От основ к мастерству

Предисловие Дэмиана Конвея

Изучаем глубже Perl

Рэндал Л. Шварц,
Брайан Д. Фой и Том Феникс

По договору между издательством «Символ-Плюс» и Интернет-магазином «Books.Ru – Книги России» единственный легальный способ получения данного файла с книгой ISBN 5-93286-093-9, название «Perl: изучаем глубже» – покупка в Интернет-магазине «Books.Ru – Книги России». Если Вы получили данный файл каким-либо другим образом, Вы нарушили международное законодательство и законодательство Российской Федерации об охране авторского права. Вам необходимо удалить данный файл, а также сообщить издательству «Символ-Плюс» (piracy@symbol.ru), где именно Вы получили данный файл.

Intermediate Perl

Second Edition

Randal L. Schwartz, brian d foy
& Tom Phoenix

O'REILLY®

Perl
изучаем глубже

Второе издание

Рэндал Л. Шварц, Брайан Д. Фой
и Том Феникс

Санкт-Петербург — Москва
2008

Рэндал Л. Шварц, Брайан Д. Фой и Том Феникс

Perl: изучаем глубже, 2-е издание

Перевод А. Киселева

Главный редактор	*А. Галунов*
Зав. редакцией	*Н. Макарова*
Научный редактор	*О. Цилюрик*
Редактор	*В. Овчинников*
Корректура	*О. Макарова*
Верстка	*Д. Орлова*

Шварц Р., Фой Б., Феникс Т.
Perl: изучаем глубже, 2-е издание. – Пер. с англ. – СПб: Символ-Плюс, 2007. – 320 с., ил.
ISBN-13: 978-5-93286-093-9
ISBN-10: 5-93286-093-6

Книга «Perl: изучаем глубже» – продолжение мирового бестселлера «Learning Perl» («Изучаем Perl»), известного под названием «Лама». Издание поможет вам перешагнуть грань, отделяющую любителя от профессионала, и научит писать на Perl настоящие программы, а не разрозненные сценарии. Материал изложен компактно и в занимательной форме, главы завершаются упражнениями, призванными помочь закрепить полученные знания. Рассмотрены пакеты и пространства имен, ссылки и области видимости, создание и использование модулей. Вы научитесь с помощью ссылок управлять структурами данных произвольной сложности, узнаете, как обеспечить совместимость программного кода, написанного разными программистами. Уделено внимание и ООП, которое поможет повторно использовать части кода. Обсуждаются создание дистрибутивов, аспекты тестирования и передача собственных модулей в CPAN.

Книга адресована широкому кругу программистов, знакомых с основами Perl и стремящихся повысить свою квалификацию как в написании сценариев, так и в ООП, и призвана помочь им научиться писать эффективные, надежные и изящные программы.

ISBN-13: 978-5-93286-093-9
ISBN-10: 5-93286-093-6
ISBN 0-596-10206-2 (англ)

© Издательство Символ-Плюс, 2007
Authorized translation of the English edition © 2006 O'Reilly Media Inc. This translation is published and sold by permission of O'Reilly Media Inc., the owner of all rights to publish and sell the same.

Все права на данное издание защищены Законодательством РФ, включая право на полное или частичное воспроизведение в любой форме. Все товарные знаки или зарегистрированные товарные знаки, упоминаемые в настоящем издании, являются собственностью соответствующих фирм.

Издательство «Символ-Плюс». 199034, Санкт-Петербург, 16 линия, 7,
тел. (812) 324-5353, edit@symbol.ru. Лицензия ЛП N 000054 от 25.12.98.
Налоговая льгота – общероссийский классификатор продукции
ОК 005-93, том 2; 953000 – книги и брошюры.

Подписано в печать 19.10.2007. Формат 70×100 1/16 . Печать офсетная.
Объем 20 печ. л. Тираж 2000 экз. Заказ №
Отпечатано с готовых диапозитивов в ГУП «Типография «Наука»
199034, Санкт-Петербург, 9 линия, 12.

Оглавление

Вступительное слово 11

Предисловие 12

1. Введение 19

Что вы должны знать? 20
Как быть со сносками? 20
Как быть с упражнениями? 21
Что делать, если я преподаю Perl? 21

2. Основы 22

Операторы списков 22
Организация ловушек ошибок с помощью eval 27
Исполнение программного кода, созданного динамически 28
Упражнения 29

3. Модули 31

Стандартный дистрибутив 31
Использование модулей 32
Функциональные интерфейсы 33
Как составить список импорта 34
Объектно-ориентированные интерфейсы 35
Типичный объектно-ориентированный модуль Math:BigInt 35
Единая архивная сеть Perl 36
Установка модулей из CPAN 37
Настройка списка каталогов для поиска модулей 38
Упражнения 41

4. Введение в ссылки 43

Выполнение однотипных действий с разными массивами 43
Ссылки на массивы 45
Разыменование ссылок на массивы 46
Избавляемся от фигурных скобок 48

Модификация массивов . . . 49
Вложенные структуры данных . . . 49
Упрощаем доступ к вложенным структурам с помощью стрелок . . . 52
Ссылки на хеши . . . 53
Упражнения . . . 55

5. Ссылки и области видимости . . . 57

Несколько ссылок на данные . . . 57
А если это было имя структуры? . . . 59
Подсчет ссылок и вложенные структуры данных . . . 60
Ошибки при подсчете ссылок . . . 62
Создание анонимных массивов . . . 64
Создание анонимных хешей . . . 67
Автовивификация . . . 69
Автовивификация и хеши . . . 72
Упражнения . . . 74

6. Управление сложными структурами данных . . . 76

Использование отладчика для просмотра данных со сложной структурой . . . 76
Просмотр данных со сложной структурой с помощью модуля Data::Dumper . . . 81
YAML . . . 83
Сохранение данных со сложной структурой с помощью модуля Storable . . . 84
Операторы grep и map . . . 86
Обходное решение . . . 86
Выбор и модификация данных со сложной структурой . . . 88
Упражнения . . . 89

7. Ссылки на подпрограммы . . . 91

Ссылки на именованные подпрограммы . . . 91
Анонимные подпрограммы . . . 96
Подпрограммы обратного вызова . . . 98
Замыкания . . . 99
Подпрограмма как возвращаемое значение другой подпрограммы . . . 101
Использование переменных замыканий для ввода данных . . . 104
Переменные замыканий как статические локальные переменные . . . 105
Упражнения . . . 107

8. Ссылки на дескрипторы файлов 109

Старый способ 109
Улучшенный способ 110
Способ еще лучше 112
IO::Handle 112
Ссылки на дескрипторы каталогов 117
Упражнения 118

9. Практические приемы работы со ссылками 120

Краткий обзор способов сортировки 120
Сортировка по индексам 122
Эффективность алгоритмов сортировки 124
Преобразование Шварца 126
Многоуровневая сортировка на основе преобразования Шварца 127
Данные с рекурсивной организацией 127
Построение структур данных с рекурсивной организацией 129
Отображение данных с рекурсивной организацией 132
Упражнения 133

10. Разработка больших программ 135

Ликвидация повторяющихся участков программного кода 135
Вставка программного кода с помощью eval 137
С помощью оператора do 137
С помощью директивы require 139
require и @INC 141
Конфликт имен 144
Имена пакетов как разделители пространств имен 146
Область видимости директивы package 148
Пакеты и лексические переменные 149
Упражнения 149

11. Введение в объекты 151

Если бы мы могли говорить на языке зверей... 151
Вызов метода с помощью оператора «стрелка» 153
Дополнительный параметр при вызове метода 154
Вызов второго метода с целью упрощения 155
Несколько замечаний о массиве @ISA 156
Перекрытие методов 158
Поиск унаследованного метода 160
SUPER способ добиться того же самого 161
Зачем нужен аргумент @_ 162
Что мы узнали... 162

Упражнения . . . 162

12. **Объекты и данные** . . . 164

Лошадь лошади рознь . . . 164
Вызов метода экземпляра . . . 166
Доступ к данным экземпляра . . . 166
Как создать лошадь . . . 167
Наследование конструктора . . . 168
Создание метода, работающего как с экземплярами, так и с классами . . . 169
Добавление параметров к методам . . . 170
Более сложные экземпляры . . . 171
Лошадь другого цвета . . . 172
Что возвращать . . . 173
Не открывайте черный ящик . . . 175
Оптимизация методов доступа . . . 176
Операция чтения и записи в одном методе . . . 176
Ограничение доступа к методам только по имени класса или только для экземпляров класса . . . 177
Упражнения . . . 178

13. **Уничтожение объектов** . . . 179

Уборка мусора . . . 179
Уничтожение вложенных объектов . . . 181
Вторичная переработка . . . 185
Форма косвенного обращения к объектам . . . 186
Дополнительные переменные экземпляра в подклассах . . . 188
Переменные класса . . . 190
Слабые ссылки . . . 192
Упражнения . . . 195

14. **Дополнительные сведения об объектах** . . . 196

Методы класса UNIVERSAL . . . 196
Проверка возможностей объектов . . . 197
Метод AUTOLOAD как последняя инстанция . . . 199
Применение AUTOLOAD для реализации методов доступа . . . 200
Более простой способ создания методов доступа . . . 201
Множественное наследование . . . 203
Упражнения . . . 204

15. **Экспортирование** . . . 206

Что делает директива use . . . 206
Импорт с помощью модуля Exporter . . . 208

@EXPORT и @EXPORT_OK 208
%EXPORT_TAGS 210
Экспорт имен в объектно-ориентированных модулях 211
Собственные подпрограммы импорта 213
Упражнения 215

16. Создание дистрибутива 216

Собрать дистрибутив можно разными способами 217
Программа h2xs 218
Файл README 220
Встроенная документация 226
Управление дистрибутивом с помощью Makefile.PL 229
Изменение каталога установки (PREFIX=...) 231
Тривиальная команда make test 232
Тривиальная команда make install 233
Тривиальная команда make dist 234
Дополнительные каталоги с библиотеками 235
Упражнения 236

17. Основы тестирования 237

Чем больше тестов, тем лучше программный код 237
Простейший сценарий с тестами 238
Искусство тестирования 239
Тестирующая система 242
Разработка тестов с помощью Test::More 244
Тестирование объектно-ориентированных особенностей 247
Списки To-Do тестов 249
Пропуск тестов 249
Более сложные тесты (несколько тестовых сценариев) 250
Упражнения 251

18. Дополнительные сведения о тестировании 253

Тестирование длинных строк 253
Тестирование файлов 254
Тестирование устройств STDOUT и STDERR 256
Работа с ложными объектами 258
Тестирование документации в формате POD 260
Степень покрытия тестами 261
Разработка собственных модулей Test::* 262
Упражнения 266

19. Передача модулей в CPAN . . . 267
Всемирная сеть архивов Perl . . . 267
Первый шаг . . . 268
Подготовка дистрибутива . . . 269
Передача дистрибутива на сервер . . . 270
Объявление о выпуске модуля . . . 271
Тестирование на нескольких платформах . . . 271
Подумайте о написании статьи или доклада . . . 272
Упражнения . . . 272

A. Ответы к упражнениям . . . 273

Алфавитный указатель . . . 302

Вступительное слово

Объектно-ориентированный механизм языка программирования Perl являет собой пример ловкости рук без всякого обмана. Он берет набор возможностей, не связанных с объектно-ориентированным стилем программирования, таких как пакеты, ссылки, хеши, массивы, подпрограммы и модули, и с помощью несложных заклинаний превращает их в полнофункциональные объекты, классы и методы.

