Оглавление

1. Информация о переводе

Книга «Modern Perl» была написана chromatic. Исходный код pod-страниц доступен в репозитории на github. Текст лицензирован под лицензией «Creative Commons Attribution-Noncommercial-Share Alike», что допускает свободное распространение оригинального/переработанного текста под той же лицензией, запрещено лишь использование книги для коммерческих целей.

Перевод книги «Modern Perl» на русский язык выполнен Тимуром Нозадзе. Оригинальный репозиторий на github с исходными pod-файлами находится здесь

Ковертация в pdf выполнена с помощью pandoc, latex-шаблон любезно предоставил Вячеслав Тихановский (vti).

Издание 2013 года.

2. *Предисловие

Современный Perl (*Modern Perl*) — это один из способов описать то, как работают самые эффективные в мире программисты на Perl 5. Они применяют идиомы языка. Они пользуются преимуществами CPAN. Они проявляют хороший вкус и мастерство в написании мощного, поддерживаемого, масштабируемого, лаконичного и эффективного кода. Вы тоже можете освоить эти навыки!

Perl впервые появился в 1987 году как простой инструмент для системного администрирования. Начав с объявления и захвата комфортной ниши между написанием сценариев командной оболочки и программированием на C, он превратился в семейство мощных универсальных языков. Perl 5 имеет солидную историю прагматизма и светлое будущее шлифовки и улучшения_(Perl 6 — это изобретение программирования заново, основанное на прочных принципах Perl, но это тема для другой книги._.

На протяжении долгой истории Perl — особенно 17-ти лет Perl 5 — наше понимание того, что представляет собой хорошая программа на Perl, менялось. Хотя вы можете писать эффективные программы, которые никогда не воспользуются всеми возможностями, предлагаемыми языком, глобальное Perl-сообщество изобрело, позаимствовало, улучшило и отполировало множество идей, и сделало их доступными для любого, кто захочет их изучить.

Запуск Современного Perl

Модуль `Modern::Perl` из CPAN просит Perl предупреждать о сомнительных конструкциях и опечатках и включает новые возможности, представленные в современных релизах Perl 5. Если не упомянуто иное, фрагменты кода всегда будут подразумевать следующий базовый костяк программы:

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

use Modern::Perl 2011;
use autodie;
```

...что эквивалентно следующему:

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

use 5.012;      # подразумевает "use strict;"
use warnings;
use autodie;
```

Некоторые примеры используют функции тестирования, такие как `ok()`, `like()` и `is()` . Эти программы следуют следующему шаблону:

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

use Modern::Perl;
use Test::More;

# здесь код примера

done_testing();
```

На момент написания, текущий стабильный релиз семейства Perl 5 — Perl 5.14. Примеры в этой книге лучше всего работают с версией Perl 5.12.0 или выше. Многие примеры будут работать на более старых версиях Perl 5 с небольшими изменениями, но у вас будет больше сложностей с использованием чего-нибудь старше, чем 5.10.0.

Если Perl 5 у вас не установлен (или установлена старая версия), вы можете сами установить новый релиз. Пользователям Windows нужно скачать Strawberry Perl с сайта http://www.strawberryperl.com/ или ActivePerl с http://www.activestate.com/activeperl. Пользователям других операционных систем, где уже установлен Perl 5 (а также компилятор C и другие инструменты разработчика), стоит начать с установки CPAN-модуля `App::perlbrew`*(См. инструкции по установке на http://search.cpan.org/perldoc?App::perlbrew.).*

`perlbrew` даёт возможность устанавливать и управлять несколькими версиями Perl 5. Это позволяет вам переключаться между разными версиями Perl 5, а также устанавливать Perl 5 и CPAN-модули в домашнюю директорию,

не затрагивая системные версии. Если вам когда-нибудь приходилось выпрашивать у вашего системного администратора разрешение на установку программного обеспечения, вы знаете, насколько такая возможность облегчит вам жизнь.

Благодарности

Написание этой книги не было бы возможным без вопросов, комментариев, предложений, советов, мудрости и поощрения многих, многих людей. Те, кому автор и редактор особенно благодарны:

Джон С. Дж. Андерсон (John SJ Anderson), Питер Аронофф (Peter Aronoff), Ли Эйлуорд (Lee Aylward), Алекс Балхатчет (Alex Balhatchet), Эвар Арнфьорд Бьярмасон (Ævar Arnfjörð Bjarmason), Маттиас Блох (Matthias Bloch), Джон Бокма (John Bokma), Василий Чекалкин (Vasily Chekalkin), Дмитрий Честных (Dmitry Chestnykh), Е. Короба (E. Choroba), Том Кристиансен (Tom Christiansen), Аннели Касс (Anneli Cuss), Пауло Кустодио (Paulo Custodio), Стив Дикинсон (Steve Dickinson), Курт Эдмистон (Kurt Edmiston), Фелипе (Felipe), Шломи Фиш (Shlomi Fish), Джеремая Фостер (Jeremiah Foster), Марк Фаулер (Mark Fowler), Джон Габриеле (John Gabriele), Эндрю Грангаард (Andrew Grangaard), Брюс Грей (Bruce Gray), Аск Бьорн Хансен (Ask Bjørn Hansen), Тим Хини (Tim Heaney), Грэми Хьюсон (Graeme Hewson), Роберт Хикс (Robert Hicks), Майкл Хайнд (Michael Hind), Марк Хайндесс (Mark Hindess), Яри Хлукан (Yary Hluchan), Дэниел Холц (Daniel Holz), Майк Хаффман (Mike Huffman), Гари Х. Джонс II (Gary H. Jones II), Кёртис Джуэлл (Curtis Jewell), Мохаммед Арафат Камаал (Mohammed Arafat Kamaal), Джеймс Е Кинан (James E Keenan), Кирк Киммел (Kirk Kimmel), Ювал Когман (Yuval Kogman), Ян Крыницки (Jan Krynicky), Майкл Ланг (Michael Lang), Джефф Лавалли (Jeff Lavallee), Мориц Ленц (Moritz Lenz), Энди Лестер (Andy Lester), Жан-Баптист Мазон (Jean-Baptiste Mazon), Джош МакАдамс (Josh McAdams), Гарет МакКахан (Gareth McCaughan), Джон МакНамара (John McNamara), Шон М Мур (Shawn M Moore), Алекс Мунтада (Alex Muntada), Карл Масак (Carl Mäsak), Крис Нисвандер (Chris Niswander), Нело Онья (Nelo Onyiah), Чес. Оуэнс (Chas. Owens), ww с PerlMonks, Джесс Робинсон (Jess Robinson), Дэйв Ролски (Dave Rolsky), Габриелле Рот (Gabrielle Roth), Жан-Пьер Рупп (Jean-Pierre Rupp), Эдуардо Сантьяго (Eduardo Santiago), Эндрю Сэвидж (Andrew Savige), Лорн Шахтер (Lorne Schachter), Стив Шульц (Steve Schulze), Дэн Скотт (Dan Scott), Александр Скотт-Джонс (Alexander Scott-Johns), Филлип Смит (Phillip Smith),

Кристофер E. Стит (Christopher E. Stith), Марк A. Стратман (Mark A. Stratman), Брайан Саммерсетт (Bryan Summersett), Одри Тан (Audrey Tang), Скотт Томсон (Scott Thomson), Бен Тилли (Ben Tilly), Рууд Х. Г. ван Тол (Ruud H. G. van Tol), Сэм Вилейн (Sam Vilain), Ларри Уолл (Larry Wall), Льюис Уолл (Lewis Wall), Колин Везерби (Colin Wetherbee), Фрэнк Виганд (Frank Wiegand), Дуг Уилсон (Doug Wilson), Sawyer X, Дэвид Йинлин (David Yingling), Марко Загозен (Marko Zagozen), harleypig, hbm, и sunnavy.

Любые оставшиеся ошибки — вина упрямого автора.

3. Философия Perl

Perl позволяет добиваться результата — он гибкий, прощающий и пластичный. Умелые программисты используют его каждый день для любых задач — от однострочников и разовых автоматизаций, до многолетних проектов, выполняемых множеством программистов.

Perl прагматичен. Вы — главный. Вы выбираете, как решать ваши проблемы, и Perl действует в соответствии с вашими намерениями, без фрустраций и церемоний.

Perl будет расти вместе с вами. В течение следующего часа вы узнаете достаточно, чтобы писать реальные полезные программы — и вы поймёте, *как* работает язык, и *почему* он работает именно так. Современный Perl применяет это знание и объединённый опыт глобального Perl-сообщества, чтобы помочь вам писать работающий, поддерживаемый код.

Для начала, вы должны понять, как узнать больше.

Perldoc

Perl обладает культурой полезной документации. Утилита `perldoc` — часть любой полной инсталляции Perl 5_(Однако, ваша Unix-подобная система может потребовать установки дополнительного пакета, такого как `perl-doc)` в Debian или Ubuntu GNU/Linux._. `perldoc` отображает документацию для каждого установленного в системе Perl-модуля — будь это базовый модуль или установленный из CPAN (Comprehensive Perl Archive Network, Всеобъемлющая архивная сеть Perl) — а также тысячи страниц обширной встроенной документации Perl.

Используйте `perldoc` для чтения документации по модулю или части базовой документации:

Первый пример отображает документацию, встроенную в модуль `List::Util`. Второй пример отображает отдельный файл документации, в данном случае — оглавление базовой документации. Третий пример отображает отдельный файл документации, являющийся частью CPAN-дистрибутива . `perldoc`

скрывает эти детали; нет никакой разницы между чтением документации по базовой библиотеке, такой как `Data::Dumper`, или по установленой из CPAN.

Стандартный шаблон документации включает описание модуля, демонстрацию примеров использования, и затем — детальное разъяснение модуля и его интерфейса. Объём документации различается у разных авторов, но её форма поразительно последовательна.

Утилита `perldoc` имеет много других возможностей (см. `perldoc perldoc`). Опция `-q` осуществляет поиск указанных ключевых слов по Perl FAQ. Так `perldoc -q sort` возвращает три вопроса: *How do I sort an array by (anything)?* (*Как мне отсортировать массив по (чему бы то ни было)?*), *How do I sort a hash (optionally by value instead of key)?* (*Как мне отсортировать хеш (опционально по значению вместо ключа)?*), и *How can I always keep my hash sorted?* (*Как мне всегда поддерживать мой хеш отсортированным?*).

Опция `-f` выводит документацию по встроенным функциям Perl. `perldoc -f sort` объясняет поведение оператора `sort`. Если вы не знаете имени нужной вам функции, просмотрите список доступных встроенных функций в `perldoc perlfunc`.

Опция `-v` ищет встроенную переменную. Например, `perldoc -v $PID` выводит документацию по переменной, содержащей идентификатор процесса текущей программы. В зависимости от вашей оболочки, вам, возможно, придётся соответствующим образом заключить переменную в кавычки.

Опция `-l` заставляет `perldoc` вывести *путь* к файлу документации вместо его содержимого_(Имейте ввиду, что модуль может иметь отдельный файл `.pod`) в придачу к файлу `.pm`_.

Опция `-m` отображает полное *содержимое* модуля, код и всё остальное, не выполняя никакого специального форматирования.

Система документации в Perl 5 — *POD*, или *Plain Old Documentation* (*Старая добрая документация*). `perldoc perlpod` описывает, как работает POD. В число других инструментов для работы с POD входят `podchecker`, который проверяет корректность формы вашей POD-документации, и `Pod::Webserver`, отображающий локальную POD-документацию в виде

HTML через минимальный веб-сервер.

Выразительность

Изучение Ларри Уоллом (Larry Wall) лингвистики и естественных языков повлияло на дизайн Perl. Язык предоставляет вам потрясающую свободу в решении ваших задач, в зависимости от вашего коллективного стиля, имеющегося времени, ожидаемого срока жизни программы или даже вашего вдохновения. Вы можете писать простой, прямолинейный код, или интегрировать его в большие, чётко структурированные программы. Вы можете выбирать из множества парадигм дизайна, можете избегать или приветствовать использование дополнительных возможностей.

Там, где другие языки навязывают один оптимальный способ написания кода, Perl позволят *вам* решать, какой вариант более читаемый, или полезный, или весёлый.

У Perl-хакеров есть девиз для этого: *TIMTOWTDI*, произносится как «Tim Toady» («Тим Тоуди»), или «There's more than one way to do it!» («Есть более чем один способ сделать это!»).

Хотя эта выразительность позволяет мастерам создавать изумительные программы, она же позволяет неблагоразумным и неосторожным устраивать беспорядок. Опыт и хороший вкус приведут вас к написанию хорошего кода. Выбор за вами — но не забывайте о читаемости и поддерживаемости, особенно для тех, кто придёт после вас.

Новички в Perl зачастую могут найти некоторые конструкции трудными для понимания. Многие из этих идиом предлагают большую (хотя и неочевидную) мощь. Вполне нормально избегать их, пока вы не будете чувствовать себя с ними комфортно.

Изучение Perl подобно изучению нового разговорного языка. Вы изучите несколько слов, составите предложения и вскоре будете наслаждаться простыми беседами. Мастерство приходит с практикой чтения и написания. Вам не обязательно понимать каждую деталь Perl, чтобы быть продуктивным, но принципы, изложенные в этой главе, жизненно важны для вашего роста как программиста.

Ещё одна цель дизайна Perl состоит в том, чтобы пытаться избегать того, что может удивить опытных (Perl) программистов. Например, сложение двух переменных (`$first_num + $second_num`) очевидно числовая операция ; оператор сложения должен воспринимать обе переменные как числовые значения, чтобы получить числовой результат. Независимо от содержимого `$first_num` и `$second_num`, Perl преобразует их в числовые значения . Вы выразили своё намерение работать с ними как с числами, использовав числовой оператор. Perl с удовольствием сделает это.

Адепты Perl часто называют этот принцип *DWIM*, или *do what I mean* (*делай то, что я имею ввиду*). Иначе говоря, Perl следует *принципу наименьшего удивления*. Имея поверхностное понимание Perl (особенно контекста, , должно быть возможно понять смысл незнакомого выражения. Вы разовьёте этот навык.

Выразительность Perl, кроме того, позволяет новичкам писать полезные программы без необходимости понимать всё. Получающийся код часто называют *baby Perl* (*детский Perl*), в том смысле, что почти каждый хочет помочь детям научиться хорошо говорить. Каждый начинает как новичок. Практикуясь и обучаясь у более опытных программистов, вы будете понимать и воспринимать более мощные идиомы и техники.

Например, опытный Perl-хакер может утроить список чисел таким образом:

```
my @tripled = map { $_ * 3 } @numbers;
```

...а знаток Perl может написать так:

```
my @tripled;

for my $num (@numbers)
{
    push @tripled, $num * 3;
}
```

...тогда как новичок попытается сделать следующее:

```
my @tripled;

for (my $i = 0; $i < scalar @numbers; $i++)
{
    $tripled[$i] = $numbers[$i] * 3;
}
```

Все три подхода делают одно и то же, но каждый использует Perl по-своему.

Опыт в программировании на Perl поможет вам фокусироваться на том, *что* вам нужно, а не на том, *как* это сделать. В любом случае, Perl с удовольствием будет выполнять простые программы. Вы можете разрабатывать и усовершенствовать ваши программы для повышения ясности, выразительности, пригодности к повторному использованию и поддерживаемости, частично или целиком. Пользуйтесь преимуществами этой гибкости и прагматизма: намного лучше выполнить вашу задачу эффективно сейчас, чем написать концептуально чистую и красивую программу в следующем году.

Контекст

В разговорных языка значение слова или фразы может зависеть от того, как вы их используете; локальный *контекст* помогает прояснить намерение. Например, несоответствующее употребление множественного числа во фразе «Please give me one hamburgers!»*(множественное число существительного не соответствует количеству.)* звучит неправильно, так же как неправильный пол в «la gato»*(Артикль женского рода, тогда как существительное — мужского.* заставляет носителей языка посмеиваться. Примите также во внимание местоимение «you» или существительное «sheep», которое может единственным или множественным числом в зависимости от контекста.

Контекст в Perl сходен. Он определяет как количесто, так и вид используемых данных. Perl охотно попытается предоставить именно то, чего вы просите — при условии, если вы это делаете, выбирая соответствующий контекст.

Некоторые операции в Perl ведут себя по-разному в случаях, когда вы хотите получить ноль, один или несколько результатов. Конкретная конструкция

в Perl в случае, если вы напишете «Сделай это, но результаты меня не волнуют», может делать нечто совершенно другое по сравнению с «Сделай это, и я ожидаю получить несколько результатов». Другие операции позволяют вам определённо указать, предполагаете ли вы работать с числовыми данными, текстовыми данными, или данными, содержащими «истину» или «ложь».

Контекст может быть коварным если вы пытаетесь писать или читать код на Perl как серию единичных выражений в отрыве от их окружения. Вы можете обнаружить себя хлопающим по лбу после долгой сессии отладки, когда выясните, что ваши предположения относительно контекста были неверны. Если же напротив вы осознаёте контекст, ваш код будет более правильным — а также более чистым, гибким и лаконичным.

Пустой, скалярный и списочный контекст

Контекст количества определяет, *сколько* элементов вы ожидаете получить от операции. Близкая параллель — согласование числа между субъектом и глаголом в английском языке. Даже не зная формального определения этого лингвистического принципа, вы, вероятно, поймёте ошибку в предложении «Perl are a fun language». В Perl количество элементов, которое вы запросите, определяет, сколько вы получите.

Предположим, у вас есть функция под названием `find_chores()`, которая сортирует ваш список домашних дел по приоритету задач. Способ, которым вы вызовете эту функцию, определяет, что она будет делать. У вас можете не быть времени что либо делать, в этом случае вызов функции — это просто попытка выглядеть работящим. У вас может быть достаточно времени для выполнения одной задачи, или, может быть, у вас взрыв энергии в свободный выходной, и вы хотите выполнить как можно больше.

Если вы просто вызовете функция и никак не используете возвращаемое ей значение, вы вызвали функцию в *пустом контексте*:

```
find_chores();
```

Присвоение возвращаемого функцией значения единственному элементу

вычисляет функцию в *скалярном контексте*:

```
my $single_result = find_chores();
```

Присвоение результатов вызова функции массиву или списку, или использование её в списке, вычисляет функцию в *списочном контексте*:

```
my @all_results             = find_chores();
my ($single_element, @rest) = find_chores();
process_list_of_results( find_chores() );
```

Скобки во второй строке предыдущего примера группируют объявление двух переменных , чтобы присвоение вело себя так, как вы ожидаете. Если переменная `@rest` не используется, вы можете написать и так:

```
my ($single_element)   = find_chores();
```

...в этом случае скобки поясняют парсеру Perl 5, что вы подразумеваете списочный контекст для присваивания, несмотря на то, что присваиваете только одному элементу списка. Это неочевидно, но теперь, когда вы знаете об этом, различие контекста количества между этими двумя выражениями должно быть ясно:

```
my $scalar_context = find_chores();
my ($list_context) = find_chores();
```

Вычисление функции или выражения — исключая присваивание — в списочном контексте может вызвать путаницу. Списки распространяют списочный контекст на выражения, которые они содержат. Оба этих вызова `find_chores()` происходят в списочном контексте:

```
process_list_of_results( find_chores() );
```

```
my %results =
(
    cheap_operation     => $cheap_results,
    expensive_operation => find_chores(), # УПС!
);
```

Последний пример часто удивляет начинающих программистов, так как инициализация хеша списком значений налагает списочный контекст на `find_chores`. Используйте оператор `scalar` для наложения скалярного контекста:

```
my %results =
(
    cheap_operation     => $cheap_results,
    expensive_operation => scalar find_chores(),
);
```

Почему контекст важен? Зависящая от контекста функция может проверить контекст, в котором она вызывается, и определить, как много работы она должна сделать. В пустом контексте функция `find_chores()` может законно ничего не делать. В скалярном контексте она может найти только самую важную задачу. В списочном контексте она должна отсортировать и вернуть весь список.

Числовой, строковый и булев контекст

Другой вид контекста в Perl — *контекст значения* — определяет, как Perl интерпретирует кусок данных. Вы, вероятно, уже заметили, что Perl проявляет гибкость в определении того, имеете вы число или строку, и преобразовании их в тот вид, в который вы хотите. В обмен на отсутствие обязательств объявлять (или хотя бы следить) какой именно *тип* данных содержит переменная или выдаёт функция, контекст типа в Perl предоставляет подсказки, которые говорят компилятору, как обращаться с данными.

Perl приводит значения в конкретный надлежащий тип , в зависимости от используемого вами оператора. Например, оператор `eq` проверяет, что строки содержат одну и ту же информацию *как строки*:

```
say "Catastrophic crypto fail!" if $alice eq $bob;
```

Возможно, у вас был ставящий в тупик опыт, когда вы *знаете*, что строки различаются, но сравнение всё равно показывает, что они одинаковые:

```
my $alice = 'alice';
say "Catastrophic crypto fail!" if $alice == 'Bob';
```

Оператор `eq` обращается со своими операндами как со строками, принудительно налагая на них *строковый контекст*. Оператор `==` налагает *числовой контекст*. В числовом контексте обе строки возвращают `0` . Убедитесь, что используете соответствующий оператор для того типа контекста, который вам требуется.

Булев контекст имеет место, когды вы используете значение в условном выражении. В предыдущем примере `if` вычисляет результат операторов `eq` и `==` в булевом контексте.

В редких случаях вам, возможно, понадобится принудительно задать определённый контекст в случае, когда не существует оператора подходящего типа. Чтобы принудительно задать числовой контекст, прибавьте к переменной ноль. Чтобы принудительно задать строковый контекст, конкатенируйте переменную с пустой строкой. Чтобы принудительно задать булев контекст, используйте удвоенный оператор отрицания:

```
my $numeric_x =  0 + $x;  # принудительно задаёт числовой  контекст
my $stringy_x = '' . $x;  # принудительно задаёт строковый контекс
my $boolean_x =   !!$x;  # принудительно задаёт булев    контекст
```

Контекст типа проще распознать, чем контекст количества. Как только вы узнаете, какой оператор предоставляет какой контекст , вы редко будете ошибаться.

Неявные идеи

Контекст — лишь одна из лингвистических уловок в Perl. Программисты, понимающие эти уловки, могут, бросив взгляд на код, сразу же понять его наиболее важные характеристики. Другая важная лингвистическая особенность — эквивалент местоимений в Perl.

Подразумеваемая переменная-скаляр

Подразумеваемая переменная-скаляр (ещё называемая *переменная-топик*), `$_`, наиболее примечательна своим *отсутствием*: многие встроенные операции в Perl работают с содержимым `$_` в отсутствие явно указанной переменной. Вы можете использовать `$_` и как переменную, но это зачастую излишне.

Многие скалярные операторы Perl (включая `chr`, `ord`, `lc`, `length`, `reverse` и `uc`) работают с подразумеваемой переменной-скаляром, если вы не предоставили альтернативы. Например, встроенная функция `chomp` удаляет все завершающие последовательности новых строк из своего операнда_(См. `perldoc \-f chomp)` и `$/` для более точных деталей её поведения._:

```
my $uncle = "Bob\n";
chomp $uncle;
say "'$uncle'";
```

`$_` в Perl имеет ту же самую функцию, что и местоимение *it* (*это*) в английском. Без явно указанной переменной, `chomp` удаляет все завершающие последовательности новых строк из `$_`. Perl понимает, что вы имеете ввиду говоря «`chomp`» («обгрызть»); Perl всегда обгрызёт *это*, так что следующие две строки кода эквивалентны:

```
chomp $_;
chomp;
```

Аналогично, `say` и `print` оперируют с `$_` в отстутствие других аргументов:

```
print;  # выводит $_ в текущий дескриптор файла
say;    # выводит "$_\n" в текущий дескриптор файла
```

Средства работы с регулярными выражениями в Perl по умолчанию используют `$_` для сопоставления, замены и транслитерации:

```
$_ = 'My name is Paquito';
say if /My name is/;

s/Paquito/Paquita/;

tr/A-Z/a-z/;
say;
```

Директивы циклов в Perl по умолчанию используют `$_` как итерационную переменную. Рассмотрим директиву `for`, итерирующую по списку:

```
say "#$_" for 1 .. 10;

for (1 .. 10)
{
    say "#$_";
}
```

...или `while`:

```
while (<STDIN>)
{
    chomp;
    say scalar reverse;
}
```

...или `map`, преобразовывающую список:

```
my @squares = map { $_ * $_ } 1 .. 10;
say for @squares;
```

...или `grep`, фильтрующую список:

```
say 'Brunch time!'
    if grep { /pancake mix/ } @pantry;
```

Как английский становится запутанным, если вы используете слишком много местоимений и антецедентов, так и со смешанным использованием `$_`, явным или неявным, нужно быть осторожным. Неосмотрительное параллельное использование `$_` может привести к тому, что один кусок кода молча перезапишет значение, записанное другим. Если вы пишете функцию, которая использует `$_`, вы можете сломать работу с `$_` в вызывающей функции.

Начиная с Perl 5.10 появилась возможность объявлять `$_` как лексическую переменную , чтобы предотвратить такое разрушительное поведение:

```
while (<STDIN>)
{
    chomp;

    # ПЛОХОЙ ПРИМЕР
    my $munged = calculate_value( $_ );
    say "Original: $_";
    say "Munged  : $munged";
}
```

Если *calculate_value()* или любая другая функция изменит `$_`, это изменение сохранится на протяжении текущей итерации цикла. Добавление объявления `my` предотвращает потерю существующего экземпляра `$_`:

```
while (my $_ = <STDIN>)
{
```

```
    ...
}
```

Конечно, использование именованной лексической переменной может быть таким же ясным:

```
while (my $line = <STDIN>)
{
    ...
}
```

Используйте $_, так же, как вы использовали бы слово «это» в формальном письме: экономно, в разумных и хорошо определённых границах.

Подразумеваемая переменная-массив

Perl также предоставляет две неявные переменные-массива. Perl передаёт аргументы в функции в массиве @_. Операции работы с массивами внутри функций по умолчанию воздействуют на эту переменную, поэтому два этих фрагмента кода эквивалентны:

```
sub foo
{
    my $arg = shift;
    ...
}

sub foo_explicit_args
{
    my $arg = shift @_;
    ...
}
```

Так же как $_ соответствует местоимению *it* (*это*), @_ соответствует местоимениям *they* (*они*) и *them* (*их*). *В отличие* от $_, Perl автоматически

локализует для вас `@_` когда вы вызываете другие функции. Встроенные функции `shift` и `pop` работают с `@_`, если отсутствуют другие операнды.

Снаружи всех функций подразумеваемая переменная-массив `@ARGV` содержит аргументы командной строки, переданные в программу. Операции работы с массивами в Perl (включая `shift` и `pop`) неявно работают с `@ARGV` за пределами функций. Вы не сможете использовать `@_` когда подразумеваете `@ARGV`.

`@ARGV` имеет один специальный случай. Если вы читаете из пустого дескриптора файла `<>`, Perl будет воспринимать каждый элемент в `@ARGV` как *имя* файла, который нужно открыть для чтения. (Если массив `@ARGV` пуст, Perl будет читать со стандартного ввода.) Это неявное поведение `@ARGV` полезно при написании коротких программ, таких как этот фильтр для командной строки, изменяющий порядок своих входных данных на противоположный:

```
while (<>)
{
    chomp;
    say scalar reverse;
}
```

Почему `scalar`? `say` налагает списочный контекст на свои операнды. `reverse` передаёт свой контекст своим операндам, обрабатывая их как список в списочном контексте и как конкатенированную строку в скалярном контексте. Если поведение `reverse` выглядит сбивающим с толку, ваши инстинкты верны. Perl 5 вероятно должен быть разделить «перевернуть строку» и «перевернуть список».

Если вы запустите этот код со списком файлов:

…результат будет одним длинным потоком вывода. Не передавая никаких аргументов, вы можете задать свой собственный стандартный ввод, передав его по конвейеру из другой программы или введя напрямую. Однако, Perl хорош для гораздо большего, чем маленькие программы для командной строки…

4. Perl и его сообщество

Наибольшее достижение Perl 5 — это огромное количество повторно используемых библиотек, разработанных для него. Там, где для Perl 4 приходилось делать форки, например, для подключения к базам данных, таким как Oracle или Sybase, Perl 5 имеет настоящий механизм расширений. Ларри хотел, чтобы люди создавали и поддерживали свои собственные расширения, не разбивая Perl на тысячи несовместимых диалектов — и это сработало.

Это техническое достижение почти так же важно, как рост сообщества вокруг Perl 5. *Люди* пишут библиотеки. *Люди* строят свою работу на работе других людей. *Люди* делают сообщество стоящим вступления, сохранения и расширения.

У Perl сильное и здоровое сообщество. Оно приветствует желающих стать его участниками, на всех уровнях, от новичков до разработчиков ядра. Воспользуйтесь знаниями и опытом бесчисленного количества других Perl-программистов, и вы сами станете лучше как программист.

CPAN

Perl 5 — прагматичный язык сам по себе, но ещё более прагматичное Perl-сообщество расширило этот язык и сделало свою работу доступной для всего мира. Если у вас есть требующая решения задача, велики шансы, что кто-то уже написал код на Perl для её решения — и поделился им.

Программирование в Современном Perl подразумевает активное использование CPAN (http://www.cpan.org/). Comprehensive Perl Archive Network (Всеобъемлющая архивная сеть Perl) — система загрузки и зеркалирования свободно распространяемого повторно используемого кода на Perl. Это один из самых, если не *самый* большой архив программных библиотек в мире. CPAN предлагает библиотеки для всего, от доступа к базам данных, инструментов профилирования и реализации протоколов для практически всех когда-либо созданных сетевых устройств до звуковых и графических библиотек и обёрток для разделяемых библиотек в вашей системе.

Современный Perl без CPAN — всего лишь очередной язык. Современный Perl с CPAN — изумителен.

CPAN хранит *дистрибутивы* — коллекции повторно используемого Perl-кода. Один дистрибутив может содержать один или несколько *модулей*, самодостаточных библиотек Perl-кода. Каждый дистрибутив занимает своё собственное пространство имён в CPAN и предоставляет уникальные метаданные.

Сам по себе CPAN — всего лишь зеркалирующий сервис. Авторы загружают свои дистрибутивы, и CPAN отправляет их на сайты-зеркала, с которых пользователи и CPAN-клиенты скачивают, конфигурируют, собирают, тестируют и устанавливают их. Система успешна из-за своей простоты и вклада тысяч добровольцев, строящих свою работу на этой системе распространения. В частности, развитие стандартов сообщества определило атрибуты и характеристики правильно построенного CPAN-дистрибутива. Они включают:

- стандарты для автоматизированных установщиков из CPAN;
- стандарты метаданных, описывающих, что предоставляет и чего ожидает каждый дистрибутив;
- стандарты документирования и лицензирования.

Дополнительные сервисы CPAN включают всестороннее автоматизированное тестирование и оповещение для улучшения качества сборок и корректности работы на разных платформах и версиях Perl. Каждый дистрибутив CPAN имеет свою собственную очередь тикетов на http://rt.cpan.org/ для сообщения о багах и работы с авторами. Кроме того, сайты CPAN ссылаются на предыдущие версии дистрибутивов, рейтинги модулей, комментарии к документации и т. д. Всё это доступно на http://search.cpan.org/.

Установки Современного Perl включают два клиента для подключения, поиска, скачивания, сборки, тестирования и установки CPAN-дистрибутивов, CPAN.pm и CPANPLUS. По большей части, оба этих клиента равнозначны для базовой установки. Эта книга рекомендует использовать только CPAN.pm ввиду его повсеместного распространения. Со свежими версиями (последний стабильный релиз на момент написания — 1.9800) установка модулей достаточно проста. Запустите клиент следующей командой:

Так осуществляется установка дистрибутива из клиента:

...а так — напрямую из командной строки:

Руководство Эрика Вильгельма (Eric Wilhelm) по конфигурированию CPAN.pm_(http://learnperl.scratchcomputing.com/tutorials/configuration/)_- включает отличную секцию по разрешению проблем.

Инструменты управления CPAN

Если ваша операционная система предоставляет собственную инсталляцию Perl 5, эта версия может быть устаревшей или может зависеть от определённых версий CPAN-дистрибутивов. Серьёзные Perl-разработчики часто возводят виртуальные стены между системным Perl и их собственными установками Perl, предназначенными для разработки. Несколько проектов помогают сделать это возможным.

`App::cpanminus` — относительно новый CPAN-клиенты, нацеленный на скорость, простоту и отсутствие необходимой настройки. Установите его командой `cpan App::cpanminus` или следующим образом:

`App::perlbrew` — система управления и переключения между вашими собственными инсталляциями нескольких версий и конфигураций Perl. Установить её очень просто:

CPAN-дистрибутив `local::lib` позволит вам устанавливать и управлять дистрибутивами в вашей пользовательской директории, а не в системе в целом. Это эффективный способ поддерживать CPAN-дистрибутивы, не затрагивая других пользователей. Установка несколько более сложна, чем для предыдущих двух дистрибутивов, хотя `App::local::lib::helper` может упростить процесс. Смотрите http://search.cpan.org/perldoc?local::lib и http://search.cpan.org/perldoc?App::local::lib::helper для дальнейших подробностей.

Все три проекта имеют тенденцию подразумевать Unix-подобное окружение (такое как GNU/Linux дистрибутив или даже Mac OS X). Пользователи Windows могут воспользоваться загрузкой «всё в одном» Padre (http://padre.perlide.org/download.html).

Сайты сообщества

Домашняя страница Perl на http://www.perl.org/ содержит ссылки на документацию Perl, исходные коды, обучающие материалы, списки рассылок и некоторые важные проекты сообщества. Если вы новичок в Perl, список рассылки для начинающих — дружественное место, в котором новичок может задать вопросы и получить точные и полезные ответы. См. http://learn.perl.org/faq/beginners.html.

Дом разработки Perl — http://dev.perl.org/, содержащий ссылки на важные ресурсы по разработке ядра Perl 5 и Perl 6_(Хотя так же смотрите http://www.perl6.org/)._.

На Perl.com публикуются статьи и обучающие материалы на тему Perl и его культуры. Его архивы простираются назад в 20 век. См. http://www.perl.com/.

Центральное месторасположение CPAN — http://www.cpan.org/, хотя опытные пользователи проводят больше времени на http://search.cpan.org/. Этот центральный хаб распространения повторно используемого, свободно распространяемого Perl-кода — важнейшая часть Perl-сообщества. MetaCPAN (https://metacpan.org/) — новый альтернативный фронтенд к CPAN.

PerlMonks, на http://perlmonks.org/, это сайт сообщества, посвящённый дискуссиям о программировании на Perl. Его одиннадцатилетняя история делает его одним из самых почтенных сайтов вопросов и ответов по языкам программирования.

Несколько сайтов сообщества предлагают новости и комментарии. http://blogs.perl.org/ — бесплатная блог-платформа, открытая для любого участника Perl-сообщества.

Другие сайты объединяют размышления Perl-хакеров, включая http://perlsphere.net/, http://planet.perl.org/, и http://ironman.enlightenedperl.org/. Последний — часть инициативы Enlightened Perl Organization (Организация Просвещённый Perl) (http://enlightenedperl.org/) по увеличению количества и качества публикаций о Perl в вебе.

Perl Buzz (http://perlbuzz.com/) на регулярной основе собирает и публикует некоторые из наиболее интересных и полезных новостей о Perl. Perl Weekly

(http://perlweekly.com/) предлагает еженедельную подборку новостей из мира Perl.

Сайты разрабоки

Best Practical Solutions (http://bestpractical.com/) поддерживает инсталляцию своей популярной системы отслеживания запросов, RT, как для CPAN-авторов, так и для разработки Perl 5 и Perl 6. Каждый CPAN-дистрибутив имеет собственную очередь в RT, доступную на http://rt.cpan.org/, на которую ведёт ссылка с `search.cpan.org`. Perl 5 и Perl 6 имеют отдельные очереди в RT, доступные на http://rt.perl.org/.

Список рассылки «Perl 5 Porters» (или *p5p*) — центральная точка разработки самого Perl 5. См. http://lists.cpan.org/showlist.cgi?name=perl5-porters.

На сайте Perl Foundation (http://www.perlfoundation.org/) размещена вики на тему всего, что касается Perl 5. См. http://www.perlfoundation.org/perl5.

Многие Perl-хакеры используют Github (http://github.com/) для размещения своих проектов_(...включая исходники этой книги по адресу http://github.com/chromatic/modern_perl_book/._. Обратите особенное внимание на GitPAN (http://github.com/gitpan/), где размещены репозитории, отражающие полную историю каждого дистрибутива в CPAN.

События

Perl-сообщество проводит бесчисленное количество конференций, воркшопов, семинаров и встреч. В особенности, проводимые сообществом YAPC —Yet Another Perl Conference (Ещё одна Perl-конференция) — успешная локальная низкобюджетная модель конференций, проводимых на разных континентах. См. http://yapc.org/.

Вики Perl Foundation содержит список других событий по адресу http://www.perlfoundation.org/perl5/index.cgi?perl_events.

Сотни локальных групп Perl-монгеров (Perl Mongers) часто собираются

вместе для бесед на технические темы и социального взаимодействия. См. http://www.pm.org/.

IRC

Когда Perl-монгеры не могут встретиться лично, многие сотрудничают и общаются онлайн с помощью текстовой чат-системы, известной как IRC. Многие из наиболее популярных и полезных Perl-проектов имеют свои собственные IRC-каналы, такие как *#moose* и *#catalyst*.

Главный сервер Perl-сообщества — irc://irc.perl.org/. Заслуживающие внимания каналы включают *#perl-help*, для общей помощи по Perl-программированию, и *#perl-qa*, посвящённый тестированию и другим темам контроля качества. Имейте ввиду, что канал *#perl* — канал общего назначения для обсуждения всего, что его участники захотят обсудить_(...и по этой причине в основном не служба поддержки._.

5. Язык Perl

Как и разговорный язык, Perl в целом — это комбинация нескольких меньших, но взаимосвязанных частей. В отличие от разговорного языка, где оттенок и интонация голоса и интуиция позволяют людям общаться, несмотря на некоторое недопонимание и неясные представления, компьютеры и программный код требуют точности. Вы можете писать эффективный код на Perl, не зная каждой детали каждой возможности языка, но для написания хорошего кода вы должны понимать, как они работают вместе.

Имена

Имена (или *идентификаторы*) в программах на Perl повсюду: переменные, функции, пакеты, классы и даже дескрипторы файлов. Все эти имена начинаются с буквы или знака подчёркивания и могут включать любую комбинацию букв, цифр и знаков подчёркивания. Если задействована прагма `utf8`, вы можете использовать в идентификаторах любые словарные символы UTF-8. Всё это допустимые в Perl идентификаторы:

```
my $name;
my @_private_names;
my %Names_to_Addresses;

sub anAwkwardName3;

# с включенным use utf8;
package Ingy::DE<ouml>t::Net;
```

Такие идентификаторы недопустимы в Perl:

```
my $invalid name;
my @3;
my %~flags;

package a-lisp-style-name;
```

Имена существуют в первую очередь для пользы программиста. Эти правила применяются только к именам литералов, которые появляются в вашем исходном коде «как есть», таким как `sub fetch_pie` или `my $waffleiron`. Только парсер Perl принуждает следовать правилам об именах идентификаторов.

Динамическая натура Perl позволяет вам ссылаться на сущности с помощью имён, генерируемых во время исполнения или указанных как входные данные программы. Этот *символьный поиск* обеспечивает гибкость ценой некоторого ущерба безопасности. В частности, непрямой вызов функций или методов или поиск символов в пространстве имён позволяет вам обойти парсер Perl.

Такое поведение может привести к запутанному коду. Как эффективно рекомендует Марк Джейсон Доминус (Mark Jason Dominus)*(http://perl.plover.com/varvarname.html)*, лучше используйте хеши или вложенные структуры данных .

Имена переменных и сигилы

Имена переменных всегда имеют предшествующий *сигил* (или символ), который обозначает тип значения переменной. *Скалярные переменные* используют знак доллара ($). *Переменные-массивы* используют знак «at» («собака»), (@). *Переменные-хеши* используют знак процента (%):

```
my $scalar;
my @array;
my %hash;
```

Эти сигилы обеспечивают визуальное разделение пространств имён для имён переменных. Возможно — хотя это и может привести к путанице — объявить несколько переменных с одним и тем же именем, но разными типами:

```
my ($bad_name, @bad_name, %bad_name);
```

Хотя Perl и не будет сбит с толку, люди, читающие этот код, будут.

Сигилы Perl 5 — *варьирующиеся сигилы*. Как контекст определяет, сколько элементов вы ожидаете от операции, или какого типа данные вы ожидаете получить, так и сигил определяет, как вы манипулируете данными переменной. Например, для доступа к единственному элементу массива или хеша вы должны использовать сигил скаляра ($):

```
my $hash_element  = $hash{ $key };
my $array_element = $array[ $index ]

$hash{ $key }     = 'value';
$array[ $index ]  = 'item';
```

Параллель с контекстом количества важна. Использование скалярного элемента структуры как *lvalue*, левого значения (цель присвоения, находящаяся слева от знака =), налагает скалярный контекст на *rvalue*, правое значение (присваемое значение, находящееся справа от знака =).

Аналогично, доступ к нескольким элементам хеша или массива — операция, известная как *получение среза* — использует символ @ и налагает списочный контекст_(…даже если сам список содержит ноль или один элемент_:

```
my @hash_elements  = @hash{ @keys };
my @array_elements = @array[ @indexes ];

my %hash;
@hash{ @keys }     = @values;
```

Наиболее надёжный способ определить тип переменной — скаляр, массив или хеш — обратить внимание на производимые с ней операции. Скаляры поддерживают все базовые операции, такие как строковые, числовые и булевы манипуляции. Массивы поддерживают доступ по индексу посредством квадратных скобок. Хеши поддерживают доступ по ключу посредством фигурных скобок.

Пространства имён

Perl предоставляет механизм группировки сходных функций и переменных в их собственные уникальные именованные пространства — *пространства имён* . Пространство имён — это именованная коллекция символов. Perl позволяет использование многоуровневых пространств имён, с именами, соединёнными двойным двоеточием (`::`), где `DessertShop::IceCream` ссылается на логическую коллекцию родственных переменных и функций, таких как `scoop()` и `pour_hot_fudge()`.

Внутри пространства имён вы можете использовать короткие имена его членов. Снаружи пространства имён ссылайтесь на его член, используя его *полностью определённое имя.* Таким образом, `add_sprinkles()` внутри `DessertShop::IceCream` ссылается на ту же самую функцию, что и `DessertShop::IceCream::add_sprinkles()` снаружи пространства имён.

Хотя к именам пакетов применимы стандартные правила именования, по соглашению имена определённых пользователями пакетов начинаются с заглавных букв. Ядро Perl резервирует строчные имена пакетов для прагм ядра , таких как `strict` и `warnings`. Эта политика закреплена в первую очередь рекомендациями сообщества.

Все пространства имён в Perl 5 глобально видимы. Когда Perl ищет символ в `DessertShop::IceCream::Freezer`, он ищет в символьной таблице `main::` символ, представляющий пространство имён `DessertShop::`, затем в нём — пространство имён `IceCream::`, и так далее. `Freezer::` видим снаружи пространства имён `IceCream::`. Вложенность одного в другое — всего лишь механизм хранения, и ничего не говорит об отношениях между родительскими и дочерними пакетами, или пакетами, находящимися на одном уровне. Только программист может сделать *логические* отношения между сущностями очевидными — выбрав хорошие имена и правильно их организовав.

Переменные

Переменная в Perl — это место хранения значения . Хотя простая программа может манипулировать значениями напрямую, большинство программ работают с переменными для упрощения логики кода. Переменные представляют значения; проще объяснить теорему Пифагора в терминах переменных `a`, `b` и `c`, чем демонстрируя её принцип длинным списком корректных значений. Эта концепция может выглядеть очевидной, но эффективное программирование требует от вас справляться с искусством балансирования между общим и повторно используемым с одной стороны и конкретным — с другой.

Области видимости переменных

Переменные видны частям вашей программы в зависимости от их области видимости . Большая часть переменных, с которыми вы встретитесь, имеет лексическую область видимости . Сами *файлы* задают свои собственные лексические области видимости, так что объявление `package` само по себе новую область видимости не создаёт:

```
package Store::Toy;

my $discount = 0.10;

package Store::Music;

# переменная $discount всё ещё видима
say "Our current discount is $discount!";
```

Начиная с Perl 5.14, вы можете задать блок для объявления `package`. Такой синтаксис *создаёт* лексическую область видимости:

```
package Store::Toy
{
    my $discount = 0.10;
```

```
}

package Store::Music
{
    # переменная $discount не доступна
}

package Store::BoardGame;

# переменная $discount всё ещё не доступна
```

Сигилы переменных

Сигил переменной в объявлении определяет тип переменной: скаляр, массив или хеш. Сигил, используемый при доступе к переменной, варьируется в зависимости от того, что вы делаете с переменной. Например, вы объявляете массив как `@values`. `$values[0]` — доступ к первому элементу массива — единичному значению. `@values[ @indices ]` — доступ к списку значений из массива.

Анонимные переменные

Переменные в Perl не *требуют* имён. Имена существуют чтобы помочь вам, программисту, следить за `$apple`, `@barrels` или `%cheap_meals`. Переменные, созданные в вашем программном коде *без* литеральных имён, называются *анонимными* переменными. Единственный способ обратиться к анонимной переменной — по ссылке .

Переменные, типы и приведение типов

Переменная в Perl 5 представляет как значение (курс доллара, доступные начинки для пиццы, гитарные магазины с телефонными номерами), так и контейнер, в котором хранится это значение. Система типов Perl имеет дело с *типами значений* и *типами контейнеров*. Тогда как *тип контейнера* переменной — скаляр, массив или хеш — не может измениться, относительно

типа значения переменной Perl проявляет гибкость. Вы можете сохранить строку в переменной в одной строке кода, добавить к этой переменной число в следующей и переназначить ей ссылку на функцию в третьей.

Выполнение над переменной операции, требующей конкретного типа значения, может вызвать приведение из текущего типа значения переменной.

Например, документированный способ определить количество элементов в массиве — обратиться к нему в скалярном контексте . Так как скалярная переменная может содержать только скаляр, присвоение массива скаляру налагает на операцию скалярный контекст, и массив, вычисленный в скалярном контексте, возвращает количество элементов в этом массиве:

```
my $count = @items;
```

Отношения между типами переменных, сигилами и контекстом имеют первостепенную важность.

Значения

Структура программы в большой степени зависит от способов, используя которые вы моделируете ваши данные с помощью соответствующих переменных.

Тогда как переменные позволяют производить абстрактные манипуляции с данными, значения, которые они содержат, делают программы конкретными и полезными. Чем более точны значения, тем лучше ваши программы. Эти значения — это данные: имя и адрес вашей тётушки, расстояние между вашим офисом и полем для игры в гольф на луне или вес всех печенек, которые вы съели за прошлый год. Внутри вашей программы правила, касающиеся формата этих данных, зачастую строги. Действенные программы требуют действенных (простых, быстрых, более компактных, более эффективных) способов представления их данных.

Строки

Строка — это кусок текстовых или двоичных данных без какого-либо особого форматирования или содержимого. Это может быть ваше имя, содержимое файла изображения или сама ваша программа. Строка имеет значение в программе только когда вы придаёте ей это значение.

Для представляения литеральной строки в вашей программе, окружите её парой символов заключения в кавычки. Наиболее распространённые *ограничители строк* — одиночные и двойные кавычки:

```
my $name    = 'Donner Odinson, Bringer of Despair';
my $address = "Room 539, Bilskirnir, Valhalla";
```

Символы в *строке, заключённой в одиночные кавычки*, представляют в точности себя, за двумя исключениями. Одиночная кавычка, помещаемая в заключённую в одиночные кавычки строку, должна быть экранирована предшествующим обратным слешем:

```
my $reminder = 'Don\'t forget to escape '
             . 'the single quote!';
```

Также вы должны экранировать обратный слеш, находящийся в конце строки, чтобы избежать экранирования закрывающего ограничителя и получения синтаксической ошибки:

```
my $exception = 'This string ends with a '
              . 'backslash, not a quote: \\';
```

Любой другой обратный слеш станет частью строки как есть, если только два обратных слеша не расположены друг за другом, в этом случае первый будет экранировать второй:

```
is('Modern \ Perl', 'Modern \\ Perl',
    'single quotes backslash escaping');
```

Для *строки, заключённой в двойные кавычки*, доступно ещё некоторое количество специальных символов. Например, вы можете закодировать в строке иначе невидимые пробельные символы:

```
my $tab       = "\t";
my $newline   = "\n";
my $carriage  = "\r";
my $formfeed  = "\f";
my $backspace = "\b";
```

Это демонстрирует полезный принцип: синтаксис, используемый для объявления строки, может варьироваться. Вы можете представить символ табуляции в строке как `\t` или введя его напрямую. С точки зрения Perl, обе строки ведут себя одинаково, несмотря на то, что конкретное представление строки в исходном коде может отличаться.

Объявление строки может пересекать логические переводы строк; эти два объявления эквивалентны:

```
my $escaped = "two\nlines";
my $literal = "two
lines";
is $escaped, $literal, 'equivalent \n and newline';
```

Эти последовательности зачастую проще читать, чем их пробельные эквивалентны.

Строки в Perl имеют переменную длину. По мере того, как вы манипулируете строками и модифицируете их, Perl будет соответствующим образом изменять их размер. Например, вы можете объединить несколько строк в одну большую строку с помощью оператора *конкатенации* `.`:

```
my $kitten = 'Choco' . ' ' . 'Spidermonkey';
```

Это фактически то же самое, что и инициализация строки сразу целиком.

Также вы можете *интерполировать* значение скалярной переменной или значений массива в строке, заключённой в двойные кавычки, так что *текущее* содержимое переменной станет частью строки, как если бы вы их конкатенировали:

```
my $factoid = "$name lives at $address!";

# эквивалентно
my $factoid = $name . ' lives at ' . $address . '!';
```

Включение двойной кавычки в заключённую в двойные кавычки строку требует её *экранирования* (то есть предварения её обратным слешем):

```
my $quote = "\"Ouch,\", he cried.  \"That I<hurt>!\"";
```

Когда повторение обратных слешей становится громоздким, используйте альтернативный *оператор заключения в кавычки*, для которого вы можете выбрать альтернативный ограничитель строк. Оператор `q` аналогичен одиночным кавычкам, тогда как оператор `qq` обеспечивает поведение двойных кавычек. Символ, следующий сразу за оператором, определяет, какой символ используется для ограничения строк. Если этот символ является открывающим символом симметричной пары — такой как открывающая и закрывающая скобки — закрывающий символ будет конечным ограничителем. В ином случае, сам указанный символ будет как начальным, так и конечным ограничителем.

