

[Главная](#) [Авторы](#) [Книги](#) [Жанры](#)

РЕКЛАМА

Станьте продвинутым JavaScript разработчиком →

РЕКЛАМА

Обменяйте очки опыта на бесплатный доступ к профессии! →

РЕКЛАМА

Бесплатный курс по «HTML/CSS/JavaScript» от Яндекса →

РЕКЛАМА

Создайте базу данных в облаке. Получите 3 000 Р на тест! →

 **Библиотека**  
электронной литературы в формате fb2

# Язык программирования Perl

Шохирев Михаил Васильевич



Курс знакомит с языком программирования Perl, с его принципами, основными возможностями и особенностями в объёме, достаточном, чтобы начать разрабатывать прикладные и системные задачи, включая программирование для сети Интернет.

Курс является достаточно подробным введением в язык программирования Perl. Описывается уникальная культура Perl и особенности, отличающие его от других языков программирования и во многом обусловившие его популярность. Рассматриваются основные средства программирования на языке Perl версии 5.8. Разбираются богатые возможности языка для создания самых разных приложений, а также особый стиль программирования на Perl. Курс ориентирован на студентов, начинающих программистов или разработчиков, применяющих другие языки и желающих писать прикладные или системные программы на Perl.

## Лекция 1. История развития Perl

В этой лекции излагается история развития языка программирования Perl, на особенности которого сильно повлияла личность создателя языка - Ларри Уолла. Объясняется лингвистическая основа языка Perl и его тесная связь с философией Unix. Рассказывается об оригинальной культуре Perl, объединяющей сообщество Perl-программистов. Далее говорится о сферах применения Perl и рассказывается о разработке новой, 6-й версии языка Perl. Также даются краткие сведения об установке Perl под разными операционными системами.

Цель лекции: познакомиться с историей создания языка Perl, узнать принципы, лежащие в его основе, и источники его особенностей. Кроме того, получить сведения, необходимые для установки системы программирования Perl.

**ИНТЕРНЕТ УНИВЕРСИТЕТ**  
ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ 

Язык программирования Perl создал американский программист Ларри Уолл (Larry Wall) в 1987 году, когда он работал системным программистом в компании Unisys. Цели, которые преследовал Ларри при разработке нового языка программирования, отражены в его названии - PERL, которое позднее стало расшифровываться как Practical Extraction and Report Language, то есть

"практический язык извлечения "данных" и "создания" отчетов". (Нетрудно заметить, что первые буквы всех слов названия составляют слово PEARL - "жемчуг". Одна из легенд о происхождении Perl гласит, что в то время уже существовал язык с таким названием, поэтому Ларри сократил название на одну букву, не изменив произношения. По иронии судьбы, сегодня тот язык не помнит никто, кроме историков, а Perl известен всему миру. Хотя, по другой версии, Ларри назвал созданный им язык по имени своей невесты.



Рис. 1.1. Создатель языка Perl - Ларри Уолл

После того как 18 декабря 1987 года была выпущена 1-я версия языка Perl, он быстро распространился среди пользователей сети Usenet. Несмотря на то, что в операционной системе (ОС) Unix, для которой был создан Perl, уже имелись многочисленные и разнообразные средства для обработки текстовой информации (awk, csh, grep, sed и другие), новый язык полюбился огромному числу системных администраторов и программистов. Он был легок в изучении и применении: синтаксис похож на C, Perl-программы не требовалось предварительно компилировать, исходные тексты было легко модифицировать. А самое главное - это был действительно очень практичный язык: с его помощью легко решалось большинство повседневных задач - от самых простых до очень сложных. Активно пользуясь языком Perl, программисты из разных стран направляли Ларри Уоллу предложения добавить в него новые возможности или улучшить имеющиеся. Постепенно Perl превратился из средства обработки текстов в среде Unix в мощную универсальную систему программирования. В середине 1990-х годов после победоносного распространения WWW (World Wide Web) Perl стал излюбленным инструментом web-мастеров для создания динамических сайтов и Internet-программирования. Благодаря своим мощным встроенным средствам работы с текстовыми данными Perl широко используется для обработки информации в форматах HTML и XML. Кроме того, Perl стал идеальным языком для быстрого создания прототипов сложных приложений, которые затем нетрудно превратить в реально действующие.

О его успехе и широком признании говорит тот факт, что Perl стал непременным компонентом любой поставки ОС семейства Unix (например, FreeBSD, Linux или Solaris). Кроме этого, к настоящему времени Perl реализован практически для всех современных аппаратных платформ (суперкомпьютеры, RISC, Macintosh, PC, наладонные компьютеры и т. д.) и операционных систем (AIX, Linux, MacOS, MS-DOS, NetWare, OS/2, QNX, Solaris, Windows, VMS - всех не перечислить!). Это дает возможность легко переносить популярные Perl-программы из одного операционного окружения в другое. (К слову сказать, примеры к этому курсу лекций проверялись под операционными системами SuSE Linux 10.0, MS Windows XP Professional SP2, и MS Pocket PC 2003 Premium Edition v. 4.20). Несомненно, его широкому применению способствовало и то, что он распространяется бесплатно на условиях одной из лицензий: либо GNU General Public License (GPL), либо Artistic License - на выбор. Но главное, что в нем есть все средства, чтобы отлично выручать профессионалов и неопытных программистов, когда требуется быстро решать разные системные и прикладные задачи - от самых простых до весьма сложных. В результате Perl стал одним из самых успешных проектов движения open source (с открытыми исходными кодами) - наряду с Apache, Linux, PHP и Python.

Но прежде всего популярность Perl связана с тем, что Ларри Уолл создал действительно необычный язык: принципы его разработки сильно отличаются от применявшихся в программировании до этого. Новаторский характер Perl связан с уникальными особенностями личности автора языка, Ларри Уолла, и его разносторонними интересами.

Во время обучения в университете Ларри Уолл получил не только компьютерное, но и лингвистическое (а также химическое и музыкальное) образование, и это, несомненно, сильно повлияло на особенности языка Perl. (Воспитанный в религиозной семье, молодой Ларри даже собирался стать миссионером и посвятить свою жизнь обращению в христианство туземцев, при необходимости создавая для туземных языков письменность, чтобы перевести на них Библию!) Рассказывая об истоках Perl, Ларри Уолл приводил схему, воспроизведенную на рис. 1.2, на которой показано, что Perl появился в результате слияния нескольких, на первый взгляд, несовместимых идей и дисциплин.

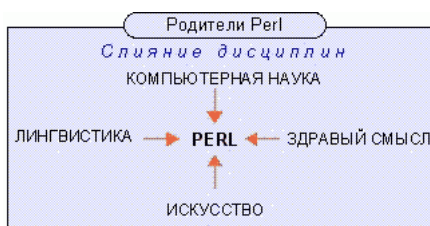


Рис. 1.2. Факторы, повлиявшие на создание Perl

Возможно, главная заслуга Ларри как автора Perl заключается в том, что ему удалось создать язык, учитывающий свойства и принципы естественного языка: ситуативную многозначность конструкций, разумную избыточность (и за счет этого стилизов

разнообразие), максимальную адаптированность к различному уровню знаний носителей языка, гибкость и выразительность синтаксических средств. Все это придает особый творческий вкус самому процессу "сочинительства" на этом языке. Пишущие о языке Perl невольно употребляют термины "идиома", "метафора", "синоним", "контекст" и т.п., обычно применяемые к естественным языкам. А сам Ларри Уолл, описывая язык Perl, широко пользуется лингвистическими терминами: существительное, глагол, предложение, единственное и множественное число, местоимение. Выдающийся лингвист Б.Л. Уорф заметил: "Язык формирует наш способ мыслить и определяет, о чем мы можем думать". Эту же мысль подтверждает Дж. Бентли в книге "Жемчужины творчества программистов", описывая случай, когда именно особенности языка программирования APL натолкнули разработчика на оригинальное решение задачи с массивами. В этом смысле Perl как язык для воплощения замыслов программиста, не сковывает фантазию разработчика, дает ему максимальную свободу самовыражения, а иногда даже подсказывает нестандартные решения. При разработке языка Perl были использованы многие лингвистические принципы. Перечислим наиболее важные из них.

Важное значение имеет принцип повторного использования. Человеческие языки тяготеют к использованию ограниченного набора конструкций для выражения разных значений и их повторному применению в различных контекстах. В соответствии с этим принципом, в Perl одни и те же конструкции языка имеют различный смысл, а их однозначное понимание определяется контекстом использования.

Принцип контекста тесно связан с предыдущим. Контекст используется в естественных языках для правильной интерпретации смысла выражения. Подчиняясь этому принципу, в языке Perl, например, многие функции возвращают одно значение или список в зависимости от контекста вызова: требует ли синтаксис выражения единичного или множественного значения.

Принцип смыслового подчеркивания в естественных языках служит для привлечения внимания к важной части высказывания. При этом выделяемая часть обычно ставится в начало предложения. В Perl программист может воспользоваться этим принципом для акцентирования смысла, выбрав простое предложение с модификатором или условную конструкцию, например:

```
$we->shall(do($it)) if $you->want($it); # или
```

```
if $you->want($it) { $we->shall(do($it)) }
```

Принцип свободы в естественных языках предполагает, что всегда есть несколько вариантов выражения одной и той же мысли. Как и в естественных языках, в Perl одного и того же результата можно достичь несколькими способами, используя различные выразительные средства языка. Это так называемый принцип TIMTOWTDI - сокращение читается "Тим Тоуди" и расшифровывается как "There is more than one way to do it": "есть более одного способа сделать что-то". В соответствии с ним каждый разработчик выбирает тот вариант языковой конструкции, который больше подходит ему в конкретной ситуации, больше соответствует его стилю или уровню знания языка.

Подобный подход полностью противоположен строго формальному взгляду на языки программирования, когда главными целями разработки становятся однозначность и минимизация языковых конструкций. Но с точки зрения психологии и языкового мышления "лингвистический подход" во многом более продуктивен, как это ни парадоксально! Ведь он стимулирует творчество, привлекая человеческий опыт применения естественного языка, - и писать программы на Perl становится увлекательно и интересно. Наверное, поэтому программирующие на Perl так любят словечко "fun" (весело, забавно).

Ларри не раз повторял, что Perl задумывался как язык, предполагающий постоянное развитие. Поэтому в процессе его совершенствования, как и при развитии "человеческих" языков, из других языков программирования было позаимствовано все лучшее, практичное и удобное для разработчика. На становление Perl повлияли языки Ada, Basic, Fortran, Lisp, Pascal и, конечно, язык C. Perl впитал в себя разные течения компьютерной науки: процедурное, модульное, функциональное и объектно-ориентированное программирование, макрообработку текста, а кроме этого - культуру ОС Unix, с ее богатым набором команд и утилит, стройной продуманной архитектурой и унифицированным подходом к представлению данных.

Решающее влияние на развитие языка Perl оказала среда Unix, в которой он разрабатывался. С самого начала эта операционная система создавалась небольшой группой программистов для самих себя. Поэтому принципы, заложенные в ней, ориентированы на удобство повседневного применения человеком: все делалось максимально функционально, кратко, единообразно. Например, большинство системных настроек хранится в обычном текстовом виде, так что их может читать и исправлять человек, вооруженный простым текстовым редактором. (Подумать только, что это было сделано во времена, когда памяти было так мало, что старались экономить даже биты данных!)

Для тех, кто знаком с операционной системой Unix, этот материал не будет новым. Но многим программистам, работающим с MS-DOS и Windows, важно познакомиться с принципами программирования, основанного на инструментальных средствах (software tools), которое зародилось и развивалось вместе с ОС Unix. О них написали замечательную книгу известные программисты Брайан Керниган (Brian W. Kernighan) и Филип Пладжер (Philip Plauger). Вот основные положения этого подхода.

1 Для решения некоторой задачи разрабатываются программы небольшого размера, каждая из которых выполняет одну функцию решаемой задачи.

2 Поставленная задача решается путем взаимодействия небольших программ за счет последовательной обработки данных каждой из

них.

3 При разработке этих небольших программ следует ориентироваться на их максимально независимое использование, чтобы их можно было применять для решения других задач. Таким образом, постепенно создаются инструментальные средства (ИС) для дальнейшего универсального применения.

4 Большинство инструментальных средств разрабатываются в виде программ, представляющих из себя фильтры, которые читают поток данных из стандартного ввода и записывают обработанные данные в стандартный вывод.

5 Объединение инструментальных средств в приложения производится средствами командного языка ОС: перенаправление ввода-вывода, создание программных конвейеров (направление выходного потока одной программы на вход другой).

6 Основным форматом хранимых данных для инструментальных средств выступают текстовые файлы, одинаково хорошо подходящие для программной обработки и чтения их человеком.

7 Для эффективной обработки слабо структурированной текстовой информации применяются регулярные выражения (средства поиска по шаблонам, о которых речь пойдет в лекции 8).

Эти несложные правила определяют особенную технологию разработки программ, при которой создаются и накапливаются гибкие инструментальные средства, легко настраиваемые и удобно комбинируемые, с помощью которых можно решать широкий круг задач. При этом отдельные инструментальные средства часто объединяются в неожиданные комбинации, изначально не предусмотренные авторами при их разработке. Это действительно очень продуктивный подход, давно и широко применяемый среди программирующих в среде ОС Unix. Вместо продолжительной разработки и отладки одной сложной многофункциональной программы, максимально используются готовые проверенные и надежные компоненты - инструментальные средства. При необходимости сравнительно быстро создаются несколько недостающих инструментальных средств (нестандартный диалоговый ввод, необычно форматированный отчет), также зачастую пригодных для дальнейшего применения. Perl развил идею инструментальных средств, функциональность многих Unix-утилит была реализована в конструкциях языка, а сам Perl стал идеальным средством для "склеивания" нескольких приложений в большие системы. Воспитанный на идеях Unix, Ларри Уолл сказал про свое детище: "Perl - это переносимая концентрация культуры Unix, оазис среди пустыни..."

В языке Perl к идеям Unix добавились достижения других языков программирования, и все это было переработано в соответствии с лингвистическими принципами и требованиями практического применения языка. Положения философии языка Perl были изложены Ларри Уоллом в его знаменитой книге "Программирование на Perl" и во множестве статей и интервью. Они часто заключены в форму изящных и остроумных афоризмов. Один из них гласит: "Perl разработан так, чтобы легко решать простые задачи, а трудные делать возможными". А эмблемой Perl стал верблюд - домашнее животное, не блистающее красотой и изяществом, но обладающее уникальными способностями для выполнения трудной, кажущейся невозможной, работы в экстремальных условиях.

Perl создавался как практичный язык, а какая же практичность без скорости? Программы на Perl традиционно отличаются высокой скоростью выполнения и по быстродействию сравнимы с откомпилированными Unix-утилитами. Быстрота работы достигается за счет того, что исполняющая система Perl читает исходный текст Perl-программы и компилирует его "на лету", затем сразу интерпретирует откомпилированную программу. Говоря об этой уникальной исполняющей системе, сочетающей в себе компилятор и интерпретатор, принято называть ее perl (строчными буквами, в отличие от языка программирования Perl).

Ларри Уолл в шутку (в которой, как водится, есть немалая доля истины) провозгласил три главных добродетели программиста: лень, нетерпение, самомнение (laziness, impatience, hubris). И Perl спроектирован так, чтобы соответствовать этим качествам разработчика. "Ленивый" программист, часто ограниченный во времени, может написать программу максимально компактно и быстро, поскольку в Perl есть множество способов кратко и просто записать довольно сложные алгоритмы. К тому же любые символы не являются обязательными, если их отсутствие не меняет смысла программы. Например, обычную условную конструкцию

```
if ($x > 0) {  
  
    print($y / $x);  
  
}
```

можно написать и по-другому - без пробелов и скобок, переместив условие в конец оператора:

```
print$y/$x if$x>0;
```

Часто внешние обстоятельства побуждают нас к быстрейшему достижению цели. Но при программировании на Perl терпение программиста подвергается минимальному испытанию, поскольку Perl-программа запускается без предварительной компиляции и выполняется очень быстро. И наконец, система программирования Perl предлагает необычайно широкий набор средств, чтобы реализовать самые амбициозные проекты любого программиста, даже с гипертрофированным самомнением.

Гуманные принципы, заложенные в язык Perl, нацелены на создание комфортной обстановки при разработке программ как для новичка, так и для опытного программиста. Одним из них стал принцип DWIM (Do What I Mean - "делай то, что я имею в виду"), в соответствии с которым в большинстве случаев исполняющая система Perl обеспечивает интуитивные ожидания автора программы без дополнительных уточнений с его стороны.

В языке Perl учтены многие психологические особенности программистов, в том числе даже программистская склонность "к экономии усилий". Например, более часто употребляемые конструкции языка записываются кратко, а редко используемые - длиннее. В этом заключается так называемый принцип "кодирования Хаффмана", название которого восходит к широко используемому методу сжатия данных битовыми последовательностями переменной длины, предложенному Дэвидом Хаффманом (D.A. Huffman).

Принцип "отсутствия встроенных ограничений", которому при всякой возможности следуют разработчики Perl, предполагает максимум свободы для программиста, например: длина имени переменной может достигать 252, в строках могут храниться любые двоичные данные, строка может занимать всю доступную память, объем используемой памяти ограничивается только ОС.

В следующих лекциях будут затронуты и другие принципы и положения философии Perl, а также проиллюстрированы уже упомянутые.

К настоящему времени в мире образовалось многочисленное международное сообщество программистов, пишущих на Perl (шутливо называемое Perlfolk - "народ Perl"). По всему миру созданы группы пользователей языка Perl. Некоторые (но далеко не все) из них зарегистрированы в списке на сайте . Развитию, продвижению и распространению языка Perl способствовало немало выдающихся программистов, среди них: Тим Банс (Tim Bunce), Грэхем Барр (Graham Barr), Малькольм Битти (Malcolm Beattie), Хуго Ван Дер Занден (Hugo van der Sanden), Илья Захаревич (Ilya Zakharevich), Ник Инг-Симмонс (Nick Ing-Simmons), Гурусами Сарати (Gurusamy Sarathy), Линкольн Штейн (Lincoln Stein) и многие другие, а также тысячи поклонников этого языка во всем мире. Большую поддержку языку Perl и всему движению программно обеспечения с открытыми исходниками (open source software) оказывает известный книжный издатель Тим О'Рейлли (Tim O'Reilly).

Энтузиастами из разных стран написано огромное число модулей, расширяющих возможности Perl, которые распространяются через "Всеобъемлющую сеть Perl-архивов" CPAN (Comprehensive Perl Archive Network). Это грандиозное собрание модулей начал Андреас Кениг (Andreas Kenig), а сейчас координатором проекта выступает Яркко Хьетаниemi (Jarkko Hietaniemi). Модули в архиве CPAN хорошо документированы, снабжены проверочными тестами, их легко устанавливать и удобно использовать. Авторы модулей постоянно совершенствуют их, а в тестировании принимают участие тысячи добровольцев, так что качество исполнения весьма высокое. На сайте CPAN и его зеркалах хранится громадное количество модулей для решения самых разных задач: от работы с биологическими данными и музыкой в формате MP3 до утилит шифрования и интерфейсов к самым экзотическим базам данных. Самые популярные модули, такие как CGI.pm, включены в стандартную поставку системы программирования Perl.

Для языка Perl написана очень полная и подробная документация. Ее можно изучить через web-интерфейс на сайте . Она также поставляется вместе со всеми версиями Perl и доступна в диалоговом режиме через утилиту просмотра документации perldoc, входящую в состав дистрибутива Perl. Например, чтобы запросить оглавление документации по Perl, нужно выполнить команду

```
perldoc perl
```

О языке Perl написано много совершенно замечательных книг, среди них нужно особо выделить классические книги издательства O'Reilly с изображениями животных на обложке. Пожалуй, вот самые знаменитые книги из этой серии:

[x]. "Книга с ламой" ("the llama Book"), в русском переводе "Изучаем Perl" - отличный учебник для начального знакомства с языком, написанный известным "Perl-проповедником" Рэндалом Шварцем (Randal L. Schwartz).

[x]. "Книга с верблюдом" ("the Camel Book"), в переводе на русский "Программирование на языке Perl" - подробное и всестороннее описание Perl и его философии, изложенное автором языка Ларри Уоллом.

[x]. "Книга с бараном" ("the Ram Book"), "Perl. Сборник рецептов" - внушительный сборник готовых решений на Perl для множества типичных задач, составленный разработчиками языка Томом Кристиансеном (Tom Christiansen) и Натаном Торкингтоном (Nathan Torkington).

Программистов, пишущих на Perl, объединяет не только любовь к этому языку, но и особый азартный и озорной дух сообщества разработчиков, которые умеют не только отлично работать, но и весело отдыхать. Веселый тон сообществу задает все тот же Ларри Уолл, известный шутник и балагур, который в официальной документации дает такое озорное определение языка: "На самом деле Perl обозначает Паталогически Эклектичный Распечатыватель Чепухи, но только не говорите никому, что это сказал я". И сам язык Perl тоже часто используется для развлечения. Например, существует состязание по созданию "стихов на Perl" - компилируемых Perl-программ, читаемых как осмысленное стихотворение на английском языке. Регулярно проводится Obfuscated Perl Contest - конкурс на самую туманную программу на Perl с использованием наиболее неудобочитаемых и запутанных конструкций языка. Есть также соревнование по написанию так называемых "однострочников" (one-liners) - полноценных и полезных программ на Perl, которые состоят из одной строки, обычно вводимой в качестве параметра при запуске

интерпретатора Perl. Познакомиться с некоторыми полезными однострочниками можно, прочитав серию статей Бена Окопника (Ben Ockopnik) для "Linux-газеты", выдержанных в детективном духе, в блестящем переводе Павла Соколова на сайте .

Система программирования Perl стабильно развивается в течение многих лет: в 1988 году вышла версия 2.0, в 1989 году выпущена версия 3.0, в 1991 году - очередная версия 4.0. В 1994 году появилась версия 5.0, при подготовке которой язык был почти полностью переписан, в него были добавлены модули и возможности объектно-ориентированного программирования. На момент написания этого учебника последней рабочей версией Perl была 5.8.7. Сейчас основные работы по развитию языка выполняет группа примерно из 200 добровольных разработчиков из разных стран, называемых perl-porters. Текущий координатор Perl-проекта, ответственный за его развитие и сопровождение, шуточно называется rumpking (от rumpkin holder - "держатель тыквы"). Информация о состоянии разработки Perl доступна на сайте . А руководит разработкой языка по-прежнему Ларри Уолл, носящий звание архитектора языка (language designer). Ежегодно на конференции Perl-разработчиков Ларри Уолл выступает с традиционной речью "The State of the Onion" ("Состояние луковицы"), в которой всегда образно и оригинально, с присущим ему юмором говорит о месте Perl среди современных технологий, о его состоянии и развитии.

В 2000 году на 4-й Конференции Perl-программистов по инициативе Джона Оруанта (Jon Orwant), редактора "The Perl Journal", было принято решение о разработке Perl 6 - полностью переработанной версии языка. Затем прошло широкое обсуждение многочисленных предложений пользователей Perl по его улучшению и развитию. Ларри Уолл подвел итоги этого обсуждения в серии посланий разработчикам (так называемых "апокалипсисов"), в которых были изложены основные проектные решения. После этого началась работа над новой версией. В ходе разработки язык полностью перепроектирован и переписывается "с нуля", в него добавлено множество новшеств и дополнений. Но при этом планируется обеспечить максимальную совместимость с большинством существующих Perl-программ. К разработке привлекаются многие программисты из сообщества Perl. Ларри Уолл говорит: "Perl 5 появился как результат переделки языка Perl мной. Я хочу, чтобы Perl 6 стал переделкой Perl со стороны сообщества и переделкой самого сообщества". Параллельно с разработкой 6-й версии языка создана и совершенствуется виртуальная машина Parrot, в среде которой выполняется Perl 6, но которая может использоваться для исполнения и других динамических языков программирования. Во главе разработки стоят известные Perl-программисты: главным разработчиком "внутренностей" нового языка является Дэн Сугальски (Dan Sugalski), Чип Зальценберг (Chip Salzenberg) - основной проектировщик Parrot, а менеджером проекта стала Эллисон Рэндал (Allison Randal). Важным промежуточным этапом при переходе на Perl 6 станет проект Ponie - версия Perl 5, исполняемая на Parrot.

Пользователи одной из ОС семейства Unix, скорее всего, будут пользоваться уже установленной системой программирования Perl, поставляемой с ОС. Но если Perl не установлен, то исходные тексты самой свежей версии всегда можно загрузить с сайта CPAN и скомпилировать, руководствуясь поставляемой подробной документацией. Готовые откомпилированные дистрибутивы Perl для самых разных операционных систем перечислены на сайте CPAN (). Все, что нужно для установки Perl на компьютеры Apple, можно найти на сайте . Для пользователей MS Windows можно порекомендовать удобный дистрибутив от компании ActiveState () или комплект RXPPerl (pixigreg.com/?rxperl), включающий в себя множество полезных библиотек, а также рабочие реализации Perl 6 и Parrot для тех, кто хочет ознакомиться с возможностями новой версии. Тем, кто намерен заниматься разработкой на Perl приложений для Интернета, можно посоветовать дистрибутив Perl в комплекте с сервером Apache и большинством необходимых модулей. Его можно загрузить с сайта perl.apache.org.

Установка Perl из бинарного дистрибутива обычно не представляет трудностей и сводится к распаковке файлов из загруженного архива. Для удобства работы имя каталога с исполняемыми файлами perl добавляется в системный список путей для поиска исполняемых программ (в переменную окружения PATH). Обычно все необходимые действия по установке выполняет программа-инсталлятор. После завершения установки нужно проверить доступность исполняющей системы perl. Это можно сделать, выполнив такую команду (в терминальном окне Unix, в командном окне Windows или в окне интерпретатора cmd на Pocket PC):

```
perl -v
```

На экране должны появиться сообщения, описывающие версию языка и авторские права:

```
This is perl, v5.8.7 built for MSWin32-x86-multi-thread
```

```
Copyright 1987-2005, Larry Wall
```

```
Perl may be copied only under the terms of either the Artistic License or the
```

```
GNU General Public License, which may be found in the Perl 5 source kit.
```

```
Complete documentation for Perl, including FAQ lists, should be found on
```

```
this system using 'man perl' or 'perldoc perl'. If you have access to the
```

```
Internet, point your browser at http://www.perl.org/, the Perl Home Page.
```

Если вы увидели подобное сообщение, значит, Perl корректно установлен и готов к использованию, так что можно уже

приступить к его изучению. Чем мы и займемся, начиная со следующей лекции.

## Лекция 2. Литералы и скалярные данные

В этой лекции рассматриваются элементарные конструкции языка Perl: литералы и скалярные данные. Описываются форматы записи чисел, строк и правила именования скалярных переменных. Вводится понятие контекста. Приводится формат записи комментариев в программе. Даются начальные сведения о документировании программ с применением формата POD.

Цель лекции: освоить правила записи элементарных элементов языка, литералов и скаляров, необходимые для правильного оформления программ на языке Perl. Научиться комментировать программы с использованием однострочных комментариев и формата встроенной документации POD.

Perl - очень практичный язык, и изучить его основы довольно просто. Поскольку большинство синтаксических конструкций Perl основаны на языке C, то для программистов, знающих языки C, C++, C#, Java, JavaScript, Python или PHP, синтаксис Perl будет очень знакомым. Но и тот, кто раньше писал на языке Pascal, Fortran или Basic, легко привыкнет к нотации Perl. Нетрудно будет и тем, кто не знает ни одного языка программирования, поскольку Perl спроектирован так, чтобы новичок смог научиться писать на небольшом подмножестве языка, а затем постепенно углублялся в его тонкости. Знакомство с языком Perl мы начнем с правил записи литералов - непосредственного представления в программе значений данных таких как числа и строки. Иногда литералы неправильно называют константами, под которыми в программировании чаще имеют в виду имена, представляющие неизменяемые данные.

Редкая компьютерная программа обходится без использования числовых литералов. В программе на Perl числа записываются самым естественным образом. Так, например, выглядят десятичные целые числа, со знаками и без:

```
12 -34 +56
```

Столь же привычно выглядят и десятичные дробные числа, положительные и отрицательные. Целая часть по англоязычной традиции отделяется от дробной части десятичной точкой. Целая или дробная часть числа может не записываться, если она равна нулю:

```
.12 34. -456.78 +9.0
```

Для удобства чтения исходной программы человеком большие числа могут записываться с символом подчеркивания "\_" в качестве разделителя разрядов:

```
123_456 -7_890_098 1_000_000_000_000
```

Очень маленькие или очень большие числовые значения, целые или дробные, удобно записывать в экспоненциальной форме (также называемой "научной" нотацией):

```
123E-4 -56e+7 8e9
```

Латинская буква "E" (заглавная или строчная) в подобных литералах читается как "умноженное на 10 в степени", то есть соответствует арифметическому выражению  $123 \cdot 10^{-4}$ . Знак "+" у основания и степени числа необязателен.

Иногда требуется записывать числа не в десятичной, а в других системах счисления. Для записи шестнадцатеричных чисел применяется префикс 0x. В этой системе счисления каждая цифра представляет 4 бита данных, а буквами от A до F (независимо от их регистра) обозначаются дополнительные "цифры" от 10 до 15. Так записываются в шестнадцатеричном виде числа 13, -10, 53392, и 1024:

```
0x0d -0x0A 0xD090 0x400
```

В некоторых случаях (например, при записи атрибутов файла в Unix) нагляднее изобразить числа в восьмеричной системе счисления. Обратите внимание, что восьмеричные числа записываются с ведущим нулем, а каждая цифра из диапазона от 0 до 7 представляет 3 бита данных, так что все числовые литералы из одних цифр с ведущим нулем рассматриваются как восьмеричные числа. Вот как будут выглядеть в восьмеричном виде числа 292, -438, 511, и 1024:

```
0444 -0666 0777 02000
```

Когда нужно представить двоичные числа, то перед ними ставится признак двоичной системы счисления 0b (каждая цифра 0 или 1 представляет 1 бит). Вот числа 17, -85, 238 и 1024, записанные как двоичные литералы:

```
0b00010001 -0b01010101 0b1110_1110 0b10000000000
```

Что касается внутреннего представления чисел в Perl, то они всегда хранятся в виде чисел с плавающей точкой двойной точности, что гарантирует максимальную точность вычислений. При необходимости предусмотрена возможность переключиться на целочисленную арифметику. Удобно и то, что при выводе числовые литералы, записанные в любой системе счисления, автоматически преобразуются к удобочитаемому десятичному виду.

В языке Perl нет специального обозначения для отдельных символов, в нем есть только символьные строки, которые иногда могут состоять из одного символа. Строковые литералы заключаются либо в двойные кавычки, либо в апострофы, называемые также одинарными кавычками, например:

```
"Это строка." "А" 'это другая строка' '.'
```

Иногда в строковых литералах требуется представить специальный символ (управляющий символ или символ, отсутствующий на клавиатуре). Для этого используется так называемая escape-последовательность (называемая также управляющей последовательностью) - это символ "\" (backslash, обратная косая черта), за которым следует один или несколько символов. Все знаки управляющей последовательности представляют один символ в строковом литерале. Например:

```
\a звонок (Alert, bell) или 0x07 в 16-теричном представлении
```

```
\b возврат на шаг (Backspace) или 0x08
```

```
\e символ "эскейп" (Escape) или 0x1B
```

```
\f прогон страницы (Form feed) или 0x0C
```

```
\n новая строка (Newline) или 0x0A
```

```
\r возврат каретки (Return) или 0x0D
```

```
\t табуляция (Tabulation) или 0x09
```

```
\033 восьмеричный код символа (например, 033)
```

```
\x1b шестнадцатеричный код символа (например, 1B)
```

```
\Cc управляющая последовательность (например, Control+C)
```

```
\x{263A} двухбайтный символ набора Unicode (например, ?)
```

```
\N{sigma} именованный символ набора Unicode (например, ?)
```

```
\" символ двойной кавычки (quote)
```

```
\' символ одинарного апострофа (apostrophe)
```

```
\\ символ обратной черты (backslash)
```

```
\$ любой другой символ, как он есть (например, знак доллара)
```

В литеральных строках, заключенных в двойные кавычки, выполняется замена каждой escape-последовательности на соответствующее значение специального символа. Такая подстановка называется интерполяцией, например:

```
"символ перевода на новую строку:\n"
```

```
"слова\t,разделенные\t табуляцией"
```

```
"вставка \"кавычек\" в литерал, заключенный в кавычки"
```

Если интерполяция управляющих последовательностей не требуется, то строковый литерал нужно заключить в одинарные апострофы:

```
'обратная косая с буквой n:\n'
```

```
'здесь \t - это обратная косая и буква t'
```



```
'вставка \'апострофов\' в литерал, заключенный в апострофы'
```

В этом случае из escape-последовательностей только `\'` и `\\` заменяются на символы апострофа и обратной черты. А остальные последовательности, такие как `\n` или `\x00`, представляют обычные символы. Если необходимо вставить в строковый литерал апострофы, то строку заключают в двойные кавычки, и наоборот:

```
'книга "Изучаем Perl"' "книга 'Изучаем Perl'"
```

Строковые литералы, заключенные в одинарные апострофы или в двойные кавычки, могут располагаться в программе на нескольких строках, например:

```
'А это пример строкового литерала,  
  
расположенного в программе  
  
на нескольких строках'
```

Поскольку здесь сохраняются невидимые символы перехода на новую строку, многострочные литералы удобно использовать для записи текста, предназначенного для вывода на печать на нескольких строках. Строковые литералы могут не содержать ни одного символа. Это так называемые "пустые строки", которые записываются как два апострофа или две кавычки без пробела между ними (`''` или `""`).

Альтернативные способы записи строковых литералов будут рассмотрены в лекции 7, в которой излагаются возможности строковых данных и приемы работы со строками.

Как известно, переменные - это программные объекты для хранения во время выполнения программы данных об объектах реального мира. В Perl имеются две основные разновидности данных: строки и числа, называемые скалярными данными, то есть данными, представляющими единичное значение. К скалярам также относятся ссылки, которые будут рассмотрены в лекции 11.

Для хранения скалярных данных предназначены скалярные переменные, каждая из которых может содержать одно значение. Перед именем такой переменной ставится символ `$`, обозначающий скалярную величину (`$` - это стилизованное `s`, то есть `scalar`). Далее, в лекции 5, будет рассмотрен другой тип переменных - массивы, которые содержат множественные значения, логически связанные вместе. В массивах может одновременно храниться несколько скалярных значений, и имена массивов предваряются символом `@` (`@` - это стилизованное `a`, то есть `array`). Поначалу эти "забавные символы" (funny characters, шутливо называемые "окултными знаками" - sigils) перед именами переменных кажутся непривычными и даже лишними. Но позже вам раскроется глубокий смысл и удобство этих символов, официально называемых разыменовывающими префиксами. На первый взгляд очевидно, что они своим видом напоминают, какое значение содержит переменная - единичное или множественное. Скаляры представляют в Perl лингвистическое понятие единственного числа, а массивы - множественного числа. Вообще, переменные воплощают грамматическую идею существительных, в отличие от процедур или функций, исполняющих в программе роль глаголов.

Для именования пользовательских переменных в Perl применяются правила, обычно действующие и в других языках:

[x]. в имени допускается использовать латинские буквы, символы подчеркивания, которые приравниваются к буквам, и цифры;

[x]. имя переменной должно начинаться с буквы (длина имени переменной практически не ограничивается).

В Perl имена переменных принято записывать строчными буквами, при необходимости разделяя слова в имени символом подчеркивания, например:

```
$website  
  
$catch22  
  
$user_name  
  
$input_record_counter  
  
$this_is_an_example_of_a_very_long_variable_name
```

В большинстве процедурных языков каждую переменную в программе требуется объявлять, определяя, какой тип данных допустимо хранить в ней: например, `boolean`, `character`, `string`, `byte`, `short`, `integer`, `long`, `real`, `float` или `double`. В Perl можно вводить переменные в любом месте программы без объявления. Чтобы использовать переменную, надо просто упомянуть ее имя в программе, обычно это происходит при присваивании ей начального значения.

```
$background_color = 0xFF;
```

```
$version_number = 5.8;
```

```
$www_site = "www.perl.com";
```

```
$email_address = 'larry@wall.org';
```

Если значение переменной не присвоено, в ней хранится специальное неопределенное значение `undef`. Неопределенность значения переменной можно проверить с помощью встроенной функции `defined()`, возвращающей истинное значение, если значение переменной определено. Одна и та же переменная может поочередно иметь неопределенное значение или хранить значение любого из основных типов, например, строку или число (целое или дробное):

```
$variable;
```

```
$variable = 'Строка';
```

```
$variable = 25;
```

```
$variable = 3.141592653;
```

В именах переменных заглавные и строчные буквы различаются, поэтому приведенные ниже имена относятся к совершенно разным переменным:

```
$language $Language $Language $LaNgUaGe
```

Необъявленные переменные имеют глобальную область видимости в пределах пакета. Область видимости переменных определяется рамками программных единиц: блоков, подпрограмм, пакетов, о которых подробно будет рассказано в лекциях 12 и 13. Чтобы задать область видимости переменных, нужно их объявить явно. Для объявления переменных применяются такие ключевые слова:

`[x]. my` - переменные с лексической областью видимости;

`[x]. our` - глобальные переменные с лексической областью видимости;

`[x]. local` - временное скрывание глобальных переменных.

В одном объявлении можно перечислить несколько переменных. Объявляемым переменным рекомендуется сразу присваивать начальные значения, например:

```
local $_ = 25
```

```
my ($buffer = '', $count = 0, $end_of_file = 0)
```

```
our $version = 2.5, $author = 'Mike Shock'
```

Различия между этими типами переменных будут подробно рассмотрены в лекциях, посвященных подпрограммам и модулям. А пока, памятуя об общих правилах хорошего стиля программирования и требованиях к надежности программ, постараемся свести к минимуму использование глобальных переменных. Для этого достаточно взять за правило преимущественно использовать лексические переменные, объявляемые с помощью ключевого слова `my`. Применение переменных с лексической видимостью также сокращает расход памяти, поскольку они автоматически уничтожаются при выходе из области видимости.

Значения переменных, как и escape-последовательности, могут интерполироваться, если они помещены в строковый литерал, заключенный в двойные кавычки. Этим широко пользуются в программах на Perl для удобного формирования строк, в которые нужно подставить вычисленные значения переменных. Например, так:

```
"Прочитано $n строк"
```

```
"Используемая версия Perl = $"
```

```
"Письмо для $name отправлено по адресу $email"
```

Благодаря разыменовывающему префиксу `$`, переменные хорошо различимы в строковых литералах. При интерполяции в качестве имени переменной рассматривается максимальная последовательность символов, которая может быть идентификатором. Поэтому

нужно быть внимательным в случаях, когда после имени переменной в строке нет знаков препинания или пробелов. Например, при вставке в строку переменной `$delimiter` следующим образом:

```
"One$delimiterTwo"
```

будет подставлено значение несуществующей переменной `$delimiterTwo`. Чтобы явно отделить имя вставляемой переменной от последующих символов, нужно имя переменной после префикса заключить в фигурные скобки, вот таким образом:

```
"One${delimiter}Two"
```

Если в строковый литерал нужно включить символ доллара, не являющийся префиксом переменной, то можно заключить литерал в одинарные апострофы или "экранировать" символ доллара (отменить его специальное значение), поставив перед ним обратную косую черту:

```
'переменная $var не интерполируется'
```

```
"переменная \$var не интерполируется"
```

Весьма удобно, что преобразования между строками и числами выполняются автоматически в зависимости от контекста выражения, в котором они используются. В языке Perl для уточнения смысла языковых конструкций часто используется понятие контекста, под которым понимается программное окружение элемента языка (переменной, подпрограммы и так далее), определяющее его использование. Скалярные переменные, рассмотренные в этой лекции, используются в скалярном контексте (подразумеваемом использованием одного значения). А он, в свою очередь, может подразделяться на строковый и числовой контекст. Например, в переменную помещено число:

```
$year = 1987
```

При использовании ее в числовом контексте (например, в арифметическом выражении для сложения с другим числом) будет использовано числовое значение переменной. При использовании этой же переменной в строковом контексте (например, в операторе вывода) будет произведено преобразование внутреннего представления числа к строке. Другой пример: если переменной не присвоено никакое значение, то в числовом контексте ее значением будет 0, а при использовании ее в строковом контексте - пустая строка (''). К счастью, в большинстве случаев программисту вообще не приходится задумываться о контекстах, поскольку обычно perl выполняет как раз то, что имел в виду автор программы (в полном соответствии с упоминавшимся в первой лекции принципом DWIM). Но знание контекста помогает разобраться с тонкостями использования синтаксических конструкций языка Perl. В следующих лекциях мы познакомимся с другими контекстами, например, списочным и логическим.

В языке Perl существует большое число предопределенных переменных, хранящих разного рода текущую системную и пользовательскую информацию. Они называются специальными переменными, а их имена обычно состоят из одного специального символа. Вот некоторые из специальных переменных:

`$_` область ввода или поиска по образцу, используемая по умолчанию

`$.` номер текущей считанной строки из текущего входного файла

`$/` разделитель входных записей (обычно - символ новой строки `\n`)

`$]` номер версии Perl (например, 5.008007)

`$0` имя файла текущей исполняемой Perl-программы

`$@` сообщение об ошибке при выполнении в блоках `eval` или `do`

`$!` текущий номер ошибки или сообщение об ошибке

`$^E` уточненное сообщение об ошибке

`$^T` время начала выполнения программы (в формате функции `time`)

Некоторые специальные переменные доступны только для чтения, значения же других могут изменяться по усмотрению программиста. Поскольку нелегко запомнить назначение специальных переменных по их "очень специальным" именам, существует указание компилятору (`use English`), которое позволяет обращаться к ним по более понятным длинным именам, например:

`$ARG` вместо `$_`

`$INPUT_LINE_NUMBER` вместо `$.`

`$INPUT_RECORD_SEPARATOR, $RS` вместо `$/`

`$PERL_VERSION` вместо `$]`

`$PROGRAM_NAME` вместо `$0`

`$EVAL_ERROR` вместо `$@`

`$OS_ERROR, $ERRNO` вместо `$!`

`$EXTENDED_OS_ERROR` вместо `$^E`

`$BASETIME` вместо `$^T`

Полный список специальных переменных с их именами, а также советы по их использованию с отличными примерами всегда можно узнать из документации, вызвав справку утилитой

`perldoc perlvar`

О специальной переменной `$_` следует поговорить особо. По своему назначению она выполняет роль местоимения "это" или "этот" (английские `it` или `this`). Ее употребляют, чтобы обратиться к обрабатываемому в текущий момент значению или порции данных. Эту переменную еще называют "переменной по умолчанию" или "буферной переменной", и многие встроенные функции в Perl-программе работают именно с этой переменной, если явно не указан другой аргумент. Например, при чтении из файла в нее может помещаться введенная строка, а функция `print` без параметров печатает значение переменной по умолчанию `$_`.

Большинство специальных переменных являются глобальными, и программисту нужно быть очень осторожным при изменении их значений, так как в других частях программы и в подключаемых модулях может предполагаться их стандартное значение. Чтобы избежать нежелательной модификации таких переменных, нужно в каждой подпрограмме или блоке перед их изменением явно объявить их с помощью описателя `local`:

```
local $save_value = $_;
```

Тогда при выходе из блока будет восстановлено предыдущее значение специальной переменной.

В языке Perl, как и в языке командных интерпретаторов Unix, комментарий начинается с символа `#` и продолжается до конца строки:

```
$lecture_number = 2; # комментарий, занявший часть строки
```

```
# А это комментарий, занимающий всю строку
```

В первой строке программы на Perl можно увидеть особый комментарий: он начинается с символов `#!` (называемых `shebang`, от названия символов - `sharp` и `bang`), и в нем указывается путь к исполняющей системе `perl` (полный путь от корня файловой системы). В операционных системах семейства Unix эта строчка помогает сделать программу на Perl исполняемой (если установить для файла программы флаг, "исполняемый" командой `chmod +x program.pl`). В операционной среде Windows такой комментарий требуется использовать в CGI-программах для web-сервера Apache. Этот комментарий также удобен тем, что в нем можно указывать параметры для исполняющей системы Perl: например, флаг `-w` для вывода дополнительных предупреждений компилятора:

```
#!C:\usr\local\perl\bin\perl -w
```

В Perl нет многострочных комментариев, подобных `/* ... */` в языке C или Java. (Хотя эту возможность можно добавить, если подключить модуль `Acme::Comment`, доступный в хранилище модулей CPAN.) Но если требуются комментарии из нескольких строк, то можно воспользоваться командами системы документирования Perl, называемой POD (от английского "Plain Old Documentation" - "старая добрая документация"). Такой многострочный комментарий можно записать в виде

```
=pod
```

Знак `=` должен располагаться в самом начале строки.

Текст этого комментария фактически является

документацией в формате POD, встроенной в текст программы.

Конец комментария (=cut) также должен быть в начале строки.

```
=cut
```

POD представляет из себя систему разметки текста, в том числе программной документации, который можно просматривать, печатать или конвертировать в другой текстовый формат, например, в HTML. Документация может храниться в текстовых файлах, обычно с суффиксом pod. Но благодаря тому, что компилятор perl игнорирует текст, окруженный командами POD, документацию можно встраивать в нужные места исходного текста программы. Вот наиболее часто используемые команды POD для оформления документации на программу, которые встречаются при чтении исходных текстов на Perl:

```
=pod
```

Начало документации (использовать не обязательно).

```
=headN текст заголовка
```

Заголовок N-го уровня. Уровень N может быть от 1 до 4.

```
=over N
```

Абзац с отступом в N знаков, например, начало списка.

```
=item заглавие элемента
```

Начало элемента списка.

```
=back
```

Окончание списка.

```
=cut
```

Окончание POD-документации, возврат к тексту программы.

Прочитать встроенную в программу POD-документацию в отформатированном виде можно с помощью поставляемой утилиты просмотра:

```
perldoc program_with_pod
```

Описание в формате POD можно преобразовать в web-страницу поставляемой в комплекте с perl утилитой:

```
pod2html --outfile=program.html program_with_pod
```

Конечно, возможностей у системы POD гораздо больше. Узнать о них можно из поставляемой с дистрибутивом Perl документации, прочитав ее с помощью утилиты просмотра документации:

```
perldoc perlpod
```

В этой лекции изложены сведения о литералах и переменных - "молекулах" языка Perl. Они служат основой для создания выражений-"клеток", по воле программиста превращающихся в "живые организмы" - программы на языке Perl, многие из которых проживают долгую жизнь, развиваясь и принося пользу людям. В нескольких следующих лекциях будет излагаться "анатомия" Perl, без знания которой нельзя приступить к написанию программ...

### Лекция 3. Основные операции

В этой лекции описываются скалярные операции языка Perl, для удобства разделенные на группы. Рассматриваются их особенности, правила записи и порядок вычислений в выражениях.

Цель лекции: познакомиться с богатым набором операций языка Perl, узнать их особенности и научиться правильно применять их при составлении выражений.

Операции (также называемые операторами) в Perl многочисленны и разнообразны, с их помощью образуются арифметические, логические, строковые и другие выражения - конструкции, вычисляющие некоторый результат, хотя он не всегда может использоваться. Элементы выражения, над которыми производится операция, называются операндами. Результат операции в Perl может зависеть от контекста, в котором она выполняется. Но часто и сама операция устанавливает определенный контекст, влияющий на преобразование операндов. Конкретные правила вычисления выражений будут изложены при подробном рассказе о каждой из операций.

Конечно же, в Perl, как и в других языках программирования, есть традиционные арифметические операции:

\* умножение (например, `2 * 2` будет 4)

/ деление (например, `11 / 2` будет 5.5)

+ сложение (например, `111 + "999"` будет 1110)

- вычитание (например, `'26' - 1` будет 25)

Обратите внимание на особенность арифметических операций в Perl - в них могут участвовать и строки. При этом, попадая в числовой контекст, строковое представление числа автоматически преобразуется к соответствующему числовому значению. Это очень удобно, но нужно быть внимательным при преобразовании к числам строк, содержащих не-цифровые символы. Символ, не применяемый для записи десятичного числа, прерывает преобразование строки в число, например:

`' +.25x7' + 1` будет 1.25 (то есть `' +0.25' + 1` или `0.25 + 1`)

`'x.25+7' + 1` будет 1 (то есть `' ' + 1` или `0 + 1`)

`'10_000' + 1` будет 11 (то есть `'10' + 1` или `10 + 1`)

В Perl есть еще две очень удобные арифметические операции, которые имеются не во всех языках программирования:

\*\* возведение в степень (например, `2 ** 5` будет 32)

% деление по модулю или остаток от деления (например, `11 % 3` будет 2)

Из языка программирования C заимствованы операции увеличения и уменьшения на единицу, или автоинкремента и автодекремента соответственно. В отличие от перечисленных выше бинарных операций, имеющих два операнда, это унарные операции с одним операндом, применяемые к переменным. Эти операции изменяют значение своего операнда, чем отличаются от большинства других операций, не изменяющих значения операндов. Они могут записываться как в префиксной форме, когда знак операции стоит перед именем переменной, так и в постфиксной форме (также называемой суффиксной), когда знак операции стоит после переменной:

`++` автоинкремент или увеличение на 1 (например, `$n++` или `++$n`)

`--` автодекремент или уменьшение на 1 (например, `$n--` или `--$n`)

В префиксной форме значением выражения будет значение операнда после изменения, а в постфиксной - значение операнда до изменения. Особенности применения префиксной и постфиксной форм можно показать на таком простом примере:

`$n = 25; # начальное значение переменной $n`

`$x1 = $n++; # в $x1 сохранено 25, затем в $n стало 26`

`$x2 = ++$n; # в $n стало 27 и оно сохранено в $x2`

`$x3 = --$n; # в $n стало 26 и оно сохранено в $x3`

`$x4 = $n--; # в $x4 сохранено 26, затем в $n стало 25`

`--$n; # и наконец $n уменьшилось до 24`

Хотя выражение с операцией автоинкремента или автодекремента возвращает значение, часто оно отбрасывается, а сама операция применяется только для изменения значения переменной, что является побочным эффектом ее выполнения. Например:

`++$done_count; # увеличиваем счетчик обработанных строк`

```
$left_count--; # уменьшаем счетчик оставшихся строк
```

В отличие от других языков, в Perl эти операции могут применяться не только к целочисленным, но и к дробным значениям переменных:

```
$f = 2.5; # начальное значение переменной $f
```

```
$f++; # теперь в $f стало 3.5
```

Операции автоинкремента и автодекремента более естественны, чем используемые в других языках эквивалентные выражения наподобие `$n = $n + 1` (которые так шокируют математиков). К тому же они обычно и более эффективно реализованы.

В Perl есть операции унарный минус и унарный плюс, применяемые к числовым и строковым значениям. Например, если в переменной `$n` содержится число, в `$s` - строка, а в `$x` - любое значение, то унарные знаки подействуют так:

- унарный минус (`$n = -$n`; сменит знак числа в `$n` на противоположный)

- унарный минус (`$s = -$s`; добавит перед строкой в `$s` символ '-')

+ унарный плюс (`$x = +$x`; не изменит значения любой переменной)

Поскольку унарный плюс не изменяет значения выражения, он может применяться в ситуации, когда синтаксис требует наличия разделителя, а пробел использовать нежелательно. Например, при указании выражения для вычисления Perl в командной строке.

К унарным операциям также относится операция вызова функции. Многие встроенные функции языка Perl, которые будут рассмотрены в последующих лекциях, в действительности являются именованными унарными операциями. Они могут записываться в традиционном для функций виде с круглыми скобками или как унарные операции без скобок: `sin($x)` или `sin $x`. В таблице 3.1 приведены математические функции и встроенные функции работы со временем.

Таблица 3.1.

| Функция                     | Описание  | Пример использования    | Результат (округленный)  |
|-----------------------------|---|-------------------------|--------------------------|
| <code>abs \$x</code>        | абсолютное значение <code>\$x</code>                              | <code>abs -25</code>    | 25                       |
| <code>atan2 \$y, \$x</code> | арктангенс <code>y/x</code> в интервале от - $\pi$ до $+\pi$      | <code>atan2 25,5</code> | 1.37340077               |
| <code>cos \$x</code>        | косинус <code>\$x</code>  | <code>cos 25</code>     | 0.99120281               |
| <code>exp \$x</code>        | возвращает <code>e</code> в степени <code>\$x</code>              | <code>exp 0.25</code>   | 1.28402542               |
| <code>int \$x</code>        | целая часть от <code>\$x</code>                                   | <code>int 25.25</code>  | 25                       |
| <code>log \$x</code>        | натуральный логарифм <code>\$x</code>                             | <code>log 25</code>     | 3.21887582               |
| <code>rand</code>           | случайное дробное число от 0 до 1                                 | <code>rand</code>       | 0.97265625               |
| <code>rand \$x</code>       | случайное число от 0 до <code>\$x</code>                          | <code>rand 25</code>    | 23.0430603               |
| <code>srand</code>          | начинает новую случайную последовательность для <code>rand</code> | <code>srand</code>      | 1                        |
| <code>sin \$x</code>        | синус <code>\$x</code>  | <code>sin 25</code>     | -0.1323518               |
| <code>sqrt \$x</code>       | квадратный корень из <code>\$x</code>                             | <code>sqrt 25</code>    | 5                        |
| <code>time</code>           | число секунд с начала отсчета (обычно с 01.01.1970)               | <code>time</code>       | 1139738006               |
| <code>localtime</code>      | текущая или указанная дата и время                                | <code>localtime</code>  | Sun Feb 12 14:55:25 2006 |

Наверное, одной из самых популярных встроенных функций можно назвать функцию `print`, выводящую список своих операндов в стандартный поток вывода (обычно на консоль), например:

```
print "Версия Perl=$]"; # вывести номер версии Perl

print 2474.918 / 381.65; # печатать частное от деления

print "Укажите количество чисел: "; # напечатать запрос
```

Подробно об операциях ввода-вывода будет рассказано в лекции 9, где также будут изучены операции проверки файлов - другая разновидность именованных операций.

В Perl нет специальных литералов для обозначения истинного и ложного значения, подобно `true` и `false` в других языках программирования. Необходимость вычислить истинность или ложность выражения определяется логическим контекстом. Логический (или булев) контекст является разновидностью скалярного строкового, поэтому значение выражения преобразуется к строке. Если после преобразования выражения получается пустая строка (не содержащая ни одного символа) либо строка, состоящая из одного символа `'0'` (цифры "нуль"), то значение выражения считается ложным. Значения всех других выражений считаются истинными. Иногда результат вычисления истинности или ложности выражения может показаться немного непривычным, например:

```
' ' или "" пустая строка, поэтому - "ложь"

0 или 0.0 0 преобразуется в '0', поэтому - "ложь"

+0 или -0 0 преобразуется в '0', поэтому - "ложь"

5-(3+2) равно 0, который преобразуется в '0', поэтому - "ложь"

undef неопределенное значение дает в результате '', поэтому - "ложь"

'1' или 'false' не пустая строка и не '0', поэтому - "истина"

'00' или '0.0' не пустая строка и не '0', поэтому - "истина"

'-0' или '+0' не пустая строка и не '0', поэтому - "истина"

'0 but true' не пустая строка, значит - "истина"
```

Истинность или ложность значения выражения вычисляется для логических операций и операций сравнения.

В Perl есть отдельные наборы операций для сравнения чисел и строк. Обозначения операций сравнения чисел совпадают с обозначениями операций в других языках, основанных на синтаксисе языка C. В следующих примерах предположим, что в переменной `$n` хранится значение 25:

```
== равно (не путайте с присваиванием (=), например, $n == 4 ложно)

!= не равно (например, $n != 8*2 истинно)

< меньше, чем (например, $n < '16.08' ложно)

> больше, чем (например, $n > 9 истинно)

<= меньше или равно (например, $n <= 26 истинно)

>= больше или равно (например, $n >= 24 истинно)

<=> числовое сравнение (например, $n <=> 64 вернет -1)
```

Последняя операция числового сравнения `<=>` (называемая на программистском жаргоне `spaceship` - "космический корабль, челнок"), возвращает значение `-1`, `0` или `1`, если первый операнд операции соответственно меньше, равен или больше второго. Все эти операции создают числовой контекст, и строковые операнды этих операций перед сравнением преобразуются к числам. Например, сравнения `123 == '123'` или `'+123' == '123x'` преобразуются к `123 == 123`. Поэтому пустая строка считается равной нулю.



Обозначения операций сравнения строк похожи на обозначения сравнений в языке программирования Fortran. Они применяются, когда сравниваемые величины нужно рассматривать как строки. При сравнении строковых значений учитывается их положение в кодовой таблице символов: чем ближе к началу таблицы, тем меньше значение. В следующих примерах предположим, что в переменной `$s` хранится `'a'`:

`eq` равно (например, `$s eq 'a'` истинно)

`ne` не равно (например, `$s ne 'Y'` истинно)

`lt` меньше, чем (например, `$s lt 'z'` истинно)

`gt` больше, чем (например, `$s gt '9'` истинно)

`le` меньше или равно (например, `$s le 'b'` истинно)

`ge` больше или равно (например, `$s ge 'Z'` истинно)

`cmp` строковое сравнение (например, `$s cmp 'Z'` вернет результат 1)

Последняя операция строкового сравнения `cmp`, так же, как операция числового сравнения `<=>`, возвращает одно из значений: -1, 0 или 1, если первый операнд операции соответственно меньше, равен или больше второго. При сравнении строк имеет значение их длина и содержащиеся в них пробелы: равными считаются посимвольно совпадающие строки одинаковой длины. Операции сравнения строк устанавливают строковый контекст, поэтому их числовые операнды преобразуются к строкам. При этом строковое сравнение чисел дает своеобразный результат, например, `'20'` больше `'100'`, поскольку `'2'` находится в таблице символьных кодов позже, чем `'1'`. Проверка на частичное совпадение строк, которая часто требуется при обработке текста, выполняется с помощью регулярных выражений, которые будут рассмотрены в лекции 8.

