

ИЗУЧАЕМ PERL

В книге излагаются основы программирования на языке Perl, который стал стандартным для большинства UNIX-платформ и все чаще используется в среде Windows. В США книга стала бестселлером и приобрела статус учебника, предназначенного как для занятия с преподавателем, так и для самостоятельной работы. В данном издании рассматривается современная версия Perl 5.004.

В каждой главе даются упражнения, а в конце книги — их подробные решения. Приведено множество примеров небольших программ, дано введение в CGI-программирование для Web, изложены методики использования системных команд в Perl-программах, рассмотрены способы создания с помощью Perl баз данных DBM и другие вопросы.

Книга предназначена для всех, кто желает научиться программировать на языке Perl.

Содержание

Предисловие	5	Операции над массивами и функции обработки массивов	79
Введение	9	Скалярный и списочный контексты	85
1. Введение	25	<STDIN> как массив	85
История создания языка Perl	25	Интерполяция массивов	86
Назначение языка Perl	26	Упражнения	87
Доступность	27	4. Управляющие структуры	89
Основные понятия	28	Блоки операторов	89
Прогулка по стране Perl	30	Оператор if/unless	90
Упражнение	57	Оператор while/until	92
2. Скалярные данные	59	Оператор for	94
Что такое скалярные данные	59	Оператор foreach	95
Числа	59	Упражнения	96
Строки	61	5. Хеши	97
Скалярные операции	63	Что такое хеш	97
Скалярные переменные	69	Хеш-переменные	97
Скалярные операции и функции	69	Литеральное представление хеша	98
<STDIN> как скалярное значение	74	Хеш-функции	99
Вывод с помощью функции print	75	Срезы хешей	101
Значение undef	75	Упражнения	102
Упражнения	76	6. Базовые средства ввода-вывода	103
3. Массивы и списочные данные	77	Ввод из STDIN	103
Список и массив	77	Ввод из операции "ромб"	104
Литеральное представление	77	Вывод в STDOUT	105
Переменные	79		

Упражнения	106	Небольшое отступление:	145
7. Регулярные выражения	107	функция <code>die</code>	
Основные понятия	107	Использование дескрипторов	146
Основные направления	107	файлов	
использования регулярных		Операции для проверки	147
выражений		файлов	
Образцы	109	Функции <code>stat</code> и <code>lstat</code>	150
Еще об операции	116	Упражнения	151
сопоставления		11. Форматы	153
Операция замены	120	Что такое формат	153
Функции <code>split</code> и <code>join</code>	121	Определение формата	154
Упражнения	123	Вызов формата	156
8. Функции	125	Еще о поледержателях	157
Определение	125	Формат начала страницы	161
пользовательской функции		Изменение в форматах	162
Вызов пользовательской	126	установок по умолчанию	
функции		Упражнения	166
Возвращаемые значения	127	12. Доступ к каталогам	167
Аргументы	128	Перемещение по дереву	167
Локальные переменные в	129	каталогов	
функциях		Развертывание	168
Полулокальные переменные,	131	Дескрипторы каталогов	170
созданные при помощи		Открытие и закрытие	170
функции <code>local</code>		дескриптора каталога	
Создаваемые операцией <code>my()</code>	132	Чтение дескриптора каталога	171
переменные файлового		Упражнения	172
уровня		13. Манипулирование	173
Упражнения	133	файлами и каталогами	
9. Управляющие	135	Удаление файла	173
структуры		Переименование файла	174
Оператор <code>last</code>	135	Создание для файла	175
Оператор <code>next</code>	137	альтернативных имен:	
Оператор <code>redo</code>	137	связывание ссылками	
Метки	138	Создание и удаление	178
Модификаторы выражений	139	каталогов	
Операции <code>&&</code> и <code> </code> как	141	Изменение прав доступа	178
управляющие структуры		Изменение принадлежности	179
Упражнения	142	Изменение меток времени	180
10. Дескрипторы файлов и	143	Упражнения	180
проверка файлов		14. Управление процессами	183
Что такое дескриптор файла	143	Использование функций	183
Открытие и закрытие	144	<code>system</code> и <code>exec</code>	
дескриптора файла		Использование обратных	186

кавычек		18. Преобразование других программ в Perl-программы	225
Использование процессов как дескрипторов файлов	187	Преобразование awk-программ в Perl-программы	225
Использование функции fork	189	Преобразование sed-программ в Perl-программы	227
Сводка операций, проводимых над процессами	191	Преобразование shell-сценариев в Perl-программы	227
Передача и прием сигналов	192	Упражнение	228
Упражнения	195	19. CGI-программирование	229
15. Другие операции преобразования данных	197	Модуль CGI.pm	230
Поиск подстроки	197	Ваша CGI-программа в контексте	231
Извлечение и замена подстроки	198	Простейшая CGI-программа	233
Форматирование данных с помощью функции sprintf()	200	Передача параметров через CGI	235
Сортировка по заданным критериям	201	Как сократить объем вводимого текста	236
Транслитерация	204	Генерирование формы	238
Упражнения	207	Другие компоненты формы	240
16. Доступ к системным базам данных	209	Создание CGI-программы гостевой книги	244
Получение информации о паролях и группах	209	Поиск и устранение ошибок в CGI-программах	254
Упаковка и распаковка двоичных данных	213	Perl и Web: не только CGI-программирование	256
Получение информации о сети	215	Дополнительная литература	260
Упражнение	216	Упражнения	260
17. Работа с пользовательскими базами данных	217	Приложение А. Ответы к упражнениям	261
DBM-базы данных и DBM-хеши	217	Приложение Б. Библиотеки и модули	289
Открытие и закрытие DBM-хешей	218	Приложение В. Сетевые клиенты	298
Использование DBM-хеша	219	Приложение Г. Темы, которых мы не коснулись	305
Базы данных произвольного доступа с записями фиксированной длины	220	Предметный указатель	311
Базы данных с записями переменной длины (текстовые)	222		
Упражнения	224		

Предметный указатель

А		Игнорирование регистра	117
Аргументы	128	Извлечение и замена	198
Ассоциативные массивы	97	подстроки	
Б		Изменение меток времени	180
Базовые средства ввода-вывода	103	Изменение прав доступа	178
Базы данных		Изменение принадлежности файла	179
— пользовательские	217	Индексные ссылки	34
— произвольного доступа с записями фиксированной длины	220	Интерполяция переменных	63, 118
— с записями переменной длины (текстовые)	222	К	
Библиотека		Каталоги	167
- DBM	217	— создание	178
- LWP	258	— удаление	178
Библиотеки	292	Классы	246
Блоки операторов	89	Ключи	97
В		Компоненты формы	238
Возвращаемые значения	40, 127	Конечная лексема	234
Выбранный в текущий момент дескриптор файла	162	Константы	60
Г		Конструкторы	246
Глобальные переменные	126	Круглые скобки в образцах	113
Гнезда	302	Л	
Голые блоки	136	Лексемы	29
Д		Литералы	60
Действия	192	Литеральные строки	61
Деление с остатком	64	— строки в двойных кавычках	62
Дескриптор по умолчанию	163	— строки в одинарных кавычках	61
Дескриптор файла по умолчанию	147	Лицензия	
Дескрипторы		— открытая	27
— каталогов	170	— художественная	27
— файлов	43, 143	Локальные переменные в функциях	129
закрытие	144	М	
использование	146	Маркер границы слова	37
открытие	144	Массивы	77
Директивы импорта	236	Метки	138
И		Методы	246
		— классов	247
		— объектов	247
		Множители	111

— ленивые	113	— космический корабль	202
— прожорливые	113	— стр	204
Модификаторы выражений	140	— each	219
Модули	292	— tr	204
Модуль CGI.pm	231	— =~	117
О		Основная программа	29
Образцы	109	Отступы	29
— классы символов	109	П	
— обозначающие один	109	Пакеты	292
символ		Параметры	40
— приоритет	115	Передача параметров	235
— фиксированные	114	Переменные среды	185
Общий множитель	112	Переменные-массивы	79
Общий шлюзовой интерфейс (CGI)	229	Подпрограммы	40, 125
Объектно-ориентированное программирование	246	Поиск	
Объекты	246	— ошибок в CGI-программах	254
Оператор		— подстроки	197
— do {} while/until	93	— с возвратом	113
-for	94	Поледержатели	154
— foreach	95	— заполненные поля	159
— if/unless	90	— многостроковые	158
— last	135	— текстовые	157
— next	137	— числовые	158
— redo	137	Полулокальные переменные	131
— use	235	Пользовательские функции	
— while/until	92	— вызов	126
Операторы	89	— определение	125
Операции над массивами	79	Помеченные блоки	138
Операция		Порты	302
— замены	109, 120	Последовательности	111
— конструктора списка	78	Права доступа	178
— перевода	38	Прагмы	292
— повторения строки	66	Практический язык	25
— подстановки	38	извлечений и отчетов	
— присваивания	69	Пробельные символы	28
— ромб	104	Процессы	183
— сопоставления	37, 108, 116	Путающее зависшее else	90
— игнорирования регистра	37	Р	
		Рабочая область	95
		Разбивка	121
		Развертывание	168
		Реализации	129

Регулярные выражения	107
Редактирование на месте	222
С	
Связывание ссылками	175
Сервисы	302
Сетевые клиенты	303
Сети	301
Сигналы	192
Скалярные данные	59
Скалярные операции	63
Скалярные переменные	31,69
Скалярный контекст	85
Служебный язык систем	29
Сохраняемый файл	252
Списки	77
Списочные литералы	77
Списочный контекст	85
Срез	82
Ссылка на список	80
Ссылки	242
— жесткие	175
— символические	176
Стандартный ввод	103
Стандартный вывод	105
Строка	
— запроса	231
— значений полей	50
— определения полей	50
Строки	61
— значений	154
— полей	154
— связи	122
Т	
Транслитерация	204
У	
Универсальные локаторы ресурсов (URL)	231
Управляющие структуры	89
Усовершенствованная сортировка	201
Устройства графического ввода	238
Ф	

Файлы	
— переименование	174
— удаление	173
Фиксирующие точки	114
— упреждающие	115
Формат начала страницы	51,
	161
Форматы	153
— вызов	156
— определение	154
Функции обработки массивов	79
Функция	
— chdir	167
— chmod	178
— chown	179
— closedir	170
— dbmclose	219
— dbmopen	218
— die	145
— exit	190
— flock	245
— fork	189
— glob	49,
	168
— glue	122
— index	197
— link	176
— mkdir	178
— open 144,	,221
— opendir	170
— pack	221
— param	235
— print	221
— printf	200
— read	221
— readdir	171
— readlink	177
— rename	174
— rmdir	178
— seek	221
— select	163
— sort	201

— split	122	D	
— substr	198	DBM-базы данных	217
— system	183	DBM-массивы	218
— unlink	173	DBM-хеши	218
— utime	180	H	
— wait	189	Here-документы	234
— write	156	HTML-формы 232,	238
		HTTP	232
X		P	
Хеш-переменные	98	Perl	
Хеш-функции	99	— доступность	27
Хеши	97	— и awk	225
Ч		— и sed	227
Числа	59	— и shell	228
С		— история создания	25
CGI-программирование	229	— назначение	26
CGI-программы	229	W	
CPAN (Comprehensive Perl	17	World Wide Web	229
Archive Network)			

Предисловие

Внимание, класс! Внимание! Спасибо.

Приветствую вас, начинающие волшебники. Надеюсь, летние каникулы принесли вам море удовольствия и, наверное, показались слишком короткими. Позвольте мне быть первым, кто приветствует вас в Колледже колдовства, а точнее — на этом вводном курсе по Волшебству Perl. Я не постоянный ваш преподаватель, но профессор Шварц немного задерживается и попросил меня, как создателя языка программирования Perl, выступить сегодня здесь с несколькими вступительными замечаниями.

Итак, давайте посмотрим, с чего начать. Для кого из вас этот курс — новый? Вижу, вижу. М-да, в свое время бывало и похуже. Редко, но бывало. *Очень* редко.

Что вы говорите? Это была шутка. Правда! Ну хорошо, хорошо. У этих новичков нет никакого чувства юмора...

Итак, о чем я буду говорить? Конечно, есть очень много вещей, о которых я *мог* бы поговорить. Я мог бы заняться самолюбованием и поговорить о себе, вспоминая все те причуды генетики и воспитания, которые привели меня к созданию языка Perl и вообще поставили меня в глупое положение. Это было бы очень увлекательно (по крайней мере для меня).

Я мог бы также поговорить о профессоре Шварце, без чьих постоянных усилий мир Perl существенно обеднел бы, вплоть до того, что этот курс не существовал бы как таковой.

Это было бы интересно, но у меня такое чувство, что к концу изучения данного курса вы узнаете о профессоре Шварце больше меня.

Наконец, я мог бы, отбросив саморекламу, поговорить просто о самом языке Perl, который, в конце концов, является предметом курса.

А является ли? Гм-м...

При обсуждении этого курса мы, преподаватели, пришли к выводу, что он посвящен не столько Perl, сколько вам самим! Не нужно так удивляться, потому что Perl действительно посвящен вам самим, по крайней мере в некотором абстрактном смысле. Perl был создан для таких, как вы, такими, как вы, при участии таких, как вы. Волшебство Perl была соткано многими людьми, стежок за стежком, узор за узором, по довольно специфической форме в том числе и вашей души. Если вы считаете, что Perl несколько необычен, то причина — в этом.

Некоторые ученые, работающие в области информатики (в частности, редуccionисты), будут отрицать это, но человеческий ум построен очень своеобразно. Ментальный рельеф нелинеен, и его нельзя отобразить на плоскую поверхность без существенных искажений. Тем не менее в последние два десятка лет компьютерные редуccionисты сначала склонялись перед Храмом Ортогональности, а затем поднимали головы, чтобы проповедовать идеалы аскетизма всем, кто хотел их слушать.

Их горячим, но ошибочным желанием было загнать ваш ум в рамки своего умственного стандарта, расплющить ваш образ мыслей, превратив его в некий плоский мир. Каким безрадостным было бы такое существование!

Ваш собственный здравый смысл, тем не менее, дал о себе знать. Вы и ваши идейные предшественники вышли за рамки этого невыносимо скучного пейзажа и создали множество прекрасных магических компьютерных формул. (Некоторые из них временами действительно делали то, чего от них хотели.) Самые эффективные из этих формул были канонизированы как Стандарты, потому что они смогли осуществить нечто мистическое и волшебное, сотворить чудо, имя которому — “Получить То, Что Вы Хотите”.

Находясь в состоянии эйфории, вы не заметили, что компьютерные редуccionисты продолжали пытаться расплющить ваши умы, пусть и на несколько более высоком уровне бытия. Поэтому вышел указ (я уверен, что вы слышали о нем), согласно которому магическим компьютерным формулам было разрешено творить только по одному чуду. “Делать что-то одно и делать хорошо” — вот что стало главным лозунгом, и одним росчерком пера программисты, работающие с командными процессорами (более известными как shell), были обречены всю жизнь бормотать и считать бусины четок на нитках (роль которых с недавних пор стали выполнять конвейеры).

Вот тогда я и внес свой маленький вклад в дело спасения мира. Как-то раз я перебирал в руках эти самые четки и размышлял о безнадежности (и случайности) моего существования, и вдруг мне пришла в голову одна мысль: а что, если расплавить несколько этих мистических бусинок и посмотреть, как изменится волшебная сила, если сделать из них одну большую бусину? Я зажег старую бунзеновскую горелку, выбрал свои любимые бусинки и сплавил их в одну большую. И что же — ее волшебство оказалось более сильным, чем сумма волшебства ее частей!

Странно, подумал я. Почему это сплав Прилежной Бусины Регулярных Выражений с Разрушительной Бусиной Гностической Интерполяции и Стеснительной Бусиной Простой Топологии Данных должен дать больше

Волшебства, чем дают эти бусины в совокупности, нанизанные на нитку? Я спросил себя: может быть, эти бусины могут обмениваться силой друг с другом, потому что им больше не нужно общаться между собой через эту тонкую ниточку? Может быть, конвейер сдерживает поток информации подобно тому, как вино не хочет течь через горлышко знаменитой бутылки доктора фон Неймана?

Эти вопросы потребовали (от меня) более тщательного исследования.

Я сплавил полученную бусину еще с несколькими из моих любимых бусинок — и произошло то же самое, только эффект был сильнее. Это был настоящий комбинаторный взрыв потенциальных магических формул: Бейсик-Бусина Форматов Вывода и Лисп-Бусина Динамических Контекстов слились с С-рационализированной Бусиной Изобилия Операций, что дало мощный импульс силы, который распространился на тысячи машин по всему цивилизованному миру. Рассылка посвященного этому событию сообщения обошлась Сети в сотни, если не в тысячи, долларов. Очевидно, я либо что-то уже знал, либо подошел к этому знанию вплотную.

Итак, я собрался с духом и показал свою новую волшебную бусину некоторым из вас, а вы начали посылать мне свои любимые бусины, чтобы я их сплавил со своей. Волшебная сила выросла еще больше. Все происходило так, словно Вычислительные Элементарные Частицы, связанные с каждой бусинкой, взаимодействовали от вашего имени и решали проблемы за вас. Откуда этот внезапный мир на земле и добрая воля к сотрудничеству? Может быть, причина в том, что эти бусинки были вашими любимыми? А может быть, я просто хороший подборщик бус?

Скорее всего, мне просто повезло.

Как бы там ни было, волшебная бусина в конце концов превратилась в тот причудливый Амулет, который вы сегодня видите перед собой. Смотрите, как он блестит — почти как жемчуг*!

Это тоже шутка. Правда, уверяю вас! Ну хорошо, я тоже когда-то был новичком... Этот Амулет на самом деле не совершенен; при близком рассмотрении видно, что это множество сплавленных бусинок. Да-да, я признаю это. Пусть он будет крайне безобразным, но не обращайтесь на это внимания. Самое главное — Волшебная Сила... Посмотрите, кто пришел! Мой добрый приятель Мерлин, то есть профессор Шварц, который сейчас начнет рассказывать вам о том, как творить чудеса с этим маленьким Амулетом, если у вас есть желание выучить необходимые магические формулы. Я уверен, вы попали в хорошие руки. Потому что никто не может бормотать магические заклинания лучше, чем профессор Шварц. Правда, Мерлин?

Подведем итоги. Больше всего вам будет необходима смелость. Путь, на который вы встали, труден. Вы учите новый язык, полный таинственных рун и древних песен — легких и трудных, знакомых и незнакомых. У вас может возникнуть искушение бросить все и уйти, но подумайте: сколько времени вам понадобилось, чтобы выучить свой родной язык? Стоило ли этим заниматься? Я думаю, да. Закончили ли вы изучение языка? Думаю, нет.

* Игра слов: в английском языке Perl созвучно с pearl (жемчуг). — Прим. перев.

Поэтому не ждите, что вам удастся познать все тайны языка Perl за секунду — это вам не орех и не маслина. Это, скорее, банан. Чтобы насладиться вкусом банана, вы не ждете, пока он будет съеден целиком, а наслаждаетесь каждым кусочком. Каждый кусочек побуждает вас откусить еще и еще.

Поэтому, переходя к плодам труда моего друга Мерлина, я призываю вас насладиться этим курсом. Курсом как вкусным плодом, конечно. Ну-ну, это тоже шутка...

Итак, профессор, я представляю вам ваш новый класс. Кажется, у этих учеников полностью отсутствует чувство юмора, но я надеюсь, вы как-нибудь справитесь.

Класс, я представляю вам профессора Рэндала Л. Шварца, доктора синтаксиса, чародея регулярных выражений и, конечно, еще одного Perl-хакера. Благословляю его, как и вас всех. Желаю вам выучить Perl. Желаю вам творить с его помощью доброе волшебство. Желаю, наконец, получить от Perl массу удовольствия. Да будет так!

Да сделаете вы так!

