. А при большом количестве элементов в списке решение становится слишком громоздким. Например, некоторые программисты, которые хотели извлечь из данных *stat* только значение `mtime`, писали код следующего вида:

```
my(undef, undef, undef, undef, undef, undef, undef,
  undef, undef, $mtime) = stat $some_file;
```

Стоит ошибиться с подсчетом `undef`, и вместо `mtime` вы получите `atime` или `ctime`; такие ошибки весьма усложняют отладку. Но существует более удобное решение: Perl может индексировать списки по аналогии с массивами. Результат называется *срезом* (*slice*) *списка*. Значение `mtime` является элементом с индексом 9 в списке, возвращаемом *stat*[2], и его можно получить по соответствующему индексу:

```
my $mtime = (stat $some_file)[9];
```

Список элементов (в данном случае возвращаемое значение *stat*) должен быть заключен в круглые скобки. Если попытаться записать команду в таком виде, она работать не будет:

```
my $mtime = stat($some_file)[9];  # Синтаксическая ошибка!
```

---

[1] На самом деле не таким уж лишним... Но продолжайте читать. Подобные решения часто используются программистами, не понимающими, что такое срезы, и вам будет полезно ознакомиться с ними в этом разделе.

[2] Элемент стоит в десятой позиции, но его индекс равен 9, так как первый элемент имеет индекс 0. Здесь также используется индексация с отсчетом от нуля, уже знакомая нам по работе с массивами.

Срез состоит из списка в круглых скобках и следующего за ним индекса в квадратных скобках. Круглые скобки, в которые заключаются аргументы функций, не подходят.

Но вернемся к библиотеке: список, с которым мы работаем, является возвращаемым значением `split`. Мы можем воспользоваться синтаксисом срезов для извлечения элементов 1 и 5 с индексами:

```
my $card_num = (split /:/)[1];
my $count = (split /:/)[5];
```

Нельзя сказать, что подобные срезы в скалярном контексте (извлечение одного элемента из списка) чем-то плохи, но было бы проще и эффективнее, если бы нам не приходилось вызывать `split` дважды. Мы и не будем делать это дважды; оба значения можно получить, используя срез списка в списочном контексте:

```
my($card_num, $count) = (split /:/)[1, 5];
```

Команда извлекает из списка элементы с индексами 1 и 5, возвращая их в виде списка из двух элементов. Присваивая этот список двум переменным `my`, мы получаем ровно то, что хотели. Срез выполняется только один раз, и значения сразу двух переменных задаются в простой и понятной записи.

Срезы часто оказываются самым простым способом извлечения нескольких элементов из списка. В следующем примере из списка извлекается первый и последний элемент; при этом мы используем тот факт, что индекс -1 соответствует последнему элементу:[1]

```
my($first, $last) = (sort @names)[0, -1];
```

Индексы могут следовать в произвольном порядке и даже повторяться. В этом примере извлекаются 5 элементов списка, содержащего 10 элементов:

```
my @names = qw{ zero one two three four five six seven eight nine };
my @numbers = ( @names )[ 9, 0, 2, 1, 0 ];
print "Bedrock @numbers\n";  # Bedrock nine zero two one zero
```

Срезы массивов

Предыдущий пример можно сделать еще проще. При создании срезов на базе массивов (вместо списков) круглые скобки необязательны. Следовательно, срез может выглядеть так:

```
my @numbers = @names[ 9, 0, 2, 1, 0 ];
```

[1] Возможно, сортировка списка только для определения наибольшего и наименьшего элемента не самое эффективное решение. Но сортировка в Perl работает достаточно быстро, так что в общем случае такой способ приемлем (при условии, что длина списка не превышает нескольких сотен элементов).

Дело не сводится только к отсутствию круглых скобок; в действительности здесь используется другая запись обращения к элементам – срез массива. Ранее (в главе 3) мы говорили, что символ @ в @names означает «все элементы». В Perl символ $ означает отдельное значение, а символ @ – список значений.

Срез всегда является списком, и в синтаксисе среза массивов на это обстоятельство указывает знак @. Когда вы встречаете в программе Perl конструкцию вида @names[...], сделайте то, что делает Perl, и обратите внимание на знак @ в начале и на квадратные скобки в конце. Квадратные скобки означают, что происходит индексирование массива, а знак @ – что вы получаете список[1] элементов вместо одного значения (на которое бы указывал знак $). См. рис. 17.1.

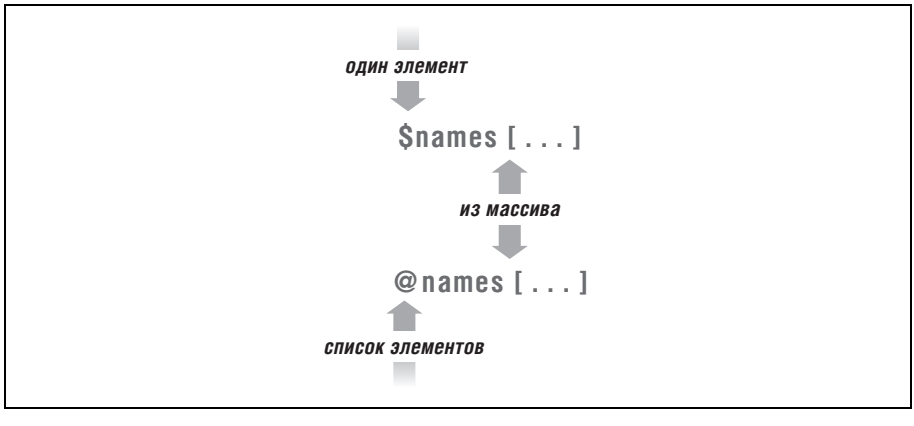


Рис. 17.1. *Срезы массивов и отдельные элементы*

Знак перед ссылкой на переменную ($ или @) определяет контекст индексного выражения. С префиксом $ выражение вычисляется в скалярном контексте. Но если в начале стоит знак @, индексное выражение вычисляется в списочном контексте для получения списка индексов.

Таким образом, @names[2, 5] означает то же самое, что ($names[2], $names[5]). Если вам нужен список значений, используйте запись со срезом массива. Везде, где требуется использовать список, вы можете использовать более простой срез массива.

Однако срезы также могут использоваться в одном месте, где простые списки использоваться не могут, – срезы могут интерполироваться прямо в строку:

---

[1] Конечно, список не обязан содержать более одного элемента – более того, он вообще может быть пустым.

```
my @names = qw{ zero one two three four five six seven eight nine };
print "Bedrock @names[ 9, 0, 2, 1, 0 ]\n";
```

При попытке интерполировать `@names` выводятся элементы массива, разделенные пробелами. Если вместо этого интерполировать `@names[ 9, 0, 2, 1, 0 ]`, вы получите только указанные элементы, разделенные пробелами.[1] Вернемся к примеру с библиотекой. Допустим, программа обновляет адрес и телефон одного из читателей, потому что он только что переехал в новый дом. Если информация о нем поставляется в виде списка `@items`, обновление двух элементов массива может выполняться примерно так:

```
my $new_home_phone = "555-6099";
my $new_address = "99380 Red Rock West";
@items[2, 3] = ($new_address, $new_home_phone);
```

И снова срез массива обеспечивает более компактную запись, чем список элементов. В этом случае последняя строка эквивалентна присваиванию `($items[2], $items[3])`, но она более компактна и эффективна.

Срезы хешей

В полной аналогии со срезами массивов часть элементов хеша тоже может быть выделена в *срез хеша*. Помните пример с хранением результатов игры в боулинг в хеше `%score`? Мы можем извлечь эти результаты как в список элементов хеша, так и в срез. Эти два приема эквивалентны (хотя второй работает эффективнее и занимает меньше места):

```
my @three_scores = ($score{"barney"}, $score{"fred"}, $score{"dino"});

my @three_scores = @score{ qw/ barney fred dino/ };
```

Срез всегда является списком, и в записи среза хеша это обстоятельство указывается знаком `@`.[2] Когда вы встречаете в программе Perl конструкцию вида `@score{ ... }`, сделайте то, что делает Perl, и обратите внимание на знак `@` в начале и фигурные скобки в конце. Фигурные скобки означают, что происходит выборка из хеша, а знак `@` – что вы получаете список элементов вместо одного (на которое бы указывал знак `$`). См. рис. 17.2.

Как и в случае со срезами массивов, знак перед ссылкой на переменную (`$` или `@`) определяет контекст индексного выражения. С префиксом `$` выражение вычисляется в скалярном контексте для получения от-

---

[1] А точнее элементы списка разделяются содержимым переменной Perl `$"`, которая по умолчанию содержит пробел. Обычно изменять ее не рекомендуется. При интерполяции списка значений во внутренней реализации Perl выполняется операция `join $", @list`, где `@list` – списочное выражение.

[2] Кажется, мы повторяемся, но мы просто хотим подчеркнуть одно важное обстоятельство: срезы хешей аналогичны срезам массивов. Или вам не кажется? Тогда повторим: срезы хешей аналогичны срезам массивов.

Рис. 17.2. *Срезы хешей и отдельные элементы*

дельного ключа.[1] Но если в начале стоит знак @, индексное выражение вычисляется в списочном контексте для получения списка ключей.

Возникает логичный вопрос: раз мы говорим о хешах, почему здесь не используется знак %? Этот знак обозначает весь хеш; срез хеша (как и любой другой срез) всегда представляет собой список, а не хеш. В Perl символ $ всегда обозначает отдельный объект данных, символ @ обозначает список, а символ % обозначает весь хеш.

Как вы видели со срезами массивов, срезы хешей могут использоваться вместо соответствующего списка элементов хеша во всех синтаксических конструкциях Perl. Таким образом, мы можем задать результаты некоторых партий в хеше (без изменения других элементов хеша) следующим образом:

```
my @players = qw/ barney fred dino /;
my @bowling_scores = (195, 205, 30);
@score{ @players } = @bowling_scores;
```

Последняя строка работает точно так же, как если бы мы выполнили присваивание списку из трех элементов:

```
($score{"barney"}, $score{"fred"}, $score{"dino"}).
```

Срезы хешей тоже могут интерполироваться. В следующем примере выводятся результаты для выбранной нами тройки игроков:

```
print "Tonight's players were: @players\n";
print "Their scores were: @score{@players}\n";
```

[1] Существует одно исключение, которое вам вряд ли встретится, во всяком случае в современном коде Perl. См. описание $; в man-странице *perlvar*.

Упражнения

Ответы к упражнениям приведены в приложении А.

1. [30] Напишите программу, которая читает список строк из файла (по одной строке). Затем пользователь вводит шаблоны, совпадения которых ищутся в строках. Для каждого шаблона программа должна сообщить количество строк, содержащих совпадения, а также вывести сами строки. Файл не должен читаться заново для каждого шаблона; храните строки в памяти. Имя файла можно жестко закодировать в программе. Если шаблон недействителен (например, если он содержит непарные круглые скобки), программа должна просто вывести сообщение об ошибке и разрешить пользователю ввести другой шаблон. Если пользователь вместо шаблона вводит пустую строку, программа завершает работу. (Если вам понадобится тестовый файл с множеством интересных строк, возьмите файл *sample_text* из примеров, которые вы, несомненно, давно загрузили с сайта O'Reilly; см. предисловие.)

A

Ответы к упражнениям

В этом приложении приведены ответы на все упражнения, помещенные в конце каждой главы.

Глава 2

1. Одно из возможных решений:

```
#!/usr/bin/perl -w
$pi = 3.141592654;
$circ = 2 * $pi * 12.5;
print "The circumference of a circle of radius 12.5 is $circ.\n";
```

Программа запускается стандартной строкой `#!`; в вашей системе может использоваться другой путь к Perl. В первой строке также включаются предупреждения.

Первая строка «настоящего» кода задает переменной `$pi` значение числа π. Хороший программист использует подобные «псевдоконстанты»[1] по нескольким причинам: на ввод числа 3.141592654 в программе уходит время, особенно если оно встречается несколько раз. Если вы случайно введете в одном месте 3.141592654, а в другом 3.14159, это может привести к погрешностям вычислений. Чтобы узнать, не ввели ли вы случайно 3.141952654 (отчего ваш космический зонд полетит не на ту планету), достаточно проверить всего одну строку. Имя `$pi` легче вводится, чем π, особенно если в вашей системе не поддерживается Юникод. Наконец, программу

[1] Если вы предпочитаете «полноценные» константы, воспользуйтесь директивой `constant`.

будет проще сопровождать, если число π вдруг изменится.[1] Затем программа вычисляет длину окружности, сохраняет ее в `$circ` и выводит сообщение. Сообщение завершается символом новой строки, как и любая строка в выводе нормальной программы. Без символа новой строки результат будет выглядеть примерно так (в зависимости от приглашения командного процессора):

```
The circumference of a circle of radius 12.5 is
78.53981635.bash-2.01$[]
```

2. Одно из возможных решений:

```
#!/usr/bin/perl -w
$pi = 3.141592654;
print "What is the radius? ";
chomp($radius = <STDIN>);
$circ = 2 * $pi * $radius;
print "The circumference of a circle of radius $radius is $circ.\n";
```

Программа очень похожа на предыдущую, но на этот раз мы запрашиваем радиус у пользователя, а затем применяем его везде, где прежде использовалось фиксированное значение 12.5. Если бы мы были предусмотрительны при написании первой программы, то включили бы в нее переменную `$radius`. Обратите внимание на обработку введенной строки функцией `chomp`. Математическая формула работала бы и без нее, потому что строка вида `12.5\n` без всяких проблем преобразуется в число 12.5. Однако выводимое сообщение выглядело бы так:

```
The circumference of a circle of radius 12.5
 is 78.53981635.
```

Символ новой строки остался в `$radius`, хотя переменная использовалась как число. Так как между `$radius` и словом `is` в команде `print` стоит пробел, вторая строка вывода начинается с пробела. Мораль: вызывайте `chomp` для вводимых данных, если только у вас нет причин этого не делать.

3. Одно из возможных решений:

```
#!/usr/bin/perl -w
$pi = 3.141592654;
print "What is the radius? ";
chomp($radius = <STDIN>);
$circ = 2 * $pi * $radius;
if ($radius < 0) {
  $circ = 0;
```

[1] Больше века назад оно едва не было изменено законодательным актом штата Индиана. См. законопроект №246, Indiana State Legislature, 1897, *http://www.cs.uwaterloo.ca/~alopez-o/math-faq/node45.html*.

```
}
print "The circumference of a circle of radius $radius is $circ.\n";
```

В этой версии добавилась проверка радиуса. Даже если заданный радиус недействителен, возвращаемое значение хотя бы не будет отрицательным. С таким же успехом можно сначала обнулить отрицательный радиус, а потом использовать его при вычислении длины окружности; есть разные способы. Собственно, фраза «Это можно сделать несколькими способами» считается девизом Perl. Именно поэтому ответ на каждое упражнение начинается со слов «Одно из возможных решений».

4. Одно из возможных решений:

```
print "Enter first number: ";
chomp($one = <STDIN>);
print "Enter second number: ";
chomp($two = <STDIN>);
$result = $one * $two;
print "The result is $result.\n";
```

Обратите внимание: в этом ответе строка `#!` отсутствует. С этого момента мы будем считать, что вы уже знаете, для чего она нужна, и не будем вставлять ее в каждую программу.