Логические операции создают логический контекст выражения, поэтому эти операции возвращают строку `'1'` при истинном значении выражения и пустую строку (`''`), если оно ложное. Обозначение традиционных логических операций в Perl также заимствованы из языка C:

`!` логическое НЕ (например, `! undef($x)` )

`&&` логическое И (например, `$d >= 1 && $d <= 31` )

`||` логическое ИЛИ (например, `$m eq 'Dec' || $m eq 'Jan'`)

Результат операции логическое И будет истинным лишь тогда, когда истинны оба операнда, причем второй операнд вычисляется только тогда, когда первый операнд истинный). Операция логическое ИЛИ возвращает истинный результат, если один из операндов истинный, при этом второй операнд вычисляется только тогда, когда первый операнд ложный. Операция логическое НЕ (или логическое отрицание) меняет значение своего операнда на противоположное. Особенности вычисления логических операций часто применяются в Perl для выполнения действий в зависимости от условия. Например, вывести на печать результат при условии, если он положителен, можно так:

```
$result > 0 && print $result;
```

В языке Perl есть еще один набор логических операций, так называемых логических операций с низким приоритетом. Они эквивалентны упомянутым выше логическим операциям, но имеют почти самый низкий приоритет по сравнению с другими операциями.

`not` логическое НЕ (например, `not undef($x)`)

`and` логическое И (например, `$d >= 1 and $d <= 31`)

`or` логическое ИЛИ (например, `$m eq 'D' or $m eq 'J' or $m eq 'F'`)

`xor` логическое ИСКЛЮЧАЮЩЕЕ ИЛИ (например, `$d==1 xor $m eq 'J'`)

Операция логическое ИСКЛЮЧАЮЩЕЕ ИЛИ возвращает истинный результат, если операнды имеют различное значение, а когда оба операнда имеют одинаковое значение (истинное или ложное), то эта операция возвращает "ложь". Низкоприоритетные логические операции тоже применяются для условного выполнения действий.

Побитовые операции (bitwise operators) выполняются над двоичными разрядами операндов. Унарная операция побитовое НЕ (или побитовое отрицание, или побитовое дополнение) меняет каждый разряд операнда на противоположный. Бинарные операции

побитовое И, побитовое ИЛИ и побитовое ИСКЛЮЧАЮЩЕЕ ИЛИ выполнят соответствующую двоичную операцию поразрядно над левым и правым операндами. Если операнды числовые, то они преобразуются в целые числа, имеющие гарантированную длину не менее 32 разрядов.

~ побитовое НЕ (~ 0b1101 даст результат 0b0010)

& побитовое И (0b1010 & 0b0110 даст результат 0b0010)

| побитовое ИЛИ (0b1010 | 0b0110 даст результат 0b1110)

^ побитовое ИСКЛЮЧАЮЩЕЕ ИЛИ (0b1010 ^ 0b0110 даст 0b1100)

Эти операции могут выполняться над строками. При этом выполняются поразрядные операции над соответствующими битами двух строк, и считается, что более короткая строка дополняется в конце нулевыми разрядами до размера длинной.

Бинарные побитовые операции сдвига выполняются над двоичными разрядами целого числа: значение левого операнда поразрядно сдвигается влево или вправо на число разрядов, указанное правым операндом. При этом освобождающиеся двоичные разряды заполняются нулями.

<< побитовый сдвиг влево (0b1010 << 2 даст результат 0b101000)

>> побитовый сдвиг вправо (0b1010 >> 3 даст результат 0b000001)

В Perl есть бинарные операции, применяемые только к строкам: конкатенация (или сцепление) двух строк, обозначаемая символом "точка", и репликация (или повторение), обозначаемая латинской строчной буквой "x", поскольку в результате этой операции строка "умножается" указанное число раз.

. конкатенация (например, 'X' . '=' . '25' даст строку 'X=25')

x репликация (например, 'Да!' x 3 даст в результате строку 'Да!Да!Да!')

Операция сцепления создает строковый контекст. Поэтому если ее операнды - числа, то они преобразуются в строки, например:

'100' . '3' даст в результате строку '1003'

100 . 3 тоже даст в результате строку '1003'

В левой части операции повторения ожидается строка, а в правой - число повторений. Если в правой части операции повторения стоит строка, то она преобразуется к числу. Причем, если правый операнд операции повторения - дробный, то берется его целое значение, а если он отрицательный или равен нулю, то результатом повторения будет пустая строка:

100 x 3 даст в результате строку '100100100'

100 x 3.9 даст в результате строку '100100100'

100 x 0 или 100 x .1 или 100 x 'x3' даст в результате пустую строку ''

100 x -20 даст в результате пустую строку ''

Рассмотренная ранее операция автоинкремента может особым образом применяться к строкам, состоящим только из латинских букв и цифр. Для таких строк выполняется увеличение значения каждого символа, начиная с правого, с переносом разрядов влево, как у чисел. При этом символ 'a' становится 'b', 'b' становится 'c' и так далее, а 'z' становится 'a' с увеличением значения символа слева. Следующие примеры поясняют сказанное:

\$s = 'xzz'; \$s++; даст в результате строку 'yaa'

\$s = 'XZZ'; \$s++; даст в результате строку 'YAA'

\$s = 'xy9'; \$s++; даст в результате строку 'xz0'

При этом операция автодекремента, примененная к символьным строкам, не обладает "магическим" действием: буквенная строка, как обычно, преобразуется к нулю, который уменьшается на единицу.

В Perl присваивание является бинарной операцией. Ее левым операндом может быть переменная или другая конструкция языка, в которой можно хранить значение. Такая конструкция называется lvalue ("L-значение", от английского left value), то есть

"стоящая в левой части присваивания". Правым операндом может быть любое выражение, а значением выражения присваивания будет значение левого операнда. Например:

```
$num = 5 * 5
```

```
$str = 'Happy New Year!'
```

Поскольку присваивание - это обычная операция, она может участвовать в выражении не один раз, при этом вычисление происходит справа налево:

```
$n1 = $n2 = 25; # иначе говоря: $n2 = 25; $n1 = $n2;
```

Подобно языку C, в Perl имеются составные операции присваивания, совмещающие вычисление результата операции над левым и правым операндами с присваиванием этого результата левому операнду. То есть обычную запись присваивания результата операции переменной

переменная = переменная операция выражение

можно записать в сокращенной форме с использованием составного присваивания:

переменная операция= выражение

В записи составных операций между знаком операции и знаком равенства не должно быть пробела. Вот список допустимых составных операций присваивания с примерами использования:

**\*\*=** присвоить результат возведения в степень (`$n **= 3`)

**\*=** увеличить в (`$n *= 4` значит увеличить `$n` в 4 раза)

**/=** уменьшить в (`$n /= 5` значит уменьшить `$n` в 5 раз)

**%=** присвоить остаток от деления на (`$n %= 6`)

**+=** увеличить на (например, `$n += 7` значит прибавить к `$n` число 7)

**-=** уменьшить на (например, `$n -= 8` значит вычесть из `$n` число 8)

**&&=** И с присваиванием (`$n &&= 2` значит присвоить `$n=2`, если `$n` истинно)

**||=** ИЛИ с присваиванием (`$n ||= 2` т.е. присвоить `$n=2`, если `$n` ложно)

**&=** И с присваиванием (`$n &= 2` значит выполнить над `$n` операцию `&2`)

**|=** ИЛИ с присваиванием (`$n |= 2` т. е. выполнить над `$n` операцию `|2`)

**^=** ИСКЛЮЧАЮЩЕЕ ИЛИ с присваиванием (`$n ^= 2` т. е. `$n = $n ^ 2`)

**<<=** сдвинуть влево и присвоить (`$n <<= 2` значит сдвинуть `$n` на 2 разряда влево)

**>>=** сдвинуть вправо и присвоить (`$n >>= 2` т. е. сдвинуть `$n` на 2 разряда вправо)

**.=** сцепить и присвоить (`$n .= '2'` значит сцепить `$n` с `'2'`)

**x=** повторить и присвоить (`$n x= 2` значит сцепить `$n` с собой 1 раз)

Составные операции присваивания привлекают программистов компактной формой записи и тем, что они часто выполняются более эффективно.

Кроме логических операций, управляющих вычислением выражений, в Perl есть другие операции, которые управляют вычислением нескольких выражений. Это операция "запятая", (или операция вычисления последовательности выражений) которая вычисляет сначала свой левый, а затем правый операнд. При этом значением выражения с этой операцией будет значение правого операнда, хотя это значение часто просто отбрасывается. Операция "запятая" применяется там, где по правилам языка должно быть одно выражение, но при этом нужно выполнить несколько вычислений. Например:

```
$grow++, $col++ # увеличить значения двух счетчиков
```

```
$a = rand 5, $b = int $a # присвоить $a случайное число
```

```
$x+=1, $y=5, $z=$x/$y # то же, что ($x+=1, $y=5), $z=$x/$y
```

Операция выбора (или условная операция) - это единственная в Perl тернарная операция, в которой участвуют три операнда. Первый операнд - условное выражение, определяющее результат операции: если первый операнд истинный, то результатом будет значение второго операнда, иначе - значение третьего операнда. После первого операнда ставится знак вопроса, а после второго - двоеточие. Например:

```
($n < 0) ? 0 : --$n
```

```
$temperature < 20 ? 'холодно' : 'тепло'
```

Перечень операций в языке Perl не ограничивается рассмотренными в этой лекции. По мере изучения других разделов будут описаны другие, более специализированные операции. Например, нам наверняка понадобится операция чтения строки из стандартного входного потока (обычно связанного с системной консолью). Она обозначается символами <> и по-английски называется *diamond*, что на русский лучше всего перевести как "кристалл" (хотя встречается русское название "ромб"). Эта операция считывает очередную строку и возвращает ее значение:

```
$line = <> # считать в $line строку из стандартного ввода
```

В следующих лекциях продолжится разговор об особенностях применения уже изученных и новых операций при работе в разных ситуациях и с другими структурами данных.

Очередность выполнения операций в выражении определяется их приоритетами и ассоциативностью, но она может быть изменена с помощью скобок. Приоритет определяет порядок вычисления операций в выражении: операции с более высоким приоритетом вычисляются раньше. Например, приоритет у операций умножения и деления выше, чем у сложения и вычитания. Ассоциативность определяет порядок вычислений, если в выражении используются операции с одинаковыми приоритетами. Операции с ассоциативностью слева вычисляются обычным образом, слева направо. Но из двух операций, имеющих ассоциативность справа, первой вычисляется стоящая справа. Например:

```
$a ** $b ** $c # ассоциативность справа, т.е. $a**($b**$c)
```

```
$a * $b / $c # ассоциативность слева, т.е. ($a * $b) / $c
```

```
$a && $b && $c # ассоциативность слева, т.е. ($a && $b) && $c
```

```
$a = $b = $c # ассоциативность справа, т.е. $a = ($b = $c)
```

```
$a | $b ^ $c # ассоциативность слева, т.е. ($a | $b) ^ $c
```

В таблице 3.2 приводится полный список операций, перечисленных в порядке убывания приоритетов, с указанием их ассоциативности. В этой таблице даны все операции языка Perl, в том числе и не рассмотренные в этой лекции. Многие из них будут изучены в следующих лекциях.

Таблица 3.2. Ассоциативность и приоритет операций (от высокого к низкому).

| Ассоциативность  | Приоритет | Операция   |
|------------------|-----------|--|
| Слева            | 24        | термы и операции над списками (справа налево)  |
| Слева            | 24        | -> (вызов метода, разыменованье)   |
| Не ассоциативные | 22        | ++ -- (автоинкремент, автодекремент)   |
| Справа           | 21        | ** (возведение в степень)  |
| Справа           | 20        | ! ~ \ + - (логическое НЕ, побитовое отрицание, операция ссылки, унарный плюс, унарный минус) |

|                  |    |   |
|------------------|----|---|
| Слева            | 19 | =~ !~ (привязка к шаблону: совпадение, несовпадение)  |
| Слева            | 18 | * / % x (умножение, деление, деление по модулю, повторение строки)                                  |
| Слева            | 17 | + - . (сложение, вычитание, конкатенация строк)   |
| Слева            | 16 | << >> (битовый сдвиг влево или вправо)  |
| Не ассоциативные | 15 | именованные унарные операции и операции над файлами   |
| Не ассоциативные | 14 | < > ≤ ≤ lt gt le ge (меньше, больше, меньше или равно, больше или равно и их строковые эквиваленты) |
| Не ассоциативные | 13 | == != <=> eq ne cmp (равно, не равно, сравнение и их строковые эквиваленты)                         |
| Слева            | 12 | & (битовое И)   |
| Слева            | 11 | ^ (битовое ИЛИ, битовое исключающее ИЛИ)  |
| Слева            | 10 | && (логическое И)   |
| Слева            | 9  | (логическое ИЛИ)  |
| Не ассоциативные | 8  | .. ... (не включающие или включающие граничные значения диапазоны)                                  |
| Справа           | 7  | ?: (операция выбора или условная операция)  |
| Справа           | 6  | = += -= *= и т. д. (присваивание и присваивание с вычислением)                                      |
| Слева            | 5  | , => (запятая и запятая-стрелка)  |
| Не ассоциативные | 4  | операции над списками (слева направо)   |
| Справа           | 3  | not (логическое НЕ)   |
| Слева            | 2  | and (логическое И)  |
| Слева            | 1  | or xor (логическое ИЛИ, логическое исключающее ИЛИ)   |

В Perl имеется большое количество операций, в том числе и весьма экзотических. Поэтому, если возникает сомнение в порядке их вычислений, то всегда можно использовать скобки для управления очередностью вычислений в выражении.

Понимание изложенного в этой лекции материала о скалярных операциях совершенно необходимо для разработки программ на Perl, хотя запоминать все приведенные здесь операции совершенно излишне. Операции и выражения являются основой для освоения материала о синтаксических правилах языка Perl, о которых пойдет речь в следующей лекции.

#### Лекция 4. Управляющие структуры

В этой лекции описываются основы синтаксиса языка Perl, ключевые слова и правила составления Perl-программы. В ней рассмотрено все многообразие управляющих структур, как традиционных, так и специфичных только для этого языка программирования. Стиль программирования на Perl подразумевает свободу выражения идей программиста, и различные синтаксические варианты максимально способствуют этому.

Цель лекции: познакомиться с синтаксическими правилами составления программ на языке Perl, которые сначала могут показаться непростыми, но обладают исключительной гибкостью и мощностью, предоставляют полный контроль над ходом выполнения программы и учитывают разнообразие стилей при разработке программ.

Минимальная синтаксическая единица языка программирования называется термом. Терм (term) - это все то, что может служить операндом в выражении, например, литерал или переменная. Выражение напоминает неоконченную фразу в естественном языке. Чтобы выражение стало законченным предложением (statement, называемым также утверждением), нужно после него поставить символ ";" (точка с запятой). Другими словами, простое предложение - это выражение, вычисляемое ради его побочного эффекта. Кроме предложений, в программе на Perl могут присутствовать объявления переменных и подпрограмм, которые будут рассмотрены позднее. Приведем примеры простых предложений:

```
$count = 0; # присваивание значения переменной

$count++; # изменение значения операнда

$result = 'Итого=' . $count . "\n"; # подготовка к печати

1; # литерал - минимальное, но корректное предложение
```

Последнее предложение, кажущееся бессмысленным, применяется в последней строке модулей для возврата "истинного" значения при успешной загрузке модуля. О модулях будет подробно рассказано в лекции 13.

Программа на Perl пишется в свободном формате. Это означает, что ее можно записывать сплошным текстом, вставляя для удобочитаемости между терминами и знаками операций любое количество пробельных символов (whitespace) таких как пробел, символ табуляции или перевод строки. По желанию автора можно прервать выражение до или после термина и продолжить его на следующей строке. Можно, конечно, вообще не применять пробельных символов и записывать программу в таком виде:

```
$count=0;$count++;$result='Итого='.$count."\n";1;
```

Но это считается дурным тоном. Да и разобраться в подобной программе будет очень сложно даже ее автору, особенно если она большого размера и прошло некоторое время после ее написания!

Простые предложения, составленные из выражений, выполняются одно за другим, образуя простую последовательность действий. Последовательность может помещаться в блок - одно или несколько предложений, обрамленных фигурными скобками, которые рассматриваются как единое целое. Блоки применяются для группировки программных конструкций, а также для задания области видимости переменных. Точка с запятой может не ставиться в конце последнего предложения в блоке (как это делается в языке Pascal), но лучше ее ставить всегда, на случай, если позднее добавится еще одно предложение. Блок предложений может быть частью управляющей конструкции, такой как цикл или условный оператор. А блок, который не входит ни в одну управляющую конструкцию, называется голым блоком (bare block).

```
{ # начало блока

# последовательность предложений,

# образующих тело блока

} # конец блока
```

Кроме последовательности, в Perl имеются составные предложения, состоящие из выражений и блоков. Составные предложения записываются с помощью ключевых слов (keywords) - специальных слов языка, которые крайне не рекомендуется (а в других языках просто запрещено) применять в качестве имен переменных или подпрограмм. Составные предложения часто называют управляющими структурами, поскольку они предназначены для управления порядком выполнения действий в программе, организуя, например, ветвления и циклы.

Как известно, условные предложения определяют выполнение тех или иных действий в программе в зависимости от проверки заданного условия. В Perl условная конструкция (или оператор if), проверяющая истинность одного условия, в самом простом виде записывается так:

```
if (условие) { # проверка истинности условия

# действия, выполняемые,

# если условие истинно

}

# продолжение программы
```

Обратите внимание, что после условного выражения, обязательно заключенного в круглые скобки, непременно должен стоять блок. Например, так можно вывести на печать значение переменной при условии, что оно - нечетное:

```
if ($count % 2 == 1) { # остаток от деления на 2 равен 1

print $count;

}
```

Другая общеизвестная форма условного предложения содержит блок, выполняемый при ложном условии, а именно:

```
if (условие) { # проверка истинности условия

# действия, выполняемые,

# если условие истинно

} else {

# действия, выполняемые в противном случае

# (если условие ложно)

}

# продолжение программы
```

В Perl имеется еще одна форма условного предложения, которая задает последовательную проверку нескольких условий, указанных в фразе `if` или `elsif`. Она выглядит следующим образом:

```
if (условие 1) { # проверка истинности 1-го условия

# действия, выполняемые,

# если условие 1 истинно}

elsif (условие 2) { # проверка истинности 2-го условия

# действия, выполняемые,

# если условие 2 истинно

# ... здесь могут быть еще фразы elsif ...

elsif (условие N) { # проверка истинности N-го условия

# действия, выполняемые,

# если условие N истинно

} else {

# действия, выполняемые,

# если все условия ложны

}

# продолжение программы
```

При этом выполнится один из блоков действий: соответствующий первому истинному условию или указанный за фразой `else`. Фраза `else` может отсутствовать, тогда при всех ложных условиях не делается ничего. Форма условного предложения с `elsif` заменяет отсутствующий в Perl оператор выбора (такой как `do-case` или `switch`).

Иногда требуется подчеркнуть, что, наоборот, ложность условия становится причиной выполнения каких-либо действий. Для этого в Perl есть еще одна разновидность условного предложения, которая записывается с помощью ключевого слова `unless`:

```
unless (условие) { # проверка ложности условия

# действия, выполняемые,

# если условие ложно

} else {

# действия, выполняемые

# в противном случае

}

# продолжение программы
```

Здесь фраза `else` также может отсутствовать, если при истинности условия не требуется выполнять никаких действий.

Как уже известно из предыдущей лекции, выражать условное выполнение действия можно и другим, очень популярным в Perl способом: с помощью логических операций. Так, например, можно напечатать результат, если первое выражение будет истинным:

```
($count % 2 == 1) and print $count;
```

Условное выполнение действия можно также задавать с помощью модификаторов, о которых речь пойдет далее в этой лекции.

Как известно, циклом называется управляющая конструкция для повторения действий в программе, а однократное выполнение предложений в цикле называется итерацией. В Perl существует множество различных способов задать циклическую обработку данных. Один из них - это операторы управления циклом. Если требуется повторение действий в зависимости от истинности условия, можно воспользоваться циклом `while` (называемый также циклом с предусловием), в котором каждый раз перед выполнением цикла проверяется условие продолжения: если оно истинно, то блок тела цикла повторяется, иначе цикл заканчивается и выполняется следующее предложение программы.

```
while (условие) { # проверка истинности условия продолжения

# действия, выполняемые,

# пока условие истинно

}

# продолжение программы
```

Например, можно воспользоваться таким циклом `while` для вывода на системную консоль десяти случайных чисел от 0 до 1, сгенерированных встроенной функцией `rand`:

```
$i = 0;

while ($i < 10) { # повторять, пока счетчик не достиг 10

print "\n", rand; # вывести с новой строки случайное число

$i++; # увеличить счетчик повторений

}
```

Иногда удобнее для управления циклом задавать условие окончания цикла. В этом случае применяется цикл `until`, в котором каждый раз перед выполнением тела цикла проверяется условие окончания повторений: если оно истинно, цикл заканчивается и выполняется следующее предложение программы, а если условие ложно, то блок тела цикла выполняется еще раз.

```
until (условие) { # проверка истинности условия окончания
```



```
# действия, выполняемые,  
  
# пока условие ложно  
  
}  
  
# продолжение программы
```

Предыдущий пример вывода случайных чисел можно переписать с использованием цикла `until`:

```
$i = 0;  
  
until ($i == 10) { # закончить, когда счетчик достигнет 10  
  
    print "\n", rand; # вывести с новой строки случайное число  
  
    $i++; # увеличить счетчик повторений  
  
}
```

Другая управляющая конструкция - цикл `for` также применяется для повторения действий с проверкой условия продолжения. Но в нем предусмотрены два дополнительных выражения для управления циклом. Первое из них выполняется один раз перед выполнением цикла, и в нем обычно выполняются начальные действия, такие как инициализация переменных. Второе выражение выполняется каждый раз после выполнения тела цикла и перед проверкой условия продолжения работы цикла. Структура этого цикла выглядит так:

```
for (первое выражение; условие; второе выражение) {  
  
    # действия, выполняемые,  
  
    # пока условие истинно  
  
}  
  
# продолжение программы
```

В заголовке цикла могут отсутствовать одно или оба выражения, а в случае отсутствия условия оно считается истинным. Однако при этом должны оставаться две точки с запятой, разделяющие выражения. Снова перепишем приведенный выше пример, на этот раз используя цикл `for`:

```
for ($i=0; $i<10; $i++) { # повторить 10 раз, увеличивая $i  
  
    print "\n", rand; # вывести с новой строки случайное число  
  
}
```

Еще один тип цикла предназначен для перебора всех элементов списка, для чего каждый его элемент последовательно помещается в указанную переменную. Это цикл `foreach`:

```
foreach переменная (список) { #  
  
    # работа с переменной, содержащей  
  
    # текущий элемент обрабатываемого списка  
  
}
```

Примеры использования цикла `foreach` будут приведены в лекции 5, посвященной спискам. В ней будет рассмотрено еще несколько конструкций, также выполняющих повторение действий.

В Perl есть несколько предложений для управления выполнением программы с помощью перехода в указанную точку программы. Обычно это требуется при работе с циклами. Когда при работе цикла требуется досрочно закончить его выполнение при наступлении какого-то события, то для этого можно воспользоваться оператором `last` (аналог оператора `break` в языке C),

который прерывает выполнение цикла и переходит к выполнению предложения, следующего за циклом. Например, напечатать 10 случайных чисел от 0 до 0,5 можно так:

```
$i = 0;

while (1) { # безусловно входим в цикл

$random = rand; # получаем случайное число

if ($i == 10) { # по достижении счетчика 10

last; # ПРЕРЫВАЕМ выполнение цикла

}

if ($random < 0.5) { # числа меньше 0.5

print "\n", $random; # выводим на консоль

$i++; # и увеличиваем счетчик повторений

}

}
```

# сюда произойдет переход по last

Оператор next (аналог оператора continue в языке C) применяется, когда требуется пропустить выполнение оставшихся предложений в теле цикла и перейти к следующей итерации, начав с проверки условия в заголовке цикла. Вот так можно изменить последний пример, применив next:

```
$i = 0;

while ($i < 10) { # пока счетчик не достигнет 10

$random = rand; # получаем случайное число

if ($random > 0.5) { # числа больше 0.5

next; # ПРОПУСКАЕМ действия в теле цикла

}

print "\n", $random; # выводим число на консоль

$i++; # и увеличиваем счетчик повторений

# сюда произойдет переход по next

}
```

Оператор redo используется, когда нужно повторить выполнение предложений в теле цикла без проверки условия в заголовке цикла. Вот таким станет последний пример, если использовать redo:

```
$i = 0;

while ($i < 10) { # пока счетчик не достигнет 10

# сюда произойдет переход по redo

$random = rand; # получаем случайное число

if ($random > 0.5) { # числа больше 0.5
```

```
redo; # СНОВА НАЧИНАЕМ действия в теле цикла

}

print "\n", $random; # выводим число на консоль

$i++; # и увеличиваем счетчик повторений

}
```

Во всех рассмотренных циклах может присутствовать необязательный блок `continue`. В нем располагаются действия, которые нужно выполнить в конце цикла, в том числе при переходе к новой итерации по `next`. Действия в блоке `continue` не выполняются при переходах по `last` и `redo`. Это может показаться странным, но голый блок рассматривается в Perl как цикл, выполняющийся один раз. В таком блоке может присутствовать фраза `continue` и использоваться переходы `last`, `next` и `redo`. С учетом предложений управления циклом и блока `continue` циклическую структуру в общем виде можно изобразить так:

```
# может быть заголовок цикла: while, until или for

{

# сюда происходит переход по redo

# действие 1, выполняемое в цикле

if (условие выхода) {

last; # выход из цикла

}

# действие 2, выполняемое в цикле

if (условие продолжения) {

next; # выполнение следующей итерации цикла

}

# действие 3, выполняемое в цикле

if (условие возобновления) {

redo; # выполнение тела цикла сначала

}

# действие N, выполняемое в цикле

# сюда происходит переход по next

} continue {

# действие, выполняемое перед новой итерацией цикла

}

# сюда происходит переход по last
```

Циклы могут быть вложены один в другой. Когда требуется прервать вложенный цикл, перед ним ставится метка. Метка - это идентификатор, состоящий из латинских букв, знаков подчеркивания и цифр и начинающийся с буквы, после которого стоит знак двоеточия. Соблюдая хороший стиль программирования, следует записывать метки заглавными буквами. В операторе управления циклом метка указывает, выполнение какого цикла нужно прервать:

```
CYCLE_1:
```

```

while (условие продолжения цикла 1) {

CYCLE_2:

while (условие продолжения цикла 2) {

if (условие выхода из всех циклов) {

last CYCLE_1;

}

CYCLE3:

while (условие продолжения цикла 3) {

if (условия прерывания 2-го цикла) {

next CYCLE_2;

}

}

# сюда произойдет переход по next CYCLE_2

}

}

# сюда произойдет переход по last CYCLE_1

```

Метка может ставиться перед любым предложением. При помощи блока и операторов управления циклом с меткой можно имитировать управляющую структуру, подобную оператору switch в языке C. Например, так можно записать только одно присваивание переменной \$say в зависимости от условия:

```

SWITCH: {

unless (defined $t) { # если $t не определено

$t = 25; redo SWITCH; # задать значение по умолчанию

}

if ($t < 10) { $say = 'холодно'; last SWITCH; }

if ($t < 18) { $say = 'прохладно'; last SWITCH; }

if ($t < 27) { $say = 'тепло'; last SWITCH; }

$say = 'жарко';

}

print "Сегодня $say\n";

```

В Perl имеется оператор перехода goto, в котором также используются метки. С его помощью можно перейти к выполнению помеченной конструкции в текущем или в вызывающем блоке. Но его нельзя применять для перехода в конструкцию, требующую инициализации: подпрограмму или цикл for. Этот оператор имеет три разновидности.

1 Переход к выполнению предложения, перед которым стоит метка:

```
goto МЕТКА;
```

2 Переход к метке, имя которой получено в результате вычисления выражения:

```
goto выражение;
```

3 Замена вызова указанной подпрограммы на подпрограмму, выполняющуюся в данный момент (применяется загрузчиками модулей Perl):

```
goto &подпрограмма;
```

Оператор `goto` заслуженно порицается теоретиками и практиками программирования, поскольку он сильно запутывает логику выполнения программы. Так что правилами хорошего стиля программирования рекомендуется использовать его только при крайней необходимости. Хотя `goto` и можно применить для выхода из цикла, но для этого лучше воспользоваться оператором `last`.

Порядок выполнения действий в простом предложении можно задавать с помощью модификаторов выражений. За любым выражением может стоять один из следующих модификаторов:

выражение `if` выражение

выражение `unless` выражение

выражение `while` выражение

выражение `until` выражение

выражение `foreach` список

Модификатор задает условие выполнения (в случае `if` или `unless`) или повторения (в случае `while`, `until` или `foreach`) выражения. Выражение модификатора вычисляется в первую очередь, хотя и стоит в конце конструкции. Хотя модификаторы похожи на условные и циклические управляющие конструкции, но они формируют простые предложения и поэтому не могут быть вложенными. Приведенную выше конструкцию выбора можно переписать с использованием условных модификаторов:

```
SWITCH: {
```

```
$t = -36, redo SWITCH unless defined $t;
```

```
$say = 'холодно', last SWITCH if $t < 10;
```

```
$say = 'прохладно', last SWITCH if $t < 18;
```

```
$say = 'тепло', last SWITCH if $t < 27;
```

```
$say = 'жарко';
```

```
}
```

Иногда удобно задавать повторение действия с помощью циклических модификаторов, например:

```
++$count, --$sum while (rand < 0.1);
```

```
$random = rand until $random > 0.7;
```

Применение модификаторов делает программу легче для понимания, поскольку акцент переносится на основное действие, стоящее в начале предложения. К тому же запись упрощается, так как не используется блок, а условное выражение в модификаторе можно не заключать в круглые скобки.

В программах на Perl можно встретить ключевое слово `do` с последующим блоком, что похоже на управляющую структуру. Но конструкция `do` выступает в качестве термина в выражении. Иначе говоря, `do` делает из блока выражение, значением которого будет значение последнего предложения в блоке. Например, в такой операции присваивания:

```
$result = do { $x=rand; $a=0; } # в $result будет присвоен 0
```

Чтобы подобное выражение стало простым предложением, после него нужно поставить "точку с запятой". Вот так можно записать третий вариант конструкции выбора, где выражение `do` будет операндом условной операции, управляющей вычислением результата:

```

SWITCH: {

    (defined $t) || do { $t = 15; redo SWITCH; };

    ($t < 10) && do { $say = 'холодно'; last SWITCH; };

    ($t < 18) && do { $say = 'прохладно'; last SWITCH; };

    ($t < 27) && do { $say = 'тепло'; last SWITCH; };

    $say = 'жарко';

}

```

Выражение `do`, как и любое другое выражение, может использоваться с модификаторами. Например, с его помощью можно организовать циклическое выполнение действий:

```
do { $sum += rand; } until ($sum > 25);
```

Но поскольку эта конструкция - выражение, а не цикл, то операторы управления циклом в ней не работают.

Иногда требуется динамически вычислить значение строкового выражения или выполнить блок предложений, изолируя возможные ошибки выполнения. Для этого используется конструкция `eval`, которая применяется в одной из двух форм:

`eval` выражение # вычислить выражение как код на Perl

`eval` блок # выполнить блок, перехватывая возможные ошибки

В любой форме `eval` возвращает значение последнего вычисленного выражения. В первой форме строковое выражение рассматривается `eval` как исходный код на Perl, который во время работы программы динамически компилируется и выполняется. Если при его компиляции или выполнении возникает ошибка, `eval` возвращает неопределенное значение, но программа не заканчивается аварийно, а сообщение об ошибке помещается в специальную переменную `$@`. Например:

```

$x = 0; $y = 5; # в выражении будет деление на 0

$expression = "$y/$x"; # строка, содержащая код для выполнения

$result = eval ($expression); # вычислить выражение

if ($@ eq '') { # проверить специальную переменную на ошибки

    print "Выражение вычислено: $result";

} else {

    print "Ошибка вычисления: $@";

}

```

Во второй форме блок предложений в конструкции `eval`, как и в конструкции `do`, становится выражением. Он компилируется обычным образом и выполняется во время работы программы, но возможные ошибки его выполнения также не приводят к аварийному завершению программы. Причину ошибки можно узнать из специальной переменной `$@`, а значением `eval` будет значение последнего предложения в блоке. Вот пример обработки ошибок в выражении `eval`:

```

$result = eval { # выполнить блок

    $x = 0; $y = 5;

    $z = $y / $x; # здесь будет деление на 0

}; # завершаем предложение точкой с запятой

unless ($@) { # проверить специальную переменную на ошибки

```

```
print "Выражение вычислено: $result";

} else {

print "Ошибка вычисления: $@";

}
```

В программе на Perl, помимо предложений и комментариев, применяются прагмы - указания компилятору и исполняющей системе выполнить какие-либо действия или начать работать в определенном режиме. Прагмы позволяют управлять поведением программы при компиляции и выполнении, их довольно много, и полное их описание можно найти в документации. Очень рекомендуется в начале любой программы включать прагмы, отвечающие за более тщательную проверку правил и ограничений:

```
use strict; # ограничить применение небезопасных конструкций

use warnings; # выводить подробные предупреждения компилятора

use diagnostics; # выводить подробную диагностику ошибок
```

Дополнительная диагностика компилятора поможет избежать многих ошибок при выполнении программы. Обычно прагмы могут включаться в любом месте программы с помощью ключевого слова `use` и выключаться при необходимости с помощью ключевого слова `no`, например:

```
use integer; # включить целочисленные вычисления

print 10 / 3; # результат: 3

no integer; # отключить целочисленные вычисления

print 10 / 3; # результат: 3.33333333333333
```

С помощью прагмы `use constant` можно определять в программе именованные константы, которые по традиции записываются заглавными буквами. Это делается таким образом:

```
use constant PI => 3.141592653; # число пи
```

С помощью прагмы `use locale` в программе включается действие национальных системных установок для некоторых встроенных функций, например, при работе со строками на русском языке:

```
use locale;
```

По ходу изучения материала следующих лекций будут рассмотрены другие полезные прагмы, а в лекции 13 будет описано применение `use` для подключения внешних модулей.

Материал этой лекции иллюстрирует упоминавшийся в лекции 1 принцип TMTOWTDI: в Perl часто существует несколько синонимичных конструкций, предоставляющих автору программы возможность наиболее точно выразить свой замысел в привычном для него стиле. Perl - демократичный язык, и каждый пишет на нем так, как ему удобнее и привычнее: начинающий программист использует простые средства, писавший ранее на другом языке найдет для себя знакомые конструкции, а опытный Perl-хакер может углубиться в синтаксические дебри. За многие годы использования Perl целой армией программистов в нем сложились устойчивые выражения (idioms, идиомы), подобные пословицам и поговоркам в естественных языках. Для примера можно привести некоторые из них:

# 1. Выполнять бесконечный цикл

```
for (;;) # читается "forever" - "всегда"
```

```
{ } # тело бесконечного цикла
```

# 2. Открыть файл или аварийно завершить программу

```
open FILE or die; # "открой файл или умри!"
```

# 3. Читать строки из входного потока и печатать их,

```
# используя буферную переменную по умолчанию
```

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

```
# 4. Присвоить переменной значение по умолчанию
```

```
# только, если ее значение не определено
```

```
$variable ||= $default_value;
```

В большинстве следующих лекций будут встречаться и другие идиоматические выражения, придающие специфический колорит программам на языке Perl.

Каждый автор волен оформлять свои программы в удобном для него стиле. Perl не навязывает разработчику никаких ограничений. Общепринятые рекомендации по стилю оформления программ изложены в разделе стандартной документации, который можно просмотреть с помощью команды:

```
perldoc perlstyle
```

В соответствии с устоявшимися традициями, типичная программа на языке Perl скорее всего будет выглядеть примерно так:

```
# вводные комментарии к программе
```

```
use strict; # включение дополнительной...
```

```
use warnings; # ... диагностики
```

```
# use Env; # подключение нужных модулей (см. лекцию 13)
```

```
# package main; # объявление пакета (см. лекцию 13)
```

```
my $message = 'Привет!'; # объявление переменных и
```

```
print $message; # запись алгоритма программы
```

```
# описание форматов отчета (см. лекцию 10)
```

```
# описание подпрограмм (см. лекцию 12)
```

```
__END__ # необязательное обозначение конца программы
```

В этой лекции рассмотрены синтаксические правила составления предложений на языке Perl, изучив которые, можно начинать писать законченные программы. Многообразие синтаксических конструкций позволяет автору, исходя из своих предпочтений, применять любые из конструкций-синонимов для выражения особенностей алгоритма задачи. Дополнительные сведения о синтаксисе предложений, снабженные многочисленными примерами, можно узнать, выполнив Perl-утилиту вывода документации:

```
perldoc perlsyn
```

## Лекция 5. Списки и массивы

В этой лекции рассмотрены списки - один из основных типов данных в Perl. Представлять данные в виде списков и массивов - очень естественно для Perl-программистов. А богатые средства работы со списками, массивами и срезами массивов, приведенные в этой лекции, предоставляют разработчику широкие возможности по обработке данных.

Цель лекции: познакомиться со списками и массивами, освоить возможности работы со списочными данными в Perl, включая встроенные функции и операции в списочном и скалярном контекстах.

Помимо уже изученных скалярных данных, в Perl широко применяется другой тип данных - списки. Если скаляры представляют в программе единичные объекты реального мира, то списки, как и в жизни, позволяют представить набор объектов, однотипных или совершенно разных, которые для решаемой задачи удобно рассматривать как единое целое (например, "список работников", "перечень документов", "опись товаров" и так далее). В то же время, всегда можно обратиться к любому элементу списка и обработать хранящуюся в нем информацию нужным образом, при необходимости повторяя действия для каждого элемента массива.



Итак, список - это упорядоченная последовательность отдельных скалярных данных в виде одного программного объекта. Способом представления значения списка в программе является списочный литерал, который записывается в виде последовательности значений, разделенных запятыми и заключенных в круглые скобки. Вот примеры списочных литералов:

```
(256, 512, 1024, 2048, 4096) # список из 5 чисел
```

```
('John', 'Paul', 'George', 'Ringo') # список из 4 строк
```

```
("Perl", 5.8) # список из строковых и числовых значений
```

Для записи списка текстовых строк, состоящих из одного слова, предусмотрена специальная форма списочного литерала, в которой после ключевого слова `qw` (сокращение от `quoted words` - "слова в кавычках") в скобках записываются строки, не заключенные в кавычки и разделяемые пробельными символами. В качестве скобок могут использоваться традиционные символы: `()`, `{}`, `//`, `\[ ]`, `<>` и даже просто парные символы, такие как `!!` или `##`. Например:

```
qw(это очень удобно) # вместо ('это', 'очень', 'удобно')
```

```
qw/John Paul
```

```
George Ringo/ # список слов, расположенный на 2 строках
```

Элементами списочного литерала могут быть не только строки и числа, но также скалярные переменные и выражения. Пустой список не содержит элементов и представляется списочным литералом без значений (одними круглыми скобками).

```
('One', $x, $x+$y-$z, 2*5) # список литералов и выражений
```

```
() # пустой список
```

Списочный литерал может содержать операцию диапазона, которая записывается в виде двух скалярных значений, разделенных двумя точками:

```
начальное_значение .. конечное_значение
```

В списочном контексте эта операция возвращает список значений. Возвращаемый список начинается со значения левого операнда, которое в цикле увеличивается на единицу, пока не будет достигнуто значение правого операнда. Приведем примеры:

```
5 .. 10 # возвратит список (5, 6, 7, 8, 9, 10)
```

```
5.3 .. 7.1 # возвратит список (5.3, 6.3), т. к. 7.3 > 7.1
```

```
7 .. 5 # возвратит пустой список (), т. к. 5 < 7
```

```
$m .. $n # диапазон, заданный значениями от $m до $n
```

Если операция диапазона применяется к строкам, то значения списка генерируются по правилам операции автоинкремента. С ее помощью удобно записывать различные списочные литералы:

```
(-2 .. 2) # список чисел (-2, -1, 0, 1, 2)
```

```
(25, 53, 77 .. 79) # список (25, 53, 77, 78, 79)
```

```
('A'..'Z', 'a'..'z') # список заглавных и строчных букв
```

```
($start .. $finish) # список значений от $start до $finish
```

Если списочный литерал состоит только из имен переменных, то он может стоять в левой части операции присваивания, в правой части которой будет список присваиваемых значений. Переменным, стоящим слева от знака "равно", последовательно присваиваются значения соответствующих элементов списка из правой части операции.

```
($a, $b, $c) = (10 .. 12); # $a = 10; $b = 11; $c = 12;
```

```
($day, $month, $year) = (18, 12, 1987); # день рождения Perl
```

```
($m, $n) = ($n, $m); # поменять местами значения $n и $m
```

Если в списке слева от знака присваивания переменных больше, чем значений в списке в правой части, то оставшиеся переменные получают неопределенные значения:

```
($hh, $mm, $ss, $ms) = (10, 20, 30); # $ms не определено
```

Если в левой части присваивания переменных меньше, чем значений в правой, то лишние значения не используются.

```
($hh, $mm, $ss) = (10, 20, 30, 400); # 400 отброшено
```

Если в левой части присваивания стоит не список, а скалярная переменная, то устанавливается скалярный контекст, в котором литеральный список возвращает значение своего последнего элемента:

```
$scalar = (10, 20, 30, 400); # то же, что $scalar = 400;
```

Значение списка может храниться в переменной, называемой массив. Перед именем переменной-массива стоит разыменовывающий префикс @ (напоминающий своим видом, что это агрег - "массив"). Одновременно с переменной-массивом в программе может существовать скалярная переменная с таким же именем, но с префиксом \$, так как имена скаляров и массивов хранятся в разных таблицах имен (symbol tables).

```
@variable # массив для хранения списка
```

```
$variable # скаляр для хранения строки или числа
```

Списочное значение помещается в массив с помощью операции присваивания. Присваивание выполняется по-разному в зависимости от контекста, который определяется левым операндом присваивания. Если в левой части присваивания стоит массив или список, то и в правой части ожидается список. Например:

```
@empty = (); # пустой массив после присвоения пустого списка
```

```
@months = (1 .. 12); # массив со списком номеров месяцев
```

```
@days = qw(Пн Вт Ср Чт Пт Сб Вс); # массив дней недели
```

```
@week = @days; # копирование значения массива @days в @week
```

```
@array = 25; # литерал 25 рассматривается как список (25)
```

```
($first) = @array; # в $first скопируется 1-й элемент массива
```

```
@first = @second = (1, 2, 3); # каскадное присваивание
```

Если в левой части присваивания стоит скалярная переменная, то устанавливается скалярный контекст и в правой части операции ожидается скалярное значение. Например, если попытаться присвоить скалярной переменной массив, то ее значением станет размер массива. Того же результата можно добиться, явно задав для массива скалярный контекст встроенной функцией scalar:

```
$array_size = @months; # число элементов (размер) массива
```

```
$array_size = scalar @months; # размер массива
```

В зависимости от контекста, системная функция localtime возвращает разные значения: в скалярном контексте она вернет строку с текущей датой и временем, а в списочном - список из девяти значений с данными о дате и времени:

```
$date_and_time = localtime;
```

```
($sec, $min, $hour, # секунды, минуты, часы
```

```
$mday, $mon, $year, # день, месяц, год,
```

```
$yday, $yday, $isdst) # день недели, день года, часовая зона
```

```
= localtime;
```

В состав списочного литерала могут входить массивы и другие списочные литералы - тогда они заменяются списком своих значений. Но результирующий массив будет одномерным, так как вложенные списки в Perl не предусмотрены. Массивы массивов организуются с помощью ссылок, что будет изучено в лекции 11. Вот примеры такой инициализации массива:

```
@small = (3, 4, 5); # этот массив будет вставлен в список

@big = (1, 2, @small, 6 .. 9); # то же, что @big = (1 .. 9);

@big = ((1, 2), (3 .. 5), (6 .. 9)); # то же, что и выше
```

С помощью списка можно легко добавить новые элементы в начало или в конец существующего массива:

```
@array = ($new_element, @array); # добавить элемент в начало

@array = (@array, $new_element); # добавить элемент в конец

@all = (@first, @second); # объединить два массива в один
```

Присваивая массив списочному литералу, можно извлечь начальные элементы из массива в скалярные переменные, поместив оставшиеся элементы в исходный массив:

```
($element1, @array) = @array; # извлечь элемент из начала
```

Массив в левой части присваивания имеет смысл ставить только в конце списка, поскольку он поглощает все значения, и стоящие после него переменные получают неопределенные значения:

```
(@array, $a, $b) = (1,2,3,4,5); # все 5 чисел попадут в @array
```

Элементы массива - это скалярные величины, доступ к которым происходит по их порядковому номеру (индексу). Поскольку элемент массива - это скаляр, то его обозначение состоит из разыменовывающего префикса \$ перед именем массива, за которым в квадратных скобках стоит индекс. Индексы элементов массива задаются целыми числами, начиная с нуля. (Номер начального индекса массивов раньше мог задаваться значением специальной переменной \$[, но сейчас эта возможность считается устаревшей, поэтому поступать так не рекомендуется.) Вот так выглядит в программе обращение к элементам массива:

```
@array # переменная-массив, хранящая список

$array[0] # первый элемент массива с индексом 0

$array[1] # второй элемент массива с индексом 1

$array[$i] # i-й элемент массива, считая с 0

$array # скаляр, не имеющий отношения к массиву @array
```

Если требуется обращаться к элементам массива, начиная с последнего, то используются отрицательные значения индексов:

```
$array[-1] # последний элемент, то есть 1-й от конца

$array[-2] # предпоследний элемент, то есть 2-й от конца

$array[-$n] # n-й элемент массива, считая с конца
```

Индекс последнего элемента массива, который всегда на единицу меньше размера массива, можно узнать, указав специальный префикс \$# перед именем массива:

```
$last_index = $#array; # индекс последнего элемента @array
```

Так, например, можно выбрать элемент массива с помощью встроенного генератора случайных чисел rand, который возвращает дробное число от 0 до числа, указанного ему в качестве аргумента. Ограничим случайные числа номером последнего индекса в массиве и будем округлять их до целых значений функцией int:

```
$random_element = $array[int(rand($#array))];
```

В `$#array` можно присвоить новое значение последнего индекса, при этом размер массива изменится. Но такое действие обычно не требуется, так как массив при необходимости увеличивается автоматически. Размер массива в Perl не ограничивается, то есть массив может занимать всю отведенную программе память.

В операции присваивания отдельные элементы массива рассматриваются как обычные скаляры. Ниже приведены примеры, по-разному выполняющие одно и то же присваивание элементам массива:

```
$birthday[0] = 18; $birthday[1] = 12; $birthday[2] = 1987;
```

```
($birthday[0], $birthday[1], $birthday[2]) = (18, 12, 1987);
```

(Хотя более естественно для подобного присваивания воспользоваться срезом массива: `@birthday[0, 1, 2] = (18, 12, 1987)`, но о срезах пойдет речь чуть позже в этой лекции.)

Если попытаться присвоить значение элементу с индексом больше текущего размера массива, массив автоматически увеличивается до необходимой длины, а добавленные элементы получают неопределенное значение:

```
$birthday[5] = 'Perl'; # размер @birthday теперь 5
```

```
# значение $birthday[3] и $birthday[4] не определено
```

При попытке обратиться к элементу массива с несуществующим индексом будет возвращено неопределенное значение, но ошибки во время выполнения программы не возникнет.

```
$array[$#array+100] # неопределенно
```

При использовании в качестве индекса массива дробного числа будет взята его целая часть, то есть `$array[3.5]` рассматривается как `$array[3]`.

Часто требуется последовательно перебрать все элементы массива, от первого до последнего, для обработки или вывода их значений. Это можно сделать вполне традиционным образом, с помощью цикла `for`, как это записывается в языках C или Java, а именно:

```
for (my $i = 0; $i < scalar @array; $i++) {  
  
    print "$array[$i] ";  
  
}
```

Perl предлагает для подобных действий более удобный способ с использованием цикла `foreach`, в котором все элементы массива поочередно совмещаются с указанной скалярной переменной. Эта переменная на время выполнения цикла становится синонимом очередного элемента массива, поэтому ее значение доступно не только для чтения, но и для изменения. Только что приведенный пример можно переписать так:

```
foreach my $element (@array) { # $element это синоним  
  
    print "$element "; # очередного элемента $array[$i]  
  
}
```

Можно то же самое записать еще более кратко с помощью модификатора `foreach`, который поочередно перебирает все элементы массива, рассматривая специальную переменную `$_` как синоним текущего элемента массива (то есть переменная `$_` совмещается с очередным элементом массива):

```
print "$_ " foreach @array;
```

Так что каждый Perl-программист выбирает свой способ перебора всех элементов массива в полном соответствии с лозунгом TIMTOWTDI.

Обратите внимание, что значения элементов массива, будучи обычными скалярами, интерполируются в строках, заключенных в двойные кавычки. Целые массивы тоже интерполируются, если имя массива появляется в строке, обрамленной двойными кавычками. При этом значения элементов массива разделяются символом, хранящимся в специальной переменной `$"` (по умолчанию - пробелом). Вот еще один способ напечатать значения всех элементов массива, разделяя их двоеточиями:

```
$" = ':'; # установим разделитель элементов
```

```
print "@array";
```

С помощью индексов можно обращаться не только к элементам массива, но и к элементам списка, в том числе и литерального. Для этого после закрывающей скобки списка указывается значение индекса в квадратных скобках:

```
$fifth = (10..15)[5]; # то же, что $fifth = 15;
```

Обращение по индексу к элементу в списке констант приобретает смысл, если индекс динамически вычисляется при выполнении программы. Вот, например, один из способов преобразовать десятичное число в шестнадцатеричное:

```
$hex = (0..9,'A'..'F')[$dec]; # при $dec==12 в $hex будет 'C'
```

Подобным же образом удобно обращаться к элементу списка, возвращаемого функцией. Например, так можно извлечь день месяца, зная, что у него 3-й индекс в результирующем списке функции `localtime`:

```
$month_day = (localtime)[3]; # элемент списка с индексом 3
```

В Perl есть удобная форма обращения к нескольким элементам массива одновременно, называемая срезом массива. Срез (*slice*) - это набор элементов массива, заданный списком индексов этих элементов. Срез обозначается квадратными скобками после имени массива, в которых перечислены индексы элементов. Поскольку значение среза - это список, при записи среза перед именем массива сохраняется префикс `@`. Срез массива в частном случае может состоять из одного значения, заданного одним индексом. Вот примеры срезов массивов:

```
@array[0,1] # то же, что ($array[0], $array[1])
```

```
@array[5..7] # то же, что ($array[5], $array[6], $array[7])
```

```
@array[3,7,1] # то же, что ($array[3], $array[7], $array[1])
```

```
@array[@indexes] # срез, заданный массивом индексов
```

```
@array[5] # список ($array[5]), а не скаляр $array[5]
```

С помощью срезов удобно одновременно манипулировать значениями нескольких элементов, находящихся в любом месте массива:

```
# присвоить значения пяти элементам:
```

```
@array[5..9] = qw(FreeBSD Linux MacOS NetWare Windows);
```

```
# поменять местами значения 1-го и последнего элементов:
```

```
@array[0,-1] = @array[-1,0];
```

```
# напечатать элементы с индексами от $start до $finish
```

```
print @array[$start .. $finish];
```

Срезы могут применяться не только к массивам, но и к любым спискам, в том числе литеральным.

Для работы с таким популярным типом данных, как массивы, в Perl существует много удобных функций. Когда требуется организовать обработку списка, поочередно извлекая из него элементы, начиная с первого, применяется встроенная функция `shift`. Она удаляет из массива первый элемент, возвращая его значение. Когда `shift` применяется к пустому списку, она возвращает неопределенное значение:

```
$first = shift @array; # извлечь первый элемент в $first
```

```
# синоним: ($first, @array) = @array;
```

С помощью этой функции можно организовать цикл обработки массива, который закончится после извлечения из него последнего элемента, например:

```
while (my $first = shift @array) { # пока @array не опустеет
```

```
print "Обработан элемент $first.";

print "Осталось ", scalar @array, " элементов\n";

}
```

Обратите внимание, что для вывода текущего размера массива нужно использовать `scalar @array` потому, что иначе `print` воспримет `@array` как список для печати и выведет значения массива. Существует противоположная `shift` функция `unshift`, которая вставляет свои аргументы в массив перед первым элементом, сдвигая существующие элементы вправо.

```
unshift @array, $e1,$e2,$e3; # вставить значения в начало
```

```
# синоним: @array = ($e1,$e2,$e3, @array);
```

С помощью массива можно организовать стек, данные в котором обрабатываются по алгоритму LIFO ("last in, first out", "последним пришел - первым ушел"). Для добавления данных в стек применяется операция `push`, которая добавляет элементы в конец массива:

```
push @stack, $new; # добавить элемент в конец массива
```

```
# синоним: @stack = (@stack, $new);
```

Для извлечения одного значения из стека служит встроенная функция `pop`, которая удаляет последний элемент массива, возвращая его значение:

```
$last = pop @stack; # изъять последний элемент массива
```

При помощи комбинации функций `push` и `shift` можно организовать список, реализующий очередь данных, у которой элементы добавляются в конец, а извлекаются из начала (в соответствии с алгоритмом FIFO, "first in, first out", "первым пришел - первым ушел").

Для удаления или замены подсписка в массиве можно использовать функцию `splice`, которая удаляет идущие подряд элементы массива, заданные индексом первого элемента и количеством удаляемых элементов, и заменяет их новым списком (если он указан), возвращая список удаленных элементов.

```
@array = (1..7); # исходный массив
```

```
$offset = 2; $size = 4; # смещение и размер удаляемого списка
```

```
@deleted = splice @array, $offset, $size, qw(новый список);
```

```
# в @array теперь (1, 2, 'новый', 'список', 7)
```

```
# в @deleted попали 4 удаленных элемента (3, 4, 5, 6)
```

Если список для вставки не указан, то подсписок от элемента с индексом `$offset` в количестве `$size` элементов просто удаляется.

Операция сортировки списка выполняется встроенной функцией `sort`, которая, не изменяя своего аргумента, возвращает список, отсортированный по возрастанию строковых значений элементов исходного списка. Поясним на примере:

```
@unsorted = (12, 1, 128, 2, 25, 3, 400, 53);
```

```
@sorted = sort @unsorted;
```

```
# в @sorted будет (1, 12, 128, 2, 25, 3, 400, 53)
```

Если нужно упорядочить список другим образом, то нужно в качестве первого аргумента функции указать блок, выполняющий сравнение двух элементов сортируемого списка и возвращающий значения `-1`, `0`, `1` - они означают, что первый элемент меньше, равен или больше второго. При сравнении чисел это проще всего сделать с помощью операции `<=>`, например:

```
@sorted = sort {$a <=> $b } @unsorted;
```

```
# в @sorted будет (1, 2, 3, 12, 25, 53, 128, 400)
```

В блоке сравнения переменные `$a` и `$b` содержат значения двух текущих сравниваемых элементов. Для выполнения сортировки по убыванию достаточно поменять переменные местами `{ $b <=> $a }`. Помните, что для сортировки в обратном порядке строковых значений нужно применить операцию сравнения строк `{ $b str $a }`. Вместо блока можно вызвать пользовательскую подпрограмму, выполняющую сколь угодно сложные сравнения элементов сортируемого списка.

Перестановку всех элементов списка в обратном порядке выполняет встроенная функция `reverse`, возвращающая инвертированный список, не меняя исходного:

```
@array = qw(Do What I Mean); # исходный список

@backwards = reverse @array; # остается неизменным

# в @backwards будет ('Mean', 'I', 'What', 'Do')
```

Вложенный вызов функций позволяет сначала отсортировать список, а потом переставить элементы в обратном порядке:

```
@backwards = reverse(sort(@array));

# в @backwards будет ('What', 'Mean', 'I', 'Do')
```

Обратите внимание, что во всех приведенных примерах по желанию программиста аргументы функций можно указывать в круглых скобках, но делать это не обязательно. Имея в своем распоряжении мощные примитивы для работы с массивами, подобные `reverse` или `splice`, программист может легко решать весьма нетривиальные задачи. Это подтверждает короткая программа на Perl, выполняющая циклический сдвиг массива тремя вызовами функции `reverse`:

```
my @array = qw/H A Ч A Л O K O H E Ц/; # исходный массив

my $i = 3; # сдвиг массива ВЛЕВО на 3 элемента

my $n = @array; # число элементов массива

# алгоритм сдвига Кена Томпсона (1971)

@array[0 .. $i-1] = reverse @array[0 .. $i-1];

@array[$i .. $n-1] = reverse @array[$i .. $n-1];

@array[0 .. $n-1] = reverse @array[0 .. $n-1];

print "@array\n"; # результат: A Л O K O H E Ц H A Ч
```

Функция `map` позволяет выполнить действия над всеми элементами массива, поэтому ее нередко используют вместо цикла. У этой функции есть две формы вызова:

```
@result = map ВЫРАЖЕНИЕ, СПИСОК

@result = map БЛОК СПИСОК
```

Она вычисляет выражение или блок для каждого элемента списка и возвращает список результатов. С ее помощью, например, можно выполнить арифметическое действие над всеми элементами списка:

```
@result = map $_*10, (11, 32, 55); # работа со списком

# в @result будет (110, 320, 550)
```

При работе `map` специальная переменная `$_` локально устанавливается как синоним текущего элемента списка, поэтому изменение переменной `$_` приводит к изменению соответствующего элемента массива. Таким способом можно изменять значения элементов массива. В этом примере воспользуемся блоком, куда поместим операторы вычисления нового значения (если значение элемента больше 20, оно будет удваиваться):

```
@array = (11, 32, 55); # исходный массив

@result = map {if ($_ > 20) {$_*=10;} else {$_;}} @array;
```

```
# в @result будет (11, 320, 550)
```

Список можно преобразовать в строку с помощью встроенной функции `join`, которая преобразует каждый элемент списка в строку, объединяет отдельные элементы списка в одну строку, вставляя между элементами указанный разделитель, и возвращает полученную строку в качестве результата. Например:

```
@array = (5..10); # объединяемый список
```

```
$delimiter = ':'; # разделитель элементов списка в строке
```

```
$string = join $delimiter, @array; # объединение в строку
```

```
# теперь $string содержит '5:6:7:8:9:10'
```

Обратную операцию разделения строки по образцу на список строк выполняет встроенная функция `split`. Она разделяет строку по указанному разделителю и возвращает список составляющих строк. Можно ограничить число разделяемых подстрок, тогда строка будет разделена не более, чем на это число элементов. Например:

```
$string = '5:6:7:8:9:10'; # исходная строка
```

```
$delimiter = ':'; # разделитель подстрок
```

```
$limit = 3; # число элементов
```

```
@strings = split $delimiter, $string, $limit; # разделение
```

```
# в @strings содержится ('5', '6', '7:8:9:10')
```

Функция `split` имеет гораздо больше возможностей, о которых будет сказано в лекции, посвященной регулярным выражениям. Подробно познакомиться с которыми можно из системной документации с помощью утилиты `perldoc` (после флага `-f` указывается имя искомой функции):

```
perldoc -f split
```

Пользовательские функции и процедуры, как встроенные функции, тоже могут обрабатывать списки: принимать списки параметров и возвращать список значений. Об этом будет подробнее рассказано в лекции 12.