Ларри Уолл
Сентябрь 1993 г.

Дополнение ко второму изданию

К тебе это тоже относится, Том.

Ларри Уолл
Май 1997 г.

Введение

О чем эта книга

Прежде всего следует сказать, что эта книга насчитывает около 320 страниц. Кроме того, она построена как последовательное введение в Perl. Изучив весь этот материал, вы будете знать самые простые операции и основные лексемы, встречающиеся в большинстве Perl-программ.

Авторы книги не ставили перед собой задачу сделать ее полным руководством по языку Perl; наоборот, чтобы она не выросла до неуправляемо больших размеров, мы решили осветить лишь те конструкции и возможности, которые вы вероятнее всего будете использовать на ранней стадии своей Perl-программистской карьеры.

Тем не менее в качестве прелюдии к более глубокому изучению языка мы включили в книгу серьезную главу о CGI-программировании. В ней также затрагиваются такие темы, как библиотечные модули, ссылки и объектно-ориентированное программирование на Perl. Надеемся, что она пробудит у вас интерес к этим более сложным темам.

В конце каждой главы дается несколько упражнений, призванных помочь вам попрактиковаться в том, о чем вы прочли в этой главе. Если вы будете читать в нормальном темпе и делать все упражнения, то сможете освоить каждую главу за два-три часа, а всю книгу — за 30-40 часов.

Мы считаем, что эту книгу нужно использовать совместно с классическим полным справочником по Perl — *Programming Perl, Second Edition*, by Larry Wall, Randal L. Schwartz, and Tom Christiansen (издательство O'Reilly & Associates).

Задуманный первоначально как язык для операционной системы UNIX, Perl сейчас работает практически везде, включая MS-DOS, VMS, OS/2, Plan 9, Macintosh и все известные разновидности Windows. Это один из

наиболее переносимых языков программирования, известных на сегодняшний день. За исключением тех нескольких разделов, которые относятся к управлению UNIX-системами, информация из большей части этой книги применима к любой платформе, на которой работает Perl.

Где найти упражнения

Упражнения, приведенные в этой книге, можно получить в электронном варианте разными способами: по FTP, FTPMAIL, BITFTP и UUCP. Самые дешевые, самые быстрые и самый легкие способы указаны первыми. Если читать сверху вниз, то самый лучший метод — это, вероятно, первый из тех, которые работают. При непосредственной работе в Internet используйте FTP. Если у вас нет прямого соединения с Internet, но вы можете посылать электронную почту на узлы Internet и получать ее с этих узлов, используйте FTPMAIL. Если вы посылаете электронную почту по BITNET, используйте BITFTP. Если ни один из этих способов не действует, используйте UUCP.

Примечание: упражнения разрабатывались с помощью UNIX-системы. Если вы работаете в среде UNIX, можете использовать их, не корректируя. При работе на другой платформе эти упражнения, возможно, придется слегка модифицировать. Например, при работе с UNIX каждая строка завершается символом новой строки (возврат каретки подразумевается), тогда как в DOS каждая строка должна завершаться явно указанными символами и новой строки, и возврата каретки. В зависимости от конфигурации вашей системы и используемого метода пересылки упражнений по сети вам, возможно, придется добавить символы возврата каретки. Дополнительную информацию по этому вопросу можно найти в файле *README*, который прилагается к упражнениям.

FTP

Чтобы использовать FTP, вам понадобится компьютер, имеющий непосредственный выход в Internet. Ниже приведен пример сеанса связи.

```
% ftp ftp.ora.com
Connected to ftp.uu.net
220 ftp.ora.com FTP server (Version 6.34 Thu Oct 22 14:32:01 EDT 1992)ready.
Name (ftp.ora.com:username): anonymous
331 Guest login ok, send e-mail address as password.
Password: username@hostname (здесь используйте свое пользовательское имя и хост-имя)
230 Guest login ok, access restrictions apply.
ftp> cd /published/oreilly/nutshell/learning_perl2
250 CWD command successful.
ftp> get README
200 PORT command successful.
150 Opening ASCII mode data connection for README (xxxx bytes).
226 Transfer complete.
local: README remote: README
```

```

xxxx bytes received in xxx seconds (xxx Kbytes/s)
ftp> binary
200 Type set to I.
ftp> get examples.tar.gz
200 PORT command successful.
150 Opening BINARY mode data connection for examples.tar.gz (xxxx bytes).
226 Transfer complete. local: exercises remote: exercises
xxxx bytes received in xxx seconds (xxx Kbytes/s)
ftp> quit
221 Goodbye.
%
```

FTPMAIL

FTPMAIL — это почтовый сервер, доступный каждому, кто имеет возможность посылать электронную почту на узлы Internet и получать ее оттуда. Доступ к FTPMAIL обеспечивают все провайдеры Internet, предоставляющие услуги электронной почты. Вот как это можно сделать.

Вы посылаете почту на узел *ftpmail@online.ora.com*. В теле сообщения дайте FTP-команды, которые хотите выполнить. Сервер запустит для вас FTP-сеанс от имени анонимного пользователя и pošлет вам необходимые файлы. Чтобы получить полный справочный файл, pošлите сообщение без указания темы и с телом, состоящим из одного слова — *help*. Ниже приведен пример сеанса работы с электронной почтой в UNIX, в ходе которого пользователь получает программы-примеры. Эта команда посылает вам перечень файлов, имеющихся в текущем каталоге, и затребованные файлы примеров. Данный перечень полезен, если существуют более поздние версии примеров, которые вас интересуют.

```

% mail ftpmail@online.ora.com
Subject:
reply-to username@hostname      (на какой компьютер вы хотите принять файлы)
open
cd /published/oreilly/nutshell/learning_perl2
dir
get README
mode binary
uuencode
get examples.tar.gz
quit
.
```

Подпись в конце сообщения приемлема, если она стоит после команды *quit*.

BITFTP

BITFTP — это почтовый сервер для пользователей сети BITNET. Вы посылаете на него сообщения электронной почты с запросами на получение файлов, а сервер посылает по электронной почте указанные вами файлы. Сейчас BITFTP обслуживает только тех пользователей, которые посылают на него почту с узлов, непосредственно включенных в BITNET, EARN или NetNorth. BITFTP — это общедоступный сервис Принстонского университета.

Чтобы воспользоваться услугами BITFTP, пошлите почтовое сообщение с необходимыми FTP-командами по адресу *BITFTP@PUCC*. Если вы хотите получить полный справочный файл, пошлите в теле сообщения слово *HELP*.

Ниже приведено тело сообщения, которое следует послать на BITFTP:

```
FTP ftp.ora.com NETDATA
USER anonymous
PASS ваш электронно-почтовый Internet-адрес (а не BITNET-адрес)
CD /published/oreilly/nutshell/perl/learning_perl2
DIR
GET README
GET examples.tar.gz
QUIT
```

Вопросы, касающиеся самого BITFTP, следует направлять на узел BITNET *MAINT@PUCC*.

UUCP

Если у вас или у вашей организации есть доступ к UUNET, это значит, что у вас должна быть система, имеющая прямое UUCP-соединение с этой сетью. Найдите эту систему и введите (в одну строку):

```
uucp uunet\!~/published/oreilly/nutshell/learning_perl2/examples.tar.gz
ваш_хост\!~/ваше_имя/
```

Если вместо *csh*-shell вы пользуетесь Bourne-shell (*sh*), обратные косые можно опустить. Файл примеров должен появиться через некоторое время (день или более) в каталоге */usr/spool/uucppublic/ваше_имя*.

Другие ресурсы

Ман-страницы Perl

Диалоговая документация на Perl, которую из-за ее UNIX-происхождения называют *ман-страницами**, разделена на несколько частей, чтобы пользователь мог быстро найти необходимый материал, не перебирая сотни страниц текста. Поскольку ман-страница верхнего уровня называется просто *perl*, то на нее вас приведет UNIX-команда *man perl* **. Эта страница, в свою очередь, поможет вам найти другие страницы. Например, команда *man perlre* выдает ман-страницу о регулярных выражениях Perl. Команда *perldoc* может помочь, если не срабатывает команда *man(1)*, особенно в случае с документацией на модули, которую ваш системный администратор не захотел установить вместе с обычными ман-страницами. С другой стороны, ваш системный администратор мог установить документацию на Perl в формате HTML, особенно в системах, отличных от UNIX. Если все остальное не работает, вы всегда можете получить документацию на Perl из CPAN; соответствующая информация приведена в разделе “Как получить Perl”.

Вот основные ман-страницы, включенные в дистрибутив Perl 5.004:

Ман-страница	Тема
<i>perl</i>	Обзор документации
<i>perldelta</i>	Изменения по сравнению с последней версией
<i>perlfaq</i>	Часто задаваемые вопросы
<i>perldata</i>	Структуры данных
<i>perlsyn</i>	Синтаксис
<i>perlop</i>	Операции и приоритет
<i>perlre</i>	Регулярные выражения
<i>perlrun</i>	Выполнение и опции
<i>perlfunc</i>	Встроенные функции
<i>perlvar</i>	Предопределенные переменные
<i>perlsub</i>	Подпрограммы
<i>perlmod</i>	Модули: принцип работы
<i>perlmodlib</i>	Библиотечные модули: методика написания и использования
<i>perlform</i>	Форматы
<i>perllocale</i>	Поддержка локализации (“культурной среды”)

* Сокращение от manual pages, т.е. “страницы руководства”. — Прим. перев.

** Если ничего не получается, то вы, вероятно, пытаетесь получить ман-страницу древней версии 4. Возможно, придется изменить переменную среды MANPATH.

Ман-страница	Тема
<i>perlref</i>	Ссылки
<i>perldsc</i>	Введение в структуры данных
<i>perllool</i>	Структуры данных: списки списков
<i>perltoot</i>	Пособие по объектно-ориентированному программированию
<i>perlobj</i>	Объекты
<i>perltie</i>	Объекты, скрытые за простыми переменными
<i>perlbot</i>	Хитрости и примеры использования объектов
<i>perlipc</i>	Межпроцессное взаимодействие
<i>perldebug</i>	Отладка
<i>perldiag</i>	Диагностические сообщения
<i>perlsec</i>	Безопасность
<i>perltrap</i>	Ловушки для неосторожных
<i>perlstyle</i>	Руководство по стилю
<i>perlpod</i>	Старая документация в виде простого текста
<i>perlbook</i>	Информация о книгах
<i>perlembed</i>	Методы встраивания Perl-кода в приложение, написанное на C или C++
<i>perlpio</i>	Внутренний интерфейс абстрагирования ввода-вывода
<i>perlxs</i>	Интерфейс прикладного программирования XS
<i>perlxsut</i>	Пособие по XS
<i>perlguis</i>	Внутренние функции для тех, кто разрабатывает расширения
<i>perlcall</i>	Правила вызова из C

Телеконференции Usenet

Телеконференции по Perl — неиссякаемый источник информации (правда, иногда беспорядочной) о языке. Телеконференция *comp.lang.perl.announce* — с низким трафиком, используется для публикации объявлений, связанных с Perl. Эти объявления часто касаются выпуска новых версий, исправления ошибок, новых расширений и модулей, часто задаваемых вопросов (FAQ).

В телеконференции *comp.lang.perl.misc*, уровень трафика в которой очень высок, обсуждается практически все — от технических вопросов и философии Perl до Perl-игр и Perl-поэзии. Как и сам Perl, эта телеконференция слывет полезной, и ни один вопрос не считается слишком глупым, чтобы его нельзя было задавать*.

* Разумеется, некоторые вопросы слишком просты, чтобы на них отвечать, особенно те, на которые уже даны ответы в FAQ

В телеконференции *comp.lang.perl.tk* обсуждаются вопросы использования популярного набора инструментальных средств Tk, входящего в состав Perl. Телеконференция *comp.lang.perl.modules* посвящена разработке и использованию Perl-модулей — самого лучшего средства получения многократно используемого кода. Когда вы будете читать эти строки, возможно, уже появятся и другие телеконференции.

Есть еще одна телеконференция, к материалам которой вы, может быть, захотите обратиться (по крайней мере если занимаетесь CGI-программированием в Web) — *comp.infosystems.www.authoring.cgi*. Хотя, строго говоря, эта конференция и не посвящена Perl как таковому, большинство обсуждаемых там программ написаны на этом языке. Именно к ее материалам следует обращаться по вопросам, связанным с применением Perl в World Wide Web.

Домашняя страница Perl

Если у вас есть доступ к World Wide Web, посетите домашнюю страницу Perl по адресу <http://www.perl.com/perl/>. Здесь вы узнаете, что нового произошло в мире Perl, сможете получить исходный код и номера портов, документацию, модули третьих фирм, базу данных ошибок Perl, информацию о списках рассылки и т.д. На этом узле имеется также сервис мультимедийного архивирования CPAN, который описан ниже.

Сборник часто задаваемых вопросов

Часто задаваемые вопросы (FAQ) по Perl — это собрание вопросов и ответов, которые часто появляются в телеконференции *comp.lang.perl.misc*. Во многих отношениях это собрание можно рассматривать как дополнение к имеющимся книгам. Здесь разъясняются понятия, в которых пользователи, возможно, не разобрались, и сообщается оперативная информация о таких вещах, как последняя редакция и лучший источник для получения исходного кода Perl.

Этот сборник FAQ периодически публикуется в телеконференции *comp.lang.perl.announce*. Кроме того, его можно найти в Web по адресу <http://www.perl.com/perl/faq>.

Начиная с версии Perl 5.004, этот FAQ включен в документацию стандартного дистрибутива. Вот его основные разделы, каждый из которых оформлен как отдельная map-страница:

perlfaq

Структурный обзор FAQ.

perlfaq1

Очень общая, высокоуровневая информация о языке Perl.

perlfaq2

Где найти исходный код и документацию на Perl, вопросы поддержки, обучения и сопутствующие вопросы.

perlfaq3

Инструментарий программиста.

perlfaq4

Обработка чисел, дат, строк, массивов, хешей и разнообразные аспекты обработки данных.

perlfaq5

Ввод-вывод, дескрипторы файлов, запись на диск, форматы, нижние колонтитулы.

perlfaq6

Сопоставление с образцами и регулярные выражения.

perlfaq7

Общие вопросы, которые нельзя отнести ни к одной из других категорий.

perlfaq8

Межпроцессное взаимодействие, управление пользовательским интерфейсом: клавиатура, экран, координатно-указательные устройства.

perlfaq9

Сети, Internet и кое-что о Web.

Сообщения о дефектах

В том невероятном случае, если вы наткнетесь на дефект не в вашей собственной программе, а в самом Perl, постарайтесь проверить его на минимальном по объему контрольном примере, а затем документировать с помощью программы *perlbug*, которая поставляется вместе с исходным кодом Perl.

Как распространяется Perl

Perl распространяется по одной из двух лицензий (на ваш выбор). Первая — стандартная форма GNU Copyleft. Коротко говоря, это означает, что если вы можете выполнять Perl в своей системе, то должны иметь доступ к полному исходному коду языка без дополнительной платы. Вторая форма — Artistic License, которую некоторые (особенно юристы) находят менее угрожающей, нежели Copyleft.

В каталоге */eg* дистрибутива Perl вы найдете ряд программ-примеров. Есть и другие лакомые кусочки — можете посвятить их поиску один из дождливых дней. Изучите исходный код Perl (если вы — C-хакер с мазохистскими наклонностями). Взгляните на тестовый комплект. Посмотрите, как *Configure* определяет наличие системного вызова *mkdir(2)*. Определите, как Perl выполняет динамическую загрузку C-модулей. В общем, делайте все, что вам по душе.

Другие книги

Programming Perl — полный справочник по Perl, тогда как нашу книгу скорее можно назвать пособием. Если вы хотите больше узнать о регулярных выражениях, используемых в Perl, предлагаем вам книгу *Mastering Regular Expressions* by Jeffrey E.F. Friedl (O'Reilly & Associates).

Посмотрите также вышедшие в издательстве O'Reilly & Associates книги *CGI Programming on the World Wide Web* by Shishir Gundavaram; *Web Client Programming with Perl* by Clinton Wong; *HTML: The Definitive Guide* by Chuck Musciano and Bill Kennedy.

Книги *The AWK Programming Language* by Aho, Kernighan, and Weinberger (Addison-Wesley) и *sed & awk* by Dale Dougherty (O'Reilly & Associates) содержат обширный базовый материал по таким вопросам, как ассоциативные массивы, регулярные выражения и общее мировоззрение, благодаря которому появился Perl. В них приведено множество примеров, которые можно перевести на Perl с помощью конвертора *awk-to-perl* (*a2p*) или конвертора *sed-to-perl* (*s2p*). Конечно, эти конверторы не выдают совершенный Perl-код, но если вы не сможете реализовать один из этих примеров в Perl, выходная информация конвертора даст вам хорошую подсказку.

Для Web-мастеров мы рекомендуем второе издание книги *How to Setup and Maintain a Web Site* by Lincoln Stein (Addison Wesley). Д-р Штейн, известный как автор Perl-модуля CGI.pm (см. главу 19), профессионально и всесторонне рассматривает все вопросы, связанные с администрированием Web-узла на платформах UNIX, Mac и Windows.

Мы также рекомендуем удобный и тщательно проработанный краткий справочник *Perl 5 Desktop Reference* by Johan Vromans (O'Reilly & Associates).

Как получить Perl

Основной пункт распространения Perl — это *Comprehensive Perl Archive Network*, или CPAN (Сеть полных Perl-архивов). Эти архивы содержат не только исходный код, но и практически все материалы, которые вам когда-либо понадобятся для работы с Perl. CPAN зеркально копируется десятками узлов по всему свету. Главный узел — *ftp.funet.fi* (128.214.248.6). Вы можете найти и более близкий к вам узел CPAN, получив файл */pub/languages/perl/CPAN/MIRRORS* с этого узла. Можно также обратиться с помощью Web-браузера к сервису мультимплексирования CPAN по адресу *www.perl.com*. Если вы запросите с этого Web-сервера файл, имя которого начинается на */CPAN/*, он соединит вас с ближайшим узлом CPAN, выбрав его по вашему доменному имени. Вот некоторые популярные адреса (URL) в CPAN:

```
http://www.perl.com/CPAN/  
http://www.perl.com/CPAN/README.html  
http://www.perl.com/CPAN/modules  
http://www.perl.com/CPAN/ports  
http://www.perl.com/CPAN/doc  
http://www.perl.com/CPAN/src/latest.tar.gz
```


Сервис мультимплексирования CPAN пробует соединить вас с локальной быстродействующей машиной через высокопроизводительный концентратор. Это, однако, получается не всегда, потому что доменные имена могут и не отражать схемы сетевых соединений. Например, ваше хост-имя может заканчиваться на *.se*, но при этом вам лучше будет соединяться не со Швецией, а с Северной Америкой. В этом случае вы можете самостоятельно выбрать узел по следующему URL:

<http://www.perl.com/CPAN>

Обратите внимание на отсутствие косой черты в конце этого URL. Если конечная косая опущена, мультимплексор CPAN выдает меню дублирующих серверов CPAN, из которых вы можете выбрать подходящий. Если ваш Web-браузер поддерживает встроенные переменные *cookies*, то мультимплексор CPAN автоматически запомнит выбранный вами узел.

Ниже перечислены машины, на которых должен быть исходный код Perl и копии списка дублирующих серверов CPAN. Эти материалы можно получить по анонимному FTP. (Попробуйте использовать не IP-адреса, а имена машин, поскольку IP-адреса могут измениться.)

<i>ftp.perl.com</i>	199.45.129.30
<i>ftp.cs.colorado.edu</i>	131.211.80.17
<i>ftp.funet.fi</i>	128.214.248.6
<i>ftp.cs.ruu.nl</i>	131.211.80.17

Местонахождение главного каталога зеркального сервера CPAN на этих машинах может быть разным, но скорее всего это нечто вроде */pub/perl/CPAN*.