Благодаря этому трюку программист, основываясь на своих познаниях языка Perl, может легко и просто перейти к объектно-ориентированному стилю программирования, не преодолевая горы нового синтаксиса и не переплывая океаны новых технологий. Это также означает, что объектно-ориентированные возможности языка Perl можно осваивать постепенно, по мере необходимости отбирая те, что наилучшим образом подходят для решения поставленных задач.

Однако здесь кроется одна проблема. Поскольку все эти пакеты, ссылки, хеши, массивы, подпрограммы и модули составляют основу объектно-ориентированного механизма, для использования объектно-ориентированных возможностей языка Perl необходимо знать принципы работы с пакетами, ссылками, хешами, подпрограммами и модулями.

Трудность именно в этом. Кривая обучения не исчезла, она всего лишь сократилась на полдесятка шагов.

Какие же *не*объектно-ориентированные возможности языка Perl надо изучить, чтобы можно было взяться за объектно-ориентированные?

Ответу на этот вопрос и посвящена данная книга. На ее страницах Рэндал, опираясь на 20-летний опыт работы с языком Perl и 40-летний опыт просмотра фильмов «Остров Джиллигана» и «Мистер Эд», описывает компоненты языка Perl, которые все вместе составляют фундамент его объектно-ориентированных возможностей. И, что еще лучше, на примерах показывает, как из этих компонентов создавать классы и объекты.

Итак, если объекты языка Perl вызывают у вас чувства, подобные тем, которые испытал Джиллиган на необитаемом острове, эта книга – как раз то, что доктор прописал.

Кроме того, вся информация в ней прямо из первых рук.

– Дэмиан Конвей (Damian Conway), май 2003

Предисловие

Более десяти лет тому назад (практически вечность по меркам Интернета) Рэндал Шварц написал первое издание книги «Learning Perl»[1]. За прошедшие годы сам Perl из «крутого» языка сценариев, используемого в первую очередь системными администраторами UNIX, вырос в полноценный объектно-ориентированный язык программирования, способный функционировать на практически любой платформе, известной человечеству.

Объем всех четырех изданий «Learning Perl» оставался практически неизменным (примерно 300 страниц), как в основном неизменным оставался и ее материал, рассчитанный на начинающих программистов. Однако времена изменились, и теперь о Perl можно рассказать значительно больше, чем когда появилось первое издание книги.

Рэндал назвал первое издание книги «Learning Perl Objects, References, and Modules», а сейчас книга получила название «Intermediate Perl» (Perl средней сложности), но на наш взгляд ей больше подошло бы название «Learning More Perl» (Изучаем Perl глубже)[2]. Данная книга продолжает обсуждение тем с того места, где оно было закончено в книге «Learning Perl». Здесь мы покажем вам, как писать большие программы на языке Perl.

Как и в «Learning Perl», мы старались сделать каждую главу настолько маленькой, чтобы ее можно было прочитать за час-другой. Каждая глава заканчивается серией упражнений, которые помогут вам на практических примерах закрепить только что прочитанный материал. Кроме того, в конце книги вы найдете приложение, в котором содержатся решения всех упражнений. Как и в «Learning Perl», материал этой книги подается в той же последовательности, что и в курсах обучения языку Perl, которые проводятся нами в компании Stonehenge Consulting Services.

---

[1] Рэндал Шварц, Том Кристиансен «Изучаем Perl». – Пер. с англ. – BHV-Киев, 1999.

[2] Не спрашивайте, почему книга не была названа именно так. Мы получили более 300 предложений по этой теме. На самом деле невозможно прекратить изучение Perl, поэтому название «Изучаем Perl глубже» фактически ничего не говорит о книге. Наш редактор выбрал название, которое говорит о том, чего следует ожидать от книги.

Чтобы извлечь максимум пользы из этой книги, вам необязательно быть гуру в UNIX и даже необязательно быть пользователем UNIX. Все, о чем говорится в этой книге, одинаково хорошо подходит как для Windows ActivePerl, так и для любой другой современной реализации. Чтобы иметь возможность пользоваться этой книгой, вам необходимо ознакомиться с книгой «Learning Perl» и гореть желанием продолжать двигаться вперед.

Структура книги

Данную книгу следует читать, начиная с первых глав, в том порядке, в каком они следуют, останавливаясь для выполнения упражнений. Материал каждой главы основан на предыдущих главах, и при обсуждении новой темы мы будем исходить из предположения, что предыдущие главы уже были вами прочитаны.

Глава 1 «Введение»

Содержит вводные положения.

Глава 2 «Основы»

Описывает некоторые промежуточные положения, знание которых потребуется при прочтении оставшейся части книги.

Глава 3 «Модули»

Описывает порядок работы с основными модулями Perl и с модулями сторонних производителей. Позже в этой же книге мы покажем, как создавать собственные модули, но в данной главе мы остановимся на использовании существующих модулей.

Глава 4 «Введение в ссылки»

Рассказывает о том, как организовать перенаправление, чтобы один и тот же программный код мог работать с различными наборами данных.

Глава 5 «Ссылки и области видимости»

Рассказывает о том, как Perl работает с указателями на данные, и дает краткое введение в анонимные структуры данных и автовивификацию.

Глава 6 «Управление сложными структурами данных»

Описывает создание структур данных с произвольной глубиной вложенности, включая массивы массивов и хеши хешей, обращение к ним и вывод их содержимого.

Глава 7 «Ссылки на подпрограммы»

Описывает поведение анонимных подпрограмм, которые могут создаваться динамически для последующего использования.

Глава 8 «Ссылки на дескрипторы файлов»

Описывает, как можно хранить дескрипторы файлов в скалярных переменных для передачи между различными частями программы или для сохранения в структурах данных.

Глава 9 «Практические приемы работы со ссылками»

Сложные операции сортировки, преобразование Шварца и работа с рекурсивно определенными данными.

Глава 10 «Разработка больших программ»

Рассматривает вопросы создания больших программ из нескольких файлов с программным кодом, разнесенным по разным пространствам имен.

Глава 11 «Введение в объекты»

Работа с классами, вызов методов, наследование и переопределение.

Глава 12 «Объекты и данные»

Экземпляры данных, конструкторы и методы доступа.

Глава 13 «Уничтожение объектов»

Описывает поведение объектов при уничтожении, включая объекты, существующие постоянно.

Глава 14 «Дополнительные сведения об объектах»

Множественное наследование, автоматические методы и ссылки на дескрипторы файлов.

Глава 15 «Экспортирование»

Как работает директива `use`, как определить, что нужно экспортировать, и как создать собственную процедуру импорта.

Глава 16 «Создание дистрибутивов»

Описывает порядок создания модулей, готовых к распространению, включая платформонезависимые инструкции по установке.

Глава 17 «Основы тестирования»

Описывает порядок тестирования программного кода с целью проверки его функциональности.

Глава 18 «Дополнительные сведения о тестировании»

Описываются более сложные аспекты тестирования программного кода и метаданных, такие как документация и покрытие тестами.

Глава 19 «Передача модулей в CPAN»

Описывает, как можно отправить свои разработки в CPAN.

Приложение A содержит решения всех упражнений.

Типографские соглашения

В книге приняты следующие соглашения по оформлению текста:

`Моноширинным шрифтом`

Выделены имена функций, модулей, файлов, переменных окружения, фрагменты программного кода и пр.

Курсивом

Выделены наиболее важные моменты и вновь вводимые термины.

Примеры программного кода

Данная книга призвана помочь вам в работе. Вы можете вставлять примеры программного кода из этой книги в свои приложения и в документацию, и для этого не надо обращаться в издательство O'Reilly за разрешением. Например, если вы пишете программу и заимствуете несколько отрывков программного кода из книги, вам не нужно обращаться за разрешением. Не требуется разрешение и для цитирования данной книги или примеров из нее при ответе на вопросы.

Если же вы собираетесь воспроизводить значительные фрагменты программного кода из этой книги (например, в документации), то разрешение необходимо. Разрешение нужно получить и в случае, если вы планируете продавать или распространять компакт-диски с примерами из этой книги.

Мы не требуем добавлять ссылку на первоисточник при цитировании (но совсем не против этого). Под ссылкой на первоисточник мы подразумеваем указание авторов, издательства и ISBN, например «Intermediate Perl, by Randal L. Schwartz, Brian D. Foy, and Tom Phoenix. Copyright 2006 O'Reilly Media, Inc., 0-596-10206-2».

За получением разрешения на использование значительных объемов программного кода примеров из этой книги обращайтесь по адресу *permissions@oreilly.com*.

Отзывы и предложения

С вопросами и предложениями, касающимися этой книги, обращайтесь в издательство:

O'Reilly Media

1005 Gravenstein Highway North

Sebastopol, CA 95472

(800) 998-9938 (в Соединенных Штатах Америки или в Канаде)

(707) 829-0515 (международный)

(707) 829-0104 (факс)

Список опечаток, файлы с примерами и другую дополнительную информацию вы найдете на сайте книги:

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

Свои пожелания и вопросы технического характера отправляйте по адресу:

bookquestions@oreilly.com

Дополнительную информацию о книгах, обсуждения, центр ресурсов издательства O'Reilly вы найдете на сайте:

http://www.oreilly.com

Safari Enabled

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

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

Благодарности

Рэндал. В предисловии к первому изданию книги «Learning Perl» я выразил свою признательность Бивертону Мак-Менамину (Beaverton McMenamin) – владельцу паба Cedar Hills (Кедровые холмы), что находится рядом с моим домом, за бесплатно предоставленный кабинет, где я имел обыкновение писать черновики книги на моем Powerbook 140. Этот паб стал для меня талисманом, приносящим удачу. Практически все свои книги (включая и эти слова) я писал здесь и очень надеюсь, что удача мне не изменит и на этот раз!

Теперь в этом пабе подают прекрасное пиво, сваренное тут же в маленькой пивоварне, и сладкие сэндвичи, но пропала моя любимая хлебная пицца, которую заменили ежевичным коктейлем (местный рецепт) и острой джамбалайей. (Кроме того, здесь появились два новых кабинета и несколько столов.) Ну а вместо Powerbook 140 у меня теперь более современный Titanium Powerbook, у которого диск в 1000 раз больше, оперативной памяти в 500 раз больше и процессор в 200 раз быстрее. На нем установлена полноценная UNIX-подобная операционная система (OS X) вместо ограниченной версии Mac OS. Все свои черновики (включая и этот) я отправляю через модем сотового телефона на скорости 144 K, могу напрямую общаться со своими рецензентами, и мне не нужно возвращаться домой к моему модему, передающему данные по телефонной линии со скоростью 9600 бод. Как изменились времена!

Еще раз большое спасибо всем, кто работает в «Кедровых холмах», за их неизменное гостеприимство.