Возможно, имена переменных выбраны не лучшим образом. В большой программе посторонний программист, занимающийся ее сопровождением, может решить, что переменная `$two` содержит значение 2. В короткой программе это несущественно, но в большой программе переменным следовало бы присвоить более содержательные имена, например `$first_response` и `$second_response`.

Если мы забудем вызвать `chomp` для переменных `$one` и `$two`, в этой программе ничего не изменится, потому что переменные все равно не используются в строковом виде. Но если на следующей неделе другой программист изменит программу так, чтобы она выводила сообщение вида `The result of multiplying $one by $two is $result.\n`, эти коварные символы новой строки напомнят о своем существовании. Еще раз: вызывайте `chomp`, если только у вас нет веских причин для обратного — как в следующем упражнении.

5. Одно из возможных решений:

```
print "Enter a string: ";
$str = <STDIN>;
print "Enter a number of times: ";
chomp($num = <STDIN>);
$result = $str x $num;
print "The result is:\n$result";
```

Программа в определенном смысле очень похожа на предыдущую: строка «умножается» на заданное число. Соответственно мы сохранили структуру предыдущего примера. Однако в этом случае первое

вводимое значение – строка – не обрабатывается вызовом `chomp`, потому что в упражнении сказано, что дубликаты должны выводиться в разных строках. Таким образом, если пользователь введет `fred` с символом новой строки и число 3, после каждого экземпляра `fred` будет выведен символ новой строки, как и требуется.

В завершающей команде `print` перед `$result` добавлен символ новой строки, потому что мы хотим, чтобы вывод начинался в отдельной строке. Другими словами, результат не должен выглядеть так, чтобы выравнивались только два экземпляра `fred` из трех:

```
The result is: fred
fred
fred
```

При этом добавлять еще один символ новой строки в конец вывода не нужно, так как `$result` и так завершается переводом строки.

Обычно Perl не следит за дополнительными пробелами в ваших программах; хотите – используйте пробелы, не хотите – не используйте. Но будьте внимательны, чтобы это не привело к искажению смысла! Если пропустить пробел между `x` и предшествующим именем переменной `$str`, Perl «увидит» имя `$strx`, и программа работать не будет.

Глава 3

1. Одно из возможных решений:

```
print "Enter some lines, then press Ctrl-D:\n"; # or maybe Ctrl-Z
@lines = <STDIN>;
@reverse_lines = reverse @lines;
print @reverse_lines;
```

...или еще проще:

```
print "Enter some lines, then press Ctrl-D:\n";
print reverse <STDIN>;
```

Большинство программистов Perl предпочтет второй вариант, если, конечно, список строк не нужно сохранять для будущего использования.

2. Одно из возможных решений:

```
@names = qw/ fred betty barney dino wilma pebbles bamm-bamm /;
print "Enter some numbers from 1 to 7, one per line, then press Ctrl-D:\n";
chomp(@numbers = <STDIN>);
foreach (@numbers) {
  print "$names[ $_ - 1 ]\n";
}
```

Введенные номера уменьшаются на 1, чтобы пользователь мог считать от 1 до 7, хотя индексы массива лежат в диапазоне от 0 до 6. Того же результата можно добиться другим способом – включением фиктивного элемента в массив @names:

```
@names = qw/ dummy_item fred betty barney dino wilma pebbles bamm-bamm /;
```

Если вы проверили, что выбор пользователя лежит в диапазоне от 1 до 7, поставьте себе дополнительный балл.

3. Одно из возможных решений, если весь вывод должен размещаться в одной строке:

```
chomp(@lines = <STDIN>);
@sorted = sort @lines;
print "@sorted\n";
```

...вариант с выводом в разных строках:

```
print sort <STDIN>;
```

Глава 4

1. Одно из возможных решений:

```
sub total {
  my $sum;  # private variable
  foreach (@_) {
    $sum += $_;
  }
  $sum;
}
```

В этой функции переменная $sum используется для накопления суммы. В начале выполнения $sum, как и все новые переменные, содержит undef. Затем цикл foreach перебирает список параметров (из @_), используя $_ в качестве управляющей переменной. (Напомним, что массив параметров @_ сам по себе никак не связан с $_, управляющей переменной циклов foreach по умолчанию.)

При первой итерации foreach первое число (в $_) прибавляется к $sum. Конечно, $sum содержит undef, ведь переменная еще не была инициализирована. Но так как мы используем ее в числовом контексте (Perl узнает об этом по числовому оператору +=), Perl считает, что переменная уже была инициализирована 0. Perl прибавляет первый параметр к 0 и заносит сумму обратно в $sum.

При следующей итерации к переменной $sum, уже не равной undef, прибавляется следующий параметр. Сумма помещается обратно в $sum; так происходит со всеми параметрами. Наконец, последняя строка возвращает $sum вызывающей стороне.

В функции присутствует потенциальная ошибка. Предположим, функция вызывается с пустым списком параметров (мы рассматри-

вали эту возможность с переписанной функцией &max в тексте главы). В этом случае переменная $sum останется равной undef; это значение и будет возвращено функцией. Но в данном случае, вероятно, в качестве суммы пустого списка будет правильнее вернуть 0, а не undef. (Конечно, если вы хотите, чтобы сумма пустого списка отличалась от суммы (3, -5, 2), правильнее вернуть undef.)

Если вы не хотите, чтобы функция возвращала неопределенное значение, проблема легко решается. Просто инициализируйте $sum нулем вместо значения по умолчанию undef:

```
my $sum = 0;
```

Теперь функция будет всегда возвращать число, даже при пустом списке параметров.

2. Одно из возможных решений:

```
# Не забудьте включить &total из предыдущего упражнения!
print "The numbers from 1 to 1000 add up to ", total(1..1000), ".\n";
```

Так как функцию нельзя вызывать из строки в кавычках[1], результат вызова передается print в отдельном элементе. Сумма должна быть равна 500500 – приятное круглое число. При этом программа выполняется почти мгновенно; передача списка из 1000 параметров – вполне рядовая ситуация для Perl.

3. Одно из возможных решений:

```
sub average {
  if (@_ == 0) { return }
  my $count = @_;
  my $sum = total(@_);                # Из предыдущего упражнения
  $sum/$count;
}

sub above_average {
  my $average = average(@_);
  my @list;
  foreach my $element (@_) {
    if ($element > $average) {
      push @list, $element;
    }
  }
  @list;
}
```

В average при пустом списке параметров происходит обычный возврат управления без конкретного значения. Вызывающая сторона получает undef[2]; это говорит о том, что для пустого списка среднее

[1] Вернее, нельзя без особо изощренных выкрутасов. В Perl трудно найти что-то *абсолютно* невозможное.

[2] Или пустой список, если &average используется в списочном контексте.

значение не определено. Если список не пуст, среднее значение вычисляется функцией &total. В принципе можно даже обойтись без временных переменных для $sum and $count, но они упрощают чтение кода.

Вторая функция, above_average, просто строит и возвращает список нужных элементов. (Почему управляющая переменная $element применяется вместо переменной по умолчанию $_?) Обратите внимание на использованный во второй функции другой способ решения проблемы пустого списка параметров.

4. Имя последнего человека запоминается в переменной состояния. Сначала переменная равна undef; по этому признаку мы определяем, что первое приветствие обращено к Fred. В конце функции текущее значение $name сохраняется в $last_name для использования при следующем вызове:

```
use 5.010;

greet( 'Fred' );
greet( 'Barney' );

sub greet {
    state $last_person;

    my $name = shift;

    print "Hi $name! ";

    if( defined $last_person ) {
        print "$last_person is also here!\n";
        }
    else {
        print "You are the first one here!\n";
        }

    $last_person = $name;
    }
```

5. Этот ответ похож на предыдущее упражнение, но на этот раз сохраняются все встречавшиеся ранее имена. Вместо скалярной переменной мы объявляем статическую переменную @names и заносим в список все имена:

```
use 5.010;

greet( 'Fred' );
greet( 'Barney' );
greet( 'Wilma' );
greet( 'Betty' );

sub greet {
    state @names;

    my $name = shift;
```

```
print "Hi $name! ";

if( @names ) {
    print "I've seen: @names\n";
    }
else {
    print "You are the first one here!\n";
    }

push @names, $name;
}
```

Глава 5

1. Одно из возможных решений:

```
print reverse <>;
```

Проще не придумаешь! Однако это решение работает; функция `print` ищет список строк, который она получает, вызывая `reverse` в списочном контексте. А `reverse` ищет список строк для перестановки, который она получает, используя оператор `<>` в списочном контексте. Таким образом, `<>` возвращает список всех строк из всех файлов, заданных пользователем. Теперь `reverse` переставляет элементы списка в обратном порядке, а `print` выводит результат.

2. Одно из возможных решений:

```
print "Enter some lines, then press Ctrl-D:\n";  # Или Ctrl-Z
chomp(my @lines = <STDIN>);

print "1234567890" x 7, "12345\n";  # Линейка до столбца 75

foreach (@lines) {
  printf "%20s\n", $_;
}
```

Сначала программа читает все строки текста и обрабатывает их вызовом `chomp`. Далее выводится «линейка» с разметкой позиций. Так как линейка используется исключительно в целях отладки, в готовой программе эту строку следует закомментировать. Конечно, для создания линейки нужной длины можно было бы повторить `"1234567890"` или даже воспользоваться копированием/вставкой текста, но мы выбрали этот способ, потому что он смотрится более эффектно.

Далее цикл `foreach` перебирает элементы списка и выводит каждую строку с преобразованием `%20s`. При желании также можно создать формат для вывода всего списка без использования цикла:

```
my $format = "%20s\n" x @lines;
printf $format, @lines;
```

В этом месте многие программисты допускают ошибку, из-за которой создаются столбцы шириной 19 символов. Это происходит в тот момент, когда вы говорите себе[1]: «А зачем вызывать `chomp`, если символы новой строки все равно еще понадобятся?» Вы выкидываете `chomp` и используете формат `%20s` (без символа новой строки)[2]... И вывод почему-то смещается на один пробел. Что произошло?

Проблема возникает тогда, когда Perl подсчитывает пробелы, необходимые для создания нужного количества столбцов. Если пользователь вводит `hello` с символом новой строки, Perl видит шесть символов вместо пяти, потому что символ новой строки тоже считается. Соответственно он выводит 14 пробелов и строку из 6 символов – вы же сами потребовали 20 символов в формате `%20s`.

Конечно, Perl не проверяет содержимое строки при определении ширины; проверяется только непосредственное количество символов. Символ новой строки (или другой специальный символ, скажем, табуляция или нуль-символ) нарушает его подсчеты.[3]

3. Одно из возможных решений:

```
print "What column width would you like? ";
chomp(my $width = <STDIN>);

print "Enter some lines, then press Ctrl-D:\n";  # или Ctrl-Z
chomp(my @lines = <STDIN>);

print "1234567890" x (($width+9)/10), "\n";      # Вывод линейки

foreach (@lines) {
  printf "%${width}s\n", $_;
}
```

Это решение очень похоже на предыдущее, но сначала программа запрашивает ширину столбца. Ширина вводится до основных данных, потому что после получения признака конца файла ввести дополнительные данные не удастся (по крайней мере, в некоторых системах). Конечно, в реальном приложении конец вводимых данных лучше обозначить специальным маркером, как вы увидите в ответах к следующим упражнениям.

По сравнению с предыдущим упражнением также изменилась команда вывода линейки. Мы используем дополнительные вычисления для построения линейки, длина которой, по крайней мере, не меньше необходимой. Убедиться в том, что эти вычисления верны, – еще одна дополнительная задача. (Подсказка: попробуйте рассмотреть длины 50 и 51 и помните, что правосторонний операнд `x` усекается, а не округляется.)

---

[1] Или Ларри, если он стоит поблизости.

[2] Если Ларри вас не остановит.

[3] Как Ларри уже должен был вам объяснить.

Для построения формата на этот раз используется выражение "%${width}s\n", которое интерполирует $width. Фигурные скобки необходимы для отделения имени от служебного символа s; без них будет интерполироваться неверная переменная $widths. Впрочем, если вы забудете, как использовать фигурные скобки в этом контексте, та же форматная строка может быть создана выражением вида '%' . $width . "s\n".

Значение $width напоминает еще об одном случае, в котором вызов chomp абсолютно необходим. Если не обработать $width вызовом chomp, полученная форматная строка будет иметь вид "%30\ns\n". Пользы от нее не будет.

Программисты, использовавшие функцию printf, могут предложить другое решение. Поскольку функция printf пришла из языка C, строковая интерполяция в ней не употребляется, и мы можем использовать трюк, знакомый программистам C. Если заменить числовое поле ширины в преобразовании звездочкой (*), то будет применено значение из списка параметров:

```
printf "%*s\n", $width, $_;
```

Глава 6

1. Одно из возможных решений:

```
my %last_name = qw{
  fred flintstone
  barney rubble
  wilma flintstone
};
print "Please enter a first name: ";
chomp(my $name = <STDIN>);
print "That's $name $last_name{$name}.\n";
```

В этом примере мы используем список qw// (с фигурными скобками в качестве ограничителя) для инициализации хеша. Для простых наборов данных такое решение удобно, к тому же оно упрощает сопровождение программы – каждый элемент данных состоит из простого имени и фамилии, никаких хитростей. Но если данные могут содержать пробелы (скажем, если в библиотеку захочет записаться Роберт де Ниро), этот простой способ не подойдет. Возможно, вы предпочтете задать каждую пару «ключ–значение» в отдельной строке:

```
my %last_name;
$last_name{"fred"} = "flintstone";
$last_name{"barney"} = "rubble";
$last_name{"wilma"} = "flintstone";
```

Обратите внимание: если вы решите объявить хеш с ключевым словом `my` (например, из-за действия директивы `use strict`), объявление должно предшествовать присваиванию каких-либо элементов. Нельзя использовать `my` только с частью переменной, как в этом примере:

```
my $last_name{"fred"} = "flintstone";  # Ошибка!
```

Оператор `my` работает только с целыми переменными и никогда с отдельными элементами массивов и хешей. Раз уж мы заговорили о лексических переменных, возможно, вы заметили, что лексическая переменная `$name` объявляется внутри вызова функции `chomp`. Это довольно распространенная практика – объявлять переменные непосредственно перед использованием, как в данном случае.

Вызов `chomp` абсолютно необходим еще в одной ситуации. Если пользователь введет строку из пяти символов `"fred\n"` и мы забудем вызвать `chomp`, то в хеше будет разыскиваться элемент `"fred\n"`, которого там нет. Конечно, сам по себе вызов `chomp` еще не обеспечивает абсолютной надежности; если пользователь введет `"fred \n"` (с завершающим пробелом), программа не поймет, что он имел в виду `fred`.

Если вы добавили проверку присутствия ключа в хеше, чтобы при ошибке выводилось соответствующее сообщение, поставьте себе дополнительный балл.

2. Одно из возможных решений:

```
my(@words, %count, $word);      # Объявления переменных (необязательно)
chomp(@words = <STDIN>);

foreach $word (@words) {
  $count{$word} += 1;           # или $count{$word} = $count{$word} + 1;
}

foreach $word (keys %count) {  # или sort keys %count
  print "$word was seen $count{$word} times.\n";
}
```

В этом фрагменте все переменные объявляются в самом начале. Программисты с опытом работы на других языках (например, Pascal), в которых переменные всегда объявляются «наверху», сочтут этот вариант более знакомым, чем объявление переменных по мере надобности. Конечно, мы объявляем эти переменные только из-за предположения о действии директивы `use strict`; по умолчанию в Perl объявления переменных необязательны.