Рассмотренные ранее операции могут вести себя иначе, если они применяются не в скалярном, а в списочном контексте. Так, операция повторения (репликации), чаще всего применяемая к строкам, может также использоваться и для многократного повторения значений списка. Обратите внимание, что она работает именно со списками, поэтому если необходимо размножить значения массива, то его следует поместить в списочный литерал. Например:

```
@two = ('A', 'B') x 2; # будет: ('A', 'B', 'A', 'B')
```

```
@three = (@one) x 3; # в @three трижды повторится @one
```

С помощью операции повторения можно присвоить одинаковые значения всем элементам массива, например, сделать их неопределенными. Таким образом результат функции `undef()` можно повторить по числу элементов массива:

```
@array = (undef()) x @array; # или (undef) x scalar(@array)
```

При необходимости ввести данные в массив можно воспользоваться упомянутой ранее операцией ввода строк `<>`, читающей строки из входного потока. В скалярном контексте операция "кристалл" считывает одну строку в переменную. Вот так можно в диалоге ввести значения в массив из пяти строк:

```
for (@stdin = (), $n = 0; $n < 5; $n++) {
```

```
    print 'Введите значение N ', $n+1, ':';
```

```
    $stdin[$n] = <>; # считать строку в $n-й элемент массива
```

```
}
```



В списочном контексте "кристалл" читает в массив за одну операцию все строки файла. Например, так можно заполнить массив данными, вводимыми с консоли:

```
print "Введите значения - по одному в строке.",  
  
"Для окончания ввода нажмите Ctrl-Z (или Ctrl-D).\n";  
  
@stdin = <>; # считать все строки и поместить их в массив  
  
print "Вы ввели ", scalar @stdin, " значений.\n";
```

Операция <> вводит строки полностью, включая завершающий символ перевода строки, который часто не нужен при дальнейшей обработке. Функция `chomp` удаляет символы перевода строки в конце строки (в скалярном контексте) или в конце каждого элемента списка (в списочном контексте) и возвращает общее число удаленных символов.

```
chomp @stdin; # убрать \n в конце всех элементов массива  
  
$n_count = chomp $string; # убрать \n в конце строки
```

Похожая функция `chop` отсекает любой последний символ у строки (в скалярном контексте) или у каждого элемента списка (в списочном контексте) и возвращает последний отсеченный символ.

```
$last_char = chop $string; # отсечь последний символ строки  
  
chop @array; # отсечь последний символ в элементах массива
```

Если нужно избавиться только от символов перевода строки, то применение функции `chomp` более безопасно, поскольку она никогда не удаляет значащие символы в конце строки.

При выполнении Perl-программы ей доступны значения специальных массивов, в которых хранится полезная служебная информация. Вот некоторые из специальных массивов:

```
@ARGV аргументы командной строки для выполняемой программы  
  
@INC список каталогов для поиска внешних Perl-программ  
  
@_ массив параметров для подпрограмм или буфер для split
```

Рассмотренные в этой лекции материалы по работе со списками и массивами предоставляют программисту мощные и выразительные средства эффективной обработки больших объемов данных. Обобщением идеи массивов стали ассоциативные массивы, которые будут рассмотрены в следующей лекции.

## Лекция 6. Хэши

В этой лекции рассматривается еще один встроенный тип данных языка Perl - хэши или ассоциативные массивы, представляющие собой эффективную реализацию словарей данных. Мощные средства работы с хэшами в Perl позволяют удобно обрабатывать данные самого разного назначения. Использование хэшей стало в Perl естественным представлением данных, часто значительно упрощающих алгоритм программы.

Цель лекции: познакомиться с многообразием средств для работы с хэшами в Perl. Освоить типичные способы применения ассоциативных массивов для решения прикладных задач.

В программировании ассоциативные связи являются одним из основных видов связей между информационными объектами наряду с наследованием (связями типа "предок-потомок") и агрегацией (связями типа "часть-целое"). Ассоциации позволяют устанавливать необходимые логические связи между сущностями по избранному программистом критерию. Ассоциативная связь подобна стрелке на схеме, направленной от одного объекта к другому. Часто ассоциации используются для нахождения по заданной величине соответствующего значения. В этом случае две части ассоциативной связи соответственно называют поисковым ключом (key) и значением (value), ассоциированным с этим ключом. На этом принципе основана классическая структура данных, называемая словарем (dictionary).

В языке Perl для выражения ассоциаций имеются ассоциативные массивы или хэш-таблицы, которые для краткости принято называть хэшами. Хэш (hash) представляет из себя набор ассоциативных связей. Ключом хэша может быть любая скалярная

величина: строка, ссылка, целое или дробное число, автоматически преобразуемое в строку. Причем значения всех ключей в хэше уникальны, поскольку внутренняя организация хэша не допускает ключей с одинаковыми значениями. Ассоциированное с ключом значение может быть любой скалярной величиной. Хэши сочетают в себе ряд привлекательных качеств: гибкость, мощь, быстроту и удобство работы. Поэтому они весьма часто используются при программировании на Perl самых различных задач. С помощью хэшей можно моделировать понятия из математики, информатики, лингвистики и других областей знаний: множества, словари, фреймы, семантические сети, программные объекты и простые базы данных. Размер хэша в Perl ограничен только доступной программе памятью, поэтому хэши позволяют эффективно обрабатывать большие объемы данных, в которых требуется выполнять быстрый поиск. Примечательно то, что в других языках ассоциативные массивы реализованы в виде коллекций объектов в библиотечных модулях, а в языке Perl хэши встроены в ядро языка, что обеспечивает их максимально эффективную работу.

В программе хэш представляется в виде переменной, имеющей тип хэша, которая записывается с размыновывающим префиксом % перед именем. Этот префикс обозначает, что это переменная-хэш, в которой хранится набор ассоциативных связей, иначе говоря, пар "ключ - значение":

```
%hash # переменная-хэш
```

Непосредственные величины ключей и значений хэша могут быть представлены в виде списочного литерала, который записывается как список в круглых скобках, состоящий из элементов хэша. Каждый элемент в литерале состоит из двух частей: поискового ключа и связанного с ним значения, разделенных символами =>, например:

```
('версия' => 5.8, 'язык' => 'Perl') # ключ - строка
```

```
(3.14 => 'число Пи') # ключ - дробь
```

```
(1 => 'one', 2 => 'two', 3 => 'three') # ключ - целое
```

```
($key1 => $value1, $key2 => $value2) # ключ в переменной
```

Операция => эквивалентна запятой, за исключением того, что она создает строковый контекст, так что ее левый операнд автоматически преобразуется к строке. Именно поэтому числа в этом примере записаны без кавычек. Литеральные списки, содержащие ассоциативные пары, обычно применяются для присваивания хэсам начальных значений:

```
%quarter1 = (1 => 'январь', 2 => 'февраль', 3 => 'март');
```

```
%dns = ($site => $ip, 'www.perl.com' => '208.201.239.36');
```

```
%empty = (); # пустой список удаляет все элементы хэша
```

Если в качестве ключа хэша используется переменная с неопределенным значением, то оно преобразуется в пустую строку, которая и станет поисковым ключом. Значения ключей в хэше уникальны, поэтому хэш часто используется для моделирования множества или простой базы данных с уникальным поисковым индексом. При добавлении нескольких элементов с одинаковыми ключами в хэше остается только последний добавленный:

```
%num2word = (10 => 'десять', 5 => 'пять', 10 => 'ten');
```

```
# в %num2word останется только (5 => 'пять', 10 => 'ten')
```

Ситуация, когда с поисковым ключом хэша ассоциируется неопределенное значение, считается нормальной. Это чаще всего означает, что связанное с ключом значение будет добавлено позднее.

Начальные значения элементов хэша могут браться из любого списка, при этом значения нечетных элементов списка становятся в хэше ключами, а четных - ассоциированными с этими ключами значениями. Так что два следующих присваивания эквивалентны:

```
%dictionary = ('я' => 'I', 'он' => 'he', 'она' => 'she');
```

```
%dictionary = ('я', 'I', 'он', 'he', 'она', 'she');
```

И конечно, для заполнения хэша элементами вместо списочного литерала можно использовать массив, содержащий пары "ключ - значение":

```
%dictionary = @list_of_key_value_pairs; # массив пар
```

В повседневной работе хэш заполняется данными из списка, который считывается из файла или генерируется при помощи пользовательской функции.

Следует иметь в виду, что, в отличие от массивов, элементы в хэше не упорядочены, и порядок следования элементов при добавлении элементов в хэш и при выборке их из хэша обычно не совпадает. Все значения, хранящиеся в хэше, можно преобразовать в список, если употребить переменную-хэш в списочном контексте в правой части операции присваивания. Вот так:

```
@key_value_list = %hash; # список ключей и значений
```

При этом в список будут помещены все ассоциативные пары из хэша, и ключи станут нечетными элементами списка, а значения - четными. Порядок копирования в массив ассоциативных пар заранее не известен.

Хэши можно рассматривать как обобщение идеи массива, элементы которого индексируются не только целыми числами, а любыми скалярными значениями. При обращении к элементу хэша в фигурных скобках после имени переменной указывается значение поискового ключа. Поскольку значение элемента хэша - это скалярная величина, при обращении к элементу хэша перед именем переменной ставится префикс \$, как у прочих скалярных значений.

```
$hash{$key} = $value; # добавление значения в хэш по ключу
```

```
$value = $hash{$key}; # извлечение значения из хэша по ключу
```

Начинающие осваивать Perl могут думать про хэши, что это такие странные массивы ("ассоциативные"), у которых индексы могут быть не только числами, но и строками, и поэтому записываются эти необычные индексы не в квадратных скобках, а в фигурных (по-английски "curly braces" - "кучерявые скобки"). Вот примеры использования элементов хэша:

```
$month = 'January';
```

```
$days_in_month{$month}= 31; # со строкой связано число
```

```
$ru{$month}= 'январе'; # со строкой связана строка
```

```
print "В $ru{$month} $days_in_month{'January'} день";
```

В некоторых программах можно встретить при записи элементов хэша строковые ключи, не заключенные в кавычки: это допускается, если ключ - одно слово, записанное по правилам написания идентификаторов, так называемое "голое слово" ("bare word").

Имена хэшей компилятор располагает в другой таблице имен, чем имена массивов или скаляров, поэтому три приведенные ниже переменные абсолютно разные:

```
$variable # скалярная переменная
```

```
@variable # переменная-массив
```

```
%variable # переменная-хэш
```

Типичным применением хэша можно считать составление частотного словаря, в котором со значением каждого слова ассоциируется счетчик его появления в тексте. Для простоты предположим, что слова в файле, содержащем текст, разделены только пробелами:

```
while (my $line = <>) { # считать строку из входного потока
```

```
chomp($line); # удалить из строки символ '\n'
```

```
@words = split(' ', $line); # разбить строку на слова
```

```
foreach my $word (@words) { # для каждого найденного слова
```

```
$hash{$word}++; # увеличить счетчик
```

```
}
```

```
}
```

```
# теперь в %hash содержатся счетчики слов
```

Позднее, в лекции, посвященной регулярным выражениям, будет сказано, как выделять из строки слова не только по пробелам.

Как это было сделано в последнем примере, программисты часто пользуются уникальностью ключей в хэше, чтобы исключить дублирование данных. Для удаления из данных повторений достаточно поместить их в хэш в качестве ключей. При этом даже не обязательно ассоциировать с ключами какие-либо значения. В результате набор ключей хэша будет гарантированно содержать только неповторяющиеся значения из обработанного набора данных.

При обработке данных в хэше часто возникает необходимость проверить наличие в нем элемента с определенным ключом. Функция `exists` проверяет, содержится ли указанный ключ в хэше. Если ключ найден, она возвращает истинное значение ('1'), - и ложное значение (пустую строку), если такого ключа в хэше нет. При этом ассоциированное с ключом значение не проверяется и может быть любым, в том числе и неопределенным. Так можно проверить наличие ключа в хэше:

```
print "ключ $key найден" if exists $hash{$key};
```

При помощи функции `defined()`, возвращающей истинное или ложное значение, можно проверить, было ли задано значение в элементе хэша, ассоциированное с указанным ключом, или оно осталось неопределенным. Например:

```
print "с ключом $key связано значение" if defined $hash{$key};
```

Проверка с помощью функции `defined($hash{$key})` отличается от проверки значения элемента на истинность значения `$hash{$key}`, так как значение элемента может быть определено, но равно нулю или пустой строке, что тоже воспринимается как ложь.

Воспользовавшись функцией `undef()`, можно удалить из хэша только значение элемента, не удаляя его ключа, то есть сделать его неопределенным:

```
undef $hash{$key }; # сделать значение неопределенным
```

После того как значение элемента было удалено функцией `undef()`, проверки наличия в хэше ключа и значения указанного элемента хэша дадут следующие результаты:

```
$hash{$key} # неопределенное значение - это ложь
```

```
defined $hash{$key} # ложь, ибо значение не определено
```

```
exists $hash{$key} # истина, ибо ключ есть
```

Неопределенное значение, хранимое в элементе хэша, означает, что необходимый поисковый ключ присутствует, но с ним не ассоциировано никакого значения.

Добавление элементов в хэш выполняется операцией присваивания, а удаление - функцией `delete`. Эта функция по указанному элементу удаляет из хэша соответствующую пару "ключ - значение" и возвращает только что удаленное значение. Это делается так:

```
$deleted_value = delete $hash{$key}; # удалить элемент
```

Если аргументом функции `delete` будет несуществующий элемент массива, то она просто вернет неопределенное значение, не вызвав ошибки при выполнении программы.

При работе с элементами хэша очень удобно иметь список всех его ключей. Его возвращает функция `keys`. Полученный список можно сохранить в массиве для дальнейшей обработки:

```
@hash_keys = keys %hash; # поместить список ключей в массив
```

Возможно также использовать список ключей для доступа в цикле ко всем значениям хэша. Так можно напечатать частотный словарь из предыдущего примера:

```
foreach my $word (keys %hash) { # для каждого ключа хэша
```

```
print "$word встретилось $hash{$word} раз\n";
```

```
}
```

Элементы хэша, как и другие скалярные величины, помещенные в обрамленную двойными кавычками строку, заменяются своими значениями. Кстати, можно переписать последний пример, добавив сортировку ключей для вывода слов в алфавитном порядке. А

для организации цикла можно применить модификатор `foreach`, совмещающий очередной элемент списка с переменной по умолчанию `$_`:

```
print "$_ встретилось $hash{$_} раз\n"
```

```
foreach (sort keys %hash);
```

Следует помнить, что если размер хэша велик, то и полученный с помощью функции `keys` массив ключей тоже будет занимать большой объем памяти. В скалярном контексте функция `keys` возвращает количество ключей в хэше, поэтому с ее помощью можно легко проверить, не пустой ли хэш:

```
if (keys %hash) { # если scalar(keys(%hash)) != 0
```

```
# обработать элементы хэша, если он не пуст
```

```
}
```

Пустой хэш в скалярном контексте возвращает ложное значение (строку `'0'`), а непустой - истинное. Поэтому проверить, пуст ли хэш, можно еще проще - употребив имя хэша в скалярном контексте, что часто используется в конструкциях, проверяющих условие:

```
while (%hash) { # или scalar(%hash) != 0 (не пуст ли хэш?)
```

```
# обработать элементы хэша
```

```
}
```

Встроенная функция `values`, дополняющая функцию `keys`, возвращает список всех значений элементов хэша в том же порядке, в каком функция `keys` возвращает ключи. С полученным списком можно поступать обычным образом: например, сохранить в массиве или обработать в цикле:

```
@hash_values = values %hash; # сохранить все значения хэша
```

```
print "$_\n" foreach (values %hash); # вывести значения
```

В скалярном контексте функция `values` возвращает количество значений в хэше, так что ее можно использовать для того, чтобы узнать размер хэша. Например:

```
$hash_size = values %hash; # число значений в хэше
```

Функция `each` является встроенным итератором - программной конструкцией, контролирующей последовательную обработку элементов какой-либо коллекции данных. Она предоставляет возможность последовательно обработать все ассоциативные пары в хэше, организуя перебор всех его элементов. При каждом вызове она возвращает двухэлементный список, состоящий из очередного ключа и значения из хэша. Пары элементов возвращаются в неизвестном заранее порядке.

```
($key, $value) = each %hash; # взять очередную пару элементов
```

После того как будет возвращена последняя пара элементов хэша, функция `each` возвращает пустой список. После этого следующий вызов `each` начнет перебор элементов хэша сначала. С ее помощью удобно организовать обработку всех элементов хэша в цикле, который закончится, когда `each` вернет пустой список, означающий "ложь":

```
while (my ($key, $value) = each %hash) { # пока есть пары
```

```
# обработать очередные ключ и значение хэша
```

```
print "с ключом $key связано значение $value\n";
```

```
}
```

Иногда требуется искать ключи хэша по их значениям. Для этого нужно создать обратный ассоциативный массив (или инвертированный хэш), поменяв местами значения и ключи хэша. Это можно сделать так:

```
while (my ($key, $value) = each %hash_by_key) { # ключи хэша
```

```
$hash_by_value{$value} = $key; # становятся значениями
```

```
}
```

Этого же результата можно достичь с помощью функции `reverse`, воспользовавшись тем, что она воспринимает хэш как список, в котором за каждым ключом идет значение, и меняет порядок всех элементов этого списка на обратный. Функция `reverse` возвращает список, в котором в каждой паре элементов за значением следует ключ, и этот список присваивается новому хэшу:

```
%hash_by_value = reverse %hash_by_key; # переворот списка
```

```
$key = $hash_by_value{$value}; # поиск по бывшему значению
```

Нечетные элементы инвертированного списка становятся ключами, а четные - значениями хэша `%hash_by_value`.

Так как весь хэш, его ключи или значения можно легко преобразовать в список, то для обработки хэшей можно применять любые функции, работающие со списками. Именно поэтому в предыдущем примере была применена функция `reverse`. Например, вывести ключи и значения хэша на печать можно так:

```
{ # организовать блок, где объявить временный массив
```

```
my @temp = %hash; # сохранить в нем хэш
```

```
print "@temp"; # и передать его функции print
```

```
} # по выходе из блока временный массив будет уничтожен
```

Можно напечатать хэш по-другому, построчно и в более облагороженном виде, при помощи функции `map`, которая также выполняет роль итератора:

```
print map {"Ключ: $_ значение: $hash{$_}\n" } keys %hash;
```

В этом примере на основании списка ключей, возвращенного функцией `keys`, функция `map` формирует список нужных строк, вставляя из хэша в каждую из них ключ и значение. Она возвращает сформированный список функции `print`, которая выводит его в выходной поток. Кстати, это типичный для Perl прием - обрабатывать данные при помощи цепочки функций, когда результат работы одной функции передается на обработку другой, как это принято делать с помощью конвейеров команд в операционных системах семейства Unix.

В приведенных выше примерах при необходимости обработки ключей хэша в алфавитном порядке они сортировались с помощью функции `sort`. Вот пример обработки хэша в порядке возрастания не его ключей, а его значений:

```
foreach $key ( # каждый элемент списка,
```

```
sort # отсортированный по порядку
```

```
{ $hash{$a} cmp $hash{$b} } # значений, ассоциированных
```

```
keys %hash) { # с ключами хэша
```

```
print "значение:$hash{$key} ключ:$key\n"; # обработать
```

```
} # в цикле
```

Здесь в блоке сравнения функции `sort` сопоставляются значения хэша, ассоциированные с очередными двумя ключами из списка, который предоставлен функцией `keys`.

Подобно тому, как при работе с массивами срезы позволяют работать со списком элементов, в Perl есть возможность обращаться сразу с несколькими элементами хэша. Это делается с помощью среза хэша. Срез хэша (`hash slice`) - это список значений хэша, заданный перечнем соответствующих ключей. Он записывается в виде имени хэша с префиксом `@` (так как срез - это список), за которым в фигурных скобках перечисляются ключи. Список ключей в срезе хэша можно задать перечислением скалярных значений, переменной-списком или списком, возвращенным функцией. Например, так:

```
@hash{$key3, $key7, $key1} # срез хэша задан списком ключей
```

```
@hash{@key_values} # срез хэша задан массивом
```

```
@hash{keys %hash} # то же, что values(%hash)
```

Если в срезе хэша список ключей состоит из единственного ключа, срез все равно является списком, хотя и из одного значения. Сравните:

```
@hash{$key} # срез хэша, заданный списком из одного ключа
```

```
$hash{$key} # значение элемента хэша, заданное ключом
```

Поскольку переменная-хэш в составе строки не интерполируется, для вставки в строку всех значений хэша можно воспользоваться срезом хэша:

```
%hash = ('0' => 'false', '1' => 'true');
```

```
"@hash{keys %hash}"; # будет "false true" или "true false"
```

Срез хэша, как и любой другой список, может стоять в левой части операции присваивания. При этом списку ключей среза должен соответствовать список присваиваемых значений в правой части присваивания. Воспользовавшись срезом, можно добавить в хэш сразу несколько пар или объединить два хэша, добавив к одному другой. Например:

```
@hash{$k1, $k2, $k3} = ($v1, $v2, $v3); # добавить список
```

```
@old{keys %new} = values %new; # добавить хэш %new к %old
```

С помощью среза хэша и функций `keys` и `values` можно поменять в хэше местами ключи и значения, то есть сделать значения ключами, а ключи - значениями.

```
@hash_keys = keys %hash; # сохранить ключи в массиве
```

```
@hash_values = values %hash; # сохранить список значений
```

```
%hash = (); # очистить хэш
```

```
@hash{@hash_values} = @hash_keys; # срезу хэша присвоить список
```

Исполняющая система Perl предоставляет программисту доступ к специальным ассоциативным массивам, в которых хранится полезная служебная информация. Вот некоторые из специальных хэшей:

`%ENV` перечень системных переменных окружения (например, `PATH`)

`%INC` перечень внешних программ, подключаемых по `require` или `do`

`%SIG` используется для установки обработчиков сигналов от процессов

Например, так при выполнении программы можно использовать значения переменных окружения: перечислить все их значения или выбрать нужные.

```
foreach my $name (keys %ENV) { print "$name=$ENV{$name}\n"; }
```

```
($who, $home) = @ENV{"USER", "HOME"}; # под Unix
```

```
($who, $home) = @ENV{"USERNAME", "HOMEPATH"}; # и Windows XP
```

Рассмотренный в этой лекции тип данных - хэш - не добавляет нового типа контекста: ключи и значения отдельных элементов хэша - это скалярные величины, а перечень всех элементов хэша, срезы хэша, выборки всех его ключей и всех его значений - это списки. Хотя переменная-хэш хранит особое значение - ассоциативный массив, но когда она применяется в левой части операции присваивания, она создает списочный контекст. На этом основаны приемы инициализации хэшей значениями списков. Поэтому же, например, при присваивании хэшу скалярной величины она рассматривается как список, состоящий из одного элемента, и этот элемент становится единственным ключом в хэше, с которым ассоциируется неопределенное (не присвоенное) значение:

```
%hash = $scalar; # то же, что %hash = ($scalar)
```

```
# defined($hash{$scalar}) будет ложно: значения не было
```

```
# exists(%hash{$scalar}) будет истинно: ключ есть
```

В этой лекции завершается изучение основных типов данных в языке Perl: скаляров, списков и хэшей. В таблице 6.1 для сравнения приведены контексты и форматы обращения к скалярным переменным, элементам массивов и хэшей и их срезам.

Таблица 6.1. Форматы записи переменных

| Конструкция         | Хранимое значение           | Описание   | Контекст (в левой части присваивания) |
|---------------------|-----------------------------|--|---------------------------------------|
| @variable           | список                      | весь массив @variable  | списочный                             |
| %variable           | хэш                         | весь хэш %variable   | списочный                             |
| \$variable          | скаляр                      | просто скалярная переменная  | скалярный                             |
| \$variable[\$index] | скаляр                      | элемент массива @variable, заданный индексом \$index               | скалярный                             |
| @variable[@list]    | список                      | срез массива @variable, заданный списком индексов @list            | списочный                             |
| @variable[\$index]  | список (из одного элемента) | срез массива @variable, заданный списком из одного индекса \$index | списочный                             |
| \$variable{\$key}   | скаляр                      | элемент хэша %variable   | скалярный                             |
| @variable{@list}    | список (значений)           | срез хэша %variable, заданный списком ключей @list                 | списочный                             |
| @variable{\$key}    | список (из одного значения) | срез хэша %variable, заданный списком из одного ключа \$key        | списочный                             |

Дополнительные сведения о хэшах можно узнать из справочной документации, обратившись к разделу о типах данных:

```
perldoc perldata
```

Хэши - это, наверное, самая популярная структура данных при программировании на Perl. Без них не обходится ни одна серьезная программа, ведь их применение делает многие алгоритмы проще, а программу - понятнее. Материал этой лекции показывает, насколько удобно и просто пользоваться хэшами. Особенный интерес представляет возможность хранения в ассоциативных массивах ссылок на другие структуры данных: массивы, хэши, объекты, подпрограммы. Это позволяет создавать сложные динамические структуры данных, о чем будет сказано в лекции 11, посвященной ссылкам.

Лекция 7. Текст, строки и символы

В этой лекции описываются средства работы с текстовой информацией, символьными и строковыми данными, которыми славится язык Perl, начиная с самых первых версий.

Цель лекции: получить углубленные знания о символьных и строковых данных в Perl и освоить специфические приемы работы с ними для успешного решения задач обработки текстовой информации.

Язык программирования Perl, в первую очередь, получил широкую известность как средство обработки текстовой информации - удобное, быстрое, мощное, гибкое. Ларри Уолл создал Perl, чтобы облегчить свою жизнь, когда ему, молодому системному администратору, пришлось заниматься обработкой больших объемов данных, преимущественно текстовых. Удобство работы с текстом заложено практически во всех языковых конструкциях: например, строковый контекст включает автоматическое преобразование чисел и ключей хэша к строкам. В систему программирования Perl встроены необходимые функции для работы с символьной информацией. Наверное, самое мощное средство работы с текстовой информацией - обработка регулярных выражений - эффективно реализована в ядре Perl. Дополнительные средства обработки текста реализованы в стандартных библиотеках. Еще больше функций и классов для работы с текстовыми данными можно найти в модулях из репозитория CPAN.



Текстовая информация хранится в Perl-программе в скалярных переменных. Поскольку Perl не накладывает искусственных ограничений на использование ресурсов компьютера, обычная практика в программах на Perl - считывание всего текста из файла вместе с разделителями строк в одну скалярную переменную, чтобы затем эффективно обработать его. Поэтому в Perl переменные, содержащие символьные данные, называют "строковыми" лишь для краткости, чтобы не говорить "скалярная переменная, содержащая строковое значение".

Уже известные из лекции 2 строковые литералы, заключаемые в апострофы и двойные кавычки, могут записываться в альтернативной форме:

'строка в апострофах' или q(строка в апострофах)

"строка в кавычках" или qq(строка в кавычках)

Подобно литеральному списку слов qw(), упомянутому в лекции лекции 5, строковые литералы в этом формате могут ограничиваться разными скобками и практически любыми парными символами: (), {}, [], <>, //, \\\, !! и так далее. Конечно, применение в качестве ограничителей строк таких символов, как &&, ||, %%, ##, '' или \$\$, допустимо, но не рекомендуется, поскольку может ввести в заблуждение читателя программы. Правила интерполяции действуют и на эту форму записи строковых литералов.

В Perl есть особые строки, очень похожие на литералы: это строки, заключенные в обратные апострофы (back-quotes, backticks) ``, для которых также есть эквивалентная запись в виде qx(). Особенность таких строк заключается в том, что их содержимое рассматривается как синхронный вызов внешней программы или команды операционной системы, которая выполняется во время работы Perl-программы. Фактически это операция выполнения программы. Результат выполнения указанной внешней программы становится значением конструкции qx(). При этом в ней производится интерполяция. Так, например, в среде MS Windows или Linux с помощью команды dir можно получить список MP3-файлов и поместить его в переменную:

```
$music_files = `dir *.mp3`; # или qx(dir \*.mp3)
```

Таким же образом можно легко воспользоваться услугами любой другой программы. Недаром Perl часто называют "склеивающим языком" (glue language): с помощью Perl-программы можно обращаться к имеющимся программам, получать результат их выполнения и обрабатывать его по усмотрению программиста. Так, упомянутый в лекции 1 прием использования программ-фильтров получил в Perl дальнейшее развитие. Другие примеры использования операции выполнения программы приведены в лекции 16.

Встречается еще один тип строковых литералов, называемых V-строки ("V-strings" - строки версий), хотя он считается устаревшим и может не поддерживаться в будущем.

```
v1.20.300.4000 # то же, что "\x{1}\x{14}\x{12c}\x{fa0}"
```

```
v9786 # "смайлик" ? (символ Unicode \x{263A})
```

```
v79.107.33 # строка 'Ok!'
```

```
79.107.33 # в литерале с несколькими точками можно без "v"
```

V-строки полезны для сравнения "номеров" версий с помощью операций строкового сравнения, например:

```
$version = v5.8.7;

print "Версия подходит\n" if $version ge v5.8.0;
```

V-строки иногда также применяются для записи сетевых адресов IPv4, например: v127.0.0.1.

Кроме escape-последовательностей, описанных в лекции 2, в Perl есть особые управляющие последовательности, предназначенные для преобразования символов в строковом литерале. Они приведены в таблице 7.1. С их помощью преобразуется либо один символ, следующий за escape-последовательностью, либо несколько символов до отменяющей последовательности.

Таблица 7.1. Преобразующие escape-последовательности

| Управляющая последовательность | Мнемоника символа | Преобразование                                     |
|--------------------------------|-------------------|--|
| \u                             | Upper case        | преобразовать следующий символ к верхнему регистру |
| \l                             | Lower case        | преобразовать следующий символ к нижнему регистру  |

|    |            |   |
|----|------------|---|
| \U | Upper case | преобразовать символы до \E к верхнему регистру     |
| \L | Lower case | преобразовать символы до \E к нижнему регистру      |
| \Q | Quote      | отменить специальное значение символов вплоть до \E |
| \E | End        | завершить действие \U или \L или \Q                 |

Применение этих преобразующих escape-последовательностей можно проиллюстрировать такими примерами:

```
use locale; # для правильной обработки кириллицы
```

```
$name = 'мария'; # будем преобразовывать значение переменной
```

```
print "\u$name"; # будет выведено: Мария
```

```
print "\U$name\E"; # будет выведено: МАРИЯ
```

```
print "\Q$name\E"; # будет выведено: \м\а\r\и\я
```

Аналогичного результата можно достигнуть при использовании некоторых строковых функций, о которых пойдет речь далее в этой лекции.

Еще одним видом непосредственной записи в программе текстовой информации являются так называемые встроенные документы (here-documents). Эта конструкция, заимствованная из командного языка Unix, представляет из себя встроенный в программу произвольный текст. Встроенный документ начинается символами <<, за которыми без пробелов указывается ограничитель, отмечающий конец документа. Все строки, начиная со следующей, рассматриваются как содержимое этого документа до тех пор, пока не встретится строка, состоящая только из указанного ограничителя. Обозначающий конец встроенного документа ограничитель должен записываться на отдельной строке с самого ее начала.

```
$here_document = <
```

```
Здесь располагается текст встроенного документа,
```

```
ограничитель которого записывается с начала
```

```
на отдельной строке.
```

```
END_OF_DOC
```

Если желательно записывать ограничитель с пробелами, то его нужно заключить в кавычки, а если он записан кириллицей, то нужно прагмой use locale включить учет национальных установок:

```
use locale;
```

```
$here_document = <<'КОНЕЦ ДОКУМЕНТА';
```

```
ЭТО НЕ КОНЕЦ ДОКУМЕНТА
```

```
КОНЕЦ ДОКУМЕНТА
```

Во встроенных документах производится интерполяция переменных, если только ограничитель here-документа не заключен в одинарные апострофы. Поэтому встроенные документы часто применяются для комбинирования предварительно отформатированного текста со значениями переменных, как это сделано в следующем примере:

```
$here_document = <<"END_OF_DOCUMENT"; # присваивание строке
```

```
Уважаемый $guests[$n]!
```

```
Приглашаем Вас на презентацию книги "$title",
```

```
которая состоится $date в $time.
```

```
Оргкомитет.
```

```
END_OF_DOCUMENT
```

```
print $here_document, '-' x 65, "\n";
```

Например, с помощью here-документа легко и удобно программно создать HTML-страницу, вставляя в нее нужную информацию:

```
$web_page = <
```

```
PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
```

```
content="text/html; charset=$encoding"/>
```

`$header`

```
$article{$number}
```

Вернуться к разделу `$topic`

```
Copyright © $year, $author.
```

HTML

Это один из способов динамического создания на web-сервере гипертекстовых страниц в ответ на запрос информации, например, хранимой в базе данных.

В предыдущих лекциях уже упоминались функции, обрабатывающие символьную информацию:

```
[x]. chomp(), удаляющая в конце строки символ-разделитель записей;
```

```
[x]. chop(), отсекающая любой последний символ строки;
```

```
[x]. join(), объединяющая элементы массива в одну строку;
```

```
[x]. split(), разделяющая строку на список подстрок.
```

В этой лекции мы познакомимся с другими встроенными функциями для работы с текстом. Хотя в приведенных далее примерах аргументы функций заключены в круглые скобки, использование скобок при вызове встроенных функций необязательно, если не возникает неоднозначности определения аргументов функции.

Часто требуется выяснить, содержит ли строка ту или иную подстроку. Функция `index()` выполняет поиск подстроки в строке, начиная с определенного смещения, и возвращает номер позиции найденной подстроки. Функция `rindex()` ищет подстроку от конца строки и возвращает позицию последней подстроки в строке перед указанным смещением. Смещение можно не указывать, тогда поиск производится во всей строке. Номера позиций подстроки и смещения начинаются с нуля. Если подстрока не найдена, возвращается `-1`. Например:

```
$pos = index($string, $sub_string, $offset); # с начала
```

```
$last_pos = rindex($string, $sub_string, $offset); # с конца
```

```
print "есть правда!" if(index($life, 'правда') != -1);
```

В следующей главе будет рассказано о регулярных выражениях, с помощью которых можно гибко управлять поиском подстроки, задавая шаблоны приблизительного соответствия и расположение подстрок относительно друг друга.

Определение длины текста - также весьма распространенная операция. Функция `length()` возвращает длину в символах значения строки или выражения, возвращающего строку или преобразованного к строке:

```
$string_length = length($string); # строка в переменной
```

```
$n *= 2 until(length($n)>10); # длина числа
```

```
print 'Текст слишком длинный' if length($s1 . $s2) > $limit;
```

Функция `substr()`, выполняющая выделение подстроки из строки, всегда была очень популярной в большинстве языков (кроме Perl, в котором это действие чаще выполняется с помощью регулярных выражений). Она копирует из строки подстроку заданной длины, начиная с указанного смещения. Если смещение отрицательное, то оно отсчитывается от конца строки. Если длина подстроки не задана, то копируется строка после смещения до самого конца:

```
$sub = substr($string, # копировать в $sub из $string,
```

```
$offset, # отступив $offset символов,
```

```
$length); # подстроку длиной $length
```

```
$e = substr($s, rindex($s, '.')); # от последней '.' до конца
```

```
$last_char = substr($string, -1, 1); # последний символ
```

Необычность функции `substr()` в Perl состоит в том, что она может применяться для изменения строки, относясь к группе так называемых левосторонних функций, которые могут употребляться в левой части операции присваивания. В этом случае значение, стоящее в правой части присваивания, заменяет подстроку, которая извлекается из строки функцией `substr()`, стоящей слева от знака присваивания. Например, вот так можно подстроку длиной в два символа, начинающуюся с символа с индексом 5, заменить новой строкой:

```
$string = 'Perl 5 нравится программистам.';
```

```
$new_string = '6 тоже по';
```

```
substr($string, 5, 2) = $new_string;
```

```
# в $string будет: 'Perl 6 тоже понравится программистам.'
```

Подобным же образом можно удалить последние 5 символов строки, заменив их пустой строкой:

```
substr($string, -5) = ''; # удалить последние 5 символов
```

Сочетая уже известные функции, можно выполнять разные манипуляции с текстовой информацией. Например, чтобы переставить слова в строке, можно воспользоваться функциями `split()`, `reverse()` и `join()` в списочном контексте:

```
$reverse_words = join(' ', reverse(split(' ', $text)));
```

В Perl есть набор функций для преобразования букв из заглавных в строчные и наоборот. Для правильного преобразования русских букв нужно включить поддержку национальных установок операционной системы с помощью прагмы `use locale`. Преобразовать текст к нижнему регистру (lower case) можно с помощью функции `lc()`, которая возвращает значение текстового выражения, преобразованное к строчным буквам:

```
use locale; # учитывать национальные установки
```

```
$lower_case = lc($text); # преобразовать к маленьким буквам
```

Функция `lcfirst()` возвращает значение строкового выражения, в котором только первый символ преобразован к нижнему регистру, например:

```
$first_char_lower = lcfirst($text); # 'Perl' станет 'perl'
```

К верхнему регистру (upper case) преобразовать текст можно с помощью функции `uc()`, которая возвращает значение символьного выражения, преобразованное к заглавным буквам.

```
use locale;
```

```
$upper_case = uc($text); # преобразовать к большим буквам
```

Функция `ucfirst()` возвращает значение строкового выражения, в котором только первый символ преобразован к верхнему регистру. Так, например, можно записать имя собственное с заглавной буквы:

```
$capitalized = ucfirst($name); # 'ларри' станет 'Ларри'
```

Встроенная функция `crypt()` выполняет шифрование строки, переданной ей в качестве аргумента, используя второй аргумент в качестве "затравки" (salt) для шифрования:

```
# незашифрованная строка из $plain шифруется в $crypted
```

```
$crypted = crypt($plain, $salt);
```

Эта функция не имеет парной расшифровывающей функции и чаще всего используется для сравнения открытого текста с существующей зашифрованной строкой, как это делается в следующем примере:

```
if (crypt($plain, $salt) eq $crypted) {
```

```
# открытый текст совпал с зашифрованным
```

```
}
```

Функция `quotemeta()` находит в символьном выражении метасимволы (о которых пойдет речь в следующей лекции) или escape-последовательности и возвращает строку, где у всех специальных символов отменено их особое значение: для этого перед каждым из них ставится символ обратной косой черты `'\'`.

```
$string_with_meta = '\n \032 \x00 text \t \v "';
```

```
$quoted = quotemeta($string_with_meta);
```

```
# в $quoted будет '\\n\\ \032\\ \\x00\\ text\\ \\t\\ \\v\\ \"'
```

В Perl имеется несколько функций преобразования строкового представления числа в числовое значение. Функция `hex()` возвращает десятичное значение выражения, представленного как шестнадцатичное число в виде строки:

```
$hexadecimal_as_string = '0x2F';
```

```
$decimal_number = hex($hexadecimal_as_string); # будет 47
```

Функция `oct()` возвращает десятичное значение строкового выражения, представляющего запись восьмеричного числа:

```
$octal_as_string = '0777';
```

```
$decimal_number = oct($octal_as_string); # будет 511
```

С помощью `oct()` можно также преобразовать к десятичному значению двоичное или шестнадцатичное число, записанное в виде строки:

```
$binary_as_string = '0b011001';
```

```
$decimal_number = oct($binary_as_string); # будет 25
```

```
$hexadecimal_as_string = '0x19';
```

```
$decimal_number = oct($hexadecimal_as_string); # будет 25
```

Ну а строку, содержащую число в десятичной системе счисления, можно преобразовать к числу, поместив ее в числовой контекст:

```
$pi_as_string = '3.141592653'; # число Пи в виде строки
```

```
$circle_length = 2 * $pi_as_string * $radius;
```

Функция `sprintf()` возвращает строку, которая сформирована в соответствии с правилами форматирования, заимствованными из языка C: на основе формата преобразования, заданного первым аргументом, в результирующую строку подставляются отформатированные значения из списка остальных аргументов функции. В общем виде вызов этой функции выглядит так: `sprintf(ФОРМАТ, СПИСОК АРГУМЕНТОВ)`. В формате преобразования располагается любой текст, в котором могут присутствовать указания преобразования. Каждое указание начинается с символа процента (%) и заканчивается символом, определяющим преобразование. Основные преобразования приведены в таблице 7.2.

Таблица 7.2. Преобразования в формате `sprintf`

| Преобразование | Синоним | Результат преобразования                         | Мнемоника символа |
|----------------|---------|--|-------------------|
| %%             |         | Знак процента                                    | %                 |
| %c             |         | Символ с указанным номером в кодовой таблице     | Character         |
| %s             |         | Строка   | String            |
| %d             | %i      | Целое со знаком в десятичном виде                | Decimal, Integer  |
| %u             |         | Целое без знака в десятичном виде                | Unsigned          |
| %b             |         | Целое без знака в двоичном виде                  | Binary            |
| %o             |         | Целое без знака в восьмеричном виде              | Octal             |
| %x             | %X      | Целое без знака в шестнадцатеричном виде         | heXadecimal       |
| %e             | %E      | Целое с плавающей точкой в научной нотации       | Exponential       |
| %f             | %F      | Число с плавающей точкой в виде десятичной дроби | Float             |
| %g             | %G      | Число с плавающей точкой в формате %e или %f     |                   |

Между знаком процента и символом в указании преобразования можно использовать дополнительные параметры преобразования, основные из которых приведены в таблице 7.3.

Таблица 7.3. Параметры преобразования в формате `sprintf`

| Параметр | Выполняемое форматирование  | Пример параметров <code>sprintf()</code> | Результат форматирования |
|----------|---|--|--------------------------|
| число    | Минимальная ширина поля вывода для результата преобразования; если она не задана или меньше ширины значения, то устанавливается равной ширине выводимого значения | '<%5s>', 25                              | < 25>                    |
| .число   | Количество цифр после десятичной точки в дробном числе  | '<%.5f>', 0.25                           | <0.25000>                |
|          | Максимальная ширина поля вывода, до которой усекается длинная строка  | '<%.5s>', '5' x 10                       | <55555>                  |
| пробел   | Вывод пробела перед положительным числом  | '<% d>', 25                              | '< 25>'                  |

|   |  |              |           |
|---|--|--------------|-----------|
| + | Вывод плюса перед положительным числом   | '<%+d>', 25  | '<+25>'   |
| 0 | Вывод нулей, а не пробелов при выравнивании по правому краю поля                                 | '<%05s>', 25 | '<00025>' |
| - | Выравнивание значения по левому краю поля  | '<%-5s>', 25 | '<25 >'   |
| # | Вывод перед восьмеричным числом 0 , перед шестнадцатеричным числом 0x , перед двоичным числом 0b | '<%#x>', 25  | '<0x19>'  |

При выполнении `sprintf()` к очередному значению из списка аргументов применяется преобразование, результат которого вставляется в формирующую строку на место указания преобразования. Например, если шаблон форматирования и аргументы функции `sprintf()` заданы так:

```
$format = "'%12s' агента <%03d> = '%+-10.2f'";
```

```
@list = ('Температура', 7, 36.6);
```

```
$formatted_string = sprintf($format, @list);
```

то после выполнения приведенного предложения в переменной `$formatted_string` будет содержаться такая отформатированная строка:

```
' Температура' агента <007> = '+36.60 '
```

Преобразования в формате этого примера обозначают следующее:

[x]. `%12s` - преобразовать аргумент в строку (string) и поместить в поле шириной в 12 символов с выравниванием вправо (т. к. ширина поля положительная);

[x]. `%03d` - преобразовать аргумент в десятичное целое (decimal) и поместить в поле шириной в 3 цифры с ведущими нулями (т. к. ширина поля задана с ведущим нулем) и выравниванием вправо (поскольку ширина положительная);

[x]. `%+-10.2f` - преобразовать аргумент в дробное число (float) с явным знаком (т.к. указан +) и поместить в поле шириной в 10 цифр, из которых 2 отводятся на дробную часть, с выравниванием влево (поскольку ширина поля отрицательная).

Функция `sprintf()` часто применяется для округления чисел - например, до трех знаков в дробной части:

```
$rounded = sprintf("%.3f", 7/3); # в $rounded будет 2.333
```

Полное описание форматов с самыми разными примерами их употребления можно прочитать в официальной документации:

```
perldoc -f sprintf
```

В дополнение к функции `sprintf()` имеется функция `printf()`, которая использует тот же самый формат преобразования, но выводит отформатированный результат в указанный выходной поток.

Иногда требуется работать не со строками и словами текста, а с его отдельными символами. В Perl есть необходимые средства работы с символами, хотя в нем нет специального типа данных, представляющих один символ, подобно типу `char` в других языках. Один символ из строки можно скопировать функцией `substr($string, $index, 1)`.

С помощью заимствованных из языка Pascal функций `ord()` и `chr()` выполняются преобразования символа (а точнее односимвольной строки) в его ASCII-код и наоборот:

```
$code = ord($char); # ord('M') вернет число 77
```

```
$char = chr($code); # chr(77) вернет строку 'M'
```

```
# синоним: $char = sprintf("%c", $code);
```

Разбить строку на отдельные символы и поместить их в массив можно с помощью уже знакомой функции `split()` с пустой строкой в качестве разделителя:

```
@array_of_char = split('', $string);
```

С помощью списков и нескольких вызовов функции `substr()` можно поменять в строке местами символы с указанными индексами, например, 1 и 11:

```
$s = 'кОт видел кИта';

(substr($s, 1, 1), substr($s, 11, 1)) =

(substr($s, 11, 1), substr($s, 1, 1));

# в $s будет 'кИт видел кОта'
```

Известная по лекции о списках функция `reverse()` в скалярном контексте возвращает значение текстового выражения, в котором символы переставлены в обратном порядке, например:

```
$palindrom = 'А РОЗА УПАЛА НА ЛАПУ АЗОРА';

$backwards = reverse($palindrom);

# в $backwards будет 'АРОЗА УПАЛ АН АЛАПУ АЗОР А'
```

Обрабатывать отдельные байты, в том числе и символы, можно также при помощи функций `pack()` и `unpack()`, которые предназначены для преобразования любых данных и будут рассмотрены в лекции, посвященной вводу-выводу.

В современном мире уже не работает формула "один символ - это один байт". Необходимость представления текстов, одновременно содержащих символы разных естественных языков, привела к появлению ряда стандартов, часто объединяемых под общим названием Unicode и разработанных международным Консорциумом Unicode. Многочисленные национальные символы языков мира кодируются последовательностями из нескольких байтов. Unicode предлагает несколько форм представления символов в виде форматов преобразования Unicode (Unicode Transformation Format, UTF) и наборов символов Unicode (Unicode Character Set, UCS). Стандарты UCS-2 и UCS-4 представляют из себя кодировки фиксированной длины по два и четыре байта. Из кодировок переменной длины самым популярным стал стандарт UTF-8, использующий для кодирования одного символа от одного до шести байт. Начиная с версии 5.6, Perl поддерживает обработку символов в кодировках Unicode. В Perl применяется кодирование символов последовательностями чисел переменной длины на основе представления UTF-8. Есть возможность записывать многобайтовые (multi-byte) символы в виде литералов, а также выполнять ввод-вывод Unicode-символов.

Для записи в исходной программе символов Unicode в представлении UTF-8 нужно включить обработку строк в этом формате прагмой `use utf8`. После этого многобайтовые символы могут использоваться наравне с однобайтовыми, например, в качестве ключей в хэшах:

```
use utf8; # включить поддержку UTF-8

$hash{'

π '} = 3.141592653; # пи (код \x{03C0})

print "$hash{'

π '}\n"; # будет выведено: 3.141592653
```

Можно даже использовать национальные алфавиты для записи идентификаторов переменных. Например, кириллицу или греческий:

```
use utf8;

$скаляр = 25; # имя скаляра на русском

$

Σ = $скаляр + 53; # имя скаляра на греческом

print "$скаляр $

Σ\n"; # будет выведено: 25 78

@массив = ($
```



```
@B, $скаляр); # имя массива на русском

print "@массив\n"; # будет выведено: 78 25
```

Для ввода текста подобной программы понадобится редактор, поддерживающий работу с Unicode. Например, в операционной системе MS Windows это можно сделать с помощью программы Notepad. А в ОС GNU/Linux для редактирования этого текста можно воспользоваться редактором KWrite или Kate. Если такой возможности нет, то символы Unicode можно записывать в программе с помощью escape-последовательностей, о чем было рассказано в лекции 2. Примеры escape-кодов для записи символов Unicode приведены во фрагменте программы далее в этой лекции.

Скалярные значения в Perl имеют специальный "признак utf8" (utf8 flag), который устанавливается, когда значение представлено в UTF-8. В этом случае правильно выполняется обработка многобайтовых символов встроенными функциями chr(), index(), length(), ord(), rindex(), substr(). Это видно на таком примере:

```
use utf8;

$u = "€500"; # знак евро (escape-код \x{20AC})

print "Длина=", length($u), "\n"; # Длина=4

$u = '
∞ ≠ ∞'; # коды \x{221E}, \x{2260}, \x{221F}

print "Бесконечности не равны\n" if $u eq reverse '???';
```

Переключить встроенные функции на работу не с символами, а с байтами можно с помощью прагмы use bytes. Снова переключиться на работу функций не с байтами, а с символами можно с помощью прагмы по bytes. Подключив прагмой use Encode стандартный модуль преобразования можно преобразовать обычную строку в строку символов Unicode с помощью функции encode(), возвращающей символьную строку в представлении UTF-8. Обратное преобразование выполняет функция decode():

```
use Encode;

my $cp1251 = 'Привет!'; # строка в кодировке windows-1251

my $utf8 = encode('utf8', $cp1251); # преобразуется в UTF-8

my $win_ru = decode('utf8', $utf8); # и наоборот
```

Поддержка наборов символов Unicode в Perl имеет свои особенности, связанные с обеспечением совместимости со старыми байт-ориентированными программами, но эти особенности заслуживают отдельного продолжительного разговора за рамками данного учебного курса.

В этой лекции рассмотрены средства работы с символьной информацией в Perl, достаточные для решения типичных задач обработки текста. Но вся прелесть языка Perl и его мощь открываются только тем, кто освоит регулярные выражения, о которых пойдет речь в следующей лекции.

## Лекция 8. Регулярные выражения

В этой лекции будет кратко рассказано о мощном средстве обработки текста - регулярных выражениях, эффективной поддержкой которых всегда славился язык Perl. Регулярные выражения - это отдельный язык для работы с текстовой информацией, который встроен в язык Perl так, что пользоваться ими можно легко и удобно.

Цель лекции: познакомиться с правилами описания регулярных выражений и со средствами работы с ними в Perl. Научиться применять регулярные выражения для поиска, извлечения и замены текстовой информации.

Когда некоторые говорят, что программы на Perl похожи на бессмысленную кучу символьного мусора, то это впечатление, скорее всего, возникло от вида какого-нибудь длинного регулярного выражения, а они действительно могут выглядеть как загадочный или бессмысленный набор символов, например:

```
m/[+]?HREF\s*=\s*["']?([^\s">]+)?["']?\s*>/ig
```

(Это всего-навсего шаблон для поиска гиперссылок в HTML-странице.) Но в этой лекции вы узнаете, что регулярные выражения - совсем не ужасные и отнюдь не хаотичные, а наоборот, очень даже логичные и упорядоченные, что употреблять их не так уж сложно, а записывать их можно вполне наглядным способом. Как сказал Джеффри Фридл в своей знаменитой книге, переведенной на русский язык: "Регулярные выражения также можно сравнить с иностранным языком - когда вы начинаете изучать язык, он перестает казаться белибердой".

Начнем с того, что регулярные выражения (regular expression, сокращенно - regexpr, regex или RE) - это отдельный язык описания образцов для обработки текста, не имеющий непосредственного отношения к Perl. Регулярные выражения использовались в Unix задолго до создания Perl, а сейчас библиотеки для работы с ними имеются в C++, C#, Java, JavaScript, PHP, Python, Ruby, Visual Basic и других языках. Поддержка регулярных выражений есть в некоторых редакторах, почтовых программах и системах управления базами данных. Другое дело, что широкое распространение Perl в свое время сделало регулярные выражения популярными на разных платформах. А в ходе развития языка Perl была отточена система обозначений для регулярных выражений, ставшая фактическим стандартом. Многие считают, что благодаря Perl регулярные выражения из математической теории превратились в рабочий инструмент тысяч и тысяч программистов. Это произошло потому, что в Perl механизмы работы с регулярными выражениями встроены в ядро языка, поэтому применять их естественно, легко и удобно. А благодаря эффективной реализации "движка" регулярных выражений, в Perl они обрабатываются чрезвычайно быстро. Регулярные выражения выполняют львиную долю работ по обработке текстовой информации и используются в Perl несколькими способами:

[x]. для поиска в тексте строк по определенному образцу;

[x]. для разделения текста на части по указанному набору разделителей;

[x]. для извлечения из строки подстрок, соответствующих заданному шаблону;

[x]. для замены в тексте найденных соответствий на новые значения.

Основная идея регулярных выражений состоит в нахождении в тексте соответствия определенному образцу, который может задаваться литералом или шаблоном. Вся текстовая строка считается соответствующей образцу, если ему соответствует какая-то ее часть.

Наверное, чаще всего регулярные выражения используются в операции сопоставления (match operator), которая проверяет, соответствует ли текст указанному образцу. Образец (pattern) - это символьная последовательность для сопоставления, записанная в специальной нотации. Простейший образец - это строковый литерал, представляющий собой последовательность символов, которая будет отыскиваться в тексте. В скалярном контексте операция сопоставления возвращает '1', если образец в строке найден, и пустую строку '', если соответствие образцу не найдено. Для указания, к какой строке применить операцию сопоставления, используется операция привязки =~ к строке:

```
'В строке образец есть' =~ /образец/; # образец найден
```

Обычно поиск образца выполняется с учетом регистра, но можно игнорировать регистр при сопоставлении строки с образцом, если в операции сопоставления задать модификатор /i (ignore case). Для корректной обработки национальных букв должна быть включена прага use locale. Например:

```
use locale;
```

```
'В строке образец есть' =~ /Образец/; # образец НЕ найден!
```

```
'В строке образец есть' =~ /Образец/i; # образец найден
```

Результат операции сопоставления в тексте можно присвоить скалярной переменной или использовать в любой из условных конструкций, например:

```
$text = 'Черный кот в темной комнате'; # ищем в этом тексте
```

```
$found = $text =~ /кот/; # в $found будет '1'
```

```
print 'Кошки нет!' unless $text =~ /кошка/; # вернет ''
```

Последнее предложение можно переписать, применив операцию отрицательной привязки к строке (!~), которая инвертирует (меняет на обратный) результат операции сопоставления:

```
print 'Кошки нет!' if $text !~ /кошка/; # вернет '1'
```

Если операция привязки к строке не используется, образец отыскивается в переменной по умолчанию `$_`. Выражение перед поиском интерполируется, поэтому весь образец поиска или его часть может содержаться в переменной. Например:

```
$_ = 'Счастье - это когда тебя понимают.'; # переменная поиска

$pattern = 'Счастье'; # образец для сопоставления

print "$pattern найдено!" if /$pattern/;
```

В составе образца поиска могут применяться не только переменные, но и escape-последовательности, известные нам из лекции 2, например:

```
print 'В строке обнаружена табуляция' if $string =~ m{\t};
```

Для успешного сопоставления строки образцу достаточно найти в строке первое совпадение. В этом примере образец совпадет с началом подстроки 'которого':

```
$text = 'У которого из котов зеленые глаза?'; # ищем здесь

$any = $text =~ /кот/; # образец совпал с 'которого'
```

Чтобы найти именно подстроку 'кот', перед которой стоит пробел, нужно задать более точный образец для сопоставления:

```
$cat = $text =~ / кот/; # образец совпадет с ' кот'
```

В операции сопоставления программист может задавать ограничители для образца: в этом случае перед ограничителями указывается буква `m//` (Операцию сопоставления часто именно так и называют: операция `m//`.) В качестве ограничителей могут выступать различного вида скобки или парные небуквенные символы, например:

```
m($pattern) m{$pattern} m[$pattern] m<$pattern>

m|$pattern| m!$pattern! m"$pattern" m#$pattern#
```

Задать собственные ограничители бывает особенно полезно, когда в шаблон поиска входит наклонная черта. Из двух приведенных вариантов второй смотрится гораздо понятнее:

```
/\usr\bin\perl/

m{/usr/bin/perl}
```

Недаром обилие левых и правых наклонных черт в первом варианте называют "ученическим синдромом зубочисток" (LTS - Learning Toothpick Syndrome). В приводимых до сих пор примерах операцию сопоставления с литералом в качестве образца вполне можно заменить вызовом функции `index()`. Самое интересное начинается тогда, когда в образце поиска применяются метасимволы для сопоставления с шаблоном.