```
my $quote     = qq{"Ouch", he said.  "That I<hurt>!"};
my $reminder  =  q^Don't escape the single quote!^;
my $complaint =  q{It's too early to be awake.};
```

Если объявление сложной строки, содержащей последовательность экранированных символов, слишком утомительно, используйте синтаксис *heredoc* (встроенных документов) для присвоения строке одной или нескольких строчек текста:

```
my $blurb =<<'END_BLURB';
```

```
He looked up. "Time is never on our side, my child.
Do you see the irony? All they know is change.
Change is the constant on which they all can agree.
We instead, born out of time, remain perfect and
perfectly self-aware. We only suffer change as we
pursue it. It is against our nature. We rebel
against that change. Shall we consider them
greater for it?"
END_BLURB
```

Синтаксис `<<'END_BLURB'` имеет три части. Двойные угловые скобки открывают встроенный документ. Кавычки определяют, будет ли он проявлять поведение одиночных или двойных кавычек. Поведение по умолчанию — интерполяция двойных кавычек. `END_BLURB` — произвольный идентификатор, который парсер Perl 5 использует как конечный ограничитель.

Будьте внимательны — независимо от отступа самого объявления встроенного документа, конечный ограничитель *должен* быть расположен в начале строки:

```
sub some_function {
    my $ingredients =<<'END_INGREDIENTS';
    Two eggs
    One cup flour
    Two ounces butter
    One-quarter teaspoon salt
    One cup milk
    One drop vanilla
    Season to taste
END_INGREDIENTS
}
```

Использование строки в нестроковом контексте приведёт к приведению типа
.

Юникод и строки

Юникод — это система представления символов письменных языков мира. В то время как большая часть английского текста использует набор символов, состоящий всего из 127 символов (что требует для хранения семи битов и замечательно вписывается в восьмибитовый байт), наивно предполагать, что вам рано или поздно не понадобится умляут.

Строки в Perl 5 могут представлять один из двух отдельных, но связанных типов данных:

- Последовательности символов Юникод

  Каждый символ имеет *код*, уникальное число, которое идентифицирует его в таблице символов Юникод.

- Последовательность октетов

  Двоичные данные — это последовательность *октетов*, восьмибитовых чисел, каждое из которых может представлять число от 0 до 255.

Юникодовые строки и двоичные строки выглядят похоже. Каждая имеет длину (`length()`). Каждая поддерживает стандартные строковые операции, такие как конкатенация, работа с подстроками и обработка регулярными выражениями. Любая строка, не являющаяся чистыми двоичными данными, есть текстовые данные, и должна быть представлена последовательностью символов Юникод.

Однако, из-за того, как ваша операционная система представляет данные на диске, или полученные от пользователей, или передаваемые по сети — как последовательность октетов — Perl не может знать, являются ли читаемые вами данные файлом изображения, или текстовым документом, или чем-нибудь ещё. По умолчанию Perl воспринимает все входящие данные как последовательность октетов. Вы должны наделить эти данные конкретным значением.

Кодировки символов Строка Юникод — это последовательность октетов, которая представляет последовательность символов. *Кодировка Юникод*

устанавливает соответствие последовательностей октетов символам. Некоторые кодировки, такие как UTF-8, могут кодировать все символы из набора символов Юникод. Другие кодировки представляют его подмножество. Например, ASCII кодирует простой английский текст без диакритических символов, тогда как Latin-1 может представлять текст на большинстве языков, использующих латинский алфавит.

Чтобы избежать большинства проблем с Юникодом, всегда используйте соответствующие кодировки для декодирования входных и выходных данных вашей программы.

Юникод в ваших дескрипторах файлов Если вы скажете Perl, что определённый дескриптор файла работает с кодированным текстом, Perl будет конвертировать входящие октеты в строки Юникод автоматически. Чтобы это сделать, добавьте слой ввода-вывода к режиму встроенной функции `open`. *Слой ввода-вывода* оборачивается вокруг входа и выхода и конвертирует данные. В данном случае, слой `:utf8` декодирует данные в UTF-8:

```
use autodie;

open my $fh, '<:utf8', $textfile;

my $unicode_string = <$fh>;
```

Также вы можете модифицировать существующий дескриптор файла с помощью `binmode`, как для ввода, так и для вывода:

```
binmode $fh, ':utf8';
my $unicode_string = <$fh>;

binmode STDOUT, ':utf8';
say $unicode_string;
```

Без режима `utf8` печать строк Юникод в дескриптор файла приведёт к предупреждению (`Wide character in %s`), потому что файлы содержат октеты, а не символы Юникод.

Юникод в ваших данных Базовый модуль `Encode` предоставляет функцию, называемую `decode()`, для конвертации скаляра, содержащего данные, в строку Юникод. Соответствующая функция `encode()` конвертирует из внутренней кодировки Perl в желаемую выходную кодировку:

```
my $from_utf8 = decode('utf8', $data);
my $to_latin1 = encode('iso-8859-1', $string);
```

Юникод в ваших программах Вы можете включать символы Юникод в ваши программы тремя способами. Самый простой — использовать прагму `utf8` , которая указывает парсеру Perl интерпретировать остальную часть файла исходного кода как имеющую кодировку UTF-8. Это позволяет вам использовать символы Юникод в строках и идентификаторах:

```
use utf8;

sub E<pound>_to_E<yen> { ... }

my $yen = E<pound>_to_E<yen>('1000E<pound>');
```

Чтобы *писать* такой код, ваш текстовый редактор должен понимать UTF-8, а вы должны сохранить файл в соответствующей кодировке.

Внутри строк в двойных кавычках вы можете использовать экранированные последовательности Юникод для представления кодировок символов. Синтаксис `\x{}` представляет один символ; поместите в фигурных скобках шестнадцатиричную форму номера символа в Юникод:

```
my $escaped_thorn = "\x{00FE}";
```

Некоторые символы Юникод имеют имена, и эти имена зачастую проще читать, чем номера Юникод. Используйте прагму `charnames` для того, чтобы их включить, и `\N{}`, чтобы на них ссылаться:

```
use charnames ':full';
use Test::More tests => 1;

my $escaped_thorn = "\x{00FE}";
my $named_thorn   = "\N{LATIN SMALL LETTER THORN}";

is $escaped_thorn, $named_thorn,
    'Thorn equivalence check';
```

Вы можете использовать формы `\x{}` и `\N{}` в регулярных выражениях, так же как и в любых местах, где допустимо использовать строку или символ.

Неявная конвертация Большинство проблем Юникод в Perl проистекают из того факта, что строка может быть или последовательностью октетов, или последовательностью символов. Perl позволяет вам комбинировать эти типы посредством неявных конвертаций. Когда эти конвертации неверны, они редко *очевидно* неверны.

Когда Perl конкатенирует последовательность октетов с последовательностью символов Юникод, он неявно декодирует последовательность октетов, используя кодировку Latin-1. Результирующая строка будет содержать символы Юникод. Когда вы печатаете символы Юникод, Perl кодирует строку, используя UTF-8, потому что Latin-1 не может отображать весть набор символов Юникод: Latin-1 — подмножество UTF-8.

Эта асимметрия может привести к строкам Юникод, закодированным как UTF-8 для вывода и декодированным как Latin-1 на вводе.

Хуже того, если текст содержит только английские символы без знаков диакритики, то ошибка скрывается — потому что обе кодировки одинаково отображают каждый символ.

```
my $hello    = "Hello, ";
```

```
my $greeting = $hello . $name;
```

Если `$name` содержит английское имя, такое как *Alice*, вы никогда не заметите никаких проблем, потому что представление Latin-1 точно такое же, как представление UTF-8. Если `$name` содержит такое имя, как *José*, `$name` может содержать несколько возможных значений:

- `$name` содержит четыре символа Юникод.
- `$name` содержит четыре октета Latin-1, представляющих четыре символа Юникод.
- `$name` содержит пять октетов UTF-8, представляющих четыре символа Юникод.

Строковый литерал имеет несколько возможных сценариев:

- Это строковый литерал ASCII, содержащий октеты.

  ```
  my $hello = "Hello, ";
  ```

- Это строковый литерал Latin-1 без определённой кодировки, содержащий октеты.

  ```
  my $hello = "E<iexcl>Hola, ";
  ```

  Строковый литерал содержит октеты.

- Это строковый литерал не-ASCII с действующей прагмой `utf8` или `encoding`, содержащий символы Юникод.

  ```
  use utf8;
  my $hello = "KuirabE<aacute>, ";
  ```

Если и `$hello`, и `$name` — строки Юникод, конкатенация порождает другую строку Юникод.

Если обе строки — потоки октетов, Perl конкатенирует их в новую строку, содержащую октеты. Если оба значения — октеты с одной и той же кодировкой, например, оба — Latin-1, конкатенация сработает корректно. Если же октеты имеют разную кодировку, например, конкатенация добавляет данные в UTF-8 к данным в Latin-1, то результирующая последовательность октетов не будет иметь смысла *ни в какой* из кодировок. Это может случиться, если пользователь ввёл имя как данные UTF-8, а приветствие было строковым литералом Latin-1, но программа не декодировала ни то, ни другое.

Если только одно из значений — строка Юникод, Perl будет декодировать другое как данные Latin-1. Если это не корректная кодировка, результирующие символы Юникод будут неверными. Например, если пользовательский ввод был данными в UTF-8, а строковый литерал был строкой Юникод, имя было бы некорректно декодировано в пять символов Юникод в виде *JosÃ©* (*именно!*) вместо *José*, потому что данные UTF-8 означают что-то совсем другое, когда декодируются как данные Latin-1.

Смотрите `perldoc perluniintro` для получения намного более детального объяснения Юникод, кодировок, и того, как управлять входящими и исходящими данными в мире Юникод_(Для *ещё)* больших подробностей на тему эффективного управления Юникодом в ваших программах, смотрите ответ Тома Кристиансена (Tom Christiansen) на вопрос «Почему Современный Perl избегает UTF-8 по умолчанию?» http://stackoverflow.com/questions/6162484/why-does-modern-perl-avoid-utf-8-by-default/6163129#6163129_.

Числа

Perl поддерживает числа в виде как целых значений, так и значений с плавающей точкой. Вы можете выражать их в научной нотации, так же как в двоичной, восьмеричной и шестнадцатиричной формах:

```
my $integer   = 42;
my $float     = 0.007;
my $sci_float = 1.02e14;
my $binary    = 0b101010;
my $octal     = 052;
```

```
my $hex        = 0x20;
```

Выделенные жирным символы — числовые префиксы для двоичной, восьмеричной и шестнадцатиричной нотации соответственно. Имейте ввиду, что ведущий ноль в целом числе *всегда* означает восьмеричный режим.

Вы не можете использовать запятые для отделения тысяч в числовых литералах, потому что парсер сочтёт запятую оператором запятой. Вместо этого используйте в числах подчёркивания. Парсер будет будет воспринимать их как невидимые символы; ваши читатели, возможно, нет. Эти записи равнозначны:

```
my $billion = 1000000000;
my $billion = 1_000_000_000;
my $billion = 10_0_00_00_0_0_0;
```

Подумайте, какой вариант наиболее читабелен.

Из-за приведения типа , Perl-программистам редко приходится беспокоиться о конвертации текста, прочитанного из-за пределов программы, в числа. Perl будет обращаться со всем, что выглядит как число, *как* с числом в числовом контексте. В редких ситуациях, когда вам нужно знать, выглядит ли нечто как число для Perl, используйте функцию `looks_like_number` из базового модуля `Scalar::Util`. Эта функция возвращает истину, если Perl будет считать переданный аргумент числовым.

Модуль `Regexp::Common` из CPAN предоставляет несколько хорошо протестированных регулярных выражений для определения более конкретных допустимых *типов* числовых значений (целое число, целое значение, значение с плавающей точкой).

Undef

Значение `undef` в Perl 5 представляет собой неприсвоенное, неопределённое и неизвестное значение. Объявленные, но не определённые скалярные переменные всегда содержат `undef`:

```
my $name = undef;   # необязательное присваивание
my $rank;           # тоже содержит undef
```

`undef` даёт ложь в булевом контексте. Использование `undef` в строковом контексте — например, интерполяция его в строке, — генерирует предупреждение `uninitialized value`:

```
my $undefined;
my $defined = $undefined . '... and so forth';
```

...выдаёт:

Встроенная функция `defined` возвращает истинное значение, если её операнд является определённым значением (любым отличным от `undef`):

```
my $status = 'suffering from a cold';

say defined $status;  # 1, истинное значение
say defined undef;    # пустая строка, ложное значение
```

Пустой список

При использовании справа от присваивания, конструкция `()` обозначает пустой список. В скалярном контексте он вычисляется в `undef`. В списочном контексте это пустой список. При использовании слева от присваивания, конструкция `()` налагает списочный контекст. Для подсчёта количества элементов, возвращаемых выражением в списочном контексте, без использования временной переменной, используйте следующую идиому :

```
my $count = () = get_all_clown_hats();
```

Из-за правой ассоциативности оператора присваивания, Perl сначала вычисляет второе присваивание, вызывая `get_all_clown_hats()` в списочном контексте. Это генерирует список.

Присваивание пустому списоку отбрасывает все значения из списка, но это присваивание происходит в скалярном контексте, что выдаёт количество элементов на правой стороне присваивания. В результате `$count` содержит количество элементов в списке, возвращаемом `get_all_clown_hats()`.

Если сейчас эта концепция кажется вам сбивающей с толку, не бойтесь. Когда вы поймёте, как фундаментальные особенности дизайна Perl взаимодействуют на практике, это будет иметь больше смысла.

Списки

Список — это разделённая запятыми группа из одного или более выражений. Списки могут появляться в исходном коде буквально как значения:

```
my @first_fibs = (1, 1, 2, 3, 5, 8, 13, 21);
```

...как цели присваивания:

```
my ($package, $filename, $line) = caller();
```

...или списки выражений:

```
say name(), ' => ', age();
```

Скобки не *создают* списки. Списки создаёт оператор запятая. Скобки, там, где они присутствуют в этих примерах, группируют выражения для изменения их *приоритета* .

Используйте оператор диапазона для создания списков литералов в компактной форме:

```
my @chars = 'a' .. 'z';
my @count = 13 .. 27;
```

Используйте оператор `qw()` чтобы разбить литеральную строку по пробелам и получить список строк:

```
my @stooges = qw( Larry Curly Moe Shemp Joey Kenny );
```

Списки могут (и это часто происходит) быть результатами выражений, но эти списки не появляются буквально в исходном коде.

Списки и массивы в Perl не взаимозаменяемы. Списки — это значения. Массивы — это контейнеры. Вы можете сохранить список в массиве или преобразовать массив в список, но это разные сущности. Например, индексация в списке всегда происходит в списочном контексте. Индексация в массиве может происходить в скалярном контексте (для единственного элемента) или списочном контексте (для среза):

```
# пока не беспокойтесь о деталях
sub context
{
    my $context = wantarray();

    say defined $context
         ? $context
             ? 'list'
             : 'scalar'
         : 'void';
    return 0;
}

my @list_slice  = (1, 2, 3)[context()];
my @array_slice = @list_slice[context()];
my $array_index = $array_slice[context()];

say context(); # списочный контекст
context();     # пустой контекст
```

Поток управления

Базовый *поток управления* в Perl — прямой. Выполнение программы начинается с начала (первая строка исполняемого файла) и продолжается до конца:

```
say 'At start';
say 'In middle';
say 'At end';
```

Директивы потока управления в Perl изменяют порядок выполнения — что в программе произойдёт следущим — в зависимости от значений их выражений.

Директивы ветвления

Директива `if` выполняет связанное действие только если её условное выражение возвращает *истинное* значение:

```
say 'Hello, Bob!' if $name eq 'Bob';
```

Эта постфиксная форма удобна для простых выражений. Блочная форма группирует несколько выражений воедино:

```
if ($name eq 'Bob')
{
    say 'Hello, Bob!';
    found_bob();
}
```

Блочная форма требует скобок вокруг своего условия, постфиксная же форма — нет:

Условное выражение может состоять из нескольких подвыражений, до тех пор пока оно вычисляется в одно выражение верхнего уровня:

```
if ($name eq 'Bob' && not greeted_bob())
{
    say 'Hello, Bob!';
    found_bob();
}
```

В постфиксной форме добавление скобок может прояснить замысел кода ценой визуальной чистоты:

```
greet_bob() if ($name eq 'Bob' && not greeted_bob());
```

Директива `unless` — отрицательная форма `if`. Perl выполнит действие в случае, если условное выражение возвращает *ложь*:

```
say "You're not Bob!" unless $name eq 'Bob';
```

Как и `if`, `unless` тоже имеет блочную форму, хотя многие программисты избегают её, потому что со сложными условиями она очень быстро становится сложной для чтения:

```
unless (is_leap_year() and is_full_moon())
{
    frolic();
    gambol();
}
```

`unless` очень хорошо работает для постфиксных условий, особенно для валидации параметров в функциях :

```
sub frolic
{
    return unless @_;

    for my $chant (@_) { ... }
}
```

Блочные формы как `if`, так и `unless`, работают с директивой `else`, предоставляющей код для выполнения в случае, если условное выражение не возвращает истину (для `if`) или ложь (для `unless`):

```
if ($name eq 'Bob')
{
    say 'Hi, Bob!';
    greet_user();
}
else
{
    say "I don't know you.";
    shun_user();
}
```

Блоки `else` позволяют переписать условия `if` и `unless` в терминах друг друга:

```
unless ($name eq 'Bob')
{
    say "I don't know you.";
    shun_user();
}
else
{
    say 'Hi, Bob!';
    greet_user();
}
```

Однако, подразумеваемое двойное отрицание при использовании `unless` с блоком `else` может сбивать с толку. Этот пример, возможно, единственное место, где вы его когда-либо увидите.

Perl не только предоставляет вам и `if`, и `unless`, чтобы дать возможность выразить ваши условия наиболее читаемым способом, вы также можете выбирать между положительными и отрицательными условными операторами:

```
if ($name ne 'Bob')
{
    say "I don't know you.";
    shun_user();
}
else
{
    say 'Hi, Bob!';
    greet_user();
}
```

...хотя двойное отрицание, создаваемое присутствием блока `else`, наводит на мысль инвертировать условие.

Одна или более директив `elsif` может следовать за блочной формой `if` и предшествовать единственной директиве `else`:

```
if ($name eq 'Bob')
{
    say 'Hi, Bob!';
    greet_user();
}
elsif ($name eq 'Jim')
{
    say 'Hi, Jim!';
    greet_user();
}
else
{
    say "You're not my uncle.";
```

```
    shun_user();
}
```

Цепочка `unless` тоже может использовать блок `elsif` *(Удачи с расшифровкой этого!)*. Однако, не существует директивы `elseunless`.

Написание `else  if` — синтаксическая ошибка_(Ларри предпочитает `elsif`) исходя из эстетических причин, а также предшествующего искусства языка програмирования Ada._:

```
if ($name eq 'Rick')
{
    say 'Hi, cousin!';
}

# внимание, синтаксическая ошибка
else if ($name eq 'Kristen')
{
    say 'Hi, cousin-in-law!';
}
```

Тернарный условный оператор

Тернарный условный оператор вычисляет условное выражение и возвращает одну из двух альтернатив:

```
my $time_suffix = after_noon($time)
                ? 'afternoon'
                : 'morning';
```

Условное выражение предшествует символу вопросительного знака (`?`), а символ двоеточия (`:`) разделяет альтернативы. Альтернативы — это выражения произвольной сложности, включая другие тернарные условные выражения.

Короткое замыкание Perl проявляет поведение, называемое *коротким замыканием*, когда встречается со сложными условными выражениями. Если Perl может определить, будет ли сложное выражение истинно или ложно, не вычисляя каждое подвыражение, он не будет вычислять последующие подвыражения. Это наиболее очевидно на примере:

```
say "Both true!" if ok( 1, 'subexpression one' )
                 && ok( 1, 'subexpression two' );

done_testing();
```

Возвращаемое значение `ok()` — это булево значение, полученное при вычислении первого аргумента, поэтому этот код выведет:

Если первое подвыражение — первый вызов `ok` — возвращает истинное значение, Perl должен будет вычислить второе подвыражение. Если бы первое подвыражение вернуло ложное значение, то не было бы необходимости проверять последующие подвыражения, так как всё выражение уже не может быть истинным:

```
say "Both true!" if ok( 0, 'subexpression one' )
                 && ok( 1, 'subexpression two' );
```

Этот пример выводит:

Хотя второе подвыражение очевидно вернёт истину, Perl никогда не вычислит его. То же самое короткое замыкание очевидно для операций логического или:

```
say "Either true!" if ok( 1, 'subexpression one' )
                   || ok( 1, 'subexpression two' );
```

Этот пример выводит:

В случае истинности первого подвыражения, Perl может избежать вычисления второго подвыражения. Если бы первое подвыражение было ложным,

результат вычисления второго подвыражения определил бы результат вычисления всего выражения.

Короткое замыкание позволяет вам не только избежать потенциально дорогих вычислений, но и избежать ошибок и предупреждений, как в случае, когда использование неопределённого значения может вызвать предупреждение:

```
my $bbq;
if (defined $bbq and $bbq eq 'brisket') { ... }
```

Контекст в условных директивах

Все условные директивы — `if`, `unless` и тернарный условный оператор — вычисляют выражение в булевом контексте . В то время как операторы сравнения, такие как `eq`, `==`, `ne` и `!=`, возвращают булевы значения при их вычислении, результаты других выражений — включая переменные и значения — Perl преобразует в булевы формы.

В Perl 5 нет единого истинного значения, как и единого ложного. Любое число, преобразуемое в 0, ложно. Это относится к `0`, `0.0`, `0e0`, `0x0` и т. д. Пустая строка (`''`) и `'0'` преобразуются в ложное значение, но строки `'0.0'`, `'0e0'` и т. д. — нет. Идиома `'0 but true'` преобразуется в 0 в числовом контексте, но истинно в булевом контексте, благодаря своему строковому содержанию.

Пустой список и `undef` преобразуются в ложное значение. Пустые массивы и хеши возвращают число 0 в скалярном контексте, поэтому в булевом контексте преобразуются в ложь. Массив, содержащий хотя бы один элемент, пусть даже `undef`, преобразуется в истину в булевом контексте. Хеш, содержащий хотя бы один элемент, даже если и ключ, и значение — `undef`, преобразуется в истинное значение в булевом контексте.

Директивы циклов

Perl предоставляет несколько директив для организации циклов и итераций. Цикл в стиле *foreach* вычисляет выражение, возвращающее список, и выполняет инструкцию или блок, пока не израсходует весь этот список:

```
foreach (1 .. 10)
{
    say "$_ * $_ = ", $_ * $_;
}
```

Этот пример использует оператор диапазона для генерации списка целых чисел от единицы до десяти включительно. Директива `foreach` циклически проходит по ним, устанавливая переменную-топик `$_` в каждое из них по очереди. Perl выполняет блок для каждого целого числа и выводит их квадраты.

Как и `if` и `unless`, этот цикл имеет постфиксную форму:

```
say "$_ * $_ = ", $_ * $_ for 1 .. 10;
```

Цикл `for` может использовать именованную переменную вместо переменной-топика:

```
for my $i (1 .. 10)
{
    say "$i * $i = ", $i * $i;
}
```

Когда цикл `for` использует переменную-итератор, область видимости этой переменной — *внутри* цикла. Perl будет назначать этой лексической переменной значение каждого элемента итерации. Perl не будет модифицировать переменную-топик (`$_`). Если вы объявили лексическую переменную `$i` во внешней области видимости, её значение сохранится снаружи цикла:

```
my $i = 'cow';

for my $i (1 .. 10)
{
    say "$i * $i = ", $i * $i;
}

is( $i, 'cow', 'Value preserved in outer scope' );
```

Эта локализация происходит даже если вы не переобъявляли переменную итерации как лексическую_(—однако *объявляйте* ваши переменные итераций как лексические, чтобы уменьшить их область видимости._:

```
my $i = 'horse';

for $i (1 .. 10)
{
    say "$i * $i = ", $i * $i;
}

is( $i, 'horse', 'Value preserved in outer scope' );
```

Итерации и псевдонимы

Цикл `for` создаёт переменную-итератор как *псевдоним* для значения в итерации, так что любые модификации значения итератора сразу же изменяют исходное значение:

```
my @nums = 1 .. 10;

$_ **= 2 for @nums;

is( $nums[0], 1, '1 * 1 is 1' );
is( $nums[1], 4, '2 * 2 is 4' );

...
```

```
is( $nums[9], 100, '10 * 10 is 100' );
```

Создание псевдонимов также работает с блочным стилем цикла `for`:

```
for my $num (@nums)
{
    $num **= 2;
}
```

...как и в итерациях с переменной-топиком:

```
for (@nums)
{
    $_ **= 2;
}
```

Однако, вы не можете использовать псевдонимы для изменения значения *констант*:

```
for (qw( Huex Dewex Louid ))
{
    $_++;
    say;
}
```

Вместо этого Perl выбросит исключение о модификации значения, предназначенного только для чтения.

Иногда вы можете увидеть использование цикла `for` с единственной скалярной переменной, для создания псевдонима этой переменной в `$_`:

```
for ($user_input)
```

```
{
    s/\A\s+//;      # удалить ведущие пробелы
    s/\s+\z//;      # удалить завершающие пробелы

   $_ = quotemeta; # экранировать символы, не являющиеся словарным
}
```

Итерации и область видимости

Учёт области видимости итератора в случае переменной-топика создаёт общий источник путаницы. Рассмотрим функцию `topic_mangler()`, целенаправленно модифицирующую `$_`. Если код, итерирующий по списку, вызывает `topic_mangler()`, не защитив `$_`, весёлая отладка гарантированна:

```
for (@values)
{
    topic_mangler();
}

sub topic_mangler
{
    s/foo/bar/;
}
```

Если вам *приходится* использовать `$_` вместо именованной переменной, сделайте переменную-топик лексической с помощью объявления `my $_`:

```
sub topic_mangler
{
    # было $_ = shift;
    my $_ = shift;

    s/foo/bar/;
    s/baz/quux/;
```

```
    return $_;
}
```

Использование именованной переменной итерации также предотвращает нежелательное создание псевдонима в `$_`.

Цикл for в стиле C

Цикл `for` в стиле C требует управления условиями итерации:

```
for (my $i = 0; $i <= 10; $i += 2)
{
    say "$i * $i = ", $i * $i;
}
```

Вы должны явно назначить значение итерационной переменной в конструкции цикла, так как этот цикл не выполняет ни создания псевдонима, ни присвоения переменной-топику. Хотя все переменные, объявленные в конструкции цикла, имеют лексическую область видимости в блоке цикла, переменные, объявленные снаружи конструкции цикла, не локализуются:

```
my $i = 'pig';

for ($i = 0; $i <= 10; $i += 2)
{
    say "$i * $i = ", $i * $i;
}

isnt( $i, 'pig', '$i overwritten with a number' );
```

Конструкция цикла может иметь три подвыражения. Первое подвыражение — секция инициализации — выполняется только один раз, перед выполнением тела цикла. Perl вычисляет второе подвыражение — условное сравнение — перед каждой итерацией тела цикла. Если оно возвращает истинное

значение, выполняется итерация. Если оно возвращает ложное значение, итерация прекращается. Последнее подвыражение выполняется после каждой итерации тела цикла.

```
for (
    # подвыражение инициализации цикла
    say 'Initializing', my $i = 0;

    # подвыражение условного сравнения
    say "Iteration: $i" and $i < 10;

    # подвыражение завершения итерации
    say 'Incrementing ' . $i++
)
{
    say "$i * $i = ", $i * $i;
}
```

Обратите внимание на отсутствие точки с запятой после последнего подвыражения, а также на использование оператора запятой и низкоприоритетного `and`; этот синтаксис на удивление привередлив. По возможности, предпочтите использование цикла в стиле `foreach`, а не `for`.

Все три подвыражения необязательны. Бесконечный цикл `for` может выглядеть так:

```
for (;;) { ... }
```

While и Until

Цикл *while* продолжается до тех пор, пока условие цикла не вернёт ложное булево значение. Идиоматический бесконечный цикл выглядит так:

```
while (1) { ... }
```

В отличие от итерации в цикле стиля `foreach`, условие цикла `while` само по себе не имеет побочных эффектов. То есть, если `@values` содержит один или несколько элементов, этот код тоже будет бесконечным циклом:

```
while (@values)
{
    say $values[0];
}
```

Чтобы предотвратить подобный бесконечный цикл `while`, используйте *деструктивное изменение* массива `@values`, модифицируя массив на каждой итерации цикла:

```
while (@values)
{
    my $value = shift @values;
    say $value;
}
```

Модификация `@values` внутри условия `while` тоже работает, но имеет ряд тонкостей, связанных с истинностью каждого значения.

```
while (my $value = shift @values)
{
    say $value;
}
```

Этот цикл прекратится, как только он достигнет элемента, являющегося ложным значением, а не обязательно, когда весь массив будет израсходован. Это может быть ожидаемым поведением, но часто удивляет новичков.

Цикл *until* меняет на противоположный смысл проверки цикла `while`. Итерация продолжается, пока условное выражение цикла возвращает ложное значение:

```
until ($finished_running)
{
    ...
}
```

Каноническое использование цикла `while` — итерация по входным данным из дескриптора файла:

```
use autodie;

open my $fh, '<', $file;

while (<$fh>)
{
    ...
}
```

Perl 5 интерпретирует этот цикл `while` как если бы вы написали:

```
while (defined($_ = <$fh>))
{
    ...
}
```

Без явного `defined`, любая строка, прочитанная из дескриптора файла, которая в скалярном контексте даёт ложное значение — пустая строка, или строка, содержащая один только символ `0` — завершит цикл. Оператор `readline` (`<>`) возвращает неопределённое значение только когда достигает конца файла.

И `while`, и `until` имеют постфиксные формы, такие как бесконечный цикл `1 while 1;`. Любое одиночное выражение годится для постфиксной формы `while` или `until`, включая классический пример «Hello, world!» из 8-битных компьютеров ранних 1980-ых:

```
print "Hello, world!  " while 1;
```

Бесконечные циклы более полезны, чем кажется на первый взгляд, особенно для петель событий в программах с графическим интерфейсом, интерпретаторах или сетевых серверах:

```
$server->dispatch_results() until $should_shutdown;
```

Используйте блок `do`, чтобы сгруппировать несколько выражений воедино:

```
do
{
    say 'What is your name?';
    my $name = <>;
    chomp $name;
    say "Hello, $name!" if $name;
} until (eof);
```

Блок `do` воспринимается парсером как единое выражение, которое может содержать несколько других выражений. В отличие от блочной формы цикла `while`, блок `do` с постфиксным `while` или `until` выполнит своё тело хотя бы раз. Эта конструкция менее распространена, чем другие формы циклов, но не менее действенна.

Циклы внутри циклов

Вы можете вкладывать циклы друг в друга:

```
for my $suit (@suits)
{
    for my $values (@card_values) { ... }
}
```

Когда вы это делаете, объявляйте именованные итерационные переменные! В противном случае вероятность возникновения путаницы с переменной-топиком и её областью видимости слишком высока.

Распространённая ошибка при вложении циклов `foreach` и `while` заключается в том, что легко израсходовать дескриптор файла в цикле `while`:

```
use autodie;

open my $fh, '<', $some_file;

for my $prefix (@prefixes)
{
    # НЕ ИСПОЛЬЗУЙТЕ; с большой вероятностью ошибочный код
     while (<$fh>)
     {
         say $prefix, $_;
     }
}
```

Открытие дескриптора файла снаружи цикла `for` сохраняет позицию в файле между итерациями цикла `for`. На его второй итерации циклу `while` будет нечего читать, и он не будет выполнен. Чтобы разрешить эту проблему, переоткрывайте файл внутри цикла `for` (просто для понимания, но не всегда хорошее использование системных ресурсов), считайте весь файл в память (что может не сработать, если файл большой) или сбрасывайте дескриптор файла назад к началу файла на каждой итерации с помощью функции `seek` (опция, которую зачастую упускают):

```
use autodie;

open my $fh, '<', $some_file;

for my $prefix (@prefixes)
{
    while (<$fh>)
    {
        say $prefix, $_;
    }

    seek $fh, 0, 0;
```

```
}
```

Контроль циклов

Иногда вам нужно разорвать цикл раньше, чем условия итерации будут исчерпаны. Стандартные механизмы управления Perl 5 — исключения и `return` — работают, но кроме того вы можете использовать операторы *контроля цикла.*

Оператор *next* перезапускает цикл со следующей его итерации. Используйте его, если вы сделали всё, что нужно, в текущей итерации. Для цикла по строкам в файле с пропуском любой строки, начинающейся с символа комментария #, напишите:

```
while (<$fh>)
{
    next if /\A#/;
    ...
}
```

Оператор *last* немедленно завершает цикл. Чтобы закончить обработку файла, как только вы достигли признака завершения, напишите:

```
while (<$fh>)
{
    next if /\A#/;
    last if /\A__END__/
    ...
}
```

Оператор *redo* перезапускает текущую итерацию без повторного вычисления условия. Это может быть полезно в тех немногих случаях, когда вы хотите сразу же изменить прочитанную строку, а затем начать обработку с начала, не затирая её следующей строкой. Так можно реализовать простой парсер файла, который объединяет строки, завершающиеся обратным слешем:

```
while (my $line = <$fh>)
{
    chomp $line;

    # поиск обратного слеша в конце строки
    if ($line =~ s{\\$}{})
    {
        $line .= <$fh>;
        chomp $line;
        redo;
    }

    ...
}
```

Использование операторов контроля цикла во вложенных списках может сбивать с толку. Если вы не можете избежать вложенных циклов — выделив внутренние циклы в именованные функции — используйте *метку цикла* для прояснения:

```
LINE:
while (<$fh>)
{
    chomp;

    PREFIX:
    for my $prefix (@prefixes)
    {
        next LINE unless $prefix;
        say "$prefix: $_";
        # здесь подразумевается next PREFIX
    }
}
```

Continue

Конструкция `continue` ведёт себя как третье подвыражение цикла `for`; Perl выполняет её блок перед следующей итерацией цикла, как при нормальном повторении цикла, так и при преждевременном повторе итерации с помощью `next`*(Эквивалент `continue` языка C в Perl — это `next`.*. Вы можете использовать её с циклами `while`, `until`, `when` или `for`. Примеры употребления редки, но она полезна, если вам нужна гарантия того, что нечто произойдёт на каждой итерации цикла, независимо от того, как заканчивается итерация:

```
while ($i < 10 )
{
    next unless $i % 2;
    say $i;
}
continue
{
    say 'Continuing...';
    $i++;
}
```

Имейте ввиду, что блок `continue` *не* выполняется, когда поток управления выходит из цикла вследствие срабатывания `last` или `redo`.

given/when

Конструкция `given` — новая возможность Perl 5.10. Она присваивает значение выражения переменной-топику и предваряет блок:

```
given ($name) { ... }
```

В отличие от `for`, она не итерирует по агрегатной переменной. Она вычисляет своё выражение в скалярном контексте и всегда осуществляет присваивание переменной-топику:

```
given (my $username = find_user())
{
    is( $username, $_, 'topic auto-assignment' );
}
```

`given` также локализует переменную-топик:

```
given ('mouse')
{
    say;
    mouse_to_man( $_ );
    say;
}

sub mouse_to_man { s/mouse/man/ }
```

`given` наиболее полезна при использовании совместно с `when` . `given` *топикализирует* значение внутри блока, так что несколько инструкций `when` могут проверять соответствие топика выражениям, используя семантику *умного сопоставления*. Так можно написать игру «камень, ножницы, бумага».

```
my @options  = ( \&rock, \&paper, \&scissors );
my $confused = "I don't understand your move.";

do
{
    say "Rock, Paper, Scissors!  Pick one: ";
    chomp( my $user = <STDIN> );
    my $computer_match = $options[ rand @options ];
    $computer_match->( lc( $user ) );
} until (eof);

sub rock
{
    print "I chose rock.  ";

    given (shift)
```

```
    {
        when (/paper/)    { say 'You win!' };
        when (/rock/)     { say 'We tie!'  };
        when (/scissors/) { say 'I win!'   };
        default           { say $confused  };
    }
}

sub paper
{
    print "I chose paper.  ";

    given (shift)
    {
        when (/paper/)    { say 'We tie!'  };
        when (/rock/)     { say 'I win!'   };
        when (/scissors/) { say 'You win!' };
        default           { say $confused  };
    }
}

sub scissors
{
    print "I chose scissors.  ";

    given (shift)
    {
        when (/paper/)    { say 'I win!'   };
        when (/rock/)     { say 'You win!' };
        when (/scissors/) { say 'We tie!'  };
        default           { say $confused  };
    }
}
```

Perl выполняет правило `default`, если ни одно из других условий не подошло.

Хвостовые вызовы

Хвостовой вызов происходит, когда последнее выражение в функции — вызов другой функции: возвращаемое значение внутренней функции будет возвращаемым значением внешней функции:

```
sub log_and_greet_person
{
    my $name = shift;
    log( "Greeting $name" );

    return greet_person( $name );
}
```

Возврат из `green_person()` напрямую в код, вызывающий `log_and_greet_person()`, более эффективен, чем возврат *в* `log_and_greet_person()` и сразу же возврат *из* `log_and_greet_person()`. Возврат напрямую *из* `greet_person()` в код, вызывающий `log_and_greet_person()` — *оптимизация хвостового вызова.*

Код с тяжёлой рекурсией , особенно взаимнорекурсивный код, может потреблять много памяти. Хвостовые вызовы уменьшают количество памяти, требуемое для внутренней организации потока управления, и делают дорогие алгоритмы легко обрабатываемыми. К сожалению, Perl 5 не выполняет эту автоматизацию автоматически; вы должны сделать это сами, если необходимо.

Встроенный оператор `goto` имеет форму, которая вызывает функцию, как если бы текущая функция никогда не была вызвана, по существу, обнуляя управление ресурсами для вызова новой функции. Уродливый синтаксис смущает людей, на слуху у которых «никогда не используйте `goto`», но это работает:

```
sub log_and_greet_person
{
    my ($name) = @_;
    log( "Greeting $name" );
```

```
    goto &greet_person;
}
```

Этот пример имеет две важных особенности. Во-первых, `goto &function_name` или `goto &$function_reference` требует использования сигила функции (&), чтобы парсер знал, что нужно выполнить хвостовой вызов вместо перехода к метке. Во-вторых, эта форма вызова функции неявно передаёт содержимое `@_` в вызываемую функцию. Вы можете модифицировать `@_` для изменения передаваемых аргументов.

Эта техника относительно редка; она наиболее полезна, когда вы хотите вмешаться в поток управления, чтобы убраться с пути других функций, проверяющих `caller` (например, если вы реализуете специальное логирование или какую-нибудь возможность отладки), или когда используете алгоритм, требующий большого количества рекурсии.

Скаляры

Основной тип данных Perl 5 — это *скаляр*, одиночное, отдельное значение. Значение может быть строкой, целым числом, значением с плавающей точкой, дескриптором файла или ссылкой — но это всегда одиночное значение. Скаляры могут быть лексическими переменными, переменными пакета или глобальными переменными. Вы можете объявлять только лексические переменные и переменные пакетов. Имена скалярных переменных должны соответствовать стандартным правилам именования переменных . Скалярные переменные всегда начинаются с сигила в виде знака доллара ($).

Скаляры и типы

Скалярная переменная может содержать любой тип скалярного значения без специальной конвертации или приведения, и тип значения, сохранённого в переменной, может меняться:

```
my $value;
$value = 123.456;
$value = 77;
$value = "I am Chuck's big toe.";
$value = Store::IceCream->new();
```

Хотя этот код *допустим*, изменение типа данных, сохраняемых в скаляре, — признак беспорядка.

Эта гибкость типов часто приводит к приведению типа значения . Например, вы можете обращаться с содержимым скаляра как со строкой, даже если вы явно не назначали ему строку:

```
my $zip_code       = 97006;
my $city_state_zip = 'Beaverton, Oregon' . ' ' . $zip_code;
```

Также вы можете производить математические операции над строками:

```
my $call_sign = 'KBMIU';

# обновить знак и вернуть новое значение
my $next_sign = ++$call_sign;

# вернуть старое значение, I<затем> обновить знак
my $curr_sign = $call_sign++;

# но I<не работает> в таком виде:
my $new_sign  = $call_sign + 1;
```

Эта операция инкремента строки превращает `a` в `b` и `z` в `aa`, учитывая набор символов и регистр. В то время как `ZZ9` становится `AAA0`, `ZZ09` становится `ZZ10` — числа проворачиваются до тех пор, пока есть большие значащие цифры для инкремента, как на автомобильном одометре.

Вычисление ссылки в строковом контексте возвращает строку. Вычисление ссылки в числовом контексте возвращает число. Ни одна из этих операций

не модифицирует ссылку, но вы не сможете восстановить ссылку из любого из результатов:

```
my $authors     = [qw( Pratchett Vinge Conway )];
my $stringy_ref = '' . $authors;
my $numeric_ref =  0 + $authors;
```

`$authors` — всё ещё пригодная к употреблению ссылка, но `$stringy_ref` — строка, не имеющая никакой связи со ссылкой, и `$numeric_ref` — число, также не имеющее никакой связи со ссылкой.

Чтобы обеспечить возможность преобразования без потери данных, скаляры Perl 5 могут содержать и числовой, и строковый компоненты. Внутренняя структура данных, представляющая скаляр в Perl 5, имеет числовой слот и строковый слот. Доступ к строке в числовом контексте производит скаляр и со строковым, и с числовым значениями. Функция `dualvar()` базового модуля `Scalar::Util` позволяет вам манипулировать обоими значениями напрямую в одном скаляре.

Скаляры не имеют отдельного слота для булевых значений. В булевом контексте пустая строка (`''`) и `'0'` — ложны. Все остальные строки — истинны. В булевом контексте числа, равные нулю (`0`, `0.0` и `0e0`), ложны. Все остальные числа — истинны.

Ещё одно значение всегда ложно: `undef`. Это значение неинициализированных переменных, кроме того, оно может быть и самостоятельным значением.

Массивы

Массивы в Perl 5 — это структуры данных *первого класса* — язык поддерживает их как встроенный тип данных — которые хранят ноль или больше скаляров. Вы можете получить доступ к отдельным элементам массива по целочисленным индексам, а также можете по желанию добавлять или удалять элементы. Сигил `@` обозначает массив. Объявить массив можно так:

```
my @items;
```

Элементы массива

Доступ к отдельному элементу массива в Perl 5 требует использования скалярного сигила. `$cats[0]` — однозначное использование массива `@cats`, потому что постфиксные квадратные скобки (`[]`) всегда означают доступ к массиву по индексу.

Первый элемент массива имеет индекс ноль:

```
# @cats содержит список объектов Cat
my $first_cat = $cats[0];
```

Последний индекс массива зависит от количества элементов в нём. Массив в скалярном контексте (в скалярном присваивании, строковой конкатенации, сложении или булевом контексте) возвращает количество элементов в массиве:

```
# скалярное присваивание
my $num_cats = @cats;

# строковая конкатенация
say 'I have ' . @cats . ' cats!';

# сложение
my $num_animals = @cats + @dogs + @fish;

# булев контекст
say 'Yep, a cat owner!' if @cats;
```

Чтобы получить *индекс* последнего элемента массива, вычтите единицу из количества элементов в массиве (помните, что индексация массива начинается с 0) или испольуйте неуклюжий синтаксис `$#cats`:

```
my $first_index = 0;
my $last_index  = @cats - 1;
```

```
# or
# my $last_index = $#cats;

say   "My first cat has an index of $first_index, "
    . "and my last cat has an index of $last_index."
```

Когда индекс менее важен, чем позиция элемента, используйте взамен отрицательные индексы. Последний элемент массива доступен по индексу `-1`. Второй с конца элемент доступен по индексу `-2` и т. д.:

```
my $last_cat           = $cats[-1];
my $second_to_last_cat = $cats[-2];
```

`$#` имеет другое применение: изменение размера массива путём *присваивания* ему. Помните, что массивы в Perl 5 могут изменяться. Они расширяются или сокращаются по необходимости. Если вы сжимаете массив, Perl отбросит значения, не помещающиеся в массив с изменённым размером. Если вы расширяете массив, Perl заполнит добавленные позиции значением `undef`.

Присваивание массивам

Присваивайте отдельной позиции в массиве напрямую, используя индекс:

```
my @cats;
$cats[3] = 'Jack';
$cats[2] = 'Tuxedo';
$cats[0] = 'Daisy';
$cats[1] = 'Petunia';
$cats[4] = 'Brad';
$cats[5] = 'Choco';
```

Если вы присвоите индексу, выходящему за пределы текущего размера массива, Perl расширит массив для соответсвия новому размеру и заполнит все промежуточные позиции значением `undef`. После первого присвоения массив будет содержать `undef` в позициях 0, 1 и 2, и `Jack` в позиции 3.

Для сокращения присваивания, инициализируйте массив из списка:

```
my @cats = ( 'Daisy', 'Petunia', 'Tuxedo', ... );
```

...но помните, что эти скобки *не* создают список. Без скобок произошло бы присваивание `Daisy` первому и единственному элементу массива, ввиду приоритета операторов .

Любое выражение, возвращающее список в списочном контексте, может быть присвоено массиву:

```
my @cats     = get_cat_list();
my @timeinfo = localtime();
my @nums     = 1 .. 10;
```

Присваивание скалярному элементу массива налагает скалярный контекст, тогда как присваивание всему массиву целиком налагает списочный контекст.

Чтобы очистить массив, присвойте ему пустой список:

```
my @dates = ( 1969, 2001, 2010, 2051, 1787 );
...
@dates    = ();
```

Операции над массивами

Иногда массив удобнее использовать как упорядоченную, изменяемую коллекцию элементов, чем соответствие индексов значениям. Perl 5 предоставляет несколько операций для манипулирования элементами массива без использования индексов:

Операторы `push` и `pop` добавляют и удаляют элементы из конца массива соответственно:

```
my @meals;

# что есть съедобного?
push @meals, qw( hamburgers pizza lasagna turnip );

# E<hellip>но ваш племянник терпеть не может овощи
pop @meals;
```

Вы можете добавить в массив список значений с помощью `push`, но с помощью `pop` можете удалить только один за раз. `push` возвращает новое количество элементов в массиве. `pop` возвращает удалённый элемент.

Поскольку `push` оперирует списками, вы легко можете добавить элементы одного или нескольких массивов в другой:

```
push @meals, @breakfast, @lunch, @dinner;
```

Аналогично, `unshift` и `shift` добавляют элементы или удаляют элемент из начала массива соответственно:

```
# расширить наши кулинарные горизонты
unshift @meals, qw( tofu spanakopita taquitos );

# пересмотрим всю эту идею с соей
shift @meals;
```

`unshift` вставляет список элементов в начало массива и возвращает новое количество элементов в массиве. `shift` удаляет и возвращает первый элемент массива.

Немногие программы используют возвращаемые значения `push` и `unshift`.

Оператор `splice` удаляет и заменяет элементы в массиве в соответствии с заданным смещением, длинной вырезаемого списка и списком для замены. И замена, и удаление — опциональны; вы можете опустить любое из этих поведений. Описание `splice` в `perlfunc` демонстрирует его

равноценность с `push`, `pop`, `shift` и `unshift`. Одно из возможных применений — удаление двух элементов из массива:

```
my ($winner, $runnerup) = splice @finalists, 0, 2;

# или
my $winner              = shift @finalists;
my $runnerup            = shift @finalists;
```

До Perl 5.12 итерация по массиву с использованием индекса требовала цикла в стиле C. Начиная с Perl 5.12, `each` может итерировать по массиву по индексу и значению:

```
while (my ($index, $value) = each @bookshelf)
{
    say "#$index: $value";
    ...
}
```

Срезы массивов

Срез массива позволяет вам получить доступ к элементам массива в списочном контексте. В отличие от скалярного доступа к элементу массива, эта операция принимает список из ноля или более индексов и использует сигил массива (@):

```
my @youngest_cats = @cats[-1, -2];
my @oldest_cats   = @cats[0 .. 2];
my @selected_cats = @cats[ @indexes ];
```

Срезы массивов удобны для присваивания:

```
@users[ @replace_indices ] = @replace_users;
```

Срез может содержать ноль или более элементов — включая один:

```
# одноэлементный срез массива; I<списочный> контекст
@cats[-1] = get_more_cats();

# доступ к одному элементу массива; I<скалярный> контекст
$cats[-1] = get_more_cats();
```

Единственное синтаксическое различие между срезом массива из одного элемента и скалярным доступом к элементу массива — предшествующий сигил. *Семантическая* разница намного больше: срез массива всегда налагает списочный контекст. Срез массива, используемый в скалярном контексте, вызовет предупреждение:

Срез массива налагает списочный контекст на выражение, используемое в качестве его индекса:

```
# функция вызывается в списочном контексте
my @hungry_cats = @cats[ get_cat_indices() ];
```

Массивы и контекст

В списочном контексте массивы разглаживаются в списки. Если вы передадите несколько массивов нормальной функции в Perl 5, они разгладятся в единый список:

```
my @cats = qw( Daisy Petunia Tuxedo Brad Jack );
my @dogs = qw( Rodney Lucky );

take_pets_to_vet( @cats, @dogs );

sub take_pets_to_vet
{
    # ГЛЮЧНО: не использовать!
    my (@cats, @dogs) = @_;
```

```
    ...
}
```

Внутри функции `@_` будет содержать семь элементов, не два, потому что списочное присваивание массивам обладает *жадностью*. Массив поглотит столько элементов из списка, сколько возможно. После присваивания `@cats` будет содержать *все* аргументы, переданные в функцию. `@dogs` будет пустым.

Это разглаживающиее поведение иногда ставит в тупик новичков, которые пытаются создать вложенные массивы в Perl 5:

```
# создаёт один массив, не массив массивов
my @numbers = (  1 .. 10,
              ( 11 .. 20,
              ( 21 .. 30 ) ) );
```

...но этот код, в сущности, то же самое, что и:

```
# создаёт один массив, не массив массивов
my @numbers = 1 .. 30;
```

...потому что скобки всего лишь группируют выражения. Они не *создают* списки в этой ситуации. Чтобы избежать этого разглаживающего поведения, используйте ссылки на массивы .

Интерполяция массивов

Массивы интерполируются в строках как списки всех приведённых в строковый вид элементов, разделённых текущим значением магической глобальной переменной `$"`. Значение по умолчанию этой переменной — один пробел. Её мнемоника в `English.pm` — `$LIST_SEPARATOR`. Поэтому получаем следующее:

```
my @alphabet = 'a' .. 'z';
say "[@alphabet]";
[a b c d e f g h i j k l m
 n o p q r s t u v w x y z]
```

Локализуйте `$"` своим разделителем чтобы облегчить отладку_(Благодарность за эту технику уходит Марку Джейсону Доминусу (Mark Jason Dominus).)_:

```
# так что там в этом массиве?
local $" = ')(';
say "(@sweet_treats)";
(pie)(cake)(doughnuts)(cookies)(raisin bread)
```

Хеши

Хеш — структура данных первого класса в Perl, которая ассоциирует строковые ключи со скалярными значениями. Так же как имя переменной соотносится с местом хранения, так и ключ в хеше ссылается на значение. Воспринимайте хеш как телефонную книгу: используйте имена ваших друзей для поиска их номеров. В других языках хеши называются *таблицами*, *ассоциативными массивами*, *словарями* или *картами*.

Хеши имеют два важных свойства: они хранят один скаляр на уникальный ключ, и они не имеют определённой сортировки ключей.

Объявление хешей

Хеши используют сигил %. Объявляются хеши так:

```
my %favorite_flavors;
```

Хеши создаются пустыми. Вы можете написать `my %favorite_flavors = ()`, но это излишне.

Хеши используют скалярный сигил `$` для доступа к отдельным элементам и фигурные скобки `{ }` для доступа по ключам:

```
my %favorite_flavors;
$favorite_flavors{Gabi}    = 'Raspberry chocolate';
$favorite_flavors{Annette} = 'French vanilla';
```

Присваивание хешу списка ключей и значений в одном выражении:

```
my %favorite_flavors = (
    'Gabi',    'Raspberry chocolate',
    'Annette', 'French vanilla',
);
```

Если вы присвоите нечётное количество элементов хешу, то получите предупреждение. Идиоматический Perl часто использует оператор *толстая запятая* (`=>`) для установки соответствия значений ключам, так как это делает разбиение на пары более заметным визуально:

```
my %favorite_flavors = (
    Gabi    < = >> 'Mint chocolate chip',
    Annette < = >> 'French vanilla',
);
```

Оператор толстая запятая ведёт себя как обычная запятая, но кроме того он автоматически заключает в кавычки предшествующее голое слово . Прагма `strict` не будет предупреждать о таком голом слове — и если у вас есть функция с таким же именем, как ключ хеша, толстая запятая *не* будет вызывать функцию:

```
sub name { 'Leonardo' }

my %address =
(
```

```
    name => '1123 Fib Place',
);
```

Ключом хеша будет `name`, а не `Leonardo`. Чтобы вызвать функцию, сделайте её вызов явным:

```
my %address =
(
    name() => '1123 Fib Place',
);
```

Для очистки хеша присвойте ему пустой список_(Иногда вам может встретиться `undef %hash`)._:

```
%favorite_flavors = ();
```

Индексация хеша

Доступ к отдельному значению хеша осуществляется с помощью операции индексации. Используйте ключ (*доступ по ключу*) для получения значения из хеша:

```
my $address = $addresses{$name};
```

В этом примере `$name` содержит строку, которая также является ключом хеша. Как и при доступе к отдельному элементу массива, сигил хеша изменился с `%` на `$` для обозначения доступа по ключу к скалярному значению.

Вы можете использовать строковые литералы как ключи хеша. Perl автоматически заключит голые слова в кавычки в соответствии с теми же правилами, что и толстые запятые:

```
# автоматическое заключение в кавычки
my $address = $addresses{Victor};

# требует заключения в кавычки; не допустимое голое слово
my $address = $addresses{'Sue-Linn'};

# вызов функции требует устранения неоднозначности
my $address = $addresses{get_name()};
```

Даже на встроенные функции Perl 5 распространяется автоматическое заключение в кавычки:

```
my %addresses =
(
    Leonardo => '1123 Fib Place',
    Utako    => 'Cantor Hotel, Room 1',
);

sub get_address_from_name
{
    return $addresses{+shift};
}
```

Унарный плюс превращает то, на что как на голое слово (`shift`) должны распространяться правила автоматического заключения в кавычки, в выражение. Как из этого следует, вы можете использовать произвольное выражение — не только вызов функции — как ключ хеша:

```
# хотя не стоит этого I<делать> на самом деле
my $address = $addresses{reverse 'odranoeL'};

# интерполяция допустима
my $address = $addresses{"$first_name $last_name"};

# как и вызовы методов
my $address = $addresses{ $user->name() };
```

Ключи хешей могут быть только строками. Всё, что вычисляется в строку, является допустимым ключом хеша. Perl пойдёт так далеко, что приведёт любую не-строку в строку, так что если вы используете объект как ключ хеша, вы получите строковую версию этого объекта вместо самого объекта:

```
for my $isbn (@isbns)
{
    my $book = Book->fetch_by_isbn( $isbn );

    # вряд ли сделает именно то, чего вы хотите
    $books{$book} = $book->price;
}
```

Существование ключа хеша

Оператор `exists` возвращает булево значение, указывающее, содержит ли хеш указанный ключ:

```
my %addresses =
(
    Leonardo => '1123 Fib Place',
    Utako    => 'Cantor Hotel, Room 1',
);

say "Have Leonardo's address"
    if exists $addresses{Leonardo};
say "Have Warnie's address"
    if exists $addresses{Warnie};
```

Использование `exists` вместо прямого доступа к ключу хеша позволяет избежать двух проблем. Во-первых, оно не проверяет булеву природу *значения* хеша; ключ хеша может существовать и иметь значение, даже если это значение является ложью (включая `undef`):

```
my  %false_key_value = ( 0 => '' );
```

```
ok( %false_key_value,
     'hash containing false key & value
      should evaluate to a true value' );
```

Во-вторых, `exists` избегает автовивификации во вложенных структурах данных .

Если ключ хеша существует, его значением может быть `undef`. Проверьте это с помощью `defined`:

```
$addresses{Leibniz} = undef;

say "Gottfried lives at $addresses{Leibniz}"
    if exists  $addresses{Leibniz}
    && defined $addresses{Leibniz};
```

Доступ к ключам и значениям хеша

Хеши — агрегатные переменные, но их парная природа даёт намного больше возможностей для итераций: по ключам хеша, по значениям хеша, или по парам ключей и значений. Оператор `keys` возвращает список ключей хеша:

```
for my $addressee (keys %addresses)
{
    say "Found an address for $addressee!";
}
```

Оператор `values` возвращает список значений хеша:

```
for my $address (values %addresses)
{
    say "Someone lives at $address";
}
```

Оператор `each` возвращает список двухэлементных списков из ключа и значения:

```
while (my ($addressee, $address) = each %addresses)
{
    say "$addressee lives at $address";
}
```

В отличие от массивов, в этих списках нет очевидной сортировки. Сортировка зависит от внутренней реализации хеша, конкретной используемой вами версии Perl, размера хеша и элемента случайности. Несмотря на это, порядок элементов хеша согласован для `keys`, `values` и `each`. Модификация хеша может изменять этот порядок, но вы можете на него полагаться, если хеш остаётся тем же самым.

Каждый хеш имеет тольк *один* итератор для оператора `each`. Вы не можете надёжно итерировать по хешу с помощью `each` более одного раза; если вы начнёте новую итерацию, когда другая ещё не завершена, предыдущая преждевременно закончится, а последующая начнётся с середины хеша. Во время таких итераций остерегайтесь вызывать какие-либо функции, которые сами могут итерировать по хешу посредством `each`.

На практике такое случается редко, но если вам это нужно, сбросить итератор хеша можно с помощью `keys` или `values` в пустом контексте:

```
# сброс итератора хеша
keys %addresses;

while (my ($addressee, $address) = each %addresses)
{
    ...
}
```

Срезы хешей

Срез хеша — это список ключей и значений хеша, индексированных в одной операции. Так можно инициализировать несколько элементов хеша сразу:

```
# %cats уже содержит элементы
@cats{qw( Jack Brad Mars Grumpy )} = (1) x 4;
```

Это эквивалентно следующей инициализации:

```
my %cats = map { $_ => 1 }
           qw( Jack Brad Mars Grumpy );
```

...за исключением того, что инициализация среза хеша не *заменяет* существующее содержимое хеша.

Срезы хеша также позволяют вам получить несколько значений из хеша в одной операции. Как и в случае срезов массива, сигил хеша изменяется для обозначения списочного контекста. Использование фигурных скобок указывает на доступ по ключу и делает хеш однозначным:

```
my @buyer_addresses = @addresses{ @buyers };
```

Срезы хешей облегчают слияние двух хешей:

```
my %addresses        = ( ... );
my %canada_addresses = ( ... );

@addresses{ keys   %canada_addresses }
          = values %canada_addresses;
```

Это эквивалентно ручному циклу по содержимому `%canada_addresses`, но намного короче.

Что если один и тот же ключ имеет место в обоих хешах? Подход со срезом хеша всегда *перезаписывает* существующие пары ключ/значение в `%addresses`. Если вам нужно другое поведение, больше подойдут циклы.

Пустой хеш

Пустой хеш не содержит ни ключей, ни значений. Он даёт ложное значение в булевом контексте. Хеш, содержащий хотя бы одну пару ключ/значение, в булевом контексте даёт истинное значение, даже если все ключи, или все значения, или и то и другое, сами по себе в булевом контексте являются ложными значениями.