Где находятся файлы

В главном каталоге CPAN вы увидите как минимум следующие подкаталоги:

authors

Этот каталог содержит многочисленные подкаталоги, по одному для каждого автора программного обеспечения. Например, если вы захотите найти знаменитый модуль CGI.pm Линкольна Штейна, будучи твердо уверены, что именно он его автор, то можете посмотреть в каталоге *authors/Lincoln_Stein*. Если бы вы не знали, что Штейн — автор этого модуля, то можно было бы посмотреть в каталоге модулей, который описывается ниже.

doc

Каталог, содержащий всевозможную документацию на Perl. Это вся официальная документация (man-страницы) в нескольких форматах (текстовый ASCII, HTML, PostScript и собственный формат Perl POD), а также сборники часто задаваемых вопросов и интересные дополнительные документы.

modules

Каталог содержит отдельные модули, написанные на C, Perl или обоих этих языках. Расширения позволяют вам эмулировать и использовать функциональные возможности других программных продуктов, например, графических средств Tk, UNIX-библиотеки curses и математических библиотек. Они позволяют также взаимодействовать с базами данных (Oracle, Sybase и др.) и создавать HTML-файлы и CGI-сценарии.

ports

Каталог содержит исходный код и (или) двоичные файлы для Perl-портов к операционным системам, не поддерживаемых непосредственно в стандартном дистрибутиве. Эти порты — результат усилий других авторов, и не все они могут функционировать так, как описано в нашей книге.

scripts

Набор разнообразных сценариев, собранных со всего мира. Если вам нужно узнать, как сделать что-либо, или если вы просто хотите посмотреть, как другие пишут программы, просмотрите этот каталог. Подкаталог *nutshell* содержит примеры, приведенные в нашей книге. (Эти тексты можно также получить на узле издательства O'Reilly & Associates, *ftp.ora.com*, в каталоге */published/oreilly/nutshell/learning_perl2/*.

src

В этом каталоге вы найдете исходный код стандартного дистрибутива Perl. Его текущая редакция всегда находится в файле *src/latest.tar.gz**. Этот большой файл содержит весь исходный код и полную документацию. Конфигурирование и инсталляция должны быть относительно простыми как в UNIX- и UNIX-подобных системах, так и в VMS и OS/2. Начиная с версии 5.004, Perl инсталлируется также в 32-разрядных Windows-системах.

Использование анонимного FTP

Если вам никогда не приходилось пользоваться анонимным FTP, разберите приведенный ниже пример сеанса с комментариями. Текст, набранный жирным шрифтом — это то, что вы должны вводить с клавиатуры; комментарии набраны курсивом. Символ % — это приглашение, которое вводить не нужно.

```
% ftp ftp.CPAN.org      (на самом деле такого узла нет)
Connected to ftp.CPAN.org
220 CPAN FTP server (Version wu-2.4(1) Fri Dec 1 00:00:00 EST 1995)
ready.
Name (ftp.CPAN.org:CPAN): anonymous
331 Guest login ok, send your complete e-mail address as password.
Password: camel@nutshell.com (здесь введите свое пользовательское имя и имя своего хоста)
230 Guest login ok, access restrictions apply.
ftp> cd pub/perl/CPAN/src
250 CWD command successful.
```

* Суффикс *.tar.gz* означает, что это стандартный Internet-формат архива, созданного программой *tar*.

```
ftp> binary      (для сжатых файлов нужно задать двоичный режим передачи)
ftp> get latest.tar.gz
200 PORT command successful.
150 Opening BINARY mode data connection for FILE.
226 Transfer complete.
.
.  (повторите этот шаг для каждого из нужных вам файлов)
.
ftp> quit
221 Goodbye.
%
```

Получив файлы, распакуйте их, а затем сконфигурируйте, постройте и установите Perl:

```
% gunzip < latest.tar.gz | tar xvf -
% cd perl5.004 (используйте реальное имя каталога)
Теперь любую из следующих двух строк:
% sh configure (строчная буква с — для автоматического конфигурирования)
% sh Configure (прописная буква С — для конфигурирования вручную)
% make (построить весь Perl)
% make test (проверить, работает ли он)
% make install (для этого вы должны быть привилегированным пользователем)
```

Как выбрать модули

Процесс выборки и построения отдельных модулей Perl протекает немного по-другому. Скажем, вы хотите построить и установить модуль CoolMod. Сначала нужно выбрать его, воспользовавшись для этого командой *fip(1)* или с помощью Web-браузера обратиться к сервису модулей из <http://www.perl.com/>, который всегда выбирает самую последнюю версию конкретного зарегистрированного модуля. Адрес, который нужно ввести в браузер, выглядит так:

```
http://www.perl.com/cgi-bin/cpan\_mod?module=CoolMod
```

Получив этот файл, сделайте следующее:

```
% gunzip < CoolMod-2.34.tar.gz | tar xvf -
% cd CoolMod-2.34
% perl Makefile.PL (создает реальный Makefile)
% make (построить весь модуль)
% make test (проверить, работает ли он)
% make install (для этого вы, наверное, должны быть привилегированным пользователем)
```

После успешной установки модуля CoolMod (он автоматически помещается в директорию Perl-библиотеки вашей системы) в ваших программах можно использовать

```
use CoolMod;
```

а вы сможете читать документацию этого модуля с помощью команды *man CoolMod* (или *perldoc CoolMod*).

Обозначения, принятые в книге

В нашей книге используются следующие обозначения:

Курсив

используется для имен файлов и команд. Курсивом также выделяются термины при первом употреблении.

Моноширинный шрифт

используется в примерах и обычном тексте для выделения операций, переменных и результатов работы команд и программ.

Моноширинный жирный

используется в примерах для выделения данных, которые вводятся пользователем с терминала.

Моноширинный курсив

используется в примерах для выделения переменных, вместо которых в зависимости от контекста нужно подставлять значения. Например, переменную *имя_файла* необходимо заменить именем реального файла.

Сноски

используются для ввода дополнительных примечаний. Читая книгу в первый раз, *не обращайтесь на них внимания*. Иногда для упрощения изложения материала в основном тексте говорится не вся правда, а в сноске приводятся необходимые уточнения. Во многих случаях материал, данный в сноске, представляет собой более сложную информацию, которая в книге вообще может не рассматриваться.

Поддержка

Perl — это детище Ларри Уолла, и он все еще продолжает с ним нянчиться. Сообщения о дефектах и требования всевозможных улучшений, как правило, учитываются в следующих редакциях, но Ларри вовсе не обязан делать это. Тем не менее ему доставляет удовольствие получать отзывы от всех нас и осознавать, что Perl полезен всему миру. На прямую электронную почту обычно даются ответы (пусть даже и его автоответчиком), причем иногда персональные. Сейчас Ларри выступает в роли архитектора группы Perl 5 Porters — плеяды очень умных людей, внесший гигантский вклад в создание нескольких последних версий языка Perl. Если бы Ларри попал под автобус, то все очень долго грустили бы, но под руководством этой группы Perl все равно продолжал бы развиваться.

Если вы нашли какой-то дефект, можете воспользоваться Perl-программой *perlbug*, которая обеспечивает сбор соответствующей информации и отправляет ее по электронной почте на узел *perlbug@perl.com*. Члены группы Perl 5 Porters читают эту почту (наряду с теми 20-100 сообщениями, которые

они ежедневно посылают друг другу) и иногда отвечают, если это действительно дефект. Но стоит вам воспользоваться этим адресом просто для того, чтобы выразить благодарность, как вам выскажут недовольство вашим поведением, поэтому сведите светские разговоры к абсолютному минимуму и воздержитесь от вызова актеров на бис.

Вместо того чтобы писать непосредственно Ларри или посылать сообщение о дефекте, гораздо полезнее воспользоваться услугами диалоговой службы Perl-поддержки, которые предоставляются через Usenet-телеконференцию *comp.lang.perl.misc*. Если вы имеете возможность посылать и получать электронную почту по Internet, но к Usenet доступа не имеете, можете подключиться к этой телеконференции, послав запрос по адресу *perl-users-request@cs.orst.edu*; этот запрос попадет к человеку, который сможет соединить вас с двунаправленным шлюзом электронной почты этой телеконференции и сообщить правила работы в ней.

Подписавшись на эту телеконференцию, вы ежедневно будете обнаруживать от 50 до 200 статей на всевозможные темы — от вопросов новичков до сложных аспектов переносимости и проблем сопряжения. Иногда там будут попадаться и довольно большие программы.

Эта телеконференция почти постоянно просматривается многими специалистами по языку Perl. В большинстве случаев ответ на ваш вопрос дается спустя считанные минуты после попадания вопроса на один из основных концентраторов Usenet. Вы только попробуйте получить поддержку на таком уровне, да еще и бесплатно, у своего любимого поставщика программных продуктов! Если же вы хотите заключить контракт на коммерческую поддержку, обратитесь к сборнику часто задаваемых вопросов по Perl.

Благодарности: первое издание

Во-первых, я от всего сердца благодарю Чика Уэбба и фирму Taos Mountain Software (Кремниевая долина). Ребята из TMS предоставили мне возможность написать для них (при значительном содействии Чика) вводный курс по языку Perl и прочитав этот курс несколько раз. Этот опыт дал мне мотивы и ресурсы для написания и многократного прочтения нового, моего собственного курса, на базе которого и построена эта книга. Не думаю, что без содействия сотрудников TMS я занимался бы этим, и желаю им всем больших успехов в маркетинге их курса. (А если они ищут хороший текст для переработки своего курса, то у меня как раз есть одно предложение...)

Спасибо моим рецензентам: “крестному отцу” языка Perl Ларри Уоллу (естественно), Ларри Кистлеру (руководителю службы подготовки кадров фирмы Pyramid), моему коллеге по преподаванию Perl Тому Кристиансену и слушателям курсов по Perl из фирм Intel и Pyramid, а также сотрудникам издательства O'Reilly & Associates Тане Херлик, Лару Кауфману, Ленни Мюльнеру, Линде Мюи и Энди Ораму.

Эта книга была полностью написана и отредактирована на моем персональном компьютере Apple Macintosh Powerbook (сначала модели 140, теперь 160). В процессе работы я чаще всего находился вне своего кабинета — иногда в парке, иногда в гостинице, иногда в горах, ожидая хорошей погоды для лыжных прогулок, но чаще всего в ресторанах. По сути дела, значительную часть книги я написал в пивном баре Beaverton McMamin's недалеко от дома. В пабах сети McM's варят и подают самое вкусное пиво, самый лучший творожный пудинг и самые жирные сэндвичи в округе. В этой идеальной для работы обстановке я выпил множество пинт пива и съел гору пудинга, а мой Powerbook поглотил массу киловатт-часов электроэнергии, устраиваясь на тех четырех столах, где есть розетки. Благодарю замечательный персонал бара McM's за эту электроэнергию, а также за гостеприимство, радушие и любезность (и бесплатное место для работы). Кроме того, благодарю ресторан Beaverton Chili's, где я начал работать над этой книгой. (Но у них возле стойки не оказалось розеток, поэтому когда я нашел McM's, то перешел туда, чтобы сберечь аккумуляторы.)

Спасибо всем пользователям Internet (особенно подписчикам телеконференции *comp.lang.perl*) за их постоянную поддержку, оказываемую Ларри и мне, и за бесконечные вопросы на тему о том, как заставить Perl работать.

Благодарю также Тима О'Рейли — за даосизм.

Наконец, особая, огромная персональная благодарность — моему другу Стиву Тэлботту, который направлял меня на каждом этапе этого пути (в частности, он предложил сделать “прогулку” по стране Perl, впечатления от которой представлены в конце первой главы). Его редакторская критика всегда была справедливой, а его удивительная способность бить меня по голове с исключительной нежностью позволила мне сделать мою книгу произведением искусства, которым я крайне доволен.

Как всегда, выражаю особую благодарность Лайлу и Джеку за то, что они научили меня почти всему тому, что я знаю о писательском ремесле.

И, наконец, безмерная благодарность — моему другу и партнеру Ларри Уоллу за то, что он дал всем нам Perl.

Одна “Л” — Рэндал — книгу написала,
Вторая — лама — на обложку прискакала.
Но кто все это изобрел?
То целых три “Л” — Ларри Уолл!

Рэндал

Благодарности: второе издание

Я хотел бы поблагодарить Ларри Уолла за создание Perl, членов группы Perl Porters за их постоянные усилия по сопровождению языка и все Perl-сообщество за готовность помогать друг другу.

Спасибо также Джону Орунту, Нэйт Торкингтону и Ларри Уоллу за рецензирование главы, посвященной CGI.

Том

Пожалуйста, пишите нам

Комментарии и вопросы по этой книге направляйте, пожалуйста, в издательство по адресу:

O'Reilly & Associates

101 Morris Street

Sebastopol, CA 95472

тел. 1-800-998-9938 (в США и Канаде)

тел. 1-707-829-0515 (международный или местный)

факс 1-707-829-0104

Сообщения можно посылать и в электронной форме. Чтобы попасть в список рассылки или запросить каталог, пошлите сообщение по адресу *nuts@ora.com*.

Сообщения с техническими вопросами и комментариями по книге направляйте по адресу *bookquestions@ora.com*.

В этой главе:

- История создания языка Perl
- Назначение языка Perl
- Доступность
- Основные понятия
- Прогулка по стране Perl
- Упразднение

1

Введение

История создания языка Perl

Слово Perl является аббревиатурой выражения Practical Extraction and Report Language (практический язык извлечений и отчетов), хотя иногда его называют Pathologically Eclectic Rubbish Lister (патологически эклектичный мусорный листер). Не стоит спорить о том, какое из этих названий более правильное, потому что оба они принадлежат Ларри Уоллу, создателю и главному архитектору, распространителю и опекуну языка Perl. Ларри создал этот язык, когда пытался формировать отчеты из иерархии файлов системы оповещения об ошибках, похожей на Usenet-новости, а возможности применявшегося в то время обработчика потоков данных *awk* оказались исчерпанными. Будучи настоящим (то есть ленивым) программистом, Ларри решил вырвать данную проблему с корнем, применив для этого какой-нибудь универсальный инструмент, который он надеялся использовать и в дальнейшем. В результате появилась первая версия языка Perl.

Позабавившись немного с этой версией, добавив кое-что, Ларри предложил ее сообществу читателей материалов телеконференций Usenet, известному также как “Сеть” (the Net). Пользователи, имеющие доступ к входящим в систему Usenet компьютерам, разбросанным по всему свету (а их в то время было несколько десятков тысяч), обеспечили для создателя Perl эффективную “обратную связь”, спрашивая, как делать одно, другое, третье. Многие из этих задач Ларри даже и не собирался ставить перед своим маленьким новым языком программирования.

В результате Perl все рос и рос, причем почти с той же скоростью, что и операционная система UNIX. (Специально для новичков: все ядро UNIX тогда требовало памяти объемом 32 К! Теперь мы счастливы, если нам удастся уместить его в несколько мегабайтов.) Выросли и его возможности.

Повысилась переносимость. То, что когда-то было компактным языком, теперь сопровождается сотнями страниц документации, в состав которой входят десятки map-страниц, 600-страничный справочник серии Nutshell, материалы множества телеконференций Usenet с 200000 подписчиков, — а теперь еще и эта скромная книга.

Ларри уже не сопровождает Perl в одиночку, но сохраняет свой эксклюзивный титул главного архитектора. А Perl все растет и растет.

Примеры программ, содержащихся в этой книге, мы проверяли на Perl версии 5.004 (на момент написания книги это был самый последний выпуск). Все программы, которые здесь рассматриваются, должны работать с Perl версии 5.0 и со всеми последующими версиями. По сути дела, даже программы, написанные на языке Perl версии 1.0, довольно неплохо работают с последними версиями, если только не сталкиваются с некоторыми эксцентричными нововведениями, сделанными во имя прогресса.

Назначение языка Perl

Назначение языка Perl — помочь программисту в выполнении рутинных задач, которые для shell слишком трудны или плохо переносимы, а также чересчур заумны, одноразовы или сложны для кодирования на C или ином используемом в UNIX языке.

Научившись пользоваться языком Perl, вы, возможно, обнаружите, что начинаете тратить меньше времени на правильное заключение в кавычки различных параметров shell (или на корректное выполнение C-объявлений), а больше — на чтение Usenet-новостей и катание с гор на лыжах, потому что Perl — замечательное средство для вашего совершенствования как программиста. Мощные конструкции этого языка позволяют создавать (с минимальной затратой сил) некоторые очень эффективные специализированные решения и универсальные инструменты. Эти инструменты можно использовать и в дальнейшем, потому что написанные на Perl программы отличаются высокой переносимостью и готовностью к использованию. В результате у вас появится *еще больше* времени для чтения Usenet-новостей и посещения с друзьями баров караоке.

Как и любой язык, Perl может быть языком “только для написания” программ, которые потом будет невозможно прочитать. Однако при правильном подходе вы можете избежать этого весьма распространенного недостатка. Да, иногда Perl-текст выглядит для непосвященных как случайный набор символов, но умудренный опытом Perl-программист знает, что у этого набора есть контрольная сумма и каждый его символ имеет свое предназначение. Если вы будете следовать указаниям, приведенным в нашей книге, ваши программы будут легкими для чтения и простыми для сопровождения — но, вероятно, не выиграют никаких “состязаний по заумности”.

Доступность

Если при попытке вызвать Perl из shell вы получите сообщение

```
perl: not found
```

это значит, что вашего системного администратора еще не охватила Perl-лихорадка. Если язык Perl не установлен в вашей системе, его можно получить бесплатно (или почти бесплатно).

Perl распространяется по открытой лицензии GNU (GNU Public License)*, согласно которой “вы можете распространять двоичные файлы языка Perl только в том случае, если предоставляете исходный код бесплатно, а если вы модифицируете их, вы должны распространять и код этих изменений”. По сути дела это означает, что Perl распространяется бесплатно. Его исходный текст можно получить по цене пустой ленты или оплатив стоимость передачи нескольких мегабайтов информации по телефонной линии. При этом никто не может “придержать” сам Perl и послать вам просто двоичные файлы, соответствующие чьему-либо конкретному представлению о “поддерживаемых аппаратных платформах”.

По сути дела, Perl не только бесплатен, но и работает достаточно хорошо почти на всем, что называет себя UNIX или UNIX-подобной системой и включает C-компилятор. Это обусловлено тем, что пакет распространяется с адаптивной программой конфигурации, называемой *Configure*, которая рыщет и шарит по системным каталогам в поисках нужных ей вещей, соответствующим образом корректирует используемые файлы и определенные символы, обращаясь к вам за подтверждением результатов своих изысканий.

Программисты настолько увлеклись языком Perl, что, помимо UNIX- и UNIX-подобных систем, начали использовать его и в системах Amiga, Atari ST, системах семейства Macintosh, VMS, OS/2, даже MS-DOS и, наконец, в Windows NT и Windows 95. К тому моменту, когда вы будете читать эти строки, Perl, вероятно, перенесут и на многие другие системы. Исходные тексты языка Perl (и многие предкомпилированные двоичные файлы для не-UNIX-архитектур) можно получить на одном из серверов сети CPAN (Comprehensive Perl Archive Network). Если вы имеете доступ к World Wide Web, посетите сервер <http://www.perl.com/CPAN>, являющийся одним из множества “зеркальных” (дублирующих) серверов. Если вы абсолютный новичок, отправьте по адресу bookquestions@ora.com послание с вопросом “Где можно получить Perl?!?” (“Where I can get Perl?!?”).

* Или по несколько более либеральной “художественной лицензии” (Artistic License), согласно которой Perl распространяется в виде дистрибутива.