Затем мы используем оператор построчного ввода `<STDIN>` в списочном контексте, чтобы прочитать все входные строки в `@words`, после чего обрабатываем их вызовом `chomp`. В результате `@words` содержит список слов из ввода (при условии, что каждое слово вводится в отдельной строке).

Затем первый цикл foreach перебирает все слова. Цикл содержит самую важную команду во всей программе – команду, которая увеличивает $count{$word} на 1, а затем помещает результат обратно в $count{$word}. Хотя при записи этой команды можно использовать как длинную, так и короткую запись (с оператором +=), короткая запись чуть эффективнее, так как Perl приходится искать $word в хеше только один раз.[1] Для каждого слова в первом цикле foreach значение $count{$word} увеличивается на 1. Таким образом, для первого слова fred значение $count{"fred"} увеличивается на 1. Конечно, так как элемент $count{"fred"} используется впервые, он содержит undef. Но поскольку мы интерпретируем его как число (с числовым оператором += или с оператором + при длинной записи), Perl автоматически преобразует undef в 0. Сумма 1 сохраняется в $count{"fred"}.

Допустим, при следующей итерации foreach встречается слово barney. Мы увеличиваем $count{"barney} на 1, также повышая его с undef до 1.

Теперь предположим, что следующее слово снова содержит fred. При увеличении на 1 значения $count{"fred"}, уже равного 1, мы получаем 2. Значение сохраняется в $count{"fred"}; счетчик показывает, что слово fred встретилось дважды.

При завершении первого цикла foreach мы знаем, сколько раз встретилось каждое слово. В хеше присутствует ключ для каждого (уникального) слова из входных данных, а соответствующее значение показывает, сколько раз оно встретилось.

Итак, второй цикл foreach перебирает ключи хеша, представляющие уникальные слова во входных данных. В цикле каждое уникальное слово встречается один раз. Для каждого слова выводится сообщение с количеством вхождений.

Чтобы задача стала более интересной, можно отсортировать ключи перед их выводом. Если выходной список содержит более десятка элементов, обычно его стоит отсортировать; это поможет программисту, отлаживающему программу, быстро найти нужный элемент.

3. Одно из возможных решений:

```
my $longest = 0;
foreach my $key ( keys %ENV ) {
    my $key_length = length( $key );
    $longest = $key_length if $key_length > $longest;
```

[1] Кроме того, по крайней мере в некоторых версиях Perl, короткая запись также предотвращает выдачу предупреждения о неопределенном значении, которое может быть выдано с длинной записью. Предупреждение можно обойти, увеличивая переменную оператором ++, но мы этот оператор еще не рассматривали.

```
    }

foreach my $key ( sort keys %ENV ) {
    printf "%-${longest}s  %s\n", $key, $ENV{$key};
    }
```

В первом цикле `foreach` программа перебирает все ключи и определяет их длину функцией `length`. Если полученная длина больше хранящейся в `$longest`, последняя заменяется новым значением.

После перебора всех ключей функция `printf` выводит ключи и значения в два столбца. При этом используется тот же прием, упоминавшийся ранее в упражнении 3 главы 5, – интерполяция `$longest` в форматную строку.

Глава 7

1. Одно из возможных решений:

```
while (<>) {
  if (/fred/) {
    print;
  }
}
```

Здесь все просто. Самая важная часть этого упражнения – запуск программы для тестовых строк. Для `Fred` совпадение не находится; это говорит о том, что регулярные выражения учитывают регистр символов. (Позже мы покажем, как изменить этот параметр поиска.) В строках `frederick` и `Alfred` совпадения обнаруживаются, поскольку каждая из этих строк содержит подстроку `fred`, состоящую из четырех символов. Также существует возможность поиска по целым словам, так что в строках `frederick` и `Alfred` совпадения обнаружены не будут, но об этом позже.

2. Одно из возможных решений:

   Замените шаблон в предыдущем упражнении на `/[fF]red/`. Также возможны варианты `/(f|F)red/` или `/fred|Fred/`, но символьный класс более эффективен.

3. Одно из возможных решений:

   Замените шаблон в первом упражнении на `/\./`. Обратная косая черта необходима, потому что точка является метасимволом. Также можно использовать символьный класс: `/[.]/`.

4. Одно из возможных решений:

   Замените шаблон в первом упражнении на `/[A-Z][a-z]+/`.

5. Одно из возможных решений:

   Замените шаблон в первом упражнении на `/(\S)\1/`. Символьный класс `\S` совпадает с одним символом, не являющимся пропуском,

а круглые скобки позволяют использовать обратную ссылку \1 для обозначения того же символа, следующего непосредственно за ним.

6. Одно из возможных решений:

```
while (<>) {
  if (/wilma/) {
    if (/fred/) {
      print;
    }
  }
}
```

Совпадение для /fred/ ищется только после успешно найденного совпадения /wilma/, но fred может находиться в строке как до, так и после wilma; проверки не зависят друг от друга.

Если вы предпочитаете обойтись без лишней вложенной проверки if, можно использовать такую запись:[1]

```
while (<>) {
  if (/wilma.*fred|fred.*wilma/) {
    print;
  }
}
```

Такое решение работает, потому что мы проверяем оба возможных случая – как wilma перед fred, так и fred перед wilma. С записью вида /wilma.*fred/ мы бы не нашли совпадение в строке вида fred and wilma flintstone, хотя в ней встречаются оба имени.

Мы предложили это упражнение «на повышенную оценку», потому что у многих читателей здесь возникают психологические затруднения. Мы описали операцию OR (вертикальная черта, |), но нигде не упоминали операцию AND. Дело в том, что в регулярных выражениях такой операции нет.[2] Если вы хотите знать, совпадают ли в строке два шаблона, просто проверьте оба.

---

[1] Программисты, знакомые с логическим оператором AND (см. главу 10), могут выполнить обе проверки /fred/ и /wilma/ в одной конструкции if. Такое решение более эффективно, лучше масштабируется...и вообще во всех отношениях лучше приведенных. Но мы еще не рассматривали логический оператор AND.

[2] Однако существуют весьма хитроумные и сложные способы выполнения того, что можно назвать «операцией AND». Впрочем, в общем случае они уступают по эффективности логическому оператору AND Perl – в зависимости от того, какие оптимизации сможет применить ядро регулярных выражений Perl.

Глава 8

1. Простое решение существует, и оно было представлено в тексте главы. Если ваша программа не выводит текст `before<match>after`, как должна, значит, вы выбрали сложное решение.
2. Одно из возможных решений:

```
/a\b/
```

   (Конечно, это шаблон, который должен использоваться в тестовой программе!) Если ваш шаблон ошибочно находит совпадение в `barney`, вероятно, вы забыли включить якорь границы слова.

3. Одно из возможных решений:

```
#!/usr/bin/perl
while (<STDIN>) {
  chomp;
  if (/(\b\w*a\b)/) {
    print "Matched: |$`<$&>$'|\n";
    print "\$1 contains '$1'\n";        # Новая строка вывода
  } else {
    print "No match: |$_|\n";
  }
}
```

   Перед нами та же самая тестовая программа (с новым шаблоном), в которую была добавлена помеченная строка для вывода `$1`.

   В этом шаблоне используется пара якорей границы слова[1] `\b` в круглых скобках, хотя шаблон работает точно так же, как если бы они находились снаружи. Дело в том, что якоря соответствуют позиции строки, а не символам; они имеют «нулевую ширину».

4. Ответ к этому упражнению практически повторяет предыдущий, отличается только регулярное выражение:

```
#!/usr/bin/perl

use 5.010;

while (<STDIN>) {
  chomp;
  if (/(?<word>\b\w*a\b)/) {
    print "Matched: |$`<$&>$'|\n";
    print "'word' contains '$+{word}'\n";        # Новая строка вывода
  } else {
```

[1] Откровенно говоря, первый якорь необязателен из-за некоторых тонкостей работы максимальных квантификаторов, которые мы рассматривать не будем. Но его присутствие немного повышает эффективность поиска и очевидно делает программу более понятной; последнее обстоятельство оказывается решающим.

```
    print "No match: |$_|\n";
  }
}
```

5. Одно из возможных решений:

```
m!
  (\b\w*a\b)       # $1: Слово, завершающееся буквой a
  (.{0,5})         # $2: до пяти символов после него
!xs                # модификаторы /x и /s
```

(Не забудьте добавить код вывода переменной `$2`. Если вы захотите снова вернуться к одной переменной, достаточно закомментировать лишнюю строку.) Если ваш шаблон не совпадает в простой строке `wilma`, возможно, вы требуете обязательного присутствия одного и более символов (вместо нуля и более символов). Возможно, вы пропустили модификатор `/s`, потому что в данных вроде бы не должно быть символов новой строки.

6. Одно из возможных решений:

```
while (<>) {
  chomp;
  if (/\s+$/) {
    print "$_#\n";
  }
}
```

В качестве маркера используется знак `#`.

Глава 9

1. Одно из возможных решений:

```
/($what){3}/
```

После интерполяции `$what` создается шаблон вида `/(fred|barney){3}/`. Без круглых скобок мы бы получили шаблон `/fred|barney{3}/`, эквивалентный `/fred|barneyyy/`. Таким образом, круглые скобки необходимы.

2. Одно из возможных решений:

```
my $in = $ARGV[0];
unless (defined $in) {
  die "Usage: $0 filename";
}

my $out = $in;
$out =~ s/(\.\w+)?$/.out/;

unless (open IN, "<$in") {
  die "Can't open '$in': $!";
}
```

```
unless (open OUT, ">$out") {
  die "Can't write '$out': $!";
}

while (<IN>) {
  s/Fred/Larry/gi;
  print OUT $_;
}
```

Программа начинается с определения переменной для единственного параметра командной строки; если параметр не задан, программа аварийно завершается. Затем параметр копируется в `$out`, а расширение файла (если оно есть) заменяется на `.out`. (Впрочем, также будет достаточно просто присоединить `.out` к имени файла.)

После открытия файловых дескрипторов `IN` и `OUT` начинается содержательная часть программы. Если вы не использовали оба модификатора `/g` и `/i`, сбросьте себе половину балла, потому что в файле должны заменяться все вхождения `fred` и `Fred`.

3. Одно из возможных решений:

```
while (<IN>) {
  chomp;
  s/Fred/\n/gi;         # Замена всех вхождений FRED
  s/Wilma/Fred/gi;      # Замена всех вхождений WILMA
  s/\n/Wilma/g;         # Замена временных заполнителей
  print OUT "$_\n";
}
```

Конечно, этот фрагмент заменяет цикл из предыдущей программы. Для выполнения парных замен такого рода потребуется временная строка-заполнитель, которая не встречается в данных. Предшествующий вызов `chomp` позволяет использовать символ новой строки (`\n`) в качестве временного заполнителя. Вместо него также можно применить другую маловероятную строку, например нуль-символ `\0`.

4. Одно из возможных решений:

```
$^I = ".bak";           # Создание резервной копии
while (<>) {
  if (/^#!/) {          # Строка с решеткой?
    $_ .= "## Copyright (C) 20XX by Yours Truly\n";
  }
  print;
}
```

Запустите программу для всех файлов, которые требуется обновить. Например, если файлам упражнений присвоены имена *ex01-1*, *ex01-2* и т. д. и все они начинаются с *ex*..., команда будет выглядеть так:

```
./fix_my_copyright ex*
```

5. Чтобы предотвратить повторное включение информации об авторских правах, файлы необходимо обрабатывать за два прохода. Сначала мы создаем хеш, ключами которого являются имена файлов, а значения могут быть любыми (мы будем использовать значение 1 для удобства):

```
my %do_these;
foreach (@ARGV) {
  $do_these{$_} = 1;
}
```

Затем программа просматривает файлы и удаляет из хеша те, которые уже содержат информацию об авторских правах. Текущее имя файла хранится в переменной $ARGV, которую мы можем использовать в качестве ключа хеша:

```
while (<>) {
  if (/^## Copyright/) {
    delete $do_these{$ARGV};
  }
}
```

Наконец, после построения сокращенного списка файлов в @ARGV следует уже знакомый код:

```
@ARGV = sort keys %do_these;
$^I = ".bak";           # Создание резервной копии
while (<>) {
  if (/^#!/) {          # Строка с решеткой?
    $_ .= "## Copyright (c) 20XX by Yours Truly\n";
  }
  print;
}
```

Глава 10

1. Одно из возможных решений:

```
my $secret = int(1 + rand 100);
# Следующую строку можно раскомментировать во время отладки
# print "Don't tell anyone, but the secret number is $secret.\n";

while (1) {
  print "Please enter a guess from 1 to 100: ";
  chomp(my $guess = <STDIN>);
  if ($guess =~ /quit|exit|^\s*$/i) {
    print "Sorry you gave up. The number was $secret.\n";
    last;
  } elsif ($guess < $secret) {
    print "Too small. Try again!\n";
  } elsif ($guess == $secret) {
    print "That was it!\n";
    last;
```

```
    } else {
      print "Too large. Try again!\n";
    }
}
```

В первой строке выбирается «секретное число» от 1 до 100. Вот как это происходит: функция `rand` генерирует случайные числа, и вызов `rand 100` дает случайное число от 0 до (но не включая) 100. Другими словами, наибольшее значение выражения равно примерно 99.999.[1] Прибавление 1 дает число от 1 до 100.999, после чего функция `int` усекает результат, оставляя нужное нам число от 1 до 100.

Закомментированная строка может пригодиться во время разработки и отладки или при вашем желании сжульничать в игре. Основной код программы состоит из бесконечного цикла `while`. В этом цикле пользователь пытается отгадать число, пока не будет выполнена команда `last`.

Все возможные строки должны проверяться перед числами. Почему? Предположим, этот порядок проверки будет нарушен; что произойдет, если пользователь введет `quit`? Команда будет интерпретирована как число (возможно, с выдачей предупреждения, если предупреждения были включены). «Число» интерпретируется как нуль, и бедный пользователь получит сообщение, что число слишком маленькое. Возможно, до проверки строк дело даже и не дойдет.

Другой способ создания бесконечного цикла основан на использовании простейшего блока с командой `redo`. По эффективности он не лучше и не хуже; это всего лишь другой способ записи. Если цикл с большей вероятностью выполняется, нежели пропускается, лучше использовать `while` (так как эта форма записи выполняется по умолчанию). Если выполнение блока скорее является не правилом, а исключением, лучше используйте простейший блок.

2. Эта программа является слегка измененной версией предыдущего ответа. «Секретное число» должно выводиться только на стадии разработки, поэтому мы выводим его, если переменная `$Debug` имеет истинное значение. Переменная `$Debug` содержит либо значение, ранее заданное в переменной среды, либо 1 по умолчанию. Используя оператор `//`, мы задаем ей значение 1 только в том случае, если переменная `$ENV{DEBUG}` не определена:

```
use 5.010;

my $Debug = $ENV{DEBUG} // 1;

my $secret = int(1 + rand 100);

print "Don't tell anyone, but the secret number is $secret.\n"
    if $Debug;
```

[1] На самом деле наибольшее возможное значение зависит от системы; если вас действительно интересует эта тема, обратитесь к документу *http://www.cpan.org/doc/FMTEYEWTK/random*.