```
use Test::More;

my %empty;
ok( ! %empty, 'empty hash should evaluate false' );

my %false_key = ( 0 => 'true value' );
ok( %false_key, 'hash containing false key
                 should evaluate to true' );

my %false_value = ( 'true key' => 0 );
ok( %false_value, 'hash containing false value
                   should evaluate to true' );

done_testing();
```

В скалярном контексте хеш возвращает строку, представляющую соотношение полных корзин в хеше — внутренние детали реализации хешей, которые вы смело можете игнорировать.

В списочном контексте хеш возвращает список пар ключ/значение, подобный тому, что вы получаете от оператора `each`. Однако вы *не можете* итерировать по этому списку так же, как итерируете по списку, выдаваемому `each`, иначе цикл никогда не прервётся:

```
# бесконечный цикл для непустых хешей
while (my ($key, $value) = %hash)
{
    ...
}
```

Вы *можете* осуществлять цикл по ключам и значениям с помощью цикла `for`, но переменная-итератор будет получать ключ на одной итерации и его значение на следующей, потому что Perl разгладит хеш в один список перемежающихся ключей и значений.

Идиомы с хешами

Поскольку каждый ключ в хеше встречается только один раз, присваивание того же самого ключа хешу несколько раз сохраняет только самый последний ключ. Используйте это для нахождения уникальных элементов списка:

```
my %uniq;
undef @uniq{ @items };
my @uniques = keys %uniq;
```

Использование `undef` со срезом хеша устанавливает значения хеша в `undef`. Эта идиома — самый дешёвый способ выполнить операции установки значения с хешем.

Хеши также полезны для подсчёта элементов, таких как IP-адреса в лог-файле:

```
my %ip_addresses;

while (my $line = <$logfile>)
{
    my ($ip, $resource) = analyze_line( $line );
    $ip_addresses{$ip}++;
    ...
}
```

Начальное значение значений хеша — `undef`. Оператор постфиксного инкремента (`++`) воспринимает его как ноль. Модификация значения инкрементирует существующее значения для этого ключа. Если значения для этого ключа не существует, Perl создаёт значение (`undef`) и немедленно

инкрементирует его до единицы, так как преобразование `undef` в число даёт значение 0.

Эта стратегия обеспечивает полезный механизм кеширования для сохранения результата дорогих операций с небольшим оверхедом:

```
{
    my %user_cache;

    sub fetch_user
    {
        my $id = shift;
        $user_cache{$id} //= create_user($id);
        return $user_cache{$id};
    }
}
```

Этот _орочий манёвр__(Or-cache, orcish, если вы любите игру слов.)_- возвращает значение из хеша, если оно существует. В противном случае, он вычисляет, кеширует и возвращает значение. Оператор определено-или-присвоить (`//=`) вычисляет свой левый операнд. Если этот операнд не определён, оператор присваивает левому значению значение своего правого операнда. Другими словами, если в хеше нет значения для заданного ключа, эта функция вызовет `create_user()` с этим ключом и обновит хеш.

Операторы определено-или и определено-или-присвоить были представлены в Per 5.10. До 5.10 большая часть кода использовала оператор или-присвоить (`||=`) для этой цели. К сожалению, некоторые валидные значения дают ложное значение в булевом контексте, так что вычисление *определённости* значений почти всегда более правильно. Этот ленивый орочий манёвр проверяет определённость кешированного значения, не истинность.

Если ваша функция принимает несколько аргументов, используйте захватывающий хеш для сбора пар ключ/значение в один хеш как именованных аргументов функции:

```
sub make_sundae
{
```

```
    my %parameters = @_;
    ...
}

make_sundae( flavor  => 'Lemon Burst',
             topping => 'cookie bits' );
```

Этот подход позволяет вам устанавливать значения по умолчанию:

```
sub make_sundae
{
    my %parameters           = @_;
    $parameters{flavor}    //= 'Vanilla';
    $parameters{topping}   //= 'fudge';
    $parameters{sprinkles} //= 100;
    ...
}
```

…или включать их в инициализацию хеша, так как последующее присваивание перезаписывает предыдущие:

```
sub make_sundae
{
    my %parameters =
    (
        < flavor    = 'Vanilla', >>
        < topping   = 'fudge', >>
        < sprinkles = 100, >>
        @_,
    );
    ...
}
```

Блокировка хешей

Так как ключи хешей — голые слова, они мало защищены от опечаток по сравнению с защитой имён функций и переменных, предлагаемой прагмой `strict`. Редко используемый базовый модуль `Hash::Util` предоставляет механизмы для улучшения этой ситуации.

Чтобы не дать кому-либо случайно добавить ключ хеша, который вы не намеревались (в случае опечатки или из ненадёжного пользовательского ввода), используйте функцию `lock_keys()` чтобы ограничить хеш его текущим набором ключей. Любая попытка добавить новый ключ в хеш выбросит исключение. Эта слабая мера безопасности годится только для предотвращения случайностей; кто угодно может использовать функцию `unlock_keys()` чтобы удалить эту защиту.

Аналогично вы можете заблокировать или разблокировать существующее значение для заданного ключа хеша (`lock_value()` и `unlock_value()`) и сделать весь хеш доступным только для чтения или лишить его этого свойства с помощью `lock_hash()` и `unlock_hash()`.

Приведение типов

Переменная в Perl может содержать в разное время значения разных типов — строки, целые и рациональные числа и т. д. Вместо присоединения информации о типе к переменным, Perl полагается на контекст, устанавливаемый операторами для понимания того, что делать со значениями. По своему дизайну, Perl пытается делать то, что вы имеете ввиду_(Это называется *DWIM), do what i mean* (*делай то, что я имею ввиду*), или *DWIM-ность., хотя вы должны чётко выражать свои намерения. Если вы обращаетесь с переменной, содержащей число, как со строкой, Perl сделает всё, что сможет, для* приведения типа_ этого числа к строке.

Булево приведение типов

Булево приведение типов происходит, когда вы проверяете *истинность* значения, как, например, в условиях `if` или `while`. Числовой 0, , пустая

строка и строка `'0'` являются ложными значениями. Все остальные значения — включая строки, которые могут быть *численно* равными нулю (такие как `'0.0'`, `'0e'` и `'0 but true'`), — являются ложными.

Если скаляр имет *оба* компонента , и строковый, и числовой, Perl 5 предпочтёт проверить на булеву истину строковый компонент. `'0 but true'` численно вычисляется в ноль, но это не пустая строка, поэтому в булевом контексте она вычисляется в истинное значение.

Строковое приведение типов

Строковое приведение типов происходит при использовании строковых операторов, таких как сравнение (`eq` и `cmp`), конкатенация, `split`, `substr` и регулярные выражения, а также при использовании значения как ключа хеша. Неопределённое значение преобразуется в пустую строку, выдавая предупреждение «use of uninitialized value». Числа *преобразуются* в строки, содержащие их значения, то есть значение `10` преобразуется в строку `10`. Вы можете даже разделить число на отдельные цифры с помощью `split`:

```
my @digits = split '', 1234567890;
```

Числовое приведение типов

Числовое приведение типов происходит при использовании операторов числового сравнения (таких как `==` и `<=>`), при выполнении математических операций и при использовании значения как индекса массива или списка. Неопределённое значение *преобразуется* в ноль и выдаёт предупреждение «use of uninitialized value». Строки, которые не начинаются с числовой части, тоже преобразуются в ноль и выдают предупреждение «Argument isn't numeric». Строки, начинающиеся с символов, разрешённых в числовых литералах, преобразуются в эти значения, не выдавая предупреждений, то есть `10 leptons leaping` преобразуется в `10`, а `6.022e23 moles marauding` — в `6.022e23`.

Базовый модуль `Scalar::Util` содержит функцию `looks_like_number()`, которая использует те же правила разбора, что и грамматика

Perl 5, для выделения числа из строки.

Ссылочное преобразование типов

Использование операции разыменования ссылки на значении, не являющемся ссылкой, превращает его *в* ссылку. Этот процесс автовивификации удобен при манипулировании вложенными структурами данных :

```
my %users;

$users{Brad}{id} = 228;
$users{Jack}{id} = 229;
```

Хотя хеш не содержал значений для ключей `Brad` и `Jack`, Perl создаёт для них ссылки на хеши и присваивает каждому пару ключ/значение с ключом `id`.

Кешированное преобразование типов

Внутреннее представление значений в Perl 5 хранит и строковое, и числовое значения. Преобразование числа в строку не *заменяет* числовое значение. Вместо этого оно *присоединяет* преобразованное в строку значение, так что представление будет содержать *оба* компонента. Аналогично, преобразование строки в число заполняет числовой компонент, оставляя строковый компонент нетронутым.

Некоторые операции в Perl предпочитают использование одного из компонентов другому — булева проверка, например, предпочитает строки. Если значение имеет кешированное представление в форме, которую вы не ожидаете, расчёт на неявную конвертацию может привести к неожиданным результатам. Вам почти никогда не нужно будет явно указывать, что вы ожидаете_(Автор может припомнить два таких случая за более чем десятилетие программирования на Perl 5)_, но знание того, что это кеширование происходит, может однажды помочь вам диагностировать странную ситуацию.

Двойные переменные

Многокомпонентная природа переменных Perl доступна пользователям в виде *двойных переменных*. Базовый модуль `Scalar::Util` предоставляет функцию `dualvar()`, которая позволяет вам обойти приведение типов Perl и манипулировать строковым и числовым компонентами значения по отдельности:

```
use Scalar::Util 'dualvar';
my $false_name = dualvar 0, 'Sparkles & Blue';

say 'Boolean true!'  if        !! $false_name;
say 'Numeric false!' unless  0  + $false_name;
say 'String true!'   if     ''  . $false_name;
```

Пакеты

Пространство имён в Perl объединяет и инкапсулирует разные именованные сущности в одной именованной категории, как ваша фамилия или название бренда. В отличие от имён в реальном мире, пространство имён не подразумевает никаких непосредственных взаимосвязей между сущностями. Такие взаимосвязи могут существовать, но не обязаны.

Пакет в Perl 5 — это набор кода в едином пространстве имён. Имеется тонкое различие: пакет представляет собой исходный код, а пространство имён — сущность, создаваемую, когда Perl парсит этот код.

Встроенная директива `package` объявляет пакет и пространство имён:

```
package MyCode;

our @boxes;

sub add_box { ... }
```

Все глобальные переменные, которые объявляются или на которые ссылаются после объявления пакета, ссылаются на символы в пространстве имён `MyCode`. Вы можете ссылаться на переменную `@boxes` из пространства имён `main` только по её *полностью определённому* имени `@MyCode::boxes`. Полностью определённое имя включает полное имя пакета, так что вы можете вызвать функцию `add_box()` только как `MyCode::add_box()`.

Область видимости пакета продолжается до следующего объявления `package` или до конца файла, что будет достигнуто раньше. Perl 5.14 усовершенствовал `package`, так что теперь вы можете добавить блок, который явно очертит область видимости объявления:

```
package Pinball::Wizard
{
    our $VERSION = 1969;
}
```

Пакет по умолчанию — `main`. Без объявления пакета, текущим пакетом будет `main`. Это правило распространяется на однострочники, автономные программы и даже файлы `.pm`.

Кроме имени пакет имеет версию и три неявных метода, `import()` , `unimport()` и `VERSION()`. `VERSION()` возвращает номер версии пакета. Этот номер представляет собой последовательность чисел, содержащуюся в глобальной переменной пакета с именем `$VERSION`. По примерному соглашению, версии имеют тенденцию быть последовательностью целых чисел, разделённых точками, как в `1.23` или `1.1.10`, где каждый сегмент — целое число.

Perl 5.12 представил новый синтаксис, предназначенный для упрощения номеров версий, как документировано в `perldoc version::Internals`. Эти более строгие номера версий должны начинаться с символа `v` и иметь как минимум три целочисленных компонента, разделённых точками:

```
package MyCode v1.2.1;
```

В Perl 5.14 опциональная блочная форма объявления `package` выглядит так:

```
package Pinball::Wizard v1969.3.7
{
    ...
}
```

В 5.10 и раньше, самым простым способом объявить версию пакета следующий:

```
package MyCode;

our $VERSION = 1.21;
```

Каждый пакет наследует метод `VERSION()` от базового класса `UNIVERSAL`. Вы можете переопределить `VERSION()`, хотя есть не так много причин это делать. Этот метод возвращает значение `$VERSION`:

```
my $version = Some::Plugin->VERSION();
```

Если вы укажете номер версии в качестве аргумента, этот метод выбросит исключение, если версия модуля меньше указанного аргумента:

```
# требует как минимум 2.1
Some::Plugin->VERSION( 2.1 );

die "Your plugin $version is too old"
    unless $version > 2;
```

Пакеты и пространства имён

Каждое объявление `package` создаёт новое пространство имён, если необходимо, и заставляет парсер разместить все глобальные символы следующего затем пакета (глобальные переменные и функции) в этом пространстве имён.

В Perl *открытые пространства имён*. Вы можете добавлять функции или переменные в это пространство имён в любом месте, как с помощью нового объявления пакета:

```
package Pack
{
    sub first_sub { ... }
}

Pack::first_sub();

package Pack
{
    sub second_sub { ... }
}

Pack::second_sub();
```

...так и с помощью указания полностью определённых имён функций в местах объявления:

```
# подразумевается
package main;

sub Pack::third_sub { ... }
```

Вы можете делать добавления в пакет в любой точке во время компиляции или выполнения, независимо от текущего файла, хотя сборка пакета из нескольких отдельных объявлений может сделать код трудным для исследования.

Пространства имён могут иметь столь много уровней, сколько требует ваша организационная схема, хотя пространства имён — не иерархические. Единственная взаимосвязь между пакетами — семантическая, не техническая. Многие проекты и бизнесы создают свои собственные пространства имён верхнего уровня. Это уменьшает вероятность глобальных конфликтов и помогает организовать код на диске. Например:

- `StrangeMonkey` — имя проекта
- `StrangeMonkey::UI` содержит код верхнего уровня для пользовательского интерфейса
- `StrangeMonkey::Persistence` содержит код верхнего уровня для управления данными
- `StrangeMonkey::Test` содержит код верхнего уровня для тестирования проекта

...и т. д.

Ссылки

Perl обычно делает то, чего вы ожидаете, даже если это требует проницательности. Посмотрите, что происходит, когда вы передаёте значения в функции:

```
sub reverse_greeting
{
    my $name = reverse shift;
    return "Hello, $name!";
}

my $name = 'Chuck';
say reverse_greeting( $name );
say $name;
```

Снаружи функции `$name` содержит значение `Chuck`, несмотря на то, что значение, переданное в функцию, переворачивается в `kcuhC`. Вероятно, этого вы и ожидаете. Значение `$name` снаружи функции отдельно от значения `$name` внутри функции. Изменение одного не оказывает эффекта на другое.

Рассмотрим альтернативу. Если бы вам было необходимо делать копии каждого значения прежде чем что-либо могло бы изменить их независимо от вас, вам пришлось бы писать большое количество дополнительного защитного кода.

Однако иногда удобно модифицировать значения прямо на месте. Если вы хотите передать хеш, полный данных, в функцию, для изменения его, создание и возврат нового хеша для каждого изменения может быть хлопотным (уж не говоря о том, что неэффективным).

Perl 5 предоставляет механизм, позволяющий ссылаться на значение, не делая его копии. Любые изменения, сделанные с этой *ссылкой*, сразу же обновят значение, так что *все* ссылки на это значение смогут получить новое значение. Ссылка в Perl 5 — скалярный тип данных первого класса, который ссылается на другой тип данных первого класса.

Ссылки на скаляры

Оператор взятия ссылки — обратный слеш (\). В скалярном контексте он создаёт одиночную ссылку, ссылающуюся на другое значение. В списочном контексте он создаёт список ссылок. Чтобы взять ссылку от `$name`:

```
my $name     = 'Larry';
my $name_ref = \$name;
```

Вы должны *разыменовать* ссылку, чтобы получить значение, на которое она ссылается. Разыменовывание требует добавления дополнительного сигила для каждого уровня разыменовывания:

```
sub reverse_in_place
{
    my $name_ref = shift;
    $$name_ref   = reverse $$name_ref;
}

my $name = 'Blabby';
reverse_in_place( \$name );
say $name;
```

Двойной скалярный сигил (`$$`) разыменовывает ссылку на скаляр.

Находящиеся в `@_` параметры ведут себя как *псевдонимы* переменных вызывающего кода_(Вспомните, что цикл `for`) имеет аналогичное поведение в плане создания псевдонимов._, так что вы можете изменять их на месте:

```
sub reverse_value_in_place
{
    $_[0] = reverse $_[0];
}

my $name = 'allizocohC';
reverse_value_in_place( $name );
say $name;
```

В большинстве случаев вы не захотите модифицировать значения таким способом — вызывающий код, например, редко этого ожидает. Присваивание параметров лексическим переменным внутри ваших функций избавляет от такого поведения.

Сложные ссылки могут потребовать использования блока в фигурных скобках для разрешения неоднозначности частей выражения. Вы можете всегда использовать такой синтаксис, хотя иногда он проясняет, а иногда затеняет:

```
sub reverse_in_place
{
    my $name_ref   = shift;
    ${ $name_ref } = reverse ${ $name_ref };
}
```

Если вы забудете разыменовать ссылку на скаляр, Perl, вероятно, выполнит приведение типа ссылки. Строковое значение будет иметь вид `SCALAR(0x93339e8)`, а числовое будет содержать часть `0x93339e8`. В этом значении зашифрован тип ссылки (в данном случае `SCALAR`) и положение ссылки в памяти.

Ссылки на массивы

Ссылки на массивы полезны в нескольких ситуациях:

- Для передачи и возвращения массивов из функций без разглаживания
- Для создания многомерных структур данных
- Чтобы избежать ненужного копирования массива
- Для сохранения анонимных структур данных

Используйте оператор взятия ссылки для создания ссылки на объявленный массив:

```
my @cards     = qw( K Q J 10 9 8 7 6 5 4 3 2 A );
my $cards_ref = \@cards;
```

Любые изменения, сделанные через `$cards_ref`, будут изменять и `@cards`, и наоборот. Вы можете получить доступ ко всему массиву целиком с помощью сигила @, например, чтобы разгладить массив или сосчитать его элементы:

```
my $card_count = @$cards_ref;
my @card_copy  = @$cards_ref;
```

Доступ к отдельным элементам осуществляется с помощью разыменовывающей стрелки (`->`):

```
my $first_card = < $cards_ref-[0]  >>;
my $last_card  = < $cards_ref-[-1] >>;
```

Стрелка необходима, чтобы различать скаляр `$cards_ref` и массив `@cards_ref`. Обратите внимание на использование скалярного сигила для доступа к единственному элементу.

Используйте разыменовывающий синтаксис с фигурными скобками для получения среза по ссылке на массив:

```
my @high_cards = @{ $cards_ref }[0 .. 2, -1];
```

Вы *можете* опустить фигурные скобки, но их группировка часто улучшает читабельность.

Для создания анонимного массива — без использования объявленного массива — окружите список значений квадратными скобками:

```
my $suits_ref = [qw( Monkeys Robots Dinos Cheese )];
```

Эта ссылка на массив ведёт себя так же, как ссылки на именованные массивы, за исключением того, что скобки анонимного массива *всегда* создают новую ссылку. Ссылка, взятая на именованный массив, всегда ссылается на *тот же самый* массив с учётом области видимости. Например:

```
my @meals      = qw( soup sandwiches pizza );
my $sunday_ref = \@meals;
my $monday_ref = \@meals;

push @meals, 'ice cream sundae';
```

...и `$sunday_ref`, и `$monday_ref` теперь содержат десерт, тогда как в следующем случае:

```
my @meals      = qw( soup sandwiches pizza );
my $sunday_ref = [ @meals ];
my $monday_ref = [ @meals ];

push @meals, 'berry pie';
```

...ни `$sunday_ref`, ни `$monday_ref` не содержат десерт. В квадратных скобках, использованных для создания анонимного массива, списочный контекст разглаживает массив `@meals` в список, не связанный с `@meals`.

Ссылки на хеши

Используйте оператор взятия ссылки на именованном хеше для создания *ссылки на хеш*:

```
my %colors = (
    black  => 'negro',
    blue   => 'azul',
    gold   => 'dorado',
    red    => 'rojo',
    yellow => 'amarillo',
    purple => 'morado',
);

my $colors_ref = \%colors;
```

Доступ к ключам или значениям хеша можно получить, предварив ссылку сигилом хеша (%):

```
my @english_colors = keys   %$colors_ref;
my @spanish_colors = values %$colors_ref;
```

Доступ к отдельным значениям хеша (для сохранения, удаления, проверки существования или получения) осуществляется с использованием разыменовывающе стрелки или двойных сигилов:

```
sub translate_to_spanish
{
    my $color = shift;
    return < $colors_ref-{$color} >>;
```

```
    # или return < $$colors_ref{$color} >;
}
```

Используйте сигил массива (@) и устраняющие неоднозначность скобки для получения среза по ссылке на хеш:

```
my @colors  = qw( red blue green );
my @colores = @{ $colors_ref }{@colors};
```

Создание анонимного хеша с помощью фигурных скобок:

```
my $food_ref = {
    'birthday cake' => 'la torta de cumpleaE<ntilde>os',
    candy           => 'dulces',
    cupcake         => 'bizcochito',
    'ice cream'     => 'helado',
};
```

Как и анонимные массивы, анонимные хеши создают новый анонимный хеш при каждом выполнении.

Автоматическое разыменование

Начиная с Perl 5.14, Perl может автоматически разыменовать некоторые ссылки от вашего имени. Имея ссылку на массив в `$arrayref`, вы можете написать:

```
push $arrayref, qw( list of values );
```

Имея выражение, возвращающее ссылку на массив, вы можете сделать то же самое:

```
push $houses{$location}[$closets], \@new_shoes;
```

То же относится и к операторам работы с массивами `pop`, `shift`, `unshift`, `splice`, `keys`, `values`, и `each`, а также к операторам работы с хешами `keys`, `values` и `each`.

Если предоставленная ссылка имеет несоответствующий тип — если она не разыменовывается должным образом — Perl выбросит исключение. Хотя это выглядит более опасным, чем явное разыменовывание ссылок напрямую, фактически, это то же самое поведение:

```
my $ref = sub { ... };

# выбросит исключение
push  $ref, qw( list of values );

# тоже выбросит исключение
push @$ref, qw( list of values );
```

Ссылки на функции

Perl 5 поддерживает *функции первого класса*, поскольку функция — такой же тип данных, как массив или хеш. Это наиболее очевидно в случае *ссылок на функции* и даёт много продвинутых возможностей . Ссылка на функцию создаётся посредством использования оператора взятия ссылки на имени функции:

```
sub bake_cake { say 'Baking a wonderful cake!' };

my $cake_ref = \&bake_cake;
```

Без *сигила функции* (&), вы получите ссылку на возвращаемое значение или значения функции.

Анонимная функция создаётся с помощью голого ключевого слова `sub`:

```
my $pie_ref = sub { say 'Making a delicious pie!' };
```

Использование встроенной директивы C *без* имени компилирует функцию, как обычно, но не помещает её в текущее пространство имён. Единственный способ получить доступ к этой функции — по ссылке, возвращённой `sub`. Вызыв функции по ссылке осуществляется с помощью разыменовывающей стрелки:

```
$cake_ref->();
$pie_ref->();
```

Воспринимайте пустые круглые скобки как обозначение операции разыменовывающе
вызова, так же как квадратные скобки указывают на поиск по индексу, а фигурные скобки служат для поиска в хеше. Передавайте аргументы в функцию с помощью круглых скобок:

```
$bake_something_ref->( 'cupcakes' );
```

Также вы можете использовать ссылки на функции как методы объектов . Это удобно, когда вы уже нашли метод :

```
my $clean = $robot_maid->can( 'cleanup' );
$robot_maid->$clean( $kitchen );
```

Ссылки на дескрипторы файлов

Когда вы используете `open` (и `opendir`) в форме с лексическими дескрипторами файлов, вы имеете дело со ссылками на дескрипторы файлов. Внутренне эти дескрипторы файлов — объекты `IO::File`. Вы можете напрямую вызывать их методы. Начиная с Perl 5.14 это можно делать так:

```
open my $out_fh, '>', 'output_file.txt';
$out_fh->say( 'Have some text!' );
```

В 5.12 вы должны написать `use IO::File;` чтобы включить эту возможность, а в Perl 5.10 и раньше — `use IO::Handle;`. Даже более старый код может получать ссылки на тайпглобы:

```
local *FH;
open FH, "> $file" or die "Can't write '$file': $!";
my $fh = \*FH;
```

Эта идиома предшествует лексическим дескрипторам файлов (представленным в Perl 5.6.0 в марте 2000 года). Вы всё ещё можете использовать оператор взятия ссылки на тайпглобах для получения ссылок на дескрипторы файлов, глобальные для пакета, такие как `STDIN`, `STDOUT`, `STDERR` или `DATA` — но это всё в любом случае глобальные имена.

Использование лексических дескрипторов файлов по возможности предпочтительно. С преимуществами явного задания области видимости, лексические дескрипторы файлов позволяют вам управлять их сроком жизни благодаря возможностям Perl 5 по управлению памятью.

Счётчики ссылок

Perl 5 использует технику управления памятью, известную как *подсчёт ссылок*. Каждое значение в Perl имеет присоединённый к нему счётчик. Perl увеличивает этот счётчик при каждом взятии ссылки на это значение, явном или неявном. Perl уменьшает этот счётчик каждый раз, когда ссылка исчезает. Когда счётчик достигает ноля, Perl может безопасно отправить это значение на переработку.

Как Perl узнаёт, когда можно безопасно освободить память, занимаемую переменной? Как Perl узнаёт, когда безопасно закрыть дескриптор файла, открытый во внутренней области видимости?

```
say 'file not open';

{
    open my $fh, '>', 'inner_scope.txt';
```

```
    $fh->say( 'file open here' );
}

say 'file closed here';
```

Во внутреннем блоке в этом примере только один дескриптор `$fh`. (Несколько строк в исходном коде упоминают его, но ссылается на него только одна переменная: `$fh`.) `$fh` существует только в области видимости этого блока. Её значение никогда не покидает блок. Когда выполнение достигает конца блока, Perl перерабатывает переменную `$fh` и уменьшает счётчик ссылок содержащегося в ней дескриптора файла. Счётчик дескриптора достигает ноля, так что Perl перерабатывает его для освобождения памяти и неявно вызывает `close()`.

Вам не обязательно понимать детали того, как всё это работает. Вам нужно понимать только то, что ваши действия по взятию ссылок и их передаче влияют на то, как Perl управляет памятью (см. .

Ссылки и функции

При использовании ссылок как аргументов функции внимательно документируйте ваши намерения. Изменение значений ссылки изнутри функции может стать неожиданностью для вызывающего кода, который не ожидает, что кто-то ещё будет модифицировать его данные. Для изменения содержимого ссылки, не воздействуя на саму ссылку, скопируйте её значение в новую переменную:

```
my @new_array = @{ $array_ref };
my %new_hash  = %{ $hash_ref  };
```

Это необходимо лишь в немногих случаях, но явное клонирование помогает избежать неприятных сюрпризов для вызывающего кода. Если вы используете вложенные структуры данных или другие сложные ссылки, рассмотрите использование базового модуля `Storable` и его функции `dclone` (*deel cloning*, *глубокое клонирование*).

Вложенные структуры данных

Агрегатные типы данных Perl — массивы и хеши — позволяют вам хранить скаляры, индексированные по целым числам или строковым ключам. Ссылки в Perl 5 позволяют получать доступ к агрегатным типам данных через специальные скаляры. Вложенные структуры данных в Perl, такие как массив массивов или хеш хешей, становятся возможными благодаря использованию ссылок.

Для объявления вложенной структуры данных используйте синтаксис объявления анонимной ссылки:

```
my @famous_triplets = (
    [qw( eenie miney moe   )],
    [qw( huey  dewey louie )],
    [qw( duck  duck  goose )],
);

my %meals = (
    breakfast => { entree => 'eggs',
                   side   => 'hash browns' },
    lunch     => { entree => 'panini',
                   side   => 'apple'         },
    dinner    => { entree => 'steak',
                   side   => 'avocado salad' },
);
```

Для доступа к элементам вложенных структур данных используйте синтаксис ссылок Perl. Сигил обозначает количество данных, которое должно быть получено, а разыменовывающая стрелка указывает, что значение части структуры данных является ссылкой:

```
my $last_nephew = $famous_triplets[1]->[2];
my $breaky_side = $meals{breakfast}->{side};
```

Единственный способ создать многоуровневую структура данных —

посредством ссылок, так что использование стрелки излишне. Вы можете опустить её для ясности, за исключением вызовов функций по ссылке:

```
my $nephew = $famous_triplets[1][2];
my $meal   = $meals{breakfast}{side};
$actions{financial}{buy_food}->( $nephew, $meal );
```

Используйте устраняющий неоднозначность блок для доступа к компонентам вложенной структуры данных, как если бы они были массивами или хешами первого класса:

```
my $nephew_count   = @{ $famous_triplets[1] };
my $dinner_courses = keys %{ $meals{dinner} };
```

—или для получения среза вложенной структуры данных:

```
my ($entree, $side) = @{ $meals{breakfast} }
                       {qw( entree side )};
```

Пробелы помогают, но не полностью устраняют зашумлённость этой конструкции. Используйте временные переменные для придания ясности:

```
my $meal_ref        = $meals{breakfast};
my ($entree, $side) = @$meal_ref{qw( entree side )};
```

...или используйте неявное создание псевдонимов в `$_` директивой `for`, чтобы избежать использования промежуточной ссылки:

```
my ($entree, $side) = @{ $_ }{qw( entree side )}
                      for $meals{breakfast};
```

`perldoc perldsc`, кулинарная книга структур данных, даёт обширные примеры того, как использовать разные структуры данных в Perl.

Автовивификация

Когда вы пытаетесь записать значение в компонент вложенной структуры данных, Perl по необходимости создаёт путь через структуру данных до пункта назначения:

```
my @aoaoaoa;
$aoaoaoa[0][0][0][0] = 'nested deeply';
```

После второй строки кода этот массив массивов массивов массивов содержит ссылку на массив в ссылке на массив в ссылке на массив в ссылке на массив. Каждая ссылка на массив содержит один элемент. Аналогично, обработка неопределённого значения так, как будто это ссылка на хеш во вложенной структуре данных, сделает его таковой:

```
my %hohoh;
$hohoh{Robot}{Santa} = 'mostly harmful';
```

Это полезное поведение называется *автовивификацией*. Оно уменьшает объём инициализирующего кода вложенной структуры данных, но не может определить разницу между честным намерением создать недостающие элементы во вложенной структуре данных и опечаткой. Прагма `autovivification` из CPAN позволит вам отключить автовивификацию в лексической области видимости для определённых типов операций.

Вас может удивить применение преимуществ автовивификации параллельно установке ограничений с помощью `strict`. Это вопрос баланса. Будет ли более удобным получить ошибки, изменяющие поведение вашей программы, ценой отключения проверок на ошибки для хорошо инкапсулированных символьных ссылок? Будет ли более удобным позволить структурам данных расти вместо чёткого указания их размера и доступных ключей?

Ответы зависят от вашего проекта. В период начальной разработки позвольте себе свободу экспериментировать. Во время тестирования и развёртывания рассмотрите увеличение строгости для предотвращения нежелательных побочных эффектов. Благодаря лексической области

видимости прагм `strict` и `autovivification` вы можете включить эти варианты поведения там, где это необходимо.

Вы *можете* проверять свои ожидания перед разыменованием каждого уровня сложной структуры данных, но результирующий код зачастую будет длинным и громоздким. Лучше избегать глубоко вложенных структур данных, пересмотрев свою модель данных в сторону лучшей инкапсуляции.

Отладка вложенных структур данных

Сложность разыменовывающего синтаксиса Perl 5, объединённая с потенциальной путаницей нескольких уровней ссылок, может сделать отладку вложенных структур данных сложной. Есть два хороших инструмента визуализации.

Базовый модуль `Data::Dumper` преобразует значения произвольной сложности в строки кода Perl 5:

```
use Data::Dumper;

print Dumper( $my_complex_structure );
```

Это удобно для определения того, что содержит структура данных, что вы должны получить, и что вы получили вместо этого. `Data::Dumper` может выводить объекты, так же как ссылки на функции (если вы установите `$Data::Dumper::Deparse` в истинное значение).

Хотя `Data::Dumper` — базовый модуль и выводит код на Perl 5, его вывод довольно многословен. Некоторые разработчики предпочитают использовать для отладки модули `YAML::XS` или `JSON`. Они не выдают код на Perl 5, но их вывод может быть гораздо проще для чтения и понимания.

Циклические ссылки

Имеющаяся в Perl 5 система управления памятью с помощью подсчёта ссылок имеет один недостаток, заметный для пользовательского кода. Две

ссылки, которые в конечном счёте указывают друг на друга, образуют *циклическую ссылку*, которую Perl не может уничтожить сам. Рассмотрим биологическую модель, где каждая сущность имеет двух родителей и ноль или больше детей:

```
my $alice  = { mother => '',     father => ''      };
my $robert = { mother => '',     father => ''      };
my $cianne = { mother => $alice, father => $robert };

push @{ $alice->{children}  }, $cianne;
push @{ $robert->{children} }, $cianne;
```

И `$alice`, и `$robert` содержат ссылку на массив, которая содержит `$cianne`. Поскольку `$cianne` — ссылка на хеш, которая содержит `$alice` и `$robert`, Perl никогда не сможет уменьшить счётчик ссылок любого из этих трёх человек до нуля. Он не распознает, что эти циклические ссылки существуют, и не сможет управлять сроком жизни этих сущностей.

Или разбейте счётчик ссылок сами вручную (очистив детей `$alice` и `$robert` или родителей `$cianne`), или используйте *слабые ссылки*. Слабая ссылка — это ссылка, которая не увеличивает счётчик ссылок значения, на которое ссылается. Слабые ссылки доступны через базовый модуль `Scalar::Util`. Его функция `weaken()` предотвращает увеличение счётчика ссылок:

```
use Scalar::Util 'weaken';

my $alice  = { mother => '',     father => ''      };
my $robert = { mother => '',     father => ''      };
my $cianne = { mother => $alice, father => $robert };

push @{ $alice->{children}  }, $cianne;
push @{ $robert->{children} }, $cianne;

< weaken( $cianne-{mother} ); >>
< weaken( $cianne-{father} ); >>
```

Теперь `$cianne` сохранит ссылки на `$alice` и `$robert`, но эти ссылки сами по себе не будут удерживать сборщик мусора Perl от разрушения этих структур данных. Большинство структур данных не требуют слабых ссылок, но когда они необходимы — они бесценны.

Альтернативы вложенным структурам данных

Хотя Perl способен обрабатывать структуры данных, вложенные так глубоко, как вы только можете представить, человеческая стоимость понимания этих структур данных и их взаимоотношений — не говоря о сложном синтаксисе — высока. После двух или трёх уровней вложенности, задумайтесь, не позволит ли вам моделирование разных компонентов вашей системы как классов и объектов сделать код чище.

6. Операторы

Некоторые называют Perl «оператор-ориентированным языком». Чтобы понять программу на Perl, вы должны понять как её операторы взаимодействуют со своими операндами.

Оператор в Perl — это набор из одного или более символов, используемый как часть синтаксиса языка. Каждый оператор оперирует нолём или более *операндов*. Воспринимайте операторы как особый вид функций, понимаемых парсером, а их операторы — как аргументы.

Свойства операторов

Каждый оператор обладает несколькими важными свойствами, определяющими его поведение: количество операндов, которыми он оперирует, его отношения с другими операторами и его синтаксические возможности.

`perldoc perlop` и `perldoc perlsyn` предоставляют обширную информацию об операторах Perl, но документация подразумевает, что вы уже знакомы с некоторыми деталями того, как они работают. Основные понятия компьютерных наук могут выглядеть внушительно, но когда вы проберётесь через их названия, они довольно очевидны. Неявно вы их уже понимаете.

Приоритет

Приоритет оператора определяет, когда Perl должен вычислить его в выражении. Порядок вычисления следует от наивысших приоритетов к наинизшим. Так как приоритет умножения выше, чем приоритет сложения, результатом вычисления `7 + 7 * 10` будет `77`, а не `140`.

Чтобы заставить какие-либо операторы вычисляться раньше других, сгруппируйте их подвыражения в круглые скобки. В выражении `(7 + 7) * 10` группировка сложения в одно целое приводит к тому, что оно вычисляется раньше умножения. Результатом будет `140`.

`perldoc perlop` содержит таблицу приоритетов. Прочитайте её, поймите, но не утруждайте себя запоминанием (почти никто этого не делает). Лучше сохраняйте свои выражения простыми и добавляйте скобки для прояснения своего замысла.

В случае, если два оператора имеют одинаковый приоритет, ситуацию разрешают другие факторы, такие как ассоциативность и фиксность .

Ассоциативность

Ассоциативность оператора определяет, вычисляется он слева направо или справа налево. Сложение левоассоциативно, поэтому `2 + 3 + 4` сначала вычисляет `2 + 3`, а затем добавляет к результату `4`. Возведение в степень правоассоциативно, поэтому `2 ** 3 ** 4` вычисляет сначала `3 ** 4`, а затем возводит `2` в 81-ую степень.

Стоит запомнить приоритет и ассоциативность распространённых математических операторов, но, опять-таки, простота вам поможет. Используйте скобки для прояснения своего замысла.

Арность

Арность оператора — это количество операндов, которыми он оперирует. *Нольарный* оператор имеет ноль операндов. *Унарный* оператор имеет один оператор. *Бинарный* оператор имеет два операнда. *Тернарный* оператор имеет три операнда. Оператор *списочной арности* оперирует списком операндов. Документация и примеры использования оператора должны сделать его арность ясной.

Например, арифметические операторы — бинарные, и обычно левоассоциативные. `2 + 3 - 4` сначала вычисляет `2 + 3`; сложение и вычитание имеют одинаковый приоритет, но они левоассоциативные и бинарные, поэтому правильный порядок вычисления применяет самый левый оператор (`+`) к двум самым левым операндам (`2` и `3`), а затем применяет правый оператор (`-`) к результату первой операции и правому операнду (`4`).

Новичков в Perl часто сбивает с толку взаимодействие между операторами

списочной арности — особенно вызовами функций — и вложенными выражениями. Хотя скобки обычно помогают, опасайтесь запутанности парсинга выражения:

```
# вероятно ошибочный код
say ( 1 + 2 + 3 ) * 4;
```

...которое выводит значение `6` и (вероятно) в целом возвращает `4` (возвращаемое значение `say`, умноженное на `4`). Парсер Perl успешно интерпретирует скобки как постциркумфиксные операторы, обозначающие аргументы `say`, а не циркумфиксные скобки, группирующие выражение для изменения приоритета.

Фиксность

Фиксность оператора — это его расположение по отношению к своим операндам:

- *Инфиксные* операторы располагаются между своими операндами. Большинство математических операторов — инфиксные операторы, такие как оператор умножения в `$length * $width`.
- *Префиксные* операторы предшествуют своим операндам. *Постфиксные* операторы следуют за операндами. Эти операторы склонны быть унарными, как математическое отрицание (`-$x`), булево отрицание (`!$y`) и постфиксный инкремент (`$z++`).
- *Циркумфиксные* операторы окружают свои операнды, как конструктор анонимного хеша (`{ ... }`) и операторы заключения в кавычки (`qq[ ... ]`).
- *Постциркумфиксные* операторы следуют за одними операндами и окружают другие, как в доступе к элементам хеша или массива (`$hash{$x}` и `$array[$y]`).

Типы операторов

Оперторы Perl предоставляют контекст значения своим операндам. Чтобы выбрать подходящий оператор, нужно понимание значения предоставляемых вами операндов, а так же значения, которое вы ожидаете получить.

Числовые операторы

Числовые операторы налагают числовой контекст на свои операнды. Эти операторы — стандартные арифметические операторы, такие как сложение (`+`), вычитание (`-`), умножение (`*`), деление (`/`), возведение в степень (`**`), остаток от деления (%), их вариации (`+=`, `-=`, `*=`, `/=`, `**=` и `%=`), а также постфиксный и префиксный автодекремент (`--`).

Оператор автоинкремента имеет специальное строковое поведение .

Некоторые операторы сравнения налагают числовой контекст на свои операнды. Это числовое равенство (`==`), числовое неравенство (`!=`), больше чем (`>`), меньше чем (`<`), больше или равно (`>=`), меньше или равно (`<=`) и оператор сравнения (`<=>`).

Строковые операторы

Строковые операторы налагают строковый контекст на свои операнды. Это положительная и отрицательная привязка регулярных выражений (`=~` and `!~` соответственно) и конкатенация (`.`).

Некоторые операторы сравнения налагают строковый контекст на свои операнды. Это строковое равенство (`eq`), строковое неравенство (`ne`), больше чем (`gt`), меньше чем (`lt`), больше или равно (`ge`), меньше или равно (`le`) и оператор строкового сравнения (`cmp`).

Логические операторы

Логические операторы налагают булев контекст на свои операнды. Это операторы `&&`, `and`, `||` и `or`. Все они инфиксные и проявляют поведение *короткого замыкания* . Словесные варианты имеют более низкий приоритет, чем их пунктуационные формы.

Оператор определено-или, `//`, проверяет *определённость* своего операнда. В отличие от `||`, который проверяет *истинность* своего операнда, `//` возвращает истинное значение даже если его операнд вычисляется в числовой ноль или пустую строку. Это особенно полезно для установки параметров по умолчанию:

```
sub name_pet
{
    my $name = shift // 'Fluffy';
    ...
}
```

Тернарный условный оператор (`?:`) принимает три операнда. Он вычисляет первый операнд в булевом контексте и возвращает второй операнд, если первый — истина, или третий — в противном случае:

```
my $truthiness = $value ? 'true' : 'false';
```

Префиксные операторы `!` и `not` возвращают логическую противоположность булева значения своих операндов. `not` — низкоприоритетная версия `!`.

Оператор `xor` — инфиксный оператор, вычисляющий исключающее или своих операндов.

Побитовые операторы

Побитовые операторы обрабатывают свои операнды численно на битовом уровне. Эти операторы не очень распространены. Они включают левый

сдвиг (<<), правый сдвиг (>>), побитовое и (&), побитовое или (|) и побитовое исключающее или (^), а также их варианты (<<=, >>=, &=, |= и ^=).

Специальные операторы

Оператор автоинкремента имеет специальное поведение. При использовании на значении с числовым компонентом , оператор инкрементирует этот числовой компонент. Если значение очевидно строка (и не имеет числового компонента), оператор инкрементирует это строковое значение так, что a становится b, zz становится aaa и a9 становится b0.