Основные понятия

Сценарий shell — это не что иное, как последовательность команд shell, оформленная в виде текстового файла. Этот файл затем превращается в исполняемый путем включения бита исполнения (посредством команды *chmod +x имя_файла*), после чего по приглашению shell вводится имя файла. Давайте рассмотрим какой-нибудь сценарий shell. Например, сценарий выполнения команды *date*, а затем команды *who* можно записать и выполнить так:

```
% echo date >somescript
% echo who >>somescript
% cat somescript
date
who
% chmod +x somescript
% somescript
[результат выполнения команды date, затем команды who]
%
```

Аналогичным образом Perl-программа — это набор Perl-операторов и определений, записанных в виде файла. Затем включается бит исполнения*, после чего по приглашению shell вводится имя файла. При этом, однако, файл должен иметь признак, указывающий, что это Perl-программа, а не программа shell.

В большинстве случаев операция введения такого признака заключается в помещении в начало файла строки

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

Если же ваш Perl инсталлирован в каком-то нестандартном каталоге или если ваша система “не понимает” строку, начинающуюся с символов *#!*, придется сделать кое-что еще. Узнайте об этом у того, кто инсталлировал вам Perl. В примерах, приведенных в книге, предполагается, что вы пользуетесь именно этим, общепринятым механизмом обозначения Perl-программ.

Perl — это, в основном, язык со свободным форматом записи программ (вроде C) — пробельные символы, включаемые между лексемами (элементами программы, например *print* или *+*), не обязательны, если две рядом стоящие лексемы невозможно принять за какую-то третью лексему. В последнем случае какой-нибудь пробельный символ является обязательным (К пробельным символам относятся пробелы, знаки табуляции, символы новой строки, символы возврата каретки, символы перехода на новую страницу.) Имеется также ряд конструкций, в которых требуется использовать определенный пробельный символ в определенном месте, но об этом мы расскажем, когда дойдем до их описания. Вы можете считать, что тип и

* Это справедливо для UNIX-систем. Указания о том, как сделать сценарий исполняемым в других системах, вы можете найти в сборнике ответов на часто задаваемые вопросы (FAQ) по Perl или в документации, приложенной к вашей операционной системе

количество пробельных символов между лексемами во всех прочих случаях могут быть произвольными.

Хотя почти каждую Perl-программу можно записать в одну строку, эти программы обычно пишут с отступами, как C-программы, причем вложенные операторы записывают с большим отступом, чем охватывающие. В этой книге вы увидите множество примеров, записанных с типичными для языка Perl отступами.

Аналогично сценарию shell, Perl-программа состоит из всех операторов Perl, имеющихся в файле и рассматриваемых в совокупности как одна большая программа, подлежащая выполнению. Понятия “основной” (main) программы, как в C, здесь нет.

Комментарии в Perl похожи на комментарии shell (современные). Комментарием является все, что следует за незаключенным в кавычки знаком # вплоть до конца строки. Многострочных комментариев, как в C, здесь нет.

В отличие от большинства shell (но аналогично *awk* и *sed*), интерпретатор языка Perl перед выполнением программы полностью разбирает ее и компилирует в свой внутренний формат. Это значит, что после запуска программы вы никогда не получите сообщение о синтаксической ошибке и что пробельные символы и комментарии не замедляют ход выполнения программы. Такой метод обеспечивает быстрое выполнение операций языка Perl после запуска и является дополнительным стимулом к отказу от использования C в качестве служебного языка систем лишь на том основании, что C — транслируемый язык.

Но процедура компиляции все же требует времени, и применение большой Perl-программы, которая быстро выполняет одну маленькую задачу (из множества тех, которые она способна выполнить), а затем заканчивает свою работу, не будет эффективным, ибо время ее выполнения окажется ничтожно малым по сравнению со временем компиляции.

Perl, таким образом, работает и как компилятор, и как интерпретатор. С одной стороны, это компилятор, потому что перед выполнением первого оператора программы она полностью считывается и разбирается. С другой стороны, Perl — интерпретатор, потому что никакого объектного кода, занимающего место на диске в ожидании исполнения, в данном случае нет. Другими словами, он сочетает в себе лучшее из компилятора и интерпретатора. Конечно же, было бы просто здорово, если бы выполнялось какое-то кэширование скомпилированного объектного кода между вызовами, а то и его трансляция в “родной” машинный код. Рабочая версия такого компилятора фактически уже существует, и сейчас планируется, что она войдет в выпуск 5.005. О текущем состоянии дел можно узнать в сборнике FAQ, посвященном Perl.

Прогулка по стране Perl

Наше путешествие по стране Perl мы начнем с небольшой прогулки. В ходе этой прогулки мы ознакомимся с некоторыми возможностями языка Perl на примере небольшого приложения. Здесь приведены лишь очень краткие пояснения; каждая тема *гораздо* подробнее освещается в соответствующей главе. Тем не менее эта короткая прогулка должна дать вам возможность быстро “почувствовать” этот язык, и вы сможете решить, будете ли вы дочитывать книгу до конца или же отправитесь обратно к своим Usenet-новостям и лыжным склонам.

Программа Hello, World

Давайте рассмотрим небольшую программу, которая делает что-то реальное. Вот ваша базовая программа, которая выводит на экран слова “Hello, World”:

```
#!/usr/bin/perl -w
print ("Hello, World\n");
```

Первая строка говорит о том, что это программа написана на языке Perl. Кроме того, первая строка является комментарием; ведь комментарием в Perl, как и во многих интерпретирующих языках программирования, являются все лексемы, стоящие после знака # и до конца текущей строки. Но, в отличие от всех остальных комментариев, включенных в эту программу, комментарий в первой строке особенный: Perl ищет здесь необязательные аргументы. В данном случае использовался ключ `-w`. Этот очень важный ключ дает Perl указание выдавать дополнительные предупреждающие сообщения о *потенциально опасных* конструкциях. Вам следует всегда использовать в своих программах ключ `-w`.

Вторая строка — это выполняемая часть данной программы. Здесь мы видим функцию `print`. В данном случае встроенная функция `print` имеет всего один аргумент, C-подобную текстовую строку. В этой строке комбинация символов `\n` обозначает символ новой строки. Оператор `print` завершается точкой с запятой (;). Как и в C, все простые операторы в Perl завершаются точкой с запятой*.

Когда вы вызываете эту программу, ядро запускает интерпретатор Perl, который разбирает всю программу (обе строки, включая первую, т.е. комментарий), а затем выполняет скомпилированный вариант. Первая и единственная операция — выполнение функции `print`, которая посылает значения своих аргументов на устройство вывода. По окончании выполнения программы этот Perl-процесс завершается и возвращает родительскому shell код успешного выполнения.

* Точку с запятой можно опустить, если оператор является последним оператором в блоке или файле либо оператором `eval`.

Скоро вы увидите Perl-программы, в которых `print` и другие функции иногда вызываются с круглыми скобками, а иногда — без них. Правило здесь простое: круглые скобки для встроенных функций Perl не являются ни обязательными, ни запрещенными. Их применение может прояснить ситуацию, а может и запутать ее, так что выработайте собственный стиль.

Как задавать вопросы и запоминать результат

Давайте немного усложним пример, ведь приветствие `Hello, World` — слишком простое и статичное. Сделаем так, чтобы программа называла вас по имени. Для этого нам нужно место для хранения имени, способ задания вопроса об имени и способ получения ответа.

Одно из мест для хранения значений (вроде имени) — *скалярная переменная*. Для хранения вашего имени в нашей программе мы используем скалярную переменную `$name`. В главе 2, “Скалярные данные”, мы более подробно узнаем о том, что можно хранить в этих переменных и что можно с ними делать. Пока же предположим, что мы можем хранить в скалярной переменной только одно число или строку (последовательность символов).

Программа должна иметь возможность спросить у вас имя. Для этого нам нужен способ выдачи приглашения и способ принятия от вас данных. Предыдущая программа показала нам, как можно приглашать, используя для этого функцию `print`. Получение же строки с терминала осуществляется с помощью конструкции `<STDIN>`, которая (в нашем случае) получает одну строку введенных данных. Введенные данные мы присваиваем переменной `$name`. Это дает нам следующую программу:

```
print "What is your name? "; # Как ваше имя?  
$name = <STDIN>;
```

Значение переменной `$name` пока содержит завершающий символ новой строки (имя `Randal` поступает как `Randal\n`). Чтобы избавиться от этого символа, мы воспользуемся функцией `chomp`, которая в качестве своего единственного аргумента принимает скалярную переменную и удаляет из ее строкового значения завершающий символ перехода на новую строку (пробельный символ), если он присутствует:

```
chomp ($name);
```

Теперь нам нужно лишь сказать `Hello` и указать значение переменной `$name`, что мы можем сделать так, как в `shell`, поместив эту переменную в заключенную в кавычки строку:

```
print "Hello, $name!\n";
```

Как и в `shell`, если нам нужен именно знак доллара, а не ссылка на скалярную переменную, мы можем предварить этот знак обратной косой чертой.

Сложив все вместе, получаем:

```
#!/usr/bin/perl -w
print "What is your name? ";
$name = <STDIN>;
chomp ($name);
print "Hello, $name'\n";
```

Добавляем возможность выбора

Допустим теперь, что у нас припасено какое-то особое приветствие для пользователя по имени Рэндал, а для остальных — обычное. Для этого нам нужно сравнить имя, которое было введено, со строкой Randal, и, если оно совпадает, сделать что-то особое. Давайте добавим в программу С-подобную ветвь *if-then-else* и операцию сравнения:

```
#!/usr/bin/perl
print "What is your name? ";
$name = <STDIN>;
chomp ($name);
if ($name eq "Randal") {
    print "Hello, Randal' How good of you to be here'\n";
} else {
    print "Hello, $name'\n"; # обычное приветствие
}
```

В операции `eq` сравниваются две строки. Если они равны (т.е. совпадают все символы и длина строк одинакова), результатом будет “истина”. (В С и С++ подобной операции нет*).

Оператор `if` выбирает, какой блок операторов (заклученных между парными фигурными скобками) выполняется; если выражение дает в результате значение “истина”, выполняется первый блок, в противном случае выполняется второй блок.

Секретное слово

Теперь, когда у нас есть имя, давайте сделаем так, чтобы человек, выполняющий эту программу, угадывал секретное слово. У всех, кроме Рэндала, программа будет непрерывно требовать ввести это слово, пока пользователь не наберет слово правильно. Сначала мы приведем текст программы, потом дадим пояснение к ней:

```
#!/usr/bin/perl -w
$secretword = "llama"; # секретное слово
print "What is your name? ";
$name = <STDIN>;
chomp $name;
if ($name eq "Randal") {
    print "Hello, Randal' How good of you to be here'\n";
```

* Для получения аналогичного результата можно использовать стандартную подпрограмму `libc`. Но это не операция

```

} else {
    print "Hello, $name! \n"; # обычное приветствие
    print "What is the secret word? ";
    $guess = <STDIN>;
    chomp ($guess) ;
    while ($guess ne $secretword) {
        print "Wrong, try again. What is the secret word? ";
        $guess = <STDIN>;
        chomp ($guess);
    }
}

```

Сначала мы задаем секретное слово, помещая его в скалярную переменную `$secretword`. После приветствия программа спрашивает (посредством вызова еще одной функции `print`) у пользователя (не Рэндала) его вариант секретного слова. Этот вариант сравнивается с заданным секретным словом в операции `ne`. Данная операция возвращает значение “истина”, если сравниваемые строки не равны (т.е. данная операция противоположна операции `eq`). Результат сравнения управляет циклом `while`, который выполняет этот блок операторов до тех пор, пока сравнение дает значение “истина”.

Конечно, эта программа плохо защищена, потому что любой, кому надоест угадывать секретное слово, может просто прервать ее выполнение и вернуться к приглашению, а то и подсмотреть секретное слово в исходном тексте. Мы, однако, не пытались разработать систему обеспечения безопасности, а лишь хотели привести подходящий для данного раздела пример.

Несколько секретных слов

Давайте посмотрим, как можно модифицировать эту программу так, чтобы она принимала несколько секретных слов. Используя то, что мы уже видели, можно было бы многократно сравнивать вариант-догадку с рядом правильных ответов, хранящихся в отдельных скалярных переменных. Такой список, однако, было бы трудно корректировать или модифицировать в зависимости от дня недели и даты.

Более эффективное решение — хранить все возможные ответы в структуре данных, которая называется *список*, или (предпочтительнее) *массив*. Каждый *элемент* массива — это отдельная скалярная переменная, которой можно присваивать значение и затем использовать ее независимо от других. Можно также одним махом присвоить значение всему массиву. Мы имеем право присвоить значение всему массиву с именем `@words` так, чтобы он содержал три возможных правильных пароля:

```
@words = ("camel", "llama", "alpaca");
```

Имена переменных-массивов начинаются с символа `@`, что позволяет отличать их от имен скалярных переменных. Существует еще один способ записи этой конструкции так, чтобы не нужно было ставить все эти кавычки — с помощью операции `qw()`, например:

```
@words = qw(camel llama alpaca);
```


Это абсолютно то же самое; операция `qw` работает так, как будто мы взяли в кавычки каждую из трех строк.

Присвоив значения элементам массива, мы можем обращаться к каждому из них по индексной ссылке. Так, `$words[0]` — это `camel`, `$words[1]` — `llama`, а `$words[2]` — `alpaca`. Индекс может быть и выражением, поэтому если мы присвоим `$i` значение 2, то элементом `$words[$i]` будет `alpaca`. (Индексные ссылки начинаются с символа `$`, а не с `@`, потому что они обозначают один элемент массива, а не весь массив.) Вернемся к нашему предыдущему примеру:

```
#!/usr/bin/perl -w
@words = qw(camel llama alpaca);
print "What is your name? " ;
$name = <STDIN>;
chomp ($name);
if ($name eq "Randal") {
    print "Hello, Randal! How good of you to be here!\n";
} else {
    print "Hello, $name!\n";          # обычное приветствие
    print "What is the secret word? ";
    $guess = <STDIN>;
    chomp ($guess) ;
    $i = 0; # сначала попробуем это слово
    $correct = "maybe";              # догадка верна или нет?
    while ($correct eq "maybe") {    # продолжаем проверку
        if ($words[$i] eq $guess) {   # верно?
            $correct = "yes";          # да!
        } elsif ($i < 2) {             # посмотреть еще слова?
            $i = $i + 1;               # в следующий раз посмотреть следующее слово
        } else {                      # больше слов нет, должно быть, неверно
            print "Wrong, try again. What is the secret word?";
            $guess = <STDIN>;
            chomp ($guess);
            $i = 0;                    # вновь начать проверку с первого слова
        }
    } # конец цикла while для неверных слов
} # конец цикла "не Рэндал"
```

Заметьте, что мы используем скалярную переменную `$correct` для того, чтобы показать, все еще ищем мы правильный пароль или уже нашли его.

В этой программе показан также блок `elsif` оператора `if-then-else`. Такой конструкции нет ни в одном другом языке программирования; это сокращенная запись блока `else` с новым условием `if`, но без вложения еще одной пары фигурных скобок. Сравнение набора условий в каскадной цепочке `if-elsif-elsif-elsif-else` очень характерно для языка Perl. В нем нет эквивалента оператору `switch` языка C или оператору `case` языка Паскаль, но вы можете сами без особых хлопот создать такой оператор. Подробности см. в главе 2 книги *Programming Perl* и на странице руководства `perlsyn(1)`.

Разные секретные слова для разных пользователей

В предыдущем случае любой пользователь мог угадать одно из трех секретных слов и получить доступ к программе. Если мы хотим, чтобы для каждого пользователя было задано свое секретное слово, нам нужна примерно такая таблица соответствий:

Пользователь	Секретное слово
Fred	camel
Barney	llama
Betty	alpaca
Wilma	alpaca

Обратите внимание: у последних двух пользователей одинаковые секретные слова. Такое допускается.

Самый простой способ сохранить такую таблицу — использовать *хеш*. В каждом элементе хеша содержится отдельное скалярное значение (как и в массиве любого другого типа), но в соответствие каждому элементу хеша ставится *ключ*. Ключом может быть любое скалярное значение (любая строка или число, в том числе нецелые и отрицательные числа). Чтобы создать хеш под именем `%words` (обратите внимание на то, что используется символ `%` вместо `@`) с ключами и значениями, данными в приведенной выше таблице, мы присвоим `%words` значение (почти так же, как мы делали раньше с массивом):

```
%words = qw(  
    fred    camel  
    barney  llama  
    betty   alpaca  
    wilma   alpaca  
);
```

Каждая пара в этом списке представляет в хеше один ключ и соответствующее ему значение. Обратите внимание на то, что мы разбили эту процедуру присваивания на несколько строк без каких-либо символов продолжения строк, потому что пробельные символы в Perl-программах обычно никакой роли не играют.

Чтобы найти секретное слово для Betty, мы должны использовать имя Betty как ключ в ссылке на хеш `%words` с помощью выражения вроде `$words{"betty"}`. Значение этой ссылки — `alpaca`, это похоже на то, что мы видели раньше, работая с другим массивом. Как и раньше, ключом может быть любое выражение, поэтому установка `$person` в значение `betty` и вычисление `$words{$person}` также дает в результате `alpaca`.

Сведя все это воедино, мы получаем такую программу:

```
#!/usr/bin/perl -w
%words = qw(
    fred      camel
    barney    llama
    betty     alpaca
    wilma     alpaca
);
print "What is your name? ";
$name = <STDIN>;
chomp ($name) ;
if ($name eq "Randal") {
    print "Hello, Randal! How good of you to be here!\n";
} else {
    print "Hello, $name!\n";      # обычное приветствие
    $secretword = $words{$name}; # получить секретное слово
    print "What is the secret word? ";
    $guess = <STDIN>;
    chomp ($guess) ;
    while ($guess ne $secretword) {
        print "Wrong, try again. What is the secret word? ";
        $guess = <STDIN>;
        chomp ($guess) ;
    }
}
}
```

Обратите внимание на то, как происходит поиск секретного слова. Если имя не найдено, то значением переменной `$secretword` будет пустая строка*, и мы можем использовать оператор `if`, чтобы задать секретное слово по умолчанию для кого-нибудь еще. Вот как это выглядит:

```
[... остальная часть программы удалена ...]
$secretword = $words{$name}; # получить секретное слово
if ($secretword eq "") {     # не найдено
    $secretword = "groucho"; # конечно, можно использовать
}
print "What is the secret word? ";
[... остальная часть программы удалена ...]
```

Обработка различных вариантов ввода секретного слова

Если вместо `Randal` вы введете `Randal L. Schwartz` или `randal`, то тем самым лишите Рэндала права на особое приветствие, потому что сравнение `eq` предполагает точное равенство. Давайте рассмотрим один способ решения задачи обработки различных вариантов ввода.

* На самом деле это значение `undef`, но для операции `eq` оно выглядит как пустая строка. Если бы в командной строке вы использовали ключ `-w`, то получили бы предупреждение на этот счет. Именно поэтому мы его здесь опустили.

Допустим, вы хотите найти все строки, которые начинаются со слова Randal, а не просто строку, равную Randal. В *sed*, *awk* или *grep* это можно сделать с помощью регулярного выражения — шаблона, определяющего совокупность соответствующих строк. Как и в *sed*, *awk* или *grep*, в Perl регулярным выражением, которое соответствует любой строке, начинающейся со слова Randal, будет `^Randal`. Чтобы сравнить его со строкой, содержащейся в скалярной переменной `$name`, мы используем операцию сопоставления:

```
if ($name =~ /^Randal/) {
    ## да, совпадает
} else {
    ## нет, не совпадает
}
```

Обратите внимание на то, что регулярное выражение выделяется косой чертой с обеих сторон. Пробелы и другие пробельные символы, заключенные между косыми, имеют значение, поскольку они являются частью строки.