Очень часто требуется искать в тексте не конкретные строки, а символьные последовательности, определенные приблизительно: "число в скобках", "четвертое слово с начала строки", "список из пар имя = значение, разделенных запятыми" и тому подобное. В таких случаях в качестве аргумента поиска задается шаблон, который описывает такую последовательность. Шаблон - это образец, в котором, помимо литеральных значений, содержатся метасимволы. Метасимволы (metacharacter) - это знаки, имеющие специальное значение при записи образцов. Вот какие метасимволы применяются при записи регулярных выражений:

```
{ } [ ] ( ) ^ $ . | * + ? \
```

При необходимости включить в образец поиска один из этих знаков не как метасимвол, а как обыкновенный символ, нужно отменить его особое значение ("экранировать"), поставив перед ним обратную косую черту (backslash):

```
$text =~ m"\." # содержится ли в тексте точка?
```

Как метасимвол точка обозначает в регулярном выражении один любой символ, кроме знака перевода новой строки (`\n`). Например, для поиска похожих слов можно составить такой шаблон:

```
/само.а./ # соответствуют: 'самовар', 'самокат', 'самосад'...

# НЕ соответствуют: 'самолюб', 'самогон', 'самоход'...
```

В регулярном выражении можно задать несколько вариантов образца, любой из которых будет считаться соответствием строки образцу. Варианты образца - это набор возможных альтернатив, разделенных знаком "вертикальная черта" ('|'), который называется "метасимвол альтернатив" (alternation metacharacter). Поиск считается успешным, если найдено соответствие любой из альтернатив, например:

```
$text = 'Черная кошка в темной комнате'; # будем искать здесь

print "Нашли кошку!" if $text =~ /кот|кошка|котенок/;
```

Сравнение текста с вариантами образца выполняется слева направо, поэтому, если начало альтернатив совпадает, более длинную альтернативу нужно помещать в начало списка вариантов. Иначе всегда будет найдена более короткая. Значит шаблон в предыдущем примере правильнее записать в виде /котенок|кот|кошка/, чтобы в первую очередь поискать котенка, а затем - кота:

```
$text = 'Черный котенок в темной комнате'; # ищем здесь

print "Нашли котенка!" if $text =~ /кот.нок|кот|кошка/;
```

Чтобы сделать образец более универсальным, в первой альтернативе литерал заменен на шаблон с метасимволом "точка", чтобы находились соответствия слову "котенок" в любом написании - через "е" и через "ё". Часто применение регулярного выражения с альтернативами выглядит гораздо изящнее, чем длинное условное выражение:

```
return if $command =~ /exit|quit|stop|bye/i;
```

Если в образце после выбора из нескольких альтернатив применяются другие шаблоны или литералы, то конструкцию выбора нужно заключить в круглые группирующие скобки. Например:

```
$lotr =~ /(Bilbo|Frodo) Baggins/; # один из хоббитов
```

С помощью метасимволов можно обозначить в шаблоне один символ из заданного набора. Для этого нужно определить класс символов, указав в квадратных скобках набор символов, включаемых в класс. Классы символов похожи на шаблон с вариантами, в котором альтернативами могут быть только отдельные символы. Ради примера запишем шаблон для слов, отличающихся первой буквой из указанного набора:

```
/[вклрт]от/ # соответствуют: 'вот', 'кот', 'лот', 'рот', 'тот'
```

Вот пример шаблона с несколькими классами символов, каждый из которых представляет одну букву в последовательности из четырех символов:

```
/[мс][ул][хо][ан]/ # соответствуют: 'муха', 'слон'
```

# а также: 'суоа', 'млхн', 'слоа' и так далее

В классе символов вместо перечисления можно указывать диапазон от начального до конечного символа, разделенных минусом:

```
[0-9] вместо [0123456789]
```

```
[A-Z] вместо [ABCDEFGHIJKLMNOPQRSTUVWXYZ]
```

Указывая несколько диапазонов в одном классе, запишем шаблон для шестнадцатеричной цифры:

```
/[0-9a-fA-F]/# соответствуют: '5', 'b', 'D' и так далее
```

Чтобы включить в символьный класс знак '-', нужно поместить его в начале или в конце перечисленных в классе символов или экранировать обратной чертой. Помещенные в символьный класс, все метасимволы (кроме '|') рассматриваются как обычные символы. Поэтому так могут выглядеть шаблоны для поиска знака препинания или одной из скобок:

```
[-.,;:!?] # знаки препинания
```

```
[(){}\{\}] # скобки: \] представляет скобку ']'
```

Иногда требуется выразить понятие "все, кроме указанных символов": для этого в описании класса символов сразу после открывающей квадратной скобки ставится метасимвол отрицания ('^'). Например, так можно записать шаблоны для "любого символа, кроме знаков препинания" или "любого нецифрового символа":

`[^-.,:;!~] # все, кроме этих знаков препинания`

`^[0-9] # не цифры`

Чтобы включить в символьный класс символ '^', нужно поставить его не первым в списке символов или отменить его специальное значение с помощью символа '\\':

`[*^] или так: [\\^]`

Для сокращенной записи классов символов в регулярных выражениях предусмотрены специальные обозначения, состоящие из латинской буквы с обратной косой чертой перед ней. Вот они:

`\\d` - любая десятичная цифра, то есть `[0-9]`

`\\D` - любой символ, кроме цифры: `^[0-9]` или `[^\\d]`

`\\w` - символ, пригодный для записи идентификатора: `[a-zA-Z0-9_]`

`\\W` - противоположность символа `\\w`, то есть `[^\\w]`

`\\s` - пробельный символ: пробел, `\\t`, `\\n`, `\\r` или `\\f`

`\\S` - любой не пробельный символ, то есть `[^\\s]`

С помощью этих метасимволов можно составлять гораздо более интересные образцы. Например, проверим, содержится ли в тексте число из четырех цифр, окруженное любыми пробельными символами:

```
$text = "Альбом 'Dire Straits'\\tГод 1978\\tВремя 41:21";
```

```
$text =~ m{\\s\\d\\d\\d\\d\\s}; # найдет ' 1978\\t'
```

Записывать несколько метасимволов подряд для указания в шаблоне последовательности из однотипных символов утомительно и неудобно, да и ошибиться при этом легко. Облегчить жизнь составителям регулярных выражений помогают квантификаторы.

Квантификатор (quantifier) - это обозначение числа повторений предыдущего шаблона при поиске соответствия. Количество повторений может задаваться одним или парой десятичных чисел в фигурных скобках:

`{n}` повторяется точно n раз

`{n,}` повторяется n и более раз

`{n,m}` повторяется от n до m раз включительно

Квантификатор, также иногда называемый множителем, указывается сразу после конструкции в шаблоне, которую нужно повторить несколько раз, например:

`/\\d{5}/` # ровно пять цифр, то есть: `\\d\\d\\d\\d\\d`

`/\\s{1,}/` # один и более пробельных символов

`/[A-Z]{1,8}/` # от 1 до 8 заглавных латинских букв

Опишем с применением квантификаторов шаблон для поиска в тексте последовательности, похожей на телефонный номер, в следующем формате:

символ + `\\+`

код страны: не менее 1 цифры `\\d{1,}`

открывающая скобка ( `\\(`

код города: 3 цифры и более `\\d{3,}`

закрывающая скобка ) `\\)`

номер абонента: от 4 до 7 цифр `\d{4,7}`

Перед знаками "+", "(" и ")" ставится обратная наклонная черта, чтобы они не воспринимались как метасимволы. Вот какое регулярное выражение получится в результате:

```
m"\+\d{1,}\(\d{3,}\)\d{4,7}"
```

Для наиболее часто встречающихся квантификаторов предусмотрены удобные односимвольные сокращения:

\* повторяется 0 или более раз: то же, что `{0,}`

? повторяется не более 1 раза: то же, что `{0,1}`

+ повторяется как минимум 1 раз: то же, что `{1,}`

Составим регулярное выражение с использованием односимвольных квантификаторов, чтобы найти в тексте "идентификатор, перед которым могут стоять пробельные символы и за которым стоит хотя бы один из перечисленных знаков препинания":

```
m/\s*\w+[-.,;?]+/ # соответствует, например: ' count--;'
```

Если квантификатор нужно применить к нескольким шаблонам, то нужно сгруппировать шаблоны, заключив их в круглые скобки. Составим регулярное выражение для поиска IP-адреса, которое находит число, состоящее из одной цифры и более (`\d+`), за которой может стоять точка (`\.`), причем эта последовательность повторяется ровно четыре раза (`{4}`):

```
$pattern = '(\d{1,3}\.){4}\d{1}'; # шаблон для IP-адреса
```

```
$text = 'address=208.201.239.36,site=www.perl.com';
```

```
$text =~ m/$pattern/; # соответствие: '208.201.239.36'
```

Программисты шутят: "При составлении шаблона главное, чтобы регулярное выражение соответствовало тому, что нужно, и не соответствовало тому, что не нужно". В следующем примере мы будем искать "более одного символа, за которыми идет буква 'й' и пробел", ожидая, что будет найдено слово 'Какой '. Но нас ожидает неприятный сюрприз:

```
my $text = 'Какой хороший компакт-диск!';
```

```
$text =~ /.+й\s/; # жадный квантификатор
```

```
# найдено соответствие: 'Какой хороший '
```

Это произошло потому, что по умолчанию квантификаторы подразумевают максимальную последовательность символов, соответствующих указанному шаблону. Такое поведение квантификаторов называется "жадным" (greedy quantifier). Чтобы заставить квантификатор вести себя не "жадно", а "лениво" (lazy quantifier), нужно поставить сразу после него символ '?'. Тогда квантификатор будет описывать минимальную последовательность символов, соответствующих образцу. Исправленный с учетом этого образец найдет то, что нужно:

```
$text =~ /.+?й\s/; # ленивый квантификатор
```

```
# найдено соответствие: 'Какой '
```

Таким же образом можно ограничивать "жадность" и других квантификаторов, заставляя их прекращать поиск как можно раньше, что обычно и требуется в большинстве ситуаций.

Часто нам бывает небезразлично, в каком месте содержимое строки совпадет с шаблоном. Мы бы хотели уточнить: "в начале строки", "в конце слова" и так далее. Для того чтобы более точно задать положение в тексте, где должно быть найдено соответствие, в регулярных выражениях можно указывать так называемые утверждения. Утверждение (assertion) не соответствует какому-либо символу, а совпадает с определенной позицией в тексте. Поэтому их можно воспринимать как мнимые символы нулевого размера. Чаще всего используются следующие утверждения (другие приведены в таблице 8.1):

^ позиция в начале строки

\$ позиция в конце строки (или перед `\n` в конце строки)

\b граница слова: позиция между `\w` и `\W` или `\W` и `\w`

\B любая позиция, кроме границы слова \b

Вот пример шаблонов поиска, где уточняется, что нужно проверить наличие числа в определенном месте строки:

```
$log = '20060326 05:55:25 194.67.18.73 ... 200 797';
```

```
print "Число в начале\n" if $log =~ /^\\d+/;
```

```
print "Число в конце\n" if $log =~ /\\d+$/;
```

Утверждение, которое используется для фиксирования части образца относительно положения в строке, иногда называется якорем (anchor). Якори применяются, чтобы указать, в каком именно месте строки нужно искать соответствие образцу.

Когда операция сопоставления находит в строке соответствие указанному регулярному выражению, она присваивает результаты своей работы нескольким специальным переменным:

[x]. в переменную \$` помещается часть строки до найденного соответствия;

[x]. в переменную \$& помещается часть строки, соответствующая образцу;

[x]. в переменную \$' помещается часть строки после найденного соответствия;

[x]. в переменную \$+ помещается последнее найденное совпадение для последнего шаблона в скобках.

Если поиск окончился неудачей, то этим переменным новые значения не присваиваются. Посмотрим на примере, что сохранится в этих переменных после поиска такого соответствия:

```
$htm = "Регулярные выражения";
```

```
$htm =~ m|HREF=("['](\\S+?)["]>|; # поиск URL сайта
```

При успешном совпадении с шаблоном в специальные переменные будут помещены такие значения:

```
$` = '
```

```
$& = 'HREF='http://regex.ru/'>'
```

```
$' = 'Регулярные выражения'
```

```
$+ = 'http://regex.ru/'
```

Значениями этих переменных можно пользоваться при успешном сопоставлении с образцом, например:

```
print $& if $text =~ m/$pattern/; # выведет соответствие
```

В регулярном выражении можно указать, что при успешном сопоставлении строки с шаблоном найденные соответствия нужно сохранить для дальнейшей обработки. С этой целью записываемые части шаблона нужно заключить в круглые скобки. Это также называется захватом значений. Найденные совпадения для всех заключенных в скобки частей шаблона будут доступны через специальные переменные с именами \$1, \$2 и так далее. Составим регулярное выражение для поиска и сохранения в служебных переменных информации о сайте в том же тексте:

```
$pattern = q|HREF=("['](\\S+?)["]>([\\^<]+?)|; # шаблон
```

```
$htm =~ m/$pattern/; # поиск соответствия в $htm
```

```
# в $1 = 'http://regex.ru/'
```

```
# в $2 = 'Регулярные выражения'
```

Сохраненные совпадения доступны и во время обработки регулярного выражения, но через переменные с именами \\1, \\2 и так далее. Эти переменные называются обратными ссылками (backreference) на найденные соответствия. Так, например, можно найти два одинаковых слова, стоящих в тексте друг за другом через пробелы (возможно, по ошибке):

```
my $string = "Уже скоро скоро наступит весна!";
```

```
my $pattern = '(\S+)\s+\1';

# (\S+) сохранит значение 'скоро' в \1

$string =~ m/$pattern/; # соответствие: 'скоро скоро'
```

Операция сопоставления, употребленная в списочном контексте, возвращает список найденных соответствий, для которых было предусмотрено сохранение значений. Поэтому удобно сохранять найденные значения в массиве или в списке скалярных переменных. Например, извлечем из текстовой строки последовательность цифр, похожую на время:

```
my $text = 'Начало в 12:25:00.'; # строка с данными

my $pattern = '(\d\d):(\d\d):(\d\d)'; # образец для поиска

my @time = $text =~ m/$pattern/; # сохраним в массиве

my ($hh, $mm, $ss) = $text =~ m/$pattern/; # и в списке
```

Можно находить любое количество соответствий образцу в одной операции сопоставления. Это делается с помощью модификатора глобального поиска.

До сих пор операция сопоставления прекращала работу и возвращала результат, когда находилось первое соответствие строки указанному шаблону. Если для операции сопоставления указать модификатор /g (global), то она будет искать в строке все соответствия образцу, организуя неявный цикл обработки регулярного выражения. Например, так можно найти все числа в строке с помощью одного шаблона:

```
my @numbers = 'He 12.5, a 25!' =~ /(\d+)/g; # глобальный поиск

# в @numbers будет (12, 5, 25)
```

Ранее в этой лекции уже упоминался модификатор /i, устанавливающий поиск с игнорированием разницы между заглавными и строчными буквами. Перечислим модификаторы для операции сопоставления:

```
[x]. /g - искать в тексте все соответствия образцу (Global);

[x]. /i - искать соответствие образцу без учета регистра букв (case-Insensitive);

[x]. /s - рассматривать текст как одну строку (Single-line);

[x]. /m - рассматривать текст как многострочный (Multi-line) с учетом \n ;

[x]. /o - один раз откомпилировать регулярное выражение (Once);

[x]. /x - использовать расширенный синтаксис регулярных выражений (eXtended).
```

Из всех модификаторов, пожалуй, самый интересный - последний, который позволяет записывать регулярные выражения в структурированном и понятном для человека виде и даже сопровождать комментариями! Так, например, можно более понятно и красиво переписать регулярное выражение, приведенное в начале лекции:

```
m/ # начало регулярного выражения

[^>]+? # далее могут быть любые символы, кроме >

HREF # определение гиперссылки

\s*=\s* # знак =, возможно окруженный пробелами

['"]? # может быть открывающая кавычка или апостроф

( # начало захвата значения

[\"'\" >]+? # адрес ссылки: все, кроме ', ", пробела и >
```



```
) # конец захвата значения

[""]? # может быть закрывающая кавычка или апостроф

\s* # за которым могут быть пробелы

> # конец тега

/ix; # конец регулярного выражения
```

# соответствует, например:

Записанное в таком виде, регулярное выражение становится доступным для понимания, анализа и модификации. А поскольку регулярные выражения компилируются, то пробельные символы и комментарии не влияют на быстродействие программы.

Кроме поиска, регулярные выражения часто применяются для замены найденных совпадений на новые значения. Для этого существует операция замены (substitution), которая пытается найти в строковой переменной соответствие образцу, а если находит, то заменяет найденную подстроку на указанное значение. Операция замены выглядит так:

```
$variable =~ s/образец/замена/;

# в переменной $variable отыскивается строка 'образец',

# и если найдена, то она заменяется на 'замена'
```

Все, что говорилось до этого про операцию сопоставления, применимо для левой части операции замены, в которой указывается образец поиска. Левая и правая части операции замены интерполируются, поэтому там могут использоваться escape-последовательности и переменные.

```
$pattern = 'шило'; # образец

$replacement = 'мыло'; # замена

$text =~ s/$pattern/$replacement/; # поменять 'шило' на 'мыло'
```

В правой части операции замены могут использоваться обратные ссылки на найденные значения. Так, например, можно поменять местами два крайних слова в тройке слов, разделенных пробельными символами:

```
$text = 'мать любит дочь';

$text =~ s/(\S+)\s+(\S+)\s+(\S+)/\3 \2 \1/;

# в $text будет 'дочь любит мать'
```

Для операции замены `s///` можно применять все модификаторы, упомянутые для операции сопоставления `m//`. Например, модификатор `/g` указывает, что должны быть заменены все найденные в тексте соответствия. Например:

```
$our_computers =~ s/Windows/Linux/g;
```

У операции замены есть дополнительный модификатор `/e` (expression evaluation), при включении которого заменяющая часть вычисляется как выражение. При этом в заменяющей части можно использовать ссылки на захваченные при помощи круглых скобок соответствия. Это можно применять для более "интеллектуальной" замены найденных соответствий. Так, например, можно перевести температуру из шкалы Цельсия в шкалу Фаренгейта:

```
$text = 'Бумага воспламеняется при 233C.';

$text =~ s/(\d+\.?\d*)C\b/int($1*1.8+32).'F'/e;

# в $text будет: 'Бумага воспламеняется при 451F.'
```

Регулярные выражения применяются во многих конструкциях. В функции `split()` первым параметром может использоваться регулярное выражение, которое будет служить для поиска разделителей при разделении строки на части. Так, например, можно разбить строку на подстроки по любому из пробельных символов:

```
@substrings = split /\s+/, $text; # разбить на части
```

Регулярные выражения часто применяются в функциях, работающих с массивами для фильтрации нужных элементов. Например, функция `grep()` возвратит список элементов массива, соответствующих указанному образцу:

```
@result = grep /$pattern/, @source; # отобрать элементы
```

С помощью функции `map` можно применить операцию замены в соответствии с регулярным выражением ко всем элементам массива, например:

```
@hrefs = ('http://regex.info', 'http://regexp.ru');

map s{http://}{}, @hrefs; # убрать 'http://' из ссылок
```

Регулярные выражения дают программисту новый взгляд на текстовые данные: вместо отдельных символов и простых подстрок он начинает мыслить обобщенными шаблонами, что помогает ему находить более простые и эффективные решения. В таблице 8.1 для справки приведены основные обозначения, применяемые для записи регулярных выражений в Perl. Дополнительные сведения о регулярных выражениях можно почерпнуть из стандартной документации по Perl и перевода уникальной книги Джеффри Фридля "Регулярные выражения".

Таблица 8.1. Основные обозначения для записи регулярных выражений

| Обозначение | Описание   | Примеры          |
|-------------|--|------------------|
| //          | ограничители регулярного выражения по умолчанию                                      | /\$pattern/      |
| \           | отмена специального значения следующего символа                                      | m{C:\\windows}   |
| ()          | группировка шаблонов или сохранение значения   | /(\w\w\w)+/      |
|             | выбор из нескольких альтернатив  | /кошелек жизнь/  |
| []          | класс символов: любой символ из перечисленных  | /[0-9a-fA-F]/    |
| [^]         | инвертированный класс символов: любой символ, кроме перечисленных                    | /[^0-9]/         |
| Метасимволы |  |                  |
| .           | любой символ, кроме \n (соответствует любому символу, включая \n с модификатором /s) | /(.+)/           |
| \d          | десятичная цифра   | m{Время=\d+ сек} |
| \D          | не десятичная цифра  | /(\D*)\d+/       |
| \w          | алфавитно-цифровой знак  | /\s+\w+\s+/      |
| \W          | не алфавитно-цифровой знак   | /\W\W\W/         |
| \s          | пробельный символ  | s/\s+/ /         |
| \S          | любой символ, кроме пробельного  | /\S+/            |
| Утверждения |  |                  |
| ^           | начало строки (соответствует началу каждой строки с модификатором /m)                | /^\w+/           |
| \$          | конец строки (соответствует концу каждой строки с модификатором /m)                  | /\d+\$/          |
| \b          | граница слова (между \w и \W или \W и \w)  | /stop\b/         |
| \B          | любая позиция, кроме границы слова   | /stop\B/         |

|                                |   |  |
|--------------------------------|---|--|
| <code>\A</code>                | только начало строки, даже с модификатором <code>/m</code>  | <code>/\A[#]/</code>                     |
| <code>\z</code>                | только конец строки, даже с модификатором <code>/m</code>   | <code>/\w+\z/</code>                     |
| <code>\Z</code>                | только конец строки или перед <code>\n</code> в конце строки, даже с модификатором <code>/m</code>                                      | <code>/\w+\Z/</code>                     |
| <code>\G</code>                | позиция в строке, равная значению функции <code>pos()</code>  |  |
| Escape-последовательности      |   |  |
| <code>\t \n \r \f \a \b</code> | управляющие символы: <code>\b</code> в классе символов выступает как символ <code>Backspace (0x08)</code> , вне его - как граница слова | <code>/[\a\b\f\r\n\t]/</code>            |
| <code>\0 \x \c \N</code>       | коды символов   | <code>/\033\x1F\cZ/<br/>\x{263a}/</code> |
| <code>\l \L \u \U \Q \E</code> | преобразующие последовательности  | <code>/\Q\$pattern\E/</code>             |
| Квантификаторы                 |   |  |
| <code>* *?</code>              | любое число повторений, включая 0 (максимальный и минимальный квантификаторы)   | <code>/\s*/ \S*?/</code>                 |
| <code>+ +?</code>              | одно и более повторений (максимальный и минимальный квантификаторы)   | <code>/\d+/ \D+?/</code>                 |
| <code>? ??</code>              | ноль или одно повторение (максимальный и минимальный квантификаторы)  | <code>/./? /[.a-z]?/?</code>             |
| <code>{n} {n}?</code>          | ровно n повторений (максимальный и минимальный квантификаторы)  | <code>/\w{8}/ \w{5}?/</code>             |
| <code>{n,} {n,}?</code>        | n и более повторений (максимальный и минимальный квантификаторы)  | <code>/\d{2,}/ \d{5,}?/</code>           |
| <code>{n,m} {n,m}?</code>      | от n до m повторений включительно (максимальный и минимальный квантификаторы)   | <code>/[A-Z]{1,12}/ [a-z]{0,3}?/</code>  |

Из этой лекции вы узнали о регулярных выражениях далеко не все, но достаточно, чтобы начать свободно пользоваться ими. По мере накопления опыта применения языка Perl, регулярные выражения станут вашим привычным и надежным инструментом. И тогда задачи, при другом подходе требующие много времени и больших усилий, с помощью регулярных выражений будут решаться быстро, элегантно и эффективно.

## Лекция 9. Средства ввода-вывода

В этой лекции разбирается организация ввода-вывода данных в Perl. Рассмотрены средства работы с каталогами, файлами и содержимым файлов. Материалы этой лекции позволят вам писать полноценные программы, "общающиеся с внешним миром".

Цель лекции: познакомиться с возможностями подсистемы ввода-вывода в Perl и освоить основные приемы чтения и записи внешних данных, а также научиться работать со средствами манипулирования файлами и каталогами.

Система ввода-вывода Perl основана на принципах, заложенных в системе Unix и распространившихся на все современные операционные системы. Одним из основных понятий работы в программе с внешними данными являются потоки ввода-вывода. В программе обращение к потоку ввода-вывода производится через файловый манипулятор (file handle), иногда неправильно называемый дескриптором файла. При запуске любой программы автоматически открывается три потока: стандартный ввод (stdin), стандартный вывод (stdout) и стандартный протокол (stderr). Поток стандартного ввода в диалоговой операционной среде связывается с клавиатурной, а потоки стандартного вывода и стандартного протокола - с дисплейной частью консоли операционной системы. Со стандартными потоками в Perl связываются три предопределенных файловых манипулятора: соответственно STDIN, STDOUT и STDERR. Связывание имени файла с пользовательским файловым манипулятором в программе выполняется с помощью операции `open()`, открывающей поток обмена данными с указанным файлом. Требования надежности рекомендуют обязательно проверять все операции ввода-вывода на успешное завершение. Поэтому в случае возникновения ошибки при открытии файла программа обычно аварийно завершается с помощью функции `die()`, которая может выводить диагностическое сообщение в стандартный поток протокола. Например, так открывается файл и создается файловый манипулятор `FILE_HANDLE`:

```
# открыть для чтения файл по имени, взятом из $file_name

open(FILE_HANDLE, $file_name)

# или аварийно завершить программу с выдачей сообщения

or die("Ошибка открытия файла $file_name: $!\n"); #
```

В случае успешного открытия файла функция `open()` помещает в свой первый аргумент готовый к использованию файловый манипулятор. Имя файлового манипулятора записывается без разыменовывающего префикса и по традиции выделяется заглавными буквами. Рекомендуется при открытии файла сохранять файловый манипулятор в скалярной переменной, что позволяет локализовать файловый манипулятор для использования только в текущем блоке или подпрограмме. Кроме того, скалярную переменную с файловым манипулятором можно удобно передавать в подпрограммы для выполнения в них операций ввода-вывода. Итак, вот предпочтительный способ открытия файла:

```
open my $file_handle, $file_name

or die "Ошибка открытия файла $file_name: $!\n";
```

(Как и при вызове других функций в Perl, если не возникает неоднозначности, программист решает, заключать аргументы функций в круглые скобки или нет. Среди пишущих на Perl широко распространен стиль программирования без использования круглых скобок.)

При открытии файла функции, помимо файлового манипулятора и имени файла в файловой системе (абсолютного или относительного), указывается режим открытия файла. Он обозначается такими же символами, как переназначение потоков ввода-вывода в командном интерпретаторе операционной системы. Основные режимы открытия потоков ввода-вывода приведены в таблице 9.1.

Таблица 9.1. Основные режимы открытия потоков ввода-вывода

| Обозначение | Режим открытия                             | Пример использования                                 |
|-------------|--|--|
| <           | Чтение (существующего файла с начала)      | <code>open(\$fh, '&lt;/temp/buffer.txt')</code>      |
| >           | Перезапись (с начала файла)                | <code>open(\$fh, '&gt;/temp/buffer.txt')</code>      |
| >>          | Дозапись (в конец файла)                   | <code>open(\$fh, '&gt;&gt;/temp/buffer.txt')</code>  |
| +<          | Чтение и запись (файл должен существовать) | <code>open(\$fh, '+&lt;/temp/buffer.txt')</code>     |
| +>          | Запись и чтение (файл усекается)           | <code>open(\$fh, '+&gt;/temp/buffer.txt')</code>     |
| +>>         | Дозапись и чтение                          | <code>open(\$fh, '+&gt;&gt;/temp/buffer.txt')</code> |

Применяются две формы записи функции `open()`: старая с двумя аргументами, когда режим открытия указывается перед именем файла, и новая - с тремя аргументами, в которой режим открытия указывается отдельно вторым параметром. Сравните:

```
open $fh, '  
  
open $fh, '<', '/temp/buffer.txt';
```

Программисты, знающие язык C, могут воспользоваться для открытия потоков функцией `sysopen()`, которая аналогична функции открытия потоков в C, и к тому же позволяет более тонко настраивать режимы открытия файлов. А вообще в комплекте с Perl идет целый учебник по функции `open()`, который можно прочитать утилитой чтения документации:

```
perldoc perlopenut
```

Связь файлового манипулятора в программе с обрабатываемым файлом разрывается функцией закрытия потока `close()`, закрывающей поток ввода-вывода. Ей передается файловый манипулятор открытого файла:

```
close(FILE) or die("Ошибка при закрытии файла: $!\n");
```

```
close $handle or die "Ошибка закрытия файла: $!\n";
```

В новой многоуровневой подсистеме ввода-вывода Perl предусматривает при открытии потока вместе с режимом открытия указывать кодировку читаемых и записываемых данных, что позволяет автоматически преобразовывать информацию из одной кодировки в другую, например, так:

```
open my $in, "<:encoding(UTF-8)", 'utf8.txt' or die;
```

```
open my $out, ">:encoding(cp1251)", 'cp1251 .txt' or die;
```

```
while(<$in){ print $out $_; }
```

```
close $in or die;
```

```
close $out or die;
```

В некоторых операционных системах (например, в MS-DOS), после открытия файла, но перед чтением двоичных данных требуется установить для файлового манипулятора режим работы с двоичными данными с помощью функции `binmode($file_handle)`. При работе в операционных системах, не требующих установки этого режима, вызов функции `binmode()` не оказывает никакого действия.

В огромном числе случаев ввод данных в Perl-программу и вывод из нее результатов производится построчно, а для разделения строк файла используется разделитель входных записей, хранящийся в специальной переменной `$/` (`$INPUT_RECORD_SEPARATOR`). Для чтения одной строки из входного потока используется операция "кристалл" (`diamond`), которой в качестве аргумента передается файловый манипулятор или переменная, содержащая манипулятор. Если аргумент не указан, данные читаются из стандартного входного потока.

```
$input = <>; # чтение строки в $input из STDIN
```

```
$line = ; # чтение строки в $line из потока FILE
```

```
$in = <$handle>; # чтение строки в $in из потока $handle
```

Операция чтения "кристалл" в скалярном контексте возвращает одну строку вместе с разделителем записей, а когда достигается конец файла, она возвращает неопределенное значение `undef`, которое воспринимается как ложное. Поэтому типичный цикл построчного чтения данных проверяет прочитанное значение и заканчивается, когда оно становится неопределенным:

```
open my $fh, "< $file" or die "Ошибка открытия: $!";
```

```
while (my $line = <$fh>) { # чтение строки в переменную $line
```

```
chomp $line; # удаление разделителя строк
```

```
print length $line, " $line\n"; # обработка строки
```

```
}
```

```
close $fh or die "Ошибка закрытия: $!";
```

Операция чтения "кристалл" в списочном контексте возвращает список всех строк с разделителями записей. Так, например, можно считать файл в массив, попутно отсортировав его:

```
@lines= sort(<$fh>); # в @lines отсортированные строки из $fh
```

Построчный вывод данных выполняет функция `print()`, которая по умолчанию выводит список значений в текущий поток вывода, по умолчанию - в `STDOUT`. Если требуется направить информацию в другой поток, то перед списком выводимых данных указывается файловый дескриптор. Обратите внимание, что между файловым дескриптором и списком выводимых значений запятая не ставится. Вот примеры вывода данных:

```
print($list, $of, $output, $values); # вывод в STDOUT
```

```
print STDOUT $list, $of, $output, $values; # вывод в STDOUT
```

```
print(STDERR $list, $of, $output, $values); # вывод в STDERR
```

```
print FILE $list, $of, $output, $values; # вывод в FILE

print($file $list, $of, $output, $values); # вывод в $file
```

Для форматирования выводимой информации применяется функция `printf()`, которая преобразует выходные данные при помощи форматов преобразования, подробно объясненных в лекции 7 при описании функции `sprintf()`. Например, так можно вывести отформатированное текущее время в разные выходные потоки:

```
my ($hh, $mm, $ss) = (localtime)[2, 1, 0];

# выбрать из списка нужные значения: часы, минуты, секунды

my $format = "%02d:%02d:%02d\n"; # формат вывода

printf $format, $hh, $mm, $ss; # вывод в STDOUT

printf(STDERR $format, $hh, $mm, $ss); # вывод в STDERR

printf $file $format, $hh, $mm, $ss; # вывод в $file
```

Задавая различные форматы преобразования, можно выводить данные в требуемом представлении или в виде колонок указанной ширины. Более тонкое управление выводимыми данными организуется средствами форматирования отчетов, которые будут изучены в следующей лекции.

В файле с исходным текстом программы на Perl может располагаться встроенный файл с данными, которые помещаются в конце программного файла после специальной лексемы `__END__` (в основной программе) или `__DATA__` (в программном модуле). При выполнении программы данные из этого встроенного файла доступны для чтения при помощи специального файлового манипулятора `DATA`. Во встроенном файле удобно хранить тестовые данные для проверки работы программы после ее модификации. Вот пример чтения данных из встроенного файла:

```
while (my $line = ) { # читаем построчно данные

    print $line; # обрабатываем данные

}

__END__
```

Это данные из встроенного файла

Двоичные данные обычно хранятся в файлах без разделителей записей в блоках фиксированной длины. После открытия двоичного файла функцией `open()` нужно установить режим обработки двоичных данных с помощью функции `binmode()`.

```
open(my $fh, ">$file") or die("Ошибка открытия: $!");

binmode($fh);
```

Запись двоичных данных или данных фиксированной длины может выполняться с помощью функции `print($fh $record)`. Также имеется функция небуферизованного вывода `syswrite()`, которой при вызове указываются три аргумента: файловый манипулятор, скалярная переменная с выводимыми данными и размер записываемого блока данных. Эта функция возвращает число фактически записанных байт (в случае ошибки `syswrite` возвращает `undef`), что можно использовать для проверки успешности записи. Это делается так:

```
syswrite($fh, $record, length($record)) == length($record)

or die("Ошибка записи: $!");
```

Преобразование данных к двоичному виду производит функция `pack()`, которая упаковывает в скалярную переменную список значений в соответствии с указанным шаблоном. В шаблоне каждое преобразуемое поле обозначается с помощью латинской буквы. Полный перечень шаблонов преобразования для функций `pack()` и `unpack()` приводится в таблице 9.2. За каждым символом в шаблоне может следовать десятичное число, которое рассматривается как ширина преобразуемого поля. Поля в шаблоне могут разделяться пробелами для удобства чтения.

Таблица 9.2. Шаблоны упаковки и распаковки данных

| Шаблон             | Мнемоника      | Описание преобразования  |
|--------------------|----------------|--|
| <code>a</code>     | Arbitrary      | произвольная последовательность байтов, дополненная нулевым байтом <code>\0</code>               |
| <code>A</code>     | ASCII          | строка символов ASCII, дополненная пробелами   |
| <code>b / B</code> | Bit string     | строка битов с возрастающим / убывающим порядком битов   |
| <code>c / C</code> | Character      | однобайтовые символы со знаком / без знака   |
| <code>f / d</code> | Float / Double | число с плавающей точкой одинарной / двойной точности  |
| <code>F</code>     | Float          | число с плавающей точкой одинарной точности во внутреннем представлении (NV)                     |
| <code>D</code>     | long Double    | длинное число с плавающей точкой двойной точности  |
| <code>h / H</code> | Hex string     | шестнадцатеричная строка с младшим / старшим полубайтом (nybble) в начале                        |
| <code>i / I</code> | Integer        | целое ( $\geq 32$ бита) число со знаком / без знака  |
| <code>j / J</code> |                | целое во внутреннем представлении со знаком (IV) / без знака (UV)                                |
| <code>l / L</code> | Long           | длинное (32 бита) целое со знаком / без знака  |
| <code>n / N</code> | Network        | беззнаковое короткое (16 битов) / длинное (32 бита) целое с сетевым порядком байтов (big endian) |
| <code>p / P</code> | Pointer        | указатель на строку, оканчивающуюся <code>\0</code> / фиксированной длины                        |
| <code>q / Q</code> | Quad           | сверхдлинное (64 бита) целое число со знаком / без знака   |
| <code>s / S</code> | Short          | короткое (16 битов) целое со знаком / без знака  |
| <code>u</code>     | uuencoded      | строка, кодированная по алгоритму uuencode   |
| <code>U</code>     | Unicode        | строка символов Unicode  |
| <code>v / V</code> | VAX            | беззнаковое короткое (16 битов) / длинное (32 бита) целое с VAX-порядком байтов (little endian)  |
| <code>w</code>     |                | целое, сжатое в соответствии с кодировкой BER  |
| <code>x</code>     |                | вставка <code>\0</code> (pack) / пропуск байта по направлению вперед (unpack)                    |
| <code>X</code>     |                | пропуск байта по направлению назад   |
| <code>Z</code>     | ASCIIZ         | строка ASCIIZ (оканчивающаяся <code>\0</code> ), дополненная <code>\0</code>                     |
| <code>@</code>     |                | заполнение <code>\0</code> до указанной позиции  |

Например, целочисленное значение, возвращаемое функцией `time()`, и дробное значение, возвращаемое функцией `rand()`, можно упаковать в переменную `$record` с помощью шаблона `'l1 d1'`, который означает: "одно длинное целое число (long) и одно число с плавающей точкой двойной точности (double)".

```
$record = pack 'l1 d1', time(), rand(); #
```

Вот еще несколько несложных примеров использования разных шаблонов для функции `pack()`:

```
$bin = pack('a5', 'Yes'); # в $bin будет: 'Yes\0\0'
```

```
$bin = pack('A5', 'Yes'); # в $bin будет: 'Yes '
```

```
$bin = pack('a4', 'abcd','x','y','z'); # в $bin: 'abcd'
```

```
$bin = pack('aaaa', 'abcd','x','y','z'); # в $bin: 'axyz'
```

```
$bin = pack('C2', 65,66,67); # в $bin будет: 'AB'
```

```
$bin = pack('U2', 0x263A, 0x263B); # в $bin будет: '??'
```

```
$bin = pack('cххс', 65,66); # в $bin будет: 'A\0\0B'
```

Для преобразования данных из двоичного вида применяется функция `unpack()`, которая распаковывает из скалярной переменной в список или массив значения двоичных данных в соответствии с указанным шаблоном.

```
@list_of_values = unpack($template, $binary_record);
```

Кроме того, с помощью функции `unpack()` можно из строки извлекать подстроки фиксированной длины. Например, так можно извлечь из записи файла поля определенной длины в переменные:

```
# Поля данных в записи файла:
```

```
# с 1 по 7 байт - номер телефона
```

```
# с 8 длиной 30 - фамилия, имя, отчество абонента
```

```
# с 38 длиной 25 - адрес
```

```
# 1234567Бендер Остап Ибрагимович РСФСР, Черноморск
```

```
($phone, $name, $address)= unpack('A7A30A25', $record);
```

Чтобы пропустить ненужные поля, достаточно указать в шаблоне пропуск определенного количества байтов. Например, так можно не извлекать поле с телефонным номером:

```
($name, $address)= unpack('x7A30A25', $record);
```

Подробное описание шаблонов и работы функций `pack()` и `unpack()` можно найти в стандартной документации с помощью все той же утилиты чтения документации:

```
perldoc perlpacktut
```

Для чтения двоичных данных или текстовых данных фиксированной длины применяется функция `read()`, которой в качестве аргументов передаются файловый манипулятор, скалярная переменная для вводимых данных и размер считываемого блока данных. Вот так, например, выглядит типичный цикл чтения двоичных данных:

```
until(eof($fh)) { # читать до достижения конца файла
```

```
# считать очередной блок данных и проверить его длину
```

```
read($fh, $record, $record_size) == $record_size
```

```
or die('Неправильная длина данных');
```

```
# распаковать данные по шаблону из $record в @data
```

```
@data = unpack($template, $record);
```

```
# обработать введенные данные...
```

```
}
```

При работе с данными фиксированной длины обычной практикой является считывание или запись данных в произвольном месте файла, например, при изменении только что считанного блока данных. Для этого нужно позиционировать позицию чтения или записи. Это делается с помощью функции `seek()`, которой передается три аргумента: файловый манипулятор, смещение в байтах и



указатель позиции отсчета. Позиция отсчета задается числами: 0 - от начала файла, 1 - от текущей позиции, 2 - от конца файла. Например:

```
seek($handle, 64, 0); # переместиться на 64 байта от начала

seek($handle, 25, 1); # сместиться на 25 байт вперед

seek($handle, -10, 2); # установиться на 10 байт до конца

seek($handle, 0, 0); # установить позицию в начало файла
```

С помощью функции tell(), которая возвращает смещение относительно начала файла, можно узнать текущую позицию чтения-записи и использовать ее для дальнейших перемещений по файлу.

```
$pos = tell($handle); # запомнить текущую позицию в $pos

seek($handle, $pos-5, 1); # сместиться на 5 байт назад
```

В следующем примере увеличивается поле счетчика длиной 2 байта, расположенное в файле с позиции \$new\_pos:

```
seek($file, $new_pos, 0); # установить позицию чтения

$pos = tell($file); # и запомнить ее в переменной

read($file, $number, 2); # прочитать 2-байтовое поле

seek($file, $pos, 0); # установить в исходную позицию

syswrite($file, ++$number, 2); # записать новое значение
```

Операции ввода-вывода с произвольным доступом часто используются для работы с базами данных, основанных на записях фиксированной длины, например, с файлами в формате DBF. Они позволяют организовать быструю выборку данных и запись измененных данных на прежнее место.

Перед выполнением операций ввода-вывода часто требуется узнать информацию об объектах файловой системы. В Perl есть набор унарных операций для удобной проверки различных характеристик файлов и каталогов. Они имеют вид флагов из одной латинской буквы с предшествующим знаком минус, после которого указывается имя проверяемого файла. Полный перечень операций проверки файлов приведен в таблице 9.3.

Таблица 9.3. Операции проверки файлов

| Операции | Описание проверок   |
|----------|---|
| -r -w -x | Файл доступен для чтения / записи / исполнения (no effective UID+GID) |
| -R -W -X | Файл доступен для чтения / записи / исполнения (no real UID+GID)      |
| -o -O    | Файл принадлежит текущему пользователю по effective / real UID        |
| -e -z    | Файл существует (exists) / имеет нулевую длину (zero)                 |
| -s       | Файл имеет ненулевой размер: возвращает размер в байтах (size)        |
| -f -d    | Файл является обычным файлом (file) / каталогом (directory)           |
| -l -S -p | Файл является ссылкой / сокетом / именованным FIFO-каналом (pipe)     |
| -b -c    | Файл является блочным / символьным специальным файлом                 |
| -u -g -k | Для файла установлен бит setuid / setgid / sticky                     |
| -t       | Файловый манипулятор связан с терминалом (tty)                        |

|          |   |
|----------|---|
| -T -B    | Файл является текстовым (text) / двоичным (binary)  |
| -M -A -C | Время изменения (modification) / доступа (access) / изменения (change) индексного узла (inode) файла в днях относительно времени начала выполнения программы (\$^T) |

Вот несколько типичных примеров использования операций проверки файлов для контроля доступности данных:

```
open($f1, "<$file1") # открыть файл на чтение,

if (-e $file1) && # если он существует и

(-r $file1); # он доступен на чтение

open $f2, ">$file2" # открыть файл на запись,

if -w $file2; # если в него можно писать

$file_size = -s $file; # узнать размер файла

print "$file - является каталогом!" if -d $file;
```

В Perl есть целый набор встроенных функций для работы с файлами, с помощью которых можно манипулировать с самими файлами, а не с данными, хранящимися в них.

Функция `rename()` переименовывает файл, возвращая логическое значение: 1 при успешном изменении имени и 0 - в противном случае. Ей передаются старое и новое имя файла с абсолютным или относительным путем.

```
$ok = rename("$path/$old_name", "$path/$new_name");
```

Функция `unlink()` удаляет файл или список файлов, возвращая число 1 при успешном удалении и 0 - при ошибке.

```
unlink($list, $of, $files); # удалить список файлов

unlink $file if -e $file; # удалить файл, если он существует
```

Функция `truncate()` усекает файл до указанного размера и возвращает число 1 в случае успешного выполнения усечения и неопределенное значение `undef` - при возникновении ошибки. Файл может задаваться именем или файловым манипулятором. Например:

```
$ok = truncate($file, $size);
```

Функция `stat()` возвращает информацию о файле в виде списка или пустой список при неудаче. Файл может задаваться манипулятором файла или выражением со значением имени файла:

```
@info = stat($file); # получить всю информацию о файле

$size = $info[7]; # размер файла в байтах

$modified = localtime($info[9]); # время изменения файла
```

Подробное описание всех элементов информационного списка можно найти в документации, указав утилите чтения документов имя функции следующим образом:

```
perldoc -f stat
```

Функция `utime()` изменяет у файла (или нескольких файлов из заданного списка) время доступа и время модификации, задаваемые числовыми значениями.

```
utime($access_time, $modified_time, @list_of_files);
```

Кроме упомянутых, есть еще встроенные функции для изменения прав доступа и владельца файла, для чтения и создания символических и жестких ссылок. Немало разных функций для работы с файлами имеются в стандартной библиотеке модулей Perl, еще больше можно найти в хранилище модулей CPAN.

Помимо работы с файлами, в Perl есть необходимый набор встроенных функций для работы с каталогами. Нужно иметь в виду, что успешность выполнения операций с каталогами зависит от набора прав пользователя, от лица которого выполняется Perl-программа.

Функция `mkdir()` создает каталог с указанным именем. Вторым аргументом функции можно задавать права доступа для создаваемого каталога (в соответствии со стандартом POSIX). Возвращает число 1 при успешном создании или 0 при ошибке. Причину неудачи при создании каталога можно узнать из специальной переменной `!`, которая содержит сообщение об ошибке.

```
$ok = mkdir $directory_name; # создать каталог
```

```
mkdir $dir, $access_rights; # создать каталог, задав права
```

Функция `rmdir()` удаляет каталог по его имени, если он пуст, возвращает число 1 при успешном удалении каталога или 0 в случае ошибки. Тогда из переменной `!` можно выяснить подробности неудачи:

```
$ok = rmdir $directory_name; # удалить каталог
```

Функция `chdir()` изменяет текущий каталог на значение строкового выражения, содержащего имя нового текущего каталога. Если имя каталога не задано, она переходит в домашний каталог пользователя, используя значение элемента `$ENV{"HOME"}` или `$ENV{"LOGNAME"}` из специального хэша `%ENV`, который содержит значения переменных окружения операционной системы. Возвращает число 1 при успешном переходе в другой каталог или 0 в случае возникновения ошибки.

```
$ok = chdir $new_dir; # перейти в каталог $new_dir
```

```
chdir; # перейти в домашний каталог
```

Много других возможностей по работе с каталогами предоставляют функции из стандартной библиотеки модулей Perl. Например, функция `cwd()` из стандартного модуля `Cwd` возвращает имя текущего (рабочего) каталога во время выполнения программы:

```
use Cwd; # подключить библиотечный модуль Cwd
```

```
my $current_work_directory = cwd; # запросить текущий каталог
```

Иногда при выполнении программы нужно определить каталог, откуда она была запущена, чтобы организовать доступ к его подкаталогам. Переменная `$Bin`, устанавливаемая в модуле `FindBin` из стандартной библиотеки, содержит имя каталога, из которого программа была запущена, например:

```
use FindBin; # подключить библиотечный модуль FindBin
```

```
my $program_start_dir = $FindBin::Bin; # стартовый каталог
```

Perl также предоставляет набор функций, организующий чтение содержимого каталога, подобно чтению записей файла. Элементами каталога могут быть обыкновенные файлы и другие каталоги, включая вложенные подкаталоги, текущий каталог (обозначаемый одной точкой) и родительский каталог (обозначаемый двумя точками).

Перед чтением содержимого каталога его необходимо открыть. Функция `opendir()` открывает каталог по его имени и ассоциирует его с указанным манипулятором каталога. Функция возвращает число 1 при успешном открытии, `undef` - при неудаче. Вот так, например, открывается каталог и создается манипулятор каталога `DIR_HANDLE`:

```
$ok = opendir DIR_HANDLE, $directory_name;
```

Современный стиль программирования рекомендует при открытии каталога сохранять манипулятор каталога в скалярной переменной. Это ограничивает область его видимости текущим блоком или подпрограммой и предотвратит случайное изменение манипулятора в других частях программы. Например, так:

```
$ok = opendir my $dir_handle, $directory_name;
```

Функция `closedir()` закрывает каталог, открытый функцией `opendir()`, используя манипулятор каталога. Возвращает число 1 при успешном закрытии или `undef` - при неудаче. Хотя открытые каталоги автоматически закрываются по окончании программы, рекомендуется все же делать это явно:

```
$ok = closedir $dir_handle; # закрыть каталог
```

Функция `readdir()` в скалярном контексте читает очередной элемент каталога, возвращая неопределенное значение `undef`, когда будет прочитан последний элемент. Например:

```
my $file_name = readdir $dir_handle;
```

Таким образом можно организовать обработку всех элементов каталога в цикле, исключая текущий и родительский каталоги:

```
while (my $file_name = readdir $dir_handle) {  
  
    if ($file_name ne '.' && $file_name ne '..') {  
  
        print "каталог $file_name\n" if -d $file_name;  
  
        print "файл $file_name\n" if -f $file_name;  
  
    }  
  
}
```

В списочном контексте функция `readdir()` возвращает список всех элементов каталога, что часто бывает весьма удобно:

```
@file_names = readdir $dir_handle; # считать весь каталог
```

Иногда требуется, не закрывая каталога, начать его обработку сначала. Функция `rewinddir()` устанавливает позицию чтения в начало открытого каталога, после чего чтение начнется с первого элемента:

```
rewinddir $dir_handle; # 'перемотать' на начало каталога
```

Функция `telldir()` возвращает текущую позицию в каталоге, которую можно использовать для перемещения в эту позицию с помощью функции `seekdir()`:

```
$dir_position = telldir $dir_handle; # позиция чтения
```

Используя возвращенное функцией `telldir()` значение, встроенная функция `seekdir()` устанавливает новую позицию для чтения каталога с помощью функции `readdir()`:

```
seekdir $dir_handle, $dir_position; # вернуться к позиции
```

После изучения представленных в этой лекции средств ввода-вывода вполне можно писать законченные и полезные программы на Perl. Но пройдена только половина пути: в следующих лекциях курса будет рассказано о важных конструкциях и механизмах системы программирования Perl, без которых невозможно разрабатывать современные программы на профессиональном уровне.

## Лекция 10. Отчеты

В этой лекции описываются средства создания форматированных отчетов и технология формирования в Perl отчетов с помощью форматов, которая позволяет представлять выходные данные в форме, удобной для просмотра человеком или для вывода на принтер.

Цель лекции: научиться описывать формат заголовка страницы и формат строк отчета, разобраться с тонкостями описания полей в форматах. На примерах освоить заполнение формата отчета данными и вывод отчетов в разные выходные потоки.

Вся информация, хранящаяся в компьютерных системах, в конечном счете предназначена для людей, а людям нравится видеть ее упорядоченной и аккуратно представленной. Представление данных, специально подготовленное для просмотра на экране или на бумаге, принято называть отчетом. Отчеты создаются для отображения информации в удобном для восприятия виде, поэтому при формировании отчета данные сортируются, обобщаются, снабжаются пояснительными заголовками и располагаются по возможности компактно и эстетично. Все мы постоянно сталкиваемся с отчетами, когда видим данные в "причесанном" отформатированном виде: это списки, реестры, таблицы, формы, карточки и другие документы. Одной из целей разработки Perl было создание удобного инструмента для подготовки отчетов. Perl предоставлял механизм для создания простых отчетов и диаграмм, начиная с самых ранних версий, и он незначительно изменился с тех пор. Хотя имеющиеся в нем средства работы с отчетами весьма лаконичны, они достаточны, чтобы в большинстве ситуаций приготовить данные для форматированного постраничного вывода на экран, печатающее устройство или в файл. Подготовка отчета начинается с описания его формата.

Внешний вид отчета и расположение данных в нем описывается с помощью формата. Формат - это шаблон отчета, который состоит из литеральных строк (заголовков, пояснительных надписей, констант) и описания полей, куда при выводе отчета будут подставляться значения данных. Формат описывается с помощью ключевого слова `format`, после которого указывается имя формата и знак равенства. Далее со следующей строки располагается описание строк формата отчета (хотя они могут и отсутствовать). Описание формата заканчивается отдельной строкой, состоящей из единственной точки. Например, описание формата с именем `FORMAT_NAME` будет выглядеть так:

```
format FORMAT_NAME =
```

```
описание формата отчета
```

```
.
```

Форматы не исполняются, поскольку являются описаниями, как и определения подпрограмм, поэтому они могут помещаться в любом месте программного файла. Обычно они располагаются в конце исходного текста программы перед определением подпрограмм. Имя формата представляет собой правильный идентификатор. Его принято записывать заглавными буквами, и оно обычно совпадает с именем файлового дескриптора выходного потока, куда будет выводиться отчет. Имена форматов хранятся в отдельном пространстве имен, поэтому они не конфликтуют с именами переменных, подпрограмм, меток и файловых манипуляторов. Имя текущего формата для каждого потока хранится в специальной переменной `$~` (или `$FORMAT_NAME` при включенной прагме `use English`). Если имя формата в описании не указано, подразумевается `STDOUT`.

Кроме основного формата, часто требуется описание отдельного формата для шапки страницы, которая будет выводиться в начале каждой страницы отчета. Формат заголовка страницы отчета отличается от основного формата тем, что к имени формата добавляется `'_TOP'`. Он описывается так:

```
format FORMAT_NAME_TOP =
```

```
описание формата шапки отчета
```

```
.
```

Имя текущего формата для шапки страницы, связанного с текущим выходным потоком, хранится в специальной переменной `$^` (`$FORMAT_TOP_NAME`). При выводе отчета, если очередная строка не помещается на текущей странице, в выходной поток выводится символ прогона страницы, счетчик страниц увеличивается на единицу, и в начале следующей страницы в текущий поток выводится заголовок страницы отчета.

В описании формата отчета применяется несколько видов строк. Строка шаблонов (или строка полей) описывает поля отчета для заполнения данными. За каждой строкой шаблона следует одна или несколько строк аргументов (или строк значений), содержащих список источников данных (литералов, имен переменных и выражений, разделенных запятыми) для заполнения полей. Литеральная строка содержит информацию, без изменения выводимую в отчет. После литеральной строки строка значений не указывается. Строка комментариев, начинающаяся с символа `#`, служит для записи в формате примечаний и не выводится в отчет. В следующем примере приводятся образцы разных типов строк в описании формата:

```
format STDOUT =
```

```
1. Строка шаблонов содержит поля: @<<<< и @###.##
```

```
'поле1', $field2
```

```
# 3. Комментарий: во 2-й строке данные для вставки в 1-ю
```

```
4. Литеральная строка выводится "как есть".
```

```
.
```

В комментариях и литеральных строках нет ничего особенного, самое интересное содержится в строках шаблонов, содержащих описания полей отчета.

Поле отчета - это пространство указанной ширины, расположенное в определенном месте отчета, куда помещаются данные в нужном представлении. (Например, конкретное поле в отчете может быть описано таким образом - "в начале первой строки заголовка каждой страницы должна выводиться текущая дата в следующем виде: день (две цифры с ведущим нулем), месяц (две цифры с ведущим нулем) и год (четыре цифры с ведущим нулем), разделенные точками".) Поля отчета в описании формата представлены в виде так называемых полейдержателей (`fieldholders`). Полейдержатель (или переменная поля) представляет из себя шаблон, описывающий тип поля, его ширину в символах, выравнивание значения внутри поля и другие преобразования, которые

нужно выполнить над данными при размещении их в этом поле во время формирования отчета. Поледержатель начинается с символа начала шаблона (@ или ^), за которым следуют символы описания шаблона. Число символов в шаблоне, включая символ начала шаблона, определяет ширину помещаемых в отчет данных. Несколько примеров поледержателей с пояснениями приведены в таблице 10.1:

Таблица 10.1. Примеры описания полей в формате отчета

| Поледержатель | Описания формата и преобразований  |
|---------------|--|
| @<<<<<<<<<<<  | Вывести текстовое значение в поле шириной в 12 символов. Выводить его по левому краю, дополнив при необходимости пробелами справа до ширины поля. Слишком длинное значение усесть до ширины поля   |
| @<<<<<<<<...  | Аналогично предыдущему примеру, но с выводением в конце поля многоточия, если значение усечено   |
| @####.###     | Вывести числовое значение в поле шириной в 9 символов, отведя 5 цифр под целую и 3 цифры - под дробную часть числа. Выводить его по правому краю и дополнить при необходимости целую часть числа пробелами слева до ширины поля и округлить дробную часть до 3 знаков. При попытке вывести число, целая часть которого не умещается в ширину поля, заполнить поле символом '#' как признак |
| @0###.###     | Так же, как в предыдущем примере, но с дополнением целой части значения ведущими нулями до ширины поля   |

Полный список символов, применяемых для описания полей и форматов, приводится в таблице 10.2.