То же можно сделать и без использования специфических возможностей Perl 5.10, просто придется выполнить чуть больше работы:

```
my $Debug = defined $ENV{DEBUG} ? $ENV{DEBUG} : 1;
```

3. Одно из возможных решений, позаимствовавшее ряд идей из ответа к упражнению 3 в главе 6.

   В начале программы задаются некоторые переменные среды. Ключи ZERO и EMPTY имеют ложные, но определенные значения, а ключ UNDEFINED не определен.

   Позднее в списке аргументов printf оператор // используется для выбора строки (undefined) только в том случае, если $ENV{$key} имеет неопределенное значение:

```
use 5.010;

$ENV{ZERO}      = 0;
$ENV{EMPTY}     = '';
$ENV{UNDEFINED} = undef;

my $longest = 0;
foreach my $key ( keys %ENV )
    {
    my $key_length = length( $key );
    $longest = $key_length if $key_length > $longest;
    }

foreach my $key ( sort keys %ENV )
    {
    printf "%-${longest}s  %s\n", $key, $ENV{$key} // "(undefined)";
    }
```

   Оператор // обеспечивает правильную интерпретацию ложных значений, например, содержащихся в ключах ZERO и EMPTY.

   Чтобы сделать то же самое без Perl 5.10, используйте тернарный оператор:

```
printf "%-${longest}s  %s\n", $key,
    $ENV{$key} ? $ENV{$key} : "(undefined)";
```

Глава 11

1. В этом ответе используется ссылка на хеш (эта тема обсуждается в книге «Intermediate Perl»[1] (O'Reilly)), но мы привели соответствующую часть кода, чтобы избавить вас от хлопот. Вам не нужно знать, как работает этот код, – достаточно знать, что он работает. Вы добиваетесь желаемого результата, а подробности узнаете уже потом.

---

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

Одно из возможных решений:

```
#!/usr/bin/perl

use Module::CoreList;

my %modules = %{ $Module::CoreList::version{5.006} };

print join "\n", keys %modules;
```

Глава 12

1. Одно из возможных решений:

```
foreach my $file (@ARGV) {
  my $attribs = &attributes($file);
  print "'$file' $attribs.\n";
}

sub attributes {
  # Вывод атрибутов файла
  my $file = shift @_;
  return "does not exist" unless -e $file;

  my @attrib;
  push @attrib, "readable" if -r $file;
  push @attrib, "writable" if -w $file;
  push @attrib, "executable" if -x $file;
  return "exists" unless @attrib;
  'is ' . join " and ", @attrib;  # Возвращаемое значение
}
```

Здесь тоже будет удобно применить пользовательскую функцию. Главный цикл выводит строку атрибутов для каждого файла, например сообщая нам, что файл `'cereal-killer'` является исполняемым, а `'sasquatch'` не существует.

Функция выдает информацию об атрибутах для заданного имени файла. Конечно, если файл не существует, другие проверки не нужны, поэтому мы начинаем именно с этой проверки. Если файл не существует, функция сразу возвращает управление.

Если файл существует, мы строим список атрибутов. (Начислите себе лишние баллы, если вы использовали в этих проверках специальный дескриптор _ вместо `$file`, чтобы предотвратить многократные обращения к системе за каждым новым атрибутом.) Функция легко дополняется новыми проверками по аналогии с тремя, приведенными здесь. Но что произойдет, если ни один из атрибутов не является истинным? Даже если о файле нельзя сказать ничего больше, по крайней мере, мы знаем, что он существует, поэтому в конце выводится соответствующее сообщение. Конструкция `unless` использует тот факт, что значение `@attrib` интерпретируется как истина (в логическом контексте как особом случае скалярного контекста), если

массив содержит хотя бы один элемент. Если массив содержит атрибуты, мы объединяем их связкой «and» и ставим в начало «is », чтобы описание стало более понятным и удобочитаемым.

Если в командной строке имена файлов не указаны, программа ничего не выводит. Это вполне логично; тот, кто запрашивает информацию о нуле файлов, получает нуль строк данных. Но сравните это решение с тем, которое используется следующей программой в аналогичном случае.

2. Одно из возможных решений:

```
die "No file names supplied!\n" unless @ARGV;
my $oldest_name = shift @ARGV;
my $oldest_age = -M $oldest_name;

foreach (@ARGV) {
  my $age = -M;
  ($oldest_name, $oldest_age) = ($_, $age)
    if $age > $oldest_age;
}

printf "The oldest file was %s, and it was %.1f days old.\n",
  $oldest_name, $oldest_age;
```

Программа сразу сообщает о том, что в командной строке не было передано ни одного имени файла. Ведь программа должна сообщить нам имя самого старого файла, а при отсутствии файлов его и быть не может.

В программе применяется уже знакомый нам «алгоритм водяного горизонта». Несомненно, первый файл – самый старый из всех, которые встречались до настоящего момента. Его «возраст» сохраняется в переменной `$oldest_age`.

Для каждого из остальных файлов программа определяет его «возраст» проверкой `-M`, подобно тому, как это делалось для первого файла (разве что здесь используется аргумент по умолчанию `$_`). Под «возрастом» обычно понимается время последней модификации, хотя вы можете использовать другую временную метку. Если «возраст» текущего файла превышает `$oldest_age`, имя и «возраст» обновляются списочным присваиванием. Списочное присваивание в данном случае необязательно, но это удобный способ одновременного обновления нескольких переменных.

«Возраст», полученный при помощи `-M`, сохраняется во временной переменной `$age`. А что произошло бы, если бы мы каждый раз использовали `-M` вместо переменной? Прежде всего, без использования специального файлового дескриптора `_` мы бы каждый раз обращались к операционной системе за информацией о возрасте файла, а это относительно медленная операция (впрочем, замедление может быть реально заметно лишь при обработке сотен или тысяч файлов). Но есть и другое, более важное обстоятельство: представь-

те, что кто-то изменит файл во время его проверки. Сначала мы видим «возраст» файла, наибольший из всех встретившихся нам. Но затем непосредственно перед повторным использованием -M файл модифицируется и временная метка заменяется текущим временем. В результате значение, сохраненное в $oldest_age, на самом деле представляет *наименьший* возможный «возраст» файла. Таким образом, файл фактически исключается из общей проверки. Эту ошибку будет очень сложно найти в процессе отладки!

Наконец, в завершающей части программы мы используем printf для вывода имени и «возраста», округленного до ближайшей десятой части дня. Запишите себе дополнительный балл, если вы не поленились преобразовать «возраст» в дни, часы и минуты.

3. Одно из возможных решений:

```
use 5.010;

say "Looking for my files that are readable and writable";

die "No files specified!\n" unless @ARGV;

foreach my $file ( @ARGV ) {
    say "$file is readable and writable" if -o -r -w $file;
    }
```

Сгруппированные проверки файлов поддерживаются только в Perl 5.10, поэтому программа начинается с директивы use, проверяющей правильность версии Perl. Если @ARGV не содержит элементов, программа аварийно завершается, в противном случае эти элементы перебираются в цикле foreach.

В программе необходимо использовать три оператора проверки файлов: -o проверяет владельца, -r проверяет возможность чтения, -w проверяет возможность записи. Группировка -o -r -w образует составное условие, которое выполняется только при выполнении всех трех условий.

Все это можно сделать и в версии Perl, предшествующей 5.10, хотя и с чуть большим объемом кода. Команды say заменяются командами print с добавленными символами новой строки, а сгруппированные проверки заменяются отдельными условиями, объединенными оператором &&:

```
print "Looking for my files that are readable and writable\n";

die "No files specified!\n" unless @ARGV;

foreach my $file ( @ARGV ) {
    print "$file is readable and writable\n"
        if( -w $file && -r _ && -o _ );
    }
```

Глава 13

1. Одно из возможных решений с использованием глоба:

```
print "Which directory? (Default is your home directory) ";
chomp(my $dir = <STDIN>);
if ($dir =~ /^\s*$/) {          # A blank line
  chdir or die "Can't chdir to your home directory: $!";
} else {
  chdir $dir or die "Can't chdir to '$dir': $!";
}

my @files = <*>;
foreach (@files) {
  print "$_\n";
}
```

Сначала программа выводит приглашение и читает содержимое каталога, обрабатывая данные вызовом `chomp` по мере надобности (без `chomp` программа попытается перейти к каталогу, имя которого завершается символом новой строки; в UNIX это разрешено, поэтому функция `chdir` не может просто проигнорировать этот символ).

Затем, если имя каталога не пусто, программа переходит в этот каталог; сбои приводят к аварийному завершению. Если имя каталога пусто, вместо него выбирается домашний каталог.

Наконец, глоб `<*>` получает все имена файлов в новом рабочем каталоге, отсортированные по алфавиту. Полученные имена последовательно выводятся командой `print`.

2. Одно из возможных решений:

```
print "Which directory? (Default is your home directory) ";
chomp(my $dir = <STDIN>);
if ($dir =~ /^\s*$/) {          # Пустая строка
  chdir or die "Can't chdir to your home directory:
$!";
} else {
  chdir $dir or die "Can't chdir to '$dir': $!";
}

my @files = <.* *>;         ## Теперь с включением .*
foreach (sort @files) {     ## Теперь с сортировкой
  print "$_\n";
}
```

Два отличия от предыдущей программы: во-первых, в глоб включен шаблон `.*`, совпадающий со всеми именами, начинающимися с точки. Во-вторых, полученный список теперь необходимо отсортировать, потому что некоторые имена, начинающиеся с точки, должны быть вставлены в соответствующую позицию списка без начальной точки.

3. Одно из возможных решений:

```
print "Which directory? (Default is your home directory) ";
chomp(my $dir = <STDIN>);
if ($dir =~ /^\s*$/) {          # Пустая строка
  chdir or die "Can't chdir to your home directory:
$!";
} else {
  chdir $dir or die "Can't chdir to '$dir': $!";
}

opendir DOT, "." or die "Can't opendir dot: $!";
foreach (sort readdir DOT) {
  # next if /^\./; ##  если пропускаем файлы с точкой
  print "$_\n";
}
```

Здесь используется та же структура, что и в двух предыдущих программах, но на этот раз мы выбрали решение с дескриптором каталога. После смены рабочего каталога мы открываем текущий каталог, представленный дескриптором `DOT`.

Почему `DOT` (то есть «точка»)? Если пользователь запросит абсолютное имя каталога (например, */etc*), с открытием проблем не будет. Но давайте посмотрим, что произойдет для относительного имени (например, `fred`). Сначала функция `chdir` переходит к каталогу `fred`, который затем открывается вызовом `opendir`. Но в этом случае имя `fred` будет относиться к новому, а не к исходному каталогу. Только имя «`.`» всегда обозначает «текущий каталог» (по крайней мере, в UNIX и других сходных системах).

Функция `readdir` читает все имена файлов из каталога. Прочитанные данные сортируются и выводятся программой. Если бы это решение было применено в первом упражнении, нам пришлось бы пропустить «файлы с точкой»; для этого достаточно раскомментировать строку в цикле `foreach`.

Возможно, вы спрашиваете себя: «А зачем было начинать с вызова `chdir`? Функцию `readdir` со всем прочим можно использовать с любым каталогом, не только с текущим». Прежде всего потому, что мы хотели предоставить пользователю удобное средство для перехода к домашнему каталогу одним нажатием клавиши. Однако эта программа также может послужить отправной точкой для создания утилиты управления файлами. Например, на следующем этапе пользователь может указать, какие файлы в этом каталоге должны быть перемещены на архивную ленту.

4. Одно из возможных решений:

```
unlink @ARGV;
```

...или если вы хотите предупредить пользователя о возможных проблемах:

```
foreach (@ARGV) {
  unlink $_ or warn "Can't unlink '$_': $!, continuing...\n";
}
```

Каждый аргумент командной строки помещается в `$_` и используется при вызове `unlink`. Если при удалении вдруг возникнут проблемы, предупреждение сообщит, что произошло.

5. Одно из возможных решений:

```
use File::Basename;
use File::Spec;
my($source, $dest) = @ARGV;

if (-d $dest) {
  my $basename = basename $source;
  $dest = File::Spec->catfile($dest, $basename);
}

rename $source, $dest
  or die "Can't rename '$source' to '$dest': $!\n";
```

Вся основная работа в этой программе выполняется последней командой, но предшествующий код необходим для перемещения в другой каталог. После объявления используемых модулей аргументам командной строки присваиваются удобные имена. Если `$dest` является каталогом, необходимо извлечь базовое имя из `$source` и присоединить его к каталогу (`$dest`). Когда аргумент `$dest` примет нужный вид, функция `rename` делает все остальное.

6. Одно из возможных решений:

```
use File::Basename;
use File::Spec;

my($source, $dest) = @ARGV;

if (-d $dest) {
  my $basename = basename $source;
  $dest = File::Spec->catfile($dest, $basename);
}

link $source, $dest
  or die "Can't link '$source' to '$dest': $!\n";
```

Как сообщается в подсказке, эта программа очень похожа на предыдущую. Существенное отличие только одно: вызов `rename` заменяется вызовом `link`. Если ваша система не поддерживает жесткие ссылки, последнюю команду можно заменить следующей:

```
print "Would link '$source' to '$dest'.\n";
```

7. Одно из возможных решений:

```
use File::Basename;
use File::Spec;
```

```
my $symlink = $ARGV[0] eq '-s';
shift @ARGV if $symlink;

my($source, $dest) = @ARGV;
if (-d $dest) {
  my $basename = basename $source;
  $dest = File::Spec->catfile($dest, $basename);
}

if ($symlink) {
  symlink $source, $dest
    or die "Can't make soft link from '$source' to '$dest': $!\n";
} else {
  link $source, $dest
    or die "Can't make hard link from '$source' to '$dest': $!\n";
}
```

Начальные строки кода (после двух директив `use`) проверяют первый аргумент командной строки. Если он содержит `-s`, значит, создается символическая ссылка, и это обстоятельство обозначается истинным значением `$symlink`. Аргумент `-s` удаляется из списка в следующей строке. Следующий фрагмент напрямую скопирован из ответов к предыдущим упражнениям. Наконец, в зависимости от состояния `$symlink` выбирается тип создаваемой ссылки (символическая или жесткая). Также изменяется аварийное сообщение; в него включается информация о том, какой тип ссылки мы пытались создать.

8. Одно из возможных решений:

```
foreach (<.* *>) {
  my $dest = readlink $_;
  print "$_ -> $dest\n" if defined $dest;
}
```

Каждый элемент каталога, полученный в результате применения глоба, последовательно помещается в `$_`. Если элемент является символической ссылкой, функция `readlink` возвращает определенное значение, и ссылка выводится программой. В противном случае условие оказывается ложным, и команда вывода пропускается.

Глава 14

1. Одно из возможных решений:

```
my @numbers;
push @numbers, split while <>;
foreach (sort { $a <=> $b } @numbers) {
  printf "%20g\n", $_;
}
```

Вторая строка выглядит совершенно невразумительно, не так ли? Мы сделали это нарочно. Хотя мы рекомендуем писать понятный код, некоторым людям нравится, когда их код сложен для понимания[1], поэтому будьте готовы к худшему. Когда-нибудь вам придется сопровождать запутанный чужой код.