Это почти решает нашу задачу, но не позволяет выбрать `randal` или отклонить `Randall`. Чтобы принять `randal`, мы добавляем опцию *игнорирования регистра* — прописную букву `i` после закрывающей косой. Чтобы отклонить `Randall`, мы вводим в регулярное выражение специальный маркер *границы слова* (подобно тому как это делается в *vi* и в некоторых версиях *grep*) в форме `\b`. Это гарантирует, что символ, следующий в регулярном выражении за первой буквой `l`, не является еще одной буквой. В результате наше регулярное выражение принимает вид `/^randal\b/i`, что означает «слово `randal`, стоящее в начале строки, за которым нет ни буквы, ни цифры, при этом регистр не имеет значения».

Объединив этот фрагмент с остальной частью программы, получим:

```
#!/usr/bin/perl
%words = qw(
    fred      camel
    barney     llama
    betty      alpaca
    wilma      alpaca
);
print "What is your name? ";
$name = <STDIN>;
chomp ($name);
if ($name =~ /^randal\b/i) {
    print "Hello, Randal! How good of you to be here!\n";
} else {
    print "Hello, $name! \n";      # обычное приветствие
    $secretword = $words{$name}; # получить секретное слово
    if ($secretword eq "") {      # не найдено
        $secretword = "groucho"; # конечно, можно использовать
    }
    print "What is the secret word? ";
    $guess = <STDIN>;
```

```

chomp ($guess);
while ($guess ne $secretword) {
    print "Wrong, try again. What is the secret word? ";
    $guess = <STDIN> ;
    chomp ($guess) ;
}
}

```

Как видите, эта программа уже довольно далека от простенькой Hello, World. Хотя она и очень мала, но вполне работоспособна, причем краткость программы достигается весьма небольшими усилиями. В этом — стиль Perl

В Perl имеется все, что необходимо для работы с регулярными выражениями, т.е. он предоставляет все возможности, которые обеспечивает любая стандартная утилита UNIX (и даже некоторые нестандартные). Способ сопоставления строк, используемый в Perl, является чуть ли не самым быстрым сравнительно с другими языками, поэтому производительность системы при выполнении Perl-программ никоим образом не снижается. (Написанная на Perl grep-подобная программа часто превосходит прилагаемую поставщиками программу *grep* на C*. Это значит, что *grep* не выполняет толком даже единственную свою задачу.)

Справедливость для всех

Итак, теперь я могу ввести Randal, randal или Randal L. Schwartz, но как быть с остальными? Барни должен вводить в точности barney (ему нельзя ввести даже пробел после barney).

Чтобы быть справедливыми по отношению к Барни, мы должны *перед* поиском имени в таблице взять первое слово из того, что введено, а затем заменить все его символы символами нижнего регистра. Это делается с помощью двух операций — операции *подстановки*, которая находит регулярное выражение и заменяет его строкой, и операции *перевода*, которая переводит символы этой строки в нижний регистр.

Сначала — операция подстановки: мы хотим взять содержимое переменной \$name, найти первый специальный (не использующийся в словах) символ и убрать все символы, начиная с того места, где он стоит, и до конца строки. Искомое регулярное выражение имеет вид /\W.*/. Здесь \W обозначает специальный символ (т.е. все кроме буквы, цифры и знака подчеркивания), а .* обозначают любые символы с этого места до конца строки. Чтобы убрать эти символы, нужно взять ту часть строки, которая совпадает с рассматриваемым регулярным выражением, и заменить ее пустой строкой:

```
$name =~ s/\W.*//;
```

Мы используем ту же операцию =~, что и раньше, но справа у нас теперь стоит операция подстановки — буква s, за которой следуют заключенные между двумя косыми регулярное выражение и строка. (Строка в данном

* Однако GNU-версия утилиты *egrep* выполняет эту операцию гораздо быстрее, чем Perl

примере — это пустая строка между второй и третьей косыми.) Эта операция выглядит и выполняется во многом так же, как операции подстановки в программах-редакторах.

Теперь для того, чтобы перевести все оставшиеся символы в нижний регистр, мы преобразуем эту строку с помощью операции `tr*`. Она очень похожа на UNIX-команду `tr`, т.е. получает список искомых символов и список символов, которыми искомые символы заменяются. В нашем примере мы, чтобы перевести содержимое переменной `$name` в нижний регистр, используем такую запись:

```
$name =~ tr/A-Z/a-z/;
```

Между косыми заключены списки искомых и заменяющих их символов. Дефис между буквами `A` и `Z` обозначает все символы, находящиеся между ними, т.е. у нас есть два списка, в каждый из которых включено по 26 символов. Когда `tr` находит символ из какой-либо строки первого списка, он заменяется соответствующим символом из второго списка. В результате все прописные буквы `A`, `B`, `C` и т. д. становятся строчными**.

Объединяя эти строки с остальной частью программы, получаем:

```
#!/usr/bin/perl
%words = qw(
    fred      camel
    bamey     llama
    betty     alpaca
    wilma     alpaca
),
print "What is your name? ";
$name = <STDIN>;
chomp ($name);
$original_name = $name; # сохранить для приветствия
$name =~ s/\W.*//; # избавиться от всех символов, следующих после первого слова
$name =~ tr/A-Z/a-z/; # перевести все в нижний регистр
if ($name eq "randal") { # теперь можно так сравнить
    print "Hello, Randal! How good of you to be here!\n";
} else {
    print "Hello, $original_name! \n"; # обычное приветствие
    $secretword = $words{$name};      # получить секретное слово
    if ($secretword eq "") {           # не найдено
        $secretword = "groucho";      # конечно, можно использовать
    }
    print "What is the secret word? ";
    $guess = <STDIN>;
    chomp ($guess);
    while ($guess ne $secretword) {
```

* Символы с диакритическими знаками такому переводу не поддаются. Подробности см. на map-странице `perllocale(1)` в версии языка Perl5 004

** Специалисты заметят, что мы также могли написать нечто вроде `s/(\S*) .*/\L$1/`, чтобы сделать все это за один присест, но специалисты, вероятно, не будут читать данный раздел

```
print "Wrong, try again. What is the secret word? ";
$guess = <STDIN>;
chomp ($guess);
```

Обратите внимание на то, что сопоставление с регулярным выражением для слова Randal вновь выполняется с помощью обычной операции сравнения. Дело в том, что и Randal L. Schwartz, и Randal после подстановки и перевода превращаются в randal. Для всех остальных пользователей справедливо то же самое, потому что Fred и Fred Flinstone превращаются во fred; Barney Rubbie и Barney, the little guy — в barney и т.д.

Итак, благодаря всего нескольким операторам наша программа стала гораздо более дружелюбной. Вы увидите, что проведение сложных манипуляций со строками посредством всего лишь нескольких нажатий клавиш — одна из многих сильных сторон языка Perl.

Отметим, однако, что в процессе обработки имени (т.е. при его модификации, необходимой для проведения операции сравнения и поиска соответствия в таблице) первоначально введенное имя уничтожается. Поэтому перед обработкой имени программа сохраняет его в переменной \$original_name. (Как и имена в С, имена переменных в Perl состоят из букв, цифр и знаков подчеркивания, причем длина их практически не ограничена.) Благодаря этому мы впоследствии сможем ссылаться на \$original_name.

В Perl имеется много способов, позволяющих проводить анализ и изменение символов в строках. С большинством из них вы познакомитесь в главах 7 и 15.

Повышение степени модульности

Теперь, когда мы добавили так много строк к нашему первоначальному коду, нам при его просмотре будет непросто уловить общую логику построения программы. Поэтому было бы неплохо отделить высокоуровневую логику (запрос имени, циклы, используемые для обработки введенных секретных слов) от низкоуровневой (сравнение введенного секретного слова с заданным). Это необходимо сделать, например, для облегчения понимания программы другими пользователями, или по той причине, что один человек пишет высокоуровневую часть, а другой — низкоуровневые фрагменты.

В Perl существует понятие *подпрограммы*, имеющей *параметры* и *возвращаемые значения*. Подпрограмма определяется в программе один раз, но использоваться может многократно путем вызова ее из любого места программы.

Давайте создадим для нашей маленькой, но быстро растущей программы подпрограмму good_word, которая будет принимать имя и вариант слова и возвращать значение “истина”, если это слово введено правильно, и “ложь”, если слово набрано неправильно. Определение такой подпрограммы выглядит следующим образом:

```

sub good_word {
    my($somename,$someguess) = @_; # назвать параметры
    $somename =~ s/\W.*//; # избавиться от всех символов, стоящих после
                          # первого слова
    $somename =~ tr/A-Z/a-z/; # перевести все символы в нижний регистр
    if ($somename eq "randal") { # не нужно угадывать
        return 1; # возвращаемое значение — true
    } elsif (($words{$somename} || "groucho") eq $someguess) {
        return 1; # возвращаемое значение — true
    } else {
        return 0; # возвращаемое значение — false
    }
}

```

Во-первых, определение подпрограммы состоит из зарезервированного для этих целей слова `sub`, за которым идет имя подпрограммы и блок ее кода (выделенный фигурными скобками). Это определение может стоять в тексте программы где угодно, но большинство программистов помещают его в конец.

Первая строка в данном конкретном определении — это операция присваивания, с помощью которой значения двух параметров подпрограммы копируются в две локальные переменные с именами `$somename` и `$someguess` (директива `my()` определяет эти переменные как локальные для блока, в который они входят (в данном случае для всей подпрограммы), а параметры первоначально находятся в специальном локальном массиве с именем `@_`.)

Следующие две строки удаляют символы, стоящие после имени (точно так же, как в предыдущей версии программы).

Оператор `if-elsif-else` позволяет определить, является ли введенный пользователем вариант слова (`$someguess`) верным для имени (`$somename`). Имя `Randal` не должно попасть в эту подпрограмму, но даже если и попадет, то любой вариант его ввода будет принят как правильный.

Для того чтобы подпрограмма немедленно возвращала в вызвавшую ее программу указанное в подпрограмме значение, можно воспользоваться оператором возврата. В отсутствие явного оператора возвращаемым значением является последнее выражение, вычисленное в подпрограмме. Мы посмотрим, как используется возвращаемое значение, после того как дадим определение подпрограммы.

Проверка в части `elsif` выглядит довольно сложной; давайте разобьем ее на фрагменты:

```
($words{$somename} || "groucho") eq $someguess
```

Первый элемент в круглых скобках обеспечивает проведение уже знакомого нам хеш-поиска, в результате которого отыскивается некоторое значение в массиве `%words` на основании ключа, полученного из массива `$somename`. Знак `||`, стоящий между этим значением и строкой `groucho`, обозначает операцию ИЛИ, аналогичную той, что используется в языке `C`, `awk` и в различных `shell`. Если поиск в хеше даст некоторое значение (это

значит, что ключ `$somename` находился в хеше), то оно и будет являться значением данного выражения. Если ключ найден не был, то используется строка `groucho`. Это весьма характерно для Perl: приводится некоторое выражение, а затем с помощью операции `||` для него указывается значение по умолчанию на тот случай, если результатом поиска является значение “ложь”.

В любом случае, будь то значение из хеша или принимаемое по умолчанию значение `groucho`, мы сравниваем его с вариантом, вводимым пользователем. Если результат сравнения положителен, возвращается единица, в противном случае возвращается ноль.

Выразим все это в виде правила: если имя — `randal` или если вводимый пользователем вариант соответствует одному из имен, находящемуся в массиве `%words` (со значением по умолчанию `groucho`, если имя в массиве не найдено), то подпрограмма возвращает 1; иначе подпрограмма возвращает 0.

Теперь давайте свяжем все новые строки с остальной частью программы:

```
#!/usr/bin/perl
%words = qw(
    fred      camel
    barney     llama
    betty      alpaca
    wilma      alpaca
);
print "What is your name? ";
$name = <STDIN>;
chomp ($name);
if ($name =~ /^randal\b/i) { # обратно на другой путь :-}
    print "Hello, Randal! How good of you to be here!\n";
} else {
    print "Hello, $name! \n"; # обычное приветствие
    print "What is the secret word? ";
    $guess = <STDIN>;
    chomp ($guess);
    while (! good_word( $name, $guess)) {
        print "Wrong, try again. What is the secret word? ";
        $guess = <STDIN>;
        chomp ($guess) ;
    }
}
[... здесь вставляется определение подпрограммы good_word() ...]
```

Обратите внимание: мы вновь вернулись к использованию регулярного выражения для проверки наличия имени `Randal` в массиве, потому что теперь уже в основной программе не требуется выделять первое имя и заменять все его символы символами нижнего регистра.

Наибольшее отличие этой программы от предыдущей состоит в том, что здесь используется цикл `while`, содержащий подпрограмму `&good_word`. При вызове этой подпрограммы ей передаются два параметра, `$name` и `$guess`. Значение `$somename` устанавливается равным первому параметру,

в данном случае `$name`. Аналогичным образом `$someguess` передается во втором параметре, `$guess`.

Значение, возвращаемое этой подпрограммой (1 или 0, если вы помните приведенное выше правило), логически инвертируется префиксной операцией `!` (логическое НЕ). Эта операция возвращает значение “истина”, если следующее за ней выражение ложно, или “ложь”, если оно истинно. Результат этой операции управляет циклом `while`. Можете читать такую запись как “до тех пор, пока слово угадано неправильно...”. Многие хорошо написанные Perl-программы очень похожи на обычный английский язык — если, конечно, не позволять себе много вольностей ни с языком Perl, ни с английским. (Но Пулитцеровскую премию даже за очень хорошую программу вам не дадут.)

Обратите внимание: при разработке этой подпрограммы предполагалось, что значение хеша `%words` задается в основной программе.

Столь деликатный подход к использованию глобальных переменных вызван тем, что обращаться с ними нужно очень аккуратно. Говоря в общем, переменные, не созданные с помощью `my`, глобальны для всей программы, тогда как переменные `my` действуют только до завершения выполнения блока, в котором они были объявлены. Но не беспокойтесь: в языке Perl имеется множество других разновидностей переменных, включая переменные, локальные для файла (или пакета), и переменные, локальные для функции, которые сохраняют свои значения от вызова к вызову — а это как раз то, что мы могли бы здесь использовать. Впрочем, на данном этапе вашего знакомства с Perl изучение этих переменных только осложнило бы вам жизнь. Когда вы будете достаточно готовы к этому, посмотрите, что говорится о контекстах, подпрограммах, модулях и объектах в книге *Programming Perl*, или обратитесь к диалоговой документации, имеющейся на ман-страницах `perlsub(1)`, `perlmod(1)`, `perlobj(1)` и `perltoot(1)`.

Перенос списка секретных слов в отдельный файл

Допустим, вы хотели бы использовать список секретных слов в трех программах. Если вы сохраните этот список так, как мы уже это делали, нам придется корректировать все три программы (если, например, Бетти решит, что ее секретным словом должно быть не `alpaca`, а `swine`). Это может стать настоящим кошмаром, особенно если Бетти отличается непостоянством.

Поэтому давайте поместим список слов в файл, а затем, чтобы ввести список в программу, просто прочитаем файл. Для этого нужно создать канал ввода-вывода, который называется *дескриптором файла*. Ваша Perl-программа автоматически получает три дескриптора файлов, `STDIN`, `STDOUT` и `STDERR`, которые соответствуют трем стандартным каналам ввода-вывода в большинстве сред программирования. Мы уже используем дескриптор `STDIN` для чтения данных, поступающих от пользователя, запускающего нашу программу. Теперь нужно просто создать для выбранного нами файла другой дескриптор.

Это делается с помощью следующего кода:

```
sub init_words {
  open (WORDSList, "wordslst");
  while ($name = <WORDSList>) {
    chomp ($name);
    $word = <WORDSList>;
    chomp ($word);
    $words{$name} = $word;
  }
  close (WORDSList);
}
```

Мы помещаем его в подпрограмму, чтобы не загромождать основную программу. Это означает также, что позже мы сможем изменить место хранения списка слов и даже его формат.

Произвольно выбранный формат списка слов — один элемент в строке с чередованием имен и секретных слов. Для нашей базы данных мы имели бы такой список:

```
fred
camel
barney
llama
betty
alpaca
wilma
alpaca
```

Функция `open` инициализирует дескриптор файла `WORDSList`, связывая его с файлом `wordslst`, находящимся в текущем каталоге. Отметим, что перед этим дескриптором не ставится никакого забавного символа, вроде тех трех, что предваряют наши переменные. Кроме того, дескрипторы файлов обычно записываются прописными буквами (хотя это и не обязательно); причины этого мы рассмотрим позднее.

При выполнении цикла `while` читаются строки из файла `wordslst` (через дескриптор файла `WORDSList`) по одной при каждом проходе цикла. Каждая строка заносится в переменную `$name`. По достижении конца файла операция `<WORDSList>` возвращает пустую строку*, которая для цикла `while` означает “ложь”, и завершает цикл.

Если бы вы выполняли программу с ключом `-w`, вам пришлось бы проверять, определено ли полученное возвращаемое значение. Пустая строка, которую возвращает операция `<WORDSList>`, не совсем пуста: это опять значение `undef`. В тех случаях, когда это важно, проверка выражения на значение `undef` производится функцией `defined`. При чтении строк из файла эта проверка выполнялась бы следующим образом:

```
while ( defined ($name = <WORDSList> ) ) {
```

* На самом деле это опять `undef`, но для понимания данного материала сказанного достаточно.

Но если бы вы были еще более осторожны, вы, вероятно, проверили бы также и то, возвращает ли функция `open` значение “истина”. Это, кстати, в любом случае неплохая идея. Для выхода из программы с сообщением об ошибке в случае, если что-то работает не так, часто используется встроенная функция `die`. Мы рассмотрим пример этой функции в следующей версии нашей программы.

С другой стороны, при нормальном развитии событий мы считываем строку (включая символ новой строки) в переменную `$name`. Сначала с помощью функции `chomp` убирается символ новой строки, затем нужно прочитать следующую строку, чтобы получить секретное слово и сохранить его в переменной `$word`. Символ новой строки при этом тоже убирается.

Последняя строка цикла `while` помещает `$word` в `%words` с ключом `$name`, чтобы переменную `$word` могла использовать остальная часть программы.

По завершении чтения файла его дескриптор можно использовать повторно, предварительно закрыв файл с помощью функции `close`. (Дескрипторы файлов автоматически закрываются в любом случае при выходе из программы, но мы стараемся быть аккуратными. Однако если бы мы были по-настоящему аккуратными, мы бы даже проверили бы, возвращает ли `close` значение “истина” в случае, если раздел диска, в котором был файл, решил “отдохнуть”, если сетевая файловая система стала недостижимой или произошла еще какая-нибудь катастрофа. Такое ведь иногда случается. Законы Мерфи никто не отменял.)