Таблица 10.2. Символы, применяемые при описании полей и форматов

| Символ | Описание  | Примеры использования     |
|--------|---|---------------------------|
| @      | начало обычного поля  | @ @<< @    @>> @##        |
| ^      | начало специального поля  | ^ ^<< ^    ^>> ^##        |
| <      | текстовое поле с выравниванием значения влево и добавлением пробелов справа   | @<<<<< ^<<<               |
|        | текстовое поле с центрированием значения и добавлением пробелов с обеих сторон  | @      ^                  |
| >      | текстовое поле с выравниванием значения вправо и добавлением пробелов слева   | @>>>>> ^>>>               |
| #      | числовое поле с выравниванием значения вправо с добавлением пробелов слева  | @##### ^###               |
| 0      | (вместо первого #) числовое поле с выравниванием значения вправо и добавлением нулей слева  | @0### ^0##                |
| .      | десятичная точка в числовом поле  | @.### @0#.##              |
| ...    | закончить текстовое поле многоточием, чтобы показать усечение значения  | @<<<<<...                 |
| @*     | поле переменной ширины со значением, состоящим из нескольких строк  | @*                        |
| ^*     | поле переменной ширины для следующих строк многострочного значения  | ^*                        |
| ~      | подавление вывода строки с пустыми значениями полей   | ^* ~                      |
| ~~     | повторять строку, пока все значения полей не станут пустыми   | ~~ ^*                     |
| {}     | группировка списка значений, который располагается на нескольких строках аргументов   | { \$one, \$two, \$three } |
| #      | (первым символом в строке) строка комментария в описании формата (не может располагаться между строкой шаблонов и строкой аргументов) | # это комментарий         |

|   |   |                                    |
|---|---|------------------------------------|
| . | (единственным символом на отдельной строке) конец формата | format REPORT = описание формата . |
|---|---|------------------------------------|

То, как применяются полейдержатели при описании формата, можно увидеть из следующего примера:

```
format STDOUT =
```

Учетная карточка пользователя N @0###

\$number

-----

Фамилия @<<<<<<<<<< | Login @<<<<<<

```
$last_name, $login
```

Имя @<<<<<<<<< | Группа @<<<<<<<<<<<<<<<<<

```
$first_name, $group
```

Отчество @<<<<<<<<<<<<<<<<

\$middle\_name

[illegible]

```
$email, $phone
```

Ограничение дискового пространства @####.## Мегабайт

\$quota

-----

Дата регистрации @# @&lt;&lt;&lt;&lt;&lt;&lt;&lt; @### года

 $\{ \$day,$ 

```
$month_name,$year}
```

■

Из примера понятно, что формат отчета записывается в виде, максимально похожем на представление страницы отчета на экране или на бумаге. Каждому полю в строке шаблонов должно соответствовать скалярное значение в строке аргументов. Имена переменных в строке аргументов для наглядности часто располагаются под соответствующими поледержателями в предыдущей строке шаблонов, хотя это совсем не обязательно. Список переменных может находиться на нескольких строках аргументов (как это сделано в описании последней строки формата); в этом случае он должен заключаться в фигурные скобки. Имейте в виду, что скалярные переменные и массивы в строке аргументов разворачиваются в единый список скаляров, из которого по порядку берутся значения для заполнения полей.

Для форматированного вывода отчетов применяется функция `write()`, которая оформляет очередную порцию данных в соответствии с форматом отчета и выводит их в указанный выходной поток. Обращение к функции `write()` иногда называют вызовом формата. В качестве аргумента функции `write()` может передаваться файловый манипулятор выходного потока. Вызванная без аргументов, она направляет отчет в текущий выходной поток. Перед обращением к ней нужно заполнить новыми данными переменные, перечисленные в строках аргументов текущего формата. Обычно `write()` вызывается в цикле для вывода в отчет очередной строки. По историческим причинам для заполнения полей отчета часто используются глобальные переменные. Лексические переменные, объявленные с помощью `tu()`, доступны в формате только тогда, когда формат и лексические переменные объявлены в одной области видимости. Подробно об областях видимости переменных будет рассказано в лекции 12.

Если для выходного потока описан формат начала страницы отчета, то перед выводом строк отчета функцией `write()` в начале каждой страницы автоматически размещаются данные шапки страницы в соответствии с форматом начала страницы. Программа для вывода данных по формату, заданному в предыдущем примере, может выглядеть таким образом:

```
# данные в записи входного файла разделены запятыми

open my $in, '<', 'users.txt' or die;

while (my $line = <$in>) {

    local ($last_name, $first_name, $middle_name,

    $login, $group, $email, $phone, $quota, $number,

    $day, $month_name, $year) = split ',', $line;

    # данные для отчета помещены в переменные

    write STDOUT; # данные выводятся в STDOUT по формату

}

close $in or die;

# здесь располагается описание формата...
```

В результате выполнения этой программы в поток STDOUT будет выведен отчет, состоящий вот из таких карточек:

Учетная карточка пользователя N 00001

-----

Фамилия Wall | Login larry

Имя Larry | Группа root

Отчество |

E-mail larry@wall.org | Телефон +123456789

Ограничение дискового пространства 9876,54 Мегабайт

-----

Дата регистрации 18 декабря 1987 года

В каждую из выводимых в отчет карточек помещаются данные из одной записи входного файла.

Без указания файлового манипулятора вывод отчета функцией write() и обычный вывод данных функцией print() происходит в выходной поток по умолчанию (STDOUT). С помощью функции select() можно назначить другой выходной поток по умолчанию. При вызове ей передается файловый манипулятор, и она переключается на новый поток, который становится текущим выходным потоком по умолчанию. Функция select() возвращает имя ранее выбранного манипулятора, и это значение используется для восстановления предыдущего выходного потока. Это происходит таким образом:

```
$old_handle = # сохранить файловый манипулятор

select $new_handle; # переключиться на новый поток

write; # вывести в новый поток

select $old_handle; # восстановить предыдущий поток
```

При формировании сложного отчета может потребоваться возможность переключаться на разные форматы отчета. Установить для какого-либо потока определенный формат отчета можно путем присваивания имени формата переменной \$~ (\$FORMAT\_NAME). Подобным же образом для конкретного потока устанавливается нужный формат заголовка страницы отчета: переменной \$^ (\$FORMAT\_TOP\_NAME) присваивается имя формата для шапки страницы. Это делается так:

```
$old_handle = select $out; # выбрать поток для отчета
```



```
$^ = 'REPORT_TOP'; # назначить формат для шапки отчета
```

```
$~ = 'REPORT'; # назначить формат для отчета
```

```
write $out; # вывести в $out по формату REPORT
```

```
select $old_handle; # вернуться к предыдущему потоку
```

Назначать для определенного потока формат отчета и заголовок страницы гораздо удобнее с помощью функций `format_name()` и `format_top_name()` из стандартного библиотечного модуля `FileHandle`. Это выглядит так:

```
use FileHandle; # подключить модуль работы с файлами
```

```
# назначить для потока $report формат отчета REPORT
```

```
format_name $report REPORT;
```

```
# назначить для потока $report формат заголовка PAGE
```

```
format_top_name $report PAGE;
```

```
# используя назначенные форматы,
```

```
write $report; # вывести строку отчета в $report
```

Обратите внимание, что при обращении к функциям `format_name()` и `format_top_name()` после файлового манипулятора не ставится запятая, так же как при вызове функции `print()`.

Пока что в примерах использовались только обычные поля (*regular fields*), которые описываются поледержателями, начинающимися с символа `@`. Поледержатели, описание которых начинается с символа `^`, представляют так называемые специальные поля (*special fields*), обладающие возможностью дополнительной обработки данных. Так, специальные числовые поля (например, `^###`), содержащие неопределенное значение (`undef`), заполняются пробелами. Обычные числовые поля (например, `@###`) в этом случае выводят нули. Это демонстрирует следующий пример:

```
format STDOUT =
```

```
обычное: '@##.###' специальное: '^#####'
```

```
undef, undef
```

```
.
```

```
write STDOUT; # вывод данных по формату в STDOUT
```

```
# выведет: обычное: ' 0.00' специальное: ' '
```

Специальные текстовые поля (например, `^<<<`) используются для вывода в отчет данных, располагающихся на нескольких строках.

Есть несколько способов описать в формате данные, занимающие в отчете несколько строк. Если нужно поместить на странице отчета многострочное текстовое значение, то можно воспользоваться поледержателем `@*`, который просто выведет значение полностью, сохраняя имеющиеся в нем все символы перевода строки, кроме последнего. Например, так делается в этой программе:

```
format STDOUT =
```

```
Гамлет:
```

```
@*
```

```
$multi_line_text
```

```
(У. Шекспир)
```

```
.
```

```
$multi_line_text = "Быть\nИли не быть?\nВот в чем вопрос.";
```

```
write STDOUT;
```

В результате ее выполнения будет выведено известное высказывание с сохранением его разбивки на несколько строк:

Гамлет:

Быть

Или не быть?

Вот в чем вопрос.

(У. Шекспир)

Поледержатель `^*` описывает в формате текстовое поле, значение которого должно выводиться на несколько строк определенной ширины. В строке аргументов такому полю должно соответствовать имя переменной: скаляра, элемента массива или хэша. Когда значение этой переменной помещается в поле отчета, из переменной извлекается часть текстового значения до первого разделителя строк. Если переменная употребляется в формате несколько раз, то ее значение уменьшается на число извлеченных символов при каждом обращении к ней. Вот как это выглядит на примере:

```
# выводимое многострочное значение
```

```
$text = "Что значит имя?\nРоза пахнет розой\n"
```

```
. "Хоть розой назови ее, хоть нет.";
```

```
write STDOUT;
```

```
# описание формата отчета
```

```
format STDOUT =
```

```
^*
```

```
$text
```

```
^*
```

```
$text
```

```
^* ~
```

```
$text
```

```
^* ~
```

```
$text
```

```
У.Шекспир, "Ромео и Джульетта"
```

```
.
```

Обратите внимание, что в формате для вывода значения переменной `$text` предусмотрены четыре строки. Причем первые две строки отчета (с шаблоном `^*`) будут выводиться в любом случае: даже если `$text` содержит только одну строку текста, вторая строка будет заполнена пробелами. Чтобы не выводить пробельных строк, в описании третьей и четвертой строк указан шаблон подавления пустых строк (одна тильда, `~`), который может располагаться в любом месте строки шаблона. После выполнения приведенной программы текст будет выведен в таком виде:

Что значит имя?

Роза пахнет розой

Хоть розой назови ее, хоть нет.

У.Шекспир, "Ромео и Джульетта"

Если заранее неизвестно минимальное количество выводимых строк, можно применить шаблон повторения многострочного поля (две тильды подряд в любом месте строки, ~~). В этом случае строка шаблона будет применяться многократно, пока не будет исчерпано многострочное значение, при этом пустые строки выводиться не будут. Тильды в шаблоне подавления пустых строк и в шаблоне повторения строк при выводе отчета заменяются на пробелы. Таким образом, формат в последнем примере можно заменить на следующий:

```
format STDOUT =
```

$$\wedge^* \sim \sim$$

\$text

У.Шекспир, "Ромео и Джульетта"

•

В результате будет выведен точно такой же отчет, как и с использованием предыдущего формата, но последний формат гораздо короче и выводит любое количество строк.

Для описания в формате текстового значения, которое должно выводиться на несколько строк, применяется поделератель следующего вида: ^<<<<. Это специальное поле иногда называется "заполняемым полем". Оно предназначается для форматирования текстового значения, которое при выводе в отчет делится на строки, не превышающие ширину шаблона. Деление на строки производится по границам слов. Источником выводимого текста обязательно должна быть переменная со скалярным значением, из которой при каждом ее употреблении в формате извлекается столько слов, сколько поместится в соответствующем поле отчета. Заполнение шаблона текстом и вывод отчета в несколько строк иллюстрирует следующий пример:

```
my $shakespeare = 'Две равно уважаемых семьи '
```

. 'В Вероне, где встречаются нас события, '

. 'Ведут междоусобные бои '

. 'И не хотят унять кровопролития.';

```
my $text = $shakespeare;
```

```
write STDOUT;
```

```
# описание формата вывода
```

```
format STDOUT =
```

$\sim \Lambda$

\$text

У.Шекспир, "Ромео и Джульетта"

•

В этом примере для организации неявного цикла вывода также применяется шаблон повторения строк, поскольку неизвестно, сколько строк будет заполнено выводимым текстом. При выполнении примера будет выведен текст в таком виде:

Две равно уважаемых семьи В Вероне,

где встречают нас события, Ведут

междоусобные бои И не хотят унять

кровопролиться.

У.Шекспир, "Ромео и Джульетта"

Подобным образом в отчет выводятся блоки текстовой информации: примечания, описания, адрес и т. п.

Кроме переменных, в которых хранятся имена формата (\$~) и заголовка страницы формата (\$^), есть еще несколько специальных переменных для хранения информации о форматах. Номер текущей страницы выводимого отчета содержится в переменной \$% (\$FORMAT\_PAGE\_NUMBER), и ее часто включают в формат отчета. В переменной \$= (\$FORMAT\_LINES\_PER\_PAGE) хранится число строк на странице: по умолчанию - 60, но его можно изменить на нужное значение перед выводом отчета. В переменной \$- (\$FORMAT\_LINES\_LEFT) содержится число оставшихся на странице строк. Переменная \$^L (\$FORMAT\_FORMFEED) хранит символ перевода страницы (formfeed character), который используется в отчетах для прогона принтера до новой страницы.

Специальная переменная \$: (\$FORMAT\_LINE\_BREAK\_SEPARATOR) содержит набор символов разрыва строки, после которых строка может быть разделена при заполнении в формате специальных полей продолжения. Специальная переменная \$^A (\$ACCUMULATOR) является аккумулятором выводимых данных для функций formline() и write(), в котором накапливаются данные отчета перед их отправкой в выходной поток. При считывании данных для отчета из файла может пригодиться переменная \$. (\$INPUT\_LINE\_NUMBER), в которой хранится номер прочитанной из входного файла строки, что можно использовать для нумерации строк в отчете.

Дополнительные сведения о форматах и отчетах в Perl можно узнать из стандартной документации, обратившись за помощью к утилите

```
perldoc perlform
```

В завершение лекции приведем пример законченной программы (с образцом исходных данных), выводящей отчет о книгах по языку Perl.

```
open my $report, '>', '/report.txt' or die;
```

```
$old_handle = select $out; # выбрать поток для отчета
```

```
select $report;
```

```
$^ = 'HEAD';
```

```
$~ = 'REPORT';
```

```
# описание форматов для отчета
```

```
while() { # чтение одной записи данных
```

```
($authors, $title, $year, $nick) = split ':';
```

```
write $report; # вывод одной строки отчета
```

```
}
```

```
close $report or die;
```

```
# формат для заголовка страницы
```

```
format HEAD =
```

```
Классические книги по языку Perl
```

```
издательства O'Reilly
```

```
Лист @#
```

```
$%
```

```
-----+-----+-----+-----
```

```
Авторы | Заглавие | Год | Прозвище
```

•

```
format REPORT =
```

$\wedge<<<<<<<<<<<|\wedge<<<<<<<<<<<<<<<<<<|@###|@>>>>>>>>>$

```
$authors, $title, $year, $nick
```

 $\Lambda \lll \lll \lll \lll \lll \lll \lll \lll | \Lambda \lll \lll \lll \lll \lll \lll \lll \lll \lll | \quad || \sim \sim$ 

```
$authors, $title
```

-----+-----+-----

•

\_\_DATA\_\_

Cozens S.:Advanced Perl Programming,2nd ed.:2005:Panther Book

Friedl J.E.F.:Mastering Regular Expressions:1997:Owls Book

• • •

Результатом работы этой программы будет такой отчет, размещенный в файле report.txt:

## Классические книги по языку Perl

издательства O'Reilly

Лист 1

Авторы | Заглавие | Год | Прозвище

Cozens S. |Advanced Perl |2005|Panther Book

|Programming, 2nd ed. | |

Friedl J.E.F. |Mastering Regular |1997| Owls Book

| Expressions | |

---

Schwartz R.L., |Learning Perl, 4th |2005| Llama Book

Phoenix T., |ed. | |

brian d foy | | |

Конечно, изученные в этой лекции средства отчетов не могут сравниться с современными специализированными построителями отчетов, но во многих случаях бывает достаточно форматирования выходных данных в виде простых отчетов, которое в Perl делается достаточно легко, просто и наглядно. Эта лекция была "лирическим отступлением" перед тем как начать углубленное изучение техники программирования на языке Perl.

## Лекция 11. Ссылки

В этой лекции будут изучены ссылки и ссылочные структуры данных, которые играют очень важную роль в Perl, так как позволяют создавать многомерные массивы, массивы записей и различные динамические структуры данных произвольной сложности: очереди, списки, деревья, графы. Кроме того, умение работать со ссылками необходимо для понимания объектно-ориентированного программирования в Perl.

Цель лекции: научиться обращаться со ссылками, объектами ссылок и структурами данных, основанными на ссылках, чтобы применять их при программировании задач со сложными структурами данных.

Ссылки явно или неявно применяются во всех языках программирования. Они позволяют гибко создавать динамические структуры данных неограниченной сложности. Ссылки являются одним из скалярных типов данных языка Perl, наряду с числами и строками. Ссылка (reference) - это информация о том, где при выполнении программы располагается в памяти объект определенного типа. Эту информацию можно использовать для доступа к объекту ссылки (referent). Ссылка - это возможность обратиться к какой-то информации не по имени, которое известно при компиляции, а по ее расположению в памяти при выполнении программы. В отличие от указателей в некоторых других языках, в Perl ссылки реализованы надежно и эффективно. Программист не должен заботиться о явном удалении из памяти объектов ссылок, поскольку занимаемая память автоматически освобождается встроенным сборщиком мусора, когда объекты ссылок перестают использоваться (то есть когда на них больше не указывает ни одна ссылка). Для создания ссылки на существующий программный объект предусмотрена операция взятия ссылки, обозначаемая обратной косой чертой (backslash), которая ставится перед объектом ссылки. В следующем примере показано, как можно получить ссылку на скалярную переменную и сохранить ее в другой переменной:

```
my $scalar = 'Скаляр'; # объект ссылки

my $ref2scalar = \ $scalar; # ссылка на скаляр
```

На рис. 11.1 показан результат сохранения ссылки на значение скалярной переменной в другой скалярной переменной.



Рис. 11.1. Ссылка на скалярное значение

При помощи операции '\' можно брать ссылку на любые объекты, например, на литерал или на любое другое выражение, только при этом значение объекта ссылки нельзя будет изменить. В этом случае при выполнении программы выражение вычисляется, а результат сохраняется в анонимной области памяти, ссылка на которую и возвращается. Например:

```
my $ref2literal = 'Литерал'; # ссылка на литерал

my $ref2expression = \($n1*$n2); # ссылка на выражение
```

Анонимные данные - это значения, не связанные ни с одной переменной, доступ к которым происходит только по ссылке. Ссылка всегда указывает на значение конкретного типа: скаляр, массив, хэш, подпрограмму (или элемент таблицы символов, о которых речи пока не было). На какой именно тип объекта указывает ссылка, можно узнать с помощью функции `get()`, которая возвращает строку, описывающую тип значения объекта ссылки. Например:

```
print ref($ref2scalar); # выведет: 'SCALAR'
```

А что получится, если вывести значение самой ссылки? Ее значение будет преобразовано в строку, содержащую тип объекта ссылки и его адрес в виде шестнадцатичного числа, например:

```
print $ref2scalar; # выведет, например: 'SCALAR(0x335b04)'
```

Обратно преобразовать в ссылку это строковое представление адреса не удастся.

Чтобы получить доступ к значению, на которое указывает ссылка, нужно выполнить разыменование ссылки (dereference). Для этого переменная, содержащая ссылку, заключается в фигурные скобки и перед ней ставится нужный разыменовывающий префикс: \$ для скаляра, @ для массива, % для хэша, & для подпрограммы. (Другими словами, после разыменовывающего префикса, определяющего тип хранящегося в переменной значения, вместо имени переменной записывается содержащая ссылку переменная или выражение, возвращающее ссылку.) Если не возникает неоднозначности в интерпретации выражения, то переменную, хранящую ссылку, в фигурные скобки можно не заключать. Вот примеры разыменования ссылок на скалярные значения:

```
print "${ref2scalar} "; # или: $$ref2scalar

print "${ref2literal} "; # или: $$ref2literal

print "${ref2expression} "; # или: $$ref2expression
```

Значение скалярной переменной при доступе по ссылке, естественно, может изменяться, но попытка с помощью ссылки изменить литерал вызовет ошибку при выполнении программы ("Modification of a read-only value attempted"):

```
${ref2scalar} = 'Новый скаляр'; # вполне законно
```

```
${ref2literal} = 'Новый литерал'; # ОШИБКА!!!
```

Когда на какое-то значение ссылается несколько ссылочных переменных, то обращаться к этому значению можно с помощью любой из них. Значение доступно до тех пор, пока на него имеется хотя бы одна ссылка. Например:

```
my $ref2scalar = \ $scalar; # ссылка на скаляр

my $one_more_ref = $ref2scalar; # копия ссылки на скаляр

# будет выведено одно и то же значение $scalar:

print "${ref2scalar} ${one_more_ref}";
```

На рис. 11.2 показана ситуация, когда несколько ссылок указывают на одну скалярную переменную.

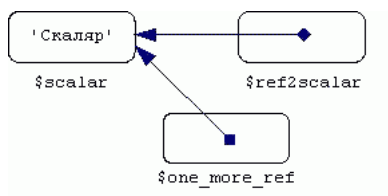


Рис. 11.2. Несколько ссылок на скаляр

Если переменная `$ref2scalar` перестанет ссылаться на `$scalar` (например, после `undef $ref2scalar`), то значение `$scalar` все еще будет доступно через переменную `$one_more_ref`.

Значением объекта ссылки также может быть ссылка (косвенная ссылка) на другой объект, возможно, тоже на ссылку. Ссылка даже может содержать ссылку на саму себя! Таким образом при необходимости можно построить цепочку ссылок любой длины. Например:

```
$value = 'Полезное значение';

$ref1 = \ $value; # ссылка на значение

$ref2 = \ $ref1; # ссылка на ссылку на значение

$ref3 = \ $ref2; # ссылка на ссылку на ссылку на значение
```

Можно организовать многоуровневые косвенные ссылки без использования промежуточных переменных, несколько раз применяя операцию взятия ссылки на значение. Например, создадим такую цепочку ссылок:

```
$ref_chain = \ \ \ $value; # цепочка из трех ссылок
```

Для доступа по такой цепочке ссылок к исходному значению правило разыменования применяется нужное число раз. Например, в цепочке из трех ссылок с помощью трех префиксов `$` мы последовательно получаем доступ к ссылочным переменным, а еще один префикс нужен для доступа к полезному значению:

```
# выведем исходное значение через $ref3:

print ${${${ref3}}}; # или короче: print $$$ref3;

# или через $ref_chain:
```

```
print $$$ref_chain;
```

Если применить функцию `ref()` к переменной, содержащей ссылку на другой объект, то она вернет строку 'REF'. Если преобразовать в строку значение ссылки на ссылку, то будет выведено обозначение ссылочного типа и адрес объекта ссылки, например:

```
print ref($ref_chain); # выведет: 'REF'
```

```
print $ref_chain; # выведет, например: 'REF(0x334e8c)'
```

Подобным же образом можно работать со ссылками на массивы. Ссылка на переменную типа "массив" также создается с помощью операции взятия ссылки:

```
my @array = ('Это', 'список', 'в', 'массиве');
```

```
my $ref2array = \@array; # ссылка на массив
```

Если обращение к массиву будет происходить только по ссылке, то можно обойтись без переменной типа "массив", а создать анонимный массив и присвоить ссылку на него в скалярную переменную. Ссылка на анонимный массив создается с помощью квадратных скобок, в которые заключается список начальных значений безымянного массива:

```
my $ref2anon = [ # ссылка на анонимный массив
```

```
'Это', 'анонимный', 'массив'
```

```
]; # конец присваивания ссылки
```

```
my $ref2empty = []; # ссылка на пустой анонимный массив
```

Анонимные массивы удобно использовать для создания ссылки на копию массива. Для этого существующий массив помещается в квадратные скобки, и его значение будет скопировано в созданный анонимный массив:

```
my $ref2copy = [@array]; # ссылка на копию массива
```

Ссылка на именованный массив и ссылка на анонимный массив изображены на рис. 11.3.

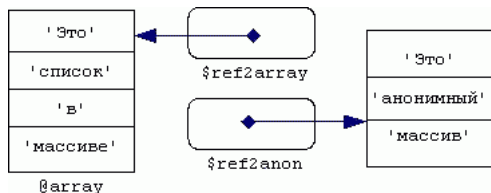


Рис. 11.3.Ссылки на обычный и анонимный массивы

Разыменование ссылки на массив производится аналогично разыменованию ссылки на скалярную переменную, только с использованием префикса массива @:

```
# будет выведено одно и то же значение @array:
```

```
print "@{$ref2array} @{$ref2array}\n";
```

Естественно, что, обращаясь к массиву по ссылке, можно выполнять с ним любые действия, как и с именованным массивом, например:

```
my @array_copy = @{$ref2array}; # копия массива
```

```
@{$ref2array}[0,1] = ('Новый', 'список'); # срез массива
```

Разыменование ссылки на элемент массива оформляется так: перед ссылочной переменной, которая может заключаться в фигурные скобки, указывается префикс скалярного значения \$, а после ссылочной переменной указывается индекс элемента в квадратных скобках. Другими словами, для обращения к элементу массива по ссылке имя массива заменяется ссылочной переменной:

```
print ${$ref2array}[0]; # или: $$ref2array[0]
```



Обращение по ссылке к элементу массива более наглядно записывается с помощью инфиксной операции `->`, слева от которой записывается имя переменной, содержащей ссылку на массив, а справа - индекс элемента массива в квадратных скобках. Операция "стрелка" наглядно представляет ссылку, символы `->` в ней записываются без пробела между ними. Вот пример:

```
# доступ по ссылке к значению элемента массива:
```

```
my $element_value = $ref2array->[0];
```

```
# изменение значения элемента массива:
```

```
$ref2array->[0] = $new_value;
```

Как к обычным скалярным значениям можно обращаться по ссылке к отдельным элементам массива, например:

```
$ref2element = \ $array[0]; # ссылка на элемент массива
```

```
${$ref2element} = $new_value; # изменение элемента массива
```

В элементах массива можно хранить ссылки на другие массивы: это позволяет создавать в Perl многомерные массивы или "массивы массивов", как это делается в языке Java. В этом случае доступ к элементам многомерного массива также обычно записывается с использованием операции "стрелка", которая употребляется нужное количество раз:

```
@{$ref2NxM->[$n]} # вложенный массив
```

```
$ref2NxM->[$n]->[$m] # скалярный элемент двумерного массива
```

```
$ref2NxMxP->[$n]->[$m]->[$p] # элемент 3-мерного массива
```

Для удобства чтения программы допускается не записывать операцию "стрелка" между парами индексов массива в квадратных скобках:

```
$ref2NxM->[$n][$m] # так гораздо симпатичнее!
```

```
$ref2NxMxP->[$n][$m][$p] # а тем более так...
```

Для примера приведем программу создания двумерного массива из трех строк по пять элементов в каждой строке:

```
my $ref2RxC = []; # ссылка на анонимный массив массивов
```

```
for (my $row = 0; $row < 3; $row++) { # цикл по строкам
```

```
$ref2RxC->[$row] = []; # строка: вложенный массив
```

```
for (my $col = 0; $col < 5; $col++) { # по колонкам
```

```
$ref2RxC->[$row]->[$col] = ($row+1).'.'($col+1);
```

```
}
```

```
}
```

Небольшие многомерные массивы удобно создавать, используя вложенные анонимные массивы. Это присваивание создаст такой же массив, что и в предыдущем примере:

```
$ref2RxC = [ # ссылка на двумерный анонимный массив
```

```
[1.1, 1.2, 1.3, 1.4, 1.5], # 1-я "строка"
```

```
[2.1, 2.2, 2.3, 2.4, 2.5], # 2-я "строка"
```

```
[3.1, 3.2, 3.3, 3.4, 3.5] # 3-я "строка"
```

```
]; # конец присваивания ссылки
```

На рис. 11.4 изображен получившийся в результате "массив массивов" (Array of Arrays, AoA), представляющий собой многомерный массив.

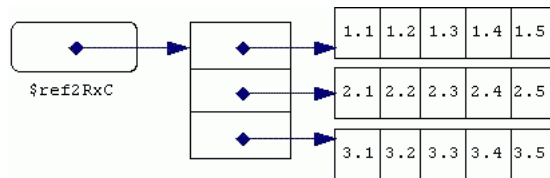


Рис. 11.4. Организация многомерного 'массива массивов'

Для вывода значений многомерного массива обычно используется нужное число вложенных циклов `for` или других циклических конструкций:

# цикл по строкам (элементам массива верхнего уровня)

```
for (my $row = 0; $row < @{$ref2RxC}; $row++) {
```

# цикл по столбцам (элементам вложенных массивов)

```
for (my $col = 0; $col < @{$ref2RxC->[$row]}; $col++) {
```

```
print "$ref2RxC->[$row][$col] ";
```

```
}
```

```
print "\n";
```

```
}
```

В результате выполнения этой программы построчно будет выведено значение всех элементов из массива массивов:

```
1.1 1.2 1.3 1.4 1.5
```

```
2.1 2.2 2.3 2.4 2.5
```

```
3.1 3.2 3.3 3.4 3.5
```

В любой массив можно поместить список ссылок на другие программные объекты, например, таким образом:

```
@reference_list = (\$scalar, \@array, \%hash);
```

Можно записать то же самое более простым способом, поставив операцию взятия ссылки перед списком объектов в круглых скобках:

```
@reference_list = \($scalar, @array, %hash);
```

Списки ссылок на объекты могут, например, передаваться в подпрограмму для изменения перечисленных в списке объектов. Передача аргументов в подпрограммы по ссылке и по значению будет рассмотрена в следующей лекции.

Если попытаться разыменовать ссылку на несуществующий объект, то он автоматически будет создан. В этом случае работает удивительный механизм, называемый автосозданием объекта ссылки (буквально: "автооживление" - *autovivification*). Например, во время обращения по ссылке к элементу массива автоматически создается массив из пяти элементов, ссылка на него присваивается в переменную `$array_ref`, а пятый элемент получает начальное значение:

```
$array_ref->[4] = '5-й элемент'; # присваивание значения
```

```
print ref($array_ref); # вызывает к жизни массив
```

```
print scalar(@{$array_ref}); # из 5 элементов!
```

```
print $$array_ref[4]; # печатаем значение
```

Подобным образом применяя автосоздание объектов, можно создать цепочку ссылок, указывающих на некоторое значение:

```
$$ref = 25; # при попытке присвоить значение

# создаются 2 ссылочных переменных и 1 скалярная

print "$ref $$ref $$$ref $$$ref\n";

# выведет: REF(0x334dd8) REF(0x334e8c) SCALAR(0x334e98) 25
```

Все, что говорилось о ссылках на массивы, применимо к ссылкам на хэши. Ссылка на переменную типа "хэш" получается с помощью операции взятия ссылки:

```
my %hash = ('Хэш' => 'ассоциативный массив');

my $ref2hash = \%hash; # ссылка на весь хэш

print ref($ref2hash); # вернет: HASH
```

Ссылка на анонимный хэш создается с помощью фигурных скобок, в которых записываются начальные значения хэша:

```
my $ref2anon = { # ссылка на анонимный хэш

'language' => 'Perl',

'author' => 'Larry Wall',

'version' => 5.8

}; # конец присваивания ссылки
```

При помощи анонимного хэша удобно создавать копию существующего хэша, чтобы затем работать с ним через ссылку:

```
my $ref2copy = {%hash}; # ссылка на копию хэша
```

Разыменование ссылки на хэш записывается так же, как разыменование ссылки на массив, но с префиксом хэша %. При разыменовании ссылки на хэш переменную, содержащую ссылку, для наглядности можно обрамлять фигурными скобками:

```
# будет выведено одно и то же значение %hash:

print %{$ref2hash}, %$ref2hash;
```

При помощи ссылок с хэшами можно выполнять любые действия, обращая внимание на правильное разыменование ссылки:

```
%hash_copy = %{$ref2hash}; # копия хэша

@hash_slice = @{$ref2hash}{$key1, $key2}; # срез хэша (массив)

@hash_keys = keys %{$ref2hash}; # ключи хэша (массив)
```

Разыменование ссылки на элемент хэша также записывается уже знакомым нам способом, когда перед ссылочной переменной ставится префикс скаляра \$, а после нее - ключ элемента хэша в фигурных скобках. Ссылочная переменная может заключаться в фигурные скобки:

```
${$ref2hash}{ключ} = 'значение'; # изменение значения

print $$ref2hash{ключ}; # доступ к значению элемента хэша
```

В другой форме разыменования ссылки к переменной, содержащей ссылку на хэш, применяется операция "стрелка", после которой в фигурных скобках указывается ключ элемента хэша. Вот так:

```
$ref2hash->{'термин'} = 'определение'; # добавим элемент

$value = $ref2hash->{'Хэш'}; # найдем значение по ключу
```

Если ссылка используется как ключ хэша, она, как и любой ключ хэша, автоматически преобразуется в строку. Такие строки невозможно применять для доступа к объектам ссылки, но они могут служить отличными уникальными ключами хэша, поскольку строковое значение ссылки содержит адрес объекта в памяти, например, 'SCALAR(0x335b04)' или 'ARRAY(0x334dd8)'. Если все-таки требуется использовать ссылки в качестве ключей хэша, то можно воспользоваться модулем Tie::RefHash.

Аналогично созданию "массива массивов" создаются и другие разновидности ссылочных структур данных: массивы хэшей, хэши массивов и хэши хэшей. Ссылочные структуры применяются для структурированного представления взаимосвязанных данных. Для хранения в каждом элементе массива нескольких значений применяется массив хэшей (Array of Hashes, AoH). Вот пример массива, содержащий ссылки на анонимные хэши, в каждом из которых хранятся сведения о каком-либо объекте:

```
my $AoH = [ # этапы "Формулы-1" '2006 года

{grandprix=>'Бахрейна', date=>'2006.03.12'},

{grandprix=>'Малайзии', date=>'2006.03.19'},

{grandprix=>'Австралии', date=>'2006.04.02'},

{grandprix=>'Сан-Марино', date=>'2006.04.23'},

# и так далее...

];

# напечатать хэш, на который ссылается 4-й элемент массива

print "Гран-при $AoH->[3]->{grandprix} $AoH->[3]->{date}";

# выведет: Гран-при Сан-Марино 2006.04.23
```

Для того чтобы ассоциировать с каждым ключом хэша список скалярных значений, применяется хэш массивов (Hash of Arrays, HoA). Приведем пример хэша массивов, где в каждом элементе хэша хранится ссылка на анонимный список ассоциированных значений:

```
my $HoA = { # годы создания языков программирования

1964 => ['SIMULA', 'BASIC', 'PL/1'],

1970 => ['Forth', 'Pascal', 'Prolog'],

1979 => ['Ada', 'Modula-2'],

1987 => ['Perl', 'Haskell', 'Oberon'],

1991 => ['Python', 'Visual Basic']

};

# напечатать список, ассоциированный с 1987 годом

foreach my $language (sort @{$HoA->{1987}}) {

print "$language ";

} # выведет: Haskell Oberon Perl
```

Элементы хэша также могут хранить ссылки на другие хэши, образуя хэш хэшей (Hash of Hashes, HoH). Вот пример описания хэша хэшей, где с каждым поисковым ключом ассоциируется анонимный хэш с информацией об объекте:

```
my $HoH = { # авторы и годы создания языков программирования

'Pascal' => {author=>'Niklaus Wirth', year=>1970},

'Perl' => {year=>1987, author=>'Larry Wall'},
```

```
'C' => {author=>'Dennis Ritchie', year=>1972}

};

# в каком году был создан Pascal?

print $HoH->{'Pascal'}->{'year'}; # выведет: 1970

# кто создал язык Си?

print $HoH->{'C'}->{'author'}; # выведет: Dennis Ritchie
```

Имеющиеся в других языках программирования записи (record) или структуры (struct), в Perl чаще всего представляются в виде хэшей, в которых ключи используются в качестве имен полей и применяются для доступа к значениям полей записи. Для завершающего примера создадим набор записей с информацией о людях. Каждая запись будет анонимным хэшем, а ссылки на записи будут храниться в массиве. В каждой записи дату рождения представим в виде анонимного массива, содержащего год, месяц и день. Вот таким образом:

```
my $family = [ # массив записей о семье

{name => 'Михаил', birthday => [1958, 11, 12]},

{name => 'Ирина', birthday => [1955, 03, 23]},

{name => 'Маша', birthday => [1980, 07, 27]},

{name => 'Миша', birthday => [1981, 11, 28]},

{name => 'Лев', birthday => [1988, 06, 24]}

];

# напечатаем год рождения Маши:

print "$family->[2]->{birthday}->[0]"; # или проще:

print "$family->[2]{birthday}[0]"; # выведет: 1980
```

Подобные структуры легко динамически модифицировать при выполнении программы. Например, добавим в каждую запись новое поле - 'address', в котором сохраним ссылку на запись о месте проживания человека. Адрес оформим в виде анонимного хэша из нескольких полей:

```
# адрес в виде анонимного хэша, в $address - ссылка на него:

$address = {country => 'Россия', index => 641870}; # и т.д.

# добавить поле адреса и поместить туда $address:

foreach my $person (@{$family}) { # пропишем всех

$person->{address} = $address; # по одному адресу

}

# выведем почтовый индекс для Ирины

print "$family->[1]->{address}->{index}\n"; # 641870
```

На рис. 11.5 приведена ссылочная структура данных, которая получилась в результате выполнения программы. Для доступа по ссылкам ко всем элементам этой структуры используется единственная именованная переменная \$family.

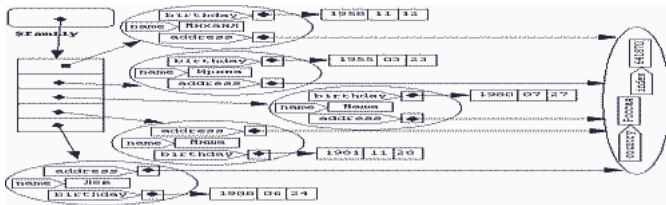


Рис. 11.5.Пример ссылочной структуры данных

С помощью ссылок создаются и другие динамические структуры данных: связанные списки, деревья и графы. Подытоживая все сказанное о ссылках, в таблице 11.1 приводится сводная информация о синтаксических конструкциях при работе со ссылками.

Таблица 11.1. Синтаксические конструкции для работы со ссылками на данные

|   | Скаляр                            | Массив  | Хэш   |
|---|-----------------------------------|---|---|
| Взятие ссылки на объект                                     | <code>\$sref = \ \$scalar;</code> | <code>\$aref = \@array;</code>                        | <code>\$href = \%hash;</code>                     |
| Создание ссылки на анонимный объект                         | <code>\$sref = 'Литерал';</code>  | <code>\$aref = [\$a, \$b];</code>                     | <code>\$href = {\$a =&gt; \$b};</code>            |
| Доступ к значению объекта ссылки                            | <code>\$\$sref</code>             | <code>@{\$aref} @{\$aref}</code>                      | <code>%{\$href} %\$href</code>                    |
| Доступ к значению элемента объекта ссылки                   |                                   | <code>\$aref-&gt;[\$index] \${\$aref}[\$index]</code> | <code>\$href-&gt;{\$key} \${\$href}{\$key}</code> |
| Доступ к срезу объекта ссылки                               |                                   | <code>@{\$aref}[\$i1, \$i2]</code>                    | <code>@{\$href}{\$k1, \$k2}</code>                |
| Значение функции <code>ref(\$ref)</code> для объекта ссылки | SCALAR                            | ARRAY   | HASH  |

Программирующие на Perl на каждом шагу пользуются интерполяцией в строках скалярных переменных и массивов. Иногда требуется включить в строку результат вычисления какого-либо выражения. С помощью ссылок можно интерполировать любое выражение, например, вызов функции. Чтобы включить в строку значение скалярного выражения, его надо заключить в круглые скобки и взять на него ссылку операцией `\`, а затем разыменовать ссылочное выражение как скаляр с помощью префикса `$`. Вот таким образом:

```
$s = "localtime() ${\($x=localtime)} в скалярном контексте";
```

# значение выражения, например: 'Sun Mar 26 20:17:36 2006'

Чтобы включить в строку значение выражения, возвращающего список, его надо заключить в квадратные скобки, организовав ссылку на анонимный массив, а потом разыменовать ссылочное выражение как массив с помощью префикса @. Вот так:

```
$a = "localtime() @[localtime()]" в списочном контексте;
```

# значение выражения, например: '36 17 20 26 2 106 0 84 1'

Ссылки, о которых до этого шла речь, называются жесткими ссылками. Жесткая ссылка (hard reference) - это программный объект, хранящий в памяти адрес референта и тип его значения. В Perl имеется еще одна разновидность ссылок, называемых символическими ссылками. Символическая ссылка (symbolic reference) - это строковое значение, которое хранится в скалярной переменной и представляет из себя имя глобальной переменной:

```
$referent1 = 'Референт'; # объект ссылки
```

```
$symlink = 'referent' . 1; # символическая ссылка
```

# доступ по символической ссылке к значению объекта

```
print ${$symlink}; # будет выведено: 'Референт'
```

Символические ссылки используются значительно реже и бывают нужны, когда требуется во время выполнения программы программно создавать имена переменных в виде строк, чтобы затем получать доступ к их значениям. Использование символических ссылок может приводить к трудно обнаруживаемым ошибкам, поэтому лучше запрещать использование в программе символических ссылок с помощью прагмы `use strict 'refs'`.

В этой лекции изучены ссылки и средства построения с помощью ссылок динамических структур данных. Ссылки дают программисту мощные и гибкие средства доступа к программным объектам. Поэтому многие механизмы работы с данными в Perl организованы с использованием ссылок, и мы будем обращаться к ссылкам в следующих лекциях при изучении подпрограмм, библиотечных модулей и объектного программирования.

## Лекция 12. Подпрограммы

В этой лекции будут изучены подпрограммы - единицы структурирования программного кода на Perl, которые также имеют свои особенности: в формах определения и вызова, в способах передачи и обработки параметров, в вариантах возврата значений. Кроме того, в этой лекции будут рассмотрены принципы определения видимости имен переменных в Perl-программе.

Цель лекции: получить знания, необходимые для разработки на Perl структурированных модульных программ, учитывая своеобразие реализации в нем механизма подпрограмм. Научиться использовать на практике особенности работы с подпрограммами. Изучить способы задания области видимости переменных.

Для моделирования в программе внешнего мира программист создает необходимые сущности в виде переменных, подобно тому как человек обозначает существительными предметы и явления. Подпрограммы позволяют программисту создавать собственные глаголы для именования серии повторяющихся действий или для замены длинной последовательности действий одним идентификатором. Определение подпрограммы напоминает определение термина в словаре, например: "сортировать - расположить элементы набора данных в определенной последовательности путем их сравнения и перестановки". Затем все, кто знает определение термина, могут пользоваться им для указания выполнить соответствующее действие, например: "сортировать список книг по возрастанию названий".

Кроме того, подпрограммы - это средство структурирования программы. Довольно часто программный комплекс разрабатывается методом "сверху вниз", когда сначала определяются крупные части программы и порядок их выполнения без уточнения деталей реализации. Затем в ходе проектирования выясняются способы взаимодействия этих частей и фиксируются их интерфейсы (входные и выходные данные). После чего каждая крупная часть программного проекта подвергается аналогичной декомпозиции на более мелкие составляющие. В результате такого процесса "пошагового уточнения" (stepwise refinement) определяются программные компоненты в виде подпрограмм, выполняющих определенные действия. После чего пишется код подпрограмм, реализующий конкретные алгоритмы. Ну и конечно, применение подпрограмм дает возможность повторного использования (code reuse) готовых и проверенных программ, которые оформляются в виде библиотек модулей проекта.

В процедурных языках программирования традиционно выделяют две разновидности подпрограмм: функции, возвращающие сформированное в результате своей работы значение, и процедуры, не возвращающие значения и вызываемые ради побочного эффекта. В Perl существует только одна разновидность структурных единиц - подпрограммы, всегда возвращающие значение.

Подобно определению термина в словаре, при определении подпрограммы (subroutine definition) с ее именем сопоставляется последовательность действий, которую нужно выполнить. Подпрограммы определяются с помощью ключевого слова `sub`, за которым следует имя подпрограммы и блок, называемый телом подпрограммы, в котором содержатся исполняемые предложения подпрограммы. Имя подпрограммы - это идентификатор, который записывается по тем же правилам, что и имя переменной. В соответствии со сложившимся стилем программирования на Perl имена подпрограмм записываются строчными буквами, а логические составляющие имени разделяются символами подчеркивания. Часто для имен подпрограмм выбирают подходящие по смыслу глаголы. Вот пример определения подпрограммы и ее вызова:

# определение подпрограммы:

```
sub say_hello { # имя подпрограммы

if ($language eq 'ru') { # тело подпрограммы

print 'Здравствуйте!';

} elsif ($language eq 'ja') {

print 'Konnichi-wa!';

} else {
```

```
print 'Hello!';

}

}

$language = 'ru'; # напечатать приветствие по-русски

say_hello; # вызов подпрограммы
```

Определение подпрограмм может располагаться в любом месте программного файла. (Можно даже перемежать определения подпрограмм выполняемыми предложениями программы, поскольку определения пропускаются при выполнении. Но разбираться в логике такой программы будет непросто!) Из соображений практического удобства, определения подпрограмм часто располагают в исходном тексте после основной программы, чтобы вначале ознакомиться с общей логикой программы. Но это дело вкуса: например, разработчики, привыкшие программировать на Pascal или C, располагают определения подпрограмм в начале исходного текста.

В Perl все подпрограммы всегда возвращают значения. Если не указано ничего иного, возвращаемым значением будет значение последнего вычисленного в подпрограмме выражения. В следующем примере функция `dice()` возвращает список из двух случайных чисел от 1 до 6, имитируя бросок игральных костей:

```
sub dice { # бросаем игральные кости:

(int(rand 6)+1, int(rand 6)+1); # два случайных числа

}
```

Также в Perl есть встроенная функция возврата из подпрограммы `return`, которая завершает выполнение подпрограммы и возвращает значение указанного выражения. Так делается в пользовательской функции приветствия:

```
sub greeting { # приветствие в зависимости от времени суток

my $hours = (localtime)[2]; # текущие часы

if ($hours >= 4 and $hours < 12) {

return 'Доброе утро';

} elsif ($hours >= 12 and $hours < 18) {

return 'Добрый день';

} elsif ($hours >= 18 and $hours < 22) {

return 'Добрый вечер';

} else {

return 'Доброй ночи';

}

}
```

```
print greeting(), '!';
```

Если выражение не указано, возвращается пустой список в списочном контексте и неопределенное значение `undef` - в контексте скалярном. В следующем примере функция проверки размера файла `get_file_size()` возвращает неопределенное значение как сигнал об отсутствии файла.

```
sub get_file_size { # узнать размер файла

return -s $file # вернуть размер, в т.ч. 0,
```



```

if -e $file; # если файл существует

return; # файла нет, вернуть undef

}

```

Использование `return` в подпрограммах относится к хорошим привычкам программиста, поскольку делает исходный текст более понятным для того, кто будет читать программу и заниматься ее сопровождением. Ведь хорошо известно, что даже автору потребуются значительное время и усилия, чтобы вспомнить в деталях логику давно написанной им программы.

Помня о том, что списки в Perl - одномерные, становится понятным, что подпрограмма в Perl может возвращать только один список. Например, если в ней записано `return (@array1, @array2)`, будет возвращен объединенный список из элементов `@array1` и `@array2`. Поэтому при необходимости вернуть несколько списочных объектов возвращаются ссылки на них, например: `return (\@array1, \@array2)`.

Пока что в приведенных примерах подпрограммы использовали значение глобальных переменных, но сама идея применения подпрограмм предполагает применение заложенного в них алгоритма для обработки передаваемых им параметров. Когда подпрограмма вызывается с набором аргументов, ей передается специальный массив с предопределенным именем `@_`, содержащий список переданных аргументов. В теле подпрограммы переданные значения доступны в виде элементов массива `@_`, что видно из следующего примера:

```

sub cube { # вычислить куб числа

return $_[0] * $_[0] * $_[0]; # умножить аргумент

}

print cube(2); # будет напечатано 8

```

Количество переданных аргументов можно выяснить, запросив число элементов массива `@_`. Для обработки списка аргументов переменной длины часто используется встроенная функция `shift()`, которая извлекает из массива параметров очередное значение, переданное подпрограмме:

```

print2files($message, $file1, $file2, $file3);

sub print2files { # вывести текст в несколько файлов

my $text = shift; # 1-й параметр - текст

while (@_) {

my $file = shift; # очередное имя файла

open my $fh, ">>$file" or die;

print $fh $text;

close $fh or die;

}

}

```

Если переданные аргументы заданы переменными, то массив параметров `@_` совмещается с переданными аргументами. Это означает, что изменение элементов массива приведет к изменению значений соответствующих переменных в вызывающей программе. Это можно проиллюстрировать следующим (несколько искусственным) примером:

```

sub sum2 { # вычислить сумму 2-х чисел

$_[0] = $_[1] + $_[2]; # поместить сумму в 1-й аргумент

return;

}

```

```
my $a = 1, $b = 2, $sum = 0;

sum2($sum, $a, $b);

print "$a+$b=$sum"; # будет напечатано: 1+2=3
```

Опыт показывает, что изменение значения аргументов ведет к трудно обнаруживаемым ошибкам и осложняет сопровождение программы, поэтому должно использоваться в исключительных случаях и всегда оговариваться в комментариях. Общепринятым способом работы с параметрами подпрограммы является присваивание значения аргументов списку переменных: это предохраняет аргументы от изменения и позволяет работать не с элементами массива, а с удобно названными переменными. Это видно из следующего примера:

```
sub get_file { # считать данные из файла

my ($path, $file) = @_; # присвоить аргументы в переменные

return unless -e "$path/$file"; # авария: файла нет

open my $fh, '<', "$path/$file" or return;

my @lines = <$fh>; # прочитать все строки файла в массив

close $fh or return;

return @lines; # вернуть массив строк файла

}

my @data = get_file('/tmp', 'log.txt');
```

Хотя подпрограмма может изменять значения глобальных переменных в вызывающей программе, требования надежности предписывают всегда передавать исходные данные в подпрограмму в виде аргументов, а результат ее работы получать в виде возвращаемого значения.

По той же причине, по которой подпрограмма не может возвращать несколько списков, она не может получать несколько отдельных списков в виде аргументов. Например, если подпрограмма вызвана так: `subr1(@array1, @array2)`, то ей будет передан объединенный список из элементов двух массивов `@array1` и `@array2`. Поэтому если необходимо передать несколько списочных объектов, то передаются ссылки на них, например: `subr1(\@array1, \@array2)`.

При необходимости можно объявить подпрограмму до ее использования (*forward declaration*), а определение ее отнести в конец программного файла. При объявлении тело подпрограммы не записывается. Например:

```
sub factorial; # вычислить факториал числа
```

В объявлении подпрограмм могут указываться прототипы, о которых речь пойдет немного позднее в этой лекции.

Подпрограммы в Perl вызываются, когда их имя употребляется в каком-либо выражении. В этот момент выполняются определенные в подпрограмме действия, а выражение получает возвращенный подпрограммой результат. Хотя довольно часто возвращаемое подпрограммой значение игнорируется. Про такое обращение к подпрограмме говорят, что она вызвана в пустом контексте (*void context*). Минимальное выражение для вызова подпрограммы состоит из одного имени подпрограммы, и это выражение превращается в предложение программы, когда после него стоит точка с запятой. Вот пример выражения, состоящего только из вызова подпрограммы в пустом (безразличном) контексте:

```
yellow_submarine('We all live in a');
```

Кстати, в пустом контексте можно употреблять любые другие выражения, хотя, конечно, смысл в этом есть далеко не всегда:

```
2 * 2; # результат отброшен, есть смысл, если стоит...
```

```
'Истина где-то рядом'; # ...в конце подпрограммы
```

```
$x++; # используется ради побочного эффекта
```

Обращение к подпрограмме может записываться различными способами - главное, чтобы компилятор Perl мог определить, что встретившийся идентификатор - это имя вызываемой подпрограммы. Дать подсказку об этом компилятору можно по-разному. В ранних версиях Perl при вызове перед именем подпрограммы требовался разыменовывающий префикс &. Например:

```
&sub_without_parameters; # вызов подпрограммы без параметров
```

```
&sub_with_parameters($arg1, $arg2); # и с параметрами
```

В современном Perl эта устаревшая форма вызова с префиксом & допустима и иногда используется. Гораздо чаще обращение к подпрограмме обозначается использованием круглых скобок после имени подпрограммы, даже если она вызывается без параметров. Как в этих примерах:

```
format_c(); # вызов подпрограммы без параметров
```

```
format_text($text, $font, $size); # и с параметрами
```

Чтобы обращаться к пользовательской подпрограмме в стиле встроенных функций без круглых скобок, нужно чтобы определение или объявление подпрограммы было известно компилятору раньше ее вызова.

```
sub circle; # объявление пользовательской функции
```

```
$area_of_circle = circle $radius; # вызов функции
```

```
sub circle { # определение пользовательской функции
```

```
return 3.141592653*$_[0]*$_[0]; # площадь круга
```

```
}
```

В Perl эффективно реализована рекурсия, поэтому традиционные рекурсивные алгоритмы можно оформлять в виде вызова в подпрограмме самой себя. Например, как в классической функции вычисления факториала:

```
sub factorial ($) { # вычислить N!
```

```
my $n = shift;
```

```
return ($n <= 1) ? 1 : $n * factorial($n-1);
```

```
}
```

Для разработки универсальных подпрограмм программисту нужно знать, в каком контексте была вызвана подпрограмма - какого возвращаемого значения от нее ожидают. Для этого в Perl предусмотрена функция wantarray(). Она возвращает истинное значение, если подпрограмма вызвана в списочном контексте, ложное значение, если подпрограмма вызвана в скалярном контексте, и неопределенное значение, если подпрограмма вызвана в пустом контексте. Проверка ожидаемого значения в подпрограмме и примеры ее вызова могут выглядеть так:

```
sub list_or_scalar {
```

```
my @result = fill_result(); # формируем результаты
```

```
if (!defined wantarray) { # пустой контекст -
```

```
return; # не возвращаем значения
```

```
} elsif (wantarray) { # списочный контекст -
```

```
return @result; # возвращаем список
```

```
} else { # скалярный контекст -
```

```
return "@result"; # возвращаем скаляр
```

```
}
```

```
}

list_or_scalar(); # вызов в пустом контексте

my @list = list_or_scalar(); # вызов в списочном контексте

my $scalar = list_or_scalar(); # вызов в скалярном контексте
```

В Perl программисту предоставляется возможность выполнить во время компиляции ограниченную проверку количества и типов параметров у подпрограммы. Это делается с помощью прототипа списка параметров. Для этого в определении и в объявлении подпрограммы после ее имени в круглых скобках указывается прототип. Прототип представляет из себя последовательность разменовывающих суффиксов, определяющих количество параметров подпрограммы и типы их контекстов. Вот несколько примеров определения подпрограмм с прототипами:

```
# определение подпрограммы с 1-м параметром-скаляром

sub list_mp3 ($) {

my $path = $_[0];

# ...

}

# определение подпрограммы с 2-мя скалярными параметрами

sub translate ($$) { # и списком скаляров

my ($from_lang, $to_lang, @words) = @_;

# ...

}

sub generate_test(); # объявление подпрограммы без параметров
```

Для подпрограмм, определенных с прототипами, компилятор контролирует количество передаваемых аргументов и устанавливает ожидаемый подпрограммой контекст для каждого из аргументов. В приведенном далее вызове подпрограммы вместо массива ей будет передано количество элементов массива, поскольку прототип подпрограммы устанавливает скалярный контекст для единственного аргумента:

```
list_mp3 @dirs; # будет передан 1 скаляр: scalar @dirs
```

Перечень символов, применяемых для описания прототипов, с примерами определения подпрограмм приведен в таблице 12.1.

Таблица 12.1. Обозначение прототипов подпрограмм

| Прототип | Требования к параметрам   | Пример определения / описания       | Пример вызова                                       |
|----------|---|-------------------------------------|---|
| ()       | отсутствие аргументов   | sub mytime ()                       | mytime;   |
| \$       | скалярное значение  | sub myrand (\$) sub myrename (\$\$) | myrand 100; myrename \$old, \$new;                  |
| @        | список скалярных значений (поглощает остальные параметры, поэтому употребляется последним в списке) | sub myreverse (@) sub myjoin (\$@)  | myreverse \$a, \$b, \$c; myjoin ':', \$x, \$y, \$z; |
| &        | подпрограмма  | sub mygrep (&@)                     | mygrep {/pat/} \$a, \$b, \$c;                       |

|   |  |   |   |
|---|--|---|---|
| * | элемент таблицы символов (например, дескриптор файла)                  | sub myopen (*\$)                                      | myopen HANDLE, \$name;  |
| \ | взятие ссылки на следующий за ней прототип                             | sub mykeys (\%) sub<br>mypop (\@) sub<br>mypush (\@@) | mykeys %hash; mypop<br>@array; mypush @stack,<br>\$a, \$b;    |
| ; | разделитель обязательных параметров от необязательных (в конце списка) | sub mysubstr (\$\$;\$)                                | mysubstr \$str, \$pos;<br>mysubstr \$str, \$pos,<br>\$length; |

Проверки на соответствие прототипам не выполняются, если подпрограмма вызывается устаревшим способом (с префиксом &), а также для методов и подпрограмм, вызываемых через ссылки.