Так как в строке используется модификатор `while`, она эквивалентна следующему циклу:

```
while (<>) {
  push @numbers, split;
}
```

Уже лучше, но все еще непонятно. Цикл `while` читает входные данные по одной строке (из источников, заданных пользователем, на что указывает оператор `<>`), а `split` по умолчанию разбивает его по пропускам, создавая список слов или в данном случае список чисел. Не забывайте, что входные данные представляют поток чисел, разделенных пропусками. Какой бы вариант записи вы ни выбрали, цикл `while` помещает все числа из входных данных в `@numbers`.

Цикл `foreach` берет отсортированный список и выводит каждое число в отдельной строке, используя числовой формат `%20g` для выравнивания чисел по правому краю столбца. Также можно было использовать формат `%20s`. Что бы от этого изменилось? Так как формат является строковым, все строки остались бы в выводе в неизменном виде. Вы обратили внимание, что во входных данных присутствуют 1.50 и 1.5, 04 и 4? При выводе в строковом формате дополнительные нули останутся в выходных данных, а с числовым форматом `%20g` равные числа будут одинаково отображаться в выводе. Строго говоря, правильным может считаться любой из двух форматов – все зависит от того, что вы собираетесь сделать.

2. Одно из возможных решений:

```
# Не забудьте включить хеш %last_name
# из текста упражнения или из загруженного файла

my @keys = sort {
  "\L$last_name{$a}" cmp "\L$last_name{$b}"  # По фамилии
   or
  "\L$a" cmp "\L$b"                         # По имени
} keys %last_name;
```

[1] Конечно, мы не рекомендуем так поступать в *обычном* программировании, но написание запутанного кода можно превратить в занятную игру. Также бывает познавательно взять чей-нибудь пример запутанного кода и потратить пару выходных, пытаясь разобраться в том, что он делает. Если вас интересуют примеры такого кода или вам понадобится помощь с расшифровкой, расспросите участников следующей встречи Perl Mongers. Или поищите JAPH-программы в Интернете, или попробуйте самостоятельно расшифровать блок запутанного кода в конце ответов этой главы.

```
foreach (@keys) {
  print "$last_name{$_}, $_\n";              # Rubble,Bamm-Bamm
}
```

Решение понятно без пояснений: мы размещаем ключи в нужном порядке, а затем выводим их. Мы выбрали формат «фамилия-запятая-имя» просто для развлечения (в описании упражнения этот выбор предоставлялся на ваше усмотрение).

3. Одно из возможных решений:

```
print "Please enter a string: ";
chomp(my $string = <STDIN>);
print "Please enter a substring: ";
chomp(my $sub = <STDIN>);

my @places;

for (my $pos = -1; ; ) {                    # Необычный цикл for
  $pos = index($string, $sub, $pos + 1);    # Поиск следующей позиции
  last if $pos == -1;
  push @places, $pos;
}

print "Locations of '$sub' in '$string' were: @places\n";
```

Программа начинается достаточно просто: пользователю предлагается ввести исходные строки, после чего объявляется массив для хранения позиций подстрок. Но в цикле `for` снова встречается код, «оптимизированный для умных»; подобные трюки допустимы только для развлечения, никогда не используйте их в реальном коде. Впрочем, здесь продемонстрирован вполне допустимый прием, который может быть полезен в некоторых случаях, так что давайте посмотрим, как он работает.

Лексическая переменная `$pos` объявляется приватной в области видимости цикла `for` с начальным значением `-1`. Сразу скажем, что в ней будет храниться позиция подстроки в большей строке. Секции проверки и приращения в цикле `for` оставлены пустыми, так что цикл будет бесконечным. (Конечно, рано или поздно его выполнение будет прервано командой `last`.)

Первая команда в теле цикла ищет первое вхождение подстроки, начиная с позиции `$pos + 1`. Это означает, что при первой итерации, когда переменная `$pos` все еще равна `-1`, поиск начнется с позиции 0 (то есть от начала строки). Индекс вхождения подстроки сохраняется в `$pos`. Если он равен `-1`, цикл поиска завершен, и команда `last` прерывает его выполнение. Если значение отлично от `-1`, позиция сохраняется в `@places`, а цикл переходит к следующей итерации. На этот раз значение `$pos + 1` означает, что поиск начнется от позиции, в которой было найдено предыдущее вхождение. В результате мы благополучно находим все вхождения подстрок.

Если вам не нравится хитроумное использование цикла `for`, аналогичного результата можно добиться и так:

```
{
  my $pos = -1;
  while (1) {
    ... # Тело цикла совпадает с предыдущим решением
  }
}
```

Внешний простейший блок ограничивает область видимости `$pos`. Вообще говоря, он необязателен, но переменные обычно следует объявлять с наименьшей возможной областью видимости. Это означает, что в любой точке программы будет «существовать» меньше переменных, а следовательно, снижается вероятность случайного использования имени `$pos` для других целей. По тем же причинам, если переменная не объявляется в наименьшей области видимости, ей обычно присваивается более длинное имя, снижающее вероятность ее случайного повторного использования. Вероятно, в данном примере стоит выбрать имя вида `$substring_position`.

С другой стороны, если вы стараетесь намеренно запутать свой код (пусть вам будет стыдно!), создайте чудовищную конструкцию следующего вида (пусть нам будет стыдно!):

```
for (my $pos = -1; -1 !=
  ($pos = index
    +$string,
    +$sub,
    +$pos
    +1
  );
push @places, ((((+$pos)))) {
    'for ($pos != 1; # ;$pos++) {
      print "position $pos\n";#;';#' } pop @places;
}
```

Исходный «странный» цикл `for` заменяется этим – еще более странным – кодом. Вероятно, к этому моменту ваши знания уже позволят вам самостоятельно расшифровать этот код... или дополнительно запутать его, чтобы поразить друзей и привести в замешательство врагов. Пожалуйста, используйте свою силу только в благих целях.

Да, и что же вы получили при поиске `t` в строке `This is a test.`? Позиции 10 и 13. Позиция 0 в ответ не входит; регистр символов различается, поэтому подстрока не совпадает.

Глава 15

1. Одна из возможных переработанных версий программы с угадыванием чисел из главы 10. Умное сравнение использовать необязательно, но мы применяем конструкцию `given`:

```
use 5.010;

my $Verbose = $ENV{VERBOSE} // 1;

my $secret = int(1 + rand 100);

print "Don't tell anyone, but the secret number is $secret.\n"
    if $Verbose;

LOOP: {

    print "Please enter a guess from 1 to 100: ";
    chomp(my $guess = <STDIN>);

    my $found_it = 0;

    given( $guess ) {
        when( ! /^\d+$/ )    { say "Not a number!" }
        when( $_ > $secret ) { say "Too high!"   }
        when( $_ < $secret ) { say "Too low!"    }
        default              { say "Just right!"; $found_it++ }
        }

    last LOOP if $found_it;
    redo LOOP;

}
```

В первой секции `when` мы проверяем, что введено именно число. Если введенная строка содержит символы, отличные от цифр, или просто является пустой строкой, это позволяет предотвратить выдачу предупреждений при последующих числовых сравнениях.

Обратите внимание: мы не включили `last` в блок `default`. Вообще-то сначала мы именно так и сделали, однако это приводит к выдаче предупреждения в Perl 5.10.0 (возможно, это предупреждение исчезнет в будущих версиях).

2. Одно из возможных решений:

```
use 5.010;

for (1 .. 105) {
    my $what = '';
    given ($_) {
        when (not $_ % 3) { $what .= ' fizz'; continue }
        when (not $_ % 5) { $what .= ' buzz'; continue }
        when (not $_ % 7) { $what .= ' sausage' }
    }
    say "$_$what";
}
```

3. Одно из возможных решений:

```
use 5.010;

for( @ARGV )
    {
```

```
say "Processing $_";

when( ! -e ) { say "\tFile does not exist!" }
when( -r _ ) { say "\tReadable!"; continue }
when( -w _ ) { say "\tWritable!"; continue }
when( -x _ ) { say "\tExecutable!"; continue }
}
```

Решение обходится без `given`, так как условия `when` можно разместить прямо в блоке `for`. Сначала мы проверяем, что файл существует (а вернее, что он *не* существует). При выполнении первого блока `when` программа сообщает, что файл не существует, а неявно включаемая команда `break` предотвратит выполнение всех остальных условий. Во втором блоке `when` мы проверяем, что файл доступен для чтения, при помощи `-r`. Также в нем используется специальный виртуальный дескриптор `_`, содержащий кэшированную информацию от последнего вызова `stat` (функции, используемой проверками для получения информации). Без дескриптора `_` тоже можно обойтись, но программа станет чуть менее эффективной. Блок `when` завершается командой `continue`, поэтому следующее условие `when` тоже будет проверено.

4. Одно из возможных решений, использующее `given` и умное сравнение:

```
use 5.010;

say "Checking the number <$ARGV[0]>";

given( $ARGV[0] ) {
    when( ! /^\d+$/ ) { say "Not a number!" }

    my @divisors = divisors( $_ );

    my @empty;

    when( @divisors ~~ @empty ) { say "Number is prime" }

    default { say "$_ is divisible by @divisors" }
    }

sub divisors {
    my $number = shift;

    my @divisors = ();
    foreach my $divisor ( 2 .. $number/2 ) {
        push @divisors, $divisor unless $number % $divisor;
        }

    return @divisors;
    }
```

Сначала программа сообщает, с каким числом она работает; всегда полезно лишний раз убедиться в этом. Значение `$ARGV[0]` заключается в угловые скобки, чтобы отделить его от остальных символов строки.

Конструкция given содержит пару блоков when, рядом с которыми размещаются другие команды. Первое условие when проверяет, что введенная строка является числом; для этого к ней применяется регулярное выражение, совпадающее с последовательностью цифр. Если регулярное выражение не совпадает, программа выводит сообщение о том, что строка не является числом. Условие when содержит неявную команду break, которая прерывает выполнение given. Если проверка успешно пройдена, вызывается функция divisors(). То же самое можно было сделать и вне given, но если программа выполняется для нечисловой строки (например, если пользователь ввел строку 'Fred'), Perl выдаст предупреждение. Наш способ предотвращает выдачу предупреждения, а when выполняет функции сторожевого условия.

После получения делителей мы хотим знать, содержит ли массив @divisors хотя бы один элемент. Конечно, можно было бы использовать массив в скалярном контексте для получения количества элементов, но по условиям необходимо использовать умные сравнения. Как известно, два сравниваемых массива считаются равными только в том случае, если они содержат одни и те же элементы, следующие в одинаковом порядке. Мы создаем пустой массив @empty, не содержащий ни одного элемента. При сравнении с @divisors умное сравнение завершится успешно только при отсутствии делителей. Если это условие истинно, выполняется блок when, который также содержит неявную команду break.

Наконец, если число не является простым, выполняется блок default, который выводит количество делителей.

И небольшое добавление, о котором в этой книге упоминать не следовало бы, потому что ссылки будут рассматриваться только в «Intermediate Perl»[1]: при проверке @divisors на наличие элементов мы выполнили лишнюю работу, создавая пустой именованный массив для сравнения. С таким же успехом можно было воспользоваться анонимным массивом и обойтись без лишнего шага:

```
when( @divisors ~~ [] ) { ... }
```

5. Одно из возможных решений, созданное на базе ответа к предыдущему упражнению:

```
use 5.010;

say "Checking the number <$ARGV[0]>";
my $favorite = 42;

given( $ARGV[0] ) {
    when( ! /^\d+$/ ) { say "Not a number!" }
```

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

```
    my @divisors = divisors( $ARGV[0] );

    when( @divisors ~~ 2 ) { # 2 is in @divisors
        say "$_ is even";
        continue;
        }

    when( !( @divisors ~~ 2 ) ) { # 2 isn't in @divisors
        say "$_ is odd";
        continue;
        }

    when( @divisors ~~ $favorite ) {
        say "$_ is divisible by my favorite number";
        continue;
        }

    when( $favorite ) { # $_ ~~ $favorite
        say "$_ is my favorite number";
        continue;
        }

    my @empty;
    when( @divisors ~~ @empty ) { say "Number is prime" }

    default { say "$_ is divisible by @divisors" }
    }

sub divisors {
    my $number = shift;

    my @divisors = ();
    foreach my $divisor ( 2 .. ($ARGV[0]/2 + 1) ) {
        push @divisors, $divisor unless $number % $divisor;
        }

    return @divisors;
    }
```

В этой расширенной версии предыдущего упражнения добавлены новые блоки для новых проверяемых условий. После получения массива `@divisors` мы используем оператор умного сравнения для проверки его содержимого. Если в число делителей входит 2, значит, число является четным. Программа выводит соответствующее сообщение, а команда `continue` обеспечивает проверку следующего условия `when`. Для нечетных чисел используется та же проверка с отрицанием результата. Чтобы узнать, присутствует ли среди делителей «любимое число», мы поступаем аналогичным образом.

Глава 16

1. Одно из возможных решений:

```
chdir "/" or die "Can't chdir to root directory: $!";
exec "ls", "-l" or die "Can't exec ls: $!";
```

Первая строка выбирает корневой каталог в качестве текущего рабочего каталога. Вторая строка использует функцию `exec` с несколькими аргументами для передачи результата в стандартный вывод. Также можно использовать форму с одним аргументом, в данном случае это ни на что не влияет.

2. Одно из возможных решений:

```
open STDOUT, ">ls.out" or die "Can't write to ls.out: $!";
open STDERR, ">ls.err" or die "Can't write to ls.err: $!";
chdir "/" or die "Can't chdir to root directory: $!";
exec "ls", "-l" or die "Can't exec ls: $!";
```

Первая и вторая строки открывают дескрипторы `STDOUT` и `STDERR` для файлов в текущем каталоге (перед сменой каталогов). Затем, после вызова `chdir`, выполняется команда чтения содержимого каталога, а данные сохраняются в файлах, открытых в исходном каталоге.

Где будет сохранено сообщение последней команды `die`? Конечно, в *ls.err* – в том файле, с которым связан дескриптор `STDERR`. Сообщение `die` от вызова `chdir` может быть сохранено там же. Но куда попадет сообщение, если попытка повторного открытия `STDERR` во второй строке завершится неудачей? В старый поток `STDERR`. Если повторное открытие любого из стандартных файловых дескрипторов – `STDIN`, `STDOUT` и `STDERR` – завершится неудачей, старый дескриптор все равно останется открытым.

3. Одно из возможных решений:

```
if (`date` =~ /^S/) {
  print "go play!\n";
} else {
  print "get to work!\n";
}
```

Так как названия обоих выходных дней (Saturday и Sunday) начинаются с буквы S, а выходные данные команды *date* начинаются с дня недели, задача решается очень просто: достаточно проверить результат *date* и посмотреть, начинается ли он с буквы S. У этой задачи есть множество других, более сложных решений; многие из них уже встречались нам на семинарах.

Вероятно, в реальной программе мы бы использовали шаблон `/^(Sat|Sun)/`. Он чуть менее эффективен, но это не так существенно; зато этот шаблон будет гораздо проще понять программисту, занимающемуся сопровождением вашей программы.