Сделанное выше определение подпрограммы может идти после другого аналогичного определения или перед ним. Вместо того чтобы помещать определение `%words` в начало программы, мы можем просто вызывать эту подпрограмму в начале выполнения основной программы. Один из вариантов компоновки общей программы может выглядеть так:

```
#!/usr/bin/perl
init_words() ;
print "What is your name? ";
$name = <STDIN>;
chomp $name;
if ($name =~ /^randal\b/i) { # обратно на другой путь :-}
    print "Hello, Randal! How good of you to be here!\n";
} else {
    print "Hello, $name! \n"; # обычное приветствие
    print "What is the secret word? ";
    $guess = <STDIN>;
    chomp ($guess) ;
    while (! good_word($name, $guess)) {
        print "Wrong, try again. What is the secret word? ";
        $guess = <STDIN>;
        chomp ($guess);
    }
}
## далее — подпрограммы
```

```

sub init_words {
    open (WORDS_LIST, "wordlist") ||
        die "can't open wordlist: $!";
    while ( defined ($name = <WORDS_LIST>) ) {
        chomp ($name) ;
        $word = <WORDS_LIST>;
        chomp $word;
        $words{$name} = $word;
    }
    close (WORDS_LIST) || die "couldn't close wordlist: $!";
}

sub good_word {
    my($somename,$someguess) = @_; # перечислить параметры
    $somename =~ s/\W.*//;         # удалить все символы, стоящие после
    первого слова                  # первое слово
    $somename =~ tr/A-Z/a-z/;      # перевести все символы в нижний регистр
    if ($somename eq "randal") {   # не нужно угадывать
        return 1;                 # возвращаемое значение -- true
    } elsif (($words{$somename} || "groucho") eq $someguess) {
        return 1;                 # возвращаемое значение -- true
    } else {
        return 0;                 # возвращаемое значение -- false
    }
}

```

Теперь написанный нами код начинает выглядеть как настоящая “взрослая” программа. Обратите внимание: первая выполняемая строка — вызов подпрограммы `init_word()`. Возвращаемое значение в последующих вычислениях не используется, и это хорошо, потому что мы не возвратили ничего заслуживающего внимания. В данном случае это гарантированно значение “истина” (в частности, значение 1), потому что если бы `close` не выполнялась, то `die` вывела бы сообщение в `STDERR` и вышла из программы. Функция `die` подробно описывается в главе 10, но поскольку очень важно проверять возвращаемые значения всего, что может завершиться неудачно, мы возьмем за правило использовать эту функцию с самого начала. Переменная `$!` (тоже рассматривается в главе 10) содержит системное сообщение об ошибке, поясняющее, почему данный системный вызов завершился неудачно.

Функция `open` используется также для открытия файлов при выводе в них информации и открытия программ как файлов (здесь она лишь упомянута). Полное описание этой функции будет дано гораздо позже, в главе 10.

Как обеспечить скромный уровень безопасности

“Этот список секретных слов должен меняться минимум раз в неделю!”, — требует Главный Директор Списков Секретных Слов. Мы не можем, конечно, заставить пользователей еженедельно менять пароли, но должны хотя бы предупреждать их о том, что список секретных слов не изменялся в течение семи дней и более.

Лучше всего делать это в подпрограмме `init_words()`; мы уже работаем в ней с файлом `WORDSLIST`. Perl-операция `-M` возвращает значение, равное количеству дней, прошедшему с момента изменения файла или дескриптора файла, поэтому нам нужно просто посмотреть, превышает ли это значение число семь для дескриптора файла `WORDSLIST`:

```
sub init_words {
    open (WORDSLIST, "wordlist") ||
        die "can't open wordlist: $!";
    if (-M WORDSLIST >= 7.0) { # в соответствии с бюрократическими правилами
        die "Sorry, the wordlist is older than seven days. ";
    }
    while ($name = <WORDSLIST>) {
        chomp ($name) ;
        $word = <WORDSLIST> ;
        chomp ($word) ;
        $words{$name} = $word;
    }
    close (WORDSLIST) || die "couldn't close wordlist: $!";
}
```

Значение `-M WORDSLIST` сравнивается со значением 7. Если оно больше, то мы, выходит, нарушили правила. Здесь мы видим новую операцию, операцию `die`, которая одним махом выводит сообщение на экран* и прерывает программу.

Остальная часть программы изменений не претерпевает, поэтому в целях экономии бумаги мы ее повторять не будем.

Помимо определения “возраста” файла мы можем узнать имя его владельца, размер, время последнего доступа к нему и все остальные сведения, хранимые системой о каждом файле. Более подробно об этом написано в главе 10.

Как предупредить пользователя, если он сбился с пути

Давайте посмотрим, как можно заставить систему посылать сообщение электронной почты всякий раз, когда пользователь указывает свое секретное слово неверно. Нам нужно модифицировать только подпрограмму `good_word()` (сказывается преимущество модульности языка Perl), потому что вся необходимая информация находится у нас там.

Почтовое сообщение будет послано вам в том случае, если вы вставите свой адрес электронной почты там, где в программе записано `YOUR_ADDRESS_HERE`. Все, что нам нужно для этого сделать, это непосредственно перед тем, как возвращать из подпрограммы 0, создать дескриптор файла, который фактически будет являться процессом (*mail*):

```
sub good_word {
    my($somename,$someguess) = @_ ; # перечислить параметры
    $somename =~ s/\W.*//;           # удалить все символы, стоящие после
                                     # первого слова
}
```

* Если точнее, то в дескриптор файла `STDERR`, но обычно это и означает терминал

```

$somename =~ tr/A-Z/a-z;      # перевести все символы в нижний регистр
if ($somename eq "randal") {  # не нужно угадывать
    return 1;                 # возвращаемое значение — true
} elsif (($words{$somename}||"groucho") eq $someguess) {
    return 1;                 # возвращаемое значение — true
} else {
    open MAIL,"|mail YOUR_ADDRESS_HERE";
    print MAIL "bad news: $somename guessed $someguess\n";
    close MAIL;
    return 0;                 # возвращаемое значение — false
}
}

```

Первый новый оператор здесь — `open`, в начале второго аргумента которого стоит оператор канала (`|`). Он указывает, что мы открываем процесс, а не файл. Поскольку оператор канала находится перед именем команды, мы открываем процесс так, чтобы можно было осуществить в него запись. (Если поставить оператор канала в конец, а не в начало, то можно будет читать выходную информацию команды.)

Следующий оператор, `print`, выбирает для вывода не `STDOUT*`, а дескриптор файла, стоящий между ключевым словом `print` и подлежащими выводу на экран значениями. Это значит, что сообщение в конечном итоге станет входной информацией для команды *mail*.

Наконец, мы закрываем дескриптор файла, в результате чего запускается программа *mail* и передает свои данные.

Чтобы соблюсти все формальности, мы могли бы посылать не только ошибочный, но и правильный ответ, но тогда тот, кто заглядывает нам через плечо (или прячется в системе электронной почты), когда мы читаем сообщения, получил бы слишком много полезной информации.

Perl может также открывать дескрипторы файлов, вызывать команды с необходимыми аргументами, даже порождать копию текущей программы и выполнять две (или более) копии программы одновременно. Обратные кавычки (как в *shell*) дают возможность получать результаты работы команды как данные. Все это описывается в главе 14, так что читайте дальше.

Несколько файлов секретных слов в текущем каталоге

Давайте слегка изменим способ определения имени файла секретных слов. Вместо файла с именем `wordlist` будем искать в текущем каталоге нечто, заканчивающееся на `.secret`. Попросим *shell* выдать краткий перечень таких имен.

```
echo *.secret
```

* Говоря техническим языком — выбранный в текущий момент дескриптор файла. Об этом, однако, мы поговорим позже.

Как вы скоро увидите, Perl применяет похожий синтаксис имен с использованием метасимволов.

Еще раз вернемся к определению подпрограммы `init_words()`:

```
sub init_words {
    while ( defined($filename = glob("*.secret")) ) {
        open (WORDSLIST, $filename) ||
            die "can't open wordlist: $!";
        if (-M WORDSLIST >= 7.0) {
            while ($name = <WORDSLIST>) {
                chomp $name;
                $word = <WORDSLIST>;
                chomp $word;
                $words{$name} = $word;
            }
        }
        close (WORDSLIST) || die "couldn't close wordlist: $!";
    }
}
```

Сначала мы поместили в новый цикл `while` основную часть подпрограммы из предыдущей версии. Новый элемент здесь — функция `glob`. По историческим причинам она называется *filename glob*. Эта функция работает почти так же, как `<STDIN>`: при каждом обращении к ней она возвращает очередное значение из списка имен файлов, которые соответствуют образцу `shell`, в данном случае `*.secret`. Если таких имен файлов нет, возвращается пустая строка*.

Таким образом, если текущий каталог содержит файлы `fred.secret` и `barney.secret`, то при первом выполнении цикла `while` значением переменной `$filename` будет `barney.secret` (имена даются по алфавиту). При втором выполнении цикла значением `$filename` будет `fred.secret`. Поскольку при третьем вызове функции `glob` она возвращает пустую строку, то третий проход не делается, так как цикл `while` интерпретирует это значение как “ложь”, что приводит к выходу из подпрограммы.

В ходе выполнения цикла `while` мы открываем файл и проверяем, достаточно ли давно он обновлялся (с момента последнего изменения должно пройти не более семи дней). С этими достаточно новыми файлами мы работаем так же, как и раньше.

Отметим, что в отсутствие файлов, имена которых совпадали бы с шаблоном `*.secret` и “возраст” которых не превышал бы семи дней, подпрограмма завершится, не поместив ни одного секретного слова в массив `%words`. Это значит, что всем придется пользоваться словом `groucho`. Прекрасно. (В *реальном* коде перед выходом из подпрограммы следовало бы ввести операцию проверки количества элементов в массиве `%words` — и при неудовлетворительном результате выполнить функцию `die`. Обратите внимание на функцию `keys`, когда мы дойдем до определения хешей в главе 5.)

* Да-да, опять `undef`

Как получить список секретных слов

Итак, Главный Директор Списков Секретных Слов желает получить отчет обо всех секретных словах, используемых в текущий момент, с указанием их “возраста”. Если мы на минутку расстанемся с программой проверки секретного слова, у нас будет время написать для Директора программу формирования необходимого ему отчета.

Сперва давайте получим все наши секретные слова, воспользовавшись для этого частью кода из подпрограммы `init words()` :

```
while ( defined($filename = glob("*.secret")) ) {
open (WORDSLIST, $filename) || die "can't open wordlist: $!";
if (-M WORDSLIST >= 7.0) {
    while ($name = <WORDSLIST>) {
        chomp ($name);
        $word = <WORDSLIST> ;
        chomp ($word) ;
### отсюда начинается новый код }
    }
}
close (WORDSLIST) || die "couldn't close wordlist: $!"
}
```

К моменту достижения того места программы, где дан комментарий “отсюда начнется новый код”, мы знаем три вещи: имя файла (содержится в переменной `$filename`), чье-то имя (в переменной `$name`) и секретное слово этого человека (содержится в `$sword`). Здесь и нужно использовать имеющиеся в Perl инструменты формирования отчетов. Для этого где-то в программе мы должны определить используемый формат (обычно это делается в конце, как и для подпрограмм):

```
format STDOUT =  
@<<<<<<<<<<< @<<<<<<< @<<<<<<<<<  
$filename, $name, $word  
.
```

Определение формата начинается строкой `format STDOUT=`, а завершается точкой. Две строки между первой строкой и точкой — это сам формат. Первая строка формата — это *строка определения полей*, в которой задается число, длина и тип полей. В этом формате у нас три поля. Строка, следующая за строкой определения полей — это всегда *строка значений полей*. Строка значений содержит список выражений, которые будут вычисляться при использовании формата; результаты вычисления этих выражений вставляются в поля, определенные в предыдущей строке.

Вызывается определенный таким образом формат функцией *write*, например:

```
#!/usr/bin/perl
while ( defined($filename = glob("*.secret")) ) {
    open (WORDSLIST, $filename) || die "can't open wordlist: $!";
```


Чтобы это определение заработало, попробуйте присоединить его к предыдущей программе. Perl отыщет формат начала страницы автоматически.

В Perl имеются также поля, которые центрируются и выравниваются по правому краю области вывода. Этот язык, кроме того, поддерживает одновременное выравнивание и по правому, и по левому краям. Подробно об этом мы поговорим, когда дойдем до форматов, в главе 11.

Как сделать старые списки слов более заметными

Просматривая файлы *.secret в текущем каталоге, мы, возможно, обнаружим слишком старые файлы. До сих пор мы просто пропускали их. Давайте сделаем очередной шаг и переименуем такие файлы в *.secret.old, чтобы в перечне содержимого каталога сразу было видно — по имени — какие файлы необходимо обновить.

Вот как выглядит подпрограмма `init_words()`, модифицированная для выполнения такой операции:

```
sub init_words {
    while ( defined($filename = glob("*.secret")) ) {
        open (WORDSLIST, $filename) ||
            die "can't open wordlist: $!";
        if (-M WORDSLIST >= 7.0) {
            while ($name = <WORDSLIST>) {
                chomp ($name);
                $word = <WORDSLIST>;
                chomp ($word);
                $words {$name} = $word;
            }
        } else { # переименовать файл, чтобы он стал более заметным
            rename ($filename, "$filename.old") ||
                die "can't rename $filename to $filename.old: $!";
        }
        close (WORDSLIST) || die "couldn't close wordlist: $!";
    }
}
```

Обратите внимание на новую часть оператора `else` в блоке проверки “возраста” файлов. Если файл не обновлялся семь дней и более, функция `rename` переименовывает его. Эта функция принимает два параметра и переименовывает файл, заданный первым параметром, присваивая ему имя, указанное во втором параметре.

В Perl имеется полный набор операций, необходимых для манипулирования файлами; все, что можно сделать с файлом в C-программе, можно сделать с ним и в Perl.

Ведение базы данных о времени правильного ввода секретных слов

Давайте проследим за тем, когда каждый пользователь последний раз правильно вводил свой вариант секретного слова. Очевидно, единственная структура данных, которая годится для этого — хеш. Например, оператор

```
$last_good{$name} = time;
```

присваивает значение текущего времени, выраженное во внутреннем формате (некоторое большое целое, свыше 800 миллионов, число, которое увеличивается на единицу каждую секунду) элементу хеша `%last_good`, имеющему указанное имя в качестве ключа. В последующем это даст базу данных о времени последнего правильного ввода секретного слова каждым из пользователей, который вызывал эту программу.

При этом, однако, в промежутках между вызовами программы хеш не существует. Каждый раз, когда программа вызывается, формируется новый хеш, т.е. по большому счету мы всякий раз создаем одноэлементный хеш, а по завершении выполнения программы немедленно про него забываем.

Функция `dbmopen`* отображает хеш в файл (фактически в пару файлов), известный как *DBM-файл*. Она используется следующим образом:

```
dbmopen (%last_good,"lastdb",0666) ||  
die "can't dbmopen lastdb: $!";  
$last_good{$name} = time;  
dbmclose (%last_good) || die "can't dbmclose lastdb: $!";
```

Первый оператор выполняет отображение, используя имена файлов `lastdb.dir` и `lastdb.pag` (это общепринятые имена для файлов `lastdb`, образующих DBM-файл). Если эти файлы необходимо создать (а это бывает при первой попытке их использования), то для них устанавливаются права доступа `0666`**. Такой режим доступа означает, что все пользователи могут читать и осуществлять запись в эти файлы. Если вы работаете в UNIX-системе, то описание битов прав доступа к файлу вы найдете на man-странице `chmod(2)`. В других системах `chmod()` может работать так же, а может и не работать. Например, в MS-DOS для файлов не устанавливаются права доступа, тогда как в Windows NT — устанавливаются. Если уверенности нет, прочтите описание версии вашей системы.

Второй оператор показывает, что мы используем этот преобразованный хеш как первоначальный. При этом, однако, при создании или корректировке какого-либо элемента хеша автоматически обновляются файлы, образующие DBM-файл. При последующем обращении к хешу значения его элементов поступают непосредственно из его отображения на диске. Благодаря этому

* Можно также использовать низкоуровневую функцию `tie` с конкретной базой данных; этот вариант подробно описан в главах 5 и 7 книги *Programming Perl* и на man-страницах `perlite(1)` и *AnyDBM_File* (3).

** Фактически права доступа к этим файлам определяются в результате выполнения логической операции И над числом `0666` и текущим значением переменной `umask` вашего процесса

В Perl используются также удобные способы создания и ведения текстовых баз данных (например, файла паролей) и баз данных с фиксированной длиной записей (таких, как база данных “последней регистрации”, которую ведет программа *login*). Эти способы описаны в главе 17.

Окончательные варианты программ

Здесь вашему вниманию предлагаются программы, которые мы писали в этой главе, в окончательном виде.

Сначала — программа-приветствие:

```
#!/usr/bin/perl
&init_words ( ) ;
print "what is your name? " ;
$name = <STDIN>;
chomp ($name);
if ($name =~ /^randal\b/1) { # обратно на другой путь :-}
    print "Hello, Randal! How good of you to be here'\n";
} else {
    print "Hello, $name! \n"; # обычное приветствие
    print "What is the secret word? ";
    $guess = <STDIN> ;
    chomp $guess ;
    while (! good_word( $name, $guess)) {
        print "Wrong, try again. What is the secret word? ";
        $guess = <STDIN> ;
        chomp $guess;
    }
}
dbmopen (%last_good, "lastdb", 0666);
$last_good{$name} = time;
dbmclose (%last_good);
sub init_words {
    while ($filename = <*.secret>) {
        open (WORDS_LIST, $filename) ||
            die "can't open $filename: $!";
        if (-M WORDS_LIST < 7) {
            while ($name = <WORDS_LIST>) {
                chomp ($name) ;
                $word = <WORDS_LIST> ;
                chomp ($word);
                $words {$name} = $word ;
            }
        } else { # rename the file so it gets noticed
            rename ($filename, "$filename.old") ||
                die "can't rename $filename.old: $!";
        }
        close WORDS_LIST;
    }
}
sub good_word {
    my($somename,$someguess) = @_ ; # перечислить параметры
    $somename =~ s/\W.*//; # удалить все символы, стоящие после первого слова
    $somename =~ tr/A-Z/a-z/; # перевести все символы в нижний регистр
    if ($somename eq "randal") { # не нужно угадывать
```


Добавьте к этим программам списки секретных слов (файлы с именами *что-то.secret*, находящиеся в текущем каталоге) и базу данных `lastdb.dir` и `lastdb.pag`, и у вас будет все, что нужно.

Упражнение

Большинство глав завершаются упражнениями, ответы к которым даются в приложении А. Для этой главы ответы уже были даны выше.

1. Наберите программы-примеры и заставьте их работать. (Вам понадобится создать списки секретных слов.) Если потребуется помощь — обратитесь к местному Perl-гuru.

В этой главе:

- *Что такое скалярные данные*
- *Числа*
- *Строки*
- *Скалярные операции*
- *Скалярные переменные*
- *Скалярные операции и функции*
- *<STDIN> как скалярное значение*
- *Вывод с помощью функции print*
- *Значение undef*
- *Упражнения*

2

Скалярные данные

Что такое скалярные данные

Скаляр — это простейший вид данных, которыми манипулирует Perl. Скаляр — это либо число (допустим, 4 или 3.25e20), либо строка символов (например, `hello` или `Gettysburg Address`). Хотя в общем-то числа и строки — это совершенно разные вещи, в Perl они используются практически как взаимозаменяемые понятия, поэтому мы опишем их в одной главе.

Над скалярной величиной можно производить операции (например, суммирование или конкатенацию), полученный результат, как правило, также является скаляром. Скалярную величину можно сохранять в скалярной переменной. Скаляры можно читать из файлов и с устройств, а также записывать в файлы и на устройства.

Числа

Хотя скаляр — это либо число, либо строка*, в данный момент нам будет полезно рассмотреть их отдельно. Итак, сначала числа, а через минуту — строки.

В Perl для всех чисел используется один и тот же внутренний формат

Как станет ясно из нескольких следующих абзацев, можно задавать и целые (чисто числовые значения, например 17 или 342), и числа с плавающей запятой (действительные числа, например 3,14 или 1,35, умноженное на 10^{25}). При этом

* Или ссылка, но это более сложная тема

во внутренних вычислениях Perl использует только значения с плавающей запятой двойной точности*. Это значит, что внутренних *целых* величин в Perl нет; целочисленная константа в программе рассматривается как эквивалентное значение с плавающей запятой**. Вы, вероятно, не заметите этого преобразования (или попросту проигнорируете его), но в любом случае не нужно искать целочисленные операции (как дополнение к операциям с *плавающей запятой*), ибо их попросту нет.

Литералы с плавающей запятой

Литерал — это способ представления величины в тексте Perl-программы. В своей программе вы можете называть литерал *константой*, но мы будем пользоваться термином *литерал*. Литералы — это способ представления данных в исходном тексте вашей программы как входной информации для компилятора Perl. (Данные, которые читаются из файлов или записываются в файлы, трактуются похоже, но не совсем так.)