Как и в четвертом издании книги «Learning Perl», я должен отметить, что многим обязан своим студентам из Stonehenge Consulting Services за их каверзные (?) вопросы, которые появлялись, когда сложность материала превышала уровень их подготовки. Благодаря им я смог продолжить совершенствование материала, который лег в основу этой книги.

Следует заметить, что все началось с полудневного курса «Что нового в Perl 5?» Марджи Левин (Margie Levine) из Silicon Graphics, а также моего собственного четырехдневного курса «Лама» (Llama) (в то время основанного на Perl версии 4). Со временем у меня возникла идея превратить эти короткие заметки в полноценный курс и подтолкнуть сотрудника компании Stonehenge Джозефа Холла (Joseph Hall) к участию в решении этой задачи. (Он один из тех, кто отбирал примеры программного кода для курса.) Джозеф разработал двухдневный курс для Stonehenge и одновременно написал прекрасную книгу «Effective Perl Programming», которая затем стала использоваться как учебник.

За эти годы в разработке курса «Пакеты, ссылки, объекты и модули» принимали участие многие преподаватели из Stonehenge, включая Чипа Зальценберга (Chip Salzenberg) и Тэда Мак-Клеллана (Ted McClellan). Но большая часть изменений и дополнений была внесена Томом Фениксом (Tom Phoenix), который становился «служащим месяца» в компании Stonehenge настолько часто, что мне, наверное, придется уступить ему мое привилегированное место на парковке. Том отлично управляется с материалами (как Тэд управляется с делами), благодаря чему я могу спокойно сосредоточиться на своих обязанностях президента и дворника компании Stonehenge.

Том Феникс написал большую часть примеров для этой книги и своевременно предоставлял свои рецензии во время моей работы над книгой. Его рукой были написаны целые абзацы, так что мне оставалось только вставлять их на место той бессмыслицы, что была написана мною. У нас получилась отличная команда, которая дружно работает и в аудитории, и над книгой. Именно за приложенные усилия мы признали Тома соавтором, но я готов взять всю вину на себя, если какая-либо часть книги вам не понравится, потому что, скорее всего, в этом нет вины Тома.

И последний, но не в последнюю очередь, кому я хотел бы выразить свою благодарность, – это Брайан Д. Фой (brian d foy), который примкнул к работе над книгой, начиная со второго ее издания, и предложил массу изменений и дополнений к этому изданию.

Разумеется, книга не состоялась бы, не будь темы для обсуждения и каналов распространения, поэтому я хочу выразить свою признательность Ларри Уоллу (Larry Wall) и Тиму О'Рейли (Tim O'Reilly).

Спасибо вам, ребята, за то что вы создали компанию, которая оплачивала мне мои затраты в течение 15 последних лет.

И, как обычно, отдельное спасибо Лейле и Джеку, которые научили меня всему, что я знаю о писательской деятельности, и убедили меня в том, что я должен быть чуть большим (?), чем программист, который умеет писать. Благодаря им я стал писателем, который умеет программировать. Спасибо вам.

Спасибо и вам, уважаемый читатель. Именно для вас я трудился долгие часы, потягивая холодное пиво и поедая пудинг, стараясь не залить клавиатуру моего ноутбука. Спасибо вам за то, что вы читаете мою книгу. Я искренне надеюсь, что внес свой вклад (пусть и незначительный) в ваше образование. Если вы встретите меня когда-нибудь на улице, просто скажите: «Привет!».[1] Мне это будет приятно. Спасибо вам.

Брайан. В первую очередь я хотел бы сказать спасибо Рэндалу, так как впервые я познакомился с Perl благодаря первому изданию его книги «Learning Perl» и многое узнал, преподавая курсы «Лама» и «Альпака» в компании Stonehenge Consulting. Учить других – часто лучший способ научиться самому.

Мне удалось убедить Рэндала в необходимости обновить книгу «Learning Perl», а когда эта работа была закончена, я заметил ему, что пора обновить и эту книгу. Наш редактор Элисон Рэндал (Allison Randal) согласилась с этим и приложила максимум усилий, чтобы не нарушить наш график.

Отдельное спасибо Стейси (Stacey), Бастеру (Buster), Мими (Mimi), Роско (Rosco), Амелии (Amelia), Лиле (Lila) и всем тем, кто пытался отвлечь меня от работы, когда я был занят.

От нас обоих. Спасибо нашим рецензентам: Дэвиду Адлеру (David H. Adler), Стефену Дженкинсу (Stephen Jenkins), Кевину Мельтцеру (Kevin Meltzer), Мэттью Масгрову (Matthew Musgrove), Эндрю Сэвиджу (Andrew Savige) и Риккардо Сигнесу (Ricardo Signes) за их комментарии к рукописи этой книги.

Спасибо нашим студентам, которые помогли нам понять, какие части курса необходимо пересмотреть и дополнить. Именно благодаря вам мы испытываем чувство гордости за свою работу.

Спасибо членам группы Perl Mongers, кто принимал нас в своих городах как родных. Давайте встретимся еще когда-нибудь.

И наконец, огромное спасибо Ларри Уоллу за его большие и мощные игрушки, которые позволили нам сделать свою работу намного быстрее, проще и с увлечением.

---

[1] К тому же вы можете спросить меня что-нибудь о Perl. Я не возражаю.

4

Введение в ссылки

Ссылки – это основа сложных структур данных, объектно-ориентированного программирования (ООП) и необычной работы с подпрограммами. Ссылки были добавлены в Perl между версиями 4 и 5.

Скалярные переменные Perl могут хранить одиночное значение. Массив представляет собою упорядоченный список из одного или более скалярных значений. Хеши могут хранить коллекции скаляров – одни в виде ключей, другие в виде значений. Хотя скаляр и может быть любой строкой, в которую можно «втиснуть» массив или хеш, ни один из этих трех типов данных не может использоваться для описания сложных взаимосвязей между данными. Для этого лучше всего подходят ссылки. Чтобы прочувствовать, насколько важны ссылки, начнем с одного примера.

Выполнение однотипных действий с разными массивами

Прежде чем «Minnow» (Пескарь) сможет отплыть от пристани (например, на трехчасовую экскурсию), мы должны проверить всех пассажиров и членов экипажа на наличие у них всего необходимого. Скажем, для экскурсии по морю каждый, кто находится на борту, должен иметь солнечные очки, крем или лосьон от загара, фляжку с водой и накидку от дождя. Чтобы проверить, как подготовился к этому мероприятию Шкипер, достаточно написать совсем немного программного кода:

```
my @required = qw(солнечные_очки крем фляжка_с_водой накидка);
my @skipper = qw(голубая_рубашка шляпа накидка солнечные_очки крем);

for my $item (@required) {
    unless (grep $item eq $_, @skipper) { # нет в списке?
```

```
            print "Шкипер: отсутствует $item.\n";
        }
    }
```

В скалярном контексте grep возвращает количество раз, когда в результате вычисления выражения $item eq $_ было получено значение «истина» (1 – это истина, 0 – ложь).[1] Если получилось значение 0 (то есть «ложь»), мы выводим сообщение.

Разумеется, чтобы проверить экипировку Джиллигана или Профессора, нам придется написать:

```
my @gilligan = qw(красная_рубашка шляпа счастливые_носки фляжка_с_водой);
for my $item (@required) {
    unless (grep $item eq $_, @gilligan) { # нет в списке?
        print "Джиллиган: отсутствует $item.\n";
    }
}
my @professor = qw(крем фляжка_с_водой рулетка батарейки радиоприемник);
for my $item (@required) {
    unless (grep $item eq $_, @professor) { # нет в списке?
        print "Профессор: отсутствует $item.\n";
    }
}
```

Вы наверняка заметили, что в этом примере постоянно повторяется слишком большой объем программного кода. Неплохо было бы выделить повторяющиеся строки в виде отдельной подпрограммы (и вы совершенно правы):

```
sub check_required_items {
    my $who = shift;
    my @required = qw(солнечные_очки крем фляжка_с_водой накидка);
    for my $item (@required) {
        unless (grep $item eq $_, @_) { # нет в списке?
            print "$who: отсутствует $item.\n";
        }
    }
}

my @gilligan = qw(красная_рубашка шляпа счастливые_носки фляжка_с_водой);
check_required_items('Джиллиган', @gilligan);
```

Изначально подпрограмме передаются 5 аргументов в виде массива @_: имя Джиллиган и четыре предмета, принадлежащие Джиллигану. После выполнения функции shift в массиве @_ останутся только предметы. Затем grep будет вынимать предметы из контрольного списка и проверять их наличие в списке имеющихся предметов.

[1] Есть более эффективный способ проверки наличия элемента в очень больших списках. Но для нескольких элементов приведенный способ, пожалуй, самый простой.

Пока все идет неплохо. Аналогичным образом мы можем проверить Шкипера и Профессора, добавив всего несколько строк:

```
my @skipper = qw(голубая_рубашка шляпа накидка солнечные_очки крем);
my @professor = qw(крем фляжка_с_водой рулетка батарейки радиоприемник);
check_required_items('Шкипер', @skipper);
check_required_items('Профессор', @professor);
```

Для проверки других пассажиров корабля придется повторить те же действия. Этот программный код отвечает первоначальным требованиям, но в нем имеются две неувязки:

- Чтобы создать массив `@_`, Perl вынужден копировать все содержимое проверяемого массива. Это не очень важно, если объем массива невелик, но если массив большой, то копировать его элементы только ради того, чтобы передать их в подпрограмму, слишком расточительно.
- Предположим, что требуется изменить оригинальный массив, чтобы, например, включить в него обязательные элементы. Поскольку подпрограмме передается копия массива («передача аргумента по значению»), все изменения, произведенные в массиве `@_`, будут потеряны после выхода из подпрограммы.[1]

Любую из этих неувязок можно устранить, передав массив не по значению, а по ссылке. И это как раз то, что доктор (то есть Профессор) прописал.

Ссылки на массивы

Кроме всего прочего, символом обратного слэша (\) обозначается оператор «взятия ссылки». Когда этот оператор ставится перед именем массива, например `\@skipper`, мы получаем *ссылку* на этот массив. Ссылка на массив сродни указателю: она указывает на массив, но сама не является массивом.

Ссылка может применяться там же, где и скалярные величины. Ссылки могут указывать на элементы массива или хеша или на обычные скалярные переменные, например так:

```
my $reference_to_skipper = \@skipper;
```

Ссылки можно копировать:

```
my $second_reference_to_skipper = $reference_to_skipper;
```

или даже:

```
my $third_reference_to_skipper = \@skipper;
```

[1] На самом деле, после того как `shift` изменит соответствующую переменную, скалярным элементам массива `@_` можно присвоить новые значения, но это по-прежнему не дает нам возможности расширять массив дополнительными элементами.

Мы можем оперировать всеми тремя ссылками и даже утверждать, что они идентичны, поскольку ссылаются на один и тот же массив.

```
if ($reference_to_skipper == $second_reference_to_skipper) {
    print "Эти ссылки идентичны.\n";
}
```

В данном случае сравниваются числовые представления двух ссылок. Числовое представление – это уникальный адрес в памяти массива `@skipper`, структура которого не меняется в течение всего времени жизни переменной. Если попробовать посмотреть на строковое представление ссылки (с помощью оператора `eq` или `print`), мы увидим строку:

```
ARRAY(0x1a2b3c)
```

которая содержит уникальный адрес массива (о чем говорит шестнадцатеричная форма представления). Кроме того, данная строка свидетельствует, что это ссылка на массив. Разумеется, если нечто подобное появится в выводе нашей программы, то почти наверняка будет означать ошибку, поскольку дампы шестнадцатеричных данных не представляют интереса для пользователей!