Глава 17

1. Одно из возможных решений:

```
my $filename = 'path/to/sample_text';
open FILE, $filename
  or die "Can't open '$filename': $!";
chomp(my @strings = <FILE>);
while (1) {
  print "Please enter a pattern: ";
  chomp(my $pattern = <STDIN>);
  last if $pattern =~ /^\s*$/;
  my @matches = eval {
    grep /$pattern/, @strings;
  };
  if ($@) {
    print "Error: $@";
  } else {
    my $count = @matches;
    print "There were $count matching strings:\n",
      map "$_\n", @matches;
  }
  print "\n";
}
```

Блок `eval` перехватывает все ошибки, происходящие при использовании регулярного выражения. Внутри блока функция `grep` отбирает из списка строки с совпадениями.

После завершения `eval` выводится либо сообщение об ошибке, либо строки с совпадениями. Обратите внимание на подготовку строк для вывода, своего рода «`chomp` наоборот»: функция `map` добавляет в каждую строку завершающий символ новой строки.

B

Темы, не вошедшие в книгу

В книге мы рассказали о многом, но не обо всем. В этом приложении мы рассмотрим, на что еще способен Perl, и приведем ссылки на некоторые источники, к которым следует обращаться за подробностями. Часть материала находится на «передовых рубежах» развития Perl, и к тому моменту, когда вы будете читать эту книгу, информация может измениться. Это одна из причин, по которым в тексте часто встречаются отсылки к документации. Вряд ли многие читатели прочитают это приложение «от корки до корки», но мы надеемся, что вы хотя бы просмотрите заголовки. И когда кто-нибудь скажет вам: «Perl не подойдет для проекта X, потому что он не умеет Y», вы будете готовы дать отпор.

Самое важное, что необходимо запомнить (чтобы нам не пришлось повторяться в каждом абзаце): основная часть материала, *не* представленного в этой книге, собрана в книге «Intermediate Perl»[1] (O'Reilly), также известной как «книга с альпакой». Вам определенно стоит прочитать «книгу с альпакой», особенно если вы пишете программы длиной более 100 строк (в одиночку или с другими программистами).

После «книги с альпакой» вы будете готовы перейти к книге «Mastering Perl», также известной как «книга с викуньей». В ней рассматриваются многие повседневные задачи программирования на Perl, в том числе хронометраж и профилирование, настройка конфигурации программ и ведение журнала. Также вы научитесь работать с кодом, написанным другими людьми, и интегрировать его в свои приложения.

Дополнительная документация

Документация, входящая в поставку Perl, на первый взгляд кажется необъятной. К счастью, компьютер поможет вам найти в ней нужные

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

ключевые слова. При поиске по конкретной теме часто бывает полезно начать с документа *perltoc* (содержание) и *perlfaq* (часто задаваемые вопросы). В большинстве систем команда *perldoc* ведет всю документацию Perl, включая установленные модули и вспомогательные программы (вместе с самой программой *perldoc*). Та же документация доступна в Интернете на сайте *http://perldoc.perl.org*, хотя этот сайт всегда относится к последней версии Perl.

Регулярные выражения

Да, мы описали не все возможности регулярных выражений. «Mastering Regular Expressions»[1] Джеффри Фридла (Jeffrey Friedl) (O'Reilly) – одна из лучших технических книг, которые нам доводилось читать.[2] Половина книги посвящена регулярным выражениям вообще, другая половина – регулярным выражениям Perl. В книге подробно рассматриваются правила внутренней работы ядра регулярных выражений, а также объясняется, почему один вариант записи шаблона может оказаться значительно эффективнее другого. Каждый программист, серьезно относящийся к Perl, должен прочитать эту книгу. Также см. man-страницу *perlre* (и сопроводительные страницы *perlretut* и *perlrequick* в новых версиях Perl). В «книге с альпакой» также содержится дополнительная информация о регулярных выражениях.

Пакеты

Пакеты[3] позволяют создавать изолированные пространства имен. Представьте, что 10 программистов работают над одним большим проектом. Если они применяют глобальные имена `$fred`, `@barney`, `%betty` и `&wilma` в своих частях проекта, что произойдет, когда вы случайно используете одно из этих имен в своей части? Благодаря пакетам эти имена существуют независимо друг от друга; я могу обратиться к вашей переменной `$fred`, вы – к моей, но не случайно, а намеренно. Пакеты улучшают масштабируемость Perl и упрощают работу над большими проектами. Пакеты очень подробно описаны в «книге с альпакой».

---

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

[2] Вовсе не потому, что эта книга тоже была издана в O'Reilly. Просто это действительно очень хорошая книга.

[3] Возможно, термин «пакет» выбран не совсем удачно: многие программисты подразумевают под этим термином упакованный блок программного кода (в Perl это модуль или библиотека). Пакет всего лишь определяет пространство имен (совокупность глобальных символических имен вида `$fred` или `&wilma`). Пространство имен *не является* блоком программного кода.

Расширение функциональности Perl

Один из самых полезных советов в дискуссионных форумах Perl гласит: «Не надо изобретать велосипед». Другие люди уже написали код, который вы можете использовать в своей работе. Чаще всего готовый код используется в форме библиотек и модулей. Многие библиотеки и модули входят в поставку Perl, другие загружаются из архива CPAN. Конечно, ничто не мешает вам создавать собственные библиотеки и модули.

Библиотеки

Многие языки программирования, как и Perl, позволяют создавать библиотеки. Библиотека (чаще всего) представляет собой подборку пользовательских функций, предназначенных для определенной цели. В современном Perl модули используются чаще, чем библиотеки.

Модули

Модуль представляет собой «умную библиотеку». Как правило, модуль содержит коллекцию пользовательских функций, в целом работающих примерно так, как если бы они были встроенными функциями. Модули названы «умными», потому что информация о их внутреннем устройстве хранится в отдельном пакете, из которого имена импортируются только по запросу программиста. Это предотвращает конфликты между символическими именами модуля и вашей программы.

Многие полезные модули написаны на «чистом» Perl; другие пишутся на таких языках, как C. Например, алгоритм MD5 представляет собой алгоритм вычисления контрольной суммы с расширенными возможностями.[1] В нем используются низкоуровневые поразрядные операции, которые могут выполняться в Perl, но в сотни раз медленнее[2]; этот алгоритм проектировался в расчете на эффективную реализацию C. В модуле `Digest::MD5` используется откомпилированный код C. Когда вы используете этот модуль, все выглядит так, словно в Perl появилась встроенная функция для вычисления дайджестов MD5.

---

[1] На самом деле это не контрольная сумма, но для этого объяснения сойдет.

[2] Модуль `Digest::Perl::MD5` содержит реализацию алгоритма MD5 на «чистом» Perl. Мы обнаружили, что на одном наборе данных он работает примерно в 280 раз медленнее, чем модуль `Digest::MD5` (конечно, на вашем компьютере конкретные цифры могут быть другими). Не забудьте, что многие поразрядные операции в C компилируются в *одну* машинную команду; таким образом, целые блоки кода выполняются за минимальное количество тактов. Perl работает быстро, но нужно быть реалистом.

Поиск и установка модулей

Возможно, нужный вам модуль уже установлен в системе. Но как узнать, какие модули в ней установлены? Используйте программу *inside*, которую можно загрузить из архива CPAN по адресу *http://www.cpan.org/authors/id/P/PH/PHOENIX/*.

Если ни один из модулей, установленных в системе, не подходит для ваших целей, поищите модули Perl в CPAN по адресу *http://search.cpan.org/*. За информацией об установке модулей в системе обращайтесь к man-странице *perlmodinstall*.

При использовании модуля в начало программы обычно помещаются необходимые директивы `use`. Программист, который устанавливает вашу программу в новой системе, сможет сразу увидеть, какие модули ей нужны.

Написание собственных модулей

В тех редких случаях, когда готового модуля для конкретной задачи не существует, опытный программист может написать новый модуль на Perl или другом языке (обычно на C). За дополнительной информацией обращайтесь к man-страницам *perlmod* и *perlmodlib*. О том, как писать, тестировать и распространять модули, рассказано в «книге с альпакой».

Основные модули

В этом разделе описаны важнейшие возможности[1] самых главных модулей.[2] Модули, о которых пойдет речь, обычно имеются на каждом компьютере с Perl (если только в тексте явно не указано обратное). Последние версии этих модулей всегда можно загрузить из архива CPAN.

Модули CGI

Многие программисты используют Perl для написания программ, выполняемых веб-сервером (так называемые *программы CGI*). Модуль `CGI` входит в поставку Perl. Простой пример был приведен в главе 11; другие примеры также рассматриваются ниже в этом приложении.

---

[1] В тексте упоминаются только самые важные возможности каждого модуля; за дополнительной информацией обращайтесь к документации каждого модуля.

[2] Разумеется, существует много других важных модулей, использование которых выходит за рамки книги, поскольку требует понимания ссылок и объектов Perl.

Модуль Cwd

В некоторых ситуациях требуется определить имя текущего каталога. (Конечно, часто можно воспользоваться элементом «.», но возможно, вы хотите сохранить имя для последующего возвращения к этому каталогу.) Модуль `Cwd`, входящий в поставку Perl, содержит функцию `cwd` для определения текущего рабочего каталога:

```
use Cwd;

my $directory = cwd;
```

Модуль Fatal

Если вам надоело писать `"or die"` после каждого вызова `open` или `chdir`, возможно, модуль `Fatal` вам пригодится. Просто сообщите ему, с какими функциями он должен работать, и эти функции будут автоматически проверяться на предмет сбоев, как если бы после каждого вызова следовала запись `"or die"` с соответствующим сообщением. Действие `Fatal` не распространяется на вызовы в других пакетах (например, в коде используемых модулей), поэтому `Fatal` не следует использовать для исправления небрежно написанного кода. Это всего лишь средство экономии времени, прежде всего для простых программ, в которых вам не нужно полностью контролировать сообщение об ошибке. Пример:

```
use Fatal qw/ open chdir /;

chdir '/home/merlyn';  # "or die" подставляется автоматически
```

Модуль File::Basename

Этот модуль упоминался в главе 11. Он обеспечивает портируемое извлечение базового имени файла или имени каталога из полного имени файла:

```
use File::Basename;

for (@ARGV) {
  my $basename = basename $_;
  my $dirname = dirname $_;
  print "That's file $basename in directory $dirname.\n";
}
```

Модуль File::Copy

Если вам потребовалось скопировать или переместить файлы, модуль `File::Copy` для вас. (Часто возникает искушение просто вызвать системную программу, но такое решение непортируемо.) Модуль предоставляет функции перемещения и копирования, которые используются практически так же, как аналогичные системные программы:

```
use File::Copy;

copy("source", "destination")
  or die "Can't copy 'source' to 'destination': $!";
```

Модуль File::Spec

Если вам потребуется выполнить операции с именами файлов (а вернее, с «файловыми спецификациями»), решения, предоставляемые модулем `File::Spec`, обычно обладают лучшей портируемостью и надежностью, чем самостоятельные реализации. Например, функция `catfile` строит полное имя файла, объединяя имя каталога с именем файла (см. главу 11), но при этом вам не нужно думать, какой разделитель используется в системе, в которой эта программа выполняется, – косая черта или другой символ. Функция `curdir` возвращает имя текущего каталога (`.` в системах UNIX).

Модуль `File::Spec` является объектно-ориентированным, но чтобы пользоваться им, необязательно разбираться в объектах. Просто вызывайте нужные функции (а на самом деле «методы») с префиксом из имени модуля `File::Spec` и маленькой стрелки `->`:

```
use File::Spec;

my $current_directory = File::Spec->curdir;
opendir DOT, $current_directory
  or die "Can't open current directory '$current_directory': $!";
```

Модуль Image::Size

При работе с файлом графического изображения часто требуется узнать его ширину и высоту. (Например, в программах, генерирующих код HTML, при создании тега `IMG` указываются размеры изображения.) Модуль `Image::Size`, доступный в CPAN, поддерживает основные форматы графики GIF, JFIF (JPEG) и PNG, а также ряд других форматов. Пример:

```
use Image::Size;

# Получение размера fred.png
my($fred_height, $fred_width) = imgsize("fred.png");
die "Couldn't get the size of the image"
  unless defined $fred_height;
```

Модуль Net::SMTP

Если вы хотите, чтобы ваша программа могла отправлять электронную почту с сервера SMTP (а большая часть электронной почты в наши дни отправляется именно так), воспользуйтесь модулем `Net::SMTP`.[1]

[1] Да, это означает, что Perl можно использовать для рассылки спама. Пожалуйста, не делайте этого.

Этот модуль, также доступный из архива CPAN, является объектно-ориентированным, но вы можете просто использовать его, соблюдая синтаксис вызова. Измените имя хоста SMTP и другие параметры конфигурации, чтобы сценарий работал в вашей системе. О том, какие значения следует использовать, можно узнать у системного администратора или местного специалиста. Пример:

```
use Net::SMTP;

my $from = 'YOUR_ADDRESS_GOES_HERE';          # Например, fred@bedrock.edu
my $site = 'YOUR_SITE_NAME_GOES_HERE';        # Например, bedrock.edu
my $smtp_host = 'YOUR_SMTP_HOST_GOES_HERE';   # Например, mail или mailhost
my $to = 'president@whitehouse.gov';

my $smtp = Net::SMTP->new($smtp_host, Hello => $site);

$smtp->mail($from);
$smtp->to($to);
$smtp->data( );

$smtp->datasend("To: $to\n");
$smtp->datasend("Subject: A message from my Perl program.\n");
$smtp->datasend("\n");
$smtp->datasend("This is just to let you know,\n");
$smtp->datasend("I don't care what those other people say about you,\n");
$smtp->datasend("I still think you're doing a great job.\n");
$smtp->datasend("\n");
$smtp->datasend("Have you considered enacting a law naming Perl \n");
$smtp->datasend("the national programming language?\n");

$smtp->dataend( );                            # Не datasend!
$smtp->quit;
```

Модуль POSIX

Для тех, кому потребуется доступ к функциям POSIX (IEEE Std 1003.1), был создан модуль `POSIX`. Он объединяет множество функций, хорошо знакомых программистам C: тригонометрических (`asin`, `cosh`), общих математических (`floor`, `frexp`), идентифицирующих (`isupper`, `isalpha`), низкоуровневых функций ввода/вывода (`creat`, `open`) и других (`asctime`, `clock`). Вероятно, эти функции стоит вызвать по «полному» имени, то есть с префиксом из имени `POSIX` и двоеточия:

```
use POSIX;

print "Please enter a number: ";
chomp(my $str = <STDIN>);

$! = 0;  # Сброс признака ошибки
my($num, $leftover) = POSIX::strtod($str);

if ($str eq '') {
  print "That string was empty!\n";
} elsif ($leftover) {
```

```
    my $remainder = substr $str, -$leftover;
    print "The string '$remainder' was left after the number $num.\n";
  } elsif ($!) {
    print "The conversion function complained: $!\n";
  } else {
    print "The seemingly-valid number was $num.\n";
  }
```

Модуль Sys::Hostname

Модуль `Sys::Hostname` предоставляет функцию `hostname`, которая возвращает сетевое имя вашего компьютера, если его удается определить. (Если имя не определяется, например при отсутствии подключения к Интернету или неправильных настройках компьютера, автоматически происходит аварийное завершение; использовать `or die` здесь бессмысленно.) Пример:

```
use Sys::Hostname;
my $host = hostname;
print "This machine is known as '$host'.\n";
```

Модуль Text::Wrap

Модуль `Text::Wrap` предоставляет функцию `wrap`, реализующую простой перенос текста по словам. Первые два параметра определяют отступ первой строки и остальных строк соответственно; остальные параметры образуют текст абзаца:

```
use Text::Wrap;

my $message = "This is some sample text which may be longer " .
  "than the width of your output device, so it needs to " .
  "be wrapped to fit properly as a paragraph. ";
$message x= 5;
print wrap("\t", "", "$message\n");
```

Модуль Time::Local

Преобразование времени (например, полученного функцией `time`) в список, содержащий значения года, месяца, дня, часов, минут и секунд, можно выполнить встроенной функцией Perl `localtime` в списочном контексте[1] (в скалярном контексте функция выдает отформатированную строку с временем, которая часто оказывается удобнее списка). Но если преобразование потребуется выполнить в обратном направлении, вы можете воспользоваться функцией `timelocal` из модуля `Time::Local`. Будьте внимательны: для марта 2008 года значения `$mon` и `$year` равны вовсе не 3 и 2008, как можно было бы предположить;

[1] Реальное возвращаемое значение `localtime` в списочном контексте несколько отличается от ожидаемого; см. документацию.