```
my $num = 1;
my $str = 'a';

$num++;
$str++;
is( $num,   2, 'numeric autoincrement' );
is( $str, 'b', 'string autoincrement'  );

no warnings 'numeric';
$num += $str;
$str++;

is( $num, 2, 'numeric addition with $str'    );
is( $str, 1, '... gives $str a numeric part' );
```

Оператор повторения (x) — инфиксный оператор со сложным поведением. В списочном контексте, когда передан список, он возвращает этот список, повторённый количество раз, определяемое вторым операндом. В списочном контексте если передан скаляр, он выдаёт строку, состоящую из строкового значения его первого операнда, конкатенированного с собой количество раз, определяемое вторым операндом.

В скалярном контексте оператор всегда выдаёт конкатенированную строку, соответствующим образом повторённую. Например:

```
my @scheherazade = ('nights') x 1001;
```

```
my $calendar     =  'nights'  x 1001;
my $cal_length   =  length $calendar;

is( @scheherazade, 1001, 'list repeated' );
is( $cal_length,   1001 * length 'nights',
                   'word repeated' );

my @schenolist   =  'nights'  x 1001;
my $calscalar    = ('nights') x 1001;

is( @schenolist, 1, 'no lvalue list' );
is( length $calscalar,
    1001 * length 'nights', 'word still repeated' );
```

Инфиксный оператор *диапазона* (`..`) генерирует список элементов в списочном контексте:

```
my @cards = ( 2 .. 10, 'J', 'Q', 'K', 'A' );
```

Он может генерировать простые инкрементирующиеся диапазоны (как целочисленные, так и строковые), но не может угадать шаблоны более сложных диапазонов.

В булевом контексте оператор диапазона становится оператором *триггера*. Этот оператор выдаёт ложное значение до тех пор, пока его левый операнд — истина. Это значение остаётся истинным, пока правый оператор — истина, после чего вновь становится ложным, пока левый операнд снова не станет истинным. Представьте разбор текста формального письма следующим образом:

```
while (/Hello, $user/ .. /Sincerely,/)
{
    say "> $_";
}
```

Оператор *запятая* (`,`) — инфиксный оператор. В скалярном контексте он вычисляет свой левый операнд, а затем возвращает значение, полученное

при вычислении правого операнда. В списочном контексте он вычисляет оба операнда в порядке слева направо.

Оператор толстой запятой (`=>`) кроме того автоматически заключает в кавычки любое голое слово, использованное как его левый операнд .

7. Функции

Функция (или *подпрограмма*) в Perl — это обособленная, инкапсулированная единица поведения. Программа — это набор маленьких чёрных ящиков, в котором взаимодействие функций руководит потоком управления. Функция может иметь имя. Она может потреблять входную информацию. Она может генерировать выходную информацию.

Функции — первичный механизм абстракции, инкапсуляции и повторного использования в Perl 5.

Объявление функций

Используйте встроенную директиву `sub` для объявления функции:

```
sub greet_me  { ... }
```

Теперь функция `greet_me()` доступна для вызова в любом месте программы.

Вы не обязаны *определять* функцию в той точке, в которой вы её объявляете. *Предварительное объявление* указывает Perl запомнить имя функции, несмотря на то, что вы объявите её позже:

```
sub greet_sun;
```

Вызов функций

Используйте постфиксные круглые скобки и имя функции для вызова этой функции и передачи необязательного списка аргументов:

```
greet_me( 'Jack', 'Brad' );
greet_me( 'Snowy' );
greet_me();
```

Хотя эти скобки не являются строго обязательными для этих примеров — даже с включенной прагмой `strict` — они обеспечивают ясность для читателей и парсера Perl. Если есть сомнения, используйте их.

Аргументы функций могут быть произвольными выражениями, включая простые переменные:

```
greet_me( $name );
greet_me( @authors );
greet_me( %editors );
```

...хотя стандартная обработка параметров в Perl 5 иногда удивляет новичков.

Параметры функций

Функция получает свои параметры в едином массиве, `@_` . Perl *разглаживает* все входные параметры в единый список. Функция должна либо распаковать все параметры в переменные, или работать с `@_` напрямую:

```
sub greet_one
{
    my ($name) = @_;
    say "Hello, $name!";
}

sub greet_all
{
    say "Hello, $_!" for @_;
}
```

Большая часть кода использует `shift` или распаковку списка, однако `@_` ведёт себя как обычный массив, так что вы можете ссылаться на отдельные элементы по индексу:

```
sub greet_one_shift
{
    my $name = shift;
    say "Hello, $name!";
}

sub greet_two_no_shift
{
    my ($hero, $sidekick) = @_;
    say "Well if it isn't $hero and $sidekick. Welcome!";
}

sub greet_one_indexed
{
    my $name = $_[0];
    say "Hello, $name!";

    # or, less clear
    say "Hello, $_[0]!";
}
```

... или использовать `shift`, `unshift`, `push`, `pop`, `splice` и `slice` с `@_`.

Присваивание скалярного параметра из `@_` требует использования `shift`, доступа по индексу к `@_` или скобок списочного контекста в левом значении. В противном случае, Perl 5 с удовольствием вычислит для вас `@_` в скалярном контексте и присвоит количество переданных параметров:

```
sub bad_greet_one
{
    my $name = @_;  # бажно
    say "Hello, $name; you look numeric today!"
}
```

Списочное присваивание нескольких параметров зачастую яснее, чем несколько строчек `shift`. Сравните:

```
sub calculate_value
```

```
{
    # несколько вызовов shift
    my $left_value  = shift;
    my $operation   = shift;
    my $right_value = shift;
    ...
}
```

...и:

```
sub calculate_value
{
    my ($left_value, $operation, $right_value) = @_;
    ...
}
```

Иногда требуется извлечь несколько параметров из `@_`, а остальные передать в другую функцию:

```
sub delegated_method
{
    my $self = shift;
    say 'Calling delegated_method()'

    $self->delegate->delegated_method( @_ );
}
```

Используйте `shift`, если ваша функция требует только одного параметра. Используйте списочное присваивание для доступа к нескольким параметрам.

Разглаживание

Разглаживание параметров в `@_` происходит на стороне вызывающего кода вызова функции. Передача хеша в качестве аргумента генерирует список пар ключ/значение:

```
my %pet_names_and_types = (
    Lucky   => 'dog',
    Rodney  => 'dog',
    Tuxedo  => 'cat',
    Petunia => 'cat',
);

show_pets( %pet_names_and_types );

sub show_pets
{
    my %pets = @_;
    while (my ($name, $type) = each %pets)
    {
        say "$name is a $type";
    }
}
```

Когда Perl разглаживает `%pet_names_and_types` в список, порядок пар ключ/значение из хеша будет различаться, но список всегда будет включать значение, следующее сразу за своим ключом. Присваивание хеша внутри `show_pets()` работает по существу также, как более явное присваивание `%pet_names_and_types`.

Разглаживание часто полезно, но остерегайтесь смешивать в списке параметров скаляры с разглаживаемыми агрегатными переменными. При написании функции `show_pets_of_type()`, в которой один параметр — это тип животного для вывода, передавайте этот тип как *первый* аргумент (или используйте `pop` для удаления его из конца `@_`):

```
sub show_pets_by_type
{
    my ($type, %pets) = @_;

    while (my ($name, $species) = each %pets)
    {
        next unless $species eq $type;
        say "$name is a $species";
    }
```

```
}

my %pet_names_and_types = (
    Lucky   => 'dog',
    Rodney  => 'dog',
    Tuxedo  => 'cat',
    Petunia => 'cat',
);

show_pets_by_type( 'dog',   %pet_names_and_types );
show_pets_by_type( 'cat',   %pet_names_and_types );
show_pets_by_type( 'moose', %pet_names_and_types );
```

Проглатывание

Списочное присваивание агрегатной переменной всегда проявляет жадность, поэтому присваивание `%pets` *проглатывает* все оставшиеся значения из `@_`. Если параметр `$type` находится в конце `@_`, Perl предупредит о присваивании нечётного числа элементов хешу. Вы *можете* обойти это:

```
sub show_pets_by_type
{
    my $type = pop;
    my %pets = @_;

    ...
}
```

…ценой ясности. Тот же самый принцип, конечно же, применим к присваиванию массиву как параметру. Используйте ссылки чтобы избежать нежелательного разглаживания и проглатывания агрегатных значений.

Создание псевдонимов

`@_` имеет тонкость; он *создаёт псевдонимы* для аргументов функции, так что вы можете модифицировать их напрямую. Например:

```
sub modify_name
{
    $_[0] = reverse $_[0];
}

my $name = 'Orange';
modify_name( $name );
say $name;

# выводит C<egnarO>
```

Модифицируйте элемент `@_` напрямую, и вы модифицируете исходный параметр. Будьте осторожны, и тщательно распаковывайте `@_`.

Функции и пространства имён

Каждая функция содержится в некотором пространстве имён . Функции в необъявленном пространстве имён — функции, объявление которых не расположено после явной директивы `package` — находятся в пространстве имён `main`. Кроме того, вы можете объявить функцию в другом пространстве имён, добавив префикс к её имени:

```
sub Extensions::Math::add {
    ...
}
```

Это объявит функцию и создаст необходимое пространство имён. Помните, что пакеты в Perl 5 открыты для изменений в любой точке. Вы можете объявить только одну функцию с определённым именем в одном пространстве

имён. В противном случае Perl 5 выдаст предупреждение о переопределении функции. Отключите эти предупреждения с помощью `no warnings 'redefine'` — если точно уверены, что это именно то, что вам нужно.

Вызов функций в других пространствах имён осуществляется с помощью их полностью определённых имён:

```
package main;

Extensions::Math::add( $scalar, $vector );
```

Функции в пространствах имён *видимы* снаружи этих пространств имён по их полностью определённым именам. Внутри пространства имён вы можете использовать короткие имена для доступа к любой функции, объявленной в этом пространстве имён. Также вы можете импортировать имена из других пространств имён.

Импорт

При загрузке модуля с помощью встроенной директивы `use` , Perl автоматически вызывает метод `import()` этого модуля. Модули могут предоставлять свои собственные методы `import()`, делающие некоторые или все определённые символы доступными в вызывающем пакете. Любые аргументы после имени модуля в директиве `use` будут переданы в метод `import()` модуля. Так:

```
use strict;
```

...загружает модуль `strict.pm` и вызывает `strict->import()` без аргументов, тогда как:

```
use strict 'refs';
use strict qw( subs vars );
```

...загружает модуль `strict.pm`, вызывает `strict->import( 'refs' )`, а затем вызывает `strict->import( 'subs', vars' )`.

`use` имеет специальное поведение по отношению к `import()`, но вы можете вызвать `import()` и напрямую. Пример использования `use` эквивалентен следующему:

```
BEGIN
{
    require strict;
    strict->import( 'refs' );
    strict->import( qw( subs vars ) );
}
```

Встроенная директива `use` добавляет неявный блок `BEGIN` вокруг этих выражений, так что вызов `import()` происходит *сразу* после того, как парсер скомпилирует всю директиву `use`. Это гарантирует, что любые импортируемые символы будут видимы при компиляции остальной части программы. В противном случае любые функции, *импортированные* из других модулей, но не *объявленные* в текущем файле, будут выглядеть как голые слова, тем самым нарушая режим `strict`.

Оповещение об ошибках

Внутри функции проверить контекст её вызова можно с помощью встроенной функции `caller`. Если не передано никаких аргументов, она возвращает список из трёх элементов, содержащий имя вызывающего пакета, имя файла, содержащего вызов, и номер строки файла, на которой произошёл вызов:

```
package main;

main();

sub main
{
```

```
    show_call_information();
}

sub show_call_information
{
    my ($package, $file, $line) = caller();
    say "Called from $package in $file:$line";
}
```

Для инспектирования доступна полная цепочка вызовов. Передайте единственный целочисленный аргумент *n* в `caller()` чтобы просмотреть вызов вызова вызова *n* раз. Другими словами, если бы `show_call_information()` использовала `caller(0)`, то получила бы информацию о вызове из `main()`. Если же она использовала бы `caller(1)`, то получила бы информацию о вызове из начала программы.

Этот необязательный аргумент также требует от `caller` предоставить дополнительные возвращаемые значения, включая имя функции и контекст вызова:

```
sub show_call_information
{
    my ($package, $file, $line, $func) = caller(0);
    say "Called $func from $package in $file:$line";
}
```

Стандартный модуль `Carp` успешно использует эту технику для оповещения об ошибках и выбрасывания предупреждений в функциях. При использовании вместо `die` в коде библиотеки, `croak()` выбрасывает исключение с точки зрения своего вызывающего кода. `carp()` сообщает о предупреждении из файла, сообщая номер строки своего вызывающего кода .

Это поведение наиболее полезно при валидации параметров и предварительных условий фукнций, для указания того, что вызывающий код в чём-то неверен.

Валидация аргументов

Хотя Perl старается как может делать то, что имеет ввиду программист, он предоставляет немного нативных способов проверки валидности аргументов, передаваемых в функции. Для определения того, что *количество* параметров, переданных в функцию, корректно, вычислите `@_` в скалярном контексте:

```
sub add_numbers
{
    croak 'Expected two numbers, received: ' . @_
        unless @_ == 2;

    ...
}
```

Проверка типа более трудна из-за свойственного Perl оператор-ориентированного преобразования типов . Модуль `Params::Validate` из CPAN предлагает больше точности.

Продвинутые функции

Функции — фундамент многих продвинутых возможностей Perl.

Учёт контекста

Встроенные функции Perl 5 знают, вызвали ли вы их в пустом, скалярном или списочном контексте. Как могут и ваши функции. Неудачно названная_(См. `perldoc -f wantarray`) для подтверждения._ директива `wantarray` возвращает `undef` для обозначения пустого контекста, ложное значение для обозначения скалярного контекста и истинное значение для обозначения списочного контекста.

```
sub context_sensitive
```

```
{
    my $context = wantarray();

    return qw( List context )   if         $context;
    say    'Void context'   unless defined $context;
    return 'Scalar context' unless         $context;
}

context_sensitive();
say my $scalar = context_sensitive();
say context_sensitive();
```

Это может быть полезно для функций, которые могут производить дорогие возвращаемые значения, чтобы избежать этого в пустом контексте. Некоторые идиоматические функции возвращают список в списочном контексте и первый элемент списка или ссылку на массив в скалярном контексте. Но помните, что для использования `wantarray` не существует единых наилучших рекомендаций. Иногда понятнее будет написать отдельные однозначные функции.

Рекурсия

Предположим, вы хотите найти элемент в отсортированном массиве. Вы *можете* в поисках цели обойти в цикле каждый элемент массива отдельно, но в среднем вам придётся проверить половину элементов в массиве. Другой подход — разделить массив пополам, выбрать элемент посередине, сравнить, затем повторить либо с нижней, либо с верхней половиной. Разделяй и властвуй. Когда закончатся элементы для проверки или будет найден искомый, остановитесь.

Автоматизированный тест для этой техники может быть таким:

```
use Test::More;

my @elements =
(
    1, 5, 6, 19, 48, 77, 997, 1025, 7777, 8192, 9999
```

```
);

ok   elem_exists(     1, @elements ),
        'found first element in array';
ok   elem_exists(  9999, @elements ),
         'found last element in array';
ok ! elem_exists(   998, @elements ),
        'did not find element not in array';
ok ! elem_exists(    -1, @elements ),
        'did not find element not in array';
ok ! elem_exists( 10000, @elements ),
        'did not find element not in array';

ok   elem_exists(    77, @elements ),
        'found midpoint element';
ok   elem_exists(    48, @elements ),
        'found end of lower half element';
ok   elem_exists(   997, @elements ),
        'found start of upper half element';

done_testing();
```

Рекурсия — обманчиво простое понятие. Каждый вызов функции в Perl создаёт новый *фрейм вызова*, внутреннюю структуру данных, представляющую сам вызов, включая лексическое окружение текущего вызова функции. Это значит, что функция может вызвать сама себя, *рекурсивно*.

Чтобы добиться прохождения предыдущего теста, напишите функцию под названием `elem_exists()`, которая знает, как вызывать себя, каждый раз деля список пополам:

```
sub elem_exists
{
    my ($item, @array) = @_;

   # прервать рекурсию, если не осталось элементов для поиска
    return unless @array;

    # смещаемся вниз при нечётном количестве элементов
```

```
    my $midpoint = int( (@array / 2) - 0.5 );
    my $miditem  = $array[ $midpoint ];

    # возвращаем истину если элемент найден
    return 1 if $item  == $miditem;

    # возвращаем ложь, если остался только один элемент
    return   if @array == 1;

    # разбиваем массив надвое и продолжаем рекурсивно
    return elem_exists(
        $item, @array[0 .. $midpoint]
    ) if $item < $miditem;

    # разбиваем массив надвое и продолжаем рекурсивно
    return elem_exists(
         $item, @array[ $midpoint + 1 .. $#array ]
    );
}
```

Хотя вы *можете* написать этот код в процедурном виде и самостоятельно управлять делением списка, этот рекурсивный подход позволяет Perl управлять процессом.

Лексические переменные

Каждый новый вызов функции создаёт собственный *экземпляр* лексической области видимости. Даже несмотря на то, что объявление `elem_exists()` создаёт единственную область видимости для лексических переменных `$item`, `@array`, `$midpoint` и `$miditem`, каждый *вызов* `elem_exists()` — даже рекурсивный — сохраняет значения этих лексических переменных отдельно.

`elem_exists()` не только может вызывать саму себя, но и лексические переменные каждого вызова будут защищены и разделены:

```
use Carp 'cluck';
```

```
sub elem_exists
{
    my ($item, @array) = @_;

    cluck "[$item] (@array)";

    # далее следует другой код
    ...
}
```

Хвостовые вызовы

Один из *недостатков* рекурсии — то, что вы должны корректно указать условия возврата, иначе ваша функция вызовет себя саму бесконечное количество раз. По этой причине функция `elem_exists()` имеет несколько директив `return`.

Perl выводит полезное предупреждение `Deep recursion on subroutine`, если подозревает неконтролируемую рекурсию. Ограничение в 100 рекурсивных вызовов произвольно, но зачастую удобно. Отключите это предупреждение, указав `no warnings 'recursion'` в области видимости рекурсивного вызова.

Так как каждый вызов функции требует нового фрейма вызова и пространства для хранения лексических переменных, глубокорекурсивный код может использовать больше памяти, чем итеративный код. *Устранение хвостовых вызовов* может помочь.

Хвостовой вызов — это вызов функции, который напрямую возвращает результаты этой функции. Эти рекурсивные вызовы `elem_exists()`:

```
# разбиваем массив надвое и продолжаем рекурсивно
return elem_exists(
    $item, @array[0 .. $midpoint]
) if $item < $miditem;

# разбиваем массив надвое и продолжаем рекурсивно
```

```
return elem_exists(
     $item, @array[ $midpoint + 1 .. $#array ]
);
```

...кандидаты для устранения хвостовых вызовов. Эта оптимизация позволит избежать возврата в текущий вызов и последующего возврата в родительский вызов. Вместо этого, она возвращается в родительский вызов напрямую.

К сожалению, Perl 5 не устраняет хвостовые вызовы автоматически. Это делается вручную с помощью специальной формы встроенной директивы `goto`. В отличие от формы, зачастую производящей спагетти-код, функциональная форма `goto` заменяет текущий вызов функции вызовом другой функции. Вы можете указывать функцию с помощью имени или ссылки. Для передачи аргументов присваивайте напрямую в `@_`:

```
# разбиваем массив надвое и продолжаем рекурсивно
if ($item < $miditem)
{
    @_ = ($item, @array[0 .. $midpoint]);
    goto &elem_exists;
}

# разбиваем массив надвое и продолжаем рекурсивно
else
{
    @_ = ($item, @array[$midpoint + 1 .. $#array] );
    goto &elem_exists;
}
```

Иногда оптимизации довольно уродливы.

Ловушки и недостатки

Perl 5 всё ещё поддерживает старомодный вызов функций, полученный в наследство от старших версий Perl. Тогда как сейчас вы можете вызывать Perl-функции по имени, предыдущие версии Perl требовали их вызова

с использованием ведущего символа амперсанда (&). Perl 1 требовал использования встроенной директивы do:

```
# устаревший стиль; избегайте
my $result = &calculate_result( 52 );

# стиль Perl 1; избегайте
my $result = do calculate_result( 42 );

# сумасшедшая мешанина; в самом деле избегайте, правда
my $result = do &calculate_result( 42 );
```

Кроме того, что этот рудиментарный синтаксис — визуальный хаос, форма с ведущим амперсандом имеет несколько неожиданных поведений. Во-первых, она отключает любую проверку прототипа. Во-вторых, она *неявно* передаёт содержимое @_ немодифицированным, если вы сами не укажете аргументы. И то, и другое может приводить к неожиданному поведению.

Последняя ловушка проистекает из опускания скобок из вызова фукнции. Парсер Perl 5 использует несколько эвристических методов для разрешения неоднозначных голых слов и количества параметров, передаваемых в функцию. Эвристики могут ошибаться:

```
# внимание; содержит хитрый баг
ok elem_exists 1, @elements, 'found first element';
```

Вызов elem_exists() поглотит описание теста, которое должно было быть вторым аргументом ok(). Так как elem_exists() использует проглатывающий второй параметр, это может пройти незамеченным до тех пор, пока Perl не выдаст предупреждение о сравнении нечислового значения (описание теста, которое он не может сконвертировать в число) с элементом в массиве.

Хотя лишние скобки могут препятствовать читаемости, обдуманное использование скобок может прояснить код и сделать коварные баги маловероятными.

Область видимости

Область видимости в Perl обозначает продолжительность жизни и видимость именованных сущностей. В Perl всё, что имеет имя (переменная, функция), имеет область видимости. Ограничение области видимости помогает усилить *инкапсуляцию* — сохранение связанных понятий вместе и предотвращение их вытекания наружу.

Лексическая область видимости

Лексическая область видимости — это область видимости, видимая по ходу *чтения* программы. Компилятор Perl определяет эту область видимости во время компиляции. Блок, ограниченный фигурными скобками, создаёт новую область видимости, будь это голый блок, блок конструкции цикла, блок объявления `sub`, блок `eval` или любой другой блок, не заключающий в кавычки.

Лексическая область видимости определяет видимость переменных, объявленных с помощью `my` — *лексических* переменных. Лексическая переменная, объявленная в одной области видимости, видима в этой области видимости и любых областях видимости, вложенных в неё, но невидима в областях видимости, находящихся на том же уровне вложенности или выше:

```
# внешняя лексическая область видимости
{
    package Robot::Butler

    # внутренняя лексическая область видимости
    my $battery_level;

    sub tidy_room
    {
        # более глубокая вложенная область видимости
        my $timer;

        do {
           # самая глубокая лексическая область видимости
```

```
            my $dustpan;
            ...
        } while (@_);

    # лексическая область видимости, находящаяся на том же уровн
        for (@_)
        {
        # отдельная самая глубокая лексическая область видимости
            my $polish_cloth;
            ...
        }
    }
}
```

...переменная `$battery_level` видима во всех четырёх областях видимости. `$timer` видима в методе, блоке `do` и цикле `for`. `$dustpan` видима только в блоке `do`, а `$polish_cloth` — только в цикле `for`.

Объявление во внутренней области видимости лексической переменной с тем же именем, что и лексическая переменная во внешней области видимости, скрывает, или *затеняет*, внешнюю переменную во внутренней области видимости. Зачастую это именно то, что вам нужно:

```
my $name = 'Jacob';

{
    my $name = 'Edward';
    say $name;
}

say $name;
```

Эта программа выводит `Edward` и затем `Jacob`*(Члены семьи, а не вампиры.)*, несмотря на то, что переобъявление лексической переменной с тем же именем и типом *в той же самой лексической области видимости* приводит к предупреждающему сообщению. Затенение лексической переменной — особенность инкапсуляции.

Некоторые лексические объявления имеют тонкости, как, например, лексическая переменная, используемая как переменная-итератор в цикле `for`. Её объявление находится снаружи блока, но её область видимости — *внутри* блока цикла:

```
my $cat = 'Brad';

for my $cat (qw( Jack Daisy Petunia Tuxedo Choco ))
{
    say "Iterator cat is $cat";
}

say "Static cat is $cat";
```

Аналогично, `given` создаёт *лексический топик* (как `my $_`) внутри своего блока:

```
$_ = 'outside';

given ('inner')
{
    say;
    $_ = 'this assignment does nothing useful';
}

say;
```

...так что выход из блока восстанавливает предыдущее значение `$_`.

Функции — именованные и анонимные — создают лексическую область видимости для своих тел. Это помогает созданию замыканий .

Область видимости our

Внутри заданной области видимости можно объявить псевдоним переменной пакета с помощью встроенной директивы `our`. Как и `my`, `our` обеспечивает

лексическую область видимости псевдонима. Полностью определённое имя доступно из любого места, но лексический псевдоним виден только внутри своей области видимости.

`our` наиболее полезна в применении к глобальным переменным пакета, таким как `$VERSION` и `$AUTOLOAD`.

Динамическая область видимости.

Динамическая область видимости похожа на лексическую область видимости по своим правилам видимости, но вместо поиска наружу в области видимости времени компиляции, поиск проходит назад через контекст вызова. В то время как глобальная переменная пакета может быть *видима* внутри всех областей видимости, её *значение* изменяется в зависимости от локализации (`local`) и присваивания:

```
our $scope;

sub inner
{
    say $scope;
}

sub main
{
    say $scope;
    local $scope = 'main() scope';
    middle();
}

sub middle
{
    say $scope;
    inner();
}

$scope = 'outer scope';
main();
```

```
say $scope;
```

Программа начинается с объявления `our`-переменной, `$scope`, а так же трёх функций. Заканчивается она присваиванием `$scope` и вызовом `main()`.

Внутри `main()` программа выводит текущее значение `$scope`, `outer scope`, затем локализует переменную с помощью `local`. Это изменяет видимость символа внутри текущей лексической области видимости, *так же как* и в любых функциях, вызываемых из *текущей* лексической области видимости. Таким образом, `$scope` содержит `main()  scope` внутри тела как `middle()`, так и `inner()`. После возврата из `main()`, когда поток управления достигает конца её блока, Perl восстанавливает исходное значение локализованной `$scope`. Последняя директива `say` снова выводит `outer scope`.

Переменные пакетов и лексические переменные в Perl имеют разные правила видимости и механизмы хранения. Каждая область видимости, содержащая лексические переменные, имеет специальную структуру данных, называемую *лексическая записная книжка* (*lexical pad*, или *lexpad*), которая может хранить значения содержащихся в ней лексических переменных. Каждый раз, когда поток управления входит в одну из этих областей видимости, Perl создаёт ещё одну лексическую записную книжку для значений этих лексических переменных для этого конкретного вызова. Это позволяет функции работать корректно, особенно в рекурсивных вызовах .

Каждый пакет имеет единую *символьную таблицу*, содержащую переменные пакета, а также именованные функции. Импорт инспектирует и манипулирует этой символьной таблицей. Также делает и `local`. Вы можете локализовать только глобальные переменные и глобальные переменные пакета — но не лексические переменные.

`local` чаще всего используется с магическими переменными. Например, `$/`, разделитель входных записей, определяет, сколько данных операция `readline` прочитает из дескриптора файла. `$!`, переменная системной ошибки, содержит номер ошибки последнего системного вызова. `$@`, переменная ошибки `eval`, содержит любую ошибку из последней операции `eval`. `$|`, переменная автосброса, определяет, будет ли Perl сбрасывать текущий выбранный (`select`) дескриптор файла после каждой операции записи.

Локализация этих переменных в как можно более узкой области видимости ограничивает эффект ваших изменений. Это может предотвратить странное поведение в других частях вашего кода.

Область видимости state

Perl 5.10 добавил новую область видимости для поддержки встроенной директивы `state`. Область видимости state похожа на лексическую область видимости в отношении условий видимости, но добавляет однократную инициализацию, а также персистентность значения:

```
sub counter
{
    state $count = 1;
    return $count++;
}

say counter();
say counter();
say counter();
```

При первом вызове `counter` Perl выполняет единственную инициализацию `$count`. При последующих вызовах `$count` сохраняет своё предыдущее значение. Эта программа выведет `1`, `2` и `3`. Замените `state` на `my`, и программа будет выводить `1`, `1` и `1`.

Вы можете использовать выражение для установки начального значения `state`-переменной:

```
sub counter
{
    state $count = shift;
    return $count++;
}

say counter(2);
```

```
say counter(4);
say counter(6);
```

Несмотря на то, что простое прочтение кода может привести к предположению, что выведено будет 2, 4 и 6, на самом деле выведено будет 2, 3 и 4. Первый вызов процедуры `counter` установит значение переменной `$count`. Последующие вызовы не будут изменять её значение.

`state` может быть полезна для установки значения по умолчанию или подготовки кеша, но если вы её используете, убедитесь, что понимаете её инициализационное поведение:

```
sub counter
{
    state $count = shift;
    say 'Second arg is: ', shift;
    return $count++;
}

say counter(2, 'two');
say counter(4, 'four');
say counter(6, 'six');
```

Счётчик для этой программы выведет 2, 3 и 4, как и ожидалось, но значения подразумеваемого второго аргумента вызовов `counter()` будут `two`, 4 и 6 — потому что `shift` для первого аргумента происходит только при первом вызове `counter()`. Или измените API для предотвращения этой ошибки, или защититесь от неё:

```
sub counter
{
    my ($initial_value, $text) = @_;

    state $count = $initial_value;
    say "Second arg is: $text";
    return $count++;
}
```

```
say counter(2, 'two');
say counter(4, 'four');
say counter(6, 'six');
```

Анонимные функции

Анонимная функция — это функция без имени. В остальном она ведёт себя в точности так же, как именованная функция — вы можете вызывать её, передавать в неё аргументы, возвращать из неё значения и копировать ссылки на неё. Однако единственный способ работать с ней — по ссылке .

Распространённая идиома Perl 5, известная как *таблица диспетчеризации*, использует хеши для сопоставления входных данных с соответствующим поведением:

```
my %dispatch =
(
    plus     => \&add_two_numbers,
    minus    => \&subtract_two_numbers,
    times    => \&multiply_two_numbers,
);

sub add_two_numbers      { $_[0] + $_[1] }
sub subtract_two_numbers { $_[0] - $_[1] }
sub multiply_two_numbers { $_[0] * $_[1] }

sub dispatch
{
    my ($left, $op, $right) = @_;

    return unless exists $dispatch{ $op };

    return $dispatch{ $op }->( $left, $right );
}
```

Функция `dispatch()` принимает аргументы в форме `(2, 'times', 2)`

и возвращает результат вычисления операции.

Объявление анонимных функций

Встроенная директива `sub`, использованная без имени, создаёт и возвращает анонимную функцию. Используйте эту ссылку на функцию в любом месте, где вы могли бы использовать ссылку на именованную функцию, как, например, объявление функций таблицы диспетчеризации прямо на месте:

```
my %dispatch =
(
    plus      => sub { $_[0]  + $_[1] },
    minus     => sub { $_[0]  - $_[1] },
    times     => sub { $_[0]  * $_[1] },
    dividedby => sub { $_[0]  / $_[1] },
    raisedto  => sub { $_[0] ** $_[1] },
);
```

Вам также может встретиться передача анонимных функций как аргументов функции:

```
sub invoke_anon_function
{
    my $func = shift;
    return $func->( @_ );
}

sub named_func
{
    say 'I am a named function!';
}

invoke_anon_function( \&named_func );
invoke_anon_function( sub { say 'Who am I?' } );
```

Имена анонимных функций

Имея ссылку на функцию, вы можете определить, именованная она или анонимная, с помощью интроспекции_(...или с помощью `sub_name` из CPAN-модуля `Sub::Identitiy`._:

```
package ShowCaller;

sub show_caller
{
    my ($package, $file, $line, $sub) = caller(1);
    say "Called from $sub in $package:$file:$line";
}

sub main
{
    my $anon_sub = sub { show_caller() };
    show_caller();
    $anon_sub->();
}

main();
```

Результат может быть неожиданным:

```
Called from ShowCaller::main
         in ShowCaller:anoncaller.pl:20
Called from ShowCaller::__ANON__
         in ShowCaller:anoncaller.pl:17
```

`__ANON__` во второй строке вывода показывает, что анонимная функция не имеет имени, которое Perl мог бы распознать. Это может усложнить отладку. Функция `subname()` из CPAN-модуля `Sub::Name` позволяет вам закреплять имена за анонимными функциями:

```
use Sub::Name;
```

```
use Sub::Identify 'sub_name';

my $anon  = sub {};
say sub_name( $anon );

my $named = subname( 'pseudo-anonymous', $anon );
say sub_name( $named );
say sub_name( $anon );

say sub_name( sub {} );
```

Эта программа выведет:

```
__ANON__
pseudo-anonymous
pseudo-anonymous
__ANON__
```

Имейте ввиду, что обе ссылки ссылаются на одну и ту же лежащую в основе функцию. Использование `subname()` на одной ссылке на функцию будет изменять имя этой анонимной функции так, что все остальные ссылки на неё будут видеть это новое имя.

Неявные анонимные функции

Perl 5 позволяет объявлять анонимные фукнции неявно, используя прототипы . Хотя официально эта возможность существует для того, чтобы дать программистам возможность писать свой собственный синтаксис, подобный `map` и `eval`, интересным примером является использование *отложенных* функций, которые не выглядят как функции.

Рассмотрите CPAN-модуль `Test::Fatal`, который принимает анонимную функцию как первый аргумент своей функции `exception()`:

```
use Test::More;
```

```
use Test::Fatal;

my $croaker = exception { die 'I croak!' };
my $liver   = exception { 1 + 1 };

like( $croaker, qr/I croak/, 'die() should croak'   );
is(   $liver,   undef,       'addition should live' );

done_testing();
```

Вы можете переписать это более многословно следующим образом:

```
my $croaker = exception( sub { die 'I croak!' } );
my $liver   = exception( sub { 1 + 1 } );
```

... или передать именованную функцию по ссылке:

```
sub croaker { die 'I croak!' }
sub liver   { 1 + 1 }

my $croaker = exception \&croaker;
my $liver   = exception \&liver;

like( $croaker, qr/I croak/, 'die() should die'     );
is(   $liver,   undef,       'addition should live' );
```

...но вы *не* можете передавать их как скалярные ссылки:

```
my $croak_ref = \&croaker;
my $live_ref  = \&liver;

# ОШИБОЧНО: не работает
my $croaker   = exception $croak_ref;
my $liver     = exception $live_ref;
```

...потому что прототип изменяет то, как парсер Perl 5 интерпретирует этот код. Он не может со стопроцентной точностью определить, *что* будут содержать `$croaker` и `$liver`, и поэтому выбросит исключение.

Кроме того, имейте ввиду, что функция, принимающая анонимную функцию как первый из нескольких аргументов, не может иметь замыкающую запятую после блока функции:

```
use Test::More;
use Test::Fatal 'dies_ok';

dies_ok { die 'This is my boomstick!' }
        'No movie references here';
```

Это временами сбивающий с толку изъян в остальном полезного синтаксиса, учтивость в отношении причуд парсера Perl 5. Синтаксическая ясность, обеспечиваемая повышением голых блоков до анонимных функций, может быть полезной, но используйте её экономно и тщательно документируйте API.

Замыкания

Каждый раз, когда поток управления входит в функцию, эта функцию получает новое окружение, представляющее лексическую область видимости этого вызова. Это в той же мере применимо и к анонимным функциям . Следствия этого очень значимы. Термин компьютерных наук *функции высшего порядка* ссылается на функции, манипулирующие другими функциями. Замыкания — пример этой значимости.

Создание замыканий

Замыкание — это функция, использующая лексические переменные из внешней области видимости. Вы, вероятно, уже создавали и использовали замыкания, не осознавая этого:

```
package Invisible::Closure;

my $filename = shift @ARGV;

sub get_filename { return $filename }
```

Если этот код выглядит для вас очевидным, отлично! *Конечно* функция `get_filename()` может видеть лексическую переменную `$filename`. Именно так работают области видимости!

Предположим, вы хотите пройти в цикле по списку элементов без необходимости самостоятельно управлять итератором. Вы можете создать функцию, которая возвращает функцию, которая, будучи вызванной, вернёт следующий элемент в итерации:

```
sub make_iterator
{
    my @items = @_;
    my $count = 0;

    return sub
    {
        return if $count == @items;
        return $items[ $count++ ];
    }
}

my $cousins = make_iterator(
    qw( Rick Alex Kaycee Eric Corey Mandy Christine )
);

say $cousins->() for 1 .. 5;
```

Хотя и произошёл возврат из `make_iterator()`, анонимная функция, сохранённая в `$cousins`, замкнула значения этих переменных как они существовали внутри вызова `make_iterator()`. Их значения продолжают существовать .

Так как каждый вызов `make_iterator()` создаёт отдельное лексическое окружение, анонимная функция, которую она создаёт и возвращает, замыкается на уникальном лексическом окружении:

```
my $aunts = make_iterator(
    qw( Carole Phyllis Wendy Sylvia Monica Lupe )
);

say $cousins->();
say $aunts->();
```

Так как `make_iterator()` не возвращает эти лексические переменные по значению или по ссылке, никакой другой Perl-код, кроме замыкания, не сможет получить к ним доступ. Они инкапсулированы так же эффективно, как любая другая лексическая инкапсуляция, хотя любой код, разделяющий лексическое окружение, может получить доступ к этим значениям. Эта идиома предоставляет лучшую инкапсуляцию того, что в ином случае было бы глобальной переменной файла или пакета:

```
{
    my $private_variable;

    sub set_private { $private_variable = shift }
    sub get_private { $private_variable }
}
```

Имейте ввиду, что вы не можете *вкладывать* именованные функции. Именованные функции имеют глобальную область видимости пакета. Любые лексические переменные, разделяемые между вложенными функциями, перестанут быть разделяемыми, как только внешняя функция уничтожит своё первое лексическое окружение_(Если это приводит вас в замешательство, представьте реализацию.)_.

Использование замыканий

Итерирование по списку постоянного размера с помощью замыкания — интересный пример, но замыкания могут делать гораздо больше, как, например, итерирование по списку, который слишком дорог для вычисления или слишком велик, чтобы держать его в памяти целиком. Рассмотрим функцию, создающую последовательность чисел Фибоначчи по мере того, как вам понадобятся её элементы. Вместо повторного вычисления последовательности рекурсивно, используйте кеш и ленивое создание требуемых элементов:

```
sub gen_fib
{
    my @fibs = (0, 1);

    return sub
    {
        my $item = shift;

        if ($item >= @fibs)
        {
            for my $calc (@fibs .. $item)
            {
                $fibs[$calc] = $fibs[$calc - 2]
                             + $fibs[$calc - 1];
            }
        }
        return $fibs[$item];
    }
}
```

Каждый вызов функции, возвращаемой `get_fib()`, принимает один аргумент, *n*-ный элемент последовательности Фибоначчи. Функция генерирует все предшествующие значения в последовательности по мере необходимости, кеширует их и возвращает запрошенный элемент — и даже откладывает вычисление до абсолютной необходимости. Однако здесь шаблон, характерный для кеширования, переплетается с числовыми последовательностями. Что получится, если вы выделите связанный с кешированием код (инициализация

кеша, выполнение нужного кода для заполнения элементов кеша и возврат вычисленного или полученного из кеша значения) в функцию `generate_caching_closure()`?

```
sub gen_caching_closure
{
    my ($calc_element, @cache) = @_;

    return sub
    {
        my $item = shift;

        $calc_element->($item, \@cache)
            unless $item < @cache;

        return $cache[$item];
    };
}
```

Теперь `gen_fib()` превращается в:

```
sub gen_fib
{
    my @fibs = (0, 1, 1);

    return gen_caching_closure(
        sub
        {
            my ($item, $fibs) = @_;

            for my $calc ((@$fibs - 1) .. $item)
            {
                $fibs->[$calc] = $fibs->[$calc - 2]
                               + $fibs->[$calc - 1];
            }
        },
        @fibs
    );
```

```
}
```

Программа ведёт себя так же, как и раньше, но использование ссылок на функции и замыканий отделяет инициализирующее кеш поведение от вычисления следующего числа последовательности Фибоначчи. Кастомизация поведения кода — в данном случае `get_caching_closure()` — с помощью передачи функции обеспечивает потрясающую гибкость.

Замыкания и частичное применение

Также замыкания могут *удалить* нежелательную универсальность. Рассмотрим случай функции, принимающей несколько параметров:

```
sub make_sundae
{
    my %args = @_;

    my $ice_cream = get_ice_cream( $args{ice_cream} );
    my $banana    = get_banana( $args{banana} );
    my $syrup     = get_syrup( $args{syrup} );
    ...
}
```

Мириады возможностей кастомизации могут быть очень полезны в полноценном магазине мороженого, но в случае передвижного фургончика с мороженым, в котором подают только мороженое Французская ваниль на бананах Кавендиш, каждый вызов `make_sundae()` требует передачи аргументов, которые никогда не изменяются.

Техника, называемая *частичное применение*, позволяет вам привязать *некоторые* из аргументов к функции, а остальные предоставлять позже. Оберните функцию, которую намереваетесь вызывать, в замыкание, и передайте связанные аргументы.

Рассмотрим фургончик с мороженым, продающий только мороженое Французская ваниль на бананах Кавендиш:

```
my $make_cart_sundae = sub
{
    return make_sundae( @_,
        ice_cream => 'French Vanilla',
        banana    => 'Cavendish',
    );
};
```

Вместо вызова `make_sundae()` напрямую, вызовите ссылку на функцию в `$make_cart_sundae` и передайте только нужные аргументы, не опасаясь позабыть о неизменяющихся или передать их некорректно_(Вы можете даже использовать `Sub::Install`) из CPAN для импорта этой функции напрямую в другое пространство имён._.

Это только начало того, что вы можете делать с функиями высшего порядка. *Higher Order Perl* Марка Джейсона Доминуса (Mark Jason Dominus) — канонический справочник по функциям первого класса и замыканиям в Perl. Вы можете прочитать его онлайн по адресу http://hop.perl.plover.com/.

Состояния против замыканий

Замыкания пользуются возможностями лексической области видимости для предоставления опосредованного доступа к полу-приватным переменным. Даже именованные функции могут возпользоваться лексическими привязками:

```
{
    my $safety = 0;

    sub enable_safety  { $safety = 1 }
    sub disable_safety { $safety = 0 }

    sub do_something_awesome
    {
        return if $safety;
        ...
    }
}
```

Инкапсуляция функций для обеспечения безопасности позволяет всем трём функциям разделять состояние, не открывая лексическую переменную прямому воздействию внешнего кода. Эта идиома хорошо работает для тех случаев, когда внешний код должен иметь возможность изменять внутреннее состояние, но слишком тяжеловесна, если только одной функции нужно управлять состоянием.

Предположим, что каждый сотый покупатель вашего магазина мороженого получает бесплатную посыпку:

```
my $cust_count = 0;

sub serve_customer
{
    $cust_count++;

    my $order = shift;

    add_sprinkles($order) if $cust_count % 100 == 0;

    ...
}
```

Этот подход *работает*, но создание новой лексической области видимости для единственной функции создаёт больше сопутствующей сложности, чем нужно. Встроенная директива `state` позволяет вам объявлять переменные лексической области видимости со значением, сохраняющимся между вызовами:

```
sub serve_customer
{
    state $cust_count = 0;
    $cust_count++;

    my $order = shift;
    add_sprinkles($order)
        if ($cust_count % 100 == 0);
```

```
    ...
}
```

Вы должны явно подключить эту возможность, используя модуль `Modern::Perl`, прагму `feature` или требуя конкретную версию Perl 5.10 или выше (например, с помощью `use 5.010;` или `use 5.012;`).

`state` также работает внутри анонимных функций:

```
sub make_counter
{
    return sub
    {
        state $count = 0;
        return $count++;
     }
}
```

...хотя у этого подхода немного очевидных преимуществ.

Состояния против псевдосостояний

Perl 5.10 не рекомендует использование техники из предыдущих версий Perl, с помощью который вы могли в сущности эмулировать `state`. Именованная функция может замкнуться на своей предыдущей лексической области видимости с помощью злоупотребления нюансом реализации. Использование постфиксного условия, возвращающего ложь, с объявлением `my` позволяет избежать *повторной инициализации* лексической переменной значением `undef` или её инициализирующим значением.

Любое использование постфиксного условного выражения, модифицирующего объявление лексической переменной, теперь вызывает предупреждение о нерекомендуемости. С использованием этой техники слишком легко написать непреднамеренно ошибочный код; вместо этого используйте `state` там где доступно, или настоящие замыкания в ином случае. Перепишите эту идиому, если она вам попадётся:

```
sub inadvertent_state
{
   # my $counter  = 1 if 0; # НЕ РЕКОМЕНДУЕТСЯ; не используйте
     state $counter = 1;      # используйте вместо

     ...
}
```

Атрибуты

Именованные сущности в Perl — переменные и функции — могут иметь дополнительные метаданные, присоединённые к ним в виде *атрибутов*. Атрибуты — это произвольные имена и значения, используемые в определённых видах метапрограммирования .

Синтаксис объявления атрибутов довольно неуклюж, а использование атрибутов, в сущности, скорее искусство, чем наука. Большинство программ никогда их не используют, но при правильном использовании они обеспечивают ясность и удобство поддержки.

Использование атрибутов

Простой атрибут — это идентификатор с предшествующим ему двоеточием, привязанный к объявлению:

```
my $fortress      :hidden;

sub erupt_volcano :ScienceProject { ... }
```

Эти объявления будут приводить к вызову обработчиков атрибутов, названных `hidden` и `ScienceProject`, если они существуют для соответствующего типа (скаляров и функций соответственно). Эти обработчики могут делать *что угодно*. Если соответствующего обработчика не существует, Perl выбросит исключение времени компиляции.

Атрибуты могут включать списки параметров. Perl обрабатывает эти параметры как списки строковых и только строковых констант. Модуль `Test::Class` из CPAN с успехом использует такие параметрические аргументы:

```
sub setup_tests           :Test(setup)    { ... }
sub test_monkey_creation :Test(10)       { ... }
sub shutdown_tests        :Test(teardown) { ... }
```

Атрибут `Test` указывает на методы, содержащие тестовые проверки, и опционально устанавливает число проверок, которые метод намерен запустить. Хотя интроспекция этих классов может обнаружить соответствующие тестовые методы, имея хорошо спроектированные надёжные эвристики, атрибут `:Test` делает ваши намерения ясными.

Параметры `setup` и `teardown` позволяют тестовым классам определить свои собственные методы поддержки, не беспокоясь о конфликтах с другими подобными методами в других классах. Это отделяет проблему указания того, что этот класс должен делать, от проблемы того, как другие классы делают свою работу, и обеспечивает высокую гибкость.

Недостатки атрибутов

У атрибутов есть свои недостатки. Каноническая прагма для работы с атрибутами (прагма `attributes`) обозначала свой интерфейс как экспериментальный в течение многих лет. Базовый модуль `Attribute::Handlers` Демьена Конвея (Damian Conway) упрощает их реализацию. `Attribute::Lexical` Эндрю Мейна (Andrew Main) — более новый подход. По возможности предпочтите использование любого из них использованию `attributes`.

Наихудшая особенность атрибутов — их склонность создавать странные синтаксические действия на расстоянии. Имея фрагмент кода с атрибутами, можете ли вы предугадать их действие? Хорошо написанная документация помогает, но если невинно выглядящее объявление лексической переменной где-то сохраняет ссылку на эту переменную, ваши ожидания относительно срока жизни этой переменной могут быть ошибочны. Подобным же образом обработчик может обернуть функцию в другую функцию и заменить её в

символьной таблице, не осведомив вас об этом, — рассмотрите атрибут `:memoize`, который автоматически вызывает базовый модуль `Memoize`.

Атрибуты в вашем распоряжении, когда они нужны вам для решения сложных проблем. Они могут быть очень полезными при правильном использовании — но большинству программ они никогда не понадобятся.

AUTOLOAD

Perl не требует от вас объявлять каждую функцию, прежде чем её вызывать. Perl охотно попытается вызвать функцию, даже если она не существует. Расмотрим программу:

```
use Modern::Perl;

bake_pie( filling => 'apple' );
```

Когда вы её запустите, Perl вызовет исключение по причине вызова неопределённой функции `backe_pie()`.

Теперь добавьте функцию, называемую `AUTOLOAD()`:

```
sub AUTOLOAD {}
```

Теперь, когда вы запустите программу, ничего очевидного не произойдёт. Perl вызовет функцию пакета с именем `AUTOLOAD()` — если она существует — в случае, если обычная диспетчеризация не сработает. Измените `AUTOLOAD()` для вывода сообщения:

```
sub AUTOLOAD { say 'In AUTOLOAD()!' }
```

...чтобы продемонстрировать, что она вызывается.

Основные возможности AUTOLOAD

Функция `AUTOLOAD()` получает аргументы, переданные в неопределённую функцию, напрямую в `@_`, а *имя* неопределённой функции доступно в глобальной переменной пакета `$AUTOLOAD`. Манипулируйте этими аргументами по своему усмотрению:

```
sub AUTOLOAD
{
    our $AUTOLOAD;

    # оформленный вывод аргументов
    local $" = ', ';
    say "In AUTOLOAD(@_) for $AUTOLOAD!"
}
```

Объявление `our` ограничивает `$AUTOLOAD` в пределах тела функции. Переменная содержит полностью определённое имя неопределённой функции (в данном случае `main::bake_pie`). Удалить имя пакета можно с помощью регулярного выражения :

```
sub AUTOLOAD
{
    my ($name) = our $AUTOLOAD =~ /::(\w+)$/;

    # оформленный вывод аргументов
    local $" = ', ';
    say "In AUTOLOAD(@_) for $name!"
}
```

Оригинальный вызов получит то, что вернёт `AUTOLOAD()`:

```
say secret_tangent( -1 );

sub AUTOLOAD { return 'mu' }
```

До сих пор приведённые примеры всего лишь перехватывали вызовы к неопределённым функциям. Но есть и другие возможности.

Редиспетчеризация методов в AUTOLOAD()

Распространённый шаблон в ОО-программировании — *делегировать* или *проксировать* определённые методы одного объекта другому, зачастую содержащемуся в первом или доступному из него каким-либо иным способом. Проксирование логирования может помочь с отладкой:

```
package Proxy::Log;

sub new
{
    my ($class, $proxied) = @_;
    bless \$proxied, $class;
}

sub AUTOLOAD
{
    my ($name) = our $AUTOLOAD =~ /::(\w+)$/;
    Log::method_call( $name, @_ );

    my $self = shift;
    return $$self->$name( @_ );
}
```

Эта функция `AUTOLOAD()` логирует вызов метода. Затем она разыменовывает проксируемый объект из благословлённой скалярной ссылки, извлекает имя неопределённого метода, а затем вызывает этот метод на проксируемом объекте с переданными параметрами.

Генерация кода в AUTOLOAD()

Эта двойная диспетчеризация легка для написания, но неэффективна. Каждый вызов метода на прокси должен провалить нормальную диспетчеризацию, чтобы попасть в `AUTOLOAD()`. Вы можете заплатить эту цену лишь раз, устанавливая новые методы в прокси-класс по мере того, как они понадобятся программе:

```
sub AUTOLOAD
{
    my ($name) = our $AUTOLOAD =~ /::(\w+)$/;

    my $method = sub
    {
        Log::method_call( $name, @_ );

        my $self = shift;
        return $$self->$name( @_ );
    }

    no strict 'refs';
    *{ $AUTOLOAD } = $method;
    return $method->( @_ );
}
```

Тело предыдущей функции `AUTOLOAD()` стало замыканием , привязанным к *имени* неопределённого метода. Установка этого замыкания в соответствующую таблицу символов позволяет всем последующим обращениям к этому методу найти созданное замыкание (и обойти `AUTOLOAD()`). Наконец, этот код вызывает метод напрямую и возвращает результат.

Хотя этот подход чище и почти всегда прозрачнее, чем обработка поведения прямо в `AUTOLOAD()`, код, *вызванный* `AUTOLOAD()`, может определить, что обращение прошло через `AUTOLOAD()`. Короче говоря, `caller()` будет отображать двойную диспетчеризацию обеих приведённых техник. Хотя забота о том, что это происходит, может быть нарушением инкапсуляции, утечка деталей того, *как* объект предоставляет метод, тоже может быть нарушением инкапсуляции.

Некоторый код использует хвостовые вызовы для *замены* текущего вызова AUTOLOAD() вызовом целевого метода:

```
sub AUTOLOAD
{
    my ($name) = our $AUTOLOAD =~ /::(\w+)$/;
    my $method = sub { ... }

    no strict 'refs';
    *{ $AUTOLOAD } = $method;
    goto &$method;
}
```

Это приводит к тому же эффекту, что и вызов $method напрямую, за исключением того, что AUTOLOAD() не будет появляться в списке вызовов, доступном через caller(), так что это будет выглядеть так, как будто сгенерированный метод был просто вызван напрямую.

Недостатки AUTOLOAD

AUTOLOAD() может быть полезным инструментом, хотя его и непросто использовать правильно. Наивный подход к генерированию методов во время исполнения означает, что метод can() не будет выдавать правильную информацию о возможностях объектов и классов. Самое простое решение — предварительно объявить все функции, которые вы планируете генерировать в AUTOLOAD(), с помощью прагмы subs:

```
use subs qw( red green blue ochre teal );
```

Плюсом этой техники является документирование ваших намерений, но недостаток в том, что вам придётся поддерживать статический список функций или методов. Перегрузка can() иногда работает лучше:

```
sub can
```

```
{
    my ($self, $method) = @_;

    # использовать результат родительского can()
    my $meth_ref = $self->SUPER::can( $method );
    return $meth_ref if $meth_ref;

    # добавить фильтрацию
    return unless $self->should_generate( $method );

    $meth_ref = sub { ... };
    no strict 'refs';
    return *{ $method } = $meth_ref;
}

sub AUTOLOAD
{
    my ($self) = @_;
    my ($name) = our $AUTOLOAD =~ /::(\w+)$/;>

    return unless my $meth_ref = $self->can( $name );
    goto &$meth_ref;
}
```

`AUTOLOAD()` — большой молоток; он может ловить функции и методы, которые вы не собирались автозагружать, такие как `DESTROY()`, деструктор объектов. Если напишете метод `DESTROY` без имплементации, Perl с удовольствием направится к нему вместо `AUTOLOAD()`:

```
# пропустить AUTOLOAD()
sub DESTROY {}
```

Если вы смешаете функции и методы в едином пространстве имён, наследующем от другого пакета, имеющего собственную `AUTOLOAD()`, вы можете увидеть странную ошибку:

Если с вами это произошло, упростите свой код; вы вызвали несуществующую функцию в пакете, наследующем от класса, содержащего свою собственную

AUTOLOAD(). Проблема состоит из нескольких частей: смешивание функций и методов в одном пространстве имён зачастую является недостатком дизайна, наследование и AUTOLOAD() очень быстро становятся сложными, и делать выводы о коде, когда вы не знаете, какие методы предоставляет объект, трудно.

AUTOLOAD() полезна для быстрого и грязного программирования, но надёжный код её избегает.

8. Регулярные выражения и сопоставление

Мощь Perl в обработке текста проистекает из использования им *регулярных выражений*. Регулярное выражение (*regular expression*, *regex* или *regexp*) — это *шаблон*, описывающий свойства куска текста. *Механизм регулярных выражений* интерпретирует шаблоны и применяет их для сопоставления или изменения кусков текста.

Базовая документация Perl 5 по регулярным выражениям включает туториал (`perldoc perlretut`), справочное руководство (`perldoc perlreref`) и полную документацию (`perldoc perlre`). Книга Джефри Фридла (Jeffrey Friedl) *Освоение регулярных выражений* (*Mastering Regular Expressions*) объясняет теорию и механику того, как работают регулярные выражения. Хотя овладение регулярными выражениями — непростой процесс, даже небольшое знание даст вам большие возможности.

Литералы

Регулярные выражения могут быть простыми шаблонами подстрок:

```
my $name = 'Chatfield';
say 'Found a hat!' if $name =~ /hat/;
```

Оператор сопоставления (*match*, `m//`, сокращёно `//`) распознаёт регулярное выражение, в этом примере — `hat`. Этот шаблон — *не* слово. Он означает «символ `h`, за которым следует символ `a`, за которым следует символ `t`». Каждый символ в шаблоне — неделимый элемент, или *атом*. Он либо совпадает, либо нет.

Оператор связывания с регулярным выражением (`=~`) — инфиксный оператор , который применяет регулярное выражение, являющееся его вторым операндом к строке, указанной в качестве первого операнда. При использовании в скалярном контексте, сопоставление возвращает истинное значение в случае успеха. Отрицательная форма оператора связывания (`!~`) возвращает истинное значение в случае неудачного сопоставления.

Оператор замены, `s///`, это, в некотором смысле, циркумфиксный оператор с двумя операндами. Его первый операнд — регулярное выражение для поиска соответствия при использовании с оператором связывания с регулярным выражением. Второй операнд — подстрока, используемая для замены сопоставленной части первого операнда. Например, вылечить надоедливые летние аллергии можно так:

```
my $status = 'I feel ill.';
$status    =~ s/ill/well/;
say $status;
```

Оператор qr// и объединения регулярных выражений

Оператор `qr//` создаёт регулярные выражения первого класса. Для использования интерполируйте их в оператор сопоставления:

```
my $hat = qr/hat/;
say 'Found a hat!' if $name =~ /$hat/;
```

...или объедините несколько регулярных выражений в составные шаблоны:

```
my $hat   = qr/hat/;
my $field = qr/field/;

say 'Found a hat in a field!'
    if $name =~ /$hat$field/;

like( $name, qr/$hat$field/,
              'Found a hat in a field!' );
```

Квантификаторы

Регулярные выражения становятся более мощными благодаря использованию *квантификаторов регулярных выражений*, который позволяют вам указать,

сколько раз компонент регулярного выражения может появляться в строке. Самый простой квантификатор — *квантификатор «ноль или один»*, или `?`:

```
my $cat_or_ct = qr/ca?t/;

like( 'cat', $cat_or_ct, "'cat' matches /ca?t/" );
like( 'ct',  $cat_or_ct, "'ct' matches /ca?t/"  );
```

Любой атом в регулярном выражении, за которым следует символ `?`, означает «найти ноль или одни таких атомов». Это регулярное выражение проходит сопоставление, если один или более символов `a` следуют сразу за символом `c` и непосредственно предшествуют символу `t`, будь это литеральная подстрока `cat` или `ct`.

Квантификатор «один или более», или `+`, проходит сопоставление только если есть как минимум одно вхождение квантифицируемого атома:

```
my $some_a = qr/ca+t/;

like( 'cat',    $some_a, "'cat' matches /ca+t/" );
like( 'caat',   $some_a, "'caat' matches/"      );
like( 'caaat',  $some_a, "'caaat' matches"      );
like( 'caaaat', $some_a, "'caaaat' matches"     );

unlike( 'ct',   $some_a, "'ct' does not match"  );
```

Нет теоретического ограничения максимального количества квантифицированных атомов, которые могут пройти сопоставление.

Квантификатор «ноль или более», `*`, находит соответствие ноля или более экземпляров квантифицируемого атома:

```
my $any_a = qr/ca*t/;

like( 'cat',    $any_a, "'cat' matches /ca*t/" );
like( 'caat',   $any_a, "'caat' matches"       );
```

```
like( 'caaat',  $any_a, "'caaat' matches"      );
like( 'caaaat', $any_a, "'caaaat' matches"     );
like( 'ct',     $any_a, "'ct' matches"         );
```

Как бы глупо это ни выглядело, это позволяет вам указать необязательные компоненты регулярного выражения. Всё же, используйте это экономно: это грубый и дорогой инструмент. *Большинство* регулярных выражений получают гораздо больше выгоды от использования `?` и `+`, чем от `*`. Точность намерений зачастую увеличивает ясность.

Количественные квантификаторы выражают конкретное количество возможных повторений атома. `{n}` означает, что совпадение должно быть найдено ровно *n* раз.