Ссылки допускают копирование и передачу подпрограммам, и это обстоятельство можно использовать для передачи нашей подпрограмме ссылки на массив:

```
my @skipper = qw(голубая_рубашка шляпа накидка солнечные_очки крем);
check_required_items("Шкипер", \@skipper);

sub check_required_items {
    my $who = shift;
    my $items = shift;
    my @required = qw(солнечные_очки крем фляжка_с_водой накидка);
    ...
}
```

Теперь переменная `$items` внутри подпрограммы представляет собой ссылку на массив `@skipper`. Но как добраться до самого массива, имея ссылку на него? Для этого достаточно *разыменовать* ссылку.

Разыменование ссылок на массивы

Если внимательно посмотреть на имя массива `@skipper`, можно заметить, что оно состоит из двух частей: символа `@` и собственно имени массива. Точно так же запись `$skipper[1]` синтаксически состоит из имени массива и дополнительной синтаксической конструкции, которая указывает на второй элемент массива (индекс массива со значением 1 соответствует второму элементу массива, поскольку первый элемент имеет индекс 0).

Хитрость заключается в том, чтобы заключить ссылку в фигурные скобки. Таким образом, везде, где необходимо обратиться к массиву, доста-

точно написать имя ссылки, заключенное в фигурные скобки: { $items }. Например, следующие две строки ссылаются на весь массив:

```
@ skipper
@{ $items }
```

а следующие две – на второй элемент массива:

```
$ skipper [1]
${ $items }[1]
```

Посредством ссылок на массивы нам удалось отделить программный код обращения к массиву от самого массива. Посмотрим, как это повлияло на оставшуюся часть подпрограммы:

```
sub check_required_items {
    my $who = shift;
    my $items = shift;
    my @required = qw(солнечные_очки крем фляжка_с_водой накидка);

    for my $item (@required) {
        unless (grep $item eq $_, @{$items}) { # нет в списке?
            print "$who: отсутствует $item.\n";
        }
    }
}
```

Мы всего лишь заменили @_ (копию проверяемого списка) на @{$items}, то есть разыменовали ссылку на оригинальный массив. Теперь, как и прежде, мы можем вызвать подпрограмму несколько раз:

```
my @skipper = qw(голубая_рубашка шляпа накидка солнечные_очки крем);
check_required_items('Шкипер', \@skipper);

my @professor = qw(крем фляжка_с_водой рулетка батарейки радиоприемник);
check_required_items('Профессор', \@professor);

my @gilligan = qw(красная_рубашка шляпа счастливые_носки фляжка_с_водой);
check_required_items('Джиллиган', \@gilligan);
```

Каждый раз $items будет указывать на разные массивы, что позволяет выполнять проверки посредством одного и того же программного кода. Этот пример демонстрирует один из самых важных случаев применения ссылок: отделение логики программы от структур данных позволяет многократно задействовать один и тот же программный код.

Передача массива в подпрограмму по ссылке устраняет первую из двух неувязок, о которых мы говорили выше. Теперь, вместо того чтобы целиком копировать исходный массив в массив @_, мы передаем подпрограмме единственный элемент – ссылку на массив.

Можно ли убрать два вызова shift в начале подпрограммы? Конечно, но для этого придется пожертвовать ясностью кода подпрограммы:

```
sub check_required_items {
    my @required = qw(солнечные_очки крем фляжка_с_водой накидка);
```

```
    for my $item (@required) {
        unless (grep $item eq $_, @{$_[1]}) { # нет в списке?
            print "$_[0]: отсутствует $item.\n";
        }
    }
}
```

У нас в массиве @_ по-прежнему два элемента. Первый – это имя пассажира или члена экипажа, которое фигурирует в выводе сообщения об ошибке. Второй элемент – это ссылка на массив с экипировкой, который передается оператору grep.

Избавляемся от фигурных скобок

В большинстве случаев ссылки служат для обеспечения доступа к скалярным величинам, таким как @{$items} или ${$items}[1]. В таких ситуациях фигурные скобки можно опустить и записывать ссылки так: @$items или $$items[1].

Однако избавиться от фигурных скобок невозможно, если значение в скобках не является скалярной величиной. Например в такой форме записи: @{$_[1]}, которую мы использовали в последней версии подпрограммы. Здесь мы не можем избавиться от фигурных скобок, поскольку в данном случае элемент массива представляет собой массив, а не скалярную величину.

В соответствии с этим правилом нетрудно определить, где должны стоять «отсутствующие» фигурные скобки. Так, если мы встречаем запись в виде $$items[1], то можно с уверенностью утверждать, что фигурные скобки должны обрамлять простую скалярную переменную $items. Отсюда следует вывод, что $items представляет собой ссылку на массив.

Таким образом, самая простая для восприятия человеком версия подпрограммы может выглядеть следующим образом:

```
sub check_required_items {
    my $who = shift;
    my $items = shift;

    my @required = qw(солнечные_очки крем фляжка_с_водой накидка);
    for my $item (@required) {
        unless (grep $item eq $_, @$items) { # нет в списке?
            print "$who: отсутствует $item.\n";
        }
    }
}
```

Эта версия отличается от предыдущей лишь отсутствием фигурных скобок.

Модификация массивов

Теперь вы знаете, как с помощью ссылок избавиться от ненужного копирования информации. Посмотрим, как внести изменения в оригинальный массив.

Каждый отсутствующий элемент экипировки мы будем добавлять в массив, обращая на него внимание пассажира:

```
sub check_required_items {
    my $who = shift;
    my $items = shift;

    my @required = qw(солнечные_очки крем фляжка_с_водой накидка);
    my @missing = ( );

    for my $item (@required) {
        unless (grep $item eq $_, @$items) { # нет в списке?
            print "$who: отсутствует $item.\n";
            push @missing, $item;
        }
    }

    if (@missing) {
        print "Добавлены @missing в саквояж @$items пассажира $who.\n";
        push @$items, @missing;
    }
}
```

Обратите внимание на дополнительный массив `@missing`. Если в процессе проверки багажа пассажира обнаруживается отсутствующие обязательные элементы экипировки, мы записываем их в массив `@missing`. Если по окончании проверки массив не пустой, мы добавляем его содержимое к оригинальному массиву с экипировкой.

Ключевой является последняя строка подпрограммы. Здесь мы разыменовали ссылку на оригинальный массив `$items` и добавили в него содержимое массива `@missing`. При отсутствии ссылки на массив мы смогли бы изменить только локальную копию данных, что никак не повлияло бы на сам массив.

Кроме того, запись `@$items` (и более универсальная форма `@{$items}`) может применяться внутри строк, заключенных в двойные кавычки. Между символом `@` и остальной частью ссылки нельзя вставлять пробелы, зато внутри фигурных скобок они вполне допустимы, как в обычном программном коде Perl.

Вложенные структуры данных

В данном примере массив `@_` содержит два элемента, один из которых сам является массивом. А как быть, если мы получаем ссылку на массив, который сам содержит ссылки на другие массивы? Такая сложная организация данных может оказаться весьма удобной.

Например, мы можем обойти массивы Шкипера, Джиллигана и Профессора и построить один большой список предметов экипировки, имеющихся в наличии:

```
my @skipper = qw(голубая_рубашка шляпа накидка солнечные_очки крем);
my @skipper_with_name = ('Шкипер', \@skipper);
my @professor = qw(крем фляжка_с_водой рулетка батарейки радиоприемник);
my @professor_with_name = ('Профессор', \@professor);
my @gilligan = qw(красная_рубашка шляпа счастливые_носки фляжка_с_водой);
my @gilligan_with_name = ('Джиллиган', \@gilligan);
```

В данном случае массив @skipper_with_name содержит два элемента, второй из которых представляет собой ссылку на массив, аналогичную той, что мы передавали в подпрограмму. Теперь можно объединить всю информацию в единый список:

```
my @all_with_names = (
    \@skipper_with_name,
    \@professor_with_name,
    \@gilligan_with_name,
);
```

Обратите внимание: мы получили три элемента, каждый из которых является ссылкой на массив, состоящий из двух элементов: имени и списка имеющихся в наличии предметов. Изображение этой структуры приводится на рис. 4.1.

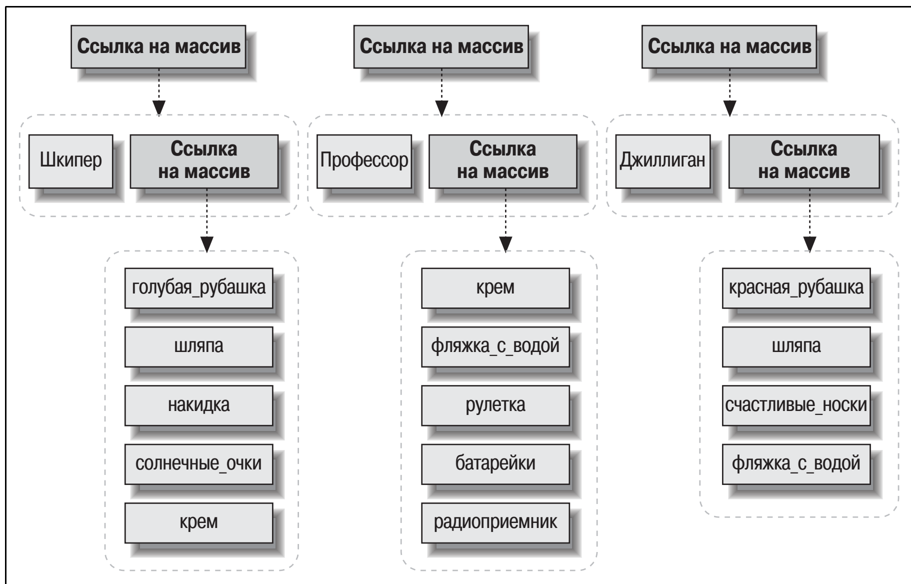

Рис. 4.1. *В массиве @all_with_names сосредоточена вся многоуровневая структура данных, состоящая из строк и ссылок на массивы*

Таким образом, `$all_with_names[2]` представляет собой ссылку на массив с информацией о Джиллигане. Если разыменовать ссылку как `@{$all_with_names[2]}`, получится массив из двух элементов: «Джиллиган» и ссылка на другой массив.

Как организовать доступ к данным по этой ссылке? Опять применяя наше правило: `${$all_with_names[2]}[1]`. Или, говоря другими словами, берем ссылку `$all_with_names[2]`, разыменовываем ее и обращаемся как к обычному массиву, например `$DUMMY[1]`, где `DUMMY` заменяется на `$all_with_names[2]`.