обязательно прочитайте документацию перед тем, как использовать модуль. Пример:

```
use Time::Local;
my $time = timelocal($sec, $min, $hr, $day, $mon, $year);
```

Прагмы

Прагмы (pragmas) представляют собой специальные модули, которые включаются в каждую версию Perl и передают внутреннему компилятору Perl информацию о коде. Мы уже использовали прагму `strict`. Прагмы, доступные для вашей версии Perl, перечислены в man-странице *perlmodlib*.

Прагмы используются почти так же, как обычные модули, – директивой `use`. Некоторые прагмы обладают лексической областью видимости по аналогии с лексическими (`my`) переменными, а следовательно, действуют в границах наименьшего окружающего блока или файла. Другие применяются ко всей программе или к текущему пакету. (Если пакеты не используются, прагмы применяются ко всей программе.) В общем случае прагмы рекомендуется перечислять в начале исходного кода. Область видимости каждой прагмы должна быть указана в документации.

Прагма constant

Если вы работали на других языках программирования, вероятно, вам уже встречались возможности объявления констант в той или иной форме. Константы удобны тем, что их значения присваиваются всего один раз в начале программы и при необходимости могут легко обновляться. В Perl эта задача решается прагмой `constant` с пакетной областью видимости. Эта прагма сообщает компилятору, что идентификатор имеет неизменное значение, а следовательно, может оптимизироваться в местах использования. Пример:

```
use constant DEBUGGING => 0;
use constant ONE_YEAR => 365.2425 * 24 * 60 * 60;

if (DEBUGGING) {
  # Этот код исключается в результате оптимизации,
  # если режим DEBUGGING не будет включен
  ...
}
```

Прагма diagnostics

Диагностические сообщения Perl часто выглядят загадочно, особенно когда вы встречаете их впервые. Но вы всегда можете узнать, что они означают, в man-страницах *perldiag*; нередко здесь приводятся возможные причины возникшей проблемы и способы ее решения. Однако

вы можете избавиться от хлопот с поисками man-страницы, воспользовавшись прагмой diagnostics, которая приказывает Perl найти и вывести дополнительную информацию по каждому сообщению. В отличие от большинства прагм, эта не предназначена для повседневного использования, так как программе приходится при запуске загружать всю страницу *perldiag* (что сопряжено со значительными затратами времени и памяти). Используйте прагму diagnostics только во время отладки, когда вы ожидаете получить сообщение с неясным смыслом. Действие прагмы распространяется на всю программу. Синтаксис:

```
use diagnostics;
```

Прагма lib

Модули всегда лучше устанавливать в стандартные каталоги, чтобы они были доступны для всех пользователей, но это может сделать только системный администратор. Если вы устанавливаете собственные модули, вам придется сохранить их своих каталогах – как тогда Perl узнает, где их искать? Для этого и нужна прагма lib. Она сообщает Perl, что поиск модулей следует начинать с указанного каталога. (Отсюда следует, что прагма будет удобна также во время тестирования пробной версии модуля.) Действие прагмы распространяется на все модули, загружаемые с этого момента. Синтаксис:

```
use lib '/home/rootbeer/experimental';
```

Будьте внимательны: не передавайте в аргументе относительное имя. Неизвестно, какой каталог будет использоваться в качестве рабочего во время выполнения программы. Это особенно важно для программ CGI (то есть программ, выполняемых веб-сервером).

Прагма strict

Вы уже довольно давно используете директиву use strict, даже не сознавая, что это прагма. Она обладает лексической областью видимости и определяет более жесткие правила «хорошего стиля» программирования. О том, какие ограничения use strict устанавливает для вашей версии Perl, можно узнать в документации. В «книге с альпакой» рассматриваются другие задачи, решаемые модулем strict.

Прагма vars

В тех редких случаях, когда вам действительно потребуется глобальная переменная при действующей директиве use strict, объявите переменную прагмой vars.[1] Эта прагма с пакетной областью видимости

[1] Если ваша программа не будет использоваться с версиями Perl до 5.6, примените ключевое слово our вместо прагмы vars.

сообщает Perl, что вы намеренно используете одну или несколько глобальных переменных:

```
use strict;
use vars qw/ $fred $barney /;

$fred = "This is a global variable, but that's all right.\n";
```

Прагма `vars` более подробно рассматривается в «книге с альпакой».

Прагма warnings

В Perl версии 5.6 появилась возможность включения предупреждений с лексической областью видимости при помощи прагмы `warnings`.[1] Иначе говоря, вместо того чтобы включать или отключать предупреждения сразу для всей программы ключом `-w`, можно указать, например, что в одной секции кода не нужны предупреждения о неопределенных значениях, тогда как все остальные предупреждения должны выводиться. Прагма также говорит программисту, который будет заниматься сопровождением: «Я знаю, что этот код выдает предупреждения, но так надо». Категории предупреждений, доступные в вашей версии Perl, перечислены в документации прагмы `warnings`.

Базы данных

Perl также поддерживает работу с базами данных. В этом разделе описаны наиболее распространенные типы баз данных. Модуль `DBI` кратко упоминался ранее в главе 11.

Прямой доступ к базам данных

Perl напрямую работает с некоторыми системными базами данных, иногда с помощью модуля. К их числу относятся системный реестр Windows (содержащий конфигурационные данные уровня компьютера), база данных паролей UNIX и база данных доменных имен (для преобразования IP-адресов в имена компьютеров, и наоборот).

Доступ к базам данных в плоских файлах

Для работы с базами данных, оформленными в виде плоских (неструктурированных) файлов, в Perl существуют специальные модули, причем новые модули появляются каждые месяц-два, поэтому любой список будет заранее устаревшим.

---

[1] Если программа может использоваться с версией Perl до 5.6, не применяйте прагму `warnings`.

Другие операторы и функции

Да, существуют и другие операторы и функции, которые можно отнести к этой категории: от скалярного оператора «. . » до скалярного оператора «, », от `wantarray` до `goto` (!), от `caller` до `chr`. Обращайтесь к man-страницам *perlop* и *perlfunc*.

Оператор tr///

Оператор `tr///` напоминает регулярное выражение, но в действительности он осуществляет преобразования между группами символов. Кроме того, он может эффективно использоваться для подсчета некоторых символов. Обращайтесь к man-странице *perlop*.

Внутренние документы

Внутренние документы (here documents) – полезная форма многострочного оформления строковых данных; см. man-страницу *perldata*.

Математика

Perl способен выполнить практически любые математические вычисления, которые только можно представить.

Расширенные математические функции

Все базовые математические функции (квадратный корень, косинус, натуральный логарифм, абсолютная величина и многие другие) доступны в виде встроенных функций; за подробностями обращайтесь к man-странице *perlfunc*. Некоторые функции (например, тангенс или десятичный логарифм) отсутствуют, но они легко конструируются на базе существующих функций или загружаются из простых модулей, в которых эти функции реализованы. (Многие стандартные математические функции собраны в модуле `POSIX`.)

Комплексные числа

Несмотря на отсутствие прямой поддержки комплексных чисел в Perl, существуют специальные модули для работы с ними. Они перегружают обычные операторы и функции так, чтобы вы могли умножать оператором `*` и извлекать квадратный корень функцией `sqrt` даже при использовании комплексных чисел. См. описание модуля `Math::Complex`.

Большие числа и числа с повышенной точностью

Математические вычисления можно выполнять со сколь угодно большими числами и с произвольной точностью. Например, можно вычислить факториал 2000 или вычислить π с точностью до 10 000 цифр. См. описания модулей `Math::BigInt` и `Math::BigFloat`.

Списки и массивы

Perl содержит разнообразные средства, упрощающие работу со списками и массивами.

map и grep

Операторы обработки списков `map` и `grep` уже упоминались ранее (см. главу 17). Однако в книге описаны не все их возможности; за дополнительной информацией и примерами обращайтесь к man-странице *perlfunc*. Другие возможности использования `map` и `grep` также описаны в «книге с альпакой».

Оператор splice

Оператор `splice` добавляет элементы в середину массива или удаляет их с соответствующим увеличением или сокращением массива. (В общих чертах он позволяет делать со списками то же, что `substr` делает со строками.) Фактически это отменяет необходимость в связанных списках в Perl. См. man-страницу *perlfunc*.

Битовые строки

Оператор `vec` работает с массивами битов (*битовыми строками*): например, вы можете установить бит с номером 123, сбросить бит с номером 456 и проверить состояние бита 789. Битовые строки могут иметь произвольную длину. Оператор `vec` также может работать не только с одиночными битами, но и с битовыми блоками других размеров, равных малой степени 2. См. man-страницу *perlfunc*.

Форматы

Форматы Perl чрезвычайно удобны для построения шаблонных отчетов фиксированного формата с автоматическим построением заголовков страниц. Собственно, они были одной из основных причин, по которым Ларри создавал Perl как «Практический язык выборки данных и построения *отчетов*». Но, к сожалению, их возможности не безграничны. Самое жестокое разочарование поджидает программиста, когда выясняется, что ему нужно чуть больше того, на что способны форматы. Обычно ему приходится выбрасывать всю часть вывода и заменять ее кодом, не использующим форматы. Впрочем, если вы твердо уверены, что форматы – то, что вам нужно, *все*, что вам нужно, и *ничего больше* совершенно точно не понадобится, они весьма неплохи. См. man-страницу *perlform*.

Сети и IPC

Если в вашей системе предусмотрены механизмы, посредством которых программы могут «общаться» друг с другом, Perl, вероятно, справится и с этой задачей. В этом разделе представлены самые распространенные механизмы межпроцессных взаимодействий.

System V IPC

Perl поддерживает все стандартные функции System V IPC (Interprocess Communication); вы можете использовать очереди сообщений, семафоры и общую память. Конечно, массив в Perl не хранится в одном блоке памяти, как массивы C[1], поэтому общая память не может использоваться для обмена данными Perl в исходном виде. Однако существуют модули, обеспечивающие преобразование данных, которые позволяют имитировать хранение данных Perl в общей памяти. См. man-страницы *perlfunc* и описание модуля *perlipc*.

Сокеты

Perl обладает полноценной поддержкой сокетов TCP/IP. Это означает, что на Perl можно написать веб-сервер или броузер, сервер или клиент новостей Usenet, демон или клиент *finger*, демон или клиент FTP, сервер или клиент SMTP/POP/SOAP, а также другие разновидности серверов и клиентов для практически любого протокола, используемого в Интернете. Конечно, вам не придется заниматься всеми низкоуровневыми подробностями; для всех стандартных протоколов существуют готовые модули. Например, вы можете создать веб-сервер или клиент на базе модуля `LWP` с одной-двумя строками дополнительного кода.[2] Модуль `LWP` (а по сути плотно интегрированный набор модулей, совместно реализующих практически любые взаимодействия в Web) также является отличным примером высококачественного кода Perl, если вы хотите копировать из лучших источников. Если вам потребуется поддержка других протоколов, поищите модуль, имя которого совпадает с именем протокола.

---

[1] В общем случае нельзя даже утверждать, что массив Perl «хранится в блоке памяти», потому что его содержимое почти всегда разделено на множество разных блоков.

[2] Хотя `LWP` позволяет легко создать простой «веб-броузер» для загрузки страницы или изображения, возникает другая проблема с отображением полученной информации. Для управления экраном X11 можно воспользоваться виджетами Tk или Gtk, а для вывода на символьный терминал применить curses. В конечном итоге все сводится к загрузке и установке нужных модулей из CPAN.

Безопасность

Perl содержит ряд мощных функций безопасности, благодаря которым программа, написанная на Perl, может оказаться более безопасной, чем аналогичная программа на C. Вероятно, самой важной из них является анализ потока данных, также называемый *taint-проверкой*. При включении этого режима Perl следит за тем, какие фрагменты данных поступили от пользователя или среды (а следовательно, являются небезопасными). Когда любой фрагмент таких «небезопасных» данных пытается повлиять на другой процесс, файл или каталог, Perl отменяет операцию и аварийно завершает программу. Taint-проверка не идеальна, но это мощный механизм предотвращения некоторых ошибок, связанных с безопасностью. За дополнительной информацией обращайтесь к man-странице *perlsec*.

Отладка

Вместе с Perl поставляется очень хороший отладчик с поддержкой точек прерывания, отслеживания значений, пошагового выполнения – и вообще всех функций, которые вам хотелось бы иметь в отладчике Perl командной строки. Сам отладчик тоже написан на Perl (интересно, и как искать ошибки в отладчике?). Но это означает, что наряду со всеми обычными командами отладчика в нем также можно выполнять код Perl (вызывать функции, изменять переменные и даже переопределять пользовательские функции) во время выполнения вашей программы. За последней информацией обращайтесь к man-странице *perldebug*. Отладчик подробно описан в «книге с альпакой».

Другая тактика отладки основана на использовании модуля `B::Lint`. Этот модуль предупреждает о потенциальных проблемах, которые упускает даже ключ `-w`.

CGI

В области веб-программирования Perl чаще всего используется для написания программ CGI (Common Gateway Interface). Эти программы выполняются веб-сервером для обработки результатов формы, выполнения поиска, генерирования динамического веб-контента или подсчета обращений к веб-странице.

Модуль `CGI`, входящий в поставку Perl, предоставляет простой механизм доступа к параметрам форм и генерирования кода HTML. Возможно, у вас возникнет искушение отказаться от использования модуля и просто скопировать один из фрагментов кода, которые якобы предоставляют доступ к параметрам форм, но почти все такие фрагменты

содержат ошибки.[1] При написании программ CGI необходимо учитывать ряд важных обстоятельств, из-за которых эта тема становится слишком обширной для подробного изложения в книге.

Безопасность, безопасность, безопасность

Трудно переоценить, насколько важна безопасность. Около половины успешных атак на сетевые компьютеры обусловлено ошибками безопасности в программах CGI.

Параллелизм

Несколько процессов могут одновременно попытаться обратиться к одному файлу или ресурсу.

Совместимость

Как бы вы ни старались, скорее всего, вам удастся тщательно протестировать свою программу только для 1 или 2% броузеров и серверов, используемых в наши дни.[2] Существуют буквально тысячи всевозможных сетевых программ, причем каждую неделю появляются новые. Следуйте стандартам, и ваш продукт будет работать со всеми этими программами.[3]

Диагностика и отладка

Программы CGI работают в среде, напрямую для вас недоступной, поэтому вам придется освоить новые приемы диагностики и отладки.