Скалярные переменные могут хранить ссылки не только на данные, но и на подпрограммы. В операции взятия ссылки имя подпрограммы должно использоваться с разыменовывающим префиксом &, как это показано в следующем примере:

```
$ref2max = \&max; # взятие ссылки на подпрограмму

sub max { # вычисляет максимум из списка значений

my $maximum = shift;

foreach (@_) { $maximum = $_ if ($_ > $maximum); }

return $maximum;

}

print ref($ref2max); # будет выведено: CODE
```

Первый способ обращения к подпрограмме через ссылочную переменную оформляется аналогично обращению к элементу массива или хэша: после имени переменной, содержащей ссылку на подпрограмму, записывается операция разыменования ссылки (->), за которой обязательно указываются круглые скобки (со списком аргументов или без него), которые показывают, что это обращение к подпрограмме:

```
$max_of_list = $ref2max->(@list_of_numbers);
```

Другая форма обращения к подпрограмме с использованием ссылочной переменной предполагает использование префикса &:

```
$max_of_list = &$ref2max(@list_of_numbers);
```

# можно окружить ссылочную переменную фигурными скобками

```
$max_of_list = &{$ref2max}(@list_of_numbers);
```

Вызов подпрограммы без параметров в этом случае можно записывать без круглых скобок, а при использовании -> скобки обязательны (иначе как узнать, что это обращение к подпрограмме?):

```
&$reference_to_procedure; # с префиксом подпрограмм
```

```
$reference_to_procedure->(); # с операцией разыменования
```

Если предполагается, что доступ к подпрограмме будет происходить не по имени, а только по ссылке, то можно определить анонимную подпрограмму и присвоить ссылку на нее в скалярную переменную, через которую будет осуществляться вызов подпрограммы.

```
my $ref2sum = sub { # определение анонимной подпрограммы

my $sum = 0; # вычисляет сумму списка значений

$sum += $_ foreach (@_);

return $sum;
```

```
}; # конец операции присваивания переменной $ref2sum
```

```
print $ref2sum->(1..5), " \n";
```

Ссылки на подпрограммы бывает удобно хранить в массивах, например, когда над одними и теми же данными нужно выполнить целый список преобразований. Примерно так:

```
my @refs = ($ref2read, $ref2calc, $ref2format);
```

```
for (my $i = 0; $i < @refs; $i++) {
```

```
@data = $refs[$i]->(@data); # обработать данные
```

```
}
```

В других случаях ссылки на подпрограммы предпочтительнее поместить в хэш, чтобы в каждом его элементе подпрограмма ассоциировалась с нужным поисковым ключом. Как в этом примере:

```
my %refs2subs = ('SUM' => $ref2sum, 'MAX' => $ref2max);
```

```
print $refs2subs{'SUM'}->(1..3), " \n";
```

В Perl переменные по умолчанию видны в пределах всей программы (точнее, в пределах пакета, но об этом будет рассказано позднее). Практика доказала, что использование глобальных переменных противоречит принципу модульности, поскольку связывает части программы зависимостями от глобальных данных. Поэтому во всех языках программирования предусмотрены средства ограничения видимости переменных. Как уже упоминалось в лекции 2, в Perl это делается с помощью объявления переменных. Чтобы ограничить видимость переменных рамками блока или подпрограммы, нужно объявить для них лексическую область видимости с помощью функции `my()`, как это уже делалось в приводимых ранее примерах. Когда при помощи `my` объявляется несколько переменных, то все они должны заключаться в круглые скобки, как показано ниже:

```
my ($var1, $var2, $var3) = (1, 2, 3); # правильно
```

```
# запись my ($var1=1, $var2=2, $var3=3) ошибочна
```

```
my $var4 = 4, $var5 = 5; # $var5 - глобальная, а не my
```

Чтобы проследить, как изменяются значения переменных, объявленных в главной программе и подпрограммах, внимательно прочитайте следующий пример (скучный, но полезный для понимания):

```
use strict;
```

```
my $var = 'm'; # лексическая $var в main
```

```
print "1(main)='$var'\n"; # выведет: 1(main)='m'
```

```
sub1();
```

```
print "7(main)='$var'\n"; # выведет: 7(main)='z'
```

```
sub sub1 {
```

```
print "2(sub1)='$var'";
```

```
$var = 's'; # изменяется $var из main!
```

```
print "-->'$var'\n"; # выведет: 2(sub1)='m'-->'s'
```

```
my $var = '1'; # изменена $var из sub1
```

```
print "3(sub1)='$var'\n"; # выведет: 3(sub1)='1'
```

```
sub2();
```

```
# снова видима $var1 из sub1
```

```

print "6(sub1):'$var'\n"; # выведет: 6(sub1):'1'

}

sub sub2 { # снова видима $var из main

print "4(sub2):'$var'";

$var = 'z'; # изменяется $var из main!!

print "-->'$var'\n"; # выведет: 4(sub2):'s'-->'z'

my $var = '2'; # изменена $var1 из sub2

print "5(sub2)='$var'\n"; # выведет: 5(sub2)='2'

}

```

Обратите внимание, что лексическая переменная `$var`, объявленная в главной программе, видима в обеих подпрограммах `sub1` и `sub2`, поскольку они статически объявлены в рамках той же программы. Но при выполнении программы в подпрограмме `sub2` не видима переменная `$var`, объявленная в процедуре `sub1`. Из приведенного примера видно, что после объявления в подпрограмме лексических переменных с помощью `my()`, изменения этих переменных не затрагивают других переменных с теми же именами. Поэтому, чтобы избежать нежелательного изменения значений переменных в других частях программы, рекомендуется всегда объявлять для переменных лексическую область видимости. Проконтролировать наличие объявлений для переменных в программе поможет прагма `use strict`. Другая разновидность лексических переменных, описываемых с помощью функции `our`, будет рассмотрена в следующей лекции.

В Perl имеется функция `local()`, также влияющая на область видимости переменных. Многие считают, что более удачным названием для нее было бы `save()`, потому что ее основное назначение - скрыть текущее значение глобальных переменных. Эта функция не создает локальных переменных, а делает "локальными" значения существующих глобальных переменных в текущей подпрограмме, блоке, `eval` или программном файле. Это значит, что после выполнения `local` текущие значения указанных переменных сохраняются в скрытом стеке, и новые значения переменных будут видимы вплоть до выхода из выполняемой подпрограммы, блока или файла, после чего восстанавливаются сохраненные значения переменных. На время действия `local` переменные остаются глобальными, поэтому новые временные значения переменных будут видимы и в вызываемых подпрограммах. Из-за временного характера действия функции `local` иногда говорят, что она описывает динамическую область видимости. Несколько переменных, чьи значения делаются временно скрытыми при помощи `local`, должны заключаться в круглые скобки, как показано ниже:

```

local $_; # временно скрыть значение буферной переменной

local ($global1, $equant) = (1, 2); # правильно

```

Посмотрите, как изменится результат, если переписать предыдущий пример с использованием `local` вместо `my` в подпрограмме `sub1`:

```

$var = 'm'; # ГЛОБАЛЬНУЮ $var можно скрыть через local

print "1[main]='$var'\n"; # выведет: 1[main]='m'

sub1();

print "7[main]:'$var'\n"; # выведет: 7[main]:'s'

sub sub1 {

print "2[sub1]='$var'";

$var = 's'; # изменена $var из main!

print "-->'$var'\n"; # выведет: 2[sub1]='m'-->'s'

local $var = '1'; # значение ГЛОБАЛЬНОЙ $var скрывается

print "3[sub1]#'$var'\n"; # выведет: 3[sub1]# '1'

```

```

sub2();

print "6[sub1]:'$var'\n"; # выведет: 6[sub1]:'1'

}

sub sub2 { # видна ГЛОБАЛЬНАЯ $var из sub1

print "4[sub2]:'$var'";

$var = 'z'; # изменена $var из sub1!

print "-->'$var'\n"; # выведет: 4[sub2]:'1'-->'z'

my $var = '2'; # изменена $var из sub2

print "5[sub2]='$var'\n"; # выведет: 5[sub2]='2'

}

```

Сравнивая эту программу с предыдущим примером, можно отметить следующие отличия.

- 1 Переменную `$var` в главной программе пришлось сделать глобальной, так как `local` не может скрывать лексические переменные.
- 2 Действие `local` распространяется до конца подпрограммы `sub1`, а также на вызываемую подпрограмму `sub2`.
- 3 При выходе из подпрограммы `sub1` действие `local` заканчивается и восстанавливается значение, которое содержала глобальная переменная `$var` до применения к ней `local`.

В современных программах в основном используют функцию `my` для задания переменным лексической области видимости. Оправданное применение функции `local` в Perl обычно сводится к следующим случаям:

- 1 Временное скрытие значения глобальных переменных, в том числе у специальных переменных.
- 2 Временная модификация отдельных элементов массивов и хэшей, даже имеющих лексическую область видимости.
- 3 Создание локальных файловых манипуляторов в версиях Perl до 5.6, не поддерживающих использование лексических переменных для хранения файловых манипуляторов.

С помощью ссылок, подпрограмм и лексических переменных создаются очень интересные информационные объекты, называемые замыканиями (`closure`). Они основаны на известном принципе, что объект ссылки сохраняется до тех пор, пока на него указывает хотя бы одна ссылка. А переменная может хранить ссылку на значение лексической переменной, динамически создаваемой при входе в блок и автоматически уничтожаемой при выходе из него. Это видно из следующего примера:

```

my $ref; # переменная для хранения ссылки

{ # в блоке создается

my $lex_var = 'Суслик'; # переменная $lex_var

$ref = \ $lex_var; # в $ref помещена

} # ссылка на переменную

# $lex_var освобождается при выходе из блока

print "Ты суслика видишь? И я не вижу. А он есть: ";

print ${$ref}; # объект ссылки доступен через $ref

```

Подобным образом можно хранить ссылку на анонимную подпрограмму, из которой будут доступны динамически созданные лексические переменные. Такая подпрограмма, вызванная по ссылке, будет иметь доступ к области видимости этих переменных. Приведем пример простого замыкания:



```

my $ref; # переменная для хранения ссылки

{ # в блоке создается

my $lex_var = 'Верблюд'; # переменная $lex_var

$ref = sub { return $lex_var }; # в $ref помещена

} # ссылка на подпрограмму

# $lex_var освобождается при выходе из блока

print $$ref; # объект возвращается подпрограммой по $ref

```

Замыкания можно создавать динамически при выполнении программы. Приведем пример функции, которая при каждом вызове создает замыкание и возвращает ссылку на него. При этом каждый раз создается новый экземпляр лексической переменной, замкнутый от доступа извне:

```

sub make_closure { # функция создания замыканий:

my ($animal) = @_; # В лексической переменной

# сохраняется аргумент функции

my $ref2closure = sub { # и ссылка на

# анонимную подпрограмму,

return $animal; # которая имеет доступ

}; # к лексической переменной.

return $ref2closure; # возвращает ссылку на подпрограмму

}

# создаем 2 замыкания, сохраняя в них разные значения:

my $camel1 = make_closure('дромадер'); # одногорбый верблюд

my $camel2 = make_closure('бактриан'); # двугорбый верблюд

print $$camel1, ' ', $camel2->(); # доступ по ссылкам

```

В этой лекции изложены основные сведения о подпрограммах в Perl. Мы продолжим изучение подпрограмм в лекции 13, где будет рассказано о библиотечных модулях, и в лекции 14, посвященной объектному программированию на Perl.

```
perldoc perlsub
```

### Лекция 13. Библиотеки, пакеты и модули

В этой лекции рассмотрена модульная организация программ на Perl. Знать ее совершенно необходимо, поскольку типичные программы сами размещаются в нескольких исходных файлах и не обходятся без подключения внешних библиотечных модулей. Правила организации программных единиц и приемы работы с пространствами имен и составляют тему обсуждения в этой лекции.

Цель лекции: освоить приемы использования в программе готовых модулей, а также научиться создавать собственные модули и управлять пространствами имен в программе с помощью пакетов.

Современные языки программирования предоставляют программистам средства для того, чтобы упорядочить свои наработки. Нетривиальная программа обычно представляет из себя некоторое количество файлов с исходными текстами, расположенных в нескольких каталогах. А большие программные комплексы образуют внушительную иерархию подкаталогов, содержащих десятки и сотни программных файлов. Универсальные подпрограммы принято сохранять в отдельных библиотечных файлах, чтобы обращаться к

ним из разных программ. Сходные по назначению процедуры и функции объединяются в библиотеки подпрограмм. Библиотека программ на Perl – это файл, обычно с суффиксом .pl, где хранится произвольный набор подпрограмм для использования в других программах. (Этот суффикс иногда применяется для прикладных программ на Perl, но для них рекомендуется использовать суффикс .plx.) Для примера напомним небольшую программную библиотеку для работы с данными о музыкальных дисках и сохраним ее в файле 'lib/music\_lib.pl':

```
# библиотека подпрограмм 'music_lib.pl'

sub albums_by_artist { # найти альбомы указанного артиста

my ($artist) = @_; # аргумент поиска: артист

my @albums = (); # возвращаемый список

seek DATA, 0, 0; # ищем с начала файла

while () { # просматриваем альбомы в файле

push @albums, $1 if /$artist;(.*?) /; # и выбираем

} # все подходящие

return @albums; # результат: список альбомов

}

# ... другие подпрограммы библиотеки...

__DATA__ # конец библиотечного файла
```

Рассмотрим, как происходит обращение к библиотечным подпрограммам во время выполнения программы.

Чтобы воспользоваться подпрограммой из библиотечного файла, нужно в вызывающей программе загрузить библиотеку командой `do 'file.pl'`. Команда `do` загружает любую Perl-программу из внешнего файла во время выполнения программы. Причем в библиотеке в свою очередь могут загружаться программные файлы по команде `do`. Кроме этого, `do` регистрирует загруженные программы в специальном хэше `%INC`. Если `do` не может найти или прочитать файл, она возвращает неопределенное значение `undef` и присваивает специальной переменной `!` признак ошибки. Если `do` прочитала файл, но не может его скомпилировать, она возвращает `undef` и помещает сообщение об ошибке в специальную переменную `$_`. Если файл успешно скомпилирован, `do` возвращает значение последнего вычисленного выражения.

После того как внешний файл был успешно загружен, можно вызывать подпрограммы из загруженного библиотечного файла, как если бы они были описаны в текущей программе. В нашем случае это будет выглядеть так:

```
do 'lib/music_lib.pl'; # загрузить библиотеку

# вызвать библиотечную подпрограмму

my @albums = albums_by_artist('Elton John');

print "$_\n" foreach(@albums); # напечатать найденный список
```

Обратите внимание, что в нашем примере явно указан путь к подкаталогу с библиотекой, что предполагает запуск программы из определенного каталога. Гораздо лучше сделать так, чтобы расположение библиотек не зависело от местонахождения вызывающей программы. Для этого в Perl имеется специальный массив `@INC`, в котором хранится список каталогов для поиска загружаемых исходных файлов. В этот массив по умолчанию включаются каталоги, где находятся системные библиотеки Perl. Чтобы на время выполнения программы добавить в этот список свои каталоги с библиотеками, можно воспользоваться опцией `-I`, указываемой в командной строке при запуске компилятора `perl`. Например, запустим на выполнение программу и укажем дополнительно искать файлы в каталоге `/Shock/Mike/lib` и его подкаталогах:

```
perl -I/Shock/Mike/lib program.pl
```

Тогда в программе можно указывать путь к библиотекам относительно одного из перечисленных в массиве `@INC` каталогов:

```
do 'music_lib.pl'; # искать библиотеку в списке @INC
```

Другой способ добавить в массив @INC каталог для поиска загружаемых файлов - использовать директиву (прагму) use lib. Для этого в программе перед командой загрузки библиотеки нужно указать ее расположение, например так:

```
use lib('/Shock/Mike/lib'); # добавить путь к списку @INC
```

Команда загрузки внешних программ до считается устаревшей и употребляется все реже и реже. Вместо нее применяется команда require, имеющая целый ряд преимуществ. Эта команда вызывает ошибку, если загружаемая программа не найдена в массиве @INC, компилируется с ошибками или не возвращает истинного значения. Поэтому в конце загружаемого файла должно быть любое выражение, возвращающее истинное значение. По устоявшейся традиции выражение '1;' помещается в последней исполняемой строке файла. Поэтому, чтобы библиотека в нашем примере без ошибок загружалась командой require, в конце библиотечного файла добавим требуемую строку:

```
# ... другие подпрограммы библиотеки...
```

```
1; # вернуть истинное значение для require
```

```
__DATA__ # конец библиотечного файла
```

Команда require также регистрирует загруженные программы в специальном хэше %INC, поэтому не загружает их повторно. Однако использование библиотек и прочих внешних файлов в таком виде рано или поздно приводит к проблеме совпадения имен глобальных переменных и подпрограмм. Разные библиотеки, созданные разными программистами, неизбежно будут содержать одинаковые идентификаторы. Конфликт между совпадающими именами в разных программных файлах можно разрешить с помощью механизма пакетов.

Пакеты используются в Perl для разделения глобального пространства имен на задаваемые программистом подпространства. Отдельные пространства имен позволяют использовать в каждом из них собственный набор идентификаторов, не конфликтующих с одноименными идентификаторами в других пространствах. Пакет объявляется с помощью команды package, за которой указывается имя пакета. Имена пакетов, задаваемые программистом, принято начинать с заглавной буквы, в отличие от системных, которые записываются строчными буквами. Например:

```
package Package; # объявить пакет с именем Package
```

Подпрограммы и глобальные переменные, определенные после команды package, относятся к объявленному пакету. Действие команды package распространяется до конца текущего блока, файла, блока eval или до следующей команды package, начинающей или продолжающей указанный в ней пакет. Каждое употребление команды package означает переключение на соответствующее пространство имен, идентификаторы которого хранятся в собственной таблице имен. Специальная лексема \_\_PACKAGE\_\_ содержит имя текущего пакета. Поясним сказанное таким примером:

```
package Package; # начало пакета Package
```

```
$variable = 'переменная'; # скаляр из пакета Package
```

```
sub subroutine { # подпрограмма из пакета Package
```

```
    return "$variable";
```

```
}
```

```
package Another; # начало пакета Another
```

```
$variable = 'переменная'; # скаляр из пакета Another
```

```
sub subroutine { # подпрограмма из пакета Another
```

```
    return "$variable";
```

```
}
```

```
package Package; # продолжение пакета Package
```

```
@array = (1..5); # массив из пакета Package
```

В любом пакете можно обратиться к переменной или подпрограмме из другого пакета, указав ее полное имя. Полное имя (или квалифицированное имя) каждого нединамического программного объекта в Perl состоит из имени пакета и идентификатора.

Символы `::` разделяют эти две части таким образом:

`$Package::variable` - скалярная переменная из пакета `Package`

`$Another::variable` - скалярная переменная из пакета `Another`

`&Package::subroutine` - подпрограмма из пакета `Package`

`Package::subroutine` - префикс подпрограммы можно не писать

Если глобальные имена не описаны явно в составе какого-либо пакета, то по умолчанию они относятся к пакету `main`. Можно считать, что объявление этого пакета неявно присутствует в начале любой Perl-программы. Поэтому упоминавшиеся до сих пор глобальные переменные, в том числе большинство специальных, на самом деле относятся к этому пакету. Имя пакета `main` обычно не указывается, но при необходимости принадлежность к нему можно указать явно:

```
%pseudo_name = ('Marilyn Monroe' => 'Norma Jean');
```

```
print $main::pseudo_name{'Marilyn Monroe'};
```

Следующие варианты записи имени обозначают одну и ту же переменную из пакета по умолчанию:

```
@main::array # с явным именем пакета main
```

```
@::array # с пустым именем пакета
```

```
@array # без имени пакета
```

Имена пакетов не применяются к лексическим переменным, объявленным с помощью функций `my()` и `our()` и существующим в собственном пространстве имен. Причем область действия переменных, определенных с помощью `my()`, не может распространяться за пределы исходного файла, а переменные, определенные с помощью `our()`, видны в пределах пакета, даже если части пакета определены в разных программных файлах. Вот пример сосуществования одноименных переменных из лексической области видимости и пространства имен пакета:

```
$variable = 'глобальная'; # переменная из пакета main
```

```
my $variable = 'лексическая'; # переменная из текущего блока
```

```
print "$main::variable $variable";
```

```
# будет напечатано: 'глобальная лексическая'
```

В Perl допускается использование пакетов, которые имеют составные имена следующего вида: `Package::Subpackage`. В этом случае имена пакетов образуют иерархию, а исходные файлы должны храниться в соответствующих вложенных каталогах. Составные имена пакетов соответствуют пути, по которому компилятор будет искать файл с исходным текстом программы. Загружать командой `require` исходный файл можно по полному имени файла с указанием подкаталогов, например:

```
use lib("$path/lib"); # добавить путь к списку поиска
```

```
# загрузить внешнюю программу по имени файла
```

```
require 'Package/Subpackage/Program.pm';
```

Если в команде `require` указано полное имя пакета в виде "голового слова" (bareword), а не в виде строки или переменной, то имя загружаемого файла формируется по следующему правилу. Разделитель пакетов `::` в имени пакета заменяется на разделитель каталогов в используемой операционной системе, а к имени файла добавляется суффикс `.pm`. Суффикс `.pm` используется для файлов, содержащих Perl-модули, и подразумевается по умолчанию командами `require` и `use`. Тогда предыдущую команду можно переписать так:

```
# загрузить внешнюю программу по имени пакета
```

```
require Package::Subpackage::Program;
```

```
# вызвать подпрограмму из загруженного пакета
```

```
print Package::Subpackage::Program::subroutine(), "\n";
```

Теперь рассмотрим другой способ обращения к внешним программам - подключение их на этапе компиляции, что часто дает дополнительные преимущества.

Хотя есть немало случаев, когда нужно загружать внешние программы во время выполнения программы, гораздо чаще библиотеки модулей подключаются во время компиляции. Это делается с помощью команды `use`. Преимущество использования `use` по сравнению с `require` заключается в том, что все возможные ошибки выявляются до выполнения программы на этапе компиляции. Кроме того, команда `use` выполняет импорт имен, определенных в подключаемом модуле, после чего их можно удобно употреблять без имени пакета.

```
use Package::Subpackage::Module; # подключить модуль
```

С помощью команды `use` также подключаются многие стандартные модули Perl. Другие примеры использования команды `use` встретятся по ходу этой лекции при обсуждении работы с модулями.

Команды `require` и `use` также применяются для контроля версии Perl, требуемой для компиляции и выполнения программы. Для этого параметром каждой из этих команд должно быть число, интерпретируемое как минимальный номер версии, который нужен, чтобы программа корректно выполнялась. Например:

```
use 5.005; # использовать версию Perl не ниже указанной
```

```
require 5.008007; # использовать Perl 5.8.7 и выше
```

Если не выполняется требование, заданное в `use`, то компиляция завершится аварийно. Невыполнение требования, указанного в `require`, приведет к ошибке во время выполнения программы.

Модуль - это специальным образом оформленная библиотека подпрограмм, предназначенных для многократного использования. Модули появились в Perl, начиная с версии 5, и с тех пор подавляющее большинство универсальных Perl-программ оформляются в виде модулей. В отличие от обычных библиотек, модули имеют внешний интерфейс - ограниченный набор переменных и функций, предназначенных для использования другими программами. Доступ к внешнему интерфейсу модуля в вызывающей программе организуется с помощью механизма импортирования имен, реализованному в стандартном модуле `Exporter`. Приведем пример оформления типичного модуля (сохраненного в файле `Module.pm`):

```
package Module; # пространство имен модуля
```

```
use 5.006001; # использовать версию Perl не ниже указанной
```

```
use strict; # включить дополнительные проверки
```

```
use warnings; # и расширенную диагностику
```

```
our $VERSION = '1.00'; # версия модуля
```

```
require Exporter; # загрузить стандартный модуль Exporter
```

```
our @ISA = qw(Exporter); # неизвестные имена искать в нем
```

```
our @EXPORT = qw( subroutine );
```

```
# имена, экспортируемые по умолчанию
```

```
our @EXPORT_OK = qw( $variable );
```

```
# имена, экспортируемые по запросу
```

```
$Module::variable = 'переменная 1'; # скаляр из модуля Module
```

```
sub subroutine { # подпрограмма из модуля Module
```

```
return "'подпрограмма 1 $Module::variable'";
```

```
}
```

```
1;
```

```
__END__
```

Автоматически сгенерировать скелет нового модуля (а также сопутствующие файлы, необходимые для подготовки модуля к распространению) можно с помощью утилиты `h2xs`, входящей в состав дистрибутива Perl. Например, создать модуль с именем `'Module::Name'` версии `1.00` можно такой командой:

```
h2xs -AX -n Module::Name -v 1.00
```

Если программа, которая обращается к этому модулю, использует только экспортированные по умолчанию имена, то используется форма команды `use` только с именем модуля:

```
use Module; # подключить модуль и
```

```
# импортировать из него имена по умолчанию
```

```
subroutine(); # вызвать подпрограмму &Module::subroutine()
```

В программе, в которой нужно явно затребовать перечисленные имена (с помощью метода `import`), применяется форма команды `use` со списком импортируемых имен:

```
use Module qw($variable); # затребовать импорт нужных имен
```

```
print "$variable\n"; # скаляр $Module::variable
```

Антонимом команды `use` является команда `no`, которая неявно выполняет вызов метода `unimport` для отмены импортированных имен. В команде `use` также можно проверить, что версия подключаемого модуля соответствует требованиям. Для этого после имени модуля указывается минимальная требуемая версия:

```
use Module 1.00; # подключить модуль не ниже указанной версии
```

Помимо процедурного, модули могут иметь объектно-ориентированный интерфейс, который будет рассмотрен в следующей лекции.

В каждой Perl-программе могут присутствовать исполняемые блоки, фактически являющиеся специальными подпрограммами, которые обрабатываются особым образом. Они имеют зарезервированные имена и записываются заглавными буквами: `BEGIN`, `END`, `CHECK`, `INIT`. Каждый из таких блоков может присутствовать несколько раз в любом месте программы. Эти блоки вызываются автоматически в определенное время в начале и в конце выполнения Perl-программы.

Блок `BEGIN` выполняется как можно раньше: во время компиляции сразу после того, как он полностью определен. Если определено несколько блоков `BEGIN`, то они выполняются в порядке их описания. Они используются командой `use` для загрузки внешних файлов во время компиляции программы.

Блок `END` выполняется как можно позже: после того как perl закончил выполнение программы, перед завершением работы интерпретатора. Он выполняется даже в случае аварийного завершения программы. Несколько блоков `END` выполняются в порядке, обратном их размещению в файле. Блоки `END` не выполняются, если при запуске Perl заказана только компиляция (опцией `-c`), или если компиляция завершается аварийно. При работе `END` доступна специальная переменная `$_`, содержащая код завершения программы, который можно изменить.

Блок `CHECK` выполняется после того, как Perl закончил компиляцию программы. Можно определить несколько блоков `CHECK`, тогда они будут выполняться в порядке, обратном их описанию. Блоки `CHECK` выполняются, если Perl запущен с опцией `-c` только для компиляции программы.

Блок `INIT` выполняется перед тем, как интерпретатор начнет выполнение программы, поэтому могут использоваться для инициализации модулей. Несколько блоков `INIT` выполняются в порядке их описания. Поясним последовательность выполнения специальных блоков на коротком примере:

```
print " 8. выполнение 1\n";
```

```
END { print "14. (1-й END)\n" }
```

```
INIT { print " 5. (1-й INIT)\n" }
```

```
CHECK { print " 4. (1-й CHECK)\n" }
```

```
print " 9. выполнение 2\n";

BEGIN { print " 1. (1-й BEGIN)\n" }

END { print "13. (2-й END)\n" }

CHECK { print " 3. (2-й CHECK)\n" }

INIT { print " 6. (2-й INIT)\n" }

print "10. выполнение 3\n";

END { print "12. (3-й END)\n" }

BEGIN { print " 2. (2-й BEGIN)\n" }

INIT { print " 7. (3-й INIT)\n" }

print "11. выполнение 4\n";
```

Сравните результаты запуска этого примера при обычном выполнении и только при компиляции:

Обычное выполнение: Только компиляция (perl -c)

1. (1-й BEGIN) 1. (1-й BEGIN)
2. (2-й BEGIN) 2. (2-й BEGIN)
3. (2-й CHECK) 3. (2-й CHECK)
4. (1-й CHECK) 4. (1-й CHECK)
5. (1-й INIT)
6. (2-й INIT)
7. (3-й INIT)
8. выполнение 1
9. выполнение 2
10. выполнение 3
11. выполнение 4
12. (3-й END)
13. (2-й END)
14. (1-й END)

Использование специальных блоков позволяет программисту гибко организовать контроль над программой на всех этапах ее жизненного цикла - от компиляции до завершения.

В поставке Perl имеется большое число стандартных библиотек модулей. Их описание можно прочитать в документации, обратившись к известной утилите:

```
perldoc perlmodlib
```

Стандартные модули находятся в библиотечных каталогах дистрибутива Perl и подключаются с помощью команды use. Встроенные системные функции находятся в специальном пакете CORE, поэтому, если в области видимости находится одноименная функция, то чтобы обратиться к встроенной функции, нужно вызвать ее по полному имени, например:

```
use Cwd 'chdir'; # подключить стандартный модуль

chdir '/temp'; # вызвать Cwd::chdir()

CORE::chdir '/temp'; # вызвать встроенную функцию chdir
```

Стандартные модули подразделяются на несколько групп. Одна из них - это модули прагм (pragmatic modules), которые контролируют поведение компилятора и исполняющей системы Perl. В качестве примера таких модулей можно привести `constant`, `lib`, `locale`, `strict`, `utf8`, `warnings` и другие. Другая группа - стандартные модули (standard modules), поставляемые вместе с системой программирования Perl. Приведем примеры стандартных модулей: `AutoLoader`, `CPAN`, `Cwd`, `Encode`, `Exporter`, `File::Find`, `Math::BigInt`, `Time::localtime`, `Win32` и многие другие.

Следующая группа - это модули расширения (extension modules), написанные на языке C и предназначенные для взаимодействия с операционной системой. Примерами модулей расширения могут служить `Socket`, `Fcntl` и `POSIX`.

В лекции 1 уже шла речь о Comprehensive Perl Archive Network (CPAN), что можно перевести как "Всеобъемлющая сеть Perl-библиотек", зеркальные сайты которой имеются по всему миру. Среди российских зеркал CPAN можно упомянуть и . CPAN - это огромный архив, где хранятся дистрибутивы Perl для разных операционных систем, документация, программы, библиотеки и модули по самой разной тематике, распространяемые бесплатно. На сайте имеется хорошая система поиска модулей. Кроме того, все модули расклассифицированы по логическим категориям и по именам пакетов, что облегчает поиск схожих модулей. В репозитории CPAN на первом уровне иерархии насчитывается более 600 каталогов, в каждом из которых хранятся сотни модулей. Среди основных категорий модулей можно упомянуть важнейшие:

- [x]. интерфейсы операционных систем (такие как Win32);
- [x]. интерфейсы к системам управления базами данных;
- [x]. пользовательские интерфейсы;
- [x]. интерфейсы к другим языкам программирования;
- [x]. работа с файлами и файловыми системами;
- [x]. обработка строк и текстовой информации;
- [x]. интернационализация и локализация;
- [x]. аутентификация, безопасность и криптография;
- [x]. работа с сетями, WWW, HTML, HTTP, CGI, e-mail;
- [x]. архивирование и сжатие данных;
- [x]. работа с изображениями, чертежами, векторной и растровой графикой;
- [x]. работа с мультимедийными данными;
- [x]. и многое другое.

Поэтому прежде, чем написать свою программу, имеет смысл поискать на CPAN модуль, подходящий для решения возникшей задачи. Во многих случаях "изобретать велосипед" не придется: найденный модуль либо можно сразу использовать, либо взять за основу своей разработки. Поскольку большинство современных Perl-модулей - объектно-ориентированные, они предполагают довольно легкую адаптацию путем определения подклассов, расширяющих возможности описанных в модуле классов. Даже если нужный модуль не найден, можно почерпнуть решения, реализованные в сходных модулях, изучив тексты программ, поскольку все модули распространяются в исходных кодах. CPAN является выдающимся собранием культурных ценностей, коллективной сокровищницей программистской мысли, реализованной в виде готовых к эксплуатации модулей.

Когда требуемый модуль отсутствует в вашей системе, его можно установить из репозитория CPAN. Для этого нужно обратиться к одному из зеркал этого архивного сайта и загрузить с него последнюю версию модуля, которые по традиции распространяются в виде архивов в формате `tar.gz`.

Программисты, работающие с одной из операционных систем семейства Unix, могут устанавливать модули несколькими способами. Самый простой из них - воспользоваться утилитой `cpan` из дистрибутива Perl. Тогда для установки модуля прямо с ближайшего из зеркал сайта CPAN нужно выполнить, например, такую команду:



```
срар -i Module::Name
```

При первом запуске этой утилиты задаются вопросы, касающиеся настроек программы. Затем она загружает и устанавливает указанный модуль. Другой способ - установить модуль вручную, особенно если он уже загружен с сайта CPAN и в данный момент нет доступа в Internet. Для этого нужно распаковать архив с модулем во временный каталог, например такой командой:

```
tar -zxvf Module-Name-1.00.tar.gz
```

Затем нужно выполнить в этом каталоге несколько команд, которые подготовят, протестируют и установят требуемый модуль. Эти команды всегда описаны в файле `README`, поставляемом с модулем. В большинстве случаев нужно выполнить такие команды:

```
perl Makefile.PL
```

```
make
```

```
make test
```

```
make install
```

Если все команды установки отработали без ошибок, модуль будет установлен и зарегистрирован в вашей системе.

То же самое можно выполнить и в среде Microsoft Windows. Для распаковки архивов нужно воспользоваться Windows-версиями архиваторов `tar` и `gzip` или WinRAR. Установка модулей, полностью написанных на Perl, выполняется так же, как в Unix-системах, только утилита компоновки называется не `make`, а `pmake` (ее можно загрузить с сайта Microsoft). Но для некоторых модулей в процессе установки может потребоваться компиляция исходных программ на языке C, а для этого нужно, чтобы в системе был установлен компилятор C (что в Unix-системах является нормой). Можно воспользоваться бесплатно распространяемыми средствами разработки Visual C++ Toolkit и Platform SDK, загрузив их с сайта компании Microsoft (<http://msdn.microsoft.com/visualc/vctoolkit2003/>). Другой вариант - установить свободно распространяемый компилятор Free MinGW GCC for Windows (). После установки C-компилятора также можно пользоваться утилитой `срар`.

Еще проще под ОС Windows устанавливать готовые бинарные дистрибутивы модулей от компании ActiveState, ориентированные на систему программирования ActivePerl. В ее комплект входит менеджер пакетов `ppm`, который выполняет все необходимые действия по загрузке модуля с сервера и его установке. Установка модуля из Internet запускается примерно такой командой ('::' в имени модуля заменяются на '-'):

```
ppm install Module-Name
```

Или можно загрузить архив с модулем вручную, после чего запустить установку из локального каталога без подключения к Internet. Адрес сервера для загрузки готовых Perl-модулей в формате `ppd` можно узнать на сайте .

Библиотеки, пакеты и модули, изученные в этой лекции, позволяют программисту рационально организовать собственные программы, когда они начинают разрастаться в объеме. А изученные приемы работы со стандартными библиотеками и дополнительными модулями помогут рационально воспользоваться компонентами, которые создали, отладили и протестировали другие разработчики.

## Лекция 14. Объектное программирование

Лекция посвящена разработке программ на Perl с использованием объектного подхода. Это общепринятая современная технология программирования, позволяющая бороться со сложностью создаваемых программ путем классификации объектов и моделирования их поведения.

Цель лекции: научиться писать программы на Perl с применением технологии объектного программирования. Освоить способы описания классов и приемы работы с объектами, включая свойства и методы классов.

Технология объектно-ориентированного программирования представляет из себя современный подход к созданию информационных систем, основанный на применении программных объектов, моделирующих состояние и поведение реальных сущностей. Не углубляясь в теорию, можно дать такие неформальные определения основным терминам, применяемым в объектно-ориентированном программировании:

[x]. Класс (`class`) - это именованное описание для однотипных сущностей их неотъемлемых характеристик (атрибутов) и их поведения (методов). Примеры классов: "Личность", "Хоббит", "Маг".

[x]. Объект (object) - это сущность, относящаяся к определенному классу, хранящая набор конкретных значений данных и предоставляющая для их обработки методы, предусмотренные классом. Примеры объектов: "хоббит по имени Фродо Бэггинс", "маг по имени Гэндальф". Синоним термина "объект": экземпляр (instance) класса.

[x]. Атрибут (attribute) - описание характеристики объекта, значение которой будет храниться в объекте. Примеры атрибутов: "имя", "рост". Набор конкретных значений атрибутов является текущим состоянием (state) объекта. Пример значения атрибута: "имя - Гэндальф". Синонимы термина "атрибут": свойство (property), переменная объекта (object variable), переменная экземпляра (instance variable), данные-элементы (member data).

[x]. Метод (method) - это действие объекта, изменяющее его состояние или реализующее другое его поведение. Пример методов: "назвать свое имя", "стать невидимым". Синонимы термина "метод": операция (operation), функция-элемент (member function).

[x]. Инкапсуляция (encapsulation) - это (1) объединение в объекте набора данных и методов работы с ними; (2) принцип ограничения доступа к данным объекта, когда работать с ними можно только через его методы (скрытие данных).

[x]. Наследование (inheritance) - это создание нового класса на основе существующего с целью добавить к нему новые атрибуты или изменить его поведение. Пример наследования: на основании класса "Личность" создаются его подклассы "Хоббит", "Маг", "Эльф" и "Человек", каждый из которых обладает свойствами и поведением "Личности", но добавляет собственные свойства и меняет поведение.

[x]. Полиморфизм (polymorphism) - это различное поведение объектов, принадлежащих к различным классам, при обращении к одинаково названному методу. Пример полиморфизма: в ответ на призыв "К оружию!" гном схватит боевой топор, эльф приготовит лук и стрелы, а хоббит спрячется за дерево.

Но программистам, пишущим на Perl, можно не запоминать эти труднопроизносимые термины. Объектный подход к программированию реализуется в Perl изящно и легко - при помощи уже известных нам понятий и конструкций.

В нынешней версии Perl нет специальных синтаксических конструкций для выражения идей объектно-ориентированной технологии. Поэтому Perl нельзя назвать объектно-ориентированным языком, но он поддерживает объектный подход при разработке программ. Для создания программ с использованием объектов применяются имеющиеся в языке средства, которые сводятся к нескольким простым соглашениям:

[x]. Класс - это пакет, в котором описаны методы, реализующие поведение создаваемых объектов.

[x]. Объект - это переменная (или замыкание), на которую указывает ссылка, связанная с именем пакета.

[x]. Метод - это подпрограмма из пакета, доступ к которой происходит по ссылке на объект, которую он получает в качестве первого аргумента.

[x]. Атрибуты объекта хранятся в динамически создаваемых переменных, чаще всего - в анонимных хэшах.

[x]. Наследование - это поиск методов, не найденных в текущем пакете, в пакетах, перечисленных в специальном массиве @ISA.

Теперь рассмотрим примеры описания классов средствами языка Perl и приемы работы с объектами.

Класс описывается в виде одноименного пакета, в котором размещаются определения методов, реализующих поведение объектов этого класса. Описания одного или нескольких классов сохраняются в виде модуля. Как минимум один из методов класса отвечает за создание объектов класса. Такой метод называется конструктором (constructor) и обычно носит имя new (или его имя совпадает с именем класса). Для хранения атрибутов объекта очень часто применяется анонимный хэш, ключи которого задают имена атрибутов. Первым аргументом конструктор получает имя класса, которое он использует для преобразования ссылки на анонимный хэш в ссылку на объект указанного класса. Это "магическое" превращение выполняется с помощью встроенной функции bless ("благословить"), благодаря которой каждый созданный объект помечается принадлежащим к определенному классу. После этого при обращении к методам объекта они отыскиваются в пакете с таким же именем. Вот как происходит превращение объекта "ссылка" в объект определенного класса:

```
my $class = 'Hobbit'; # имя класса в виде строки

my $object = { }; # ссылка на анонимный хэш,

# где будут храниться данные объекта,

bless($object, $class); # "благословляется" указывать

# на объект класса $class
```

Для примера опишем класс "Личность" (Person), сохранив его в файле Person.pm. Начало описания класса будет выглядеть так:

```
package Person; # класс - это пакет

sub new { # метод-конструктор объектов

my $class = shift; # 1-й параметр ссылка на имя класса

my $self = {}; # контейнер для атрибутов объекта

$self->{name} = ''; # начальные значения атрибутов

bless($self, $class); # "благословить" объект ссылки

return $self; # вернуть ссылку на созданный объект

}
```

Затем в описании класса обычно определяются методы для доступа к атрибутам объекта. Для примера определим метод для доступа (accessor) к атрибуту 'name' ("имя") и метод для изменения его значения (modifier).

```
sub say_name { # метод доступа (accessor) к атрибуту name

my ($self) = @_; # получить ссылку на объект

return $self->{name}; # вернуть значение атрибута

}

sub give_name { # метод изменения (modifier) атрибута name

my $self = $_[0]; # 1-й аргумент: ссылка на объект

$self->{name} = $_[1]; # 2-й аргумент: новое значение

}

1; # истинное значение требуется для use

__END__ # конец описания класса
```

В классе описываются методы для работы с атрибутами объектов класса, причем часто один метод используется для чтения и для изменения значения атрибута. В примере опишем метод для чтения и записи (mutator) свойства 'height' ("рост"):

```
sub height { # метод чтения и записи атрибута height

my $self = shift; # извлечь ссылку на объект

$self->{height} = shift # присвоить новое значение,

if @_; # если передан аргумент

return $self->{height}; # вернуть значение атрибута

}
```

Обратите внимание, что описание класса значительно проще, чем описание традиционного модуля. Для работы с классом не требуется никаких списков экспортирования имен. Вместо этого описываются методы, которые можно рассматривать как сервисы, предоставляемые классом для взаимодействия с каждым из конкретных экземпляров класса. Набор методов для управления поведением объекта называют его интерфейсом. Для работы с объектами класса достаточно знать этот интерфейс, не вдаваясь в детали реализации поведения объектов.

В программе, в которой применяются объекты описанного класса, мы увидим вполне знакомую нотацию, когда подпрограммы вызываются при помощи ссылочных переменных и операции ->. В объектной терминологии это называется обращением к методам

объектов (или отправка сообщения объекту). Приведем пример создания двух объектов одного класса, каждый из которых обладает собственными значениями свойств:

```
# способ обращения к методам через ссылки на объекты

use Person; # будем использовать этот класс

# создать объект класса,

my $hobbit = Person->new(); # вызвав его конструктор

# задать значение свойства, обратившись к методу объекта

$hobbit->give_name('Фродо Бэггинс');

# создать другой объект

my $dwarf = Person->new; # () не обязательны

$dwarf->give_name('Гимли'); # задать значение свойства

# запросить значения свойств, обратившись к методам

print $hobbit->say_name(), ' ', $dwarf->say_name, "\n";
```

Взаимодействие с объектом строится на обращении к его методам. Обращение к методу происходит при помощи ссылки на экземпляр конкретного объекта, и при этом первым аргументом в метод автоматически передается ссылка на этот объект. Например:

```
$hobbit->give_name('Бильбо Бэггинс'); # соответствует вызову:

Person::give_name($hobbit, 'Бильбо Бэггинс');
```

Внутри метода ссылка на экземпляр объекта используется для доступа к данным этого экземпляра и обращения к другим методам. Для обращения к конструктору используется имя класса, так как во время работы конструктора уже существует класс, а экземпляр объекта только должен быть создан конструктором.

Если к ссылке на объект класса `Person` применить функцию `ref()`, то она вернет значение не `'HASH'`, как можно было бы предположить, а `'Person'`! Это результат "благословения" объекта ссылки функцией `bless()`.

```
print "Класс объекта: '", ref($hobbit), "'\n"; # 'Person'
```

Кроме нотации с оператором "стрелка" `->`, традиционно используемой при работе со ссылками, для доступа к методам применяются синтаксические конструкции с использованием косвенных объектов. При использовании этого стиля имя метода стоит перед именем класса или ссылкой на объект, после которой идет список аргументов, иногда заключаемый в круглые скобки. Использование косвенных объектов может сделать текст программы более наглядным и понятным. Приведем пример обращения к объектам в новом стиле:

```
# способ обращения к методам через косвенные объекты

use Person; # используем класс Person

my $magician = new Person; # "этот маг - новая личность"

give_name $magician 'Гэндальф'; # "назовем мага 'Гэндальф'"

my $name = say_name $magician; # "назови себя, маг"

print $name, "\n";
```

В качестве иллюстрации к сказанному на рис. 14.1 изображены языковые конструкции, применяемые при работе с объектами, и их взаимосвязи.

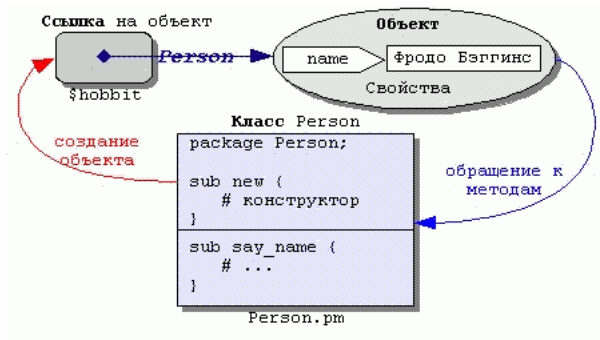


Рис. 14.1. Конструкции объектного программирования в Perl

Наследование - это мощный механизм конструирования нового класса, позволяющий уточнить существующий класс, изменить его поведение родительского класса или добавить к нему новые свойства. В Perl это делается легко и просто: нужно упомянуть имя родительского класса в специальном массиве @ISA текущего класса. Исполняющая система, не найдя вызванного метода в текущем модуле, продолжает его поиск в пакетах, перечисленных в массиве @ISA. Приведем пример описания класса Wizard, производного от класса Person:

```

package Wizard; # класс "Маг"

our @ISA = qw(Person); # является подклассом Person

use Person; # и использует пакет Person

# ... описание методов класса Wizard...

1; # вернуть истину для use
  
```

Смысл наследования - в создании подклассов, изменяющих поведение базового класса. Для этого в дочерних классах описываются новые методы или переопределяются существующие. В качестве примера опишем для класса Wizard новый метод для работы со свойством 'magic' ("тип магии" - белая или черная):

```

sub magic { # магия - вот что отличает волшебника

my $self = shift; # извлечь ссылку на объект

$self->{magic} = shift if @_; # изменить значение

return $self->{magic}; # вернуть значение

}
  
```

Кроме того, переопределим конструктор объектов класса new() так, чтобы он принимал два аргумента для инициализации свойств 'name' и 'magic'. Для создания объекта воспользуемся конструктором родительского класса, затем зададим начальные значения свойств, и, наконец, "дадим благословение" объекту ссылки быть магом:

```

sub new { # конструктор объектов

my $class = $_[0]; # имя класса в 1-м аргументе

my $self = new Person; # маг - это личность

$self->{name} = $_[1]; # задать имя из 2-го аргумента

$self->{magic} = $_[2]; # и тип магии из 3-го

bless($self, $class); # "благословить" мага

return $self; # вернуть ссылку на объект

}
  
```

Вызывающая программа, использующая производный класс, будет выглядеть следующим образом:

```

use Wizard; # подключить производный класс

# создать нового черного мага - Сарумана

my $wizard = new Wizard('Саруман', 'black');

my $name = say_name $wizard; # "назови себя, маг"

print $name, ' ', $wizard->magic(); # 'Саруман black'

print ref($wizard); # тип объекта ссылки - 'Wizard'

```

Естественно, что у объекта класса `Wizard` можно вызывать не только методы собственного класса, но и любые методы, унаследованные из родительского класса `Person`.

В классе может быть описан специальный метод, автоматически вызываемый исполняющей системой при уничтожении каждого объекта. Такой метод называется деструктор (`destructor`), и он должен иметь зарезервированное имя - `DESTROY`. Деструктор вызывается при освобождении памяти, занимаемой объектом: это происходит при выходе из блока, где был создан объект, при удалении последней ссылки на объект функцией `undef($object)` или при окончании программы. Приведем пример шуточного деструктора для класса `Person`, который при удалении объекта направляет прощание в поток `STDERR`, называя имя объекта методом `say_name()`:

```

sub DESTROY {

warn('Прощайте, я ухожу... ' . shift->say_name);

}

```

Деструктор может использоваться, если при окончании работы с объектом нужно выполнить какие-то завершающие действия: например, удалить динамически созданные структуры или сохранить данные объекта в файле. Конструктор в этом случае может считывать сохраненные значения из файла, чтобы присвоить объектам начальные значения.

Анонимные хэши - это самый распространенный, но не единственный способ хранить значения атрибутов объекта. Для этого может применяться массив или даже скалярная переменная, лишь бы при создании объекта в конструкторе это хранилище значений было связано с именем класса функцией `bless()`. Недостатком этого подхода можно считать то, что ограничение доступа к свойствам достигается лишь на уровне соглашения пользоваться только методами объекта. И поскольку существует возможность изменить значение атрибута напрямую, это может нарушить корректную работу программы. Ведь в методе изменение состояния объекта сопровождается необходимыми проверками, чего не происходит при непосредственном изменении атрибута. Тем более, что в некоторых случаях атрибуты вообще должны быть доступны только для чтения (`read-only attribute`). Например, при использовании хэша для хранения атрибутов вполне возможно такое некорректное присваивание:

```
$hobbit->{magic} = 'пёстрая'; # добавлен ошибочный атрибут
```

Для того чтобы надежно обеспечить ограничение доступа к данным, которые хранятся в объекте, применяются замыкания. Чтобы показать, как можно организовать полностью закрытые атрибуты (`private attributes`) с помощью замыканий, напомним класс `Private::Person`. В новой версии класса значения атрибутов также хранятся в анонимном хэше, но при создании объекта возвращается ссылка не на него, а на анонимную подпрограмму доступа к данным. Этой функции будет передаваться имя атрибута (и, возможно, новое значение), а она будет возвращать значение атрибута, используя имя атрибута как ключ поиска в анонимном массиве. Это выглядит так:

```

package Private::Person; # класс "Личность"

sub new { # прототипом может быть

my $invocant = shift; # класс или объект

my $class = ref($invocant) || $invocant;

my $self = { # значения атрибутов:

NAME => '', # имя и

HEIGHT => 0.0 # рост

```

```

};

my $closure = sub { # функция доступа к данным

my $field = shift; # по имени атрибута

$self->{$field} = shift if @_; # изменим и

return $self->{$field}; # вернем значение

}; # объектом будет

bless($closure, $class); # ссылка на функцию

}

# метод доступа к атрибуту name

sub name {

my $self = shift; # ссылка на объект-функцию

&{$self}("NAME", @_); # доступ к скрытому значению

}

# метод доступа к атрибуту height

sub height { # то же, что выше, но несколько короче:

&{ $_[0] }("HEIGHT", $_[1 .. $_[ ] ] )

}

```

```
1;
```

Методы доступа к свойствам объектов получают первым аргументом ссылку на функцию доступа к значениям атрибутов и вызывают ее, передавая ей имя поля и остальные свои аргументы. Приведем пример создания объектов нового класса и обращения к их методам. В вызывающей программе все выглядит так, как будто данные по-прежнему хранятся в анонимном массиве:

```

package main; # вызывающая программа

use Private::Person; # использовать этот класс

my $self = Private::Person->new; # создать объект и

$self->name("Леголас"); # задать значения

$self->height(189); # его атрибутам

# получить доступ к значениям атрибутов объекта

print $self->name, ' ', $self->height, ' ';

print ref($self), "\n"; # тип референта: 'Private::Person'

```

Из примера видно, что имя класса может быть составным, отражая иерархию классов. Поскольку классы – это пакеты, хранящиеся в файле-модуле, то все, что говорилось в предыдущей лекции об именовании модулей, относится и к классам.

Обратите также внимание на то, что конструктор класса `Private::Person` определен так, что он может вызываться с использованием либо имени класса, либо ссылки на существующий объект. Это проверяется в следующей строке:

```
my $class = ref($invocant) || $invocant;
```

Если первым аргументом передана ссылка на объект, то определяется имя его класса, иначе считается, что передано имя класса. Поэтому в программе можно создавать однотипные объекты, обращаясь к методу `new()` существующего объекта. Например, так:

```
my $hobbit = Private::Person->new; # вызов с именем класса

$hobbit->name("Bilbo Baggins");

my $frodo = $hobbit->new; # вызов со ссылкой на объект

$frodo->name("Frodo Baggins");
```

В классе могут быть определены методы, не предназначенные для работы с конкретными объектами. Такие методы называются методами класса или статическими методами. Для обращения к ним, так же как для обращения к конструктору, используется имя класса, а не ссылка на объект. Часто эти методы обслуживают данные, общие для всех объектов класса (то есть объявленные глобально на уровне класса). Подобные данные называются атрибутами класса. В качестве примера опишем класс `Magic::Ring`, где метод класса `count()` будет использоваться для доступа к значению атрибута класса `$Magic::Ring::count`, в котором будет храниться количество созданных волшебных колец.

```
package Magic::Ring; # класс "Магическое Кольцо"

sub new { # конструктор

my ($class, $owner) = @_; # имя класса и значение атрибута

$Magic::Ring::count++; # сосчитать новое Кольцо

bless({owner => $owner}, $class); # "благословить" хэш

}

sub owner { # метод чтения и записи атрибута owner

my $self = shift; # извлечь ссылку на объект

$self->{owner} = shift if @_; # изменить значение атрибута

return $self->{owner}; # вернуть значение атрибута

}

$Magic::Ring::count = 0; # атрибут класса: число Колец

sub count { # метод класса

return $Magic::Ring::count;

}

1; # конец описания класса Magic::Ring
```

В программе, использующей класс `Magic::Ring`, создается набор объектов. При каждом обращении к конструктору увеличивается счетчик созданных магических колец `$Magic::Ring::count`.

```
package main;

use Magic::Ring; # использовать класс "Магическое Кольцо"

my @rings = ();

for (1..3) { # "Три кольца - премудрым эльфам..."

push @rings, new Magic::Ring('эльф');
```



```

}

for (1..7) { # "Семь колец - пещерным гномам..."

    push @rings, new Magic::Ring('гном');

}

for (1..9) { # "Девять - людям Средиземья..."

    push @rings, new Magic::Ring('человек');

}

# "А Одно - всесильное - Властелину Мордора..."

push @rings, new Magic::Ring('Саурон');

# Сколько всего было сделано колец?

print Magic::Ring->count, "\n"; # будет выведено: 20

```

В стандартную библиотеку модулей Perl входит модуль `Class::Struct`, который облегчает жизнь программистам по описанию классов, предоставляя для объявления класса функцию `struct()`. Эта функция генерирует описание класса в указанном пакете, включая методы для доступа к атрибутам класса. Причем помимо имени атрибута она позволяет задавать его тип с помощью разыменовывающего префикса: скаляр (`$`), массив (`@`), хэш (`%`), ссылка на подпрограмму (`&`) или объект. Насколько просто и удобно пользоваться функцией `struct`, можно судить по такому примеру:

```

use Class::Struct; # подключаем стандартный модуль

# описываем класс Performer ("Исполнитель")

struct Performer => { # атрибуты класса:

    name => '$', # "имя" - скаляр

    country => '$', # "страна" - скаляр

    artists => '%', # "артисты" - хэш

};

my $performer = new Performer; # создаем исполнителя

$performer->name('Pink Floyd'); # задаем значения атрибутов

$performer->country('Great Britain');

# заполняем атрибут-хэш:

$performer->artists('David Gilmour', 'гитары, вокал');

$performer->artists('Roger Waters', 'бас-гитара, вокал');

$performer->artists('Nick Mason', 'ударные');

$performer->artists('Richard Wright', 'клавишные');

# описываем класс Album ("Альбом")

struct Album => { # атрибуты класса:

    title => '$', # "название" - скаляр

```

```

year => '$', # "год выхода" - скаляр

tracks => '@', # "композиции" - массив

performer => 'Performer', # "исполнитель" - объект

};

```

```

my $album = Album->new; # создаем альбом

```

```

$album->title('Dark Side of the Moon');

```

```

$album->year(1973);

```

```

# заполняем атрибут-массив:

```

```

$album->tracks(0, 'Breathe');

```

```

$album->tracks(1, 'Time');

```

```

# и так далее...

```

```

$album->performer($performer); # задаем атрибут-объект

```

Чтобы добавить к полученному описанию класса дополнительный метод, достаточно описать его в соответствующем пакете. Вот пример добавления метода `Album::print` и его использования в главной программе:

```

package Album; # переключаемся на нужный пакет

sub Album::print { # и описываем дополнительный метод

my $self = shift;

printf("%s '%s' (%d)\n",

$self->performer->name, $self->title, $self->year);

foreach my $artist (keys%{$self->performer->artists}) {

printf("\t%s - %s\n",

$artist, $self->performer->artists($artist));

}

}

}

```

```

package main; # переключаемся на основную программу

```

```

$album->print; # и вызываем метод объекта

```

В заключение рассмотрим несколько распространенных приемов для работы с классами и объектами.