А как тогда обращаться к подпрограмме `check_required_items()`, имея такую структуру данных? Ниже приводится достаточно простой пример:

```
for my $person (@all_with_names) {
    my $who = $$person[0];
    my $provisions_reference = $$person[1];
    check_required_items($who, $provisions_reference);
}
```

Для этого не надо вносить изменения в саму подпрограмму. В процессе исполнения цикла переменная `$person` будет поочередно получать значения `$all_with_names[0]`, `$all_with_names[1]` и `$all_with_names[2]`. Когда разыменовывается ссылка `$$person[0]`, мы поочередно будем получать строки «Шкипер», «Профессор» и «Джиллиган». Соответственно разыменованная ссылка `$$person[1]` будет представлять список вещей, принадлежащих этим персонажам.

Разумеется, мы можем сократить объем кода, отказавшись от промежуточных переменных:

```
for my $person (@all_with_names) {
    check_required_items(@$person);
}
```

или даже так:

```
check_required_items(@$_) for @all_with_names;
```

Как видите, различные уровни оптимизации могут приводить к снижению ясности программного кода. Представьте себе, что может вам прийти в голову, когда через месяц вы попытаетесь разобраться в своем собственном программном коде. Если этот довод кажется вам неубедительным, представьте себе, каково будет другому программисту, который пожелает подхватить ваши начинания.[1]

[1] В издательстве O’Reilly вышла прекрасная книга «Perl Best Practicies» Дэмиана Конвея (Damian Conway), в которой приводятся 256 советов, как сделать программный код на языке Perl удобочитаемым и более простым в поддержке.

Упрощаем доступ к вложенным структурам с помощью стрелок

Взглянем еще раз на форму разыменования ссылок с помощью фигурных скобок. Согласно нашему предыдущему примеру ссылка на массив с предметами, принадлежащими Джиллигану, записывается как `${$all_with_names[2]}[1]`. Как теперь получить название первого предмета из списка? Для этого надо разыменовать данный элемент еще раз, то есть добавить еще одни фигурные скобки: `${${$all_with_names[2]}[1]}[0]`. Очень неудобная форма записи. А есть ли способ попроще? Конечно есть!

Форму записи `${DUMMY}[$y]` во всех случаях можно заменить более краткой формой `DUMMY->[$y]`. Другими словами, можно разыменовать ссылку на массив, указав в квадратных скобках требуемый элемент массива после выражения, определяющего ссылку на массив, и стрелки.

В нашем примере ссылку на массив с предметами, принадлежащими Джиллигану, можно получить так: `$all_with_names[2]->[1]`, а так – доступ к первому элементу этого массива: `$all_with_names[2]->[1]->[0]`. Вот это да! Такая форма записи воспринимается гораздо проще!

Есть и еще одно правило (как будто *эта форма* еще недостаточно проста): если стрелка встречается между элементами, обозначающими индекс, к примеру между квадратными скобками, то ее можно опустить: `$all_with_names[2]->[1]->[0]` превращается в `$all_with_names[2][1][0]`. Эта форма более удобочитаема.

Оператор стрелки обязательно должен присутствовать после имени ссылки, а между индексами его можно опустить. Почему? Представьте себе следующую ссылку на массив `@all_with_names`:

```
my $root = \@all_with_names;
```

Как теперь получить название первого предмета, принадлежащего Джиллигану?

```
$root -> [2] -> [1] -> [0]
```

В соответствии с правилом, определенным выше, стрелки между индексами можно опустить:

```
$root -> [2][1][0]
```

Но мы не можем опустить оператор стрелки, следующей за именем ссылки, потому что такая форма записи будет означать обращение к третьему элементу массива `$root`, то есть обращение к данным с совершенно иной структурой. Теперь сравним эту форму записи с универсальной, где есть фигурные скобки:

```
${${${$root}[2]}[1]}[0]
```

Форма записи со стрелкой выглядит намного лучше. Однако чтобы получить по ссылке весь массив с информацией о Джиллигане, нам пона-

добится форма записи с фигурными скобками. Например, чтобы получить весь массив с предметами, мы должны записать:

```
@{$root->[2][1]}
```

Такая форма записи должна читаться изнутри наружу:

- Берем ссылку `$root`.
- Разыменовываем ее как ссылку на массив и берем третий элемент массива (элемент с индексом 2).
- Разыменовываем его как ссылку на массив и берем второй элемент массива (элемент с индексом 1).
- Разыменовываем его как ссылку на массив и берем весь массив целиком.

Последний шаг не имеет краткой формы записи. Ну и ладно.[1]

Ссылки на хеши

Аналогично тому, как можно получить ссылки на массивы, точно так же можно получить ссылки на хеши. Здесь тоже используется символ обратного слэша (\), который является оператором «взятия ссылки»:

```
my %gilligan_info = (
    name => 'Джиллиган',
    hat => 'Белый',
    shirt => 'Красный',
    position => 'Первый помощник капитана',
);
my $hash_ref = \%gilligan_info;
```

Чтобы извлечь информацию о Джиллигане с помощью ссылки, мы должны разыменовать ее. Стратегия работы со ссылками на хеши похожа на стратегию работы со ссылками на массивы. Сначала мы записываем обращение к ссылке, как если бы это был сам хеш, а затем добавляем пару фигурных скобок. Например, выбрать значение конкретного ключа можно так:

```
my $name = $ gilligan_info { 'name' };
my $name = $ { $hash_ref } { 'name' };
```

В этом случае фигурные скобки имеют двойное назначение. Первая пара скобок обозначает выражение, которое возвращает ссылку, а вторая определяет ключ, значение которого должно быть получено.

Для выполнения действий над всем хешем можно записать:

```
my @keys = keys % gilligan_info;
my @keys = keys % { $hash_ref };
```

[1] Эта проблема неоднократно обсуждалась разработчиками Perl, но никто из них не смог придумать синтаксис, который был бы обратно совместим с универсальной формой записи.

Как и в случае ссылок на массивы, при некоторых обстоятельствах мы можем использовать сокращенную форму записи, опустив фигурные скобки. Например, если элемент в фигурных скобках является скалярной величиной (как показано в этих примерах), фигурные скобки можно опустить:

```
my $name = $$hash_ref{'name'};
my @keys = keys %$hash_ref;
```

Как и в случае с массивами, при обращении к конкретным элементам хеша допускается применять оператор стрелки:

```
my $name = $hash_ref->{'name'};
```

Поскольку ссылки допускаются везде, где допускаются скалярные переменные, мы можем создать массив ссылок на хеши:

```
my %gilligan_info = (
    name => 'Джиллиган',
    hat => 'Белый',
    shirt => 'Красный',
    position => 'Первый помощник капитана',
);
my %skipper_info = (
    name => 'Шкипер',
    hat => 'Черный',
    shirt => 'Голубой',
    position => 'Капитан',
);
my @crew = (\%gilligan_info, \%skipper_info);
```

Здесь элемент `$crew[0]` представляет ссылку на хеш с информацией о Джиллигане. Имя Джиллигана в этом случае можно получить любым из следующих способов:

```
${ $crew[0] } { 'name' }
my $ref = $crew[0]; $$ref{'name'}
$crew[0]->{'name'}
$crew[0]{'name'}
```

В последней строке мы просто опустили оператор стрелки «между индексами», хотя левая часть выражения – это массив, а правая – элемент хеша.

Попробуем вывести список членов экипажа:

```
my %gilligan_info = (
    name => 'Джиллиган',
    hat => 'Белый',
    shirt => 'Красный',
    position => 'Первый помощник капитана',
);
my %skipper_info = (
    name => 'Шкипер',
```

```
    hat => 'Черный',
    shirt => 'Голубой',
    position => 'Капитан',
);
my @crew = (\%gilligan_info, \%skipper_info);

my $format = "%-15s %-7s %-7s %-15s\n";
printf $format, qw(Name Shirt Hat Position);
for my $crewmember (@crew) {
    printf $format,
        $crewmember->{'name'},
        $crewmember->{'shirt'},
        $crewmember->{'hat'},
        $crewmember->{'position'};
}
```

Последняя часть выглядит не очень красиво из-за повторяющихся элементов. Мы можем сократить объем кода, использовав синтаксис обращения к части хеша. Универсальная форма записи имеет следующий вид:

```
@gilligan_info { qw(name position) }
```

Обращение к хешу по ссылке будет выглядеть следующим образом:

```
@{ $hash_ref } { qw(name position) }
```

Мы можем отбросить первую пару фигурных скобок, поскольку внутри них находится обычная скалярная величина:

```
@$hash_ref { qw(name position) }
```

Теперь заключительный цикл можно записать в следующей форме:

```
for my $crewmember (@crew) {
    printf $format, @$crewmember{qw(name shirt hat position)};
}
```

Для доступа к части массива или хеша по ссылке с помощью оператора стрелки (`->`) краткая форма записи отсутствует, точно так же как она отсутствует для доступа ко всему массиву или хешу.

При выводе ссылки на хеш она будет выглядеть как `HASH(0x1a2b3c)`, демонстрируя шестнадцатеричный адрес хеша в памяти. Эта информация совершенно бессмысленна для пользователя и может пригодиться только программисту как признак отсутствия операции разыменования.

Упражнения

Ответы на эти вопросы вы найдете в приложении А в разделе «Ответы к главе 4» .

Упражнение 1 [5 мин]

К скольким различным элементам обращаются следующие строки?

```
$ginger->[2][1]
${$ginger[2]}[1]
$ginger->[2]->[1]
${$ginger->[2]}[1]
```

Упражнение 2 [30 мин]

Опираясь на последнюю версию подпрограммы `check_required_items`, напишите подпрограмму `check_items_for_all`, которая принимала бы ссылку на хеш в качестве единственного аргумента и выводила бы список пассажиров, взошедших на борт «Minnow», и список предметов, поднятых ими на борт.

Хеш ссылок можно сконструировать примерно следующим образом:

```
my @gilligan = ... вещи, принадлежащие Джиллигану ...;
my @skipper = ... вещи, принадлежащие Шкиперу ...;
my @professor = ... вещи, принадлежащие Профессору ...;
my %all = (
    Gilligan => \@gilligan,
    Skipper => \@skipper,
    Professor => \@professor,
);
check_items_for_all(\%all);
```

Созданная подпрограмма должна обращаться к `check_required_items` для проверки каждого из пассажиров и членов экипажа и дополнения списка необходимыми предметами.

7

Ссылки на подпрограммы

До сих пор мы рассматривали ссылки на три основных типа данных языка Perl – скаляры, массивы и хеши. Но есть возможность получать ссылки и на подпрограммы.

Для чего это нужно? Скажем так: ссылки на массивы позволяют выполнять однотипные действия над разными массивами в разное время, а ссылки на подпрограммы позволяют из одного и того же программного кода вызывать разные подпрограммы в разное время. Кроме того, ссылки позволяют конструировать очень сложные структуры данных. Ссылки на подпрограммы дают возможность сделать подпрограммы частью этих сложных структур данных.

Иначе говоря, переменные или структуры данных являются хранилищами информации в программе, а ссылки на подпрограммы аналогичным образом можно назвать хранилищами поведенческих реакций программы. Примеры из этого раздела помогут вам разобраться во всех этих хитросплетениях.