Perl принимает полный набор литералов с плавающей запятой, которыми пользуются программисты, работающие на C. Допускаются числа с десятичными запятыми и без них (включая необязательный префикс “плюс” или “минус”). Кроме того, можно использовать показатель степени числа десять (экспоненциальное представление) с буквой E. Например:

```
1.25      # около единицы с четвертью
7.25e45   # 7,25, умноженное на 10 в 45-й степени (большое число)
-6.5e24   # минус 6,5, умноженное на 10 в 24-й степени
          # ("большое" отрицательное число)
-12e-24   # минус 12, умноженное на 10 в минус 24-й степени
          # (очень маленькое отрицательное число)
-1.2E-23  # еще одна форма записи этого числа
```

Целочисленные литералы

Целочисленные литералы также весьма просты, например:

```
12
15
-2004
3485
```

Не начинайте целое число с нуля, потому что Perl поддерживает восьмеричные и шестнадцатеричные литералы. Восьмеричные числа начинаются с нуля, а шестнадцатеричные — с 0x или 0X***. Шестнадцатеричные цифры

* Значение с плавающей запятой двойной точности — это то, что компилятор C, который компилировал Perl, использовал для объявления double.

** Если только вы не используете “целочисленный режим”, но по умолчанию он не включен.

*** “Начальный ноль” работает только в литералах, но не действует при автоматическом преобразовании строк в числа. Строку данных, выглядящую как восьмеричное или шестнадцатеричное значение, можно преобразовать в число с помощью функций oct или hex

от А до F (в любом регистре) обозначают обычные цифровые значения от 10 до 15. Например:

```
0377      # восьмеричное 377, то же самое, что десятичное 255
-0xfff     # отрицательное шестнадцатеричное FF, то же самое, что десятичное -255
```

Строки

Строки — это последовательности символов (например, `hello`). Каждый символ представляет собой 8-битовое значение из 256-символьного набора (при этом символ NUL ничего особенного, как в некоторых языках, собой не представляет).

Самая короткая из возможных строк не содержит ни одного символа. Самая длинная строка заполняет всю наличную память (но сделать с ней что-либо вы вряд ли сможете). Это соответствует принципу “отсутствия встроенных ограничений”, которому Perl следует при каждой возможности. Обычно строки представляют собой последовательности букв, цифр и знаков препинания, коды которых лежат в диапазоне ASCII 32 — 126. Однако возможность наличия в строке любого символа с кодом от 0 до 255 означает, что вы можете создавать, просматривать необработанные двоичные данные и манипулировать ими как строками — то, что вызвало бы серьезные трудности в большинстве других языков. (Например, можно “залатать” свою операционную систему, прочитав нужный фрагмент кода как Perl-строку, внося изменение и записав результат обратно на диск.)

Как и числа, строки могут иметь литеральное представление (способ представления строкового значения в Perl-программе). Литеральные строки бывают двух видов: *в одинарных кавычках* и *в двойных кавычках**. Есть еще одна похожая форма: строка в обратных кавычках (``вот такая``). Это не столько литеральная строка, сколько способ выполнения внешних команд и получения их результатов. Более подробно этот вопрос рассматривается в главе 14.

Строки в одинарных кавычках

Строка в одинарных кавычках представляет собой последовательность символов, заключенную в одинарные кавычки. Одинарные кавычки частью самой строки не являются, они служат лишь для того, чтобы Perl мог определить начало и окончание строки. Все символы, стоящие между кавычками (в том числе символы новой строки, если строка разбивается на несколько экранных строк), действительны внутри строки. Два исключения: чтобы вставить одинарную кавычку в строку, уже заключенную в одинарные кавычки, поставьте перед ней обратную косую черту. Для того чтобы вставить

* Есть также *here*-строки, похожие на *here*-документы shell. Они рассматриваются в главе 19, *Программирование CGI*. См. также главу 2 книги *Programming Perl* и map-страницу `perldata(1)`.

в строку в одинарных кавычках обратную косую, поставьте перед ней еще одну обратную косую черту. Примеры:

```
'hello'      # пять символов: h, e, l, l, o
'don\'t'     # пять символов: d, o, n, одинарная кавычка, t
''          # пустая строка (без символов)
'silly\\me'  # silly, обратная косая, me
'hello\\n'   # hello, обратная косая, n
'hello
there'      # hello, новая строка, there (всего 11 символов)
```

Отметим, что пара символов `\n` внутри строки в одинарных кавычках интерпретируется не как символ новой строки, а как два символа: обратная косая и `n`. (Обратная косая имеет специальное значение только в том случае, если за ней следует еще одна обратная косая или одинарная кавычка.)

Строки в двойных кавычках

Строка в двойных кавычках действует почти так же, как C-строка. Это тоже последовательность символов, но на этот раз — заключенная в двойные кавычки. Теперь, однако, обратная косая приобретает полную силу и может задавать определенные управляющие символы и вообще любой символ в восьмеричном и шестнадцатеричном форматах. Вот некоторые строки в двойных кавычках:

```
"hello world\n" # hello world и новая строка
"new \177"     # new, пробел и символ удаления (восьмеричное 177)
"coke\tsprite" # coke, знак табуляции, sprite
```

Широко используются сочетания обратной косой черты с различными символами (так называемая *управляющая последовательность с обратной косой*). Полный перечень управляющих последовательностей, которые применяются в строках в двойных кавычках, приведен в табл. 2.1.

Таблица 2.1. Управляющие последовательности

Конструкция	Значение
<code>\n</code>	Переход на новую строку (Newline)
<code>\r</code>	Возврат к началу строки (Return)
<code>\t</code>	Табуляция
<code>\f</code>	Переход к новой странице (Formfeed)
<code>\b</code>	Возврат на предыдущую позицию с удалением символа (Backspace)
<code>\a</code>	Сигнал
<code>\e</code>	Escape

Конструкция	Значение
\007	Восьмеричное ASCII-значение (в данном случае 07 = сигнал)
\x7f	Шестнадцатеричное ASCII-значение (в данном случае 7f = удалить)
\cC	Управляющий символ (здесь Ctrl+C)
\\	Обратная косая
\ "	Двойная кавычка
\l	Перевод следующей буквы в нижний регистр
\L	Перевод в нижний регистр всех последующих букв до \E
\u	Перевод в верхний регистр следующей буквы
\U	Перевод в верхний регистр всех последующих букв до \E
\Q	Заключить в обратные косые все небуквенные и все нецифровые символы до \E
\E	Отменить действие последовательности \L, \U или \Q

Еще одна особенность строк в двойных кавычках состоит в том, что в них производится интерполяция переменных, т.е. при использовании строки все скалярные переменные и переменные-массивы в ней заменяются их текущими значениями. Мы, однако, формально еще не знакомы с тем, что такое переменная (за исключением краткого обзора, данного в главе 1), поэтому мы вернемся к этому вопросу позднее.

Скалярные операции

Оператор обозначает проведение определенной операции, благодаря которой создается новое значение (*результат*) из одного или нескольких других значений (*операндов*). Например, символ + обозначает операцию, потому что при его использовании берутся два числа (операнда, например, 5 и 6) и создается новое значение (11, результат).

Вообще говоря, операции и выражения Perl представляют собой надмножество операций и выражений, имеющихся в большинстве других АЛГОЛ-и Паскаль-подобных языков программирования (вроде C или Java). При выполнении операции предполагается наличие или числовых, или строковых операндов (либо сочетания операндов этих типов). Если вы введете строковый операнд там, где должно быть число, или наоборот, Perl автоматически преобразует этот операнд, используя весьма нечетко сформулированные правила, которые мы подробно рассмотрим ниже в разделе “Преобразование чисел в строки и обратно”.

Операции над числами

В Perl применяются обычные операции сложения, вычитания, умножения, деления и т.д. Например:

```
2 + 3           # 2 плюс 3, или 5
5.1 - 2.4       # 5,1 минус 2,4, или приблизительно 2,7
3 * 12          # 3 умножить на 12 = 36
14 / 2          # 14 делить на 2, или 7
10.2 / 0.3      # 10,2 делить на 0,3, или приблизительно 34
10 / 3          # всегда означает деление чисел с плавающей запятой,
                # поэтому результат приблизительно равен 3,3333333...
```

Кроме того, в Perl используется ФОРТРАН-подобная операция *возведения в степень*, по которой многие тоскуют в Паскале и С. Эта операция обозначается двумя звездочками, например $2^{**}3$ равно двум в степени три, или восьми. (Если этот результат “не помещается” в число с плавающей запятой двойной точности, например, отрицательное число возводится в нецелую степень или большое число возводится в большую степень, выдается сообщение об ошибке — fatal error)

Perl поддерживает также операцию *деления с остатком*. Значение выражения $10 \% 3$ — остаток от деления 10 на 3, или 1. Оба значения сначала сокращаются до целых, т.е. $10.5 \% 3.2$ вычисляется как $10 \% 3$.

Операции логического сравнения следующие: $<$, $<=$, $=$, $>=$, $>$, $!=$. Эти операции сравнивают два значения в числовом формате и возвращают значение “истина” (*true*) или “ложь” (*false*). Например, операция $3 > 2$ возвращает значение “истина”, потому что три больше, чем два, тогда как операция $5 != 5$ возвращает “ложь”, потому что утверждение, что пять не равно пяти — неправда. Определение значений “истина” и “ложь” рассматривается позже, а пока можно считать, что “истина” — это единица, а “ложь” — нуль. (Вы еще увидите эти операции в табл. 2.2.)

Вас, вероятно, удивило слово “приблизительно” в комментариях к примерам, которые мы привели в начале этого раздела. Разве при вычитании 2,4 из 5,1 не получается точно 2,7? На уроке математики, может быть, и получается, но в компьютерах, как правило, — нет. В вычислительной технике получается приближенное значение, которое точно лишь до определенного числа десятичных разрядов. Числа в компьютерах хранятся не так, как их представляет себе математик. Впрочем, если вы не делаете чего-нибудь сверхсложного, то, как правило, увидите именно те результаты, которых ожидаете.

Сравнивая приведенные ниже операторы, вы увидите, что в действительности компьютер получил в результате вышеупомянутого вычитания (функция `printf` описывается в главе 6):

```
printf("%.51\n", 5.1 - 2.4)
# 2.699999999999999733546474089962430298328399658203125

print(5.1 - 2.4, "\n");
# 2.7
```

Не обращайтесь на это особого внимания: стандартный формат функции `print` для вывода на экран чисел с плавающей запятой обычно скрывает такие незначительные неточности представления. Если это создает какую-то проблему, следует воспользоваться объектными модулями `Math::BigInt` и `Math::BigFloat` — в них реализована арифметика с бесконечной точностью для целых и чисел с плавающей запятой, только выполняются эти операции несколько медленнее. Подробности вы найдете в главе 7 книги *Programming Perl* и в интерактивной (сетевой) документации на эти модули.

Операции над строками

Строковые значения можно конкатенировать операцией `“.”`. (Да, это именно одна точка.) Данная операция изменяет строки-операнды не более, чем операция `2+3` изменяет 2 или 3. Строку-результат (более длинную) можно использовать в дальнейших вычислениях или сохранить в переменной.

```
"hello" . "world"          # то же самое, что "helloworld"
'hello world' . "\n"       # то же самое, что "hello world\n"
"fred" . " " . "barney"    # то же самое, что "fred barney"
```

Отметим, что конкатенацию нужно задавать явно при помощи знака `“.”`, а не просто располагать два значения рядом друг с другом.

Еще одна группа операций над строками — операции сравнения. Эти операции похожи на соответствующие операции ФОРТРАНа, например `lt` (меньше чем) и т.д. Операции сравнения строк сравнивают ASCII-значения символов строк обычным способом. Полный набор операций сравнения (как для чисел, так и для строк) приведен в таблице 2.2.

Таблица 2.2. Операции сравнения чисел и строк

Сравнение	Числовое	Строковое
Равно	<code>==</code>	<code>eq</code>
Не равно	<code>!=</code>	<code>ne</code>
Меньше чем	<code><</code>	<code>lt</code>
Больше чем	<code>></code>	<code>gt</code>
Меньше чем или равно	<code><=</code>	<code>le</code>
Больше чем или равно	<code>>=</code>	<code>ge</code>

Вы, возможно, спросите, почему предусмотрены отдельные операции для чисел и строк, если числа можно автоматически преобразовывать в строки и наоборот. Давайте рассмотрим два значения, 7 и 30. Если их сравнить как числа, то 7, очевидно, меньше 30. Если же сравнить их как строки, то строка `"30"` идет *перед* строкой `"7"` (ведь ASCII-значение цифры 3 меньше ASCII-значения цифры 7), поэтому 30 меньше 7. Perl всегда требует указывать тип сравнения, то есть конкретизировать, какое именно сравнение, числовое или строковое, будет проводиться.

Если вы уже имеете опыт программирования на shell в среде UNIX, вы, возможно, заметили, что эти числовые и строковые операции сравнения приблизительно противоположны тому, что подразумевается под ними в UNIX-команде *test*. В частности, для числового сравнения используется *eq*, а для строкового применяется знак *=*.

Есть еще одна строковая операция — операция *повторения строки*, знак которой — строчная буква *x*. В результате выполнения этой операции возвращается столько конкатенированных копий левого операнда (строки), сколько указано в правом операнде (числе). Например:

```
"fred" x 3          # "fredfredfred"
"barney" x (4+1)    # "barney" x 5, или
                   # "barneybarneybarneybarneybarney"
(3+2) x 4           # 5 x 4, или "5" x 4, т.е. "5555"
```

Последний пример стоит объяснить подробнее. Круглые скобки указывают на то, что эту часть выражения нужно вычислить первой. (Круглые скобки здесь работают так же, как в обычной математике.) Однако в операции повторения строки левый операнд должен быть строкой, поэтому число 5 преобразуется в односимвольную строку "5" (правила этого преобразования мы рассмотрим позднее). Эта строка копируется четыре раза, что в итоге дает четырехсимвольную строку 5555. Если бы мы поменяли операнды местами, то у нас было бы пять копий строки 4, т.е. 44444. Это показывает, что повторение строки — некоммутативная операция.

При необходимости число копий (правый операнд) сначала усекается до целого значения (4, 8 превращается в 4). Если указать число копий, меньшее единицы, в результате выполнения операции получится пустая строка (строка нулевой длины).

Приоритет и ассоциативность операций

Разным операциям присваивается разный приоритет. Это позволяет избежать неоднозначности такого рода, когда две операции пытаются манипулировать тремя операндами. Например, что выполнять в первую очередь в выражении $2+3*4$ — сложение или умножение? Если сначала выполнить сложение, мы получим $5*4$, или 20. Если же сначала выполнить умножение (как нас учили на уроках математики), то мы получим $2+12$, или 14. К счастью, в Perl выбран стандартный математический вариант, т.е. умножение выполняется первым. Поэтому мы говорим, что умножение имеет *более высокий приоритет*, чем сложение.

Порядок выполнения операций, определяемый приоритетом, можно изменить с помощью круглых скобок. Выражение, заключенное в скобки, вычисляется в первую очередь, и лишь затем выполняется операция, стоящая за скобками (так, как нас учили на уроках математики). Следовательно, если бы нам захотелось выполнить сложение до умножения, необходимо было бы записать $(2+3)*4$, что дало бы в результате 20. Если вы хотите напомнить, что умножение выполняется перед сложением, можете добавить "декоративные", функционально бесполезные круглые скобки: $2+(3*4)$.

При выполнении операций сложения и умножения* очередность их выполнения определяется достаточно просто, но, скажем, при конкатенации строк в сочетании с возведением в степень могут возникнуть проблемы. Единственно верный путь разрешить их — обратиться к официальной, не предполагающей никаких исключений, таблице приоритетов Perl-операций. (Это табл. 2.3. Отметим, что некоторые из операций нами еще не описаны; более того, они могут так и не появиться на страницах этой книги. Пусть, однако, этот факт не отпугнет вас.) Операции, аналогичные операциям C, имеют тот же приоритет, что и в этом языке.

Таблица 2.3. Ассоциативность и приоритет операций: от высокого к низкому

Ассоциативность	Операция
Слева	Операции над списками (справа налево)
Слева	-> (вызов метода, разыменование)
Неассоциативные	++ -- (автоинкремент, автодекремент)
Справа	** (возведение в степень)
Справа	! ~ \ + - (логическое отрицание, побитовое отрицание, операция ссылки, унарное сложение, унарное вычитание)
Слева	=~ !~ (совпадает, не совпадает)
Слева	* / % x (умножение, деление, деление с остатком, повторение строки)
Слева	+ - . (сложение, вычитание, конкатенация строк)
Неассоциативные	Именованные унарные операции (например, <code>chomp</code>)
Слева	& (побитовое И)
Слева	^ (побитовое ИЛИ, побитовое исключающее ИЛИ)
Слева	&& (логическое И)
Слева	(логическое ИЛИ)
Неассоциативные (не включающие или включающие граничные значения диапазоны)
Справа	? : (операция выбора, if-then-else)
Справа	= += -= *= и т.д. (присваивание и присваивание с вычислением)
Слева	, => (запятая и запятая-стрелка)
Неассоциативные	Операции над списками (слева направо)
Справа	not (логическое НЕ)
Слева	and (логическое И)
Слева	or xor (логическое ИЛИ, логическое исключающее ИЛИ)

* Вы хорошо помните алгебру? Если нет, то в повсеместном использовании круглых скобок ничего предосудительного нет.

В этой таблице каждая операция имеет более высокий приоритет, чем перечисленные за ней, и более низкий приоритет, чем перечисленные перед ней.

Операции одного уровня приоритетов разрешаются в соответствии с правилами *ассоциативности*. Как и приоритет, ассоциативность определяет порядок выполнения операций, если две операции с одинаковым приоритетом пытаются манипулировать тремя операндами:

```
2 ** 3 ** 4      # 2 ** (3 ** 4), или 2 ** 81, или около 2.41e24
72 / 12 / 3      # (72 / 12) / 3, или 6/3, или 2
30 / 6 * 3       # (30/6)*3, или 15
```

В первом случае операция `**` имеет ассоциативность справа, поэтому круглые скобки подразумеваются справа. Операции `*` и `/` имеют ассоциативность слева, поэтому круглые скобки подразумеваются слева.

Преобразование чисел в строки и обратно

Если строковое значение используется как операнд в операции с числами (например, в операции сложения), Perl автоматически преобразует эту строку в эквивалентное числовое значение, как если бы оно было введено в виде десятичного числа с плавающей запятой*. Нечисловые окончания и начальные пробельные символы игнорируются, поэтому `" 123.45fred"` (с начальным пробелом) без какого-либо предупреждения преобразуется в `123.45**`. Самый крайний из подобных случаев — когда нечто, не являющееся числом, вообще без предупреждения преобразуется в нуль (как строка `fred`, которую мы использовали здесь как число).

Если же, наоборот, там, где должно быть строковое значение, вводится числовое (например, в операции конкатенации строк), это числовое значение конвертируется в строку, которая была бы выведена на экран вместо него. Например, если вы хотите конкатенировать строку `X` с результатом умножения 4 на 5, это можно записать как

```
"X" . (4 * 5)  # то же самое, что "X" . 20, или "X20"
```

(Помните, что круглые скобки заставляют сначала выполнить `4*5`, а затем выполнять операцию конкатенации.)

Другими словами, вам не нужно (в большинстве случаев) беспокоиться о том, что указано в качестве операнда — строка или число, поскольку Perl выполняет все необходимые преобразования самостоятельно.

* Шестнадцатеричные и восьмеричные значения этому автоматическому преобразованию не подлежат. Для интерпретации таких значений следует использовать функции `hex` и `oct`.

** Если в командной строке не указана опция `-w`. В целях обеспечения безопасности всегда задавайте эту опцию.

Скалярные переменные

Имя переменной — это имя контейнера, который содержит одно или более значений. Имя переменной остается постоянным на протяжении всей программы, но значение (или значения), содержащееся в этой переменной, как правило, в ходе выполнения программы непрерывно изменяется.