```
# эквивалентно qr/cat/;
my $only_one_a = qr/ca{1}t/;

like( 'cat', $only_one_a, "'cat' matches /ca{1}t/" );
```

`{n,}` требует *как минимум n* повторений:

```
# эквивалентно qr/ca+t/;
my $some_a = qr/ca{1,}t/;

like( 'cat',    $some_a, "'cat' matches /ca{1,}t/" );
like( 'caat',   $some_a, "'caat' matches"          );
like( 'caaat',  $some_a, "'caaat' matches"         );
like( 'caaaat', $some_a, "'caaaat' matches"        );
```

`{n,m}` означает, что должно быть как минимум *n*, но не более *m* совпадений:

```
my $few_a = qr/ca{1,3}t/;

like( 'cat',    $few_a, "'cat' matches /ca{1,3}t/" );
like( 'caat',   $few_a, "'caat' matches"           );
```

```
like( 'caaat',  $few_a, "'caaat' matches"          );

unlike( 'caaaat', $few_a, "'caaaat' doesn't match" );
```

Вы можете выразить символьные квантификаторы в терминах количественных квантификаторов, но большинство программ, как правило, используют первые.

Жадность

Квантификаторы `+` и `*` — *жадные*, они пытаются сопоставить настолько большую часть входной строки, насколько можно. Это особенно пагубное поведение. Рассмотрите наивное использование шаблона «ноль или более символов, не являющихся символами перевода строки» `.*`:

```
# плохое регулярное выражение
my $hot_meal = qr/hot.*meal/;

say 'Found a hot meal!'
    if 'I have a hot meal' =~ $hot_meal;

say 'Found a hot meal!'
     if 'one-shot, piecemeal work!' =~ $hot_meal;
```

Жадные квантификаторы начинают с того, что сопоставляют *всё*, а затем возвращаются на один символ за раз только если очевидно, что сопоставление не успешно.

Модификатор `?`, добавленный к квантификатору, делает жадный квантификатор бережливым:

```
my $minimal_greedy = qr/hot.*?meal/;
```

При использовании нежадного квантификатора, механизм регулярных выражений предпочтёт по возможности *наикратчайшее* потенциальное

совпадение и будет увеличивать количество символов, идентифицируемых токеном `.*?`, только если текущее количество не проходит сопоставление. Поскольку * требует нуля или более повторений, минимальное потенциальное совпадение для этого токена — ноль символов:

```
say 'Found a hot meal'
if 'ilikeahotmeal' =~ /$minimal_greedy/;
```

Для нежадной проверки одного или более элементов используйте `+?`:

```
my $minimal_greedy_plus = qr/hot.+?meal/;

unlike( 'ilikeahotmeal', $minimal_greedy_plus );

like( 'i like a hot meal', $minimal_greedy_plus );
```

Модификатор `?` применим также и к квантификатору `?` (ноль или более совпадений), как и к квантификаторам диапазона. В каждом случае он заставляет регулярное выражение захватывать настолько мало входных данных, насколько возможно.

Жадные шаблоны `.+` и `.*` соблазнительны, но опасны. Кроссвордист_-(Любитель разгадывания кроссвордов.)_, которому нужно заполнить 7 по вертикали («Богатая почва») найдёт слишком много неверных кандидатов со следующим шаблоном:

```
my $seven_down   = qr/l$letters_only*m/;
```

Придётся отбросить `Alabama`, `Belgium` и `Bethlehem` задолго до того, как программа предложит `loam`. Эти слова не только слишком длинные, но и соответствие начинается с середины слов. Рабочее понимание жадности поможет, но ничто не заменит обширное тестирование на реальных рабочих данных.

Якоря регулярных выражений

Якоря регулярных выражений заставляют механизм регулярных выражений начинать или заканчивать сопоставление на абсолютной позиции. *Якорь начала строки* (`\A`) требует, чтобы любое соответствие начиналось с начала строки:

```
# также соответствует "lammed", "lawmaker" и "layman"
my $seven_down = qr/\Al${letters_only}{2}m/;
```

Якорь конца строки (`\Z`) требует, чтобы соответствие заканчивалось в конце строки.

```
# также соответствует "loom", но уже очевидное улушение
my $seven_down = qr/\Al${letters_only}{2}m\Z/;
```

Якорь границы слова (`\b`) находит соответствие только на границе между символом слова (`\w`) и несловарным символом (`\W`). Используйте регулярное выражение с якорями, чтобы найти `loam`, отбросив `Belgium`:

```
my $seven_down = qr/\bl${letters_only}{2}m\b/;
```

Метасимволы

Perl интерпретирует некоторые символы в регулярных выражениях как *метасимволы*, символы, представляющие нечто отличное от их буквальной интерпретации. Метасимволы пользователям регулярных выражений силу, далеко выходящую за рамки всего лишь поиска подстрок. Механизм регулярных выражений обрабатывает все метасимволы как атомы.

Метасимвол `.` означает «соответствие любому символу, исключая перевод строки». Запомните это пояснение; многие новички его забывают. Простой поиск по регулярному выражению — не принимая в счёт явное улучшение с

использованием якорей — для 7 по вертикали может быть `/l..m/`. Конечно, всегда есть более чем один способ получить правильный ответ:

```
for my $word (@words)
{
    next unless length( $word ) == 4;
    next unless $word =~ /l..m/;
    say "Possibility: $word";
}
```

Если потенциальные совпадения в `@words` — более, чем простые английские слова, вы получите ложноположительные результаты. `.` также соответствует знакам пунктуации, пробельным символам и числам. Будьте точны! Метасимвол `\w` представляет все буквенно-числовые символы и символ подчёркивания:

```
    next unless $word =~ /l\w\wm/;
```

Метасимвол `\d` соответствует цифрам (тоже в смысле Юникод):

```
# ненадёжная проверка телефонных номеров
next unless $number =~ /\d{3}-\d{3}-\d{4}/;
say "I have your number: $number";
```

Используйте метасимвол `\s` для сопоставления пробельным символам, будь это литеральный пробел, символ табуляции, возврат каретки, подача страницы или перевод строки:

```
my $two_three_letter_words = qr/\w{3}\s\w{3}/;
```

Классы символов

Если все эти метасимволы недостаточно точны, определите свой собственный *класс символов*, заключив их в квадратные скобки:

```
my $ascii_vowels = qr/[aeiou]/;
my $maybe_cat    = qr/c${ascii_vowels}t/;
```

Символ дефиса (`-`) позволяет указывать в классе непрерывный диапазон символов, как в регулярном выражении `$ascii_letters_only`:

```
my $ascii_letters_only = qr/[a-zA-Z]/;
```

Чтобы использовать дефис как член класса, переместите его в начало или в конец:

```
my $interesting_punctuation = qr/[-!?]/;
```

...или экранируйте его:

```
my $line_characters = qr/[|=\-_]/;
```

Используйте символ каре (`^`) как первый элемент класса символов, чтобы сказать «любые символы *кроме* этих»:

```
my $not_an_ascii_vowel = qr/[^aeiou]/;
```

Захват

Регулярные выражения позволяют вам группировать и захватывать части совпадения для дальнейшего использования. Извлечь из строки американский телефонный номер вида `(202) 456-1111` можно следующим образом:

```
my $area_code    = qr/\(\d{3}\)/;
my $local_number = qr/\d{3}-?\d{4}/;
my $phone_number = qr/$area_code\s?$local_number/;
```

Обратите особое внимание на экранирование круглых скобок в `$area_code`. Скобки имеют особое значение для регулярных выражений Perl 5. Они группируют атомы в единицы большего размера, а также захватывают части сопоставленных строк. Для поиска литеральных скобок, экранируйте их с помощью обратных слешей, как сделано в `$area_code`.

Именованные захваты

В Perl 5.10 были добавлены *именованные захваты*, которые позволяют вам захватывать части совпадения при применении регулярного выражения и получать к ним доступ позже, как например при нахождении телефонного номера в строке, содержащей контактную информацию:

```
if ($contact_info =~ /(?<phone>$phone_number)/)
{
    say "Found a number $+{phone}";
}
```

Регулярные выражения имеют тенденцию выглядеть как пунктуационный суп, пока вы не соберёте разные части вместе, как кусочки. Синтаксис именованного захвата имеет следующий вид:

```
(?<capture name> ... )
```

Скобки окружают захват. Конструкция `?< name >` именует этот конкретный захват, и должна следовать сразу же за левой скобкой. Оставшаяся часть захвата — регулярное выражение.

Когда будет найдено соответствие заключённому в конструкцию шаблону, Perl сохранит совпадающую с шаблоном часть строки в магическую переменную `%+`. В этом хеше ключ — это имя захвата, а значение — соответствующая часть найденной строки.

Нумерованные захваты

С давних пор Perl поддерживает *нумерованные захваты*:

```
if ($contact_info =~ /($phone_number)/)
{
    say "Found a number $1";
}
```

Эта форма захвата не предоставляет идентифицирующего имени и не сохраняет в `%+`. Вместо этого, Perl сохраняет захваченную подстроку в последовательности магических переменных. *Первый* найденный захват попадает в `$1`, второй — в `$2` и так далее. Подсчёт захватов начинается с *открывающей* скобки захвата; так что первая левая скобка начинает захват в `$1`, вторая — в `$2` и т. д.

Хотя синтаксис именованных захватов длиннее, чем нумерованных, он предоставляет дополнительную ясность. Подсчёт левых скобок — утомительный труд, и объединение регулярных выражений, содержащих нумерованные захваты, слишком сложно. Именованные захваты улучшают поддерживаемость регулярных выражений — хотя коллизии имён возможны, они относительно редки. Минимизируйте риск путём использования нумерованных захватов только в регулярных выражениях верхнего уровня.

В списочном контексте сопоставление регулярному выражению возвращает список захваченных подстрок:

```
if (my ($number) = $contact_info =~ /($phone_number)/)
{
    say "Found a number $number";
}
```

Нумерованные захваты также полезны в простых заменах, где именованные захваты могут быть слишком громоздкими:

```
my $order = 'Vegan brownies!';
```

```
$order =~ s/Vegan (\w+)/Vegetarian $1/;
# или
$order =~ s/Vegan (?<food>\w+)/Vegetarian $+{food}/;
```

Группировка и выбор

Все предыдущие примеры применяли квантификаторы к простым атомам. Вы можете применять их к любому элементу регулярного выражения:

```
my $pork  = qr/pork/;
my $beans = qr/beans/;

like( 'pork and beans', qr/\A$pork?.*?$beans/,
     'maybe pork, definitely beans' );
```

Если вы расширите регулярное выражение вручную, результат может вас удивить:

```
my $pork_and_beans = qr/\Apork?.*beans/;

like( 'pork and beans', qr/$pork_and_beans/,
    'maybe pork, definitely beans' );
like( 'por and beans', qr/$pork_and_beans/,
     'wait... no phylloquinone here!' );
```

Иногда определённость помогает аккуратности шаблона:

```
my $pork  = qr/pork/;
my $and   = qr/and/;
my $beans = qr/beans/;

like( 'pork and beans', qr/\A$pork? $and? $beans/,
    'maybe pork, maybe and, definitely beans' );
```

Некоторым регулярным выражениям нужно сопоставить одному варианту или другому. Метасимвол *выбора* (`|`) выражает это намерение:

```
my $rice  = qr/rice/;
my $beans = qr/beans/;

like( 'rice',  qr/$rice|$beans/, 'Found rice'  );
like( 'beans', qr/$rice|$beans/, 'Found beans' );
```

Метасимвол выбора указывает, что любой из предшествующих фрагментов может быть сопоставлен. Имейте ввиду, что выбор имеет приоритет даже ниже, чем атомы:

```
like(   'rice',  qr/rice|beans/, 'Found rice'   );
like(   'beans', qr/rice|beans/, 'Found beans'  );
unlike( 'ricb',  qr/rice|beans/, 'Found hybrid' );
```

Хотя легко воспринять `rice|beans` как `ric`, за которым следует либо `e`, либо `b`, и далее `eans`, выбор всегда включает *весь* фрагмент до ближайшего ограничителя регулярного выражения, будь это начало или конце шаблона, круглые скобки, другой символ выбора или квадратная скобка.

Чтобы уменьшить путаницу, используйте именованные фрагменты в переменных (`$rice|$beans`) или группируйте альтернативные варианты в *незахватывающие группы*:

```
my $starches = qr/(?:pasta|potatoes|rice)/;
```

Последовательность `(?:)` группирует серию атомов, не осуществляя захват.

Другие экранированные последовательности

Для поиска *литерального* экземпляра метасимвола, *экранируйте* его с помощью обратного слеша (`\`). Вы встречали это раньше, где `\(` обозначало

одну левую круглую скобку, а `\]` — одну правую квадратную скобку. `\.` обозначает литеральный символ точки вместо атома, захватывающего что угодно, кроме символа перевода строки.

Вам, вероятно, понадобится экранировать метасимвол выбора (`|`), так же как метасимвол конца строки (`$`) и квантификаторы (`+`, `?`, `*`).

Символы отключения метасимволов (`\Q` and `\E`) отключают интерпретацию метасимволов в своих границах. Это особенно полезно при получении текста для сопоставления из источника, который вы не контролируете при написании программы:

```
my ($text, $literal_text) = @_;

return $text =~ /\Q$literal_text\E/;
```

Аргумент `$literal_text` может содержать всё что угодно — например, строку `** ALERT **`. С фрагментом, заключённым в `\Q` и `\E`, Perl воспримет регулярное выражение как `\*\* ALERT \*\*` и попытается найти соответствие литеральным символам звёздочки, а не жадным квантификаторам.

Проверки

Якоря регулярных выражений, такие как `\A`, `\b`, `\B` и `\Z` — форма *проверок регулярных выражений*, которые требуют, чтобы строка удовлетворяла определённым условиям. Эти проверки не сопоставляются отдельным символам строки. Независимо от того, что содержит строка, регулярное выражение `qr/\A/` *всегда* будет совпадать.

Проверки нулевой ширины соответствуют *шаблону*. Что ещё более важно, они при этом не потребляют часть шаблона, которой сопоставляются. Например, чтобы найти кота самого по себе, вы можете использовать проверку границы слова:

```
my $just_a_cat = qr/cat\b/;
```

...но если хотите найти животное, не вызывающее бедствий, вы можете использовать *проверку нулевой ширины отрицательного заглядывания вперёд*:

```
my $safe_feline = qr/cat(?!astrophe)/;
```

Конструкция `(?!...)` соответствует фразе `cat`, только если сразу за ней не следует фраза `astrophe`.

Проверка нулевой ширины положительного заглядывания вперёд:

```
my $disastrous_feline = qr/cat(?=astrophe)/;
```

...соответствует фразе `cat`, только если сразу за ней следует фраза `astrophe`. Хотя того же самого можно добиться с помощью обычного регулярного выражения, рассмотрите регулярное выражение для поиска всех некатастрофических слов в словаре, начинающихся с `cat`:

```
my $disastrous_feline = qr/cat(?!astrophe)/;

while (<$words>)
{
    chomp;
    next unless /\A(?<cat>$disastrous_feline.*)\Z/;
    say "Found a non-catastrophe '$+{cat}'";
}
```

Утверждение нулевой ширины не потребляет ничего из исходной строки, оставляя для сопоставления фрагмент с якорем `.*\Z`. В противном случае захват захватил бы только часть `cat` из исходной строки.

Чтобы проверить, что ваше животное никогда не появляется в начале строки, вы можете использовать *утверждение нулевой ширины отрицательного заглядывания назад*. Эти проверки должны иметь фиксированный размер; вы не можете использовать квантификаторы:

```
my $middle_cat = qr/(?<!\A)cat/;
```

Конструкция `(?<!...)` содержит шаблон фиксированной ширины. Также вы можете выразить, что `cat` всегда появляется сразу после символа пробела, с помощью *утверждения нулевой ширины положительного заглядывания назад*:

```
my $space_cat = qr/(?<=\s)cat/;
```

Конструкция `(?<=...)` содержит шаблон фиксированной ширины. Этот подход может быть полезен при объединении глобального сопоставления регулярному выражению с модификатором `\G`, но это продвинутая возможность, которую вы, вероятно, не будете использовать часто.

Новейшая возможность регулярных выражений Perl 5 — проверка *сохранения* `\K`. Это проверка нулевой ширины положительного заглядывания назад *может* иметь переменную длину:

```
my $spacey_cat = qr/\s+\Kcat/;

like( 'my cat has been to space', $spacey_cat );
like( 'my  cat  has  been  to  doublespace',
     $spacey_cat );
```

`\K` на удивление полезна для некоторых замен, которые удаляют конец шаблона:

```
my $exclamation = 'This is a catastrophe!';
$exclamation    =~ s/cat\K\w+!/./;

like( $exclamation, qr/\bcat\./,
                      "That wasn't so bad!" );
```

Модификаторы регулярных выражений

Некоторые модификаторы изменяют поведение операторов регулярных выражений. Эти модификаторы располагаются на конце операторов сопоставления, замены или `qr//`. Например, так можно включить регистронезависим сопоставление:

```
my $pet = 'CaMeLiA';

like( $pet, qr/Camelia/,  'Nice butterfly!'  );
like( $pet, qr/Camelia/i, 'shift key br0ken' );
```

Первый `like()` провалится, потому что строка содержит другие буквы. Второй `like()` будет успешен, потому что модификатор `/i` заставляет регулярное выражение игнорировать различия в регистре. `M` и `m` во втором регулярном выражении эквивалентны, благодаря модификатору.

Также вы можете включать модификаторы в шаблон:

```
my $find_a_cat = qr/(?<feline>(?i)cat)/;
```

Синтаксис `(?i)` включает регистронезависимый поиск только внутри включающей его группы: в данном случае, в именованном захвате. Вы можете использовать несколько модификаторов в этой форме. Отключите определённые модификаторы, предварив их знаком минуса (`-`):

```
my $find_a_rational = qr/(?<number>(?-i)Rat)/;
```

Многострочный оператор, `/m`, позволяет якорям `\A` и `\Z` соответствовать любому переводу строки, входящему в строку.

Модификатор `/s` воспринимает исходную строку как единую строку, так что метасимвол `.` будет соответствовать в том числе символу перевода строки. Демьен Конвей (Damian Conway) предлагает мнемонику: `/m` модифицирует

поведение *нескольких* (*multiple*) метасимволов регулярного выражения, а `/s` модифицирует поведение *одного* (*single*) метасимвола.

Модификатор `/r` заставляет операцию замены возвращать результат замены, оставляя исходную строку как есть. Если замена успешна, результатом будет модифицированная копия оригинала. Если замена проваливается (потому что шаблон не соответствует), результатом будет немодифицированная копия оригинала:

```
my $status     = 'I am hungry for pie.';
my $newstatus  = $status =~ s/pie/cake/r;
my $statuscopy = $status
               =~ s/liver and onions/bratwurst/r;

is( $status, 'I am hungry for pie.',
    'original string should be unmodified' );

like( $newstatus,    qr/cake/,      'cake wanted' );
unlike( $statuscopy, qr/bratwurst/, 'wurst not'   );
```

Модификатор `/x` позволяет вам включать в шаблон дополнительные пробельные символы и комментарии. При использовании этого модификатора, механизм регулярных выражений игнорирует пробельные символы и комментарии. Результат зачастую получается намного более читабельный:

```
my $attr_re = qr{
    \A                      # начало строки

    (?:
      [;\n\s]*              # пробелы и двоеточия
      (?:/\*.*?\*/)?        # комментарии в стиле C
    )*

    ATTR

    \s+
    (   U?INTVAL
      | FLOATVAL
```

```
      | STRING\s+\*
    )
}x;
```

Это регулярное выражение не *простое*, но комментарии и пробельные символы улучшают его читабельность. Даже если вы составляете регулярные выражения из компилированных фрагментов, модификатор `/x` всё ещё может улучшить ваш код.

Модификатор `/g` проверяет соответствие регулярному выражению глобально по всей строке. Это имеет смысл при использовании в замене:

```
# успокоить Mitchell estate
my $contents = slurp( $file );
$contents    =~ s/Scarlett O'Hara/Mauve Midway/g;
```

При использовании с сопоставлением — не заменой — метасимвол `\G` позволяет вам обработать строку в цикле по одному куску за раз. `\G` соответствует позиции, на которой завершилось самое последнее соответствие. Чтобы обработать плохо закодированный файл, полный американских телефонных номеров в логических кусках, вы можете написать:

```
while ($contents =~ /\G(\w{3})(\w{3})(\w{4})/g)
{
    push @numbers, "($1) $2-$3";
}
```

Имейте ввиду, что якорь `\G` будет занимать последнюю точку в строке, где произошла предыдущая итерация совпадения. Если предыдущее совпадение закончилось жадным совпадением, таким как `.*`, следующее соответствие будет иметь меньше доступной строки для сопоставления. Проверки заглядывания вперёд тоже могут помочь.

Модификатор `/e` позволяет вам писать произвольный код Perl 5 на правой стороне операции замены. Если сопоставление будет успешным, механизм регулярных выражений выполнит код и использует возвращаемое им

значение как значение для замены. Приведённый ранее пример глобальной замены может быть выполнен проще с помощью такого кода:

```
# успокоить Mitchell estate
$sequel  =~ s{Scarlett( O'Hara)?}
            {
               'Mauve' . defined $1
                       ? ' Midway'
                       : ''
            }ge;
```

Каждое дополнительное вхождение модификатора `/e` будет приводить к ещё одному вычислению результата выражения, хотя только Perl-гольферы используют что-нибудь дальше `/ee`.

Умное сопоставление

Оператор умного сопоставления, `~~`, сравнивает два операнда и возвращает истинное значение, если они успешно сопоставлены. Расплывчатость определения демонстрирует умность оператора: тип сравнения зависит от типов обоих операндов. `given` выполняет неявное умное сопоставление.

Оператор умного сопоставления — инфиксный оператор:

```
say 'They match (somehow)' if $loperand ~~ $roperand;
```

Тип сравнения обычно зависит в первую очередь от типа правого операнда, а затем — от левого. Например, если правый операнд — скаляр с числовой составляющей, сравнение будет использовать числовое равенство. Если правый операнд — регулярное выражение, сравнение будет использовать grep или сопоставление шаблону. Если правый операнд — массив, сравнение будет выполнять grep или рекурсивное умное сопоставление Если правый операнд — хеш, сравнение будет проверять существование одного или более ключей. Большая и устрашающая диаграмма в `perldoc perlsyn` даёт

гораздо больше деталей обо всех сравнениях, которые может выполнять оператор умного сопоставления.

Серьёзное предложение для 5.16 предполагает существенное упрощение умного сопоставления. Чем более сложны ваши операнды, тем больше у вас шансов получить сбивающий с толку результат. Избегайте сравнения объектов и придерживайтесь простых операций между двумя скалярами или одним скаляром и одной агрегатной переменной для достижения наилучших результатов.

С учётом вышесказанного, умное сопоставление может быть полезным:

```
my ($x, $y) = (10, 20);
say 'Not equal numerically' unless $x ~~ $y;

my $z = '10 little endians';
say 'Equal numeric-ishally' if $x ~~ $z;

# сопоставление регулярному выражению
my $needle = qr/needle/;

say 'Pattern match' if 'needle' ~~ $needle;

say 'Grep through array' if @haystack ~~ $needle;

say 'Grep through hash keys' if %hayhash ~~ $needle;

say 'Grep through array' if $needle ~~ @haystack;

say 'Array elements exist as hash keys'
    if %hayhash    ~~ @haystack;

say 'Smart match elements' if @straw ~~ @haystack;

say 'Grep through hash keys' if $needle ~~ %hayhash;

say 'Array elements exist as hash keys'
    if @haystack  ~~ %hayhash;
```

```
say 'Hash keys identical' if %hayhash ~~ %haymap;
```

Умный поиск соответствия работает даже если один из операндов — *ссылка* на заданный тип данных:

```
say 'Hash keys identical' if %hayhash ~~ \%hayhash;
```

9. Объекты

Программирование — управляющая деятельность. Чем больше программа, тем большим количеством деталей вам приходится управлять. Наша единственная надежда справиться с этой сложностью — использовать абстракцию (обращаться с похожими вещами похожим способом) и инкапсуляцию (групировать вместе связанные части).

Одних функций недостаточно для крупных задач. Существует несколько техник, группирующих функции в единицы связанного поведения. Одна из популярных техник — *объектное ориентирование* (ОО), или *объектно-ориентированное программирование* (ООП), где программы работают с *объектами* — обособленными, уникальными сущностями, обладающими собственной индивидуальностью.

Moose

Стандартная объектная система Perl 5 гибка, но минималистична. Вы можете построить замечательные вещи на её основе, но она даёт лишь небольшую помощь в некоторых базовых задачах. *Moose* — это полноценная объектная система для Perl5_(См. `perldoc Moose::Manual`) для подробной информации._. Она предоставляет более простой стандартный функционал и продвинутые возможности, заимствованные из таких языков как Smalltalk, Common Lisp и Perl 6. Код Moose работает совместно со стандартной объектной системой, и на данный момент является лучшим способом написания объектно-ориентированного кода в современном Perl 5.

Классы

Объект в Moose — это конкретный экземпляр *класса*, который представляет собой шаблон, описывающий данные и поведение, характерные для объекта. Классы используют пакеты для создания пространств имён:

```
package Cat
{
```

```
    use Moose;
}
```

Этот класс `Cat` *выглядит* так, как будто ничего не делает, но это всё, что нужно Moose для создания класса. Объекты (или *экземпляры*) класса `Cat` создаются с помощью следующего синтаксиса:

```
my $brad = Cat->new();
my $jack = Cat->new();
```

Как стрелка разыменовывает ссылку, также она и вызвает метод объекта или класса.

Методы

Метод — это функция, ассоциированная с классом. Как функции принадлежат пространствам имён, также и методы принадлежат классам, с двумя отличиями. Во-первых, метод всегда оперирует *инвокантом*. Вызов `new()` на `Cat` в сущности отправляет классу `Cat` сообщение. Имя класса, `Cat`, будет инвокантом `new()`. Если вы вызываете метод на объекте, этот объект будет инвокантом:

```
my $choco = Cat->new();
$choco->sleep_on_keyboard();
```

Во-вторых, вызов метода всегда задействует стратегию *диспетчеризации*, когда объектная система выбирает соответствующий метод. Учитывая простоту `Cat`, стратегия диспетчеризации очевидна, но существенная часть мощи ОО происходит из этой идеи.

Внутри метода первым аргументом будет инвокант. Идиоматический Perl 5 использует `$self` как его имя. Предположим, что может мяукать:

```
package Cat
```

```
{
    use Moose;

    sub meow
    {
        my $self = shift;
        say 'Meow!';
    }
}
```

Теперь все экземпляры `Cat` могут разбудить вас с утра из-за того, что их ещё не покормили:

```
my $fuzzy_alarm = Cat->new();
$fuzzy_alarm->meow() for 1 .. 3;
```

Методы, пользующиеся доступом к данным инвоканта — это *методы экземпляра*, потому что для корректной работы они требуют присутствия соответствующего инвоканта. Методы (такие как `meow()`), не обращающиеся к данным экземпляра, — это *методы класса*. Вы можете вызывать методы класса на классах и методы классов и экземпляров на экземплярах, но нельзя вызывать методы экземпляра на классах.

Конструкторы, которые *создают* экземпляры, очевидно являются методами класса. Moose предоставляет вам конструктор по умолчанию.

Методы класса — это, в сущности, глобальные функции в пространстве имён. Без доступа к данным экземпляра они имеют немного преимуществ над функциями в пространстве имён. Большая часть ОО кода должным образом использует методы экземпляров, так как у них есть доступ к данным экземпляра.

Атрибуты

Каждй объект в Perl 5 уникален. Объекты могут содержать приватные данные, ассоциированные с каждым уникальным объектом — это *атрибуты*,

данные экземпляра, или *состояние* объекта. Определите атрибут, объявив его как часть класса:

```
package Cat
{
    use Moose;

    < has 'name', is = 'ro', isa => 'Str'; >>
}
```

Это читается как «Объекты `Cat` имеют атрибут `name`. Он доступен только для чтения и является строкой.»

Moose предоставляет функцию `has()`, которая объявляет атрибут. Первый аргумент, в данном случае `'name'`, это имя атрибута. Пара аргументов `is => 'ro'` объявляет, что этот атрибут доступен только для чтения (`r`ead `o`nly), так что вы не сможете его изменять после того, как установили. Наконец, пара `isa => 'Str'` объявляет, что значение этого атрибута может быть только строкой (`str`ing).

Как результат использования `has` Moose создает метод-*аксессор* с именем `name()` и позволяет вам передавать параметр `name` в конструктор класса `Cat`:

```
for my $name (qw( Tuxie Petunia Daisy ))
{
    my $cat = Cat->new( name => $name );
    say "Created a cat for ", $cat->name();
}
```

Moose пожалуется, если вы передадите что-нибудь, не являющееся строкой. Атрибуты не *обязаны* иметь тип. В таком случае подойдёт что угодно:

```
package Cat
{
    use Moose;
```

```
    has 'name', is => 'ro', isa => 'Str';
    < has 'age',  is = 'ro'; >>
}

my $invalid = Cat->new( name => 'bizarre',
                        age  => 'purple' );
```

Указание типа позволяет Moose выполнить для вас некоторую валидацию данных. Иногда подобная строгость бесценна.

Если вы пометите атрибут как доступный для чтения *и* записи (с помощью `is => rw`), Moose создаст метод-*мутатор*, который может изменять значение этого атрибута:

```
package Cat
{
    use Moose;

    has 'name', is => 'ro', isa => 'Str';
    has 'age',  is => 'ro', isa => 'Int';
    < has 'diet', is = 'rw'; >>
}

my $fat = Cat->new( name => 'Fatty',
                    age  => 8,
                    diet => 'Sea Treats' );

say $fat->name(), ' eats ', $fat->diet();

< $fat-diet( 'Low Sodium Kitty Lo Mein' ); >>
say $fat->name(), ' now eats ', $fat->diet();
```

`ro`-аксессор, используемый как мутатор, выбросит исключение `Cannot assign a value to a read-only accessor at ...`.

Использование `ro` или `rw` — вопрос дизайна, удобства и чистоты. Moose не принуждает к какой-то определённой философии в этой области. Некоторые

предлагают делать все данные экземпляра `ro`, чтобы необходимо было передавать данные экземпляра в конструктор . В примере с `Cat age()` всё ещё может быть аксессором, но конструктор может принять *год* рождения кота и вычислить возраст сам, исходя из текущего года. Этот подход усиливает код валидации и гарантирует, что все создаваемые объекты будут иметь валидные данные.

Данные экземпляра начинают демонстрировать значение объектного-ориентированного подхода. Объект содержит связанные данные и может выполнять действия с этими данными. Класс только описывает эти данные и действия.

Инкапсуляция

Moose позволяет вам объявить, *какими* атрибутами обладают экземпляры класса (у кота есть имя), а также атрибуты этих атрибутов (вы не можете изменить имя кота; вы можете его только читать). Moose сам решает, как *хранить* эти атрибуты. Вы можете изменить это, если захотите, но позволение Moose управлять вашим хранилищем поддерживает *инкапсуляцию*: скрытие внутренних деталей объекта от внешних пользователей этого объекта.

Рассмотрим изменение того, как `Cat` управляет своим возрастом. Вместо передачи значения возраста в конструктор, передадим год рождения кота и вычислим возраст при необходимости:

```
package Cat
{
    use Moose;

    has 'name',        is => 'ro', isa => 'Str';
    has 'diet',        is => 'rw';
    < has 'birth_year',  is = 'ro', isa => 'Int'; >>

    sub age
    {
        my $self = shift;
        my $year = (localtime)[5] + 1900;
```

```
        < return $year - $self-birth_year(); >>
    }
}
```

Хотя синтаксис *создания* объектов `Cat` изменился, синтаксис *использования* объектов `Cat` остался прежним. Снаружи `Cat age()` ведёт себя так же, как и раньше. *Как* это работает внутренне — это забота класса `Cat`.

Вычисление возраста имеет другое преимущество. *Значение атрибута по умолчанию* будет вести себя правильно в случае, если кто-нибудь создат новый объект `Cat`, не передав год рождения:

```
package Cat
{
    use Moose;

    has 'name', is => 'ro', isa => 'Str';
    has 'diet', is => 'rw', isa => 'Str';

    < has 'birth_year', >
        < is      = 'ro', >>
        < isa     = 'Int', >>
        < default = sub { (localtime)[5] + 1900 }; >>
}
```

Ключевое слово `default` у атрибута принимает ссылку на функцию_-(Вы можете использовать простое значение, такое как число или строка, напрямую, но для чего-либо более сложного используйте ссылку на функцию.)_, которая возвращает значение по умолчанию для этого атрибута при создании нового объекта. Если код, создающий объект, не передаст в конструктор значение для этого атрибута, объект получит значение по умолчанию:

```
my $kitten = Cat->new( name => 'Choco' );
```

...и этот котёнок будет иметь возраст `0` до следующего года.

Полиморфизм Инкапсуляция полезна, но настоящие возможности объектного-ориентированного подхода намного шире. Хорошо спроектированная ОО-программа может управлять множеством типов данных. Когда хорошо спроектированные классы инкапсулируют конкретные детали объектов в соответствующих местах, происходит нечто любопытное: код зачастую становится *менее* конкретным.

Перемещение деталей того, что программа знает об отдельных объектах `Cat` (атрибуты), и того, что программа знает о том, что объекты `Cat` могут делать (методы) в класс `Cat` означает, что код, работающий с экземплярами `Cat`, может успешно игнорировать то, *как* `Cat` делает то, что он делает.

Рассмотрим функцию, выводяющую детали объекта:

```
sub show_vital_stats
{
    my $object = shift;

    say 'My name is ', $object->name();
    say 'I am ',       $object->age();
    say 'I eat ',      $object->diet();
}
```

Очевидно (из контекста), что эта функцию работает, если вы передадите ей объект `Cat`. Фактически, она будет делать то, что нужно, для любого объекта, имеющего три соответствующих аксессора, независимо от того, *как* этот объект предоставляет эти аксессоры, и независимо от того, *что* это за объект: `Cat`, `Caterpillar` или `Catbird`. Функция имеет достаточно общий вид, чтобы любой объект, обеспечивающий этот интерфейс, был корректным параметром.

Это свойство *полиморфизма* означает, что вы можете заменить объект одного класса объектом другого класса, если они предоставляют один и тот же внешний интерфейс.

`show_vital_stats()` заботится о том, чтобы инвокант был валиден, только в смысле поддержки этих трёх методов, `name()`, `age()` и `diet()`, каждый из которых не имеет аргументов и возвращает нечто, что можно конкатенировать в строковом контексте. Вы можете иметь в своём коде

сотню разных классов, не имеющих каких-либо очевидных взаимосвязей, но они будут работать с этим методом, если соответствуют этому ожидаемому поведению.

Подумайте, как вы могли бы обработать целый зоопарк животных без этой полиморфной функции? Выгода универсальности должна быть очевидна. Также, любые специфичные детали того, как рассчитать возраст оцелота или осьминога, могут принадлежать соответствующему классу — где они имеют наибольшее значение.

Конечно, одно лишь существование методов с именами `name()` или `age()` само по себе не определяет поведение объекта. Для объекта `Dog` `age()` может быть аксессором, с помощью которого вы можете узнать, что `$rodney` — 9 лет, а `$lucky` — 4 года. Объект `Cheese` может иметь метод `age()`, который позволяет вам контролировать, как долго хранить `$cheddar` чтобы его вкус заострился. `age()` может быть аксессором в одном классе, но не в другом:

```
# сколько лет коту?
my $years = $zeppie->age();

# хранить сыр на складе шесть месяцев
$cheese->age();
```

Иногда полезно знать, *что* делает объект, и что это *означает.*

Роли

Роль — это именованная совокупность поведений и состояний_(Смотрите дизайн-документы Perl 6 на тему ролей по адресу http://feather.perl6.nl/syn/S14.html) и исследуйте трейты Smalltalk на странице http://scg.unibe.ch/research/traits для получения глубоких подробностей._. Тогда как класс организует поведения и состояние в шаблон для создания объектов, роль организует именованную коллекцию поведений и состояния. Вы можете создать экземпляр класса, но не роли. Роль — это что-то, что делает класс.

Если у нас есть `Animal`, имеющий возраст, и `Cheese`, имеющий возраст, разница между ними может быть в том, что `Animal` выполняет роль `LivingBeing`, тогда как `Cheese` выполняет роль `Storable`:

```
package LivingBeing
{
    use Moose::Role;

    requires qw( name age diet );
}
```

Всё, что имеет эту роль, должно предоставлять методы `name()`, `age()` и `diet()`. Класс `Cat` может явно обозначить, что он имеет эту роль:

```
package Cat
{
    use Moose;

    has 'name', is => 'ro', isa => 'Str';
    has 'diet', is => 'rw', isa => 'Str';

    has 'birth_year',
        is      => 'ro',
        isa     => 'Int',
        default => sub { (localtime)[5] + 1900 };

    with 'LivingBeing';

    sub age { ... }
}
```

Строчка `with` заставляет Moose *скомпоновать* роль `LivingBeing` в класс `Cat`. Компоновка гарантирует, что все атрибуты и методы роли являются частью класса. `LivingBeing` требует от любого компонуемого класса предоставлять методы с названиями `name()`, `age()` и `diet()`. `Cat` удовлетворяет этим ограничениям. Если `LivingBeing` была бы

скомпонована в класс, который не предоставляет эти методы, Moose бы выбросил исключение.

Теперь все экземпляры `Cat` будут возвращать истинное значение, если их запросить, предоставляют ли они роль `LivingBeing`. А объекты `Cheese` — не будут:

```
say 'Alive!' if $fluffy->DOES('LivingBeing');
say 'Moldy!' if $cheese->DOES('LivingBeing');
```

Эта техника проектирования отделяет *возможности* классов и объектов от *реализации* этих классов и объектов. Поведение вычисления года рождения класса `Cat` само по себе тоже может быть ролью:

```
package CalculateAge::From::BirthYear
{
    use Moose::Role;

    has 'birth_year',
        is      => 'ro',
        isa     => 'Int',
        default => sub { (localtime)[5] + 1900 };

    sub age
    {
        my $self = shift;
        my $year = (localtime)[5] + 1900;

        return $year - $self->birth_year();
    }
}
```

Выделение этой роли из `Cat` делает полезное поведение доступным для других классов. Теперь `Cat` может компоновать обе роли:

```
package Cat
```

```
{
    use Moose;

    has 'name', is => 'ro', isa => 'Str';
    has 'diet', is => 'rw';

    with 'LivingBeing',
         'CalculateAge::From::BirthYear';
}
```

Обратите внимание, как метод `age()` из `CalculateAge::From::BirthYear` удовлетворяет требование роли `LivingBeing`. Также заметьте, что любая провера того, что `Cat` выполняет роль `LivingBeing`, возвращает истинное значение. Выделение `age()` в роль изменило только детали того, *как* `Cat` вычисляет возраст. Это всё ещё `LivingBeing`. `Cat` может выбрать, имплементировать ли свой собственный возраст или получить его откуда-то ещё. Всё что имеет значение — это то, что он предоставляет `age()`, который удовлетворяет ограничениям `LivingBeing`.

Как полиморфизм означает, что вы можете работать с разными объектами, имеющими одно и то же поведение, одним и тем же способом, так и этот *алломорфизм* означает, что объект может реализовывать одно и то же поведение разными способами.

Глубокий алломорфизм может уменьшить размер ваших классов и увеличить количество кода, разделяемого между ними. Он также позволяет вам именовать специфичные и обособленные поведения — очень полезно для проверки возможностей вместо реализаций.

Для сравнения ролей с другими техниками проектирования, такими как примеси, множественное наследование и манкипатчинг (monkeypatching) см. http://www.modernperlbooks.com/mt/2009/04/the-why-of-perl-roles.html.

Роли и DOES() Если вы компонуете роль в класс, класс и его экземпляры будут возвращать истинное значение если вы вызовете на них `DOES()`:

```
say 'This Cat is alive!'
    if $kitten->DOES( 'LivingBeing' );
```

Наследование

Объектная система Perl 5 поддерживает *наследование*, которое устанавливает связи между двумя классами, так, что один уточняет другой. Дочерний класс ведёт себя так же, как его родитель — он имеет то же самое количество и типы атрибутов, и может использовать те же самые методы. Он может иметь дополнительные данные и поведение, но вы можете использовать любой экземпляр дочернего класса там, где код ожидает его родителя. В некотором смысле, подкласс предоставляет роль, предполагаемую существованием его родительского класса.

Рассмотрите класс `LightSource`, который предоставляет два публичных атрибута (`enabled` и `candle_power`) и два метода (`light` и `extinguish`):

```
package LightSource
{
    use Moose;

    has 'candle_power', is      => 'ro',
                        isa     => 'Int',
                        default => 1;

    has 'enabled', is      => 'ro',
                   isa     => 'Bool',
                   default => 0,
                   writer  => '_set_enabled';

    sub light
    {
        my $self = shift;
        $self->_set_enabled(1);
    }

    sub extinguish
    {
        my $self = shift;
        $self->_set_enabled(0);
    }
}
```

(Заметьте, что опция `writer` атрибута `enabled` создаёт приватный аксессор, используемый в классе для установки значения.)

Наследование и атрибуты Подкласс `LightSource` может определить супер-свечу, дающую в сто раз больше света:

```
package SuperCandle
{
    use Moose;

    extends 'LightSource';

    has '+candle_power', default => 100;
}
```

`extends` принимает список имён классов для использования их в качестве родителей текущего класса. Если бы это была единственная строка в этом классе, объекты `SuperCandle` вели бы себя в точности так же, как объекты `LightSource`. Они имели бы оба атрибута, `candle_power` и `enabled`, так же как и методы `light()` и `extinguish()`.

`+` в начале имени атрибута (как у `candle_power`) указывает, что текущий класс делает с этим атрибутом что-то особенное. Здесь супер-свеча переопределяет значение по умолчанию источника света, так что любой новый созданный объект `SuperCandle` будет иметь световое значение в 100 свечей.

Когда вы вызываете `light()` или `extinguish()` на объекте `SuperCandle`, Perl будет искать методы в классе `SuperCandle`, а затем в каждом из его родителей. В данном случае, эти методы находятся в классе `LightSource`.

Наследование атрибутов работает аналогично (см. `perldoc Class::MOP`).

Порядок диспетчеризации методов *Порядок диспетчеризации методов* (*или порядок разрешения методов*, *method resolution order*, или *MRO*) очевиден для классов, имеющих одного родителя. Искать в классе объекта, затем в его родителе, и так далее, до тех пор, пока метод не будет найден или

родители не закончатся. Классы, наследующие от нескольких родителей (*множественное наследование*) — `Hovercraft` расширяет и `Boat`, и `Car` — требуют более хитрой диспетчеризации. Суждение о множественном наследовании сложно. По возможности избегайте множественного наследования.

Perl 5 использует стратегию разрешения методов поиском в глубину. Он ищет класс родителя, названного *первым*, и всех родителей этого родителя рекурсивно, прежде чем искать в классах следующих родителей. Прагма `mro` предоставляет альтернативные стратегии, включая стратегию C3 MRO, которая ищет в непосредственных родителях заданного класса, прежде чем искать в их родителях.

См. `perldoc mro` для дальнейших деталей.

Наследование и методы Как и с атрибутами, подклассы могут переопределять методы. Представьте свет, который вы не можете потушить:

```
package Glowstick
{
    use Moose;

    extends 'LightSource';

    sub extinguish {}
}
```

Вызов `extinguish()` на светящейся палочке не будет делать ничего, несмотря на то, что метод из `LightSource` что-то делает. Диспетчеризация методов найдёт метод подкласса. Вы, возможно, не имели намерения так делать. Если имели, используйте функцию Moose `override` чтобы ясно выразить своё намерение.

Внутри переопределённого метода функция Moose `super()` позволит вам вызвать переопределённый метод:

```
package LightSource::Cranky
```

```
{
    use Carp 'carp';
    use Moose;

    extends 'LightSource';

    override light => sub
    {
        my $self = shift;

        carp "Can't light a lit light source!"
            if $self->enabled;

        super();
    };

    override extinguish => sub
    {
        my $self = shift;

        carp "Can't extinguish unlit light source!"
            unless $self->enabled;

        super();
    };
}
```

Этот подкласс добавляет предупреждение при попытке зажечь или потушить источник света, который уже находится в нужном состоянии. Функция `super()` отправляет к реализации текущего метода в ближайшем родителе, согласно стандартному порядку разрешения методов Perl 5.

Наследование и isa() Метод `isa()` в Perl возвращает истину, если его инвокант является названным классом или расширяет его. Этот инвокант может быть именем класса или экземпляром объекта.