Ссылки на именованные подпрограммы

Между Шкипером и Джиллиганом состоялся следующий диалог:

```
sub skipper_greets {
    my $person = shift;
    print "Шкипер: Эй, $person!\n";
}

sub gilligan_greets {
    my $person = shift;
    if ($person eq "Шкипер") {
        print "Джиллиган: Сэр, $person!\n";
      } else {
        print "Джиллиган: Привет, $person!\n";
```

```
    }
}

skipper_greets("Джиллиган");
gilligan_greets("Шкипер");
```

В результате получилось следующее:

```
Шкипер: Эй, Джиллиган!
Джиллиган: Сэр, Шкипер!
```

Вроде бы ничего необычного, однако обратите внимание: Джиллиган ведет себя по-разному в зависимости от того, кто к нему обратился, Шкипер или кто-либо другой.

Теперь представьте, что в хижину входит Профессор и оба члена экипажа приветствуют его:

```
skipper_greets('Профессор');
gilligan_greets('Профессор');
```

В результате получится следующее:

```
Шкипер: Эй, Профессор!
Джиллиган: Привет, Профессор!
```

Профессор должен как-то ответить на приветствие:

```
sub professor_greets {
    my $person = shift;
    print "Профессор: Насколько я понимаю, вы $person!\n";
}

professor_greets('Джиллиган');
professor_greets('Шкипер');
```

В результате ответ прозвучит следующим образом:

```
Профессор: Насколько я понимаю, вы Джиллиган!
Профессор: Насколько я понимаю, вы Шкипер!
```

Гмм! Не очень удобно. Когда в хижину входит новый персонаж, поведение которого описывается отдельной именованной подпрограммой, нам придется точно указывать, какую подпрограмму следует вызвать. Конечно, это можно сделать, пусть и за счет большого объема программного кода, сложного для сопровождения, но мы можем существенно упростить задачу, лишь чуть-чуть добавив косвенности, как мы делали это при работе с массивами и хешами.

Прежде всего, воспользуемся оператором взятия ссылки. Мы не будем давать дополнительных пояснений, поскольку это тот же обратный слэш, с которым вы уже знакомы:

```
my $ref_to_greeter = \&skipper_greets;
```

Здесь мы взяли ссылку на подпрограмму `skipper_greets()`. Обратите внимание на предшествующий символ амперсанда – его наличие со-

вершенно необходимо, так же как и отсутствие круглых скобок. Значение ссылки сохраняется в переменной `$ref_to_greeter`, которая может использоваться везде, где допускается использование обычной скалярной величины.

Есть только одна причина, чтобы вернуться к оригинальной подпрограмме при разыменовании ссылки, – это вызвать ее. Разыменование ссылки на подпрограмму напоминает процедуру разыменования любых других ссылок. Для начала рассмотрим способ, которым мы воспользовались бы, ничего не зная о ссылках (включая необязательный символ амперсанда):

```
& skipper_greets ( 'Джиллиган' )
```

Теперь заменим имя подпрограммы на имя ссылки, окруженное фигурными скобками:

```
& { $ref_to_greeter } ( 'Джиллиган' )
```

Эта конструкция вызывает подпрограмму, на которую в настоящий момент ссылается `$ref_to_greeter`, и передает ей единственный аргумент: имя Джиллиган.

Однако такая форма записи выглядит несколько уродливо, вам не кажется? К счастью, здесь применимы те же правила упрощения, что и для обычных ссылок. Если в фигурных скобках стоит обычная скалярная переменная, их можно просто опустить:

```
& $ref_to_greeter ( 'Джиллиган' )
```

Можно даже воспользоваться оператором стрелки:

```
$ref_to_greeter -> ( 'Джиллиган' )
```

Последняя форма записи особенно удобна, когда ссылка находится внутри большой структуры данных, в чем вы вскоре сможете убедиться.

Чтобы Джиллиган и Шкипер смогли поприветствовать Профессора, нам достаточно вызвать все подпрограммы в цикле:

```
for my $greet (\&skipper_greets, \&gilligan_greets) {
    $greet->('Профессор');
}
```

В первую очередь здесь создается список из двух элементов, каждый из которых является ссылкой на подпрограмму. После этого каждая ссылка разыменовывается, в результате чего происходит вызов той или иной подпрограммы, которым в виде аргумента передается строка `Профессор`.

Мы уже видели ссылки на подпрограммы в виде скалярной переменной и в виде элемента списка. А возможно ли поместить ссылку на подпрограмму в сложную структуру данных? Разумеется. Попробуем

создать таблицу, в которой определим соответствие между именами персонажей и их реакцией на приветствие других людей, а затем перепишем предыдущий пример, основываясь на этой таблице:

```
sub skipper_greets {
    my $person = shift;
    print "Шкипер: Эй, $person!\n";
}

sub gilligan_greets {
    my $person = shift;
    if ($person eq 'Шкипер') {
        print "Джиллиган: Сэр, $person!\n";
      } else {
        print "Джиллиган: Привет, $person!\n";
    }
}

sub professor_greets {
    my $person = shift;
    print "Профессор: Насколько я понимаю, вы $person!\n";
}

my %greets = (
    Gilligan  => \&gilligan_greets,
    Skipper  => \&skipper_greets,
    Professor => \&professor_greets,
);

for my $person (qw(Skipper Gilligan)) {
    $greets{$person}->('Профессор');
}
```

Обратите внимание: в переменной `$person` находится имя персонажа, по которому производится поиск требуемой ссылки на подпрограмму в хеше. После того как ссылка будет получена, мы разыменовываем ее и передаем подпрограмме имя персонажа, которого хотим поприветствовать. В результате мы получаем корректное поведение:

```
Шкипер: Эй, Профессор!
Джиллиган: Привет, Профессор!
```

Теперь попробуем реализовать ситуацию, когда все присутствующие приветствуют друг друга:

```
sub skipper_greets {
    my $person = shift;
    print "Шкипер: Эй, $person!\n";
}

sub gilligan_greets {
    my $person = shift;
    if ($person eq 'Шкипер') {
        print "Джиллиган: Сэр, $person!\n";
```

```
      } else {
        print "Джиллиган: Привет, $person!\n";
    }
}

sub professor_greets {
    my $person = shift;
    print "Профессор: Насколько я понимаю, вы $person!\n";
}

my %greets = (
    Gilligan  => \&gilligan_greets,
    Skipper  => \&skipper_greets,
    Professor  => \&professor_greets,
);

my @everyone = sort keys %greets;
for my $greeter (@everyone) {
    for my $greeted (@everyone) {
        $greets{$greeter}->($greeted)
           unless $greeter eq $greeted; # не стоит приветствовать себя самого
    }
}
```

Результат работы этой программы выглядит следующим образом:

```
Джиллиган: Привет, Профессор!
Джиллиган: Сэр, Шкипер!
Профессор: Насколько я понимаю, вы Джиллиган!
Профессор: Насколько я понимаю, вы Шкипер!
Шкипер: Эй, Джиллиган!
Шкипер: Эй, Профессор!
```

Что-то слишком сложно. Давайте позволим им входить в хижину по одному:

```
sub skipper_greets {
    my $person = shift;
    print "Шкипер: Эй, $person!\n";
}

sub gilligan_greets {
    my $person = shift;
    if ($person eq 'Шкипер') {
        print "Джиллиган: Сэр, $person!\n";
      } else {
        print "Джиллиган: Привет, $person!\n";
    }
}

sub professor_greets {
    my $person = shift;
    print "Профессор: Насколько я понимаю, вы $person!\n";
}
```

```
my %greets = (
    «Джиллиган» => \&gilligan_greets,
    «Шкипер» => \&skipper_greets,
    «Профессор» => \&professor_greets,
);

my @room; # сначала в хижине никого нет
for my $person (qw(Джиллиган Шкипер Профессор)) {
    print "\n";
    print "В хижину входит $person.\n";
    for my $room_person (@room) {
        $greets{$person}->($room_person); # приветствует
        $greets{$room_person}->($person); # получает ответ
    }
    push @room, $person; # входит и устраивается поудобнее
}
```

Так начинается обычный день на тропическом острове:

```
В хижину входит Джиллиган.

В хижину входит Шкипер.
Шкипер: Эй, Джиллиган!
Джиллиган: Сэр, Шкипер!

В хижину входит Профессор.
Профессор: Насколько я понимаю, вы Джиллиган!
Джиллиган: Привет, Профессор!
Профессор: Насколько я понимаю, вы Шкипер!
Шкипер: Эй, Профессор!
```

Анонимные подпрограммы

В последнем примере мы нигде явно[1] не обращались к таким подпрограммам, как `professor_greets()` или `skipper_greets()`. Мы вызывали их косвенно – через ссылки на подпрограммы. Однако нам пришлось приложить определенные усилия, чтобы придумать имена подпрограммам и инициализировать структуру данных. Но мы могли бы создать анонимные подпрограммы точно так же, как анонимные массивы или хеши!

Добавим в компанию обитателей острова еще один персонаж, Джинджер, и определим ее поведение с помощью анонимной подпрограммы:

```
my $ginger = sub {
    my $person = shift;
    print "Джинджер: (томным голосом) О! Салют, $person!\n";
};
$ginger->('Шкипер');
```

[1] По имени. – *Примеч. науч. ред.*

Анонимная подпрограмма объявляется так же, как и обычная, но между ключевым словом sub и открывающей скобкой блока программного кода нет имени. Кроме того, такая форма записи расценивается как часть выражения, поэтому объявление анонимной подпрограммы должно завершаться точкой с запятой или иным разделителем выражений, как и в большинстве случаев.

```
sub { ... тело подпрограммы ... };
```

Значением переменной $ginger является ссылка на подпрограмму, как если бы мы объявили обычную подпрограмму, а потом взяли бы ссылку на нее. Дойдя до последнего выражения, мы увидим такое приветствие:

```
Джинджер: (томным голосом) О! Салют, Шкипер!
```

Сейчас мы сохранили ссылку на подпрограмму в скалярной переменной, но мы можем поместить объявления sub { ... } прямо в хеш:

```
my %greets = (

    'Шкипер' => sub {
        my $person = shift;
        print "Шкипер: Эй, $person!\n";
    },

    'Джиллиган' => sub {
        my $person = shift;
        if ($person eq 'Шкипер') {
            print "Джиллиган: Сэр, $person!\n";
          } else {
            print "Джиллиган: Привет, $person!\n";
        }
    },

    'Профессор' => sub {
        my $person = shift;
        print "Профессор: Насколько я понимаю, вы $person!\n";
    },

    'Джинджер' => sub {
        my $person = shift;
        print "Джинджер: (томным голосом) О! Салют, $person!\n";
    },
);

my @room; # сначала в хижине никого нет
for my $person (qw(Джиллиган Шкипер Профессор Джинджер)) {
    print "\n";
    print "В комнату входит $person.\n";
    for my $room_person (@room) {
        $greets{$person}->($room_person); # приветствует
        $greets{$room_person}->($person); # получает ответ
    }
    push @room, $person; # входит и устраивается поудобнее
}
```

Обратите внимание, насколько проще стал выглядеть программный код. Объявления подпрограмм находятся прямо в структуре данных. Результат работы программы вполне предсказуем:

```
В комнату входит Джиллиган.

В комнату входит Шкипер.
Шкипер: Эй, Джиллиган!
Джиллиган: Сэр, Шкипер!

В комнату входит Профессор.
Профессор: Насколько я понимаю, вы Джиллиган!
Джилиган: Привет, Профессор!
Профессор: Насколько я понимаю, вы Шкипер!
Шкипер: Эй, Профессор!

В комнату входит Джинджер.
Джинджер: (томным голосом) О! Здравствуйте, Джиллиган!
Джилиган: Привет, Джинджер!
Джинджер: (томным голосом) О! Здравствуйте, Шкипер!
Шкипер: Эй, Джинджер!
Джинджер: (томным голосом) О! Здравствуйте, Профессор!
Профессор: Насколько я понимаю, вы Джинджер!
```

Чтобы добавить еще один персонаж, достаточно вставить в хеш еще одно объявление подпрограммы, определяющей его поведение, и занести имя персонажа в список лиц, входящих в хижину. Такую масштабируемость мы получили потому, что определили поведенческие реакции персонажей как обычные данные, которые можно обойти и выбрать в цикле благодаря ссылкам на подпрограммы.