Функции `bless()` не обязательно передавать имя класса: если второго аргумента нет, она помечает объект ссылки именем текущего пакета. Поскольку `bless()` возвращает значение своего первого аргумента, а подпрограмма возвращает последнее вычисленное значение, то минимальный конструктор может выглядеть так:

```

sub new { # конструктор экземпляров класса

my $self = {}; # контейнер для атрибутов объекта

bless($self); # "благословить" объект ссылки

} # и вернуть ссылку (1-й аргумент bless)

```

При создании объекта удобно сразу задавать начальные значения его атрибутов, передавая аргументы конструктору. Если для инициализации атрибутов использовать хэш, то атрибуты можно задавать в любом порядке, а в конструкторе можно определить значения по умолчанию для незадаанных атрибутов. Например, так:

```
my $language = Programming::Language->new(

NAME => 'Perl', # имя

VERSION => '5.8.7', # версия

AUTHOR = 'Larry Wall' # автор

);
```

Весьма полезно иметь в классе метод, который преобразовывает значения атрибутов объекта в строку. Такой метод обычно называется `as_string()` или `to_string()` и может применяться для отладочной печати состояния объекта. А если его определить в классе-"прародителе", то его можно будет применять к объектам всех унаследованных классов. Если использовать анонимный хэш для хранения значений атрибутов, то такой метод может выглядеть так:

```
sub to_string { # преобразование значений атрибутов в строку

my $self = shift;

my $string = '{ ';

foreach (keys %{$self}) {

$string .= "$_: '$self->{$_}' ";

}

$string .= '}';

return $string;

}
```

Благодаря тому, что Perl - это динамический язык, в нем легко создать класс, в котором свойства объектов добавляются во время выполнения программы. Для этого в классе описываются универсальные методы для работы со свойствами объекта, а затем в ходе выполнения задаются нужные свойства. Например, так:

```
package Human; # класс "Человек"

our @ISA = qw(Person); # это подкласс класса Person

use Person;

sub set { # универсальный метод изменения атрибутов объекта

my ($self, $name, $new_value) = @_;

my $old_value = $self->{$name};

$self->{$name} = $new_value;

return $old_value;

}

sub get { # универсальный метод доступа к атрибутам объекта

my ($self, $name) = @_;

return $self->{$name};

}
```

```

}

1;

package main; # главная программа

use Human; # подключить класс

my $hero = Human->new; # создать героя-человека

$hero->set ('имя', 'Арагорн'); # дать ему имя

$hero->set ('оружие', 'меч'); # и вооружить

```

В этой лекции мы научились работать с объектами. Объектный подход реализован в Perl весьма своеобразно, но понятно и эффективно. Использование этой технологии дает программисту возможность создавать приложения, соответствующие современным требованиям. А сочетание объектного программирования с динамической природой языка позволяет реализовывать оригинальные и эффективные решения.

## Лекция 15. Работа с базами данных

В этой лекции разговор пойдет о программировании баз данных на языке Perl и о средствах взаимодействия с системами управления базами данных, которые имеются в Perl. Основное внимание будет уделено DBI - универсальному интерфейсу доступа к базам данных.

Цель лекции: узнать о средствах работы с базами данных в Perl и научиться применять их в своих программах для доступа к разным типам баз данных - от автономных таблиц до серверов баз данных.

Давно прошли те времена, когда информация хранилась только в простых "плоских" файлах (flat files) в двоичном и текстовом виде. Эволюция систем обработки данных привела к появлению многочисленных баз данных (БД), хранящих информацию в собственных форматах. Основное отличие базы данных от обычного файла с данными заключается в том, что база данных, помимо пользовательской информации, также содержит метаданные, описывающие хранимые в ней сведения. Для работы с большими объемами информации были созданы системы управления базами данных (СУБД), которые теперь работают на серверах баз данных, в настольных и переносных компьютерах - от ноутбуков до карманных компьютеров. Сейчас в большинстве СУБД используются реляционные базы данных, состоящие из таблиц с фиксированным набором колонок (столбцов) и переменным числом строк (записей). Для манипулирования информацией в реляционных базах данных применяется структурированный язык запросов SQL (Structured Query Language). SQL является международным стандартом и поддерживается в большем или меньшем объеме всеми производителями СУБД. Но в последнее время с ними все больше конкурируют объектно-ориентированные и документальные базы данных (например, хранящие информацию в формате XML). Естественно, любая современная система программирования не может обойтись без средств доступа к базам данных. В Perl есть несколько способов работы с базами данных, и мы рассмотрим основные из них: ассоциативные массивы, таблицы-объекты и реляционные базы данных. Примеры работы с базами данных будут основаны на информации о моллюсках, производящих жемчужины (перлы). Каждая запись базы данных будет содержать такие сведения:

[x]. уникальный идентификатор экземпляра (ID) - пятизначное целое число;

[x]. название моллюска по-русски (NAME) - строка длиной до 35 символов;

[x]. латинское название моллюска (LATIN) - строка длиной до 30 символов;

[x]. основные районы обитания (AREA) - строка длиной до 40 символов.

Исходные данные для загрузки в базу данных, которые будут взяты из текстового файла mollusc.txt, имеют такую структуру:

65590;Перловица;Unio pictorum;реки севера России и Скандинавии

56331;Жемчужница речная;Margaritifera margaritifera;север Европы

10616;Морская жемчужница;Pinctada martensii;Японское море

36816;Королевский стромбус;Strombus gigas;Куба

Компактные, простые и быстрые, базы данных в формате Berkeley DB часто используются в операционных системах семейства Unix для хранения системных данных. Существует несколько разновидностей этого формата, которые обобщенно называются файлами DBM (от английского Database Manager). Данные в DBM-файле хранятся в двоичном виде, а логически его можно рассматривать как ассоциативный массив, хранящийся на диске. Средства работы с базами данных этого формата для разных операционных систем можно бесплатно загрузить с сайта <http://www.berkeleydb.com/>. В таких операционных системах, как Linux, FreeBSD или Solaris, Perl часто устанавливается с поддержкой этого формата данных, которая реализована в модуле DB\_File. В операционной системе MS Windows этот модуль потребуется установить дополнительно. (О том, как это делается, речь шла в лекции 13. Если используется дистрибутив Active Perl, установка выполняется командой `ppm install DB_File`.) С помощью этого модуля легко пользоваться базами данных в формате Berkeley DB, потому что с файлом базы данных можно работать как с обычным хэшем. Для этого устанавливается связь между переменной-хэшем и файлом на диске с помощью функции `tie()`, которой указывается, что для доступа к файлу (например, 'file.db') нужно использовать модуль DB\_File. Если указанный файл не существует, он создается. Когда работа с файлом базы данных через хэш-переменную закончена, связь между ними разрывается функцией `untie()`. Это делается так:

```
use DB_File; # подключить модуль для работы с Berkeley DB

my %hash; # через этот хэш будет происходить работа с БД

tie %hash, 'DB_File', 'file.db' or die; # установить связь

$hash{'КЛЮЧ'} = 'ЗНАЧЕНИЕ'; # добавить элемент в хэш и БД

untie %hash ; # разорвать связь между хэшем и БД
```

Формат DBM имеет ограничение, присущее всем ассоциативным массивам: с каждым ключом файла базы данных может ассоциироваться только одно значение. Есть много способов (снова принцип TIMTOWTDI!) обойти это ограничение, и один из них заключается в использовании модуля Storable, который предназначен для организации хранения во внешней памяти массивов, хэшей и других программных объектов. Функция `Storable::freeze()` "замораживает" данные в двоичном виде, например, перед записью на диск, а функция `thaw()` "оттаивает" информацию, восстанавливая первоначальную структуру данных. Мы воспользуемся этими функциями для преобразования данных при создании DBM-файла таким образом:

```
use DB_File; # модули для работы с DBM

use Storable qw(freeze thaw); # и сохранения данных

my %database; # хэш "привязывается"...

tie %database, "DB_File", "mollusc.db" or die; # ...к БД

open my $text, '<', 'mollusc.txt' or die; # файл, откуда

while (my $data = <$text>) { # читаем данные,

    chomp($data); # удаляя \n

    # и разбивая строку на поля по разделителю ';':

    my ($id, $name, $latin, $area) = split(';', $data);

    my %record = ( # заполняем поля записи БД:

        ID => $id, # идентификатор экземпляра

        NAME => $name, # наименование моллюска

        LATIN => $latin, # латинское название

        AREA => $area); # ареал обитания

    my $serialized = freeze \%record; # "замораживаем"

    $database{$id} = $serialized; # и сохраняем запись

}
```

```
close $text; # закрываем тестовый файл
```

```
untie %database; # и базу данных
```

После того как база данных DBM создана, мы можем обрабатывать в ней данные, используя функции работы с хэшами, хорошо знакомые нам из лекции 6. Например, так будет выглядеть поиск по ключу:

```
use DB_File; # модули для работы с DBM
```

```
use Storable qw(freeze thaw); # и сохранения данных
```

```
my %database; # хэш "привязываем"...
```

```
tie %database, "DB_File", "mollusc.db" or die; # ...к БД
```

```
my $id = 65590; # ищем "Перловицу"
```

```
if (exists $database{$id}) { # по идентификатору
```

```
my $serialized = $database{$id}; # считываем и
```

```
my %record = %{ thaw($serialized) }; # "размораживаем"
```

```
printf "%5d %s %s %s\n", # запись БД в хэш
```

```
$id, $record{NAME}, $record{LATIN}, $record{AREA};
```

```
}
```

```
untie %database; # "отвязываем" БД от хэша
```

```
# будет выведено: 65590 Перловица Unio pictorum
```

Для перебора всех записей файла DBM можно пользоваться функциями `keys()` и `each()`, а для удаления записи - применить функцию `delete()`.

С широким распространением персональных компьютеров стал популярным формат баз данных, применяемый в "настольных" СУБД dBASE, Clipper и FoxPro, семейство которых обобщенно называется XBase. Базы данных в этом формате хранятся в таблицах с суффиксом DBF (Database File), а для работы с записями такой таблицы широко применяется произвольный доступ к отдельным записям и перебор записей в цикле. (Хотя работать с ними можно также при помощи языка реляционных запросов SQL.) Одно из средств для работы с DBF-таблицами в программах на Perl - это модуль XBase, который можно загрузить из хранилища модулей CPAN. Он предоставляет объектный интерфейс для создания и изменения баз данных в формате XBase. Например, программа создания таблицы DBF будет выглядеть так:

```
use XBase; # модуль работы с БД в формате DBF
```

```
my $table = XBase->create( # метод создания таблицы
```

```
"name" => "mollusc.dbf", # имя файла
```

```
# имена полей (колонок, столбцов) таблицы:
```

```
"field_names" => ["ID", "NAME", "LATIN", "AREA"],
```

```
# типы данных (N - число, C - строка, D - дата):
```

```
"field_types" => [ "N", "C", "C", "C"],
```

```
# максимальные длины полей:
```

```
"field_lengths" => [ 5, 35, 30, 45],
```

```
# длины дробной части (для чисел):
```

```
"field_decimals" => [ 0, undef, undef, undef]
```

```
);
```

```
$table->close(); # метод закрытия файла БД
```

Далее потребуется программа добавления данных в созданную таблицу из текстового файла. Например, такая:

```
use XBase; # модуль работы с БД в формате DBF
```

```
my $table = new XBase "mollusc.dbf" # конструктор DBF
```

```
or die $table->errstr; # обработка ошибок
```

```
my $recno = 0; # добавляемые записи нумеруются с нуля
```

```
open my $text, '<', 'mollusc.txt' or die; # файл, откуда
```

```
while (my $data = <$text>) { # читаем данные,
```

```
chomp($data); # удаляя \n
```

```
# и разбивая строку на поля по разделителю ';':
```

```
my ($id, $name, $latin, $area) = split(';', $data);
```

```
# добавляем запись, указывая поля в порядке создания
```

```
$table->set_record($recno, $id, $name, $latin, $area);
```

```
$recno++; # и увеличиваем счетчик записей
```

```
}
```

```
close $text; # закрываем тестовый файл
```

```
$table->close(); # и файл базы данных
```

Модуль XBase предоставляет все необходимые методы для работы с таблицами баз данных. Многие из них основаны на возможности произвольного доступа к любой записи DBF-файла по ее номеру. Например, таким образом можно прочитать, изменить или удалить запись по номеру \$record\_number:

```
# считать запись в хэш, с доступом к нему по ссылке:
```

```
my $hash_ref = $table->get_record_as_hash($record_number);
```

```
# изменить значение поля NAME на
```

```
$table->update_record_hash($record_number, 'NAME' => $new);
```

```
# пометить запись как логически удаленную
```

```
$table->delete_record($record_number);
```

```
# восстановить логически удаленную запись
```

```
$table->undelete_record($record_number);
```

По поводу двух последних операций нужно сделать следующее пояснение. Дело в том, что записи в DBF-файле не удаляются физически, а только помечаются как удаленные. "Логически" удаленные записи игнорируются при обработке данных, но существуют в таблице "физически". Поэтому запись, помеченную как удаленная, можно восстановить для дальнейшей обработки. Один из способов прочитать записи таблицы - выбрать их во временный список записей, называемый курсором, откуда последовательно извлекать их в цикле. Это делается так:

```
my $cursor = $table->prepare_select("NAME", "LATIN", "AREA");

while (my @record = $cursor->fetch) { # прочитать запись

    print "@record\n"; # обработать запись

}
```

В модуле XBase реализовано много других методов для работы с DBF-файлами и дополняющими их индексными файлами, которые предназначены для организации быстрого поиска записей в таблице.

Но разработчики программного обеспечения давно пришли к выводу, что вместо специфических форматов данных и операций по их обработке (без которых, конечно, иногда нельзя обойтись) гораздо перспективнее применять универсальные подходы, основанные на унифицированном доступе к базам данных на базе языка SQL.

Унификация доступа к реляционным базам данных основана на разделении программного механизма доступа на несколько логических слоев. Первый слой предоставляет программисту стандартный набор операций для подключения к источнику данных и обработки данных из этого источника с помощью запросов на языке SQL. Второй слой отвечает за взаимодействие с конкретными базами данных с учетом их особенностей. Взаимодействие с конкретным источником данных возлагается на драйвер базы данных, который выступает посредником между первым слоем механизма доступа и базой данных, скрывая от программиста технические детали взаимодействия и специфические особенности БД. Драйверы баз данных обычно разрабатывают производители СУБД для своих продуктов. На этих принципах многослойной архитектуры основаны такие широко известные универсальные интерфейсы к базам данных, как ODBC (Open DataBase Connectivity) и JDBC (Java DataBase Connectivity).

Аналогичную архитектуру имеет и DBI (DataBase Interface) - основной интерфейс для доступа к базам данных в Perl. Основным компонентом этого интерфейса является модуль DBI, предоставляющий унифицированные сервисы для взаимодействия с базами данных. Благодаря методам модуля DBI программист получает в свое распоряжение единый инструмент для работы с самыми разными базами данных: и теми, что находятся на этом же компьютере, и теми, что располагаются на удаленном сервере баз данных. Модуль DBI во время работы загружает нужные компоненты, модули драйверов конкретных баз данных (DataBase Driver, DBD), например: DBD::DB2, DBD::InterBase, DBD::mysql, DBD::Oracle, DBD::Sybase. Доступ к любой базе данных при помощи DBI выполняется в несколько этапов. Перечислим основные из них.

1 Соединение с базой данных выполняется конструктором connect() класса DBI, которому передается строка с описанием источника данных, имя пользователя и пароль, а кроме того, дополнительные параметры:

```
$dbh = DBI->connect($data_source, $user, $password, \%parms);
```

В описании источника данных (data source) указывается драйвер базы данных и необходимые для его работы параметры. При успешном соединении с СУБД этот метод возвращает манипулятор базы данных (database handler), через который в дальнейшем выполняется взаимодействие с базой данных.

2 Подготовка команды к базе данных выделяется в отдельный этап, поскольку это действие требует значительных ресурсов СУБД. Подготовка команды выполняется методом prepare() манипулятора базы данных, которому передается строка, содержащая команду языка запросов SQL:

```
$sth = $dbh->prepare($sql_statement);
```

В команде SQL могут присутствовать слоты (placeholders), в которые при выполнении команды будут подставлены конкретные значения данных. Эта схема похожа на подстановку значений в поледержатели формата отчета. Подготовленная команда доступна через манипулятор команды (statement handler), возвращаемый методом prepare(), и может выполняться многократно.

3 Выполнение команды может производиться несколькими методами. Подготовленную ранее команду выполняет метод командного манипулятора execute(), которому могут передаваться значения для подстановки в выполняемое SQL-предложение:

```
$sth->execute(@bind_values); # выполнить со списком значений
```

Или же SQL-команду можно выполнить без предварительной подготовки методом do() манипулятора базы данных:

```
$dbh->do($sql_statement); # выполнить команду без подготовки
```

4 Обработка полученных данных может выполняться одной из многочисленных команд, предоставляемых интерфейсом DBI.

5 Отсоединение от базы данных выполняется методом disconnect() манипулятора базы данных, который производит необходимые завершающие действия и освобождает используемые ресурсы:



```
$dbh->disconnect; # отключиться от БД
```

Приведенная схема проста и логична, поэтому работа с базами данных через DBI быстро осваивается программистами. Но прежде чем перейти к примерам использования DBI, нужно сделать еще несколько пояснений.

В языке структурированных запросов SQL используется небольшой набор команд, но они позволяют выполнять все необходимые действия над информацией в базе данных. Основные команды SQL: создание базы данных (CREATE), добавление записей (INSERT), их изменение (UPDATE) и удаление (DELETE), а также выборка записей (SELECT) по указанному условию. Изучение языка SQL выходит за рамки этого курса, поэтому в примерах будут применяться только самые простые их формы, и смысл этих команд будет понятен из контекста.

Слоты для подстановки параметров в SQL-команду обозначаются знаками вопроса '?' и выглядят таким образом:

```
$sth = $dbh->prepare(
    'SELECT name, area FROM mollusc WHERE id>? AND id
```

При выполнении этой команды с параметрами 1000 и 9000 будут выбраны записи со значениями колонки id в заданном диапазоне. При подстановке значений аргументов в команду слоты заполняются слева направо:

```
$sth->execute(1000, 9000); # подставить числа вместо ?
```

После подстановки значений будет выполнена команда, означающая "выбрать значения столбцов name и area из таблицы mollusc у тех записей, где значение столбца id больше 1000 и меньше 9000":

```
SELECT name, area FROM mollusc WHERE id>1000 AND id<9000
```

Кроме средств выполнения SQL-команд механизм DBI предоставляет множество методов для выборки из базы данных информации в виде массивов или хэшей для более удобной обработки в программе на Perl. Более подробно с ними можно познакомиться, если почитать системную документацию, выведенную по команде

```
perldoc DBI
```

Покажем приемы работы с интерфейсом DBI на примере класса доступа к уже знакомым DBF-файлам - модуля DBD::XBase. Этот модуль нужно установить описанным ранее способом прежде, чем работать с базами данных в формате XBase. В первом примере программа создает таблицу базы данных SQL-командой CREATE:

```
use DBI; # использовать DBI

my $path = '.'; # каталог, где расположены таблицы БД

my $table = 'mollusc'; # DBF-файл

# подсоединиться к БД, используя драйвер DBD::XBase

my $dbh = DBI->connect("dbi:XBase:$path")

or die $DBI::errstr;

# создать таблицу определенной структуры

$dbh->do("CREATE TABLE $table (id INT,

name CHAR(35), latin CHAR(30), area CHAR(45))");

$dbh->disconnect; # отсоединиться от БД
```

Следующая программа в цикле заполняет созданную таблицу данными из текстового файла, добавляя в нее записи SQL-командой INSERT:

```
use DBI; # используем DBI

my $path = '.'; # каталог с таблицами БД
```

```
my $table = 'mollusc'; # DBF-файл

# подключаемся к БД, используя драйвер DBD::XBase

my $dbh = DBI->connect("dbi:XBase:$path")

or die $DBI::errstr;

# подготовим SQL-команду для многократного выполнения

my $sth = $dbh->prepare("INSERT INTO $table

(id, name, latin, area)

VALUES (?, ?, ?, ?)")

or die $dbh->errstr();

# в цикле читаем строки для загрузки в БД

open my $text, '<', 'mollusc.txt' or die; # файл, откуда

while (my $data = <$text>) { # читаем данные,

chomp($data); # удаляя \n

# и разбивая строку на поля по разделителю ';':

my ($id, $name, $latin, $area) = split(';', $data);

# добавляем запись, подставляя значения в команду

$sth->execute($id, $name, $latin, $area) or die;

}

close $text; # закрываем тестовый файл

$dbh->disconnect; # отсоединяемся от БД
```

Далее можно выполнять различные действия с данными в таблице, используя команды SQL, как это сделано в программе, где изменяются значения перечисленных колонок в записи с указанным идентификатором и удаляется запись по уникальному номеру:

```
use DBI; # использовать DBI

my $path = '.'; # каталог, где расположены таблицы БД

my $table = 'mollusc'; # DBF-файл

# соединиться с БД, используя драйвер DBD::XBase

my $dbh = DBI->connect("dbi:XBase:$path")

or die $DBI::errstr;

# изменить запись с указанным идентификатором,

# заменяя значения перечисленных полей на новые

$dbh->do("UPDATE $table SET name=?,area=? WHERE id=",

undef, 'Жемчужная пинктада', 'Австралия', 89147) or die;

# удалить запись с идентификатором 93749
```

```
$dbh->do("DELETE FROM $table WHERE id=93749") or die;
```

```
$dbh->disconnect; # отсоединиться от БД
```

Для выборки данных из таблицы используется SQL-команда SELECT, в которой можно указывать, данные из каких колонок записи нужно включить в выборку, а также по какому условию отбирать строки таблицы:

```
use DBI; # использовать DBI
```

```
my $path = '.'; # каталог, где расположены таблицы БД
```

```
my $table = 'mollusc'; # DBF-файл
```

```
# соединиться с БД, используя драйвер DBD::XBase
```

```
my $dbh = DBI->connect("dbi:XBase:$path")
```

```
or die $DBI::errstr;
```

```
# выбрать у всех строк таблицы указанные поля
```

```
my $sth =
```

```
$dbh->prepare("SELECT name,area FROM $table WHERE id>?")
```

```
or die $dbh->errstr;
```

```
$sth->execute(1000) or die $sth->errstr(); # выполнить команду
```

```
while (my @row = $sth->fetchrow_array) { # и напечатать
```

```
print "@row\n"; # выбранные строки
```

```
} # в цикле по одной
```

```
$dbh->disconnect; # отсоединиться от БД
```

Для отображения информации из базы данных можно разработать клиентское приложение с графическим интерфейсом, используя библиотеку Perl/Tk, как это показано на рис. 15.1.

|       |                         |                             |                        |
|-------|-------------------------|-----------------------------|------------------------|
| 65530 | Перловица               | Unio pictorum               | реки севера России и С |
| 66331 | Жемчужница речная       | Margaritana margaritifera   | реки северной Европы   |
| 10616 | Морская жемчужница      | Pinclada martenzi           | Японское море          |
| 89147 | Жемчужница планктонная  | Pinclada margaritifera      | Австралия              |
| 36816 | Королевский стромбус    | Strombus gigas              | Куба                   |
| 68618 | Сияющий халиотис        | Haliotis fulgens            | Калифорния             |
| 73151 | Радужный халиотис       | Haliotis iris               | Новая Зеландия         |
| 88039 | Стромбус Листера        | Strombus listeri            | Бенгальский залив      |
| 76564 | Наутилус помпиллюс      | Nautilus pompilius          | Филиппины              |
| 53447 | Миддендорфова перловица | Middendorffinaa ussuriensis | р. Уссури и ее притоки |

Рис. 15.1.Клиентская программа на Perl/Tk для работы с базой данных

Интерфейс DBI привлекает программистов тем, что время и усилия, потраченные на его изучение, окупаются сторицей, поскольку, научившись работать с одной базой данных, можно применять эти знания при работе со всеми остальными, включая "тяжеловесные" СУБД, которые выполняются на специализированных серверах. Сервер баз данных обычно находится на выделенном компьютере, а взаимодействие с ним строится по технологии "клиент-сервер". Это означает, что сервер принимает запросы, поступающие от пользовательских программ, выполняет указанные в запросе действия по обработке информации в базе данных, а затем отправляет результат обработки клиенту. Для повышения производительности, распределения нагрузки и обеспечения непрерывности работы такие СУБД объединяются в кластеры серверов баз данных, которые могут состоять из большого числа мощных компьютеров. Для работы с конкретной системой управления базой данных потребуется установка драйвера для этой СУБД. В хранилище модулей CPAN найдутся драйверы для всех основных серверов баз данных: IBM DB2, MS SQL Server/Sybase, Oracle, PostgreSQL и многих других. Помимо высокой скорости обработки больших объемов данных, СУБД предоставляют программисту дополнительные возможности по обработке информации. Вот основные из них.

1 Реализация языка манипулирования данными позволяет в запросе использовать объединения нескольких таблиц (JOIN), предусмотренные в стандарте языка SQL.

2 В SQL-запросах можно использовать подзапросы для задания дополнительных условий выборки.

3 Согласованность и непротиворечивость данных при изменении нескольких таблиц достигается при помощи использования механизма транзакций.

4 Часто выполняемые действия над информацией в базе данных можно программировать на встроенном языке базы данных в виде хранимых процедур. Когда такие процедуры вызываются в SQL-команде, то они будут эффективно выполняться на сервере.

5 Для реакции на события, возникающие при обработке информации в базе данных, можно использовать специальные хранимые процедуры - триггеры.

6 Для быстрого поиска и выборки может применяться индексация данных.

7 Доступ к информации в базе данных контролируется системой разграничения доступа СУБД на основе парольной защиты.

Взаимодействие с сервером баз данных с помощью DBI будет показано на примере работы со свободно распространяемой СУБД PostgreSQL Database Server, доступной для всех основных вычислительных платформ, включая Linux и MS Windows. Свежий дистрибутив PostgreSQL всегда можно загрузить с сайта [www.postgres.org](http://www.postgres.org), а ее установка с помощью программы-мастера не вызовет трудностей даже у начинающего программиста. Далее нужно уже описанным способом установить драйвер DBD::Pg. Кстати, СУБД PostgreSQL демонстрирует еще одно применение языка Perl: она позволяет использовать Perl для программирования хранимых процедур наряду с SQL и рядом других языков.

После установки драйвера можно выполнить предыдущие примеры из этой лекции с использованием СУБД PostgreSQL, внося в них минимальные изменения. В первую очередь изменятся параметры соединения с базой данных, где мы должны указать другой DBD-драйвер (Pg), имя сервера, имя базы данных, имя пользователя и пароль для доступа к СУБД:

```
my $host = 'localhost'; # имя сервера

my $dbname = 'postgres'; # имя базы данных

my $user_name = "postgres"; # имя пользователя

my $password = "SECRET"; # пароль пользователя

my $dbh = DBI->connect(

    "dbi:Pg:dbname=$dbname;host=$host", # источник данных

    $user_name, $password);
```

После этой модификации программа создания таблицы успешно отработает с СУБД PostgreSQL и создаст в указанной базе данных таблицу 'mollusc'. И другие примеры из этой лекции, использующие интерфейс DBI, также будут работать с PostgreSQL или другой СУБД, после того как их настроят на работу с новым источником данных. Конечно, если применять специфические SQL-команды и другие средства программирования, использующие особенности конкретного сервера баз данных, то адаптация программ для работы с другой СУБД потребует гораздо больше усилий.

С помощью DBI возможно работать не только с традиционными базами данных, но и с файлами в самых разных форматах, в чем можно убедиться, обратившись к хранилищу модулей CPAN. Например, существуют драйверы DBD для работы с электронными таблицами (DBD::Excel), поисковыми системами (DBD::Amazon, DBD::google), иерархическими каталогами LDAP (DBD::LDAP) и универсальными интерфейсами доступа к данным (DBD::ADO, DBD::JDBC, DBD::ODBC).

Часто для преобразования данных из одного формата в другой используется текстовый формат CSV (Comma-Separated Values), в котором поля данных разделены запятыми, а в первой строке перечислены имена полей. Если установить драйвер DBD::CSV и несколько сопутствующих модулей (DBD::File, SQL::Statement и Text::CSV\_XS), то с CSV-файлом можно работать как с таблицей базы данных, что часто бывает очень удобно.

Для преобразования данных также можно использовать модуль DBD::RAM, позволяющий создавать в оперативной памяти таблицы базы данных и импортировать в них информацию из различных источников данных, например: INI-файлы, файлы в формате XML, данные в формате CSV, записи с фиксированными полями и даже каталоги с MP3-композициями. Затем эти таблицы можно обрабатывать с помощью SQL-команд, после чего экспортировать в исходный или другой формат.

При разработке информационных систем средства доступа к базам данных составляют лишь один из уровней программного комплекса. Для работы с данными сложной структуры часто создают специальный класс, за объектным интерфейсом которого от пользователя скрываются конкретный формат хранения данных и возможные преобразования. Если потребуется перейти на хранение

информации в другой базе данных, в этом классе изменится только реализация методов доступа к данным, а использующие этот класс программы останутся неизменными. Подобные приемы повышают гибкость программной системы и облегчают ее модификацию.

Работа с базами данных - это будничным труд большинства программистов. Язык Perl помогает им в этом, предоставляя удобные средства доступа ко всем распространенным СУБД, настольным базам данных и многим экзотическим источникам данных.

## Лекция 16. Взаимодействие процессов

В этой лекции обсуждаются вопросы выполнения программ в многозадачной среде: пользовательские программы запускают на выполнение внешние программы, программы могут порождать параллельно выполняемые процессы, в рамках выполняемой программы может быть запущено несколько потоков управления. Все эти модели программирования поддерживаются языком Perl и будут рассмотрены в этой лекции.

Цель лекции: познакомиться со средствами языка Perl, связанными с межпроцессным взаимодействием, и научиться применять их при разработке собственных программ на языке Perl, выбирая наиболее подходящую из моделей параллельного программирования.

Современные операционные системы в том или ином виде поддерживают многозадачность (multitasking) даже на однопроцессорных компьютерах, не говоря уже о многопроцессорных системах. Операционная система (ОС) производит запуск системных и пользовательских программ в виде независимых процессов (process), выделяя для каждого из них отдельный участок оперативной памяти и другие ресурсы. Каждый процесс нумеруется своим уникальным числовым идентификатором процесса (Process Identifier, PID). Специальные модули ядра операционной системы организуют переключение процессора на обслуживание выполняющихся программ, оптимизируя распределение между ними процессорного времени. При этом работающие процессы могут инициировать выполнение других процессов, порождать зависимые подпроцессы (subprocess) в отдельной области памяти или запускать подпроцессы в области памяти основного процесса. Примерами программ, основанных на использовании подпроцессов, могут служить различные серверные программы: почтовые и web-серверы, серверы приложений и баз данных.

Во время выполнения процессы могут взаимодействовать между собой различными способами. Они могут иметь доступ к разделяемой области памяти, организовывать программные каналы (pipe), посылать друг другу сигналы (signal), обмениваться данными через сокеты, совместно использовать файлы и применять другие средства межпроцессного взаимодействия (Inter-Process Communication, IPC). При этом часто один процесс ожидает окончания выполнения каких-либо действий в другом процессе: про такую ситуацию говорят, что процессы выполняются синхронно (synchronous), то есть согласованно. В других случаях требуется, чтобы процессы выполнялись асинхронно (asynchronous), то есть одновременно и независимо друг от друга. В определенный момент процесс может перейти от асинхронного выполнения к синхронному, то есть перейти в ожидание для синхронизации с другим процессом.

Реализация этих механизмов сильно зависит от конкретной операционной системы, поэтому некоторые стандартные средства языка Perl, связанные с управлением процессами, ориентированы на работу в определенном операционном окружении. Кроме того, имеются специализированные Perl-модули для работы с процессами в операционных системах, соответствующих стандарту POSIX, или в ОС MS Windows. Конечно, в этой лекции нам удастся обсудить только основные средства языка Perl, касающиеся обширной темы межпроцессного взаимодействия. Приводимые примеры намеренно сделаны максимально простыми, чтобы продемонстрировать основные подходы к управлению процессами, избегая особенностей, которыми изобилует многозадачное программирование.

В Perl имеется операция выполнения программы, которая обозначается обратными апострофами (backticks) или синонимом - конструкцией qx(), упоминавшейся в лекции 7. Она предназначена для получения результатов выполнения внешней программы. Эта операция пытается выполнить любую внешнюю программу, ожидает окончания ее работы и возвращает то, что программа выводит в свой поток стандартного вывода. Например, так в операционных системах Linux или MS Windows можно выполнить команду dir, выводящую список файлов в текущем каталоге:

```
my $file_list = `dir`; # в скалярном контексте
```

```
my @file_list = qx(dir); # в списочном контексте
```

В зависимости от того, в каком контексте - скалярном или списочном - употребляется операция выполнения программы, результат работы внешней программы рассматривается как одна строка или как список строк.

Выполнить внешнюю программу можно также с помощью функции system, которая организует синхронный запуск программы и возвращает код завершения. Код завершения 0 означает, что команда была выполнена успешно. Приведем пример ее использования в программе, архивирующей файлы с суффиксом .pl в текущем каталоге:

```
use English; # использовать длинные имена спец. переменных
```

```
# в ОС MS Windows архивируем файлы с помощью pkzip
```

```

if ($OSNAME =~ m/win/i) {

system "pkzip", "-a", "pearls.zip", "*.pl";

# в ОС GNU/Linux архивируем файлы с помощью tar и gzip

} elsif ($OSNAME =~ m/linux/i) {

system "tar -cv *.pl | gzip > pearls.tar.gz";

}

```

При вызове с одним строковым аргументом функция `system()` использует для запуска командный интерпретатор операционной системы так же, как функции `exec()`, `open()` и операция `qx()`. При передаче ей нескольких аргументов она запускает внешнюю программу с помощью системного вызова (обращения к операционной системе). Чтобы обеспечить успешный поиск запускаемой программы, можно добавить каталог, где находится программа, в список путей поиска. Например, таким образом:

```

{ # временно помещаем каталог с программой в пути поиска

local $ENV{"PATH"} = $path_to_the_program; # каталог

system($program_to_execute); # вызов программы

} # значение $ENV{"PATH"} будет восстановлено

```

Выполнение внешних программ можно организовать с помощью функции `open`, если требуется обмениваться данными с этими программами, используя перенаправление потоков ввода-вывода. Для этого функции `open()` вместо имени файла передается командная строка с именем выполняемой программы и ее аргументами. Если нужно передать поток данных для обработки из Perl-программы в вызываемую программу, то перед командой указывается символ командного конвейера `'|'`. Как это делается, видно из очень простого примера, в котором случайным образом генерируются числовые пароли, а затем они направляются для сжатия в архив программой `gzip`:

```

# открываем выходной поток, направляем его внешней программе

open my $archive, "| gzip > passwords.gz";

for (my $i = 1; $i <= 12; $i++) { # генерируем пароли

printf $archive "%06.0f\n", rand 999999;

}

close $archive; # закрываем выходной поток

```

Когда нужно принять выходной поток внешней программы для обработки в Perl-программе, то символ конвейера команд `'|'` ставится в конце командной строки:

```

# открываем входной поток, полученный от внешней программы

open my $archive, "gzip -d < passwords.gz |";

while (my $line = <$archive>) { # читаем пароли из архива

print $line;

}

close $archive; # закрываем выходной поток

```

(Используемый в примерах архиватор `gzip` распространяется свободно, версии для самых разных ОС доступны на сайте .)

Иногда требуется организовать выполнение программы таким образом: вначале запускается загрузчик, который, в зависимости от условий, заданных в конфигурации программы, запускает вместо себя основную программу. Этот подход можно реализовать с

помощью функции `exec`, которая заменяет работающую программу на указанную. Так можно запускать не только Perl-программы. Этот прием можно проиллюстрировать таким примером:

```
print "Выполняется загрузчик: $0, PID:$$\n";

# заменить текущую программу на указанную

my $program = $ARGV[0]; # имя программы в 1-м аргументе

print "Запускается программа: $program\n";

exec 'perl', $program or die; # запуск программы

print "Это сообщение никогда не напечатается!\n";
```

При запуске этого примера с параметром `'proc_executed.pl'` будут выведены такие сообщения:

Выполняется загрузчик: `proc_exec.pl`, PID:652

Запускается программа: `proc_executed.pl`

Выполняется программа: `proc_executed.pl`, PID:1872

Для организации параллельного выполнения процессов в Perl используется функция `fork` ("разветвить"). В результате ее работы создается копия выполняющегося процесса, которая тоже запускается на выполнение. Для этого в операционных системах семейства Unix происходит обращение к системному вызову `fork`. В других операционных системах работа функции `fork()` организуется исполняющей системой Perl. Функция `fork()` в родительском процессе возвращает PID дочернего процесса, число 0 - в дочернем процессе и неопределенное значение при невозможности запустить параллельный процесс. Это значение проверяется в программе, чтобы организовать выполнение различных действий в процессе-предке и процессе-потомке. Как это делается, показано на следующем схематичном примере (где оба процесса в цикле выводят числа, но с разными задержками):

```
my $pid = fork(); # 'разветвить' текущий процесс

# fork вернет 0 в потомке и PID потомка в процессе-предке

die "fork не отработал: $!" unless defined $pid;

unless ($pid) { # процесс-потомок

    print "Начался потомок PID $$\n";

    for (1..3) {

        print "Потомок PID $$ работает $_\n";

        sleep 2; # 'заснуть' на 2 секунды

    }

    print "Закончился потомок PID $$\n";

    exit;

}

if ($pid) { # процесс-предок

    print "Начался предок PID $$\n";

    for (1..3) {

        print "Предок PID $$ работает $_\n";

        sleep 1; # 'заснуть' на 1 секунду

    }

}
```

```

}

# возможно, здесь нужно ждать завершения потомка:

# print "Предок PID $$ ждет завершения $pid\n";

# waitpid $pid, 0;

print "Закончился предок PID $$\n";

}

```

По сообщениям, выводимым при выполнении этого примера, видно, что родительский и порожденный процессы выполняются параллельно. Для того чтобы организовать в родительском процессе ожидание завершения дочернего процесса, применяется функция `waitpid()`, которой передается PID процесса-потомка (а также, возможно, дополнительные параметры). По выдаваемым сообщениям сравните два варианта выполнения приведенной выше программы - без ожидания завершения дочернего процесса и с ожиданием завершения процесса-потомка (для этого нужно раскомментировать вызов функции `waitpid()`):

Без ожидания потомка С ожиданием потомка по `waitpid()`

```

-----

Начался потомок PID -1024 Начался потомок PID -1908

Потомок PID -1024 работает 1 Потомок PID -1908 работает 1

Начался предок PID 1504 Начался предок PID 1876

Предок PID 1504 работает 1 Предок PID 1876 работает 1

Предок PID 1504 работает 2 Предок PID 1876 работает 2

Потомок PID -1024 работает 2 Потомок PID -1908 работает 2

Предок PID 1504 работает 3 Предок PID 1876 работает 3

Закончился предок PID 1504 Предок PID 1876 ждет завершения -1908

Потомок PID -1024 работает 3 Потомок PID -1908 работает 3

Закончился потомок PID -1024 Закончился потомок PID -1908

Закончился предок PID 1876

```

Выполнение всей программы заканчивается, когда заканчивается последний порожденный процесс. Ожидание окончания выполнения всех дочерних процессов можно организовать с помощью функции `wait()`, которая возвращает PID завершившегося подпроцесса и -1, если все процессы-потомки завершили работу.

В Perl есть несколько способов организации взаимодействия процессов при их параллельном выполнении. Один из них - создать программный канал (`pipe`), который представляет из себя два файловых манипулятора - приемник (`reader`) и передатчик (`writer`) - связанных таким образом, что записанные в передатчик данные могут быть прочитаны из приемника. Программный канал создается с помощью функции `pipe()`, которой передаются имена двух файловых манипуляторов: приемника и источника. Один из вариантов взаимодействия процессов через программный канал показан в следующем примере:

```

use IO::Handle; # подключаем стандартный модуль

pipe(READER, WRITER); # создаем программный канал

WRITER->autoflush(1); # включаем авто-очистку буфера

if ($pid = fork()) { # процесс-предок получает PID потомка

    close READER; # предок не будет читать из канала

```



```

print WRITER "Послано предком (PID $$):\n";

for (my $n = 1; $n <= 5; $n++) { # запись в передатчик

print WRITER "$n ";

}

close WRITER; # закрываем канал и

waitpid $pid, 0; # ждем завершения потомка

}

die "fork не отработал: $!" unless defined $pid;

if (!$pid) { # процесс-потомок получает 0

close WRITER; # предок не будет писать в канал

print "Потомок (PID $$) прочитал:\n";

while (my $line = ) { # чтение из приемника

print "$line";

}

close READER; # канал закрывается

exit; # потомок завершает работу

}

```

Во время выполнения этого примера в стандартный выходной поток будет выведено следующее:

Потомок (PID -2032) прочитал:

Послано предком (PID 372):

1 2 3 4 5

Если нужно организовать передачу данных в обратном направлении, организуется канал, в котором передатчик будет в процессе-потомке, а приемник - в процессе-предке. Так как с помощью программного канала можно передавать данные только в одном направлении, то при необходимости двустороннего обмена данными между процессами создаются два программных канала на передачу в обоих направлениях.

Кроме программных каналов, процессы могут обмениваться информацией и другими способами: через именованные каналы (FIFO) и разделяемые области памяти, если они поддерживаются операционной системой, с помощью сокетов (что будет рассмотрено в следующей лекции) и при помощи сигналов.

В операционных системах имеется механизм, который может доставлять процессу уведомление о наступлении какого-либо события. Этот механизм основан на так называемых сигналах. Работа с ними происходит следующим образом. В программе может быть определен обработчик того или иного сигнала, который автоматически вызывается, когда ОС доставляет сигнал процессу. Сигналы могут отправляться операционной системой, или один процесс может с помощью ОС послать сигнал другому. Процесс, получивший сигнал, сам решает, каким образом реагировать на него, - например, он может проигнорировать сигнал. Перечень сигналов, получение которых можно попытаться обработать, находится в специальном хэше с именем %SIG. Поэтому допустимые идентификаторы сигналов можно вывести функцией keys(%SIG). Общеизвестный пример - сигнал прерывания выполнения программы INT, который посылает программе операционная система по нажатию на консоли сочетания клавиш Ctrl+C. Как устанавливать обработчик конкретного сигнала, показано на примере обработки сигнала INT:

```

# устанавливаем обработчик сигнала INT

$SIG{INT} = \&sig_handler; # ссылка на подпрограмму

```

```
# начало основной программы

print "Работаю в поте лица...\n" while (1); # бесконечный цикл

sub sig_handler { # подпрограмма-обработчик сигнала

$SIG{INT} = \&sig_handler; # переустанавливаем обработчик

print "Получен сигнал INT по нажатию Ctrl+C\n";

print "Заканчиваю работу!\n";

exit; # завершение выполнения программы

}
```

Выполнение примера сопровождается выводом сообщений, подтверждающих обработку поступившего сигнала:

Работаю в поте лица...

Получен сигнал INT по нажатию Ctrl+C

Заканчиваю работу!

Примером реальной программы, выполняющейся в бесконечном цикле, может служить любой сервер, ожидающий запросов от клиентских программ и перечитывающий свой конфигурационный файл после получения определенного сигнала (обычно HUP или USR1). Если необходимо временно игнорировать какой-то сигнал, то соответствующему элементу хэша %SIG присваивается строка 'IGNORE'. Восстановить стандартную обработку сигнала можно, присвоив соответствующему элементу %SIG строку 'DEFAULT'.

Процесс может посылать сигналы самому себе, например, для отслеживания окончания запланированного периода времени (для обработки тайм-аута). В приведенном примере длительная операция прерывается по истечении указанного промежутка времени:

```
# устанавливаем обработчик сигнала ALRM (будильник)

$SIG{ALRM} = sub { die "Timeout"; }; # анонимная подпрограмма

$timeout = 3600; # определяем величину тайм-аута (сек.)

eval { # блок обработки возможной ошибки

alarm($timeout); # устанавливаем время отправки сигнала

# некая длительная операция:

print "Работаю в поте лица...\n" while (1);

alarm(0); # нормальное завершение операции

};

# в специальной переменной $@ - сообщение об ошибке

if ($@ =~ /Timeout/) { # проверяем причину ошибки

print "Аварийный выход по истечении времени!";

}
```

Отправка сигнала одним процессом другому также используется в качестве средства взаимодействия процессов. Сигнал отправляется процессу с помощью функции kill(), которой передаются два аргумента: номер сигнала и PID процесса. Схему реагирования порожденного процесса на сигналы, получаемые от процесса-предка, можно увидеть на следующем учебном примере:

```
my $parent = $$; # PID родительского процесса
```

```

my $pid = fork(); # 'разветвить' текущий процесс

# fork вернет PID потомка в процессе-предке и 0 в потомке

die "fork не отработал: $!" unless defined $pid;

if ($pid) { # ----- родительский процесс -----

    print "Начался предок PID $$\n";

    for (1..3) {

        print "Предок PID $$ работает $_\n";

        print "Предок PID $$ отправил сигнал\n";

        kill HUP, $pid;

        sleep 2; # 'заснуть' на 2 секунды

    }

    print "Закончился предок (PID $$)\n";

}

unless ($pid) { # ----- дочерний процесс -----

    my $counter = 0; # счетчик полученных сигналов

    $SIG{HUP} = sub { ### обработчик сигнала ###

        $counter++;

        print "\tПотомок получил $counter-й сигнал!\n";

    }; ### конец обработчика сигнала ###

    print "\tНачался потомок PID $$ предка $parent\n";

    for (1..7) {

        print "\tПотомок PID $$ работает $_\n";

        sleep 1; # 'заснуть' на 1 секунду

    }

    print "\tЗакончился потомок PID $$\n";

}

```

Поведение этих процессов во время выполнения программы можно проследить по выводимым ими сообщениям:

Начался потомок PID -800 предка 696

Потомок PID -800 работает 1

Начался предок PID 696

Предок PID 696 работает 1

Предок PID 696 отправил сигнал

Потомок получил 1-й сигнал!

Потомок PID -800 работает 2

Предок PID 696 работает 2

Предок PID 696 отправил сигнал

Потомок PID -800 работает 3

Потомок получил 2-й сигнал!

Потомок PID -800 работает 4

Предок PID 696 работает 3

Предок PID 696 отправил сигнал

Потомок PID -800 работает 5

Потомок получил 3-й сигнал!

Потомок PID -800 работает 6

Закончился предок (PID 696)

Потомок PID -800 работает 7

Закончился потомок PID -800

Сигналы нельзя считать слишком надежным и информативным средством обмена информацией: для передачи данных лучше использовать другие способы. Зато можно проверить состояние дочернего процесса, отправив ему особый нулевой сигнал функцией `kill(0, $pid)`. Этот вызов не влияет на выполнение процесса-потомка, но возвращает истину (1), если процесс "жив", и ложь (0), если он завершился или ему нельзя посылать сигналы. Одинаковая реакция на нулевой сигнал гарантируется на различных платформах. Кроме того, можно прекратить выполнение дочернего процесса, отправив ему сигнал KILL вызовом `kill(KILL, $pid)`.

В последних версиях Perl появилась еще одна модель многозадачности - легковесные процессы (light-weight processes), называемые также потоками управления или нитями. (По-английски фраза "Perl threads" звучит как каламбур и может быть переведена как "нити жемчуга" или "жемчужные ожерелья"). Нити отличаются от полновесных процессов с независимыми ресурсами тем, что выполняются в рамках одного процесса в единой области памяти. Поэтому создание нити происходит быстрее запуска отдельного процесса и требует меньше ресурсов операционной системы. Выполнение нитей в одной области памяти позволяет эффективно организовать совместный доступ к разделяемым данным. Кроме того, программист получает более полный контроль над параллельно выполняющимися потоками управления. Принципиальное различие между полновесными процессами, созданными операционной системой, и многопоточными нитями показано на рис. 16.1.

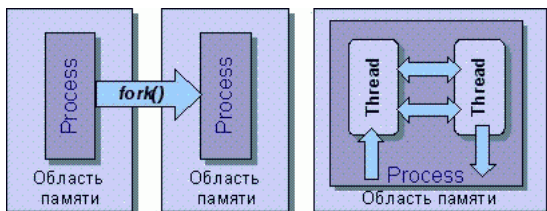


Рис. 16.1. Полновесные процессы и нити (потоки управления)

Существует несколько моделей многопоточной обработки, например DEC, Java, POSIX, Win32. Perl предлагает свою модель многопоточного программирования, отличающуюся от перечисленных и имеющую свои достоинства и недостатки. Появление в Perl кросс-платформенных средств работы с легковесными процессами стало несомненным достижением, которое заставило по-новому взглянуть на программирование параллельных процессов. Применение легковесных процессов позволяет разрабатывать эффективные приложения, одинаково выполняющиеся на разных платформах.

Работать с легковесными процессами просто. Подключив средства работы с нитями прагмой `use threads`, можно создать нить с помощью метода `threads->new` (синоним: `threads->create`). Этому методу передается ссылка на именованную или анонимную

подпрограмму, которая запускается на выполнение в виде параллельного потока управления. Результатом создания нити станет ссылка на объект типа `threads`, который будет использоваться для управления потоком. Создание нити выглядит так:

```
use threads; # подключить многопоточные средства

my $thread = threads->new(\&pearl_thread); # запустить нить

sub pearl_thread { # эта подпрограмма

print "Это нить.\n"; # будет выполняться как нить

} #
```

Итак, в определенной точке программы нить начала выполняться параллельно действиям в основной программе. Куда же должен произойти возврат, когда нить завершится? Это задается в основной программе с помощью метода `join`, который приостанавливает работу программы до завершения выполнения нити и возвращает результат, вычисленный нитью:

```
@result = $thread->join;
```

Действие, выполняемое методом `join`, называется "присоединение нити" или "объединение потоков". Как это происходит, показано на рис. 16.2.

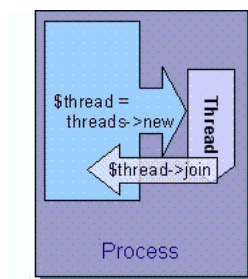


Рис. 16.2. Присоединение нити с помощью `join()`

Каждой нити присваивается числовой идентификатор (Thread Identifier, TID), который можно получить с помощью метода `tid`. Создание нескольких нитей, объединение потоков и возврат значений из параллельно выполняющихся подпрограмм можно показать на таком примере:

```
use threads; # подключить многопоточные средства

my @thread = (); # массив объектов типа threads

for (my $i = 0; $i <= 2; $i++) { # создаем 3 нити

$thread[$i] = threads->new(\&pearl_thread, $i);

print "Создана $i-я нить. TID=", $thread[$i]->tid, "\n";

}

for (my $i = 2; $i >= 0; $i--) { # присоединяем нити

print "$i-я нить вернула ", $thread[$i]->join, "\n";

}

sub pearl_thread ($) { # нить получает

my $number = shift; # число, генерирует

my $random = int(rand(7)) + 1; # случайное значение,

print "\t$number-я нить ждет $random сек.\n";

sleep $random; # и, подождяв немного,
```

```
return $random; # возвращает его

}
```

Сообщения, выводимые при выполнении этой программы, подтверждают независимое выполнение нитей и основной программы:

Создана 0-я нить. TID=1

Создана 1-я нить. TID=2

1-я нить ждет 7 сек.

0-я нить ждет 1 сек.

Создана 2-я нить. TID=3

2-я нить ждет 3 сек.

2-я нить вернула 3

1-я нить вернула 7

0-я нить вернула 1

Параллельно выполняющийся поток можно "отсоединить", игнорируя его значение: это делается методом `$thread->detach`, после выполнения которого присоединить нить будет невозможно.

Нити, выполняющиеся параллельно с основной программой, могут иметь доступ к общим переменным: скалярам, массивам и хэсам. Это делается с помощью явного указания для разделяемой переменной атрибута `shared`. Помеченная этим атрибутом переменная будет доступна для чтения и изменения в параллельном потоке. Для остальных переменных при отсоединении нити создаются локальные для каждого потока копии. Это демонстрируется таким примитивным примером:

```
use threads; # подключить многопоточные средства

use threads::shared; # и средства разделения данных

my $public : shared = 0; # разделяемая переменная

my $private = 0; # неразделяемая переменная

threads->new(sub { # нить из анонимной подпрограммы

    $public++; $private++; # изменяем значения

    print "$public $private\n"; # будет выведено: 1 1

})->join; # ожидаем результатов выполнения:

print "$public ", # 1 ($public изменена в нити)

"$private\n"; # 0 (в нити изменена копия $private)
```

Чтобы предотвратить в одной нити изменение другими нитями значения разделяемой переменной, эту переменную нужно заблокировать при помощи функции `lock()`. Разблокирование переменной происходит не с помощью функции, а при выходе из блока, в котором она была заблокирована. Это делается таким образом:

```
{ # блок для работы с разделяемой переменной

    lock $variable; # заблокировать переменную

    $variable = $new_value; # и изменить ее значение

} # здесь $variable автоматически разблокируется
```

Нити могут обмениваться между собой данными. Например, с помощью стандартного модуля `Thread::Queue` организуется очередь для синхронизированной передачи данных из одной нити в другую. Пользоваться такой очередью гораздо проще, чем рассмотренными ранее программными каналами. Небольшой пример показывает, как помещать скалярные величины в очередь методом `enqueue()` и извлекать из нее методом `dequeue()`. Метод `pending()` возвращает число оставшихся в очереди элементов, поэтому может использоваться для окончания цикла чтения из очереди:

```
use threads; # подключить средства

use Thread::Queue; # и модуль работы с очередью

my $data_queue = Thread::Queue->new; # создаем очередь

my $thread = threads->new(\&reader); # и нить

# помещаем в очередь скалярные данные:

$data_queue->enqueue(1987); # число

$data_queue->enqueue('год'); # строку

$data_queue->enqueue('рождения', 'Perl'); # список

$thread->join; # ожидаем окончания нити

exit; # перед завершением программы

sub reader { # извлекаем данные из очереди,

while ($data_queue->pending) { # пока она не пуста

my $data_element = $data_queue->dequeue;

print "'$data_element' извлечен из очереди\n";

}

}
```

Автоматическая синхронизация доступа к очереди гарантирует очередность записи в очередь и чтение из нее, что видно из выполнения этого примера:

```
'1987' извлечен из очереди

'год' извлечен из очереди

'рождения' извлечен из очереди

'Perl' извлечен из очереди
```

Конечно, имеющиеся в Perl средства работы с легковесными процессами не ограничиваются перечисленными выше. Стандартные модули предоставляют и другие возможности эффективно организовать различные алгоритмы многопоточной обработки, не говоря уже о дополнительных модулях, имеющихся в архивах CPAN.

Существует много ситуаций, когда применение многозадачности не только оправданно, но и является единственно правильным решением задачи. Поэтому знание средств управления процессами дает вам новую точку зрения на решаемую проблему и расширяет ваш арсенал программных инструментов. В 6-й версии языка Perl средства распределенного программирования будут улучшены и дополнены, появятся сопрограммы (co-routines) и другие интересные возможности.

## Лекция 17. Работа в IP-сетях

В этой лекции рассказывается об имеющихся в Perl возможностях обработки данных в IP-сетях, начиная с низкоуровневых средств и заканчивая классами для работы с основными сетевыми протоколами.

Цель лекции: узнать основные механизмы передачи данных по сети и научиться применять их для обработки данных в своих программах, используя стандартные и дополнительные модули Perl.

Сегодня большинство пользователей компьютеров воспринимают возможность обмена данными по компьютерной сети как нечто само собой разумеющееся. Это может быть небольшая офисная сеть из нескольких станций, корпоративная сеть, объединяющая многие сотни компьютеров, или подключение к глобальной сети Интернет. Легкость и удобство работы с многочисленными сетевыми сервисами стала возможной благодаря длительным усилиям многих выдающихся программистов, самых разных компаний и организаций из многих стран мира по созданию существующей инфраструктуры сетевых технологий. Эта инфраструктура основана на огромном числе стандартов, позволяющих согласованно использовать линии связи, аппаратуру передачи данных, компьютеры, операционные системы и прикладные программы для обмена информацией по сети.

Подавляющее большинство широко используемых сетей работает на основе протокола передачи данных IP (Internet Protocol), обеспечивающего надежное перемещение информации между компьютерами в разных сетях. Протокол - это система правил для согласованного взаимодействия при обмене информацией. При сетевом взаимодействии используется целый набор протоколов, обычно называемый стеком протоколов, который подразделяется на несколько уровней. На каждом из уровней выполняются определенные действия и преобразования данных. Протокол IP отвечает за сетевой уровень доставки информации, разделенной на специальные блоки данных, которые называются пакетами (packet).

Для идентификации объединенных в сети компьютеров или других сетевых устройств, обобщенно называемых хостами (host), используются последовательности из четырех чисел - IP-адреса: например, 192.168.82.83 или 172.16.2.73. Назначенный IP-адрес служит уникальным идентификатором хоста в конкретной сети. Кроме того, каждый хост, настроенный на работу с протоколом IP и даже не подключенный к сети, имеет собственный специальный адрес 127.0.0.1 - что-то вроде местоимения "я" на языке сетевых коммуникаций. Хост может иметь доменное имя, соответствующее его IP-адресу, например, имя хоста www.perl.com соответствует адресу 208.201.239.36. Собственному адресу 127.0.0.1 соответствует специальное имя localhost. Поскольку на каждом хосте может выполняться несколько сетевых программ, то для распределения между ними получаемых и отправляемых пакетов используются дополнительные числовые обозначения, так называемые номера портов. Поэтому программная точка отправления или доставки данных в IP-сетях определяется сочетанием адреса и порта, разделенных двоеточием. Многие номера портов по общепринятым соглашениям закреплены за определенными сетевыми службами. Например, обращение к web-серверу на текущей машине будет происходить по адресу и порту 127.0.0.1:80, а к почтовому серверу - по 127.0.0.1:25.

Для установления соединения между хостами и обмена данными в IP-сетях применяется механизм сокетов. Сокеты (socket) можно рассматривать как логические каналы двусторонней связи между сетевыми программами. Сокет определяется адресом хоста, номером порта и используемым протоколом обмена данными. Для организации пересылки данных между программами применяется один из двух протоколов транспортного уровня - UDP или TCP, выполняющихся поверх протокола IP. Протокол UDP (User Datagram Protocol) применяется для обмена независимыми блоками данных, называемыми дейтаграммами (datagram), без их гарантированной доставки адресату. Например, с использованием протокола UDP отправляются запросы управления устройствами или пересылается аудио- или видеотрансляция, когда потеря нескольких передаваемых пакетов не слишком существенна. Протокол TCP (Transmission Control Protocol) применяется для передачи по сети потока данных. При этом контролируется гарантированная доставка упорядоченной последовательности пакетов адресату. При помощи протокола TCP, например, отправляется электронная почта, передаются файлы и доставляются web-страницы.