```
say 'Looks like a LightSource'
```

```
    if $sconce->isa( 'LightSource' );

say 'Hominidae do not glow'
    unless $chimpy->isa( 'LightSource' );
```

Moose и Perl 5 ООП

Moose предоставляет много возможностей помимо стандартного ООП Perl 5. Хотя вы *можете* построить всё, что вы получаете с Moose, сами , или смастерить это из нескольких CPAN-дистрибутивов, Moose стоит использования. Это логически связанное целое, с хорошей документацией. Многие важные проекты успешно его используют. Он имеет зрелое и внимательное сообщество разработчиков.

Moose позаботится о конструкторах, деструкторах, аксессорах и инкапсуляции. Вам придётся объявить, чего вы хотите, но то, что вы получите, будет безопасным и простым в использовании. Объекты Moose могут расширять и работать с объектами стандартной системы Perl 5.

Moose также даёт возможность *метапрограммирования* — манипулирования вашими объектами через сам Moose. Если вы когда-нибудь интересовались, какие методы доступны в классе или объекте, или какие атрибуты объект поддерживает, эта информация доступна:

```
my $metaclass = Monkey::Pants->meta();

say 'Monkey::Pants instances have the attributes:';

say $_->name for $metaclass->get_all_attributes;

say 'Monkey::Pants instances support the methods:';

say $_->fully_qualified_name
    for $metaclass->get_all_methods;
```

Вы даже можете увидеть, какие классы расширяют заданный класс:

```
my $metaclass = Monkey->meta();

say 'Monkey is the superclass of:';

say $_ for $metaclass->subclasses;
```

Смотрите `perldoc Class::MOP::Class` для большей информации об операциях с метаклассами и `perldoc Class::MOP` для информации о метапрограммировании Moose.

Moose и его *протокол метаобъектов* (*meta-object protocol*, или MOP) даёт возможность лучшего синтаксиса для объявления и работы с классами и объектами в Perl 5. Это корректный код на Perl 5:

```
use MooseX::Declare;

role LivingBeing { requires qw( name age diet ) }

role CalculateAge::From::BirthYear
{
    has 'birth_year',
        is      => 'ro',
        isa     => 'Int',
        default => sub { (localtime)[5] + 1900 };

    method age
    {
        return (localtime)[5] + 1900
                              - $self->birth_year();
    }
}

class Cat with LivingBeing
          with CalculateAge::From::BirthYear
{
    has 'name', is => 'ro', isa => 'Str';
    has 'diet', is => 'rw';
}
```

CPAN-дистрибутив `MooseX::Declare` использует `Devel::Declare` для добавления нового Moose-специфичного синтаксиса. Ключевые слова *class*, *role* и *method* уменьшают количество текста, необходимого для написания хорошего объектно-ориентированного кода на Perl 5. Обратите особое внимание на декларативную природу этого примера, а также на отсутствие `my $self = shift;` в `age()`.

Хотя Moose и не является частью ядра Perl 5, его популярность гарантирует его доступность в дистрибутивах многих ОС. Дистрибутивы Perl 5, такие как Strawberry Perl и ActivePerl, тоже его включают. Несмотря на то, что Moose — CPAN-модуль, а не базовая библиотека, его чистота и простота делают его неотъемлемой частью современного Perl-программирования.

Благословлённые ссылки

Базовая система объектов Perl 5 сознательно минималистична. Она включает всего три правила:

- Класс — это пакет.
- Метод — это функция.
- (Благословлённая) ссылка — это объект.

Основываясь на этих трёх правилах, вы можете построить что угодно, но это всё, что вы получаете по умолчанию. Этот минимализм для больших проектов может быть непрактичен — особенно неловки и ограниченны возможности увеличения абстракции посредством метапрограммирования . Moose — более подходящий выбор для современных программ, размер которых больше чем несколько сотен строк кода, хотя большое количество унаследованного кода всё ещё использует стандартное ООП Perl 5.

Последняя часть стандартного ООП Perl 5 — благословлённая ссылка. Встроенная функция bless ассоциирует имя класса со ссылкой. Эта ссылка становится корректным инвокантом, и Perl будет выполнять на ней диспетчеризацию методов, используя ассоциированный класс.

Конструктор — это метод, создающий и благословляющий ссылку. По соглашению конструкторы имеют имя `new()`, но это не обязательно. Кроме того, конструкторы почти всегда являются *методами класса*.

`bless` принимает два аргумента, ссылку и имя класса. Она возвращает ссылку. Ссылка может быть пустой. Класс ещё не обязательно должен существовать. Вы даже можете использовать `bless` за пределами конструктора или класса (хотя все программы, кроме самых простых, должны использовать настоящие конструкторы). Канонический конструктор выглядит так:

```
sub new
{
    my $class = shift;
    bless {}, $class;
}
```

По замыслу, этот конструктор получает имя класса как инвокант метода. Но вы можете и захардкодить имя класса, потеряв при этом в гибкости. Параметрический конструктор даёт возможность повторного использования посредством наследования, делегирования или экспорта.

Используемый тип ссылки имеет значение только для того, как объект хранит свои *данные экземпляра*. Он не оказывает никакого другого влияния на результирующий объект. Ссылки на хеши наиболее распространены, но благословить можно любой тип ссылки:

```
my $array_obj  = bless [], $class;
my $scalar_obj = bless \$scalar, $class;
my $sub_obj    = bless \&some_sub, $class;
```

Классы Moose определяют атрибуты объекта декларативно, но стандартное ООП Perl 5 недостаточно формально. Класс, представляющий игроков в баскетбол, и хранящий номер футболки и позицию, может использовать такой конструктор:

```
package Player
```

```
{
    sub new
    {
        my ($class, %attrs) = @_;
        bless \%attrs, $class;
    }
}
```

...и создавать игроков так:

```
my $joel  = Player->new( number  => 10,
                         position => 'center' );

my $dante = Player->new( number   => 33,
                         position => 'forward' );
```

Методы класса могут обращаться к атрибутам объекта напрямую как к элементам хеша:

```
sub format
{
    my $self = shift;
    return '#'       . $self->{number}
         . ' plays ' . $self->{position};
}
```

...но так же это может делать и любой другой код, так что любое изменение внутреннего представления объекта может сломать другой код. Методы-аксессоры безопаснее:

```
sub number   { return shift->{number}   }
sub position { return shift->{position} }
```

...и вот вы уже начинаете вручную писать то, что Moose даёт вам бесплатно. Мало того, Moose поощряет использовать аксессоры вместо прямого доступа, скрывая код генерации аксессоров. Прощайте, соблазны.

Поиск методов и наследование

При наличии благословлённой ссылки, вызов метода осуществляется следующим образом:

```
my $number = $joel->number();
```

...ищет имя класса, ассоциированного с благословлённой ссылкой `$joel` — в данном случае `Player`. Затем Perl ищет функцию_(Вспомните, что Perl 5 не делает различия между функциями в пространстве имён и методами.)_ с именем `number()` в классе `Player`. Если такой функции не существует и если `Player` расширяет другой класс, Perl ищет в родительском классе (и так далее и так далее), до тех пор, пока не найдёт `number()`. Если Perl находит `number()`, он вызывает этот метод с `$joel` в качестве инвоканта.

Moose предоставляет `extends` для отслеживания связей наследования, Perl 5 же использует глобальную переменную пакета с именем `@ISA`. Диспетчер методов ищет в `@ISA` каждого класса имена его родительских классов. Если `InjuredPlayer` расширяет `Player`, вы можете написать так:

```
package InjuredPlayer
{
    @InjuredPlayer::ISA = 'Player';
}
```

Прагма `parent` чище_(Старый код может использовать прагму `base`), но `parent` заменила `base` в Perl 5.10._:

```
package InjuredPlayer
{
    use parent 'Player';
}
```

Moose имеет собственную метамодель, сохраняющую расширенную информацию о наследовании; это обеспечивает дополнительные возможности.

Вы можете наследовать от нескольких родительских классов:

```
package InjuredPlayer;
{
    use parent qw( Player Hospital::Patient );
}
```

...хотя есть предостережения против множественного наследования и сложности диспетчеризации методов. Рассмотрите использование в качестве замены ролей или модификаторов методов Moose.

AUTOLOAD

Если в классе инвоканта или в любом из его суперклассов нет подходящего метода, дальше Perl 5 будет искать функцию `AUTOLOAD()` в каждом классе в соответствии с выбранным порядком разрешения методов. Perl вызовет любую найденную функцию `AUTOLOAD()`, чтобы она предоставила или отклонила желаемый метод.

`AUTOLOAD()` делает множественное наследование гораздо более сложным для понимания.

Перегрузка методов и SUPER

Как и в Moose, в стандартном ООП Perl 5 вы можете перегружать методы. В отличие от Moose, стандартный Perl 5 не предоставляет механизмов для указания вашего *намерения* переопределить родительский метод. Хуже того, любая функция, которую вы предварительно определили, определили или импортировали в дочерний класс, может перегрузить метод родительского класса, если имеет то же самое имя. Даже если вы забыли использовать предоставляемую Moose систему перегрузки с `override`, она по крайней мере существует. Стандартное ООП Perl 5 не предлагает такой защиты.

Чтобы перегрузить метод в дочернем классе, объявите метод с тем же именем, что и метод в родительском классе. Внутри перегруженного

метода можно вызвать родительский метод с помощь подсказки диспетчера SUPER::

```
sub overridden
{
    my $self = shift;
    warn 'Called overridden() in child!';
    return $self->SUPER::overridden( @_ );
}
```

Префикс SUPER:: у имени метода указывает диспетчеру методов перейти к перегруженному методу с соответствующим именем. Вы можете указать собственные аргументы для перегруженного метода, но большая часть кода использует @_. Не забудьте удалить вызывающую сущность с помощью shift, если вы так делаете.

Стратегии обращения с благословлёнными ссылками

Если благословлённые ссылки выглядят минималистичными, коварными и сбивающими с толку, такие они и есть. Moose — огромное улучшение. Используйте его везде, где возможно. Если же вам придётся поддерживать код, использующий благословлённые ссылки, или вы ещё не смогли убедить свою команду полностью перейти на Moose, вы можете обойти некоторые проблемы благословлённых ссылок с помощью дисциплины.

- Повсеместно используйте методы-аксессоры, даже в методах вашего класса. Рассмотрите использование модуля наподобие Class::Accessor, чтобы избежать повторяющегося кода.
- По возможности избегайте использования AUTOLOAD(). Если вам *необходимо* его использовать, используйте предварительные объявления ваших функций , чтобы помочь Perl понять, какая функция AUTOLOAD предоставляет реализацию метода.
- Исходите из предположения, что кому-нибудь, где-нибудь рано или поздно потребуется унаследовать от ваших классов (или делегировать, или переопределить интерфейс). Упростите эту задачу, не предполагая понимания внутренних деталей вашего кода, используя двухаргументную

форму `bless`, и разбивая ваши классы на наименьшие возможные самостоятельные единицы кода.
- Не смешивайте функции и методы в одном классе.
- Используйте один файл `.pm` для каждого класса, если только класс не является маленьким, самодостаточным вспомогательным инструментом, используемым в одном месте.

Рефлексия

Рефлексия (или *интроспекция*) — это процесс получения от программы информации о ней самой во время её выполнения. Работая с кодом как с данными, вы можете управлять кодом так же, как вы управляете данными. На этом принципе основана кодогенерация .

`Class::MOP` из Moose упрощает многие задачи рефлексии для объектных систем. Если вы используете Moose, его система метапрограммирования поможет вам. Если нет, несколько других идиом стандартного Perl 5 помогут вам инспектировать и манипулировать запущенными программами.

Проверка того, что модуль загружен

Если вы знаете имя модуля, вы можете проверить, считает ли Perl, что этот модуль загружен, заглянув в хеш `%INC`. Когда Perl 5 загружает код с помощью `use` или `require`, он сохраняет запись в `%INC`, где ключ — файловый путь загружаемого модуля, а значение — полный путь к модулю на диске. Другими словами, загрузка `Modern::Perl` фактически делает следующее:

```
$INC{'Modern/Perl.pm'} =
    '.../lib/site_perl/5.12.1/Modern/Perl.pm';
```

Точный путь будет различаться в зависимости от вашей инсталляции. Чтобы проверить, что Perl успешно загрузил модуль, преобразуйте имя модуля в каноническую файловую форму и проверьте существование этого ключа в `%INC`:

```
sub module_loaded
{
    (my $modname = shift) =~ s!::!/!g;
    return exists $INC{ $modname . '.pm' };
}
```

Как и с `@INC`, любой код в любом месте может манипулировать `%INC`. Некоторые модули (такие как `Test::MockObject` или `Test::MockModule`) манипулируют `%INC` с благими намерениями. В зависимости от уровня вашей паранойи, вы можете сами проверять путь и ожидаемое содержимое пакета.

Функция `is_class_loaded()` CPAN-модуля `Class::Load` инкапсулирует эту проверку `%INC`.

Проверка существования пакета

Чтобы проверить существование пакета в вашей программе — того, что какой-то код где-то выполнил директиву `package` с заданным именем — проверьте, что пакет наследует от `UNIVERSAL`. Всё, что расширяет `UNIVERSAL`, должно так или иначе предоставлять метод `can()`. Если такого пакета не существует, Perl выбросит исключение о некорректной вызывающей сущности, так что оберните этот вызов в блок `eval`:

```
say "$pkg exists" if eval { $pkg->can( 'can' ) };
```

Альтернативный подход — копание в таблице символов Perl.

Проверка существования класса

Так как Perl 5 не делает строгих различий между пакетами и классами, лучшее, что вы можете сделать без Moose — проверить существование пакета ожидаемого имени класса. Вы *можете* проверить с помощью `can()`, что этот пакет предоставляет `new()`, но нет никаких гарантий, что найденный `new()` будет методом или конструктором.

Проверка номера версии модуля

Модули не обязаны предоставлять номера версий, но каждый пакет наследует метод `VERSION()` от универсального родительского класса `UNIVERSAL` :

```
my $mod_ver = $module->VERSION();
```

`VERSION()` возвращает номер версии заданного модуля, если он определён. В противном случае он возвращает `undef`. Если модуль не существует, метод тоже вернёт `undef`.

Проверка существования функции

Чтобы проверить, существует ли функция в пакете, вызовите `can()` как метод класса на имени пакета:

```
say "$func() exists" if $pkg->can( $func );
```

Perl выбросит исключение, если `$pkg` не является допустимым инвокантом; оберните вызов метода в блок `eval` если у вас есть какие-либо сомнения в его валидности. Имейте ввиду, что функция, реализованная в терминах `AUTOLOAD()` , может давать неправильный ответ, если в пакете функции нет корректного предварительного объявления функции или корректной перегрузки `can()`. Это ошибка в другом пакете.

Используйте эту технику для определения того, импортировал ли `import()` модуля функцию в текущее пространство имён:

```
say "$func() imported!" if __PACKAGE__->can( $func );
```

Как и в случае проверки существования пакета, вы *можете* сами покопаться в таблице символов, если у вас есть для этого достаточно терпения.

Проверка существования метода

Нет надёжного способа для рефлексии определить разницу между функцией и методом.

Копание в таблице символов

Таблица символов Perl 5 — это специальный тип хеша, где ключи — это имена глобальных символов пакета, а значения — тайпглобы (typeglob). *Тайпглоб* — это внутренняя структура данных, которая может содержать любой из или все типы — скаляр, массив, хеш, дескриптор файла и функцию.

Обратиться к символьной таблице как к хешу можно добавив двойное двоеточие к имени пакета. Например, символьная таблица пакета `MonkeyGrinder` доступна как `%MonkeyGrinder::`.

Вы *можете* проверить существование конкретных имён символов в таблице символов с помощью оператора `exists` (или манипулировать таблицей символов для *добавления* или *удаления* символов, если хотите). Однако имейте ввиду, что определённые изменения в ядре Perl 5 модифицировали детали того, что хранят тайпглобы, когда и как.

Загляните в секцию «Symbol Tables» в `perldoc perlmod` для получения подробностей, а затем отдайте предпочтение другим техникам рефлексии, описанным в этом разделе. Если вам действительно нужно манипулировать таблицами символов и тайпглобами, рассмотрите использование вместо этого CPAN-модуля `Package::Stash`.

Продвинутое ОО в Perl

Создание и использование объектов в Perl 5 с Moose просто. *Проектирование* хороших программ — нет. Вы должны выдерживать баланс между проектированием слишком много и слишком мало. Только практический опыт может помочь вам понять наиболее важные техники проектирования, но несколько принципов могут указать вам направление.

Предпочитайте компоновку наследованию

Новички в ОО-проектировании зачастую чрезмерно применяют наследование для повторного использования кода и применения полиморфизма. Результат этого — глубокая иерархия классов с ответственностями, разбросанными по неверным местам. Поддержка этого кода тяжела — кто знает, где добавить или отредактировать поведение? Что случится, если код в одном месте конфликтует с кодом, объявленным где-то ещё?

Наследование — лишь один из многих инструментов. `Car` может расширять `Vehicle::Wheeled` (*отношение is-a, является*), но лучше, если `Car` будет *содержать* несколько объектов `Wheel` в качестве атрибутов экземпляра (*отношение has-a, имеет*).

Разбиение сложных классов на меньшие, специализированные сущности (будь то классы или роли), улучшает инкапсуляцию и уменьшает возможность того, что какой-либо класс или роль слишком разрастётся. Меньшие, более простые и лучше инкапсулированные сущности проще для понимания и поддержки.

Принцип единственной ответственности

Когда вы проектируете вашу объектную систему, обдумайте ответственности каждой сущности. Например, объект `Employee` может представлять индивидуальную информацию об имени человека, контактную информацию и другие личные данные, а объект `Job` может представлять рабочие обязанности. Разделение этих сущностей по их ответственностям позволяет классу `Employee` принимать во внимание только проблему управления информацией, касающейся того, кто человек есть, а классу `Job` — представлять, что человек делает. (Например, два сотрудника `Employee` могут иметь договорённость по разделению одной работы `Job`.)

Если каждый класс имеет единственную ответственность, вы улучшаете инкапсуляцию свойственных классу данных и поведения и уменьшаете связанность между классами.

Не повторяйтесь

Сложность и дублирование затрудняют разработку и поддержку. Принцип DRY (Don't Repeat Yourself, Не повторяйтесь) — напоминание о необходимости находить и устранять дублирование внутри системы. Дублирование существует в данных, так же, как и в коде. Вместо повторения конфигурационной информации, пользовательских данных и других артефактов внутри вашей системы, создаёте единое, каноническое представление этой информации, из которого вы сможете генерировать другие артефакты.

Этот принцип помогает уменьшить вероятность того, что важные части вашей системы могут потерять синхронизированность, а также помогает найти оптимальное представление системы и её данных.

Принцип замещения Лисков

Принцип замещения Лисков предполагает, что вы должны иметь возможность поставить специализацию класса или роли на место оригинала, не нарушив API оригинала. Другими словами, объект должен быть настолько же или более общим в отношении того, что он ожидает, и как минимум так же конкретен в отношении того, что он производит.

Представьте два класса, `Dessert` и его дочерний класс `PecanPie`. Если классы следуют принципу замещения Лисков, вы можете заменить в наборе тестов каждое использование объектов `Dessert` объектами `PecanPie`, и все тесты должны пройти_(См. «IS-STRICTLY-EQUIVALENT-TO-raquo Рэга Брейтуейта (Reg Braithwaite) для дальнейших подробностей, http://weblog.raganwald.com/2008/04/is-strictly-equivalent-to.html._.

Подтипы и приведение типов

Moose позволяет вам объявить и использовать типы, а также расширять их с помощью подтипов, для формирования ещё более специализированного описания того, что представляют собой ваши данные и как они себя ведут. Эти аннотации типов помогают проверить, что данные, с которыми вы собираетесь работать в конкретных функциях и методах, соответствующи,

и даже указать механизмы преобразования данных одного типа в данные другого типа.

См. `Moose::Util::TypeConstraints` и `MooseX::Types` для дальнейшей информации.

Неизменяемость

Новички в ООП зачастую обращаются с объектами так, как будто они пачка записей, использующих методы для получения и установки внутренних значений. Эта простая техника ведёт к неприятному соблазну распространить обязанности объекта по всей системе.

В случае хорошо спроектированного объекта, вы говорите ему, что делать, но не как это делать. Как правило большого пальца, если обнаружили, что обращаетесь к данным экземпляра объекта (даже с помощью методов-аксессоров), у вас, возможно, слишком много доступа к внутренностям объекта.

Один из подходов, предотвращающих такое поведение — считать объект неизменяемым. Передайте нужные данные в их конструкторы, а затем запретите любые модификации этой информации из-за пределов класса. Не делайте видимыми методы изменения данных экземпляра. Сконструированные таким образом объекты всегда корректны с момента своего создания и не могут стать некорректными в результате внешних манипуляций. Для достижения этого требуется огромная дисциплина, но получающиеся в результате системы надёжны, легко тестируются и поддерживаются.

Некоторые варианты проектирования заходят так далеко, что запрещают изменение данных экземпляра *внутри* самого класса, хотя этого достичь гораздо труднее.

10. Стиль и эффективность

Качество имеет значение.

В программах есть ошибки. Программы требуют поддержки и расширения. Над программами работает множество программистов.

Хорошее программирование требует от нас находить баланс между выполнением работы и созданием условий для того, чтобы выполнять её успешно и в будущем. Мы можем менять друг на друга время, ресурсы и качество. То, насколько хорошо мы это делаем, определяет наш уровень как практичных профессионалов.

Понимание Perl важно. Так же как и развитие чувства хорошего вкуса. Единственный способ этого добиться — практиковаться в поддержке кода и в чтении и написании хорошего кода. На этом пути нет срезов, но есть направляющие ориентиры.

Написание поддерживаемого кода на Perl

Поддерживаемость — туманная мера простоты понимания и изменения существующей программы. Оставьте какой-нибудь код в стороне на шесть месяцев, а затем снова к вернитесь нему. Поддерживаемость определяет сложность, с которой вы встретитесь при внесении изменений.

Поддерживаемость — не вопрос синтаксиса, как и не мера того, как ваш код будет выглядеть для непрограммиста. Представим компетентного программиста, понимающего суть задачи, которую код должен решать. С какими проблемами он встретится при корректном исправлении бага или добавлении улучшений?

Способность писать поддерживаемый код приходит из заработанного тяжёлым трудом опыта и поддерживается умелым обращением с идиомами, техниками и преобладающим стилем языка. Но даже новички могут улучшить поддерживаемость своего кода, следуя нескольким принципам:

- *Избавляйтесь от дублирования.* Баги скрываются в кусках повторяющегося и одинакового кода — когда вы исправите баг в одном месте, исправите ли в другом? Когда вы внесёте изменения в одно место, внесёте ли в другое?

  Хорошо спроектированные системы содержат мало дублирования. Они используют функции, модули, объекты и роли для выделения дублирующегося кода в пригодные для повторного использования компоненты, которые точно моделируют предметную область задачи. Самый лучший дизайн позволяет вам добавлять возможности, *удаляя* код.

- *Правильно именуйте сущности.* Ваш код рассказывает историю. Каждый именованный символ — переменные, функции, модели, классы — позволяет вам прояснить или затуманить свои намерения. Простота выбора имён показывает ваше понимание проблемы и ваш дизайн. Выбирайте ваши имена тщательно.

- *Избегайте ненужной заумности.* Краткий код хорош, если он отражает намерение кода. Заумный код скрывает ваши намерения за яркими трюками. Perl даёт вам возможность писать правильный код в правильное время. Там, где возможно, используйте наиболее очевидное решение. Опыт, хороший вкус и понимание того, что на самом деле важно, будут направлять вас.

  Некоторые задачи требуют изощрённых решений. Инкапсулируйте этот код за простым интерфейсом и документируйте свою изощрённость.

- *Поощряйте простоту.* При прочих равных, более простую программа легче поддерживать, чем её более сложный аналог. Простота означает понимание того, что наиболее важно, и выполнение ровно этого.

  Всё это не может служить извинением попыток увильнуть от обработки ошибок, или модульности, или валидации, или безопасности. Простой код может использовать продвинутые возможности. Простой код может применять огромные залежи CPAN-модулей. Простой код может требовать некоторого труда для его понимания. Однако простой код решает проблемы эффективно, без выполнения ненужной работы.

  Иногда вам требуется мощный, надёжный код. Иногда вам требуется однострочник. Простота означает понимание разницы и создание только того, что вам нужно.

Написание идиоматического кода на Perl

Perl свободно заимствует у других языков. Perl позволяет вам писать код так, как вы хотите его писать. Программисты на C зачастую пишут на Perl в стиле C, также как Java-программисты пишут на Perl в стиле Java. Эффективные Perl-программисты пишут на Perl в стиле Perl, включая применение идиом языка.

- *Понимайте мудрость сообщества.* Perl-программисты зачастую ведут яростные дебаты о разных техниках. Кроме того, Perl-программисты часто делятся результатами своего труда, и не только в CPAN. Обращайте на это внимание и просвещайтесь в области компромиссов между разными идеалами и стилями.

  CPAN-разработчики, Perl-монгеры и участники списков рассылок обладают заработанным тяжёлым трудом опытом в решении задач мириадами разных способов. Говорите с ними. Читайте их код. Задавайте вопросы. Учитесь у них и давайте им учиться у вас.

- *Следуйте нормам сообщества.* Perl — сообщество инструментальщиков. Мы выполняем широкий круг задач, включая статический анализ кода (`Perl::Critic`), переформатирование (`Perl::Tidy`) и частные системы распространения (`CPAN::Mini`). Пользуйтесь инфраструктурой CPAN; следуйте модели CPAN в написании, документировании, сборке, тестировании и распространении вашего кода.

- *Читайте код.* Присоединяйтесь к спискам рассылок, таким как Perl Beginners (http://learn.perl.org/faq/beginners.html), просматривайте PerlMonks (http://perlmonks.org/) и другими способами погружайтесь в сообщество_(См. http://www.perl.org/community.html)._. Читайте код и пытайтесь отвечать на вопросы — даже если вы сами никогда их не задаёте, это отличная возможность учиться.

Написание эффективного кода на Perl

Поддерживаемость — в конечном счёте, вопрос дизайна. Хороший дизайн приходит с практикой полезных привычек:

- *Пишите тестируемый код.* Написание эффективного набора тестов упражняет те же самые навыки дизайна, что и написание эффективного кода. Код есть код. Кроме того, хорошие тесты дают вам при модификации программы уверенность в том, что она продолжает работать правильно.

- *Разбивайте на модули.* Обеспечивайте соблюдение границ инкапсуляции и абстракции. Находите правильные интерфейсы между компонентами. Правильно именуйте сущности и помещайте их туда, где им место. Модульность заставляет вас размышлять об абстракциях в ваших программах для понимания того, как всё это работает вместе. Находите куски, которые не вписываются. Улучшайте свой код до тех пор, пока они не впишутся.

- *Следуйте разумным стандартам кодирования.* Эффективные руководства определяют обработку ошибок, безопасность, инкапсуляцию, дизайн API, план проекта и другие вопросы поддерживаемости. Отличные руководства помогают разработчикам общаться друг с другом через код. Вы выполняете задачи. Говорите чётко.

- *Используйте CPAN.* Perl-программисты решают задачи. Затем мы делимся полученными решениями. Пользуйтесь этим увеличителем силы. Сначала поищите в CPAN решение или частичное решение вашей проблемы. Инвестируйте время в исследования; оно окупится.

  Если вы найдёте баг, сообщите о нём. Предложите патч, если возможно. Исправьте опечатку. Попросите реализовать возможность. Скажите «Спасибо!». Вместе мы лучше, чем по отдельности. Мы сильны и эффективны, если повторно используем код.

  Когда вы готовы, когда вы решили новую задачу, поделитесь решением. Присоединяйтесь к нам. Мы решаем задачи.

Исключения

Программировать хорошо означает предвидеть неожиданное. Файлы, которые должны существовать, не существуют. Этот огромный диск, который никогда не заполнится, оказывается полным. Всегда доступная сеть не доступна. Неломающиеся базы данных ломаются. Исключения случаются, и надёжное программное обеспечение должно их обрабатывать. Если вы можете восстановиться, отлично! Если не можете, запишите нужную информацию в лог и повторите попытку.

Perl 5 обрабатывает исключительные ситуации с помощью *исключений*: механизм потока управления динамической области видимости, разработанный для генерации и обработки ошибок.

Выбрасывание исключений

Предположим, вы хотите записать в лог файл. Если вы не смогли открыть файл, что-то пошло не так. Используйте `die` чтобы выбросить исключение:

```
sub open_log_file
{
    my $name = shift;
    open my $fh, '>>', $name
        or die "Can't open logging file '$name': $!";
    return $fh;
}
```

`die()` устанавливает значение своего операнда в глобальную переменную `$@` и немедленно выходит из текущей функции, *ничего не возвращая*. Это выброшенное исключение будет подниматься по стеку вызовов до тех пор, пока не будет где-нибудь поймано. Если исключение не будет поймано нигде, программа завершится с ошибкой.

Обработка исключений использует ту же самую динамическую область видимости , что и символы, объявленные с помощью `local`.

Отлавливание исключений

Иногда исключение, приводящее к выходу из программы, удобно. Программа, выполняемая как синхронизированный процесс, может выбросить исключение, если логи ошибок заполнены, что приведёт к отправке SMS администраторам. Однако, не все исключения должны быть фатальными. От некоторых из них хорошие программы должны уметь восстанавливаться, или по крайней мере сохранять текущее состояние и завершаться чисто.

Используйте блочную форму оператора `eval` чтобы поймать исключение:

```
# лог-файл может не открыться
my $fh = eval { open_log_file( 'monkeytown.log' ) };
```

Если файл успешно открыт, `$fh` будет содержать дескриптор файла. Если нет, `$fh` останется неопределённой и выполнение программы продолжится.

Блочный аргумента `eval` вводит новую область видимости, и лексическую, и динамическую. Если `open_log_file()` вызывает другие функции, и какая-нибудь из них в конечном счёте выбросила исключение, этот `eval` его поймает.

Обработка исключений — грубый инструмент. Он будет ловить все исключения в своей динамической области видимости. Чтобы проверить, какое исключение вы поймали (и поймали ли вообще), проверьте значение `$@`. Убедитесь, что локализовали `$@` с помощью `local`, прежде чем попытаться поймать исключение; помните, что `$@` — глобальная переменная:

```
local $@;

# лог-файл может не открыться
my $fh = eval { open_log_file( 'monkeytown.log' ) };

# поймано исключение
if (my $exception = $@) { ... }
```

Сразу же скопируйте `$@` в лексическую переменную, чтобы избежать возможности того, что последующий код перезапишет глобальную переменную `$@`. Вы никогда не знаете, что ещё могло использовать блок `eval` в каком-нибудь другом месте и сбросить `$@`.

`$@` обычно содержит строку, описывающую исключение. Проверьте её содержимое, чтобы увидеть, можете ли вы обработать исключение:

```
if (my $exception = $@)
```

```
{
    die $exception
        unless $exception =~ /^Can't open logging/;
    $fh = log_to_syslog();
}
```

Пробросьте исключение, снова вызвав `die()`. Передайте существующее исключение или новое, в зависимости от необходимости.

Применение к строковым исключениям регулярных выражений может быть ненадёжным, потому что сообщения об ошибках могут со временем меняться. Это относится и к внутренним исключениям, которые выбрасывает сам Perl. К счастью, вы можете также передавать в `die` ссылку — даже благословлённую ссылку. Это позволяет вам предоставлять в вашем исключении гораздо больше информации: номера строк, файлы и другую отладочную информацию. Получение этой информации из чего-то структурированного намного легче, чем разбор строки. Ловите эти исключения так же, как любые другие.

CPAN-дистрибутив `Exception::Class` упрощает создание и использование объектов-исключений:

```
package Zoo::Exceptions
{
    use Exception::Class
        'Zoo::AnimalEscaped',
        'Zoo::HandlerEscaped';
}

sub cage_open
{
    my $self = shift;
    Zoo::AnimalEscaped->throw
        unless $self->contains_animal;
    ...
}

sub breakroom_open
{
```

```
    my $self = shift;
    Zoo::HandlerEscaped->throw
        unless $self->contains_handler;
    ...
}
```

Предостережения об исключениях

Хотя выбрасывание исключений относительно просто, ловить их сложнее. Корректное использование `$@` требует от вас обходить некоторые неочевидные риски:

- Нелокализованное с помощью `local` использование дальше в динамической области видимости может модифицировать `$@`
- Она может содержать объкт, перегружающий своё булево значение для возврата ложного значения
- Обработчик сигнала (особенно обработчик сигнала `DIE`) может изменять `$@`
- Уничтожение объекта при выходе из области видимости может вызывать `eval` и изменять `$@`

Perl 5.14 исправил некоторые из этих проблем. Признаем, они встречаются очень редко, но зачастую сложны для диагностики и исправления. CPAN-дистрибутив `Try::Tiny` улучшает безопасность обработки исключений *и* синтаксис_(На самом деле, `Try::Tiny`) помог вдохновить на улучшения в обработке исключений в Perl 5.14._.

`Try::Tiny` прост в использовании:

```
use Try::Tiny;

my $fh = try   { open_log_file( 'monkeytown.log' ) }
         catch { log_exception( $_ ) };
```

`try` заменяет `eval`. Необязательный блок `catch` выполняется, только если `try` поймал исключение. `catch` получает пойманное исключение как переменную-топик `$_`.

Встроенные исключения

Сам Perl 5 выбрасывает несколько исключительных условий. `perldoc perldiag` перечисляет несколько «отлавливаемых фатальных ошибок». Тогда как часть из них — синтаксические ошибки, выбрасываемые в процессе компиляции, другие вы можете поймать во время выполнения. Наиболее интересны следующие:

- Using a disallowed key in a locked hash (использование недопустимого ключа в заблокированном хеше)
- Blessing a non-reference (Благословление не-ссылки)
- Calling a method on an invalid invocant (вызов метода на некорректном инвоканте)
- Failing to find a method of the given name on the invocant (не удалось обнаружить метод с заданным именем в инвоканте)
- Using a tainted value in an unsafe fashion (небезопасное использование испорченного значения)
- Modifying a read-only value (модификация значения, доступного только для чтения)
- Performing an invalid operation on a reference (выполнение недопустимой операции над ссылкой)

Конечно, вы также можете ловить исключения, производимые `autodie` , и любые лексические предупреждения, повышенные до исключений .

Прагмы

Большинство расширений Perl 5 — модули, предоставляющие новые функции или определяющие классы . Некоторые же модули вместо этого влияют на поведение самого языка, как `strict` или `warnings`. Такие

модули называют *прагмами*. По соглашению прагмы имеют имена в нижнем регистре, чтобы отличить их от других модулей.

Прагмы и область видимости

Прагмы работают посредством экспорта определённого поведения или информации в лексические области видимости вызывающего их кода. Так же как объявление лексической переменной делает символическое имя доступным в пределах области видимости, использование прагм делает их поведение действующим в пределах этой области видимости:

```
{
    # $lexical невидима; strict не действует
    {
        use strict;
        my $lexical = 'available here';
        # $lexical видима; strict действует
        ...
    }
    # $lexical снова невидима; strict не действует
}
```

Также как лексические объявления воздействуют на внутренние области видимости, прагмы сохраняют свой эффект во внутренних областях видимости:

```
# область видимости файла
use strict;

{
   # внутренняя область видимости, но strict всё ещё действует
    my $inner = 'another lexical';
    ...
}
```

Использование прагм

Подключайте прагмы с помощью `use`, как и любой другой модуль. Прагмы принимают аргументы, такие как минимальный номер используемой версии и список аргументов для изменения поведения прагмы:

```
# требует объявления переменных, запрещает голые слова
use strict qw( subs vars );
```

Иногда вам нужно *отключить* все или часть этих эффектов в глубже вложенной лексической области видимости. Встроенная директива `no` отменяет импорт , что отменяет эффекты правильно работающих прагм. Например, так можно отключить защиту `strict`, если вам нужно сделать что-нибудь символьное:

```
use Modern::Perl;
# или use strict;

{
    no strict 'refs';
    # здесь можно манипулировать символьной таблицей
}
```

Полезные прагмы

Perl 5.10.0 добавил возможность писать свои собственные лексические прагмы в виде кода на чистом Perl. `perldoc perlpragma` объясняет как это делать, а описание `$^H` в `perldoc perlvar` объясняет, как эта возможность работает.

Но и до 5.10 Perl 5 включал несколько полезных встроенных прагм.

- прагма `strict` включает проверку компилятором символических ссылок, использования голых слов и объявлений переменных.

- прагма `warnings` включает опциональные предупреждения о нерекомендуемых, непреднамеренных и неудачных поведений.
- прагма `utf8` заставляет парсер воспринимать исходный код как имеющий кодировку UTF-8.
- прагма `autodie` включает автоматическую проверку ошибок системных вызовов и встроенных функций.
- прагма `constant` позволяет вам создавать константные значения времени компиляции (см. `Const::Fast` CPAN в качестве альтернативы).
- прагма `vars` позволяет вам объявлять глобальные переменные пакета, такие как `$VERSION` или `@ISA` .
- прагма `feature` позволяет вам отдельно включать и отключать возможности Perl 5, появившиеся после 5.10. Как `use 5.14;` включает все возможности Perl 5.14 и прагму `strict`, так и `use feature ':5.14';` делает то же самое. Эта прагма более полезна для *отключения* отдельных возможностей в лексической области видимости.
- прагма `less` демонстрирует, как написать прагму.

CPAN начал собирать невстроенные прагмы:

- `autobox` включает объектоподобное поведение для встроенных типов Perl 5 (скаляров, ссылок, массивов и хешей).
- `perl5i` собирает и включает многие экспериментальные расширения языка в одно целое.
- `autovivification` отключает автовивификацию
- `indirect` предотвращает использование непрямых вызовов

Эти инструменты пока не имеют широкого использования. Два последних могут помочь вам писать более корректный код, тогда как с двумя предыдущими стоит поэкспериментировать в небольших проектах. Они показывают, чем мог бы быть Perl 5.

11. Управление реальными программами

Книга может научить вас писать маленькие программы, решающие маленькие задачи-упражнения. Этим способом вы можете научиться синтаксису. Чтобы писать реальные программы, выполняющие реальные задачи, вам нужно научиться *управлять* кодом, написанным на вашем языке. Как вам организовать код? Как убедиться, что он работает? Как сделать его надёжным перед лицом ошибок? Что делает код кратким, ясным и поддерживаемым?

Современный Perl предоставляет множество инструментов и техник для написания реальных программ.

Тестирование

Тестирование — это процесс написания и запуска маленьких кусочков кода, помогающих удостовериться, что ваше программное обеспечение ведёт себя так, как задумано. Эффективное тестирование автоматизирует процесс, который вы уже выполняли бесчисленное количество раз: написать какое-то количество кода, запустить его, и убедиться, что он работает. Эта *автоматизация* чрезвычайно важна. Вместо того, чтобы доверять людям в том, что они буду идеально выполнять повторяющиеся ручные проверки, позвольте делать это компьютеру.

Perl 5 предоставляет отличные инструменты, помогающие вам писать правильные тесты.

Test::More

Тестирование в Perl начинается с базового модуля `Test::More` и его функции `ok()`. `ok()` принимает два параметра, булево значение и строку, описывающую назначение теста:

```
ok(   1, 'the number one should be true'         );
ok(   0, '... and zero should not'               );
```

```
ok(  '', 'the empty string should be false'      );
ok( '!', '... and a non-empty string should not' );

done_testing();
```

Любое условие в вашей программе, которое вы можете протестировать, так или иначе может стать двоичным значением. Каждая тестовая *проверка* — это простой вопрос, на который можно ответить «да» или «нет»: работает ли этот маленький кусочек кода так, как я ожидаю? Сложные программы могут иметь тысячи отдельных условий, и, в общем, чем мельче разбиение, тем лучше. Изолирование конкретных поведений в отдельные проверки позволяет вам свести к минимуму баги и недопонимания, особенно когда вы будете модифицировать код в будущем.

Функция `done_testing()` сообщает `Test::More`, что программа успешно выполнила все ожидаемые тестовые проверки. Если программа столкнулась с исключением времени выполнения, или по другим причинам неожиданно завершилась прежде, чем произошёл вызов `done_testing()`, тестовый фреймворк уведомит вас, что что-то пошло не так. Без механизма, подобного `done_testing()`, как бы вы *узнали*? Правда, этот пример кода слишком прост, чтобы сломаться, но код, который слишком прост, чтобы сломаться, ломается гораздо чаще, чем кто-либо мог ожидать.

Запуск тестов

Результирующая программа теперь — полноценная программа на Perl 5, которая генерирует следующий вывод:

Этот формат соответствует стандартному выводу тестов, называемому *TAP*, *Test Anything Protocol*, *Протокол тестирования чего угодно* (http://testanything.org/). Проваленные TAP-тесты выдают диагностические сообщения для помощи отладке.

Вывод тестового файла, содержащего множество проверок (особенно множество *проваленных* проверок) может быть многословным. В большинстве случаев вам достаточно знать либо что всё прошло, либо конкретику имеющихся провалов. Базовый модуль `Test::Harness` интерпретирует

TAP, и связанная с ним программа `prove` запускает тесты и отображает только наиболее релевантную информацию:

Тут много вывода, отображающего то, что и так очевидно: второй и третий тесты проваливаются, потому что ноль и пустая строка являются ложными значениями. Эти провалы легко исправить, инвертировав смысл условий с помощью булева приведения типов :

```
ok(   ! 0, '... and zero should not'          );
ok(  ! '', 'the empty string should be false' );
```

После этих двух изменений `prove` выводит следующее:

Лучшее сравнение

Хотя сердце всех автоматизированных тестов — булево условие «истинно или ложно?», сведение всего к этому булеву условию утомительно и даёт небольшие диагностические возможности. `Test::More` предоставляет несколько других удобных функций проверок.

Функция `is()` сравнивает два значения, используя оператор `eq`. Если значения равны, тест проходит. В противном случае, тест проваливается с диагностическим сообщением:

```
is(         4, 2 + 2, 'addition should work' );
is( 'pancake',   100, 'pancakes are numeric' );
```

Как вы можете предположить, первый тест проходит, а второй — проваливается.

Тогда как `ok()` предоставляет только номер строки провалившегося теста, `is()` отображает ожидаемое и полученное значение.

`is()` налагает неявный скалярный контекст на свои значения . Это означает, например, что вы можете проверить количество элементов в массиве без необходимости явного вычисления массива в скалярном контексте:

```
my @cousins = qw( Rick Kristen Alex
                  Kaycee Eric Corey );
is( @cousins, 6, 'I should have only six cousins' );
```

...хотя некоторые для ясности предпочитают писать `scalar @cousins`.

Соответствующая функция `Test::More` `isnt()` сравнивает два значения, используя оператор `ne`, и проходит, если они не равны. Она также налагает скалярный контекст на свои операнды.

И `is()`, и `isnt()` используют *строковое сравнение* с помощью операторов Perl 5 `eq` и `ne`. Это почти всегда то, что нужно, но для сложных значений, таких как объекты с перегрузкой или двойные переменные , вы можете предпочесть явное тестирование сравнения. Функция `cmp_ok()` позволяет вам указать свой собственный оператор сравнения:

```
cmp_ok( 100, $cur_balance, '<=',
       'I should have at least $100' );

cmp_ok( $monkey, $ape, '==',
       'Simian numifications should agree' );
```

Классы и объекты предоставляют свои собственные интересные способы взаимодействия с тестами. Протестировать, что класс или объект расширяет другой класс , можно с помощью `isa_ok()`:

```
my $chimpzilla = RobotMonkey->new();
isa_ok( $chimpzilla, 'Robot' );
isa_ok( $chimpzilla, 'Monkey' );
```

`isa_ok()` предоставляет свои собственные диагностические сообщенния при провалах.

`can_ok()` проверяет, что класс или объект может выполнить запрошенный метод (или методы):

```
can_ok( $chimpzilla, 'eat_banana' );
can_ok( $chimpzilla, 'transform', 'destroy_tokyo' );
```

Функция `is_deeply()` сравнивает две ссылки, чтобы убедиться, что их содержимое идентично:

```
use Clone;

my $numbers   = [ 4, 8, 15, 16, 23, 42 ];
my $clonenums = Clone::clone( $numbers );

is_deeply( $numbers, $clonenums,
     'clone() should produce identical items' );
```

Если сравнение проваливается, `Test::More` сделает всё, что в его силах, чтобы предоставить разумную диагностику, указывающую позицию первого несоответствия между структурами. Для более настраиваемых тестов смотрите CPAN-модули `Test::Differences` и `Test::Deep`.

В `Test::More` есть ещё несколько тестовых функций, но эти наиболее полезны.

Организация тестов

CPAN-дистрибутивы должны включать директорию `t/`, содержащую один или более файлов тестов, имеющих расширение `.t`. По умолчанию, когда вы собираете дистрибутив с помощью `Module::Build` или `ExtUtils::MakeMaker`, этап тестирования запускает все файлы `t/*.t`, суммирует их вывод, и проходит успешно или проваливается в зависимости от результатов набора тестов как целого. Нет конкретных руководств, как управлять содержимым отдельных файлов `.t`, хотя наиболее популярны две стратегии:

- Каждый файл `.t` должен соответствовать файлу `.pm`.
- Каждый файл `.t` должен соответствовать некоторому функционалу.

Смешанный подход наиболее гибок; один тест может проверять, что все ваши модули компилируются, тогда как другие тесты будут проверять, что каждый модуль ведёт себя так, как ожидается. Когда дистрибутив разрастается, полезность управления тестами в привязке к функционалу становится более привлекательной; большие тестовые файлы сложнее поддерживать.

Кроме того, раздельные файлы тестов могут ускорить разработку. Если вы добавляете способность дышать огнём к вашему классу `RobotMonkey`, вы, вероятно, захотите запускать только тестовый файл `t/breathe_fire.t`. Когда же вы добъётесь удовлетворительной работы функционала, запустите весь набор тестов для проверки того, что локальные изменения не имеют непреднамеренных глобальных эффектов.

Другие модули для тестирования

`Test::More` полагается на тестовый бэкенд, известный как `Test::Builder`. Этот модуль управляет планом тестов и приводит вывод тестов в TAP. Такой дизайн позволяет множеству тестовых модулей использовать один и тот же бэкенд `Test::Builder`. Вследствие этого, на CPAN доступны сотни тестовых модулей — и все они могут работать вместе в одной и той же программе.

- `Test::Fatal` помогает протестировать, что ваш код выбрасывает (и не выбрасывает) исключения соответствующим образом. Вам также может встретиться `Test::Exception`.
- `Test::MockObject` и `Test::MockModule` позволяют вам тестировать сложные интерфейсы, *имитируя* (*mocking*) их (эмулируя, но производя другие результаты).
- `Test::WWW::Mechanize` помогает тестировать веб-приложения, тогда как `Plack::Test`, `Plack::Test::Agent` и подкласс `Test::WWW::Mec` могут делать это, не используя внешний живой веб-сервер.
- `Test::Database` предоставляет функции для тестирования правильного и неправильного использования баз данных. `DBICx::TestDatabase` помогает тестировать схемы, сгенерированные с помощью `DBIx::Class`.
- `Test::Class` предлагает альтернативный механизм организации наборов тестов. Он позволяет вам создавать классы, в которых

конкретные методы группируют тесты. Вы можете наследовать от этих тестовых классов, так же как ваши классы кода наследуют друг от друга. Это превосходный способ уменьшить дублирование в тестовых наборах. См. превосходную серию статей по `Test::Class` Кёртиса Поe (Curtis Poe)*(http://www.modernperlbooks.com/mt/2009/03/organizing-test-suites-with-testclass.html).* Более новый дистрибутив `Test::Routine` предлагает аналогичные возможности посредством использования Moose .

- `Test::Differences` тестирует строки и структуры данных на идентичность и отображает любые найденные различия в своей диагностике. `Test::LongString` добавляет похожие проверки.
- `Test::Deep` тестирует идентичность вложенных структур данных .
- `Devel::Cover` анализирует выполнение ваших наборов тестов и даёт отчёт о количестве кода, которое тесты действительно выполняют. В общем, чем больше покрытие, тем лучше — хотя стопроцентное покрытие не всегда возможно, 95% гораздо лучше, чем 80%.

См. проект Perl QA (http://qa.perl.org/) для большей информации о тестировании в Perl.

Обработка предупреждений

Хотя существует более чем один способ написать работающую программу на Perl 5, некоторые из этих способов могут быть путающими, неясными и даже некорректными в некоторых хитрых ситуациях. Система опциональных предупреждений в Perl 5 может помочь вам распознавать такие ситуации и избегать их.

Генерация предупреждений

Используйте встроенную функцию `warn` чтобы выдать предупреждение:

```
warn 'Something went wrong!';
```

`warn` выводит список значений в файловый дескриптор STDERR . Perl будет добавлять имя файла и номер строки, на которой произошёл вызов `warn`, если последний элемент списка не заканчивается переводом строки.

Встроенный модуль `Carp` предлагает другие механизмы для генерации предупреждений. Его функция `carp()` выводит предупреждение с точки зрения вызывающего кода. Если есть такая валидация параметров функции:

```
use Carp 'carp';

sub only_two_arguments
{
    my ($lop, $rop) = @_;
    carp( 'Too many arguments provided' ) if @_ > 2;
    ...
}
```

...предупреждение об арности будет включать имя файла и номер строки *вызывающего* кода, а не `only_two_arguments()`. `cluck()` из того же модуля `Carp` аналогично генерирует обратную трассировку всех вызовов функций до текущей функции.

Многословный режим `Carp` добавляет обратную трассировку ко всем предупреждениям, генерируемым `carp()` и `croak()` по всей программе:

Используйте `Carp` при написании модулей вместо `warn` или `die`.

Включение и выключение предупреждений

В старом коде вы можете встретить аргумент командной строки `-w`. Он включает предупреждения по всей программе, даже во внешних модулях, написанных и поддерживаемых другими людьми. Тут всё или ничего, хотя это и может быть полезным, если у вас есть необходимые средства для устранения и предупреждений, и потенциальных предупреждений по всей кодовой базе.

Современный подход — использование прагмы `warnings` *(...или эквивалента, такого как* `use Modern::Perl;`*..* Это включает предупреждения в *лексической* области видимости и указывает, что автор кода не подразумевает, что в нормальных условиях он будет генерировать предупреждения.

Все эти флаги, `-w`, `-W` и `-X`, воздействуют на глобальную переменную `$^W`. Код, написанный до появления прагмы `warnings` (Perl 5.6.0 весной 2000 года) может локализовать `$^W`, чтобы подавить некоторые предупреждения в пределах заданной области видимости.

Отключение категорий предупреждений

Чтобы отключить выбранные предупреждения в области видимости, используйте `no warnings;` со списком аргументов. Опускание списка аргументов отключает все предупреждения в этой области видимости.

`perldoc perllexwarn` приводит список всех категорий предупреждений, которые ваша версия Perl 5 воспринимает с прагмой `warnings`. Большая часть из них представляет действительно интересные условия, но некоторые могут быть явно бесполезны в ваших конкретных обстоятельствах. Например, предупреждение `recursion` будет происходить, если Perl обнаружил, что функция вызвала себя более чем сто раз. Если вы уверены в своей способности написать условия завершения рекурсии, вы можете отключить это предупреждение в области видимости рекурсии (хотя хвостовые вызовы, возможно, подойдут лучше; .

Если вы генерируете код или локально переопределяете символы, вам может понадобиться отключить предупреждение `redefine`.

Некоторые опытные Perl-хакеры отключают предупреждение об неинициализирован значениях (`uninitialized`) в коде обработки строк, который конкатенирует значения из множества источников. Тщательная инициализация переменных помогает избежать необходимости отключения предупреждения, но локальный стиль и лаконичность могут сделать использование этого предупреждения спорным.

Сделать предупреждения фатальными

Если ваш проект считает предупреждения такими же обременительными, как ошибки, вы можете сделать их лексически фатальными. Так можно повысить *все* предупреждения до исключений:

```
use warnings FATAL => 'all';
```

Также вы можете сделать фатальными конкретные категории предупреждений, как, например, использование нерекомендуемых конструкций:

```
use warnings FATAL => 'deprecated';
```

С надлежащей дисциплиной, это может привести к очень надёжному код — но будьте осторожны. Многие предупреждения вызываются условиями времени исполнения. Если ваш набор тестов не сможет распознать все предупреждения, с которыми вы можете столкнуться, ваша программа может завершиться при выполнении из-за непойманного исключения.

Отлавливание предупреждений

Так же как вы ловите исключения, так же вы можете ловить и предупреждения. Переменная `%SIG`*(См. `perldoc perlvar`).* содержит обработчики внеполосных сигналов, генерируемых Perl или вашей операционной системой. Чтобы поймать предупреждение, присвойте `$SIG{__WARN__}` ссылку на функцию:

```
{
    my $warning;
    local $SIG{__WARN__} = sub { $warning .= shift };

    # сделать что-нибудь рискованное
    ...
```

```
    say "Caught warning:\n$warning" if $warning;
}
```

Внутри обработчика предупреждений, первым аргументом будет сообщение предупреждения. Надо сказать, эта техника менее удобна, чем отключение предупреждений лексически — но это может оказаться полезным в тестовых модулях, таких как `Test::Warnings` из CPAN, где важен фактический текст предупреждения.

Имейте ввиду, что переменная `%SIG` глобальна. Локализуйте её в наименьшей возможной области видимости с помощью `local`, но понимайте, что это всё ещё глобальная переменная.

Регистрация своих собственных предупреждений

Прагма `warnings::register` позволяет вам создавать свои собственные лексические предупреждения, чтобы пользователи вашего кода могли включать и отключать лексические предупреждения. Используйте (`use`) прагму `warnings::register` в модуле:

```
package Scary::Monkey;

use warnings::register;
```

Это создаст новую категорию предупреждений, названную по имени пакета `Scary::Monkey`. Включите эти предупреждения с помощью `use warnings 'Scary::Monkey'` и отключите с помощью `no warnings 'Scary::Monkey'`.

Используйте `warnings::enabled()` чтобы проверить, что категория предупреждения включена в вызывающей лексической области. Используйте `warnings::warnif()` для генерации предупреждений, только если предупреждения действуют. Например, чтобы выдать предупреждение в категории `deprecated`:

```
package Scary::Monkey;
```

```
use warnings::register;

sub import
{
    warnings::warnif( 'deprecated',
        'empty imports from ' . __PACKAGE__ .
        ' are now deprecated' )
    unless @_;
}
```

См. `perldoc perllexwarn` для больших подробностей.

Файлы

Большинству программ нужно как-то взаимодействовать с реальным миром. Большинству программ нужно писать, читать или другими способами манипулировать с файлами. Происхождение Perl как инструмента для системных администраторов привело к созданию языка, отлично подходящего для обработки текста.

Ввод и вывод

Дескриптор файла представляет текущее состояние одного конкретного канала ввода или вывода. В каждой программе на Perl доступны три стандартных дескриптора файлов, `STDIN` (ввод программы), `STDOUT` (вывод программы) и `STDERR` (вывод ошибок программы). По умолчанию всё, что вы выводите с помощью `print` или `say`, отправляется на `STDOUT`, тогда как ошибки, предупреждения и всё, что вы выводите с помощью `warn()`, отправляется на `STDERR`. Такое разделение вывода позволяет вам перенаправить полезный вывод и ошибки в два разных места — например, в выходной файл и лог ошибок.

Используйте встроенную функцию `open` для получения дескриптора файла. Так файл открывается для чтения:

```
open my $fh, '<', 'filename'
    or die "Cannot read '$filename': $!\n";
```

Первый операнд — это лексическая переменная, которая будет содержать результирующий дескриптор файла. Второй оператор — *режим файла*, определяющий тип операций с дескриптором файла. Последний операнд — это имя файла. Если `open` провалится, оператор `die` выбросит исключение, с содержимым `$!`, описывающим причину, по которой открытие файла не удалось.

Кроме этого, вы можете открывать файлы для записи, добавления, чтения и записи и т. д. Здесь перечислены наиболее важные режимы файлов:

Вы даже можете создавать дескрипторы файлов, который могут читать или писать в простые скаляры Perl, используя любой существующий режим файла:

```
open my $read_fh,  '<', \$fake_input;
open my $write_fh, '>', \$captured_output;

do_something_awesome( $read_fh, $write_fh );
```

`perldoc perlopentut` предлагает гораздо больше деталей о более экзотических применениях `open`, включая её способность запускать и контролировать другие процессы, а также использование `sysopen` для более точного контроля ввода и вывода. `perldoc perlfaq5` приводит рабочий код для многих задач ввода/вывода.

Двухаргументный вызов `open` Старый код часто использует двухаргументную форму `open()`, которая объединяет режим файла с именем открываемого файла:

```
open my $fh, < " $some_file" >>
    or die "Cannot write to '$some_file': $!\n";
```

Соответственно, Perl должен извлечь режим файла из имени файла, и здесь лежат потенциальные проблемы. Каждый раз, когда Perl приходится догадываться, что вы имеете ввиду, вы получаете риск того, что он может догадаться неверно. Хуже того, если `$some_file` приходит из недоверенного пользовательского ввода, у вас возникает потенциальная проблема безопасности, так как любые неожиданные символы могут изменить поведение вашей программы.

Трёхаргументный вызов `open()` — более безопасная замена для этого кода.

Чтение из файлов Имея открытый для ввода дескриптор файла, прочитать из него можно с помощью встроенной функции `readline`, также записываемой как `<>`. Распространённая идиома читает по строке за раз в цикле `while()`:

```
open my $fh, '<', 'some_file';

while (<$fh>)
{
    chomp;
    say "Read a line '$_'";
}
```

В скалярном контексте `readline` итерирует по строкам файла, пока не достигнет его конца (`eof()`). Каждая итерация возвращает следующую строку. После достижения конца файла, каждая итерация возвращает `undef`. Эта идиома с `while` явно проверяет определённость переменной, используемой для итерации, так что только достижение конца файла завершит цикл. Другими словами, это сокращение для следующего:

```
open my $fh, '<', 'some_file';

while (defined($_ = <$fh>))
{
    chomp;
    say "Read a line '$_'";
    last if eof $fh;
}
```

Каждая строка, прочитанная с помощью `readline`, содержит символ или символы, отмечающие конец строки. В большинстве случаев это платформозависимая последовательность, состоящая из перевода строки (`\n`), возврата каретки (`\r`), или сочетания их обоих (`\r\n`). Используйте `chomp` для их удаления.

Наиболее ясный способ прочитать файл строка за строкой в Perl 5 такой:

```
open my $fh, '<', $filename;

while (my $line = <$fh>)
{
    chomp $line;
    ...
}
```

Perl по умолчанию работает с файлами в текстовом режиме. Если вы читатете *двоичные* данные, такие как медиафайл или сжатый файл — используйте `binmode` прежде чем выполнять какой-либо ввод/вывод. Это укажет Perl обрабатывать данные файла как чистые данные, никаким образом их не модифицируя_(Модификации включают перевод `\n`) в платформозависимую последовательность новой строки.. *Хотя на Unix-подобных платформах* `binmode` *может быть не всегда* обязателен_, переносимые программы играют наверняка .

Запись в файлы Имея открытый для вывода дескриптор файла, осуществляйте вывод в него с помощью `print` или `say`:

```
open my $out_fh, '>', 'output_file.txt';

print $out_fh "Here's a line of text\n";
say   $out_fh "... and here's another";
```

Обратите внимание на отсутствие запятой между дескриптором файла и следующим операндом.

И `print`, и `say` принимают список операндов. Perl 5 использует магическую глобальную переменную `$,` как разделитель значений списка. Perl также использует любое значение `$\` как последний аргумент `print` или `say`. Так что эти две строки кода дают один и тот же результат:

```
my @princes = qw( Corwin Eric Random ... );

print @princes;
print join( $,, @princes ) . $\;
```

Закрытие файлов Когда вы закончили работать с файлом, закройте его дескриптор явно с помощью `close`, или дайте ему выйти из области видимости. Perl закроет его для вас. Преимущество явного вызова `close` в том, что вы можете проверить — и восстановиться после них — специфические ошибки, такие как недостаток места на устройстве хранения или разорванное сетевое соединение.

Как обычно, `autodie` выполнит эти проверки за вас:

```
use autodie;

open my $fh, '>', $file;

...

close $fh;
```

Специальные переменные обработки файлов Для каждой прочитанной строки Perl 5 инкрементирует значение переменной `$.`, которая служит счётчиком строк.

`readline` использует текущее содержимое `$/` как последовательность завершения строки. По умолчанию значение этой переменной содержит последовательность символов завершения строки, наиболее подходящую для текстовых файлов на текущей платформе. По правде говоря, слово *строка* — неправильное название. Вы можете установить в `$/` любую

последовательность символов_(...но, к сожалению, не регулярное выражение. Perl 5 этого не поддерживает.. *Это полезно для сильно структурированных данных, в которых вы хотите считывать* запись_ за раз. Имея файл с записями, разделёнными двумя пустыми строками, установите `$/` в `\n\n` для чтения по записи за раз. Применение `chomp` к прочитанной из файла записи удалит последовательность двойного перевода строки.

Perl по умолчанию *буферизует* свой вывод, выполняя ввод/вывод только когда ожидающий вывод превышает порог по размеру. Это позволяет Perl группировать дорогие операции ввода/вывода вместо того, чтобы всегда записывать очень малые количества данных. Однако иногда вам нужно отправить данные как только вы их получаете, не ожидая буферизации — особенно если вы пишете фильтр для командной строки, соединённый с другими программами, или строко-ориентированный сетевой сервис.

Переменная `$|` контролирует буферизацию на текущем активном файловом дескрипторе вывода. При установке ненулевого значения Perl будет сбрасывать вывод после каждой записи в дескриптор файла. При установке нулевого значения Perl будет использовать свою стратегию буферизации по умолчанию.

Вместо глобальной переменной, используйте на лексическом дескрипторе файла метод `autoflush()`:

```
open my $fh, '>', 'pecan.log';
$fh->autoflush( 1 );

...
```

Начиная с Perl 5.14, вы можете использовать на дескрипторе файла любые методы, предоставляемые `IO::File`. Вам не требуется явно загружать `IO::File`. В Perl 5.12 вы должны сами загрузить `IO::File`. В Perl 5.10 и раньше вы должны вместо этого загрузить `FileHandle`.

Методы `IO::File` `input_line_number()` и `input_record_separator()` обеспечивают пофайловый доступ к тому, для чего обычно вам пришлось бы использовать глобальные `$.` и `$/`. Смотрите документацию по `IO::File`, `IO::Handle` и `IO::Seekable` для большей информации.

Директории и пути

Работа с директориями похожа на работу с файлами, за исключением того, что вы не можете *писать* в директории_(Вместо этого вы сохраняете, перемещаете, переименовываете и удаляете файлы.)_. Открыть дескриптор директории можно с помощью встроенной функции `opendir`:

```
opendir my $dirh, '/home/monkeytamer/tasks/';
```

Встроенная функция `readdir` читает из директории. Как и с `readline`, вы можете выполнить итерацию по содержимому директории по одному элементу за раз, или вы можете присвоить их списку сразу:

```
# итерация
while (my $file = readdir $dirh)
{
    ...
}

# разглаживание в список
my @files = readdir $otherdirh;
```

Perl 5.12 добавил возможность, благодаря которой `readdir` в `while` устанавливает `$_`:

```
use 5.012;

opendir my $dirh, 'tasks/circus/';

while (readdir $dirh)
{
    next if /^\./;
    say "Found a task $_!";
}
```

Любопытное регулярное выражение в этом примере пропускает так называемые *скрытые файлы* в Unix и Unix-подобных системах, где ведущая точка предотвращает появление их в листинге директории по умолчанию. Оно также пропускает два специальных файла `.` и `..`, которые представляют текущую директорию и родительскую директорию соответственно.

Имена, возвращаемые из `readdir`, *относительны* к самой директории. Другими словами, если директория `tasks/` содержит три файла, `eat`, `drink` и `be_monkey`, `readdir` вернёт `eat`, `drink` и `be_monkey`, а *не* `tasks/eat`, `tasks/drink` и `task/be_monkey`. Напротив, *абсолютный* путь — это путь, полностью определённый в файловой системе.

Закройте дескриптор директории, позволив ему выйти за пределы области видимости, или с помощью встроенной функции `closedir`.

Манипулирование путями Perl 5 предлагает взгляд в стиле Unix на вашу файловую систему и будет интерпретировать пути в стиле Unix соответствующим образом для вашей операционной системы и файловой системы. Другими словами, если вы используете Microsoft Windows, вы можете использовать путь `C:/My Documents/Robots/Bender/` так же легко, как `C:\My Documents\Robots\Caprica Six\`.

Хотя операциями Perl и управляет семантика файлов Unix, кросс-платформенные манипуляции файлами намного легче осуществлять с помощью модуля. Семейство базовых модулей `File::Spec` предоставляет абстракции, позволяющие вам манипулировать путями к файлам безопасным и переносимым образом. Эти модули уважаемы и легко понимаемы, хотя и громоздки.

Дистрибутив `Path::Class` из CPAN предоставляет более приятный интерфейс. Используйте фукнцию `dir()` для создания объекта, представляющего директорию, и функцию `file()` для создания объекта, представляющего файл:

```
use Path::Class;

my $meals = dir( 'tasks', 'cooking' );
my $file  = file( 'tasks', 'health', 'robots.txt' );
```

Вы можете получить объекты файлов из директорий и т. д.:

```
my $lunch      = $meals->file( 'veggie_calzone' );
my $robots_dir = $robot_list->dir();
```

Вы даже можете отрыть дескрипторы файлов для директорий и файлов:

```
my $dir_fh    = $dir->open();
my $robots_fh = $robot_list->open( 'r' )
                    or die "Open failed: $!";
```

И `Path::Class::Dir`, и `Path::Class::File` предлагают и другие полезные поведения — хотя имейте ввиду, что если вы используете какой-нибудь объект `Path::Class` с другим кодом на Perl 5, таким как оператор или функция, ожидающая строку, содержащую путь к файлу, вам придётся самостоятельно преобразовать объект в строку. Это постоянное, но небольшое раздражение.

```
my $contents = read_from_filename( "$lunch" );
```

Манипуляции с файлами

Помимо чтения и записи файлов, вы также можете манипулировать ими, как делали бы это напрямую из командной строки или файлового менеджера. Операторы проверки файлов, все вместе называемые операторы `-X`, из-за того, что все они представляют собой дефис и одну букву, проверяют атрибуты файла и директории. Например, так можно проверить, что файл существует:

```
say 'Present!' if -e $filename;
```

Оператор `-e` имеет один операнд, имя файла или дескриптор файла или директории. Если файл существует, выражение вернёт истинное значение.

`perldoc -f -X` содержит список всех остальных проверок файлов; наиболее популярны следующие:

- `-f`, возвращающий истинное значение, если его операнд — простой файл;
- `-d`, возвращающий истинное значение, если его операнд — директория;
- `-r`, возвращающий истинное значение, если настройка доступа к файлу, указанному как его операнд, позволяет чтение текущим пользователем;
- `-s`, возвращающий истинное значение, если его операнд — непустой файл.

Начиная с Perl 5.10.1, вы можете искать документацию по любому из этих операторов, например, как `perldoc -f -r`.

Встроенная функция `rename` может переименовать файл или переместить его между директориями. Они принимает два операнда, старое и новое имя файла:

```
rename 'death_star.txt', 'carbon_sink.txt';

# или если вы стильный:
rename 'death_star.txt' => 'carbon_sink.txt';
```

Встроенной функции для копирования файла нет, но базовый модуль `File::Copy` предоставляет как фукнцию `copy()`, так и `move()`. Используйте встроенную функцию `unlink` для удаления одного или нескольких файлов. (Встроенная функция `delete` удаляет элемент из хеша, а не файл из файловой системы.) Все эти функции возвращают истинные значения в случае успеха и устанавливают `$!` при ошибке.

Perl отслеживает свою текущую рабочую директорию. По умолчанию, это активная директория, из которой запущена программа. Функция `cwd()` базового модуля `Cwd` возвращает имя текущей рабочей директории. Встроенная функция `chdir` пытается сменить текущую рабочую директорию. Работа из корректной директории важна для работы с файлами с относительными путями.

Модули

Многие считают CPAN самой мощной возможностью Perl 5. CPAN, по сути, представляет собой систему для поиска и установки модулей. *Модуль* — это пакет, содержащийся в своём собственном файле, который можно загружать с помощью `use` или `require`. Модуль должен быть валидным кодом на Perl 5. Он должен завершаться выражением, возвращающим истинное значение, чтобы парсер Perl 5 мог понять, что он успешно загрузил и скомпилировал модуль. Других требований нет, только твёрдые соглашения.

Когда вы загружаете модуль, Perl разбивает имя пакета по двойным двоеточиям (`::`) и превращает компоненты имени пакета в имя файла. На практике, `use StrangeMonkey;` заставляет Perl искать файл с именем `StrangeMonkey.pm` во всех директориях, входящих в `@INC`, по порядку, пока он его не найдёт или не израсходует список.

Аналогично, `use StrangeMonkey::Persistence;` заставляет Perl искать файл с именем `Persistence.pm` в каждой директории с именем `StrangeMonkey/`, находящейся в любой из директорий, входящих в `@INC`, и т. д. `use StrangeMonkey::UI::Mobile;` заставляет Perl искать относительный файловый путь `StrangeMonkey/UI/Mobile.pm` в каждой директории в `@INC`.

Результирующий файл может содержать или не содержать объявление пакета, совпадающее с именем файла — такого технического *требования* нет — но вопросы поддержки рекомендуют это соглашение.

Использование и импортирование

Когда вы загружаете модуль с помощью `use`, Perl загружает его с диска, а затем вызывает его метод `import()`, передавая все указанные вами аргументы. По соглашению метод `import()` модуля принимает список имён и экспортирует функции и другие символы в вызывающее пространство имён. Это всего лишь соглашение; модуль может отказаться предоставлять `import()`, или его `import()` может выполнять другие действия. Прагмы , такие как `strict`, используют аргументы для изменения поведения вызывающей лексической области видимости вместо экспорта символов:

```
use strict;
# E<hellip>вызывает strict->import()

use CGI ':standard';
# E<hellip>вызывает CGI->import( ':standard' )

use feature qw( say switch );
# E<hellip>вызывает feature->import( qw( say switch ) )
```

Встроенная директива `no` вызывает метод `unimport()` модуля, если он существует, передавая любые аргументы. Это наиболее распространено с прагмами, использование которых модифицирует поведение через `import()`:

```
use strict;
# запрещены символьные ссылки или голые слова
# требуется объявление переменных

{
    no strict 'refs';
    # символьные ссылки разрешены
    # ограничения 'subs' и 'vars' всё ещё активны
}
```

И `use`, и `no` действуют во время компиляции, так что:

```
use Module::Name qw( list of arguments );
```

...это то же самое, что:

```
BEGIN
{
    require 'Module/Name.pm';
    Module::Name->import( qw( list of arguments ) );
}
```

Аналогично:

```
no Module::Name qw( list of arguments );
```

...это то же самое, что:

```
BEGIN
{
    require 'Module/Name.pm';
    Module::Name->unimport(qw( list of arguments ));
}
```

...включая `require` модуля.

Вы *можете* вызвать `import()` и `unimport()` напрямую, хотя за пределами блока `BEGIN` делать это мало смысла; после завершения компиляции действие `import()` или `unimport()` может не иметь особого эффекта.

В Perl 5 `use` и `require` регистрозависимы, хотя, в то время как Perl понимает разницу между `strict` и `Strict`, ваше сочетание операционной системы и файловой системы может не знать. Если бы вы написали `use Strict;`, Perl не нашёл бы `strict.pm` в регистрозависимых файловых системах. В регистронезависимых файловых системах Perl охотно загрузит `Strict.pm`, но когда он попытается вызвать `Strict->import()`, ничего не случится. (`strict.pm` объявляет пакет с именем `strict`.)

Переносимые программы придерживаются строгих правил относительно этого случая, даже если и не обязаны.

Экспортирование

Модуль может сделать определённые глобальные символы доступными другим пакетам посредством процесса, известного как *экспорт* — процесс, инициируемый вызовом `import()`, явным или неявным.

Базовый модуль `Exporter` предоставляет стандартный механизм экспорта символов из модуля. `Exporter` полагается на присутствие глобальных переменных пакета — в частности, `@EXPORT_OK` и `@EXPORT` — которые содержат список символов, экспортируемых по запросу.

Рассмотрим модуль `StrangeMonkey::Utilities`, предоставляющий несколько автономных функций, применимых по всей системе:

```
package StrangeMonkey::Utilities;

use Exporter 'import';

our @EXPORT_OK = qw( round translate screech );

...
```

Любой другой код теперь может использовать этот модуль и, опционально, импортировать любую или все три экспортируемые функции. Также вы можете экспортировать переменные:

```
push @EXPORT_OK, qw( $spider $saki $squirrel );
```

Экспортируйте символы по умолчанию, перечислив их в `@EXPORT` вместо `@EXPORT_OK`:

```
our @EXPORT = qw( monkey_dance monkey_sleep );
```

...так что любая конструкция `use StrangeMonkey::Utilities;` будет импортировать обе функции. Имейте ввиду, что указание символов для импорта *не* импортирует символы по умолчанию, вы получите только то, что запросили. Чтобы загрузить модуль, не импортируя никаких символов, явно укажите пустой список:

```
# сделать модуль доступным, но ничего не импортировать
use StrangeMonkey::Utilities ();
```

Независимо от любых списков импорта, вы всегда можете вызвать функции из другого пакета с помощью их полностью определённых имён:

```
StrangeMonkey::Utilities::screech();
```

Организация кода с помощью модулей

Perl 5 не заставляет вас использовать ни модули, ни пакеты, ни пространства имён. Вы можете разместить весь свой код в единственном файле `.pl`, или в нескольких файлах `.pl`, подключаемых с помощью `require` по необходимости. Вам предоставлена гибкость в управлении вашим кодом наиболее соответствующим способом, в зависимости от вашего стиля разработки, формальностей, рисков и вознаграждений проекта, вашего опыта и вашего комфорта в развёртывании Perl 5.

Несмотря на это, проект с более чем несколькими сотнями строк кода получает множество преимуществ от модульной организации:

- Модули помогают обеспечить логическое разделение между отдельными сущностями в системе.
- Модули предоставляют границы API, процедурного или OO.
- Модули предлагают естественную организацию исходного кода.
- Экосистема Perl 5 имеет множество инструментов, предназначенных для создания, поддержки, организации и развёртывания модулей и дистрибутивов.
- Модули предоставляют механизм повторного использования кода.

Даже если вы не используете объектно-ориентированный подход, моделирование каждой отдельной сущности или ответственности в вашей системе в виде её собственного модуля сохраняет связанный код вместе и независимый код раздельно.

Дистрибутивы

Самый простой способ управлять конфигурацией, сборкой, упаковкой, тестированием и установкой программного обеспечения — следовать соглашениям CPAN-дистрибутивов. *Дистрибутив* — это набор метаданных и одного или нескольких модулей , формирующих одну пригодную для распространения, тестирования и установки единицу.

Все эти руководства — как упаковать дистрибутив, как разрешать его зависимости, куда устанавливать программное обеспечение, как проверять, что оно работает, как отображать документацию, как управлять репозиторием — родились из примерного консенсуса тысяч контрибуторов, работающих над десятками тысяч проектов. Дистрибутив, построенный по стандартам CPAN, может быть протестирован на нескольких версиях Perl 5 на нескольких разных аппаратных платформах в течение нескольких часов после загрузки, при этом об ошибках будет автоматически сообщено авторам — и всё это без вмешательства человека.

Вы можете никогда не выпускать свой код как публичные CPAN-дистрибутивы, но вы можете использовать инструменты и соглашения CPAN даже для управления частным кодом. Сообщество Perl построило восхитительную инфраструктуру; почему бы ей не воспользоваться?

Характеристики дистрибутива

Помимо одного или нескольких модулей, дистрибутив включает несколько других файлов и директорий:

- `Build.PL` или `Makefile.PL`, программа-драйвер, используемая для конфигурации, сборки, тестирования, компоновки и установки дистрибутива.
- `MANIFEST`, список всех файлов, входящих в дистрибутив. Это помогает инструментам удостовериться, что комплект полон.
- `META.yml` и/или `META.json`, файл, содержащий метаданные о дистрибутиве и его зависимостях.
- `README`, описание дистрибутива, его предназначения, его авторских прав и информации о лицензии.

- `lib/`, директория, в которой содержатся Perl-модули.
- `t/`, директория, в которой содержатся файлы тестов.
- `Changes`, лог каждого изменения дистрибутива.

Правильно построенный дистрибутив должен содержать уникальное имя и единый номер версии (который часто берётся из его основного модуля). Любой дистрибутив, который вы скачаете из публичной CPAN, должен соответствовать этим стандартам. Публичный сервис CPANTS (http://cpants.perl.org/) проверяет каждый загруженный модуль по руководствам и соглашениям сборки и рекомендует улучшения. Следование руководствам CPANTS не означает, что код работает, но означает, что инструменты сборки CPAN должны понять дистрибутив.

Инструменты CPAN для управления дистрибутивами

Ядро Perl 5 включает несколько инструментов для установки, разработки и управления вашими собственными дистрибутивами:

- `CPAN.pm` — официальный CPAN-клиент; `CPANPLUS` — альтернатива. Они в основном аналогичны. Хотя по умолчанию эти клиенты устанавливают дистрибутивы из публичной CPAN, вы можете вместо или дополнительно к этому направить их на свой собственный репозиторий.
- `Module::Build` — это написанный на чистом Perl набор инструментов для конфигурирования, сборки, установки и тестирования дистрибутивов. Он работает с файлами `Build.PL`.
- `ExtUtils::MakeMaker` — это унаследованный инструмент, на замену которому предназначен `Module::Build`. Он всё ещё широко используется, хотя находится в режиме поддержки и получает только критические исправления ошибок. Он работает с файлами `Makefile.PL`.
- `Test::More` — базовый и самый широко используемый модуль тестирования, предназначенный для написания автоматизированных тестов для программного обеспечения на Perl.

- `Test::Harness` и `prove` запускают тесты и интерпретируют и генерируют отчёты по их результатам.

В дополнение к этому, несколько дополнительных CPAN-модулей сделают легче вашу жизнь как разработчика:

- `App::cpanminus` — это не требующий конфигурации CPAN-клиент. Он обрабатывает наиболее распространённые случаи, использует мало памяти и быстро работает.
- `App::perlbrew` помогает вам управлять несколькими инсталляциями Perl 5. Установите новые версии Perl 5 для тестирования или производственной среды, или чтобы изолировать приложения и их зависимости.
- `CPAN::Mini` и комманда `cpanmini` позволяют вам создать своё собственное (частное) зеркало публичной CPAN. Вы можете ввести в этот репозиторий свои собственные дистрибутивы и управлять тем, какие версии публичных модулей доступны в вашей организации.
- `Dist::Zilla` автоматизирует наиболее распространённые задачи распространения дистрибутивов. Хотя он использует либо `Module::Build`, либо `ExtUtils::MakeMaker`, он может заменить *ваше* использование их напрямую. Смотрите интерактивное руководство по адресу http://dzil.org/.
- `Test::Reporter` позволяет вам выводить отчёты по результатам запуска набора автоматизированных тестов дистрибутивов, которые вы устанавливаете, давая их авторам больше данных о сбоях.

Проектирование дистрибутивов

Описание процесса проектирования дистрибутива может занять целую книгу (см. *Writing Perl Modules for CPAN* Сэма Трегара (Sam Tregar)), но несколько принципов дизайна помогут вам. Начните с такой утилиты как `Module::Starter` или `Dist::Zilla`. Начальная стоимость изучения конфигурации и правил может выглядеть как чрезмерная инвестиция, но преимущества того, что всё будет правильно настроено (и в случае

`Dist::Zilla` *никогда* не устареет) освобождает вас от намного более утомительной бухгалтерии.

Затем рассмотрите несколько правил:

- *Каждый дистрибутив должен иметь единственное, чётко определённое предназначение.* Этим предназначением может быть даже сборка нескольких связанных дистрибутивов в один устанавливаемый комплект. Декомпозиция вашего программного обеспечения на отдельные дистрибутивы позволяет вам соответственно управлять их зависимостями и соблюдать их инкапсуляцию.
- *Каждый дистрибутив должен иметь единственный номер версии.* Номера версий всегда должны увеличиваться. Политика семантических версий (http://semver.org/) разумна и совместима с подходом Perl 5.
- *Каждый дистрибутив требует хорошо определённого API.* Всесторонний автоматизированный набор тестов может проверить, что вы поддерживаете этот API между версиями. Если вы используете локальное зеркало CPAN для установки своих собственных дистрибутивов, вы можете использовать инфраструктуру CPAN для тестирования дистрибутивов и их зависимостей. Вы получите лёгкий доступ к интеграционному тестированию пригодных к повторному использованию компонентов.
- *Автоматизируйте тесты вашего дистрибутива и сделайте их повторяемыми и ценными.* Инфраструктура CPAN поддерживает отчёты автоматизированного тестирования. Используйте это!
- *Предоставьте эффективный и простой интерфейс.* Избегайте использования глобальных символов и экспорта по умолчанию; позвольте людям использовать только то, что им нужно. Не загрязняйте их пространства имён.

Пакет UNIVERSAL

Встроенный пакет `UNIVERSAL` в Perl 5 — предок всех остальных пакетов — в объектно-ориентированном смысле . `UNIVERSAL` предоставляет несколько методов, которые его потомки могут унаследовать или переопределить.

Метод isa()

Метод `isa()` принимает строку, содержащую имя класса или имя встроенного типа. Вызывайте его как метод класса или метод экземпляра на объекте. Он возвращает истинное значение, если его инвокант является указанным классом или происходит от него, или если инвокант — благословлённая ссылка на заданный тип.

Пусть есть объект `$pepper` (ссылка на хеш, благословлённая в класс `Monkey`, наследующий от класса `Mammal`), тогда:

```
say $pepper->isa( 'Monkey'  );  # выводит 1
say $pepper->isa( 'Mammal'  );  # выводит 1
say $pepper->isa( 'HASH'    );  # выводит 1
say Monkey->isa(  'Mammal'  );  # выводит 1

say $pepper->isa( 'Dolphin' );  # выводит 0
say $pepper->isa( 'ARRAY'   );  # выводит 0
say Monkey->isa(  'HASH'    );  # выводит 0
```

Встроенные типы Perl 5 — `SCALAR`, `ARRAY`, `HASH`, `Regexp`, `IO` и `CODE`.

Любой класс может переопределить `isa()`. Это может быть полезно при работе с объектами-моками (см. `Test::MockObject` и `Test::MockModule` в CPAN) или с кодом, не использующим ролей . Имейте ввиду, что любой класс, *переопределяющий* `isa()`, обычно имеет для этого хорошую причину.

Метод can()

Метод `can()` принимает строку, содержащую имя метода. Он возвращает ссылку на функцию, имплементирующую этот метод, если она существует. В противном случае он возвращает ложное значение. Вы можете вызвать его на классе, объекте или имени пакета. В последнем случае он возвращает ссылку на функцию, а не метод_(…не то чтобы вы могли определить разницу, имея только ссылку._.

Имея класс, названный SpiderMonkey, с методом, названным screech, получить ссылку на метод можно так:

```
if (my $meth = SpiderMonkey->can( 'screech' )) {...}

if (my $meth = $sm->can( 'screech' )
{
    $sm->$meth();
}
```

Используйте can() чтобы проверить, что класс имплементирует конкретную функцию или метод:

```
use Class::Load;

die "Couldn't load $module!"
    unless load_class( $module );

if (my $register = $module->can( 'register' ))
{
    $register->();
}
```

Метод VERSION()

Метод VERSION возвращает значение переменной $VERSION соответствующего пакета или класса. Если вы укажете номер версии как необязательный параметр, метод выбросит исключение, если опрашиваемая $VERSION не равна или больше чем параметр.

Пусть есть модуль HowlerMonkey версии 1.23, тогда:

```
say HowlerMonkey->VERSION();    # выводит 1.23
say $hm->VERSION();             # выводит 1.23
say $hm->VERSION( 0.0  );       # выводит 1.23
```

```
say $hm->VERSION( 1.23 );       # выводит 1.23
say $hm->VERSION( 2.0  );       # исключение!
```

Нет особых причин переопределять `VERSION()`.

Метод DOES()

Метод `DOES()` появился в Perl 5.10.0. Он существует для поддержки использования в программах ролей . Передайте ему инвокант и имя роли, и метод вернёт истинное значение, если соответствующий класс как-либо выполняет эту роль — будь то через наследование, делегирование, композицию, применение роли или любой другой механизм.

По умолчанию реализация `DOES` откатывается к `isa()`, потому что наследование — единственный механизм, с помощью которого класс может выполнять роль. Пусть есть `Cappuchin`:

```
say Cappuchin->DOES( 'Monkey'       );  # выводит 1
say $cappy->DOES(    'Monkey'       );  # выводит 1
say Cappuchin->DOES( 'Invertebrate' );  # выводит 0
```

Переопределите `DOES()`, если вы вручную предоставляете роль или предоставляете другое алломорфичное поведение.

Расширение UNIVERSAL

Возникает соблазн сохранять в `UNIVERSAL` другие методы, чтобы сделать их доступными для всех других классов и объектов в Perl 5. Избегайте этого соблазна; это глобальное поведение можеть иметь неочевидные побочные эффекты, потому что оно не ограничено.

Учитывая сказанное, редкое злоупотребление `UNIVERSAL` для целей *отладки* и для исправления неправильного поведения по умолчанию можеть быть извинено. Например, дистрибутив `UNIVERSAL::ref` Джошуа

бен Джоре (Joshua ben Jore) делает практически бесполезный оператор `ref()` полезным. Дистрибутивы `UNIVERSAL::can` и `UNIVERSAL::isa` могут помочь вам отладить анти-полиморфные баги . `Perl::Critic` может обнаружить эти и другие проблемы.

За пределами очень осторожно контролируемого кода и очень специфичных, очень прагматичных ситуаций, нет причин напрямую помещать код в `UNIVERSAL`. Почти всегд есть намного лучшие альтернативы дизайна.

Кодогенерация

Новички в программировании пишут больше кода, чем нужно, частично из-за недостаточно близкого знакомства с языками, библиотеками и идиомами, но также и из-за отсутствия опыта. Они начинают с написания длинного процедурного кода, затем открывают для себя функции, затем параметры, затем объекты, и — возможно — функции высших порядков и замыкания.

По мере того, как вы становитесь лучше как программист, вы будете писать меньше кода для решения тех же самых задач. Вы будете использовать лучшие абстракции. Вы будете писать более универсальный код. Вы сможете повторно использовать код — и когда вы сможете добавлять возможности, удаляя код, вы достигнене чего-то замечательного.

Написание программ, пишуших для вас программы — *метапрограммирование* или *кодогенерация* — предлагает большие возможности для абстракции. Хотя вы можете устроить огромный бардак, но вы можете и создать изумительные вещи. Например, техники метапрограммирования делают возможным Moose :

Техника использования `AUTOLOAD` для отсутствующих функций и методов демонстрирует эту технику в ограниченной форме; система диспетчеризации функций и методов в Perl 5 позволяет вам настроить, что происходит, когда нормальный поиск проваливается.

eval

Самая простая техника кодогенерации — это формирование строки, содержащей фрагмент валидного кода на Perl, и компиляция её с помощью строкового оператора `eval`. В отличие от отлавливающего исключения блочного оператора `eval`, строковый `eval` компилирует содержимое строки в текущей области видимости, включая текущий пакет и лексические привязки.

Наиболее общее использование этой техники — предоставление возможности отката, если вы не можете (или не хотите) загружать опциональную зависимость:

```
eval { require Monkey::Tracer }
    or eval 'sub Monkey::Tracer::log {}';
```

Если `Monkey::Tracer` недоступен, его функция `log()` будет существовать, но ничего не будет делать. Однако этот простой пример обманчив. Правильное использование `eval` требует определённой работы; вы должны обработать проблемы заключения в кавычки для включения переменных в передаваемый в `eval` код. Добавьте сложности чтобы интерполировать одни переменные, но не другие:

```
sub generate_accessors
{
    my ($methname, $attrname) = @_;

    eval <<"END_ACCESSOR";
    sub get_$methname
    {
        my \$self = shift;
        return \$self->{$attrname};
    }

    sub set_$methname
    {
        my (\$self, \$value) = \@_;
```

```
        \$self->{$attrname}  = \$value;
    }
END_ACCESSOR
}
```

Горе тем, кто забыл обратный слеш! Удачи в объяснении вашей подсветке синтаксиса, что происходит! Хуже того, каждый вызов строкового `eval` создаёт новую структуру данных, представляющую код целиком, и компиляция кода тоже не бесплатна. Но даже с этими ограничениями эта техника проста.

Параметризованные замыкания

Хотя построение аксессоров и мутаторов с помощью `eval` просто, замыкания позволяют вам добавлять параметры в генерируемый код во время компиляции без необходимости дополнительного вычисления:

```
sub generate_accessors
{
    my $attrname = shift;

    my $getter = sub
    {
        my $self = shift;
        return $self->{$attrname};
    };

    my $setter = sub
    {
        my ($self, $value) = @_;
        $self->{$attrname} = $value;
    };

    return $getter, $setter;
}
```

Этот код избегает неприятных проблем с заключением в кавычки и компилирует каждое замыкание только один раз. Он даже использует

меньше памяти из-за разделения скомпилированного кода между всеми экземплярами замыканий. Всё, что различается — это привязка к лексической переменной `$attrname`. В долго выполняющихся процессах, или при большом количестве аксессоров, эта техника может быть очень полезна.

Установка в таблицу символов относительно проста, хоть и некрасива:

```
{
    my ($get, $set) = generate_accessors( 'pie' );

    no strict 'refs';
    *{ 'get_pie' } = $get;
    *{ 'set_pie' } = $set;
}
```

Странный синтаксис звёздочки_(Воспринимайте её как *сигил тайпглоба)*, где *тайпглоб* — Perl-жаргон для «таблицы символов». *разыменовывает хеш, ссылающийся на символ в текущей* таблице символов_, являющейся частью текущего пространства имён, содержащего глобально-доступные символы, такие как глобальные данные пакета, функции и методы. Присваивание ссылки записи таблицы символов устанавливает или заменяет соответствующую запись. Чтобы повысить анонимную функцию до метода, сохраните ссылку на эту функцию в таблице символов.

Присваивание таблице символов символа, содержащего строку, не литеральное имя переменной, это символическая ссылка. Для этой операции вам нужно отключить строгую проверку ссылок `strict`. Многие программы имеют неочевидный баг в аналогичном коде, так как они присваивают и генерируют в одной строчке:

```
{
    no strict 'refs';

    *{ $methname } = sub {
      # неочевидный баг: strict refs отключено здесь тоже
    };
}
```

Этот пример отключает ограничения для внешнего блока, так же как и для тела самой функции. Только присваивание нарушает строгую проверку ссылок, так что отключайте ограничения только для этой операции.

Если имя метода — строковый литерал в вашем исходном коде, а не содержимое переменной, вы можете присвоить соответсвующему символу напрямую:

```
{
    no warnings 'once';
    (*get_pie, *set_pie) =
         generate_accessors( 'pie' );
}
```

Прямое присваивание глобу не нарушает ограничения, но упоминание глоба только раз *будет* генерировать предупреждение «used only once», если вы явно не подавите его в области видимости.

Манипуляции времени компиляции

В отличие от кода, написанного явно как код, код, генерируемый с помощью строкового `eval`, компилируется во время выполнения. Там, где вы можете ожидать от нормальной функции, что она будет доступна всё время жизни программы, сгенерированная функция может не быть доступной, когда вы этого ожидаете.

Заставьте Perl выполнять код — генерирующий другой код — во время компиляции, обернув его в блок `BEGIN`. Когда парсер Perl 5 встречает блок `BEGIN`, он парсит весь блок. Если он не содержит синтаксических ошибок, блок будет выполнен немедленно. После его завершения парсинг продолжится, как если бы не было никаких прерываний.

Разница между записью:

```
sub get_age    { ... }
sub set_age    { ... }
```

```
sub get_name   { ... }
sub set_name   { ... }

sub get_weight { ... }
sub set_weight { ... }
```

...и:

```
sub make_accessors { ... }

BEGIN
{
    for my $accessor (qw( age name weight ))
    {
        my ($get, $set) =
            make_accessors( $accessor );

        no strict 'refs';
        *{ 'get_' . $accessor } = $get;
        *{ 'set_' . $accessor } = $set;
    }
}
```

...преимущественно в поддерживаемости.

Внутри модуля, любой код за пределами функций выполняется, когда вы используете его посредством `use`, из-за неявного `BEGIN`, который Perl добавляет вокруг `require` и `import` . Любой код за пределами функций, но внутри модуля, будет выполнен *перед* тем, как произойдёт вызов `import()`. Если вы загружаете модуль с помощью *require*, неявного блока `BEGIN` не будет. Выполнение кода за пределами функций произойдёт в *конце* парсинга.

Учитывайте взаимодействие между лексическими *объявлениями* (ассоциирование имени с областью видимости) и лексическими *присваиваниями*. Первые происходят во время компиляции, тогда как последние происходят в точке выполнения. Этот код имеет неочевидный баг:

```
# добавляет метод require() в UNIVERSAL
use UNIVERSAL::require;

# ведёт к ошибкам; не использовать
my $wanted_package = 'Monkey::Jetpack';

BEGIN
{
    $wanted_package->require();
    $wanted_package->import();
}
```

...потому что блок `BEGIN` будет выпволнен *перед* тем, как произойдёт присваивание строкового значения `$wanted_package`. Результатом будет исключение из-за попытки вызвать метод `require()` на неопределённом значении.

Class::MOP

В отличие от установки ссылок на функции для заполнения пространств имён и создания методов, в Perl 5 нет простого способа программного создания классов. На помощь приходит Moose с входящей в него библиотекой `Class::MOP`. Она предоставляет *протокол метаобъектов* — механизм для создания и манипулирования объектной системой в терминах её самой.

Вместо написания своего собственного хрупкого кода со строковым `eval` или попыток вручную ковыряться в таблицах символов, вы можете манипулировать сущностями и абстракциями вашей программы с помощью объектов и методов:

Так создаётся класс:

```
use Class::MOP;

my $class = Class::MOP::Class->create(
                'Monkey::Wrench'
            );
```

Добавьте атрибуты и методы в этот класс, когда создадите его:

```
my $class = Class::MOP::Class->create(
    'Monkey::Wrench' =>
    (
        attributes =>
        [
            Class::MOP::Attribute->new('$material'),
            Class::MOP::Attribute->new('$color'),
        ]
        methods =>
        {
            tighten => sub { ... },
            loosen  => sub { ... },
        }
    ),
);
```

...или в метакласс (объект, представляющий этот класс), когда он будет создан:

```
$class->add_attribute(
    experience  => Class::MOP::Attribute->new('$xp')
);

$class->add_method( bash_zombie => sub { ... } );
```

...и вы можете инспектировать метакласс:

```
my @attrs = $class->get_all_attributes();
my @meths = $class->get_all_methods();
```

Аналогично, `Class::MOP::Attribute` и `Class::MOP::Method` позволяют вам создавать, манипулировать и анализировать атрибуты и методы.

Перегрузка

Perl 5 не является глубоко объектно-ориентированным языком. Его внутренние типы данных (скаляры, массивы и хеши) не являются объектами с перегружаемыми методами, но вы *можете* контролировать поведение своих собственных классов и объектов, особенно когда они подвергаются приведению типов или контекстному вычислению. Это *перегрузка*.

Перегрузка может быть неочевидной, но мощной. Интересный пример — перегрузка того, как объект ведёт себя в булевом контексте, особенно если вы используете что-нибудь вроде шаблона ноль-объекта (http://www.c2.com/cgi/wiki?NullObject). В булевом контексте объект будет восприниматься как истинное значение, если вы не перегрузите преобразование в булево значение.

Вы можете перегрузить поведение объекта почти для любой операции или приведения типа: преобразование в строку, преобразование в число, преобразование в булево значение, итерации, вызов, обращение как к массиву, обращение как к хешу, арифметические операции, операции сравнения, умное сопоставление, побитовые операции и даже присваивание. Преобразование в строку, в число и в булево значение наиболее важны и наиболее распространены.

Перегрузка распространённых операций

Прагма `overload` позволяет вам ассоциировать функцию с операцией, которую вы можете перегрузить, передав пары аргументов, где ключ обозначает тип перегрузки, а значение — ссылка на функцию, которая должна вызываться для этой операции. Класс `Null`, перегружающий булево вычисление так, что он всегда будет восприниматься как ложное значение, может выглядеть так:

```
package Null
{
    use overload 'bool' => sub { 0 };

    ...
```

```
}
```

Можно легко добавить преобразование в строку:

```
package Null
{
    use overload
        'bool' => sub { 0 },
        < '""'   = sub { '(null)' }; >>
}
```

Переопределение преобразования в число более сложно, потому что арифметические операторы имеют тенденцию быть бинарными . Если есть два оператора, оба с перегруженными методами для сложения, какой будет приоритетнее? Ответ должен быть последовательным, простым для объяснения и понятным людям, которые не читали исходный код реализации.

`perldoc overload` пытается объяснить это в разделах *Calling Conventions for Binary Operations* и *MAGIC AUTOGENERATION*, но наиболее простое решение — перегрузить преобразование в число (имеющее ключ `'0+'`) и указать `overload` использовать предоставленные перегрузки как запасной вариант (fallback) где возможно:

```
package Null
{
    use overload
        'bool'   => sub { 0 },
        '""'     => sub { '(null)' },
        < '0+'     = sub { 0 }, >>
        < fallback = 1; >>
}
```

Установка `fallback` в истинное значение позволяет Perl использовать любые другие определённые перегрузки для выполнения запрошенной операции, если возможно. Если это невозможно, Perl будет действовать так,

как если бы не было никаких перегрузок. Это часто именно то, чего вы хотите.

Без `fallback` Perl будет использовать только конкретные перегрузки, которые вы предоставили. Если кто-нибудь попытается выполнить операцию, которую вы не перегрузили, Perl выбросит исключение.

Перегрузка и наследование

Подклассы наследуют перегрузки от своих предков. Они могут переопределить это поведение одним из двух способов. Если родительский класс использует перегрузку как показано, с напрямую предоставленными ссылками на функции, класс-потомок *должен* переопределить перегруженное поведение родителя с помощью прямого использования `overload`.

Родительские классы могут позволить своим потомкам большую гибкость, указывая *имя* метода для вызова в качестве реализации перегрузки, вместо хардкода ссылки на функцию:

```
package Null
{
    use overload
        'bool'   => 'get_bool',
        '""'     => 'get_string',
        '0+'     => 'get_num',
        fallback => 1;
}
```

В этом случае любые дочерние классы могут выполнять эти перегруженные операции иначе, переопределив методы с соответствующими именами.

Использование перегрузки

Перегрузка может выглядеть заманчивым инструментом для использования для генерации символьных сокращений для новых операций, но это редко

происходит в Perl 5 по весомым причинам. CPAN-дистрибутив `IO::All` доводит эту идею до предела в производстве умных идей для краткого и компонуемого кода. Однако на каждый блестящий API, доведённый до совершенства путём соответствующего использования перегрузки, дюжина других устраивают бардак. Иногда лучший код избегает умности в пользу простоты.

Переопределение сложения, умножения и даже конкатенации в классе `Matrix` имеет смысл только из-за того, что распространена существующая нотация для этих операций. Новая предметная область, не имеющая такой установившейся нотации — плохой кандидат для перегрузки, как и предметная область, в которой вам приходится изощряться, чтобы поставить существующие операторы Perl в соответствие с другой нотацией.

Perl Best Practices Демьена Конвея (Damian Conway) предлагают другое использование перегрузки: для предотвращения случайного неправильного использования объектов. Например, перегрузка преобразования в число таким образом, чтобы она делала `croak()`, для объектов, не имеющих разумного представления в виде одного числа, может помочь вам найти и исправить реальные ошибки.

Испорченность

Perl предоставляет инструменты для написания безопасных программ. Эти инструменты — не замена для внимательного обдумывания и планирования, но они *вознаграждают* внимание и понимание и могут помочь вам избежать неочевидных ошибок.

Использование режима испорченных данных

Режим испорченных данных (*taint mode*, или просто *taint*) добавляет метаданные ко всем данным, приходящим из-за пределов вашей программы. Любые данные, производные от испорченных данных, тоже испорчены. Вы можете использовать испорченные данные в своей программе, но если вы используете их для воздействия на окружающий мир — если вы используете их небезопасно — Perl выбросит фатальное исключение.

`perldoc perlsec` объясняет режим испорченных данных во всех подробностях.

Для включения режима испорченных данных запустите свою программу с аргументом командной строки `-T`. Если вы используете этот аргумент в строчке `#!`, вы должны запустить программу напрямую; если вы запустите её как `perl mytaintedapppl.pl` и пренебрежёте флагом `-T`, Perl выйдет с исключением. К тому моменту, когда Perl встречает флаг в строке `#!`, уже упущена возможность пометить испорченными данные окружения, например, входящие в `%ENV`.

Источники испорченных данных

Испорченные данные могут приходить из двух мест: файловый ввод и операционное окружение программы. Первое — это всё, что вы читаете из файла или получаете от пользователей в случае веб- или сетевого программирования. Второе включает любые аргументы командной строки, переменные окружения и данные из системных вызовов. Даже такие операции, как чтение из дескриптора директории, генерируют испорченные данные.

Функция `tainted()` из базового модуля `Scalar::Util` возвращает истину, если её аргумент испорчен:

```
die 'Oh no! Tainted data!'
    if Scalar::Util::tainted( $suspicious_value );
```

Удаление испорченности из данных

Чтобы удалить испорченность, вы должны выделить заведомо нормальные порции данных с помощью захвата в регулярном выражении. С захваченных данных будет снята испорченность. Если ваш пользовательский ввод содержит телефонный номер США, вы можете снять с него испорченность так:

```
die 'Number still tainted!'
    unless $number =~ /(\(/d{3}\) \d{3}-\d{4})/;

my $safe_number = $1;
```

Чем более конкретен ваш шаблон относительно того, что вы позволяете, тем более безопасной может быть ваша программа. Противоположный подход *запрета* конкретных элементов или форм имеет риск недосмотра чего-либо вредоносного. Гораздо лучше не пропустить что-нибудь безопасное, но неожиданное, чем пропустить что-нибудь опасное, но выглядящее безопасным. Несмотря на это, ничто не запрещает вам написать захват для всего содержимого переменной — но в этом случае зачем использовать режим испорченных данных?

Удаление испорченности из окружения

Суперглобальная переменная `%ENV` представляет переменные окружения системы. Эти данные испорчены, потому что силы за пределами контроля программы могут манипулировать данными в них. Любая переменная окружения, влияющая на то, как Perl или оболочка находит файлы и директории — это направление атаки. Чувствительная к испорченным данным программа должна удалить несколько ключей из `%ENV` и установить в `$ENV{PATH}` конкретный и хорошо защищённый путь:

```
delete @ENV{ qw( IFS CDPATH ENV BASH_ENV ) };
$ENV{PATH} = '/path/to/app/binaries/';
```

Если вы не установите `$ENV{PATH}` соответствующим образом, вы будете получать сообщения о его небезопасности. Если эта переменная окружения содержит текущую рабочую директорию, или если она содержит относительные директории, или если указанные директории имеют общий доступ на запись, умный атакующий может взломать системные вызовы для нанесения вреда.

По аналогичным причинам `@INC` в режиме испорченных данных не содержит текущую рабочую директорию. Perl также будет игнорировать

переменные окружения `PERL5LIB` и `PERLLIB`. Используйте прагму `lib` или флаг `-I` для `perl`, чтобы добавить библиотечные каталоги в программу.

Недостатки режима испорченных данных

Режим испорченных данных — это всё или ничего. Он либо включен, либо выключен. Это иногда приводит людей к использованию разрешительных шаблонов для снятия испорченности с данных и даёт иллюзию безопасности. Внимательно проверяйте снятие испорченности.

К сожалению, не все модули правильно обрабатывают испорченные данные. Это ошибка, которую CPAN-авторы должны принимать всерьёз. Если вам нужно сделать унаследованный код taint-безопасным, рассмотрите использование флага `-t`, который включает режим испорченных данных, но понижает его нарушения с исключений до предупреждений. Это не замена для полного режима, но это позволяет вам сделать существующие программы безопасными без подхода «всё или ничего» `-T`.

12. Perl за пределами синтаксиса

Детский Perl доведёт вас лишь до сюда. Беглость языка позволит вам использовать естественные шаблоны и идиомы языка. Эффективные программисты понимают, как возможности Perl взаимодействуют и сочетаются друг с другом.

Приготовьтесь к второй кривой обучения Perl: Perl-мышление. Результатом будет лаконичный, мощный код в стиле Perl.

Идиомы

Каждый язык — язык программирования или естественный — имеет распространённые шаблоны выражений, или *идиомы*. Земля вращается, но мы говорим о восходящем и заходящем солнце. Мы хвастаемся умными хаками и и морщимся от страшных хаков в процессе работы с кодом.

Идиомы Perl — не совсем возможности языка или техники дизайна. Это манеры и механизмы, которые, сложенные вместе, дают вашему коду перловый акцент. Вы не обязаны их использовать, но они придают Perl силу.

Объект как `$self`

Объектная система Perl 5 работает с инвокантом метода как с обычным параметром. Независимо от того, вызываете ли вы метод класса или экземпляра, первый элемент `@_` — всегда инвокант метода. По соглашению, большая часть кода на Perl 5 использует `$class` как имя инвоканта метода класса и `$self` как имя инвоканта объекта. Это соглашение достаточно сильно, чтобы полезные расширения, такие как `MooseX::Method::Signatures`, исходили из предположения, что бы используете `$self` как имя объектных инвокантов.

Именованные параметры

Обработка списков — фундаментальная составляющая обработки выражений в Perl 5. Способность Perl-программистов объединять в цепочки выражения, возвращающие списки переменной длины, предоставляет бесчисленные возможности для эффективного манипулирования данными.

Хотя простота передачи аргументов в Perl 5 (всё разглаживается в `@_`) иногда слишком проста, присваивание из `@_` в списочном контексте позволяет вам распаковывать именованные параметры как пары. Оператор толстой запятой превращает простой список в то, что очевидно является списком пар аргументов:

```
make_ice_cream_sundae(
    whipped_cream => 1,
    sprinkles     => 1,
    banana        => 0,
    ice_cream     => 'mint chocolate chip',
);
```

Вызываемая сторона может распаковать эти параметры в хеш и работать с хешем, как если бы он был одним аргументом:

```
sub make_ice_cream_sundae
{
    my %args    = @_;
    my $dessert = get_ice_cream( $args{ice_cream} );

    ...
}
```

Эта техника хорошо работает с `import()` или другими методами; обработайте столько параметров, сколько хотите, прежде чем поглотить оставшуюся часть в хеш:

```
sub import
```

```
{
    my ($class, %args)  = @_;
    my $calling_package = caller();

    ...
}
```

Преобразование Шварца

Преобразование Шварца — элегантная демонстрация всепроникающей обработки списков в Perl в виде идиомы, удачно позаимствованной из семейства языков Lisp.

Предположим, у вас есть Perl-хеш, ассоциирующий имена ваших коллег с их телефонными номерами:

```
my %extensions =
(
    '001' => 'Armon',
    '002' => 'Wesley',
    '003' => 'Gerald',
    '005' => 'Rudy',
    '007' => 'Brandon',
    '008' => 'Patrick',
    '011' => 'Luke',
    '012' => 'LaMarcus',
    '017' => 'Chris',
    '020' => 'Maurice',
    '023' => 'Marcus',
    '024' => 'Andre',
    '052' => 'Greg',
    '088' => 'Nic',
);
```

Чтобы отсортировать список по имени в алфавитном порядке, вы должны отсортировать хеш по его значениям, не ключам. Получить корректно отсортированные значения легко:

```
my @sorted_names = sort values %extensions;
```

...но вам нужен дополнительный шаг для сохранения связи имён и телефонов, тут и вступает в дело преобразование Шварца. Во-первых, сконвертируйте хеш в список структур данных, которые легко сортировать — в данном случае, двухэлементных анонимных массивов:

```
my @pairs = map  { [ $_, $extensions{$_} ] }
            keys %extensions;
```

`sort` принимает список анонимных массивов и сравнивает их вторые элементы (имена) как строки:

```
my @sorted_pairs = sort { $a->[1] cmp $b->[1] }
                        @pairs;
```

Блок, переданный в `sort`, принимает её аргументы в виде двух переменных `$a` и `$b`, принадлежащих области видимости пакета_(См. в `perldoc -f sort`) пространное обсуждение применений такой организации области видимости._. Блок `sort` принимает свои аргументы по два за раз; первый становится содержимым `$a`, а второй — содержимым `$b`. Если значение `$a` должно идти в результатах перед `$b`, блок должен вернуть -1. Если оба значения достаточно эквиваленты с точки зрения сортировки, блок должен вернуть 0. Наконец, если `$a` должно идти в результатах после `$b`, блок должен вернуть 1. Любые другие возвращаемые значения являются ошибками.

Оператор `cmp` выполняет строковое сравнение, а `<=>` выполняет числовое сравнение.

Имея `@sorted_pairs`, вторая операция `map` преобразует структуру данных в более удобную форму:

```
my @formatted_exts = map { "$_->[1], ext. $_->[0]" }
                         @sorted_pairs;
```

...и теперь вы можете всё это вывести:

```
say for @formatted_exts;
```

Преобразование Шварца само по себе использует всепроникающую обработку списков в Perl чтобы избавиться от временных переменных. Комбинация такая:

```
say for
    map  { " $_->[1], ext. $_->[0]"          }
    sort {   $a->[1] cmp   $b->[1]           }
    map  { [ $_      =>    $extensions{$_} ] }
        keys %extensions;
```

Читайте выражение справа налево, в порядке вычисления. Для каждого ключа в хеше телефонов создать двухэлементный анонимный массив, содержащий ключ и значение из хеша. Отсортировать этот список анонимных массивов по их вторым элементам, значениям из хеша. Сформировать строку вывода из этих отсортированных массивов.

Канал `map—sort—map` в преобразовании Шварца трансформирует структуру данных в другую, более подходящую для сортировки, а затем трансформирует её снова в другую форму.

Хотя этот пример сортировки прост, рассмотрите случай вычисления криптографического хеша для большого файла. Преобразование Шварца особенно полезно, потому что оно эффективно кеширует любые дорогие вычисления, выполняя их один раз в правом `map`.

Лёгкое поглощение файлов

`local` имеет большое значения для управления магическими глобальными переменными Perl 5. Вы должны понимать области видимости , чтобы эффективно использовать `local` — но если вам это удастся, вы можете использовать тесные и легковесные области видимости интересными способами. Например, для поглощения файлов в скаляр в одном выражении:

```
my $file = do { local $/ = <$fh> };

# или
my $file = do { local $/; <$fh> };

# или
my $file; { local $/; $file = <$fh> };
```

`$/` — это разделитель входных записей. Его локализация с помощью `local` устанавливает его значение в `undef`, ожидая присваивания. Эта локализация происходит *перед* присваиванием. Так как значение разделителя неопределённо, Perl охотно считает всё содержимое дескриптора файла сразу и присвоит это значение `$/`. Так как блок `do` возвращает значение последнего вычисленного внутри блока выражения, это возвращает значение присваивания: содержимое файла. Даже несмотря на то, что в конце блока `$/` немедленно возвращается к своему предыдущему состоянию, `$file` теперь вмещает содержимое файла.

Второй пример не содержит присваивания и всего лишь возвращает единственную строку, прочитанную из дескриптора файла.

Третий пример избегает повторного копирования строки, вмещающей содержимое файла; он не так красив, но использует меньше памяти.

Обращение с главным

Perl не требует специального синтаксиса для создания замыканий ; вы можете непреднамеренно замкнуть лексическую переменную. Многие программы обычно выставляют несколько лексичеких переменных области видимости файла, прежде чем передать обработку другим функциям. Возникает соблазн использовать эти переменные напрямую, вместо передачи и возврата значений из функций, особенно с ростом программы. К сожалению, эти программы могут начать полагаться на тонкости того, что происходит во время процесса компиляции Perl 5; переменная, которая, как вы *думали*, должна быть инициализирована определённым значением, может быть инициализирована лишь намного позже.

Чтобы этого избежать, оберните основной код вашей программы в простую функцию, `main()`. Инкапсулируйте ваши переменные в соответствующих областях видимости. Затем добавьте единственную строку в начало вашей программы, после подключения всех нужных модулей и прагм:

```
#!/usr/bin/perl

use Modern::Perl;

...

exit main( @ARGS );
```

Вызов `main()` *перед* чем либо ещё в программе заставляет вас быть ясными относительно инициализации и порядка компиляции. Вызов `exit` с возвращаемым `main()` значением предотвращает любой другой голый код от выполнения, хотя вы должны удостовериться, что возвращаете `0` из `main()` в случае успешного выполнения.

Контроллируемое выполнение

Фактическое различие между программой и модулем — в их намеренном использовании. Пользователи вызывают программы напрямую, тогда как программы загружают модули после того, как выполнение уже началось. Тем не менее, модуль — это Perl-код, точно так же, как и программа. Модуль легко сделать исполняемым. Так же как и заставить программу вести себя как модуль (полезно для тестирования частей существующей программы без формального преобразования её в модуль). Всё что вам нужно сделать — открыть для себя, *как* Perl начинает выполнение куска кода.

Единственный необязательный аргумент `caller` — это число фреймов вызова , которое нужно выдать. `caller(0)` выводит информацию о текущем фрейме вызова. Чтобы позволить модулю работать корректно как программа *или* модуль, поместите весь выполняемый код в функции, добавьте функцию `main()` и допишите одну строку в начало модуля:

```
main() unless caller(0);
```

Если у модуля *нет* вызывающего кода, значит, кто-то вызвал его напрямую как программу (с помощью `perl path/to/Module.pm`) вместо `use Module;`).

Постфиксная валидация параметров

В CPAN есть несколько модулей, помогающих проверить параметры ваших функций; два хороших варианта — `Params::Validate` и `MooseX::Params::Vali`. Выполнить простую валидацию легко даже без этих модулей.

Предположим, ваша функция принимает два аргумента, не больше и не меньше. Вы *могли бы* написать:

```
use Carp 'croak';

sub groom_monkeys
{
    if (@_ != 2)
    {
        croak 'Grooming requires two monkeys!';
    }
    ...
}
```

...но с лингвистической точки зрения последствия более важны, чем проверка, и заслуживают быть в *начале* выражения:

```
croak 'Grooming requires two monkeys!' if @_ != 2;
```

...что может читаться проще в следующем виде:

```
croak 'Grooming requires two monkeys!'
    unless @_ == 2;
```

Эта техника раннего возврата — особенно с постфиксными условиями — может упростить оставшуюся часть кода. Каждая такая проверка — это фактически одна строка в таблице истинности.

Регулярные выражения мимоходом

Многие идиомы Perl 5 полагаются на тот факт, что выражения возвращают значения:

```
say my $ext_num = my $extension = 42;
```

Хотя этот код очевидно неуклюж, он демонстрирует, как использовать значение одного выражения в другом выражении. Эта идея не нова; вы с большой вероятностью и раньше использовали возвращаемое значение функции в списке или как аргумент другой функции. Вы, возможно, не осознавали последствий этого.

Предположим, вы хотите извлечь имя из сочетания имени и фамилии с помощью прекомпилированного регулярного выражения в `$first_name_rx`:

```
my ($first_name) = $name =~ /($first_name_rx)/;
```

В списочном контексте успешное сопоставление регулярному выражению возвращает список всех захватов , и Perl присваивает первый из них `$first_name`.

Чтобы изменить имя, возможно, удалив все несловарные символы для создания пригодного имени пользователя для системного аккаунта, вы можете написать:

```
(my $normalized_name = $name) =~ tr/A-Za-z//dc;
```

Сперва присвойте значение `$name` `$normalized_name`, так как скобки влияют на приоритет, это присваивание произойдёт первым. Выражение присваивания возвращает *переменную* `$normalized_name`, так что эта переменная становится первым операндом оператора транслитерации.

Эта техника работает и с другими операциями модификации на месте:

```
my $age = 14;
(my $next_age = $age)++;

say "Next year I will be $next_age";
```

Унарное приведение типа

Система типов Perl 5 почти всегда делает то, что нужно, если вы выбираете правильные операторы. Используйте оператор строковой конкатенации, и Perl будет обращаться с обоими операндами как со строками. Используйте оператор сложения, и Perl будет обращаться с обоими операндами как с числовыми.

Иногда вам придётся дать Perl подсказку о том, что вы имеете ввиду, с помощью *унарного приведения типа*, чтобы заставить вычисление значения выполняться конкретным образом.

Чтобы убедиться, что Perl обрабатывает значение как числовое, прибавьте ноль:

```
my $numeric_value = 0 + $value;
```

Чтобы убедиться, что Perl обрабатывает значение как булево, используйте двойное отрицание:

```
my $boolean_value = !! $value;
```

Чтобы убедиться, что Perl обрабатывает значение как строку, конкатенируйте его с пустой строкой:

```
my $string_value = '' . $value;
```

Хотя потребность в этих приведениях типа исчезающе мала, вы должны понимать эти идиомы, если они вам встретятся. Хотя удаление «бесполезного» `+ 0` из выражения может выглядеть безопасным, это может сломать код.

Глобальные переменные

Perl 5 предоставляет несколько *суперглобальных переменных*, которые поистине глобальны, не ограничены областью видимости пакета или файла. К сожалению, их глобальная доступность означает, что любые прямые или непрямые модификации могут повлиять на другие части программы — и они немногословны. Опытные программисты на Perl 5 запоминают некоторые из них. Мало кто помнит их все. Лишь некоторые из них бывают полезны. `perldoc perlvar` содержит исчерпывающий список таких переменных.

Управление суперглобальными переменными

Perl 5 продолжает перемещать ещё больше глобального поведения в лексическое поведение, так что вы можете избежать многих из этих глобальных переменных. Когда вы не можете их избежать, используйте `local` в наименьшей возможной области видимости чтобы ограничить любые изменения. Вы всё ещё восприимчивы к любым изменениям, которые *вызываемый* вами код сделает с этими глобальными переменными, но вы уменьшаете вероятность неожиданного кода *снаружи* вашей области видимости. Как демонстрирует идиома лёгкого поглощения файла , `local` зачастую правильный подход:

```
my $file; { local $/; $file = <$fh> };
```

Эффект локализации `$/` длится только до конца блока. Невелик шанс, что какой-либо Perl-код будет выполнен как результат чтения строк из дескриптора файла_(Связанный дескриптор файла — одна из немногих вероятностей._ и изменит значение `$/` внутри блока `do`.

Не все случаюи использования суперглобальных переменных так легко защитить, но это зачастую работает.

В других случаях вам нужно *прочитать* значение суперглобальной переменной, и надеяться, что другой код его не изменил. Отлавливание исключений с помощью блока `eval` может быть восприимчиво к условиям гонки, поскольку метод `DESTROY()`, вызванный на лексической переменной, которая вышла за пределы области видимости, может сбросить `$@`:

```
local $@;

eval { ... };

if (my $exception = $@) { ... }
```

Копируйте `$@` *сразу же* после поимки исключения, чтобы сохранить её содержимое. Смотрите также `Try::Tiny` вместо этого .

Английские имена

Базовый модуль `English` предоставляет подробные имена для насыщенных пунктуацией суперглобальных переменных. Импортируйте их в пространство имён следующим образом:

```
use English '-no_match_vars';
```

Это позволит вам использовать подробные имена, документированные в `perldoc perlvar`, внутри области видимости этой прагмы.

Три связанных с регулярными выражениями суперглобальных переменных (`$&`, `$`` и `$'`) приводят к снижению производительности *всех* регулярных

выражений в программе. Если вы забудете `-no_match_vars` при импорте, ваша программа получит минус производительности, даже если вы не читаете явно из этих переменных.

Программы на Современном Perl должны использовать переменную `@-` как замену для этих ужасных трёх.

Полезные суперглобальные переменные

Большая часть программ на современном Perl 5 может обойтись использованием только нескольких из суперглобальных переменных. Вы с большой вероятностью встретите лишь немногие из этих переменных в реальных программах.

- `$/` (или `$INPUT_RECORD_SEPARATOR` из прагмы `English`) это строка из нуля или более символов, обозначающая конец записи при построчном чтении входных данных. По умолчанию, это платформозависимая последовательность символов новой строки. Если вы сделаете это значение неопределённым, Perl будет пытаться прочитать файл в память целиком. Если вы установите это значение в *ссылку* на целое число, Perl будет пытаться прочитать именно столько *байт* на запись (так что не забывайте о вопросах Юникода). Если вы установите это значение в пустую строку (`''`), Perl будет читать по параграфу за раз, где параграф — это порция текста, за которой следует произвольное количество новых строк.
- `$.` (`$INPUT_LINE_NUMBER`) содержит количество записей, прочитанных из дескриптора файла, к которому последний раз осуществлялся доступ. Вы можете читать из этой переменной, но запись в неё не даст никакого эффекта. Локализация этой переменной будет локализовать дескриптор файла, на который она ссылается.
- `$|` (`$OUTPUT_AUTOFLUSH`) управляет тем, будет ли Perl сбрасывать всё, что записано в текущий выбранный дескриптор файла, сразу же, или только когда буфер Perl заполнится. Небуферизованный вывод полезен при записи в канал, или сокет, или терминал, который не должен блокироваться ожиданием ввода. Эта переменная будет приводить любые присвоенные ей значение в булевы.
- `@ARGV` содержит агрументы командной строки, переданные в программу.

- `$!` (`$ERRNO`) это двойная переменная , содержащая результат *последнего* системного вызова. В числовом контексте она соответствует значению `errno` в C, где всё, кроме ноля, обозначает ошибку. В строковом контексте она возвращает соответствующую строку системной ошибки. Локализуйте эту переменную, прежде чем выполнить системный вызов (явно или неявно), чтобы избежать перезаписывания соответствующего значения для другого кода где бы то ни было. Многие места в самом Perl 5 делают системные вызовы, о которых вы не знаете, так что значение этой переменной может измениться независимо от вас. Копируйте его *сразу же* после выполнения системного вызова для получения наиболее точных результатов.

- `$"` (`$LIST_SEPARATOR`) это строка, используемая для разделения элементов массивов и списков, интерполируемых в строку.

- `%+` содержит именованные захваты из успешного сопоставления регулярному выражению .

- `$@` (`$EVAL_ERROR`) содержит значение, выброшенное из последнего исключения .

- `$0` (`$PROGRAM_NAME`) содержит имя выполняемой программы. Вы можете модифицировать это значение на некоторых Unix-подобных платформах чтобы изменить имя программы, под которым она предстаёт перед другими программами, такими как `ps` или `top`.

- `$$` (`$PID`) содержит идентификатор процесса текущего выполняемого экземпляра программы, как его понимает операционная система. Он будет различаться между программами, разветвлёнными с помощью `fork()`, и *может* различаться между потоками в одной и той же программе.

- `@INC` содержит список путей в файловой системе, по которым Perl будет искать файлы для загрузки с помощью `use` или `require`. Смотрите `perldoc -f require` для информации о других элементах, которые этот массив может содержать.

- `%SIG` ставит в соответствие низкоуровневым сигналам ОС и Perl ссылки на функции, используемые для обработки этих сигналов. Например, можно отловить стандартное прерывание по Ctrl-C с помощью сигнала `INT`. См. `perldoc perlipc` для большей информации о сигналах и особенно безопасных сигналах.

Альтернативы суперглобальным переменным

Самые тяжёлые обвинения о воздействии на расстоянии относятся к вводу/выводу и исключительным условиям. Использование `Try::Tiny` поможет вам защититься от хитрой семантики правильной обработки исключений. Локализация и копирование значения `$!` может помощь вам избежать странных поведений, когда Perl выполняет неявные системные вызовы. Использование `IO::File` и его методов на лексических дескрипторах файлов поможет предотвратить нежелательные глобальные изменения поведения ввода/вывода.

13. Чего следует избегать

Perl 5 не идеален. Некоторые возможности сложно использовать корректно. Другие никогда не работали хорошо. Некоторые — странные комбинации других возможностей со странными граничными случаями. Хотя лучше избегать этих возможностей, знание того, почему их нужно избегать, поможет вам найти более подходящие решения.

Голые слова

Perl — податливый язык. Вы можете писать программы в предпочитаемой вами наиболее творческой, поддерживаемой, запутанной или эксцентричной манере. Хорошие программисты заботятся о поддерживаемости, но Perl не пытается диктовать, что *вы* считаете поддерживаемым.

Парсер Perl понимает встроенные функции и операторы Perl. Он использует сигилы для идентификации переменных и другую пунктуацию для распознавания вызовов функций и методов. Однако, иногда парсеру приходится догадываться, что вы имеете ввиду, особенно когда вы используете *голое слово* — идентификатор без сигила или другой синтаксически значимой пунктуации.

Хорошее использование голых слов

Хотя прагма `strict` справедливо запрещает неоднозначные голые слова, некоторые голые слова приемлемы.

Голые слова как ключи хеша Ключи хеша в Perl 5 обычно *не* допускают неоднозначности, потому что парсер может идентифицировать их как строковые ключи; `pinball` в `$games{pinball}` — это очевидно строка.

Иногда эта интерпретация — не то, чего вы хотите, особенно если вы намеревались *вычислить* встроенную или пользовательскую функцию, чтобы сгенерировать ключ хеша. В этом случае, устраните неоднозначность,

указав аргументы, используя круглые скобки для аргументов функции, или предварите унарным плюсом, чтобы форсировать вычисление встроенной функции:

```
# ключом будет литеральный E<laquo>shiftE<raquo>
my $value = $items{shift};

# ключом будет значение, возвращённое E<laquo>shiftE<raquo>
my $value = $items{shift @_}

# унарный плюс приводит к использованию встроенной функции E<laquo
my $value = $items{+shift};
```

Голые слова как имена пакетов Имена пакетов в Perl 5 тоже являютя голыми словами. Если вы следуете соглашениям об именовании, согласно которым имена пакетов начинаются с заглавных букв, а функций — нет, маловероятно, что вы встретитесь с коллизиями имён, но парсер Perl 5 должен определить, как парсить `Package->method()`. Значит ли это «вызвать функцию с именем `Package()` и вызвать метод `method()` на возвращаемом ей значении» или «вызвать метод с именем `method()` в пространстве имён `Package`»? Ответ различается в зависимости от того, какой код парсер уже встретил в текущем пространстве имён.

Принудите парсер воспринимать `Package` как имя пакета, добавив разделитель пакетов (`::`)*(Даже среди тех, кто понимает, почему это работает, очень немногие это делают.)*:

```
# вероятно метод класса
Package->method();

# определённо метод класса
Package::->method();
```

Голые слова как имена блоков кода Специальные именованные блоки кода `AUTOLOAD`, `BEGIN`, `CHECK`, `DESTROY`, `END`, `INIT` и `UNITCHECK` — это голые слова, которые *объявляют* функции без использования встроенной директивы `sub`. Вы уже видели это раньше :

```
package Monkey::Butler;

BEGIN { initialize_simians( __PACKAGE__ ) }

sub AUTOLOAD { ... }
```

Хотя вы *можете* опустить `sub` из объявления `AUTOLOAD()`, немногие так делают.

Голые слова как константы Константы, объявленные с помощью прагмы `constant`, можно использовать как голые слова:

```
# не используйте это для реальной аутентификации
use constant NAME     => 'Bucky';
use constant PASSWORD => '|38fish!head74|';

return unless $name eq NAME && $pass eq PASSWORD;
```

Обратите внимание, что эти константы *не* интерполируются в строках, заключённых в двойные кавычки.

Константы — специальный случай прототипированных функций . Когда вы предварительно объявляете функцию с прототипом, парсер знает, как воспринимать эту функцию, и будет предупреждать об ошибках неоднозначности парсинга. Все остальные недостатки прототипов всё ещё имеют место.

Неблагоразумное использование голых слов

Независимо от того, насколько внимательно вы пишете код, голые слова всё равно приводят к неоднозначностям. Вы можете избежать большинства их использований, но вам встретится несколько типов голых слов в унаследованном коде.

Голые вызовы функций Код, написанный без `strict 'subs'`, может использовать голые имена функций. Добавление скобок заставляет код пройти эти ограничения. Используйте `perl -MO=Deparse,-p` (см. `perldoc B::Deparse`) чтобы понять, как Perl парсит их, затем расставляйте скобки соответствующим образом.

Голые значения хешей Некоторый старый код может не заботиться о том, чтобы заключить в кавычки *значения* хеш-пар:

```
# плохой стиль; не использовать
my %parents =
(
    mother => Annette,
    father => Floyd,
);
```

Если не существует ни функции `Floyd()`, ни `Annette()`, Perl будет интерпретировать эти голые слова как строки. `strict 'subs'` выдаст ошибку в этой ситуации.

Голые дескрипторы файлов До появления лексических дескрипторов файлов , все дескрипторы файлов и директорий использовали голые слова. Вы почти всегда можете безопасно переписать этот код на использование лексических дескрипторов файлов; исключения — `STDIN`, `STDOUT` и `STDERR`. К счастью, парсер Perl распознаёт их.

Голые функции sort Наконец, встроенная фукнция `sort` может принимать в качестве второго аргумента *имя* функции, которую нужно использовать для сортировки. Хотя это редко бывает неоднозначным для парсера, это может смутить читающих *людей*. Альтернатива в виде передачи ссылки на функцию в скаляре немного лучше:

```
# стиль с использованием голого слова
my @sorted = sort compare_lengths @unsorted;
```

```
# ссылка на фукнцию в скаляре
my $comparison = \&compare_lengths;
my @sorted     = sort $comparison @unsorted;
```

Второй вариант избегает использования голого слова, но результат на одну строчку длиннее. К сожалению, парсер Perl 5 *не* понимает однострочную версию вследствие специального парсинга `sort`; вы не можете использовать произвольное выражение (такое как взятие ссылки на именованную функцию) там, где может пройти блок или скаляр.

```
# не работает
my @sorted = sort \&compare_lengths @unsorted;
```

В обоих случаях то, как `sort` вызывает функцию и передаёт аргументы, может быть запутывающим (см. `perldoc -f sort` для подробностей). Где возможно, рассмотрите использование вместо этого блочной формы `sort`. Если же вы должны использовать какую-либо из форм с функцией, рассмотрите добавление объясняющего комментария.

Непрямые объекты

В Perl 5 нет оператора `new`; конструктором в Perl 5 является всё, что возвращает объект. По соглашению, конструкторы — методы классов, называемые `new()`, но вы можете выбрать всё, что захотите. Некоторые старые руководства по объектам Perl 5 поощряют использование вызовов конструкторов в стиле C++ и Java:

```
my $q = new CGI; # НЕ ИСПОЛЬЗОВАТЬ
```

…вместо очевидного вызова метода:

```
my $q = CGI->new();
```

Эти варианты синтаксиса дают одинаковое поведение, за исключением случаев, когда это не так.

Непрямые вызовы с голыми словами

В непрямой объектной форме (более точно, в *дательном* случае) первого примера глагол (метод) предшествует существительному, к которому относится (объект). Это нормально в разговорных языках, но в Perl 5 создаёт неоднозначности парсинга.

Так как имя метода — голое слово , парсер должен предсказать правильную интерпретацию кода с помощью использования нескольких эвристик. Хотя эти эвристики хорошо протестированы и *почти* всегда корректны, их режимы сбоя сбивают с толку. Хуже того, они зависят от порядка компиляции кода и модулей.

Трудность парсинга увеличивается, если конструктор принимает аргументы. Непрямой стиль может выглядеть так:

```
# НЕ ИСПОЛЬЗОВАТЬ
my $obj = new Class( arg => $value );
```

...таким образом заставляя имя класса выглядеть как вызов функции. Perl 5 *может* устранить неоднозначность во многих подобных случаях, но его эвристики зависят от того, какие имена пакетов видит парсер, какие голые слова он уже разрешил (и как он разрешил их) и от *имён* функций, уже объявленных в текущем пакете.

Представьте противоречие в случае прототипированной функции с именем, которому случилось каким-то образом вступить в конфликт с именем класса или метода, вызываемого непрямо. Это случается редко, но так неприятно для отладки, что стоит того, чтобы избегать непрямых вызовов.

Скалярные ограничения непрямой нотации

Другая опасность такого синтаксиса в том, что парсер ожидает объект как одно скалярное выражение. Вывод в дескриптор файла, сохранённый в агрегатной переменной, *выглядит* очевидным, но таковым не является:

```
# НЕ РАБОТАЕТ КАК НАПИСАНО
say $config->{output} 'Fun diagnostic message!';
```

Perl попытается вызвать `say` на объекте `$config`.

`print`, `close` и `say` — все встроенные функции, оперирующие с дескрипторами файлов — оперируют в непрямой манере. Это было нормально, когда дескрипторы файлов были глобальными переменными пакета, но лексические дескрипторы файлов делают очевидными проблемы непрямого объектного синтаксиса. Чтобы устранить это, избавьтесь от неоднозначности подвыражения, выдающего подразумеваемый инвокант:

```
say {$config->{output}} 'Fun diagnostic message!';
```

Альтернативы непрямой нотации

Нотация прямого вызова не страдает этой проблемой неоднозначности. Чтобы сконструировать объект, вызовите метод-конструктор напрямую на имени класса:

```
my $q   = CGI->new();
my $obj = Class->new( arg => $value );
```

Этот синтаксис *всё ещё* имеет проблему голого слова, в том смысле, что если у вас есть функция с именем `CGI`, Perl будет интерпретировать голое имя класса как вызов функции:

```
sub CGI;

# вы написали CGI->new(), но Perl увидел
my $q = CGI()->new();
```

Хотя это случается редко, вы можете устранить неоднозначность имён классов, добавив разделитель пакетов (`::`) или явно пометив имена классов как строковые литералы:

```
# разделитель пакетов
my $q = CGI::->new();

# не имеющий неоднозначности строковый литерал
my $q = 'CGI'->new();
```

Однако почти никто этого не делает.

Для ограниченного случая операций с дескрипторами файлов дательное использование настолько общепринято, что вы можете использовать подход непрямого вызова, если окружите свой подразумеваемый инвокант фигурными скобками. Если вы используете Perl 5.14 (или если вы загрузили `IO::File` или `IO::Handle`), вы можете использовать методы на лексических дескрипторах файлов_(Хотя почти никто не делает этого для `print`) и `say`._.

CPAN-модуль `Perl::Critic::Policy::Dynamic::NoIndirect` (плагин для `Perl::Critic`) может обнаружить непрямые вызовы во время проверки кода. CPAN-модуль `indirect` может обнаружить и запретить их использование в запущенных программах:

```
# выдать предупреждение при использовании непрямых вызовов
no indirect;

# выбросить исключение при их использовании
no indirect ':fatal';
```

Прототипы

Прототип — это набор необязательных метаданных, присоединённых к функции, которые влияют на то, как парсер понимает её аргументы. Хотя внешне они могут выглядеть как сигнатуры функций в других языках, на самом деле они очень отличаются.

Прототипы позволяют пользователям определять свои собственные функции, ведущие себя как встроенные. Рассмотрим встроенную фукнцию `push`, которая принимает массив и список. Хотя обычно Perl 5 разгладил бы массив и список в единый список, переданный в `push`, парсер знает, что не нужно разглаживать массив, чтобы `push` мог модифицировать его на месте.

Прототипы функций являются частью объявления:

```
sub foo         (&@);
sub bar         ($$) { ... }
my  $baz = sub (&&) { ... };
```

Любой прототип, привязанный к предварительному объявлению, должен совпадать с прототипом, привязанным к объявлению функции. Perl выдаст предупреждение, если это не так. Как ни странно, вы можете опустить прототип в предварительном объявлении и включить его в полное объявление — но делать так нет причин.

Встроенная функция `prototype` принимает имя функции и возвращает строку, представляющую её прототип. Используйте форму `CORE::`, чтобы увидеть прототип встроенной функции:

`prototype` вернёт `undef` для тех встроенных функций, которые вы не можете эмулировать:

Помните `push`?

Символ `@` представляет список. Обратный слеш принуждает к использованию *ссылки* для соответствующего аргумента. Этот прототип обозначает, что `push` принимает ссылку на массив и список значений. Вы можете написать `mypush` так:

```
sub mypush (\@@)
{
    my ($array, @rest) = @_;
    push @$array, @rest;
}
```

Другие символы прототипов включают `$` для скалярного аргумента, `%` для обозначения хеша (чаще всего используется как ссылка) и `&` для указания на блок кода. См. `perldoc perlsub` для полной документации.

Проблемы с прототипами

Прототипы изменяют то, как Perl парсит ваш код, и могут вызывать приведение типов аргументов. Они не документируют количество или типы аргументов, которые функция ожидает, как и не устанавливают соответствие аргументов именованным параметрам.

Приведение типов при использовании прототипов работает неочевидным образом, как навязывание скалярного контекста входным аргументам:

```
sub numeric_equality($$)
{
    my ($left, $right) = @_;
    return $left == $right;
}

my @nums = 1 .. 10;

say 'They're equal, whatever that means!'
    if numeric_equality @nums, 10;
```

...но работает только с простыми выражениями:

```
sub mypush(\@@);
```

```
# ошибка компиляции: несоответствие прототипов
# (ожидался массив, получено скалярное присваивание)
mypush( my $elems = [], 1 .. 20 );
```

Чтобы отладить это, пользователи `mypush` должны знать как то, что прототип существует, так и ограничения прототипов массивов. Хуже того, это *простые* ошибки, которые могут вызывать прототипы.

Хорошее использование прототипов

Немногие использования прототипов достаточно интересны, чтобы перевесить их недостатки, но они существуют.

Во-первых, они могут позволить вам переопределить встроенные функции. Сначала проверьте, что вы *можете* переопределить встроенную функцию, проверив её прототип в маленькой тестовой программе. Затем используйте прагму `subs`, чтобы сказать Perl, что вы собираетесь переопределить встроенную функцию, и, наконец, объявите ваше переопределение с корректным прототипом:

```
use subs 'push';

sub push (\@@) { ... }
```

Имейте ввиду, что прагма `subs` действует для всей оставшейся части *файла*, независимо от лексической области видимости.

Вторая причина использования прототипов — это определение констант времени компиляции. Когда Perl встречает функцию, объявленную с пустым прототипом (в противоположность *отсутствию* прототипа), *и* эта функция возвращает единственное константное выражение, оптимизатор превратит все вызовы этой функции в константы вместо вызовов функции:

```
sub PI () { 4 * atan2(1, 1) }
```

Весь последующий код будет использовать вычисленное значение Пи на месте голого слова `PI` или вызова `PI()`, с учётом области видимости.

Базовая прагма `constant` справляется с этими деталями за вас. Модуль `Const::Fast` из CPAN создаёт константные скаляры, которые вы можете интерполировать в строки.

Разумное использование прототипов — это расширение синтаксиса Perl 5 для оперирования анонимными функциями как блоками. CPAN-модуль `Test::Exception` использует это во благо для предоставления приятного API с отложенными вычислениями_(См. также `Test::Fatal`)_. Его функция `throws_ok()` принимает три аргумента: блок кода для выполнения, регулярное выражение для сравнения со строкой исключения и необязательное описание теста:

```
use Test::More tests => 1;
use Test::Exception;

throws_ok
    { my $unobject; $unobject->yoink() }
    qr/Can't call method "yoink" on an undefined/,
    'Method on undefined invocant should fail';
```

Экспортируемая функция `throws_ok()` имеет протоип `&$;$`. Его первый аргумент — это блок, который становится анонимной функцией. Второй аргумент — скаляр. Третий аргумент не обязателен.

Внимательные читатели, возможно, заметили отсутствие запятой после блока. Это причуда парсера Perl 5, который ожидает после прототипированного блока пробел, а не оператор запятой. Это недостаток синтаксиса прототипов.

Вы можете использовать `throws_ok()`, не пользуясь прототипами:

```
use Test::More tests => 1;
use Test::Exception;

throws_ok(
    sub { my $unobject; $unobject->yoink() },
```

```
    qr/Can't call method "yoink" on an undefined/,
    'Method on undefined invocant should fail' );
```

Последнее хорошее применение прототипов — при определении произвольной именованной функции для использования с `sort`*(Бен Тилли (Ben Tilly) предложил этот пример.)*:

```
sub length_sort ($$)
{
    my ($left, $right) = @_;
    return length($left) <=> length($right);
}

my @sorted = sort length_sort @unsorted;
```

Прототип `$$` заставляет Perl передавать сортируемые пары в `@_`. Документация `sort` предполагает, что это немного медленнее, чем использование глобальных переменных пакета `$a` и `$b`, но использование лексических переменных зачастую оправдывает любое снижение скорости.

Эквивалентность методов и функций

Объектная система Perl 5 преднамеренно минималистична . Так как класс является пакетом, Perl не делает различий между функцией и методом, хранящимися в пакете. Одна и та же встроенная директива, `sub`, объявляет и одно, и другое. Документация может прояснить ваши намерения, но Perl охотно диспетчеризует функцию, вызванную как метод. Подобным же образом вы можете вызвать метод как если бы он был функцией — полностью определённой, экспортированной или ссылкой — если вручную передадите свой инвокант.

Вызов не того, что нужно, не так, как нужно, вызывает проблемы.

Вызывающая сторона

Рассмотрим класс с несколькими методами:

```
package Order;

use List::Util 'sum';

...

sub calculate_price
{
    my $self = shift;
    return sum( 0, $self->get_items() );
}
```

Если есть объект `$o` класса `Order`, следующие вызовы этого метода *могут* выглядеть эквивалентными:

```
my $price = $o->calculate_price();

# сломано; не использовать
my $price = Order::calculate_price( $o );
```

Хотя в этом простом случае они дадут один и тот же вывод, последнее нарушает инкапсуляцию объекта, избегая поиска метода.

Если же `$o` был бы подклассом или алломорфом класса `Order`, который переопределял бы `calculate_price()`, обход диспетчеризации метода привёл бы к вызову неверного метода. Любое изменение реализации `calculate_price()`, такое как модификация наследования или делегирования через `AUTOLOAD()` — могло бы сломать вызывающий код.

В Perl есть одно обстоятельство, где это поведение может выглядеть необходимым. Если вы форсируете разрешение метода без диспетчеризации, как вы вызовете результирующую ссылку на метод?

```
my $meth_ref = $o->can( 'apply_discount' );
```

Здесь есть две возможности. Первая — отбросить возвращаемое значение метода `can()`:

```
$o->apply_discount() if $o->can( 'apply_discount' );
```

Вторая — использовать саму ссылку посредством синтаксиса вызова метода:

```
if (my $meth_ref = $o->can( 'apply_discount' ))
{
    $o->$meth_ref();
}
```

Если `$meth_ref` содержит ссылку на функцию, Perl вызовет эту ссылку с `$o` в качестве инвоканта. Это работает даже со включенным `strict`, как и при вызове метода с помощью скаляра, содержащего его имя:

```
my $name = 'apply_discount';
$o->$name();
```

Есть один небольшой недостаток в вызове метода по ссылке; если структура программы изменяется между сохранением ссылки и вызовом по ссылке, ссылка может уже не ссылаться на наиболее соответствующий метод. Если класс `Order` был изменён так, что `Order::apply_discount` уже не является правильным методом для вызова, ссылка в `$meth_ref` не будет обновлена.

Когда вы используете эту форму вызова, ограничьте область видимости ссылки.

Вызываемая сторона

Так как Perl 5 не делает различий между функциями и методами в точке объявления, и так как *возможно* (хотя и не рекомендуется) вызвать заданную функцию как функцию или метод, возможно написать функцию, которую можно вызывать любым из этих способов. Базовый модуль `CGI` — главный обвиняемый. Его функции применяют несколько эвристик для определения того, является ли их первый аргумент инвокантом.

Недостатков этого множество. Трудно точно предсказать, какой инвокант потенциально валиден для заданного метода, особенно если вам приходится иметь дело с подклассами. Также труднее создать API, который пользователи не смогут с лёгкостью неправильно использовать, как и бремя документации становится тяжелее. Что случится, если одна часть проекта использует процедурный интерфейс, а другая — объектный?

Если вам *необходимо* предоставить отдельный процедурный и ОО интерфейс к библиотеке, создайте два отдельных API.

Связывание

Тогда как перегрузка позволяет вам настраивать поведение классов и объектов для конкретных случаев приведения типа, механизм, называемый *связыванием*, позволяет вам настраивать поведение простых переменных (скаляров, массивов, хешей и дескрипторов файлов). Любая операция, которую вы можете выполнить на связанной переменной, транслируется в определённый вызов метода.

Встроенная директива `tie` изначально позволяла вам использовать место на диске как резервную память для хешей, так что Perl мог осуществлять доступ к файлам, которые больше, чем можно поместить в память. Базовый модуль `Tie::File` предоставляет похожую систему и позволяет вам обращаться с файлами как если бы они были массивами.

Класс, с которым вы связываете переменную с помощью `tie()`, должен соотвествовать определённому интерфейсу для конкретных типов данных. Смотрите `perldoc perltie` для получения общего представления, затем

обратитесь к базовым модулям `Tie::StdScalar`, `Tie::StdArray` и `Tie::StdHash` за более конкретными деталями. Начните с наследования от одного из этих классов, затем переопределите любые конкретные методы, которые вам нужно модифицировать.

Связывание переменных

Так можно связать переменную:

```
use Tie::File;
tie my @file, 'Tie::File', @args;
```

Первый аргумент — это переменная для связывания, второй — имя класса, с которым её нужно связать, а `@args` — необязательный список аргументов, требуемый для связывающей функции. В случае `Tie::File` это допустимое имя файла.

Связывающие функции напоминают конструкторы: `TIESCALAR`, `TIEARRAY()`, `TIEHASH()` или `TIEHANDLE()` для скаляров, массивов, хешей и дескрипторов файлов соответственно. Каждая функция возвращает новый объект, представляющий связанную переменную. Обе встроенные директивы, `tie` и `tied`, возвращают этот объект. Однако, большинство использует `tied` в булевом контексте.

Реализация связанных переменных

Чтобы реализовать класс связанных переменных, унаследуйте от встроенного модуля, такого как `Tie::StdScalar`*(У `Tie::StdScalar`) отсутствует собственный файл `.pm`, так что используйте `Tie::Scalar`, чтобы сделать его доступным.*, затем переопределите конкретные методы для операций, которые вы хотите изменить. В случае связанного скаляра это, скорее всего, будут `FETCH` и `STORE`, возможно, `TIESCALAR()`, и, вероятно, не `DESTROY()`.

Вы можете создать класс, который логирует все операции чтения и записи скаляра, с помощью очень небольшого количества кода:

```
package Tie::Scalar::Logged
{
    use Modern::Perl;

    use Tie::Scalar;
    use parent -norequire => 'Tie::StdScalar';

    sub STORE
    {
        my ($self, $value) = @_;
       Logger->log("Storing <$value> (was [$$self])", 1);
        $$self = $value;
    }

    sub FETCH
    {
        my $self = shift;
        Logger->log("Retrieving <$$self>", 1);
        return $$self;
    }
}

1;
```

Допустим, метод `log()` класса `Logger` принимает строку и количество фреймов вверх по стеку вызовов, которые нужно вывести.

Внутри методов `STORE()` и `FETCH()`, `$self` работает как благословлённый скаляр. Присваивание этой ссылке на скаляр изменяет значение скаляра, а чтение из неё возвращает его значение.

Аналогично, методы `Tie::StdArray` и `Tie::StdHash` воздействуют на благословлённые ссылки на массив и хеш сооветственно. Документация в `perldoc perltie` объясняет обширное количество методов, которые они поддерживают, как например, вы можете читать или записывать множественные значения в них, помимо других операций.

Когда использовать связанные переменные?

Связанные переменные выглядять как весёлые возможности для проявления ума, но они могут приводить к сбивающим с толку интерфейсам. Если у вас нет очень хорошей причины для того, чтобы заставлять объект вести себя так, как если бы он был встроенным типом данных, избегайте создания собственных связываний. К тому же, `tie` намного медленнее, чем использование встроенных типов данных, ввиду различных причин, кроющихся в реализации.

Хорошие причины включают облегчение отладки (используйте логгируемый скаляр, чтобы понять, где изменяется значение) и необходимость сделать некоторые невозможные операции возможными (доступ к большим файлам способом с эффективным использованием памяти). Связанные переменнные менее полезны как первичные интерфейсы к объектам; зачастую слишком трудно и ограничивающе пытаться вместить весь ваш интерфейс в тот, что поддерживается `tie()`.

Последнее слово предупреждения одновременно печально и убедительно; слишком много кода сбивается со своего пути, чтобы *предотвратить* использование связанных переменных, зачастую по случайности. Это неудачно, но нарушение ожиданий библиотечного кода склонно к раскрытию багов, исправить которые зачастую не в ваших силах.

14. Что упущено

Perl 5 не совершенен, но податлив — частично из-за того, что нет единой конфигурации, идеальной для каждого программиста и каждой цели. Некоторые полезные поведения доступны как базовые библиотеки. Ещё больше их доступно в CPAN. Ваша эффективность как Perl-программиста зависит от вашего использования этих улучшений.

Недостающие умолчания

Дизайн-процесс Perl 5 пытается предугадать новые направляения для языка, но в 1994 году невозможно было предсказать будущее таким, каким оно будет в 2011 году. Perl 5 расширил язык, но остался совместим с Perl 1 из 1987 года.

Лучший код на Perl 5 в 2011 году очень сильно отличается от лучшего кода на Perl 5 в 1994 году, или лучшего кода на Perl 1 в 1987 году.

Хотя Perl 5 содержит огромную базовую библиотеку, она не всеобъемлюща. Многие из лучших модулей Perl 5 существуют в CPAN , а не в ядре. Метадистрибутив `Task::Kensho` включает несколько других дистрибутивов, которые представляют собой лучшее из того, что может предложить CPAN. Если вам нужно решить задачу, загляните сначала туда.

С учётом вышесказанного, несколько базовых прагм и модулей незаменимы для серьёзных Perl-программистов.

Прагма strict

Прагма `strict` позволяет вам запретить (или вновь разрешить) разные мощные языковые конструкции, создающие потенциальную возможность непреднамеренных ошибок.

`strict` запрещает символические ссылки, требует объявления переменных и запрещает использование необъявленных голых слов . Тогда как редкое

использование символических ссылок необходимо для манипулирования таблицами символов , использование переменной как имени переменной создаёт вероятность неочевидных ошибок действия на расстоянии — или, хуже, возможность для плоховалидированного пользовательского ввода манипулировать внутренними данными со злонамеренными целями.

Требование объявлений переменных помогает определить опечатки в именах переменных и поощряет использование соответствующих областей видимости лексических переменных. Намного проще видеть намеренную область видимости лексической переменной, если все переменные имеют объявления `my` или `our` в соответствующей области видимости.

`strict` имеет лексическое действие, основанное на области видимости времени компиляции её использования и отключения (с помощью `no`). Смотрите `perldoc strict` для больших подробностей.

Прагма warnings

Прагма `warnings` контролирует сообщения о разных классах предупреждений в Perl 5, таких как попытка преобразовать в строку значение `undef` или использование со значениями неверного типа оператора. Она также предупреждает об использовании нерекомендуемых возможностей.

Наиболее полезные предупреждения объясняют, что Perl имеет сложности с пониманием того, что вы имели ввиду, и вынужден угадывать соответствующую интерпретацию. Даже несмотря на то, что Perl зачастую угадывает корректно, устранение неоднозначности с вашей стороны приведёт к уверенности, что программа работает корректно.

Прагма `warnings` имеет лексическое действие в области видимости времени компиляции её использования или отключения (с помощью `no`). Смотрите `perldoc perllexwarn` и `perldoc warnings` для больших подробностей.

`IO::File` и `IO::Handle`

До Perl 5.14 лексические дескрипторы файлов были объектами класса `IO::Handle`, но вам требовалось явно загрузить `IO::Handle`, прежде чем вы могли вызывать на них методы. С Perl 5.14 лексические дескрипторы файлов — экземпляры `IO::File`, и Perl сам загружает `IO::File` за вас.

Добавьте `IO::Handle` в код, запускающийся на Perl 5.12 или раньше, если вы вызываете методы на лексических дескрипторах файлов.

Прагма autodie

Perl 5 оставляет обработку ошибок (или игнорирование ошибок) вам. Если вы не достаточно внимательны, чтобы проверять, например, возвращаемое значение каждого вызова `open()`, вы можете попытаться прочитать из закрытого дескриптора файла — или, хуже, потерять данные при попытке записать в него. Прагма `autodie` изменяет поведение по умолчанию. Если вы напишете:

```
use autodie;

open my $fh, '>', $file;
```

...неудачный вызов `open()` выбросит исключение. Учитывая, что наиболее подходящий подход для неудавшегося системного вызова — выбросить исключение, эта прагма может удалить много шаблонного кода и подарить вам душевное спокойствие от знания того, что вы не забыли проверить возвращаемое значение.

Эта прагма появилась в ядре Perl 5 с Perl 5.10.1. Смотрите `perldoc autodie` для большей информации.