Подпрограммы обратного вызова

Очень часто ссылки на подпрограммы служат для организации *обратных вызовов*. Подпрограмма обратного вызова определяет, что необходимо сделать, когда программа достигает определенной точки в алгоритме.

Например, модуль `File::Find` экспортирует подпрограмму `find`, которая выполняет обход дерева каталогов способом, не зависящим от платформы. В простейшем случае подпрограмме `find` передаются два аргумента: начальный каталог и описание того, «что следует сделать» с именем каждого файла или каталога, которые будут обнаружены внутри заданного. В данном случае под «что сделать» подразумевается ссылка на подпрограмму:

```
use File::Find;
sub what_to_do {
    print "Найден файл или каталог $File::Find::name\n";
}
my @starting_directories = qw(.);

find(\&what_to_do, @starting_directories);
```

В этом примере поиск начинается от текущего каталога (.), в процессе которого отыскиваются все находящиеся внутри него файлы и подкаталоги. Для каждого найденного элемента вызывается подпрограмма what_to_do(), которой через глобальные переменные передаются несколько аргументов. В данном случае в переменной File::Find::name содержится полный путь к найденному файлу или каталогу (начиная от исходного каталога).

В этом примере мы передали подпрограмме find в качестве аргументов список каталогов поиска и реакцию на найденный элемент.

Не имеет большого смысла придумывать название для подпрограммы, которая вызывается лишь в одном месте, поэтому перепишем предыдущий пример, оставив подпрограмму анонимной:

```
use File::Find;
my @starting_directories = qw(.);

find(
    sub {
        print "Найден файл или каталог $File::Find::name\n";
    },
    @starting_directories,
);
```

Замыкания

Модуль File::Find позволяет получать дополнительные сведения о файлах, например их размеры. Для удобства разработки подпрограмм обратного вызова имя элемента, найденного в текущем рабочем каталоге, содержится в переменной $_.

Вы могли заметить, что в предыдущем примере мы извлекали имя элемента из переменной $File::Find::name. Как же определить, где находится настоящее имя элемента: в $_ или в $File::Find::name? В переменной $File::Find::name содержится полное имя найденного элемента относительно начального каталога поиска. Когда производится обращение к подпрограмме обратного вызова, рабочим каталогом считается тот, в котором обнаружен очередной элемент. Предположим, что требуется отыскать файлы в текущем рабочем каталоге, поэтому в качестве начального каталога мы задаем его имя ("."). Если в момент вызова подпрограммы find текущим был каталог /usr, она начнет поиск файлов в этом и всех вложенных подкаталогах этого каталога. Когда find найдет файл /usr/bin/perl, текущим рабочим каталогом (в момент обращения к подпрограмме обратного вызова) будет /usr/bin. В этом случае в переменной $_ будет храниться имя perl, а в переменной $File::Find::name — ./bin/perl, то есть путь к файлу относительно начального каталога поиска.

Все это означает, что проверки, выполняемые над файлом, такие как -s, автоматически относятся к только что найденному элементу. Это

удобно, однако имя текущего каталога внутри подпрограммы обратного вызова отличается от начального каталога поиска.

Представьте, что нам надо с помощью модуля File::Find определить суммарный размер всех файлов, которые будут найдены. Процедура обратного вызова не может принимать входных аргументов, а вызывающая сторона – значение, возвращаемое подпрограммой. Но это не имеет никакого значения. После разыменования ссылки подпрограмма получает доступ ко всем видимым переменным. Например:

```
use File::Find;

my $total_size = 0;
find(sub { $total_size += -s if -f }, '.');
print $total_size, "\n";
```

Как и прежде, мы вызываем подпрограмму find и передаем ей два аргумента: ссылку на анонимную подпрограмму и имя начального каталога. При обнаружении файлов внутри этого каталога (и всех вложенных подкаталогах) она вызывает анонимную подпрограмму.

Обратите внимание: подпрограмма обращается к переменной $total_size. Мы объявили эту переменную за пределами области видимости подпрограммы find, однако она доступна в подпрограмме обратного вызова. Таким образом, даже при том, что подпрограмма find (которая не имеет прямого доступа к переменной $total_size) обращается к подпрограмме обратного вызова, последняя способна обращаться к переменной и изменять ее значение.

Подобные подпрограммы, которые имеют доступ ко всем переменным, существовавшим на момент объявления подпрограммы, называются *замыканиями* (термин заимствован из математики). В терминах языка Perl замыкание – это подпрограмма, которая может обращаться к лексическим переменным, расположенным вне области видимости данной подпрограммы.

Кроме того, возможность доступа к переменной изнутри замыкания гарантирует, что эта переменная будет существовать, по крайней мере до тех пор, пока существует ссылка на подпрограмму-замыкание. Попробуем подсчитать количество файлов:[1]

```
use File::Find;

my $callback;
{
    my $count = 0;
```

[1] На первый взгляд в этом отрывке имеется лишняя точка с запятой – в конце строки, где выполняется присваивание переменной $callback, не так ли? Постарайтесь запомнить: конструкция sub{...} – это выражение. Значение этого выражения (ссылка на подпрограмму) присваивается переменной $callback, поэтому в конце выражения должна стоять точка с запятой.

```
    $callback = sub { print ++$count, ": $File::Find::name\n" };
}
find($callback, '.');
```

Здесь объявляется переменная, которая будет хранить ссылку на подпрограмму обратного вызова. Эту переменную нельзя объявлять в пределах блока кода, потому что тогда Perl уничтожит ее после выхода из блока. Затем переменная `$count` инициализируется значением 0. После этого следует объявление анонимной подпрограммы, ссылка на которую записывается в переменную `$callback`. Данная подпрограмма представляет собой замыкание, поскольку она обращается к лексической переменной `$count`.

По выходе из блока кода переменная `$count` исчезает из области видимости. Однако, поскольку к ней производится обращение из подпрограммы, ссылка на которую продолжает оставаться в переменной `$callback`, переменная `$count` продолжает существовать в памяти как анонимная скалярная переменная.[1] При каждом вызове подпрограмма увеличивает значение переменной `$count`, получая в результате числа 1, 2, 3 и так далее.

Подпрограмма как возвращаемое значение другой подпрограммы

Блок кода прекрасно подходит для объявления подпрограммы обратного вызова, но гораздо удобнее иметь подпрограмму, которая возвращала бы ссылку на другую подпрограмму!

```
use File::Find;

sub create_find_callback_that_counts {
    my $count = 0;
    return sub { print ++$count, ": $File::Find::name\n" };
}

my $callback = create_find_callback_that_counts( );
find($callback, '.');
```

Здесь мы имеем практически то же, что и раньше, только оформили все несколько иначе. Подпрограмма `create_find_callback_that_counts()` инициализирует лексическую переменную `$count` значением 0 и возвращает ссылку на анонимную подпрограмму – тоже замыкание, поскольку

[1] Если точнее, то объявление замыкания увеличивает счетчик ссылок, как если бы на эту переменную явно была взята еще одна ссылка. Как раз перед концом блока счетчик ссылок на переменную `$count` имеет значение 2, а по завершении блока счетчик ссылок приобретает значение 1. Больше ниоткуда в программе нельзя обратиться к переменной `$count`, но она будет храниться в памяти до тех пор, пока будет существовать ссылка на подпрограмму в переменной `$callback` или где-либо еще.

ей доступна переменная $count. Эта переменная будет существовать в памяти даже после выхода из подпрограммы create_find_callback_that_counts() до тех пор, пока не исчезнет ссылка на анонимную подпрограмму.

При повторном обращении к той же подпрограмме обратного вызова переменная сохранит свое прежнее значение, поскольку переменная инициализируется в подпрограмме create_find_callback_that_counts(), а не в анонимной подпрограмме:

```
use File::Find;

sub create_find_callback_that_counts {
    my $count = 0;
    return sub { print ++$count, ": $File::Find::name\n" };
}

my $callback = create_find_callback_that_counts( );
print "мой каталог bin:\n";
find($callback, 'bin');
print "мой каталог lib:\n";
find($callback, 'lib');
```

Данный пример выведет последовательность чисел, начиная с 1, для всех файлов, которые будут найдены в каталоге bin, и затем продолжит эту последовательность, когда поиск будет производиться уже в каталоге lib. В обоих случаях задействуется одна и та же переменная, $count. Но если подпрограмма create_find_callback_that_counts() будет вызвана дважды, то мы получим две разных переменных $count:

```
use File::Find;

sub create_find_callback_that_counts {
    my $count = 0;
    return sub { print ++$count, ": $File::Find::name\n" };
}

my $callback1 = create_find_callback_that_counts( );
my $callback2 = create_find_callback_that_counts( );
print "мой каталог bin:\n";
find($callback1, 'bin');
print "мой каталог lib:\n";
find($callback2, 'lib');
```

В этом случае у нас появятся две разные переменные $count, каждая из которых будет доступна только из своей подпрограммы обратного вызова.

А как получить общее число файлов, подсчитанное подпрограммой обратного вызова? Ранее мы делали это с помощью глобальной переменной $total_size. Если объявить переменную $total_size в подпрограмме, которая возвращает ссылку на подпрограмму обратного вызова, мы не получим доступа к этой переменной. Но мы можем позволить себе пойти на небольшую хитрость. Мы можем определить, что в случае отсутствия входных аргументов подпрограмма обратного вызова

не должна ничего возвращать, но если она получает входной аргумент, то должна возвратить суммарный размер:

```
use File::Find;

sub create_find_callback_that_sums_the_size {
    my $total_size = 0;
    return sub {
        if (@_) { # это наш фиктивный запрос
            return $total_size;
        } else { # а это вызов из File::Find:
            $total_size += -s if -f;
        }
    };
}

my $callback = create_find_callback_that_sums_the_size( );
find($callback, 'bin');
my $total_size = $callback->('dummy'); # передача фиктивного аргумента,
                                       # чтобы получить суммарный размер
print "суммарный размер файлов в каталоге bin = $total_size\n";
```

Определение различной реакции на наличие и отсутствие входного аргумента – это далеко не универсальное решение. К счастью, мы можем вернуть более чем одну ссылку на подпрограмму из `create_find_callback_that_counts()`:

```
use File::Find;

sub create_find_callbacks_that_sum_the_size {
   my $total_size = 0;
   return(sub { $total_size += -s if -f }, sub { return $total_size });
}

my ($count_em, $get_results) = create_find_callbacks_that_sum_the_size( );
find($count_em, 'bin');
my $total_size = &$get_results( );
print " суммарный размер файлов в каталоге bin = $total_size\n";
```

Поскольку обе ссылки на подпрограммы были созданы в одной и той же области видимости, они обе имеют доступ к той же самой переменной `$total_size`. Даже при том что перед вызовом любой из этих подпрограмм переменная будет находиться за пределами области видимости, тем не менее обе подпрограммы будут иметь доступ к одной и той же переменной и могут быть использованы для получения результатов вычислений.