Безопасность, безопасность, безопасность

Да, мы снова об этом. Постоянно помните о безопасности – это первое и последнее, о чем следует помнить при написании программы, доступной для всех, кто захочет ее взломать.

А ведь в списке даже не упоминаются кодирование URI, HTML-представления (HTMS entities), HTTP и коды ответов, SSL (Secure Sockets Layer), SSI (Server-Side Includes), встроенные документы, создание графики «на ходу», программное построение таблиц HTML, форм и виджетов, скрытые элементы форм, чтение и запись cookies, информация путей, перехват ошибок, перенаправление, taint-проверка, интернационализация и локализация, встраивание Perl в HTML (или на-

---

[1] Некоторые рецензенты, просматривавшие черновики книги, посчитали, что тема программирования CGI заслуживает более подробного изложения. Мы согласны, но было бы несправедливо давать читателю абсолютный минимум информации. А полноценное изложение вопросов программирования CGI повысило бы объем (и стоимость) книги, как минимум, на 50%.

[2] Вспомните, что новая версия каждого основного броузера для каждой платформы считается отдельно. Мы только усмехаемся, когда слышим, что кто-то «протестировал сайт в обоих броузерах» или «не уверен, как сайт будет работать в другом броузере».

[3] Или, по крайней мере, вам удастся переложить вину на другого программиста, который этого не сделал.

оборот), работа с Apache и `mod_perl` и использование модуля `LWP`.[1] Все эти темы изложены в любой хорошей книге по веб-программированию на Perl. Можем порекомендовать «CGI Programming with Perl»[2] Скотта Гулича (Scott Guelich) и др. (O'Reilly), а также «Network Programming with Perl» Линкольна Стейна (Lincoln Stein) (Addison-Wesley).

Параметры командной строки

Perl поддерживает множество разнообразных параметров командной строки; некоторые из них позволяют вводить программы прямо в командной строке. См. man-страницу *perlrun*.

Встроенные переменные

Десятки встроенных переменных Perl (таких как `@ARGV` and `$0`) предоставляют полезную информацию или управляют работой самого Perl. См. man-страницу *perlvar*.

Расширения синтаксиса

Синтаксис Perl предоставляет интересные возможности, не рассмотренные в книге, например блоки `continue` и блоки `BEGIN`. См. man-страницы *perlsyn* и *perlmod*.

Ссылки

Ссылки Perl схожи с указателями C, но по принципам своей работы они больше напоминают ссылки Pascal или Ada. Ссылка «указывает» на блок памяти; но так как в Perl не поддерживаются прямые математические операции со ссылками, а также прямое выделение и освобождение памяти, вы можете быть уверены в том, что любая имеющаяся ссылка является действительной. Среди прочего ссылки позволяют использовать объектно-ориентированные конструкции и сложные структуры данных. См. man-страницы *perlreftut* и *perlref*. Ссылки очень подробно рассматриваются в «книге с альпакой».

Сложные структуры данных

Ссылки используются для построения сложных структур данных в Perl. Вы можете создавать двумерные массивы[3] или более интересные

---

[1] Теперь понимаете, почему мы не пытались вместить весь этот материал в книгу?

[2] Скотт Гулич и др. «CGI-программирование на Perl», 2-е издание. – Пер. с англ. – СПб.: Символ-Плюс, 2001 (в продаже только электронная версия).

[3] Пусть не настоящие, но, по крайней мере, настолько качественную имитацию, что вы вряд ли заметите разницу.

структуры данных – массивы хешей, хеши хешей или хеши массивов хешей.[1] См. man-страницы *perldsc* и *perllol*. Эта тема, включая сложную обработку данных (сортировка, сводки и т. д.) также весьма подробно изложена в «книге с альпакой».

Объектно-ориентированное программирование

Да, в Perl существуют объекты, не уступающие по своим возможностям объектам в других языках программирования. Объектно-ориентированное программирование позволяет создавать пользовательские типы данных с такими возможностями, как наследование, переопределение и динамическое связывание методов.[2] Но в отличие от некоторых объектно-ориентированных языков, Perl не заставляет вас использовать объекты. (Даже многие объектно-ориентированные модули можно использовать, не разбираясь в объектах.) Но если объем вашей программы превышает N строк кода, возможно, для повышения эффективности работы программиста стоит сделать ее объектно-ориентированной (даже при том, что она станет работать чуть медленнее). Никто не знает точного значения N; по нашим прикидкам, оно составляет несколько тысяч... или около того. Начните с man-страниц *perlobj* и *perlboot*, а за дополнительной информацией обращайтесь к превосходной книге Дэмиана Конуэя (Damian Conway) «Object-Oriented Perl» (издательство Manning Press). Объекты также подробно рассматриваются в «книге с альпакой».

Анонимные функции и замыкания

Как бы странно это ни выглядело на первый взгляд, функция без имени тоже может быть полезной. Такие функции передаются в параметрах другим функциям или используются для реализации таблиц переходов посредством вызова через массивы и хеши. Мощная концепция *замыканий* (*closures*) была позаимствована Perl из мира Lisp. Замыкание (упрощенно говоря) представляет собой анонимную функцию с собственными приватными данными. И снова дополнительную информацию следует искать в «книге с альпакой».

Связанные переменные

Связанные (tied) переменные используются в программе точно так же, как и все остальные, но при этом автоматически выполняется заданный вами код. Таким образом, вы можете создать скалярное значение,

---

[1] На самом деле такие структуры создать невозможно. То, что мы называем «массивом массивов», в Perl реализуется как массив *ссылок* на массивы.

[2] В объектно-ориентированном программировании существует свой жаргон. Более того, некоторые термины имеют разное значение в разных объектно-ориентированных языках.

которое в действительности хранится на удаленном компьютере, или массив, который всегда хранится в отсортированном порядке. См. man-страницу *perltie*.

Перегрузка операторов

Модуль `overload` дает возможность переопределять операторы сложения, конкатенации, сравнения и даже неявного преобразования строк в числа. В частности, именно таким образом модуль реализации комплексных чисел (к примеру) позволяет умножить комплексное число на 8 и получить комплексный результат.

Динамическая загрузка

Динамическая загрузка основана на простом принципе: во время выполнения ваша программа решает, что ей необходима функциональность, не доступная в настоящий момент. Программа загружает необходимые блоки и продолжает работать. Динамическая загрузка кода Perl возможна всегда, гораздо интереснее динамически загрузить двоичное расширение.[1] Этот механизм используется при создании модулей, написанных не на Perl.

Встраивание

Встраивание (embedding) в определенном смысле является противоположностью динамической загрузки.

Допустим, вы создаете крутой текстовый редактор и начали писать его, например на C++.[2] В какой-то момент вы решаете, что пользователям будет полезно предоставить доступ к регулярным выражениям Perl как к сверхмощному средству поиска и замены, и внедряете в свою программу Perl. Потом становится ясно, что некоторые возможности Perl было бы полезно предоставить в распоряжение пользователей. Скажем, опытный пользователь пишет на Perl фрагмент кода, который преобразуется в команду меню вашей программы. Чтобы изменить какие-то особенности работы вашего редактора, пользователь пишет

---

[1] В общем случае динамическая загрузка двоичных расширений возможна только тогда, когда механизм динамической загрузки поддерживается вашей системой. В противном случае применяется статическая компиляция расширений – иначе говоря, вы создаете двоичный файл Perl со встроенным расширением.

[2] Наверное, именно этот язык мы бы выбрали для подобного проекта. Да, мы любим Perl, однако мы не подписывались кровью под отречением от других языков. Если язык X лучше всего подходит для вашей задачи, используйте язык X. Но довольно часто оказывается, что X = Perl.

небольшой фрагмент на Perl. Далее на вашем сайте открывается уголок, в котором пользователи обмениваются этими фрагментами. В результате тысячи новых программистов расширяют возможности вашей программы без малейших затрат для вашей компании. И сколько вам придется заплатить за это Ларри? Ничего – загляните в тексты лицензий из поставки Perl. Вот такой хороший человек Ларри. Хотя бы скажите ему «спасибо».

Нам такой редактор пока неизвестен, но некоторые люди уже начали применять эту методологию для построения других мощных программ. Один из примеров такого рода – модуль Apache `mod_perl`, встраивающий Perl в мощный веб-сервер. Если вы собираетесь использовать внедрение Perl, обязательно ознакомьтесь с `mod_perl`; проект распространяется с открытым кодом, и вы можете просто посмотреть, как это делается.

Перевод кода с других языков на Perl

У вас имеются старые программы *sed* и *awk*, которые вам хотелось бы переписать на Perl? Считайте, вам повезло. Perl может сделать все, на что способны эти языки. Более того, существует программа автоматического перевода, которая, вероятно, установлена в вашей системе. Поищите в документации описания *s2p* (перевод с *sed*) и *a2p* (перевод с *awk*[1]). Код, сгенерированный программой, все же уступает по качеству коду, написанному человеком, поэтому результат не всегда будет оптимальным, но, по крайней мере, вы получите хорошую отправную точку для дальнейшей настройки. Также невозможно заранее предсказать, будет ли переведенная программа работать быстрее или медленнее оригинала. Но после исправления основных неэффективных мест в сгенерированном коде Perl программа должна, по крайней мере, работать с сопоставимыми характеристиками.

Хотите перевести на Perl старые алгоритмы C? Что ж, вам отчасти повезло: код C нетрудно преобразовать в откомпилированный модуль, который может использоваться из Perl. Более того, модули могут строиться практически на любом языке, компилируемом в объектный код. Обращайтесь к man-странице *perlxs*, документации модуля `Inline` и системе SWIG.

Хотите перевести на Perl сценарий командного процессора? А вот здесь ваше везение закончилось. Средств автоматического перевода сценариев на Perl не существует. Дело в том, что командный процессор сам по себе почти ничего не делает; в основном его работа сводится к запуску других программ. Конечно, мы могли бы написать програм-

[1] Если вы используете *gawk*, *nawk* или другой вариант, программа *a2p* не сможет автоматически перевести код на Perl. Обе программы были написаны давно и обновлялись только для поддержания совместимости с новыми версиями Perl.

му, которая вызывает `system` для каждой строки исходного сценария, но такая программа будет работать намного медленнее исходного сценария. Только человеческий разум способен представить, как использование *cut*, *rm*, *sed*, *awk* и *grep* преобразуется в эффективный код Perl. Сценарий командного процессора лучше переписать с начала.

Перевод командных строк find на Perl

Одна из стандартных задач системного администратора – рекурсивный поиск некоторых элементов в дереве каталогов. В UNIX эта задача обычно решается при помощи команды *find*. Впрочем, эта задача также может быть решена непосредственно на уровне Perl.

Команда *find2perl* из поставки Perl получает те же аргументы, что и команда *find*. Но вместо того чтобы искать указанные элементы, *find2perl* выдает программу Perl для их поиска. Так как результат является программой, вы можете отредактировать ее для собственных потребностей. (Надо признать, программа написана в довольно странном стиле.)

Программа *find2perl* также поддерживает полезный аргумент `-eval`, не поддерживаемый стандартной программой *find*. Этот аргумент указывает, что следующий за ним фрагмент кода Perl должен выполняться для каждого найденного файла. При выполнении этого кода текущим каталогом является каталог, в котором находится искомый элемент, а переменная `$_` содержит имя элемента.

Рассмотрим пример использования *find2perl*. Допустим, вы являетесь системным администратором UNIX и хотите найти и удалить все старые файлы в каталоге */tmp*.[1] Следующая команда генерирует программу для решения этой задачи:

```
$ find2perl /tmp -atime +14 -eval unlink >Perl-program
```

Эта команда ищет в */tmp* (с рекурсивным поиском во всех подкаталогах) элементы, у которых с момента последнего обращения прошло более 14 дней. Для каждого найденного элемента программа должна выполнять команду Perl `unlink`, по умолчанию использующую содержимое `$_` в качестве имени удаляемого файла. Результат выполнения *find2perl* (перенаправляемый в файл *Perl-program*) представляет собой программу, которая делает все вышеуказанное. Вам остается лишь организовать ее запуск по мере надобности.

[1] Обычно эта операция выполняется заданием *cron* каждую ночь.

Параметры командной строки в ваших программах

Если вы хотите, чтобы ваши программы обрабатывали параметры командной строки (по аналогии с параметрами командной строки самого Perl, например `-w`), воспользуйтесь модулями, которые позволят вам организовать их стандартную обработку. См. документацию модулей `Getopt::Long` и `GetOpt::Std`.

Встроенная документация

Собственная документация Perl записывается в формате *pod* (Plain-Old Documentation). Она встраивается в программы для последующего преобразования в текст, HTML или многие другие форматы по мере надобности. См. man-страницу *perlpod*. Встроенная документация также рассматривается в «книге с альпакой».

Другие способы открытия файловых дескрипторов

В Perl также поддерживаются дополнительные режимы открытия файловых дескрипторов; см. man-страницу *perlopentut*.

Локальные контексты и Юникод

Что ни говори, а мир тесен. Чтобы ваша программа корректно работала даже в тех странах, где используются другие алфавиты, в Perl предусмотрена поддержка локальных контекстов (locales) и Юникода.

Локальные контексты сообщают Perl, как те или иные операции выполняются по местным стандартам. Например, где должен располагаться символ æ при сортировке – в конце алфавита или между Š и Œ? И как называется третий месяц года? См. man-страницу *perllocale* (не путайте с *perllocal*).

За последней информацией о том, как ваша версия Perl поддерживает Юникод, обращайтесь к man-странице *perlunicode*. На момент написания книги во всех последних версиях Perl встречались изменения, связанные с поддержкой Юникода, но мы надеемся, что ситуация скоро нормализуется.

Программные потоки и ветвление

Perl теперь поддерживает программные потоки (threads). Хотя данная возможность считается экспериментальной (на момент написания книги), в некоторых приложениях потоки могут быть весьма полезными.

Функция `fork` (там, где она доступна) обладает более качественной поддержкой; см. man-страницы *perlfork* и *perlthrtut*.

Графические интерфейсы (GUI)

Большой полнофункциональный набор модулей `Tk` позволяет создавать графические интерфейсы, работающие на разных платформах. См. «Mastering Perl/Tk» Нэнси Уолш (Nancy Walsh) и Стива Лиди (Steve Lidie) (O'Reilly).

И последнее...

Обратившись к списку модулей CPAN, вы найдете в нем модули для выполнения самых разнообразных задач: от построения графиков и других изображений до приема электронной почты, от расчета погашения кредита до вычисления времени восхода солнца. В CPAN постоянно добавляются новые модули, поэтому сегодня Perl обладает еще большими возможностями, чем в момент написания книги. Нам за ним все равно не угнаться, поэтому мы на этом останавливаемся.

Даже Ларри говорит, что он уже не успевает следить за развитием Perl – необъятная Вселенная Perl продолжает расширяться. А поскольку он всегда может отыскать новый уголок в этой вечно расширяющейся Вселенной, Perl ему никогда не надоест – и нам, вероятно, тоже. Спасибо, Ларри!

Алфавитный указатель

Специальные символы

A

B

C

D

E

F

G

H

S

T

U

V

W

Е

З

И

К

Л

M

H

O

П

Р

C

Ц

Ч

Ш

Э

Ю

Я