Даже если в большинстве случаев при сетевом программировании на Perl используются более высокоуровневые средства, полезно хотя бы очень бегло познакомиться с принципами обмена данными через сокеты. Особенностью Perl, отражающей его сетевую направленность, стало то, что многие примитивные сетевые операции встроены в ядро языка, например: socket, socketpair, getsockname, getpeername, setsockopt, bind, listen, accept, send, recv, shutdown. Но гораздо удобнее и надежнее пользоваться стандартными модулями, реализующими средства работы с сокетами. В стандартном модуле Socket определены вспомогательные функции для работы с сокетами. Например, функция inet\_ntoa() преобразует в строку двоичное представление IP-адреса, которое возвращает встроенная функция gethostbyname. А функция inet\_aton() преобразует строковое представление адреса в двоичный вид, требуемый для встроенной функции gethostbyaddr, определяющей доменное имя хоста по IP-адресу. Работу этих функций можно показать на таком примере:

```
use Socket; # используем модуль работы с сокетами

my $host_name = 'www.perl.com'; # по имени хоста

my $address = gethostbyname($host_name); # узнаем адрес и

my $ip_address = inet_ntoa($address); # преобразуем его

print "$ip_address $host_name\n"; # в строку

# результат: 208.201.239.36 www.perl.com

$address = inet_aton($ip_address); # и обратно
```



```
my $host_name = gethostbyaddr($address,AF_INET);# узнаем имя

print "$ip_address $host_name\n"; # по адресу

# результат: 208.201.239.36 www.perl.com
```

Класс IO::Socket предоставляет объектный интерфейс для встроенных функций и помогает справиться со многими трудностями и избежать некоторых ошибок при программировании передачи данных через сокеты. Максимально упрощенный пример демонстрирует написание сервера для приема сообщений по протоколу TCP:

```
use IO::Socket; # используем класс работы с сокетами

my $server_port = 5555; # порт для обмена

my $server = IO::Socket::INET->new( # создаем сокет

LocalPort => $server_port, # на порту

Type => SOCK_STREAM, # для потокового обмена

Proto => 'tcp', # по протоколу TCP

Listen => 10, # с 10-ю соединениями

Reuse => 1) #

or die "Ошибка запуска TCP сервера на $server_port ($@)";

while (my $client = $server->accept()) { # создаем поток для

$client->autoflush(1); # клиента, очищаем буфер,

my $message = <$client>; # читаем сообщение из него

print $client "OK\n"; # посылаем ответ клиенту

close $client; # и закрываем поток

print STDERR $message; # выводим сообщение

last if $message =~ /STOP/i; # выходим из цикла, если

} # в сообщении есть STOP,

close $server; # и закрываем сокет
```

Сокеты могут использоваться не только для обмена данными по сети, но и для межпроцессного взаимодействия, когда сервер и клиент работают на одном и том же компьютере. Для доступа к приведенному серверу можно использовать, например, такую клиентскую программу:

```
use IO::Socket; # используем модуль работы с сокетами

my $server_host = '127.0.0.1'; # адрес сервера

my $server_port = 5555; # и порт на нем

my $socket = IO::Socket::INET->new( # создаем сокет

Type => SOCK_STREAM, # для потокового обмена

Proto => 'tcp', # по протоколу TCP

PeerAddr => $server_host, # с удаленным адресом
```

```

PeerPort => $server_port) # и портом

or die "Ошибка соединения с $remote_host:$remote_port ($@)";

# сообщение задается

my $message = $ARGV[0] || # параметром программы

"Проверка связи!"; # или умолчанием

print $socket "$message\n"; # отправляем его и

my $answer = <$socket>; # принимаем ответ

print "$answer"; # выводим ответ

close $socket; # и закрываем сокет

```

Из этого незатейливого примера можно сделать такой вывод: для согласованной работы клиент и сервер должны следовать установленным "правилам общения" во время сеанса обмена данными, так называемому протоколу прикладного уровня. В нашем случае правила сводятся к тому, что обмен идет по порту 5555, сервер ждет от клиента только одно сообщение, клиент ждет обязательного ответа от сервера, который завершает работу по получении сообщения, в котором содержится строка 'STOP'. Подробные соглашения описываются в конкретных протоколах сетевого обмена, например: HTTP (передача гипертекстовых документов), SMTP (отправка электронной почты), FTP (передача файлов). Описание подобных протоколов и других соглашений публикуются в виде предложений RFC (Request For Comment) - фактических международных стандартов, на которые ориентируются разработчики сетевого программного обеспечения.

Во Всемирной сети Интернет насчитывается огромное количество файловых серверов, где хранятся архивы программ, документация и другая информация. FTP (File Transfer Protocol) - это протокол, специально созданный для передачи файлов. Обмен файлами остается одной из постоянных задач сетевого программирования, которая легко решается средствами языка Perl. В поставке Perl имеется стандартный модуль Net::FTP, реализующий FTP-клиента, который позволяет весьма просто организовать обмен файлами с сервером по протоколу FTP - например, автоматизировать автоматическое обновление нужных файлов с сервера. Далее приводится программа, которая с помощью команд FTP загружает с сервера файл, если он имеет более позднее время изменения, чем его локальная копия:

```

use Net::FTP; # используем модуль работы с FTP

my $server = 'ftp.server.org'; # имя или адрес сервера

my $file = 'file.name'; # имя файла

my $ftp = Net::FTP->new($server) # соединяемся с сервером

or die "Ошибка соединения с $server:$@";

$ftp->login('ftp','ftp') # указываем имя и пароль

or die "Ошибка регистрации:", $ftp->message;

$ftp->cwd("/pub") # переходим в каталог

or die "Ошибка смены каталога:", $ftp->message;

my $time_ftp = $ftp->mdtm($file) # время изменения на сервере

or die;

my $time_old = (stat($file))[9]; # время создания копии

if ($time_ftp > $time_old) { # если файл на сервере новее,

$ftp->binary; # то в двоичном режиме

$ftp->get($file) # загружаем файл

```

```

or die "Ошибка загрузки: ", $ftp->message;

utime($time_ftp, $time_ftp, $file); # и меняем время файла

}

$ftp->quit; # заканчиваем сеанс связи

```

Поскольку в классе Net::FTP реализованы остальные команды протокола FTP, с его помощью можно разрабатывать гораздо более сложные программы файлового обмена. Кроме этого класса, в архивах CPAN можно найти много других модулей для обмена файлами с FTP-серверами.

Электронная почта (e-mail) была и остается одним из самых популярных сетевых сервисов. Электронная почта базируется на асинхронной доставке почтового сообщения (message) с одного почтового сервера на другой с помощью протокола SMTP (Simple Mail Transfer Protocol). Почтовые сообщения каждого пользователя хранятся на сервере в отдельном почтовом ящике (mail-box). Клиентская почтовая программа (Mail User Agent, MUA) забирает почту с сервера с помощью одной из версий протокола POP (Post-Office Protocol) или предоставляет пользователю непосредственный доступ к ящику на почтовом сервере с помощью протокола IMAP (Internet Mail Access Protocol).

Естественно, что в Perl имеется множество средств работы с электронной почтой. По электронной почте можно организовать автоматическое уведомление системных администраторов или пользователей о наступлении определенного события. С помощью сообщений электронной почты организуется регулярная автоматическая рассылка информации, например, счетов клиентам. При работе с почтой можно воспользоваться стандартными модулями - Net::SMTP для отправки сообщений и Net::POP3 для их получения. Приведем пример простой программы, отправляющей сообщение электронной почты:

```

use Net::SMTP; # используем класс для отправки e-mail

my $mail_server = 'shokhirev.com'; # почтовый сервер

my $to_user = 'mikhail@shokhirev.com'; # получатель

my $from_user = 'mshock@shadrinsk.net'; # отправитель

$smtp = Net::SMTP->new(Host=>$mail_server);# соединяюсь

$smtp->mail($from_user); # пишу

$smtp->to($to_user); # получателю

$smtp->data(); # письмо

$smtp->datasend("To: $to_user\n"); #

$smtp->datasend("Subject: Lectures on Perl 5\n");

$smtp->datasend("\n");

$smtp->datasend("Сообщаем о публикации на intuit.ru\n");

$smtp->datasend("курса лекций по Perl 5\n");

$smtp->dataend(); # заканчиваю

$smtp->quit; # отсоединяюсь

```

Если нужно в программе проверить почтовый ящик на сервере, то с помощью класса Net::POP3 не составит труда написать короткую программу, принимающую электронную почту. Например, такую:

```

use Net::POP3; # используем класс для получения e-mail

my $mail_server = 'shadrinsk.net'; # почтовый сервер

my $user = 'mshock'; # почтовый ящик

```

```

my $password = 'secret'; # пароль

$pop = Net::POP3->new($mail_server); # подключаюсь

if ($pop->login($user, $password) > 0) {# регистрируюсь

my $numbers = $pop->list; # получаю номера писем

foreach my $numbers (keys %$numbers) {# по номеру

my $message = $pop->get($numbers); # получаю письмо

print @$message; # печатаю его

$pop->delete($numbers); # удаляю с сервера

}

}

$pop->quit; # отсоединяюсь

```

На серверах CPAN есть множество программ для работы с электронной почтой, включая серверы. На Perl написана свободно распространяемая почтовая система с web-интерфейсом Open WebMail (openwebmail.org). На нем же написана и одна из самых известных и эффективных серверных систем фильтрации спама - SpamAssassin.

Системы мгновенного обмена сообщениями (instant messaging), иногда также называемые Интернет-пейджерами, получили огромное распространение: миллионы людей ежедневно общаются с помощью ICQ, Jabber, AOL Instant Messenger или Yahoo!Messenger. В архивах на сайте CPAN можно найти Perl-модули для работы со всеми этими системами. Хотя этот обмен сообщениями ориентирован на взаимодействие людей, его можно использовать для программного извещения пользователя о каком-либо событии. Для примера напишем программу, отправляющую сообщение с использованием открытого протокола мгновенного обмена сообщениями XMPP/Jabber (дополнительный модуль Net::Jabber нужно установить из архива CPAN). В примере после соединения с Jabber-сервером отправляется сообщение пользователю, а после получения от него ответа работа завершается:

```

use Net::Jabber; # подключаем класс работы с Jabber

use utf8; # в Jabber используется UTF-8

my $server = 'jabber.shadrinsk-city.ru'; # сервер

my $port = 5222; # порт

my $username = 'perl'; # отправитель

my $password = 'password'; # его пароль

my $resource = 'jud'; # ресурс

my $to_user = 'mshock@jabber.ru'; # получатель

my $client = new Net::Jabber::Client(); # создаем клиента

$client->SetCallbacks( # указываем обработчики событий:

onconnect => \&on_connect, # при подключении

onauth => \&on_auth, # при регистрации

message => \&on_message, # при получении сообщения

);

$client->Execute( # соединяемся с сервером

```

```

hostname=>$server, port=>$port,

username=>$username, password=>$password,

resource=>$resource, register=>1,

connectsleep=>0, connectattempts=>1,

);

# завершение программы произойдет в on_message

sub on_connect { # выполнится при подключении к серверу

print "Подключен к $server:$port\n";

}

sub on_auth { # выполнится при регистрации на сервере

print "Зарегистрирован как $username\n";

$client->MessageSend( # ОТПРАВЛЯЕМ СООБЩЕНИЕ

to=> $to_user. '/' . $resource,

subject=>'сообщение от Perl',

body=>'Привет, Jabber!'

);

}

sub on_message { # выполнится при получении сообщения

my $sid = shift; # извлекаем номер и текст

my $message = shift; # ответа и выводим его:

print "Тема:", $message->GetSubject(), "\n";

print "Сообщение:", $message->GetBody(), "\n";

$client->Disconnect(); # отключаемся от сервера

exit(0); # и завершаем работу

}

```

В этом примере демонстрируется прием программирования, распространенный при работе в многозадачной среде: главная программа организует бесконечный цикл обработки событий, для реагирования на которые вызываются обработчики событий. Конечно, помимо обмена сообщениями Jabber предоставляет целый набор средств для полноценного общения, а Perl дает возможность работать с ними.

Есть примеры использования Perl для разработки шлюзов между IP-сетями и беспроводными сетями. На Perl написан свободно распространяемый пейджинговый шлюз Sendpage (). Класс Net::SNPP занимается отправкой сообщений на пейджер по протоколу SNPP (Simple Network Paging Protocol), а модуль Net::SMS служит для работы со службой SMS-сообщений для клиентов GSM-телефонии. Теперь мало кого удивляет, что вскоре после появления очередной новой технологии передачи данных на CPAN выкладываются Perl-модули для работы с ней.

Крупную компьютерную сеть можно сравнить с живым организмом: сетевое оборудование и компьютеры - это ее органы, а линии связи - нервные волокна. Сеть ежесекундно меняет свое состояние, ее составные части могут "болеть" и "отмирать", сеть может "задышаться" от чрезмерного объема передаваемых данных (сетевого трафика). Для мониторинга состояния сети, ее обслуживания

и диагностики неисправностей применяются различные служебные программы. Некоторые из них написаны на языке Perl. В других случаях Perl применяется для "склеивания" разных программ в единую систему управления сетью. Если вспомнить, что Perl был создан системным администратором, то станет понятным, почему имеется так много модулей, так или иначе связанных с управлением сетями. В качестве простейшего примера можно привести стандартный класс Net::Ping, с помощью которого можно проверить работоспособность хоста по его IP-адресу. В ОС MS Windows это делается такой командой:

```
perl -MNet::Ping -e"print Net::Ping->new()->ping('10.0.0.1')"
```

В ней запускается компилятор perl, который подключает модуль (-M) Net::Ping и выполняет выражение (-e), заключенное в двойные кавычки. А в выражении выводится результат обращения к методу ping класса Net::Ping с IP-адресом в качестве аргумента. В результате выполнения будет выведен адрес хоста и две 1, если хост активен, или два 0, если он не ответил.

Многие программные средства управления компьютерными сетями основаны на протоколе SNMP (Simple Network Management Protocol), использующего для обмена данными протокол UDP. Такие программные средства построены по принципу периодического опроса так называемых агентов, которые отвечают на запросы управляющей программы и передают ей информацию, накопленную во время работы подключенного к сети устройства - компьютера, принтера, маршрутизатора и так далее. Категории собираемой информации (управляемые устройства и их характеристики) имеют унифицированные имена и числовые идентификаторы объектов (Object Identifier, OID), которые присваиваются производителями устройств в соответствии со стандартами описания "базы данных управляющей информации" MIB (Management Information Base). При определенных условиях по протоколу SNMP можно не только считывать по сети информацию с устройств, но и управлять этими устройствами, изменяя их характеристики. С помощью протокола SNMP можно также организовать управление программными комплексами, например, операционными системами и СУБД. На Perl написаны модули для работы с SNMP-агентами. Для иллюстрации сказанного приведем простой пример, в котором опрашивается агент, работающий на персональном компьютере, и у него запрашиваются две характеристики - описание системы и время ее работы:

```
use Net::SNMP; # используем класс для работы с SNMP

my ($session, $error) = Net::SNMP->session( # сеанс работы

-hostname => '192.168.82.83', # с хостом,

-community => 'public', # группой

-port => 161 # и портом

);

defined($session) or die ("Ошибка сеанса SNMP: $error");

# запрашиваем информацию о компьютере по

# коду (OID) и идентификатору объекта MIB:

info('1.3.6.1.2.1.1.0', 'sysDescr'); # описание системы

info('1.3.6.1.2.1.1.3.0', 'sysUpTime'); # время работы

$session->close(); # завершаем сеанс

sub info { # подпрограмма запроса информации

my ($OID, $caption) = @_; # параметры: код и имя объекта

my $response = $session->get_request($OID);

unless (defined($response)) { # если все нормально

print 'Ошибка запроса: ', $session->error();

} else { # выводим ответ:

printf "$caption/$OID:\n\t%s\n", $response->{$OID};

}

}
```

```
}
```

В результате выполнения этой программы в ОС MS Windows будет выведена следующая информация:

sysDescr/1.3.6.1.2.1.1.0:

Hardware: x86 Family 6 Model 8 Stepping 6 AT/AT COMPATIBLE - Software: Windows 2000 Version 5.1

(Build 2600 Uniprocessor Free)

sysUpTime/1.3.6.1.2.1.1.3.0:

1 hour, 05:14.11

Для отслеживания состояния сети имеется много готовых программных продуктов различной сложности. Сбором и накоплением информации о работе сети и ее визуализацией занимаются основанные на SNMP системы: Big Sister (bigsister.sourceforge.net), Cricket (cricket.sourceforge.net) и MRTG (), которые распространяются свободно и написаны на языке Perl.

С момента своего создания язык Perl применялся системными администраторами для сетевого программирования. И постепенно было разработано неимоверное количество модулей для работы с самыми разными сетевыми сервисами. Даже упомянув лишь некоторые из них, можно составить представление о многообразии высококачественных решений сетевых задач, воплощенных в "жемчужных россыпях" хранилища модулей CPAN. В стандартном классе Net::NNTP реализован клиент новостных групп (телеконференций), располагающихся на многочисленных news-серверах. Perl-модули помогут сгенерировать документы для мобильных клиентов в формате WML (Wireless Markup Language). Модуль CDDb предоставляет интерфейс к сетевым базам данных по музыкальным компакт-дискам (Compact Disc DataBase). Имеются модули для работы с известными поисковыми системами: AltaVista, Google, Yahoo и с Интернет-магазинами Amazon и eBay.

В репозитории CPAN можно найти модули для работы со всеми распространенными сетевыми протоколами: ARP (поиск физического адреса устройства по IP-адресу), DHCP (динамическое распределение IP-адресов), LDAP (доступ к каталогам типа Active Directory и NDS), NTP (запрос времени), RADIUS (авторизация пользователей), telnet и ssh (работа на удаленном терминале), VNC (сетевое управление компьютером) и многими другими. Причем Perl-модули могут использоваться для создания не только клиентских программ, но и серверов, которые можно встраивать в собственные приложения. И конечно же, есть огромное число модулей для работы с сервисами WWW, о которых пойдет речь в следующей лекции.

Рассмотренные в этой лекции средства сетевой коммуникации дают лишь общую поверхностную картину многообразного мира распределенного программирования. Высокоуровневая поддержка многих сетевых технологий реализована в Perl-модулях, которые помогут начинающим и опытным программистам решать задачи, связанные с распределенной обработкой данных.

## Лекция 18. Web-программирование

Эта лекция посвящена разработке на Perl программ для WWW, Всемирной Паутины ресурсов, связанных гиперссылками. Ведь именно с распространением World Wide Web язык Perl получил неимоверную популярность среди программистов, которые создали с его помощью множество популярных сайтов с динамическим содержанием.

Цель лекции: вкратце ознакомиться с основами web-программирования и богатыми средствами, которые имеются в Perl, для работы с ресурсами Всемирной Паутины. На примерах узнать приемы работы с некоторыми из них.

Всемирная Паутина (World Wide Web, WWW или просто Web) стала важнейшим технологическим достижением в области обработки информации. Она не только сделала доступ к ресурсам сети Интернет простым и удобным для пользователей, но стимулировала развитие многих информационных технологий, а также утвердила в практике программирования новые подходы к работе с информационными ресурсами. Самыми важными из них можно назвать следующие.

1 Благодаря универсальной адресации самых разных ресурсов с помощью унифицированных указателей ресурсов (Uniform Resource Locator, URL), доступ к информации выполняется единообразно, а средства доступа динамически настраиваются на расположение ресурса, протокол передачи данных и его формат. (Например, для указателя на ресурс потребуется инициировать запрос к серверу по протоколу HTTP на доставку указанной HTML-страницы. А необходимость получить ресурс по указателю должна привести к организации сеанса взаимодействия с сервером по протоколу FTP для загрузки требуемого файла.)

2 Гипертекстовые (а если быть точнее - "гипермедийные") документы позволяют с помощью гиперссылок (hyper-link) логически компоновать информационные ресурсы, причем разными способами и независимо от их физического расположения. То есть можно считать, что ресурсы располагаются не в файловой системе, пусть даже и сетевой, а в гиперссылочном информационном пространстве. (Например, гиперссылка на одно и то же изображение может размещаться в электронном учебнике, присутствовать

на странице тестового упражнения и включаться в научную презентацию. А во фреймах одного окна браузера могут отображаться страницы, загруженные по гиперссылкам с серверов, расположенных в разных странах.)

3 Тим Бернерс-Ли, который придумал и в 1991 году реализовал для научных целей WWW, основанную на гиперссылках с использованием URL, конечно же, достоин бесконечного восхищения. Но не менее плодотворной идеей оказался CGI (Common Gateway Interface) - интерфейс простого шлюза для обращения web-сервера к внешним программам. Взаимодействие посредством CGI упрощенно сводится к запуску сервером в виде отдельного процесса внешней программы, которой через стандартный входной поток передаются параметры, полученные в запросе от клиента. Результат своей работы программа передает через стандартный выходной поток серверу, который возвращает его в качестве ответа на запрос клиента. Эта традиционная для Unix схема обращения к программе-фильтру позволяет бесконечно расширять возможности web-сервера: динамически создавать HTML-страницы, генерировать диаграммы и графики, черпать информацию из баз данных, "на лету" конструировать документы в формате PDF и так далее.

4 Принципы, применяемые в WWW для организации информации, оказались настолько технологичными и имели такой большой успех у пользователей, что стали использоваться в локальных сетях (Intranet) и на отдельных компьютерах. А web-браузер превратился в универсальное клиентское приложение, применяемое для доступа к базам данных, различным справочным и информационным системам.

Поэтому web-программирование стало важным и чрезвычайно востребованной отраслью информационной промышленности. (Именно промышленности, если судить по инвестициям и доходам информационных компаний во всем мире.) Ну а Perl, как обычно, не просто предоставляет для этого все необходимые средства, но и предлагает их широкий выбор: ведь принцип TIMTOWTDI продолжает работать и здесь...

Прославившийся своими богатыми средствами обработки текстовых данных, Perl оказался легко применим для работы с гипертекстом. Язык разметки гипертекста HTML (Hyper-Text Markup Language) - это подмножество довольно старого универсального языка разметки SGML, использовавшегося для форматирования документов. HTML ориентирован на разметку гипертекстовых документов. Со времени своего создания он видоизменялся под влиянием корпоративных интересов нескольких компаний, но конкурентная борьба постепенно приводит к повышению роли стандартов. Стандартизацией HTML и других видов деятельности, касающейся WWW, занимается международный консорциум W3C, который возглавляет сэр Тимоти Джон Бернерс-Ли. Чтобы отдельно от содержимого HTML-документа описывать особенности его представления (шрифт, цвет, размер, расположение и так далее), был создан еще один стандарт - таблицы каскадных стилей CSS (Cascading Style Sheets). А сам язык HTML был переработан для обеспечения совместимости с форматом XML (о котором речь пойдет далее), и новая версия была оформлена в виде стандарта XHTML.

На Perl написано множество модулей, ориентированных на работу с гипертекстовыми документами в форматах HTML и XHTML. Большинство из них связано с динамической генерацией web-страниц и работе с HTML-формами. Поисковая машина на сайте CPAN находит более тысячи модулей, в названии которых встречается "HTML". Упомянем лишь некоторые из них, доступные для загрузки из архива CPAN. Простые средства создания разметки HTML предоставляет класс HTML::Base. Модуль HTML::Parser, напротив, представляет из себя средство синтаксического разбора HTML-документа на составляющие объекты. Кстати, для преобразования "старой доброй документации" из формата POD в HTML-документ можно пользоваться утилитой pod2html. Она, естественно, написана на Perl и входит в стандартный дистрибутив. Например, в результате выполнения вот такой команды:

```
pod2html --infile=CGI.pm --outfile=CGI.html
```

будет сгенерирован гипертекстовый документ из 54 страниц, описывающий стандартный модуль для разработчиков CGI-программ на Perl. По солидному объему этого руководства можно судить о том, что многие трудности web-программирования уже решены усилиями Perl-сообщества. Примеры программногo формирования документов HTML будут рассмотрены далее в этой лекции при обсуждении средств разработки CGI-программ.

Узлы Всемирной Паутины - это разбросанные по всему миру миллионы web-серверов. Самым популярным и распространенным в сети Интернет остается web-сервер Apache (), распространяемый свободно с открытыми исходными текстами. Хорошо спроектированная архитектура сервера позволяет подключать к нему модули для расширения функциональности сервера. Одним из популярнейших модулей расширения стал mod\_perl, который позволяет интегрировать интерпретатор perl с сервером Apache. Это позволяет не только кардинально увеличить скорость работы CGI-программ, но и разрабатывать на Perl собственные модули, получая полный контроль за выполнением клиентских запросов.

Web-сервер общается с клиентскими программами по протоколу передачи гипертекста HTTP (Hypertext Transfer Protocol). Поскольку весь остальной материал этой лекции связан с передачей данных по указанному протоколу, познакомимся с ним поближе. В соответствии с протоколом HTTP запрос состоит из трех частей, которые приведены в таблице 18.1.

Таблица 18.1. Структура HTTP-запроса

| Составные части | Описание | Примеры |
|-----------------|----------|---------|
|                 |          |         |



|                   |   |   |
|-------------------|---|---|
| Строка запроса    | содержит команду, называемую методом, например, GET для запроса ресурса или POST для отправки данных на сервер, и имя ресурса       | GET /index.html HTTP/1.1  |
|                   |   | POST /cgi-bin/guestbook.pl HTTP/1.0   |
| Заголовки запроса | содержат дополнительную информацию, например, данные о клиенте или указания о языке и кодировке ответа, которые предпочитает клиент | User-agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; ru-RU; rv:1.8.0.1) Gecko/20060130 SeaMon |
|                   |   | Accept-Language: "ru-ru,ru; q=0.8,en-us;q=0.5,en;q=0.3"                                       |
| Тело запроса      | в нем может содержаться передаваемая на сервер информация (например, данные из полей HTML-формы).                                   | nick=Ray&email=ray@conchilomania.ru&comments=Your%20photos%20and%20info%20about%20perl%20are% |

Сервер обрабатывает поступающие от клиентов запросы на расположенные на сервере ресурсы. Если затребован существующий файл, то он отправляется сервером клиенту. Если запрошено обращение к CGI-программе, то сервер запускает ее и отправляет клиенту результат ее выполнения. Ответ HTTP-сервера также состоит из трех частей, которые приведены в таблице 18.2.

Таблица 18.2. Структура HTTP-ответа

| Составные части  | Описание  | Примеры                            |
|------------------|---|------------------------------------|
| Строка ответа    | содержит цифровой код ответа и текстовое описание состояния запроса                     | HTTP/1.0 200 Document follows      |
|                  |   | HTTP/1.1 404 Not Found             |
|                  |   | HTTP/1.0 500 Internal Server Error |
| Заголовки ответа | содержат дополнительную информацию, например, данные о типе и длине посылаемого ресурса | Content-type: text/html            |
|                  |   | Content-length: 1025               |
| Тело ответа      | содержит передаваемые данные  | <html>...</html>                   |

Важной особенностью протокола HTTP является то, что он ориентирован на обработку независимых запросов, то есть в нем не предусмотрено сохранение состояния взаимодействия с клиентом. Поэтому организация сеансовой работы с web-сервером ложится

на программиста.

Автоматизировать рутинные действия при обмене данными с помощью HTTP и преодолеть трудности программного взаимодействия с HTTP-серверами помогают многочисленные готовые Perl-модули. Стандартная библиотека LWP (Library for WWW in Perl) содержит разнообразные и мощные средства для работы с ресурсами WWW. С ее помощью можно легко запрограммировать простые и решить весьма нетривиальные задачи. Например, запрос документа с web-сервера записывается всего одной строкой:

```
use LWP::Simple; # использовать упрощенный интерфейс к LWP
```

```
my $page = get 'http://www.perl.com/';
```

Не сложнее обратиться с запросом к поисковым системам. Для этого нужно в URL указать аргументы поиска. Формат строки запроса к конкретной поисковой машине можно посмотреть в строке браузера. Например, по такому URL можно искать книги по Perl в поисковой системе Google:

```
$url = 'http://www.google.ru/search?q=Perl+book';
```

А чтобы найти на сайте CPAN все модули, ориентированные на работу с HTML, нужно отправить такой запрос:

```
$url= 'http://search.cpan.org/search?query=HTML&mode=module';
```

Это примеры запросов, отправляемых методом GET, когда аргументы передаются непосредственно в строке, адресующей ресурс. При другом способе запрос отправляется методом POST, а данные запроса отправляются в теле запроса. Если требуется отправить из программы данные HTML-формы на HTTP-сервер методом POST, то это столь же просто делается с помощью LWP:

```
use LWP::UserAgent; # используем класс 'Клиент' из LWP
```

```
use HTTP::Request::Common qw(POST); # и метод POST
```

```
my $user_agent = LWP::UserAgent->new; # создаем клиента
```

```
# заполняем поля формы для отправки на нужный сайт
```

```
my $form = POST 'http://site.ru/cgi-bin/guestbook.pl',
```

```
[ nick => 'user', email => 'user@mail.ru',
```

```
comments => 'Спасибо за помощь!' ];
```

```
# передаем клиенту форму для отправки на сервер
```

```
my $response = $user_agent->request($form); # получаем ответ
```

```
print $response->as_string; # и выводим его в виде строки
```

Можно долго говорить о возможностях библиотеки LWP. С ее помощью можно: работать с новостными группами (news), обмениваться файлами по протоколу FTP, отправлять запросы к информационным системам Gopher, читать локальные файлы, отправлять электронную почту и создавать пользовательских агентов для автоматического исследования сайтов (web-роботов или "пауков"). Можно даже быстро набросать простой, но вполне работоспособный web-сервер. Для этого нужно воспользоваться классом HTTP::Daemon:

```
use HTTP::Daemon; # используем классы HTTP-сервера
```

```
my $server_root = '/tmp'; # каталог для файлов сервера
```

```
# создаем экземпляр WWW-сервера, слушающего порт 8080
```

```
my $httpd = new HTTP::Daemon(LocalPort => 8080); #
```

```
while (my $connection = $httpd->accept) { # ждем соединения
```

```
# получаем запросы на соединении
```

```
while (my $request = $connection->get_request) {
```

```

if ($request->method eq 'GET') { # выполняем GET

$connection->send_file_response( # отправляем файл

$server_root . $request->url->path); # из каталога

}

}

$connection->close; # закрываем соединение

undef($connection); # удаляем объект

} # и все повторяется сначала...

```

А теперь пора перейти к созданию программ, выполняющихся на web-сервере и взаимодействующих с ним через интерфейс CGI.

Упомянутый уже интерфейс программирования CGI поддерживается всеми web-серверами. Сегодня CGI-программы разрабатываются не на чистом Perl, а с использованием различных вспомогательных модулей и библиотек. Возможно, самой популярной из них по праву считается стандартная библиотека CGI. С ее помощью можно писать CGI-программы проще, быстрее и надежнее. Разработка программ CGI на Perl описывается во многих специализированных книгах.

В качестве примера напомним простенькую "гостевую книгу" - программу, которая с помощью функций библиотеки CGI (start\_html, textfield и т.д.) выводит HTML-страницу с формой для отправки на сайт отзывов посетителей. После отправки данных формы на сервер вызывается эта же CGI-программа, которая с помощью функции ragat() проверяет, получены ли данные формы, и выводит присланный комментарий. Вот текст программы:

```

#!/C:/usr/local/apache/Perl/bin/perl.exe

# в первой строке CGI-программы указан путь к perl

use CGI qw/:standard/; # применяем стандартные средства CGI

print # выводим в выходной поток

header(-charset=>'windows-1251'), # в кодировке CP1251:

start_html('Гостевая книга'), # шапку страницы,

h3('Здесь Вы можете оставить свой отзыв'), # заголовок,

start_form, # форму, в ней

"Имя: ", # надпись,

textfield(-name=>'nick', size=>8), p, # поле ввода,

"Э-почта: ", # надпись,

textfield(-name=>'email', size=>32), p, # поле ввода,

"Комментарий: ", p, # надпись,

textarea(-name=>'comments', # область ввода

-rows=>5, -columns=>50), p, # из 5 строк на 50 колонок,

submit('Отправить'), # кнопку,

end_form, # конец формы

hr, "\n"; # и горизонтальную черту

```

```
# далее проверяем, были ли присланы данные формы

if (param) { # если присланы данные - параметры формы

print # выводим:

a({href=>"mailto:".param('email')}, # ссылку на E-mail

param('nick')), # и имя, а также

" пишет: ", p, param('comments'), p, # комментарий

hr,"\n"; # и горизонтальную черту

}

print end_html; # оформляем конец страницы
```

При первом выполнении эта программа выводит пустую HTML-форму, а после того как форма заполнена и данные формы отправлены на сервер, на странице после формы выводится последний полученный комментарий. В результате будет сгенерирована web-страница, приведенная на рис. 18.1.

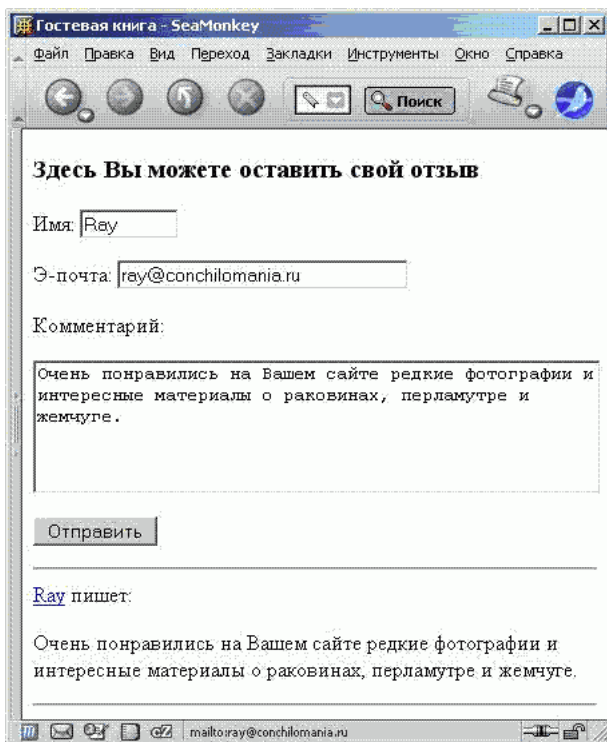


Рис. 18.1.Результат выполнения CGI-программы

Отлично протестированные подпрограммы стандартной библиотеки CGI выполняют все действия по созданию правильно оформленных web-страниц. Они скрывают от программиста трудности и тонкости при преобразовании параметров и обработке данных форм. Web-страницы можно формировать программно без использования разметки на языке HTML. Библиотека CGI также имеет объектно-ориентированный интерфейс со всеми необходимыми классами и методами для работы с объектами HTTP-запросов в CGI-программах.

Дальнейшим развитием CGI стали серверные технологии, в которых в шаблон HTML-документа включаются исполняемые фрагменты, написанные на встроенном языке программирования - C#, Java, PHP, Perl, Visual Basic или специальных языках шаблонов. На Perl написано немало систем для работы с шаблонами (templating system): от модулей, использующих несложную подстановку, до изощренных платформ программирования (application framework) для web-сервера. Perl в подобных системах применяется для обработки шаблонов (например, HTML::Template, Text::Template, Template Toolkit) и динамической генерации на основе шаблонов. Во многих системах (например, Apache::ASP, AxKit, Embperl, Mason, Apache::XPP) Perl применяется как встроенный язык, используемый для программирования действий в шаблонах. Все эти системы можно загрузить с сайта CPAN и установить обычным образом. (Подробнее об установке модулей речь шла в лекции 13.) Для работы с этими системами под ОС MS Windows проще всего загрузить с сайта perl.apache.org дистрибутив Perl, в состав которого входит сервер Apache с mod\_perl и многие из перечисленных библиотек. Каждая из систем реализует оригинальный подход и обладает интересными возможностями, но мы остановимся подробнее на той из них, которая реализует тот же подход, что и в других распространенных системах программирования на основе шаблонов: ASP, JSP и PHP.

Система разработки web-сайтов Apache::ASP предлагает кросс-платформенные средства, аналогичные используемым в системе программирования ActiveState PerlScript для web-сервера Microsoft IIS. В этом подходе сочетаются естественное представление HTML-документа и возможность использовать богатые возможности языка программирования. В шаблоне HTML-страницы между тегами <% и %> располагаются фрагменты программы на языке Perl, которые выполняются при обработке запроса на страницу. Результат выполнения этих фрагментов включается в результирующую страницу, которая отсылается клиенту. Если переписать пример с гостевой книгой, используя классы из состава Apache::ASP, то он будет выглядеть так:

```
content="text/html; charset=windows-1251">
```

Здесь Вы можете оставить свой отзыв

Имя:

```
value="<%= $Request->Form('nick') %>"/>
```

Э-почта:

```
value="<%= $Request->Form('email') %>"/>
```

Комментарий:

</p>  
<p class="paragraph">  
<%= \$Request->Form('comments') %>  
<p class="paragraph">

Отправить

```
<% if($Request->Form('nick')) { %>
```

```
<%= $Request->Form('nick') %>
```

пишет:

```
<%= $Request->Form('comments') %>
```

```
<% } %>
```

Обращение к значениям полей формы происходит с помощью метода `Form` предопределенного объекта `$Request`, хранящего информацию HTTP-запроса. Система `Apache::ASP` предоставляет программисту полный набор средств для динамического создания страниц, включая средства работы с клиентскими сеансами. Чтобы продемонстрировать возможности встроенного в шаблоны языка Perl, напомним шаблон ASP, реализующий web-интерфейс к базе данных. Perl в нем используется для извлечения из базы данных информации о моллюсках, производящих жемчужины, а также для формирования в цикле строк таблицы на основании результатов запроса. Фрагменты программы на Perl, встроенные в текст шаблона, выделены жирным шрифтом:

```
<% # начало встроенного Perl
```

```
use DBI; # используем DBI
```

```
my $table = "mollusc"; # подключаемся к БД
```

```
my $dbh = # через драйвер DBD::SQLite
```

```
DBI->connect("dbi:SQLite:dbname=$table","","") or die;
```

```
my $sth = # готовим выборку строк таблицы
```

```
$dbh->prepare("SELECT id,name,latin,area FROM $table")
```

```
or die $dbh->errstr;
```

```
$sth->execute() or die $sth->errstr(); # и выполняем запрос
```


```
%>
```

```
content="text/html; charset=windows-1251">
```

## Коллекция раковин

```
<% while (my $row = $sth->fetchrow_hashref) { %>
```

```
<% } %>
```

|  |  |
|--|--|
|  | <pre>&lt;%= \$row-&gt;{name}%&gt;</pre> <pre>&lt;%= \$row-&gt;{latin}%&gt;</pre> |
|  | <pre>&lt;%= \$row-&gt;{area}%&gt;</pre>  |

```
<% $dbh->disconnect; %>
```

HTML-страница, сформированная в результате выполнения программы, приведена на рис. 18.2. При необходимости несложно расширить функциональность этой программы, например, добавить поиск по любой из колонок таблицы.

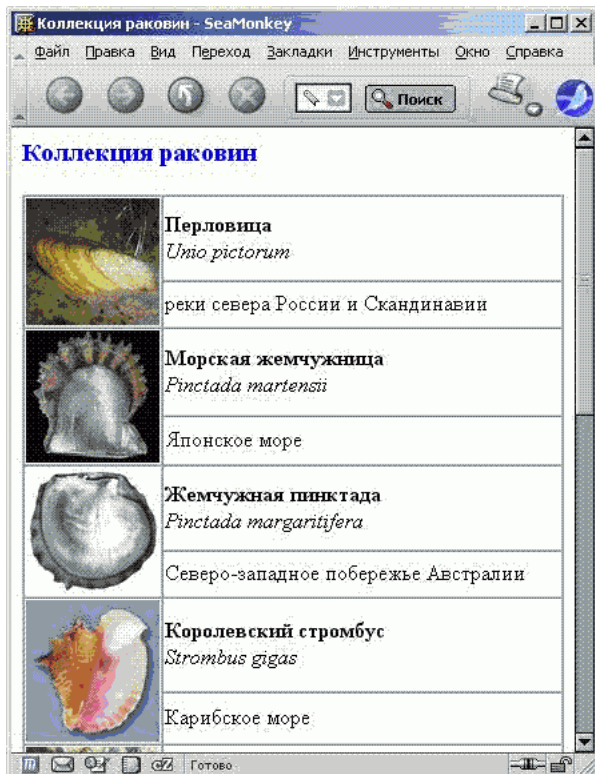


Рис. 18.2. Web-интерфейс к базе данных

Дальнейшая унификация ресурсов сети Интернет базируется на применении XML - расширяемого языка разметки (eXtensible Markup Language), стандартизованного консорциумом W3C. XML описывает правила создания прикладных языков разметки, называемых XML-приложениями (XML application). К настоящему времени созданы сотни прикладных языков на основе синтаксиса XML. Среди наиболее распространенных языков можно упомянуть CML (описание химических данных), GML (описание географических данных), Jabber (обмен сообщениями), MathML (описание математических формул), RDF (описание информационных ресурсов), SMIL (описание мультимедийных презентаций), RSS (аннотации содержимого сайтов), SVG (масштабируемая векторная графика), WDDX (обмен данными), WebDAV (web-папки), XML/EDI (обмен бизнес-данными), XML-RPC (удаленный вызов процедур), XUL (описание пользовательского интерфейса). Использование единой грамматики XML для прикладных языков разметки позволяет обрабатывать их унифицированными программными средствами. Языки разметки на основе XML создаются преимущественно для описания данных в различных областях знаний. В формате XML данные представлены в текстовом виде, чаще всего с использованием кодировки UTF-8, поэтому они без труда пересылаются по сети и обрабатываются программами на всех компьютерных платформах. Специальный расширяемый язык стилей XSL (eXtensible Style Language) создан для унифицированного преобразования XML в другие форматы, например, для визуального представления XML-данных в виде документов в формате HTML или RTF.

В Perl имеется богатый набор средств для работы с разными языками разметки на основе XML: это и универсальные инструменты, и специализированные модули для конкретных XML-приложений. Использованию XML-технологий в программировании на языке Perl посвящена книга [19] и ее перевод [45], а в книге [30] рассматривается работа с графикой в формате SVG. Для примера познакомимся с библиотекой SVG, предназначенной для программного создания масштабируемых векторных изображений в формате XML. Чтобы составить представление об этом формате и проиллюстрировать возможности этой библиотеки, напишем CGI-программу, динамически формирующую документ SVG. Она будет показывать на простой диаграмме распределение данных о посещаемости сайта по основным доменам:

```
use CGI qw/:standard/; # применим библиотеку CGI

use SVG; # и SVG

my $p = CGI->new; # создадим объект CGI и зададим

print $p->header(-type=>'image/svg+xml'); # тип документа

# создадим объект SVG размером 400 на 300 пикселей

my $svg= SVG->new(width=>400, height=>300);

# разместим синий текст, начиная с координат 32,32

$svg->text(x=>32,y=>32,

style=>"font-size:15;fill:blue"

)->cdata('Посещаемость сайта: распределение по доменам');

# вызовем подпрограмму для размещения 4-х полос графика

bar(20, 50, 'red', 55, '.RU');

bar(20, 80, 'blue', 24, '.COM');

bar(20, 110, 'green', 12, '.ORG');

bar(20, 140, 'black', 9, 'прочие');

# разместим текст с версиями программных средств:

$svg->text(x=>12,y=>200,

)->cdata("Perl $] + ". # версия Perl

"SVG.pm $SVG::VERSION + ". # версия модуля SVG

"CGI.pm $CGI::VERSION"); # версия модуля CGI

my $out = $svg->xmlify(); # отформатируем текст XML

print $out; # и отправим его браузеру

sub bar { # подпрограмма вывода одной строки графика,

# которой передаются координаты, цвет, % и заголовок

my ($x, $y, $color, $procent, $caption) = @_;

# выводим прямоугольник пропорционально проценту

$svg->rectangle(

x => $x, y => $y,

height => 30, width => $procent*10,

style => "opacity:1; fill:$color; fill-opacity:0.4"

);

# и пояснительный текст указанного цвета
```



```
$svg->text(

x=>$x+5,y=>$y+20,style=>"font-size:15;fill:$color"

)->cdata("$caption $procent %");

}
```

SVG-документ, сформированный в результате выполнения этой программы, - это текстовый файл в формате XML, который выглядит следующим образом:

```
"http://www.w3.org/TR/2001/REC-SVG-20010904/DTD/svg10.dtd">
```

```
īīññûûààîîññü
```

```
ñàéòà:
```

```
õàñīõããëëäíèä
```

```
īī äīīäíàì
```

```
style="opacity:1; fill:red; fill-opacity:0.4" />
```

```
.RU 55 %
```

```
style="opacity:1; fill:blue; fill-opacity:0.4" />
```

```
.COM 24 %
```

```
style="opacity:1; fill:green; fill-opacity:0.4" />
```

```
.ORG 12 %
```

```
style="opacity:1; fill:black; fill-opacity:0.4" />
```

ïðî:èâ 9 %

Perl 5.008007 + SVG.pm 2.33 + CGI.pm 3.10

Зная синтаксис описания SVG-графики, можно сформировать подобный документ с помощью одного из модулей Perl, генерирующих документы XML. Библиотека SVG лишь предоставляет для этого наиболее удобные средства. Если нет под рукой нужных модулей, можно даже создавать любые документы XML на чистом Perl. Текстовое представление, понятное человеку и легкое для обработки, стало одним из преимуществ XML по сравнению с применявшимися ранее двоичными форматами. На рис. 18.3 показано, как сформированный в программе SVG-документ выглядит в окне браузера в виде векторного изображения.

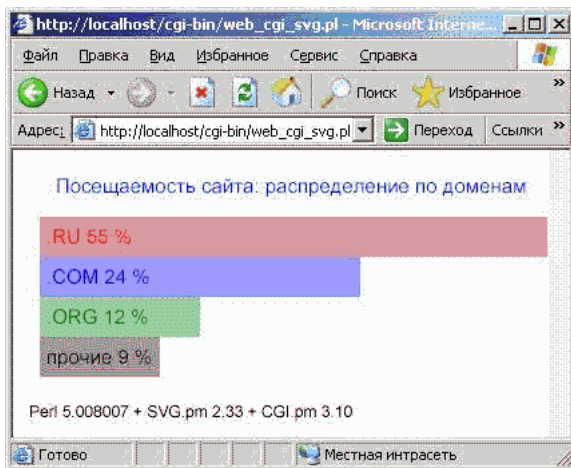


Рис. 18.3.Динамически сформированная SVG-графика

Первоначальное использование WWW только для доставки информации человеку постепенно сменяется использованием Всемирной Паутины для программного взаимодействия между информационными системами. Новым применением WWW стало использование ее в качестве пространства для распределенного компонентного программирования. В соответствии с этим подходом распределенные приложения строятся на основе сервис-ориентированной архитектуры. Для обращения к сетевым программным компонентам, называемым web-сервисами, используется протокол SOAP (Simple Object Access Protocol). Поскольку SOAP основан на стандарте XML, то он не зависит от используемого языка программирования и операционной системы. Данные передаются в виде текстовых сообщений в формате XML, поэтому могут передаваться с помощью неспециализированных протоколов, например, HTTP или SMTP. Чтобы правильно обратиться к web-службе, нужно знать ее интерфейс. Для описания методов, предоставляемых web-сервисами, и состава передаваемых данных создан язык описания web-сервисов WSDL (Web Services Description Language). А для хранения описаний web-сервисов в специальных реестрах и их поиска разработана система описания, обнаружения и интеграции UDDI (Universal Description, Discovery and Integration).

Среди средств, имеющихся в Perl для работы с web-сервисами, библиотека SOAP::Lite выделяется удобством использования и полнотой реализации необходимых протоколов. На простом примере покажем, насколько легко создать web-службу с ее помощью. Для начала напомним класс, который будет выполняться на сервере и предоставлять свои методы в виде web-сервисов. Это самый обычный класс:

```
package Calculator; # класс, реализующий простой калькулятор

sub add { # сложить

my ($self, $a, $b) = @_;

return $a + $b;

}

sub subtract { # вычесть

my ($self, $a, $b) = @_;

return $a - $b;
```

```

}

sub multiply { # умножить

my ($self, $a, $b) = @_;

return $a * $b;

}

sub divide { # разделить

my ($self, $a, $b) = @_;

return $b == 0 ? 0 : $a / $b;

}

1;

```

Затем разместим на web-сервере CGI-программу, которая будет выполнять роль диспетчера: при обращении по ее адресу будет происходить вызов требуемой web-службы. Она состоит всего из нескольких строк:

```

use SOAP::Transport::HTTP; # использовать протокол HTTP

SOAP::Transport::HTTP::CGI # для обращения через CGI

# к web-сервисам из этого каталога:

-> dispatch_to('/_Learn/Perl/web-services')

-> handle;

```

Затем напишем клиента для обращения к web-службам с использованием возможности перенаправления запросов, реализованной в библиотеке SOAP::Lite:

```

# включаем автоматическое

use SOAP::Lite +autodispatch => # перенаправление запросов

uri => 'urn:Calculator', # к классу Calculator

# при обращении по указанному адресу

proxy => 'http://localhost/cgi-bin/web_soap.cgi';

my $a = 5; # первый и

my $b = 3; # второй проверочные операнды

# вызываем методы класса на сервере:

print add($a, $b) , "\n"; # результат: 8

print subtract($a, $b), "\n"; # результат: 2

print multiply($a, $b), "\n"; # результат: 15

print divide($a, $b) , "\n"; # результат: 1.666666666666667

```

Для проверки работы этой web-службе была написана другая клиентская программа на языке VBScript. Чтобы показать, как выглядят передаваемые данные, результат работы метода multiply(5, 3) был выведен в виде неформатированного SOAP-сообщения, которое приведено на рис. 18.4.



Рис. 18.4.Пример SOAP-сообщения при обращении к web-сервису

Подобным способом через web-сервисы можно организовать доступ к методам любых других прикладных классов. В последнее время популярность получила технология AJAX (асинхронный доступ из JavaScript с помощью XML). Она может использоваться для доступа к web-службам из программ на JavaScript, превращая браузер в клиента прикладных классов, написанных на языке Perl. Пример обращения к разработанному нами web-сервису из JavaScript показан на рис. 18.5.



Рис. 18.5.Браузер как клиент web-сервисов

Программные средства, рассмотренные в этой лекции, охватывают только небольшую часть возможностей Perl для работы с ресурсами WWW. Web-программирование принесло языку Perl успех и большую популярность. Но Perl готов к переменам, которые неизбежно принесет нам будущее: ведь он создавался как расширяемый и адаптируемый язык. И он постоянно продолжает развиваться силами сообщества Perl-программистов, к числу которых теперь можете причислить себя и вы. Успехов вам в разработке программ на Perl и в дальнейшем изучении этого прекрасного языка программирования!

#### Литература

1. Barry P, Programming the Network with Perl, John Wiley & Sons, 2002
2. Bentley J, Programming Pearls, 2nd edition, Addison-Wesley, 2000
3. Blank-Edelman D.N, Perl for System Administration: Managing multiplatform environments with Perl, O'Reilly, 2000
4. Burke S.M, Perl & LWP, 1st edition, O'Reilly, 2002
5. Callender J, Perl for Web Site Management, O'Reilly, 2001
6. Christiansen T., Torkington N, The Perl Cookbook: Tips and Tricks for Perl Programmers, 2nd edition, O'Reilly, 2003
7. Conway D, Object Oriented Perl, Manning Publications, 1999
8. Descartes A., Bunce T, Programming the Perl DBI: Database programming with Perl, O'Reilly, 2000
9. Dominus M.J, Higher-Order Perl, 1st edition, Barnes & Noble, 2005
10. Friedl J.E. F, Mastering Regular Expressions: Powerful Techniques for Perl and Other Tools, O'Reilly, 1997
11. Guelich S., Gundavaram Sh., Birznies G, CGI Programming with Perl, 2nd edition, O'Reilly, 2000

12. Holzner S, Perl Black Book, 2nd edition, Paraglyph Press, 2001
13. Lidie S, Perl/Tk Pocket Reference, O'Reilly, 1998
14. Maher T, Minimal Perl For UNIX/Linux People, Manning Publications, 2006
15. Medinets D, Perl 5 by Example, Que, 1996
16. Menaker Y., Saltzman M., Oberg R.J, Programming Perl in the .NET Environment, 1st edition, Addison Wesley, 2002
17. Orwant J., Hietaniemi J., Macdonald J, Mastering Algorithms with Perl, O'Reilly, 1999
18. Randal A., Sugalski D., Totsch L, Perl 6 and Parrot Essentials, 2nd edition, O'Reilly, 2004
19. Ray E.T., McIntosh J, Perl and XML, 1st Edition, O'Reilly, 2002
20. Ray R.J., Kulchenko P, Programming Web Services with Perl, 1st edition, O'Reilly, 2002
21. Rolsky D., Williams K, Embedding Perl in HTML with Mason, 1st edition, O'Reilly, 2002
22. Schwartz R.L., Phoenix T, Learning Perl: Making Easy Things Easy and Hard Things Possible, 4th edition, O'Reilly, 2005
23. Schwartz R.L., Olson E., Christiansen T, Learning Perl on Win32 Systems, O'Reilly, 1997
24. Siever E., Spainhour S., Patwardhan N, Perl in a Nutshell, O'Reilly, 1998
25. Srinivasan S, Advanced Perl Programming, O'Reilly, 1997
26. Stein L.D, Network Programming with Perl, 1st Edition, Addison Wesley, 2000
27. Stein L., MacEachern D, Writing Apache Modules with Perl and C: The Apache API and mod\_perl, O'Reilly, 1999
28. Stubblebine T, Regular Expression Pocket Reference, O'Reilly, 2003
29. Vromans J, Perl 5 Pocket Reference: Programming Tools, 3rd edition, O'Reilly, 2000
30. Wallace Sh, Perl Graphics Programming: Creating SVG, SWF (Flash), JPEG and PNG files with Perl, O'Reilly, 2002
31. Wall L., Christiansen T., Orwant J, Programming Perl, 3rd edition, O'Reilly, 2000
32. Walsh N, Learning Perl/Tk: Graphical User Interfaces with Perl, O'Reilly, 1999
33. Walters S, Perl 6 Now: Core Ideas Illustrated With Perl 5, APress, 2004
34. Wong C, Web Client Programming with Perl: Automating Tasks on the Web, O'Reilly, 1977
35. Бентли, Д, Жемчужины творчества программистов. 1-е изд, М: Радио и связь, 1990
36. Бланк-Эдельман Д.Н, Perl для системного администрирования, СПб: Символ-Плюс, 2001
37. Браун М, Perl. Архив программ, М: БИНОМ, 2001
38. Гулич С., Гундавара Ш., Бирзнекс Г, CGI-программирование на Perl. 2-е изд, СПб: Символ-Плюс, 2001
39. Дейтел Х.М., Дейтел П.Дж., Нието Т.Р., МакФай Д.К, Как программировать на Perl, М: Бином, 2002
40. Декарт А., Банс Т, Программирование на Perl DBI, СПб: Символ-Плюс, 2000
41. Дюбуа П, Применение MySQL и Perl в web-приложениях, М.: Вильямс, 2002
42. Керниган Б., Плотджер Ф, Инструментальные средства программирования на языке Паскаль, М: Радио и связь, 1985
43. Кристиансен Т., Торкингтон Н, Perl. Сборник рецептов. 2-е изд, СПб: Питер-пресс, 2004

- 44. Ливингстон Д. и др, Perl 5. Web-профессионалам, Киев: BHV-Киев, 2001
- 45. Макинтош Дж., Рэй Э.Т, Perl & XML. Библиотека программиста, СПб: Питер, 2003
- 46. Маслов В.В, Основы программирования на языке Perl, М.: Радио и Связь, Горячая Линия-Телеком, 1999
- 47. Матросов А.В., Чаунин М.П, Perl. Программирование на языке высокого уровня, СПб: Питер, 2003
- 48. Мельтцер К., Михальски Б, Разработка CGI-приложений на Perl, М.: Вильямс, 2001
- 49. Рэндал Э., Сугальски Д., Теч Л, Perl 6 и Parrot. Справочник, М: Кудиц-Образ, 2005
- 50. Стаблибайн Т, Регулярные выражения. Карманный справочник, СПб: Питер, 2004
- 51. Фоули Р, Perl-отладчик. Карманный справочник, М: Кудиц-Образ, 2005
- 52. Фридл Дж, Регулярные выражения. 2-е изд, СПб: Питер, 2003
- 53. Шварц Р., Кристиансен Т, Изучаем Perl, Киев: BHV, 2000
- 54. Штайн Д. Л, Разработка сетевых программ на Perl, СПб: Вильямс, 2001
- 55. Уолл Л., Кристиансен Т., Орвант Дж, Программирование на Perl. 3-е изд, СПб: Символ-Плюс, 2002
- 56. Холзнер С, Perl: специальный справочник, СПб: Питер-Пресс, 2000

## Содержание

- 

Лекция 1. История развития Perl

- 

Лекция 2. Литералы и скалярные данные

- 

Лекция 3. Основные операции

- 

Лекция 4. Управляющие структуры

- 

Лекция 5. Списки и массивы

- 

Лекция 6. Хэши

- 

Лекция 7. Текст, строки и символы

- 

Лекция 8. Регулярные выражения

- 

Лекция 9. Средства ввода-вывода

- 

Лекция 10. Отчеты

- 

Лекция 11. Ссылки

- 

Лекция 12. Подпрограммы

- 

Лекция 13. Библиотеки, пакеты и модули

- 

Лекция 14. Объектное программирование

- 

Лекция 15. Работа с базами данных

- 

Лекция 16. Взаимодействие процессов

- 

Лекция 17. Работа в IP-сетях

- 

Лекция 18. Web-программирование

- 

Литература