Возврат двух ссылок на подпрограммы не приводит к их вызову. Ссылки – это лишь данные, идентифицирующие точки вызова. Они будут исполняться, только когда вызываются как подпрограммы обратного вызова или в результате разыменования ссылок.

А что произойдет, если эта новая подпрограмма будет вызвана более чем один раз?

```
use File::Find;

sub create_find_callbacks_that_sum_the_size {
    my $total_size = 0;
    return(sub { $total_size += -s if -f }, sub { return $total_size });
}

## создание подпрограмм
my %subs;
foreach my $dir (qw(bin lib man)) {
    my ($callback, $getter) = create_find_callbacks_that_sum_the_size( );
    $subs{$dir}{CALLBACK} = $callback;
    $subs{$dir}{GETTER} = $getter;
}

## собрать данные
for (keys %subs) {
    find($subs{$_}{CALLBACK}, $_);
}

## вывести результаты
for (sort keys %subs) {
    my $sum = $subs{$_}{GETTER}->( );
    print "суммарный размер файлов в каталоге $_ = $sum bytes\n";
}
```

В разделе, где создаются подпрограммы, мы создали три экземпляра пар подпрограмм обратного вызова и получения результатов. Каждой подпрограмме обратного вызова соответствует подпрограмма получения результата. В следующем разделе выполняется сбор данных, здесь подпрограмма `find` вызывается три раза, причем каждый раз с другой ссылкой на подпрограмму обратного вызова. Благодаря этому результаты вычислений всякий раз записываются в свою переменную `$total_size`. В последнем разделе с помощью подпрограмм чтения производится вывод полученных результатов.

Каждая из шести подпрограмм (и каждая из трех переменных `$total_size`) имеет свой счетчик ссылок. Если мы попытаемся изменить содержимое хеша `%subs` или он выйдет из области видимости, это приведет к уменьшению счетчиков ссылок и утилизации занимаемой памяти. (Если эти элементы в свою очередь так же ссылаются на какие-либо другие данные, то их счетчики ссылок так же будут уменьшены соответствующим образом.)

Использование переменных замыканий для ввода данных

В предыдущих примерах было показано, как можно изменять переменные замыканий, но эти же переменные могут служить для передачи входной информации в замыкания. Попробуем написать подпрограмму, создающую подпрограмму обратного вызова для работы с мо-

дулем File::Find, которая в свою очередь будет выводить имена файлов с размером, превышающим некое значение:

```
use File::Find;

sub print_bigger_than {
    my $minimum_size = shift;
    return sub { print "$File::Find::name\n" if -f and -s >= $minimum_size };
}

my $bigger_than_1024 = print_bigger_than(1024);
find($bigger_than_1024, 'bin');
```

Здесь мы передаем подпрограмме print_bigger_than число 1024, которое затем переписывается в лексическую переменную $minimum_size. К этой переменной мы обращаемся из подпрограммы, на которую ссылается возвращаемое значение подпрограммы print_bigger_than, поэтому она становится переменной замыкания, значение которой сохраняется на протяжении всего жизненного цикла ссылки на подпрограмму. Как и прежде, каждый новый вызов подпрограммы print_bigger_than приводит к созданию нового экземпляра переменной $minimum_size, связанной с соответствующей ей ссылкой на подпрограмму.

Замыкания «замыкаются» только на лексических переменных, т. к. лексические переменные рано или поздно выйдут из области видимости. Поскольку глобальные переменные никогда не выйдут из области видимости, замыкания никогда не смогут замкнуться на них. Все подпрограммы всегда будут обращаться к одному и тому же экземпляру глобальной переменной.

Переменные замыканий как статические локальные переменные

Чтобы стать замыканием, подпрограмма не обязательно должна быть анонимной. Если именованная подпрограмма обращается к лексической переменной и эта переменная выйдет за пределы области видимости, подпрограмма сохранит возможность доступа ко всем своим лексическим переменным точно так же, как это происходит в случае анонимных подпрограмм. Рассмотрим две подпрограммы, которые ведут подсчет кокосовых орехов, собранных Джиллиганом:

```
{
    my $count;
    sub count_one { ++$count }
    sub count_so_far { return $count }
}
```

Если поместить этот отрывок в начало программы, то в результате будет создана переменная $count с областью видимости, ограниченной блоком кода, а две подпрограммы, которые ссылаются на эту переменную, превратятся в замыкания. Однако подпрограммы имеют опреде-

ленные имена, поэтому они продолжат свое существование и за пределами блока (как и любые другие именованные подпрограммы). А раз подпрограммы сохраняют возможность доступа к переменной даже за пределами области ее видимости, то они превращаются в замыкания и могут продолжать обращаться к переменной на протяжении всего жизненного цикла программы.

Убедиться в справедливости этого утверждения можно, выполнив несколько обращений к подпрограммам:

```
count_one( );
count_one( );
count_one( );
print 'мы собрали ', count_so_far( ), " орехов!\n";
```

Переменная `$count` сохраняет свое значение между вызовами подпрограмм `count_one()` или `count_so_far()`, и никакая другая часть программы не может получить доступ к этой переменной.

В языке C такие переменные известны как *статические локальные переменные*, то есть переменные, которые доступны только узкому кругу функций программы[1] и сохраняют свое значение на протяжении всего времени жизни программы между вызовами функций.

А что если нам потребуется вести счет в обратном порядке? Это можно сделать примерно так:

```
{
    my $countdown = 10;
    sub count_down { $countdown-- }
    sub count_remaining { $countdown }
}

count_down( );
count_down( );
count_down( );
print 'нам осталось собрать ', count_remaining( ), " орехов!\n";
```

Данный прием эффективен только в том случае, если этот блок будет размещаться где-нибудь ближе к началу программы, то есть до того, как будет вызвана функция `count_down()` или `count_remaining()`. Догадываетесь почему?

Прием не сработает, если блок будет размещаться после вызова любой из функций. Причина проста: первая строка этого блока функционально делится на две части:

```
my $countdown = 10;
```

[1] Если быть совсем точными, то *статические локальные переменные* C/C++ доступны внутри только одной функции, в которой они определены; переменные, доступные узкому кругу функций, – это *статические переменные файла*; и те и другие обладают особенностями, на которых акцентирует внимание автор. – *Примеч. науч. ред.*

Первая часть – это объявление лексической переменной `$countdown`. Эта часть обрабатывается на *этапе компиляции*. Вторая часть – это присвоение переменной значения 10. Эта часть обрабатывается уже на *этапе исполнения*. Пока вторая часть строки не пройдет обработку на этапе исполнения, переменная `$countdown` будет иметь значение по умолчанию, а именно `undef`.

В качестве одного из решений этой проблемы можно порекомендовать преобразовать блок со статической переменной в блок `BEGIN`:

```
BEGIN {
    my $countdown = 10;
    sub count_down { $countdown-- }
    sub count_remaining { $countdown }
}
```

Ключевое слово `BEGIN` сообщает компилятору Perl, что после компиляции блока необходимо на время приостановить компиляцию программы и исполнить этот блок. Если в процессе исполнения блока не возникло фатальной ошибки, компиляция программы будет продолжена со строки, стоящей сразу за блоком. Кроме того, вслед за этим сам блок удаляется из программы, благодаря чему гарантируется, что он будет исполнен всего один раз, даже если синтаксически он находится внутри цикла или подпрограммы.

Упражнения

Ответы на эти вопросы вы найдете в приложении A в разделе «Ответы к главе 7».

Упражнение 1 [50 мин]

Профессор обновил некоторые файлы в понедельник днем и, к несчастью, забыл, какие именно. Это случается с ним постоянно. Он хочет, чтобы вы написали подпрограмму с именем `gather_mtime_between`, которой можно было бы передать начальную и конечную дату и время временного интервала и которая возвращала бы две ссылки на подпрограммы. Первая из них должна с помощью модуля `File::Find` отыскать все файлы, которые были изменены в течение указанного интервала, а вторая должна возвращать список найденных файлов.

Ниже приводится пример программного кода, который вы можете опробовать. Он отыскивает только те файлы, которые были изменены в последний понедельник, хотя его нетрудно переориентировать на любой другой день недели. (Весь этот текст не надо вводить вручную, поскольку программа в виде файла с именем `ex6-1.plx` входит в состав архива с примерами к книге, который можно загрузить с веб-сайта издательства O'Reilly.)

Подсказка: время последнего изменения файла (`mtime`) можно определить, например, так:

```
my $timestamp = (stat $file_name)[9];
```

Поскольку это часть массива, круглые скобки обязательны. Не забывайте, что имя рабочего каталога, внутри которого вызывается подпрограмма обратного вызова, не обязательно будет совпадать с именем начального каталога, которое было передано подпрограмме `find`.

```
use File::Find;
use Time::Local;

my $target_dow = 1; # воскресенье = 0, понедельник = 1, ...
my @starting_directories = (".");

my $seconds_per_day = 24 * 60 * 60;
my($sec, $min, $hour, $day, $mon, $yr, $dow) = localtime;

my $start = timelocal(0, 0, 0, $day, $mon, $yr); # сегодня в полночь
while ($dow != $target_dow) {
    # Назад на одни сутки
    $start -= $seconds_per_day; # надеюсь, что не попал на момент перехода
                                # между летним временем и зимним! :-)
    if (--$dow < 0) {
        $dow += 7;
    }
}
my $stop = $start + $seconds_per_day;

my($gather, $yield) = gather_mtime_between($start, $stop);
find($gather, @starting_directories);
my @files = $yield->( );

for my $file (@files) {
    my $mtime = (stat $file)[9]; # получение времени последнего изменения
    my $when = localtime $mtime;
    print "$when: $file\n";
}
```

Обратите внимание на комментарий по поводу перехода между летним временем и зимним. Во многих странах осуществляется переход с летнего времени на зимнее и обратно. В день перевода часов длина суток уже не будет равна 86400 секундам. В данном примере эта проблема не учитывается, однако вы, как более педантичный программист, могли бы принять ее во внимание.

По договору между издательством «Символ-Плюс» и Интернет-магазином «Books.Ru – Книги России» единственный легальный способ получения данного файла с книгой ISBN 5-93286-093-9, название «Perl: изучаем глубже» – покупка в Интернет-магазине «Books.Ru – Книги России». Если Вы получили данный файл каким-либо другим образом, Вы нарушили международное законодательство и законодательство Российской Федерации об охране авторского права. Вам необходимо удалить данный файл, а также сообщить издательству «Символ-Плюс» (piracy@symbol.ru), где именно Вы получили данный файл.