Скалярная переменная содержит одно скалярное значение (представляющее собой число, строку или ссылку). Имена скалярных переменных состоят из знака доллара и буквы, за которой могут следовать еще несколько букв, цифр и знаков подчеркивания*. Строчные и прописные буквы различаются: переменная \$A и переменная \$a — разные. Все буквы, цифры и знаки подчеркивания в имени переменной имеют значение. Так, переменная

```
$a_very_long_variable_that_ends_in_1
```

отличается от переменной

```
$a_very_long_variable_that_ends_in_2
```

Имена переменных следует, как правило, подбирать так, чтобы они имели какой-то смысл в связи со значением переменной. Например, имя \$xyz123, вероятно, не очень описательно, в отличие от \$line_length.

Скалярные операции и функции

Самая распространенная операция, выполняемая над скалярной переменной — *присваивание*, которое представляет собой способ задания значения этой переменной. Операция присваивания в Perl обозначается знаком равенства (почти как в С и ФОРТРАНе). С левой стороны ставится имя переменной, с правой — присваиваемое ей значение или выражение, при помощи которого это значение вычисляется, например:

```
$a = 17;           # присвоить переменной $a значение 17
$b = $a + 3;       # присвоить $b текущее значение $a плюс 3 (20)
$b = $b * 2;       # присвоить $b значение $b, умноженное на 2 (40)
```

Обратите внимание: в последней строке переменная \$b используется дважды: один раз для получения значения (в правой части), а второй — для указания, куда поместить вычисленное выражение (в левой части). Это допустимый, надежный и довольно распространенный прием. Распространен он настолько, что через минуту мы увидим, как все это можно записать в сокращенном виде.

Возможно, вы заметили, что скалярные переменные всегда предваряются знаком \$. В shell знак \$ используют для получения значения, а при присваивании нового значения его не указывают. В Java и С этот символ

* Их количество ограничено числом 255. Надеемся, этого достаточно.

вообще опускается. Если вы постоянно присваиваете одной или нескольким переменным новые значения, то неминуемо будете делать ошибки. (Наше решение заключалось в том, чтобы прекратить писать программы на shell, *awk* и *C*, но для вас этот путь может быть неприемлем.)

Скалярное присваивание не только является операцией, его можно использовать и в качестве значения, аналогично тому как это делается в *C*. Другими словами, выражение `$a=3` имеет значение, аналогично тому как имеет некоторое значение выражение `$a+3`. Значением является та величина, которая присваивается, т.е. `$a=3` имеет значение 3. Хотя на первый взгляд это может показаться несуразным, использование присваивания как значения полезно, если вы хотите присвоить переменной промежуточное значение в выражении или просто скопировать одно значение в несколько переменных. Например:

```
$b = 4 + ($a = 3); # присвоить 3 переменной $a, затем прибавить к 4,  
                  # в результате чего $b получит значение 7  
$d = ($c = 5);    # скопировать 5 в $c, а затем и в $d  
$d = $c = 5;      # то же самое, но без круглых скобок
```

Последнее выражение работает, потому что присваивание имеет ассоциативность справа.

Операции присваивания с вычислением

Выражения типа `$a = $a + 5` (где переменная стоит по обе стороны от оператора присваивания) встречаются довольно часто, поэтому в Perl применяется сокращенный метод записи операции изменения переменной — *операция присваивания с вычислением*. Почти все двоичные операции, с помощью которых вычисляются значения, имеют соответствующую форму с добавленным знаком равенства. Например, следующие две строки эквивалентны:

```
$a = $a + 5; # без операции присваивания с вычислением  
$a += 5;    # с операцией присваивания с вычислением
```

Эквивалентны и эти строки:

```
$b = $b * 3;  
$b *= 3;
```

В каждом из вышеприведенных случаев данная операция вызывает изменение существующего значения переменной определенным способом, а не просто замену этого значения результатом вычисления какого-то нового выражения.

Другой распространенной операцией присваивания является операция конкатенации строк:

```
$str = $str . " "; # добавить пробел к $str  
$str .= " ";      # то же самое, но с операцией присваивания
```

Почти все двоичные операции, записанные таким образом, допустимы. Например, операция *возведения в степень* записывается как `**=`. Так, `$a **= 3` означает “возвести число, содержащееся в переменной `$a`, в третью степень и поместить результат обратно в `$a`”.

Как и простая операция присваивания, эти операции также могут быть использованы в качестве значения, которым является новое значение переменной. Например:

```
$a = 3;
$b = ($a += 4); # $a и $b теперь равны 7
```

Автоинкремент и автодекремент

Казалось бы, для прибавления единицы к `$a` можно просто записать `$a += 1`, но Perl идет на шаг дальше и сокращает даже эту запись. Операция `++` (*автоинкремент*) прибавляет к своему операнду единицу и возвращает инкрементированное значение, например:

```
$a += 1; # с операцией присваивания
++$a; # с префиксным автоинкрементом
$d = 17;
$e = ++$d; # и $e, и $d теперь равны 18
```

Здесь операция `++` используется как *префиксная*, т.е. знак операции стоит слева от операнда. Автоинкремент можно записывать и в *суффиксной* форме (справа от операнда). В этом случае результатом выражения является старое значение переменной, которое она имела *до* инкрементирования. Например:

```
$c = 17;
$d = $c++; # $d равно 17, а $c равно 18
```

Поскольку значение операнда изменяется, операнд должен быть скалярной переменной, а не просто выражением. Не стоит рассчитывать, что `++($a+$b)` каким-то образом стало больше, чем сумма `$a` и `$b`.

Операция автодекремента (`--`) похожа на операцию автоинкремента, но здесь не прибавляется единица, а вычитается. Как и операция автоинкремента, операция автодекремента имеет две формы — префиксную и суффиксную. Например:

```
$x = 12;
--$x; # $x теперь равно 11
$y = $x--; # $y равно 11, а $x - 10
```

Операции автоинкремента и автодекремента выполняются и над значениями с плавающей запятой. Автоинкрементирование переменной со значением 4,2 дает, как и следовало ожидать, 5,2*.

* Операция автоинкрементирования выполняется даже над строками. См. по данному вопросу книгу *Programming Perl* или map-страницу *perlop(1)*.

Функции `chop` и `chomp`

Весьма полезной иногда бывает встроенная функция `chop`. Эта функция принимает один аргумент, указываемый в круглых скобках — имя скалярной переменной — и удаляет из строкового значения этой переменной последний символ. Например:

```
$x = "hello world";  
chop($x); # $x теперь имеет значение "hello worl"
```

Обратите внимание: значение аргумента здесь меняется, отсюда и требование к наличию скалярной переменной, а не просто скалярного значения. Писать `chop('suey')`, чтобы превратить аргумент в `'sue'`, не имеет смысла, потому что места для хранения этого значения нет. Кроме того, можно ведь и просто написать: `'sue'`.

Возвращаемое значение для этой функции — отброшенный символ (в приведенном выше примере с `"hello world"` это буква `d`). Следующий код, очевидно, неверен:

```
$x = chop($x); # НЕВЕРНО: заменяет $x ее последним символом  
chop($x); # ВЕРНО: как и выше, удаляет последний символ
```

Если в функции `chop` задать пустую строку, то она ничего не сделает и ничего не возвратит, не выдавая сообщения об ошибке и вообще никак не реагируя*. Большинство операций в Perl имеют разумные граничные условия; другими словами, вы можете использовать их в пределах накладываемых ограничений и за ними, причем часто без какой-либо реакции с их стороны. Некоторые утверждают, что это один из фундаментальных недостатков Perl, а другие пишут первоклассные программы, нисколько не утруждая себя заботой о соблюдении этих ограничений. Вам решать, к какому лагерю присоединяться.

При усечении уже усеченной строки отбрасывается еще один символ. Например:

```
$a = "hello world\n";  
chop $a; # теперь $a имеет значение "hello world"  
chop $a; # оп-ля! теперь $a имеет значение "hello worl"
```

Если вы не уверены в том, есть ли в конце переменной символ новой строки, можете использовать несколько более безопасную операцию `chomp`, которая удаляет только символ новой строки**, например:

```
$a = "hello world\n";  
chomp ($a); # теперь $a имеет значение "hello world"  
chomp ($a); # ага! никаких изменений в $a не произошло
```

* Если вы не используете соответствующий здравому смыслу ключ `-w`.

** Или иное значение, которое задано переменной `$\` в качестве разделителя входных записей

Интерполяция скаляров в строках

Если строковый литерал взят в двойные кавычки, в нем необходимо выполнить *интерполяцию переменных* (помимо проверки на наличие управляющих последовательностей с обратной косой). Это значит, что строка просматривается на предмет наличия имен скалярных переменных* — а именно комбинаций знака доллара с буквами, цифрами и знаками подчеркивания. Если ссылка на переменную имеется, она заменяется текущим значением этой переменной (или пустой строкой, если значение переменной еще не присвоено). Например:

```
$a = "fred";  
$b = "some text $a";           # $b имеет значение "some text fred"  
$c = "no such variable $what"; # $c имеет значение "no such variable "
```

Текст, который заменяет переменную, не просматривается, т.е. даже при наличии в нем знаков доллара никакие дальнейшие замены не производятся:

```
$x = '$fred'; # буквально знак доллара и слово "fred"  
$y = "hey $x"; # значение — 'hey $fred'; двойной подстановки нет
```

Если необходимо предотвратить подстановку значения вместо имени переменной, необходимо либо изменить эту часть строки так, чтобы она стояла в одинарных кавычках, либо поставить перед знаком доллара обратную косую, которая отменяет его специальное значение:

```
$fred = 'hi';  
$barney = "a test of " . '$fred'; # буквально: 'a test of $fred'  
$barney2 = "a test of \$fred";    # то же самое
```

В качестве имени переменной будет взято самое длинное из возможных имен, имеющих смысл в этой части строки. Это может вызвать проблему, если вы хотите сразу же после заменяемого значения дать какой-то постоянный текст, который начинается с буквы, цифры или знака подчеркивания. Проверяя строку на предмет наличия имен переменных, Perl посчитал бы эти символы дополнительными символами имени — а как раз этого вам и не нужно. В Perl имеется разделитель имени переменной. Просто заключите имя переменной в фигурные скобки. Другой вариант — завершить эту часть строки и начать другую часть строки знаком операции конкатенации:

```
$fred = "pay"; $fredday = "wrong!";  
$barney = "It's $fredday"; # не день зарплаты payday, а "It's wrong!"  
$barney = "It's ${fred}day"; # теперь $barney получает значение "It's payday"  
$barney2 = "It's $fred"."day"; # еще один способ  
$barney3 = "It's " . $fred . "day"; # и еще один
```

* И переменных-массивов, но этот вопрос мы будем рассматривать в главе 3 *Массивы и списочные данные*.

Для изменения регистра букв, задействованных в интерполяции переменных, можно использовать специально предназначенные для этого строковые управляющие последовательности*. Например:

```
$bigfred = "\Ufred";           # $bigfred имеет значение "FRED"
$fred = "fred"; $bigfred = "\U$fred";           # то же самое
$capfred = "\u$fred";          # $capfred имеет значение "Fred"
$barney = "\LBARNEY";           # $barney теперь имеет значение "barney"
$capbarney = "\u\LBARNEY";      # $capbarney теперь имеет значение "Barney"
$bigbarney = "BARNEY"; $capbarney = "\u\L$bigbarney"; # то же самое
```

Как видите, изменяющие регистр строковые управляющие последовательности хранятся до тех пор, пока не окажут свое воздействие, поэтому, несмотря на то, что первая буква слова BARNEY не стоит непосредственно за символами \u, она становится прописной благодаря воздействию \u.

Термин “интерполяция переменных” иногда заменяют термином “интерполяция двойных кавычек”, потому что интерполяция выполняется в строках, заключенных в двойные кавычки. (А также в строках, заключенных в обратные кавычки, которые мы рассмотрим в главе 14.)

<STDIN> как скалярное значение

Если вы — типичный хакер, то, вероятно, давно уже задаетесь вопросом: а как же ввести в Perl-программу какое-либо значение? Вот самый простой способ. Каждый раз, когда там, где требуется скалярное значение, вы используете дескриптор <STDIN>, Perl читает следующую полную строку текста со *стандартного ввода* (до первого символа новой строки) и использует ее в качестве значения этого дескриптора. Термином “стандартный ввод” может обозначаться многое, но если вы не делаете в своей программе ничего необычного, это означает терминал пользователя, вызвавшего вашу программу (вероятнее всего, ваш собственный терминал). Если строки, которую можно было бы прочитать, еще нет (типичный случай, если только вы заранее не набрали полную строку), Perl-программа останавливается и ждет, пока вы не введете какие-нибудь символы и вслед за ними символ перехода на новую строку.

В конце строкового значения дескриптора <STDIN> обычно стоит символ новой строки. Чаше всего от этого символа нужно избавляться сразу же (ведь между hello и hello\n — большая разница). Здесь и приходит на помощь знакомая нам функция `chomp`. Типичная последовательность ввода выглядит примерно так:

```
$a = <STDIN>;           # читаем текст
chomp($a);              # избавляемся от надоедливого символа новой строки
```

* Вы, возможно, придете к выводу, что легче воспользоваться функциями `uc`, `ucfirst`, `lc` и `lcfirst`.

Общепринятое сокращение этих двух строк выглядит так:

```
chomp($a = <STDIN>);
```

Присваивание внутри круглых скобок продолжает относиться к `$a` даже после того, как этой переменной уже присвоено значение. Таким образом, функция `chomp` оперирует с переменной `$a`. (Это вообще справедливо для операции присваивания; присваиваемое выражение можно использовать везде, где необходима переменная, а действия относятся к переменной, стоящей слева от знака равенства.)

Вывод с помощью функции `print`

Итак, мы вводим значения с помощью дескриптора `<STDIN>`. А как их вывести из программы? С помощью функции `print`. Эта функция принимает значения, стоящие в скобках, и выдает их без всяких украшений на стандартный вывод. Стандартный вывод — это опять-таки ваш терминал (если вы не делаете в программе ничего необычного). Например:

```
print("hello world\n"); # выводится hello world с символом новой строки
print "hello world\n"; # то же самое
```

Обратите внимание на второй пример, где показана форма функции `print` без круглых скобок. Использовать скобки или нет — это, главным образом, вопрос стиля и скорости набора, но есть несколько случаев, где скобки обязательны во избежание двусмысленности.

Мы увидим, что для функции `print` можно задавать *список* значений, но поскольку про списки мы еще не говорили, отложим этот вопрос до главы 6.

Значение `undef`

Что будет, если использовать скалярную переменную до того, как ей присвоено значение? Ничего серьезного, в общем-то, не произойдет. До присваивания значения переменные имеют значение `undef`. При использовании в качестве числа это значение выглядит как нуль, а при использовании в качестве строки — как пустая строка нулевой длины. Впрочем, при работе с ключом `-w` вы получите предупреждение — это хороший способ выявления ошибок программирования.

Многие операции возвращают `undef`, когда их аргументы выходят за пределы диапазона или не имеют смысла. Если вы не делаете ничего особенного, вы получите в подобном случае нуль или пустую строку без серьезных последствий. На практике это едва ли представляет собой проблему.

Одна из уже знакомых нам операций, которая в определенных обстоятельствах возвращает `undef` — это операция `<STDIN>`. Обычно она возвращает следующую из прочитанных строк; если же строк для чтения больше

нет (например, когда вы нажали на терминале клавиши [Ctrl+D] или когда в файле больше нет данных), то <STDIN> возвращает значение undef. В главе 6 мы посмотрим, как осуществить проверку в данной ситуации и выполнить специальные действия, если данных для чтения больше нет.

Упражнения

Ответы к упражнениям приведены в приложении А.

1. Напишите программу, которая вычисляет длину окружности с радиусом 12, 5. Длина окружности равна 2π (около $2 * 3,141592654$) радиусам.
2. Модифицируйте программу из предыдущего упражнения так, чтобы она запрашивала и принимала значение радиуса окружности у пользователя.
3. Напишите программу, которая запрашивает и считывает два числа, после чего выводит на экран результат перемножения этих чисел.
4. Напишите программу, которая считывает строку и число и выводит на экран строку столько раз, сколько задано числом, причем каждый раз с новой строки. (Совет: используйте оператор x.)

В этой главе:

- *Список и массив*
- *Литеральное представление*
- *Переменные*
- *Операции над массивами и функции обработки массивов*
- *Скалярный и списочный контексты*
- *<STDIN> как массив*
- *Интерполяция массивов*
- *Упражнения*

Массивы и списочные данные

Список и массив

Список — это упорядоченные скалярные данные. *Массив* — переменная, которая содержит список. Каждый *элемент* массива — это отдельная скалярная переменная с независимым скалярным значением. Значения в списке упорядочены, т.е. расставлены в определенной последовательности, например от младшего элемента к старшему.

Массивы могут иметь любое число элементов. Минимально возможный массив не имеет элементов вообще, тогда как максимально возможный может заполнять всю наличную память. Это еще одно подтверждение принятой в Perl стратегии “отсутствия ненужных ограничений”.

Литеральное представление

Списочный литерал (способ представления значения списка в программе) состоит из значений, отделенных друг от друга запятыми и заключенными в круглые скобки. Эти значения образуют элементы списка. Например:

```
(1,2,3)      # массив из трех значений 1, 2 и 3
("fred",4.5) # два значения — "fred" и 4,5
```

Элементы списка не обязательно должны быть константами. Это могут быть выражения, которые вычисляются при каждом использовании литерала. Например:

```
($a, 17;      # два значения: текущее значение переменной $a и 17
($b+$c,$d+$e) # два значения
```

Пустой список (список, не содержащий элементов) представляется пустой парой круглых скобок:

```
() # пустой список (нуль элементов)
```

Элемент списочного литерала может включать *операцию конструктора списка*. Это два скалярных значения, разделенных двумя точками. Данная операция создает список значений, начиная с левого скалярного значения и кончая правым скалярным значением, с шагом 1. Например:

```
(1 .. 5)           # то же самое, что (1, 2, 3, 4, 5)
(1.2 .. 5.2)       # то же самое, что (1.2, 2.2, 3.2, 4.2, 5.2)
(2 .. 6,10,12)     # то же самое, что (2,3,4,5,6,10,12)
($a .. $b)         # диапазон, заданный текущими значениями переменных $a и $b
```

Если правый скаляр меньше левого, то список будет пустым, так как в обратном направлении отсчет вести нельзя. Если последнее значение не соответствует целому числу шагов, то список заканчивается там, где приращение на единицу привело бы к появлению числа, не принадлежащего заданному диапазону:

```
(1.3 .. 6.1)      # то же самое, что (1.3,2.3,3.3,4.3,5.3)
```

Списочные литералы с множеством коротких текстовых строк со всеми этими кавычками и запятыми выглядят не очень привлекательно:

```
@a = ("fred","barney","betty","wilma"); # уф-ф!
```

Поэтому предусмотрена функция заключения в кавычки, которая создает список из разделенных пробелами слов, заключенных в круглые скобки*:

```
@a = qw(fred barney betty wilma); # так-то лучше!
@a = qw(
    fred
    barney
    betty
    wilma
); # то же самое
```

Одно из возможных применений списочного литерала — в качестве аргумента функции `print`, с которой мы уже знакомы. Элементы списка выводятся на экран без промежуточных пробельных символов:

```
print("The answer is ",$a,"\n"); # трехэлементный списочный литерал
```

Этот оператор выводит на экран слова `The answer is`, затем пробел, значение переменной `$a` и символ новой строки. Другие способы использования списочных литералов мы рассмотрим позднее.

* Как и в функциях сопоставления с образцом, о которых мы узнаем позже, в качестве разделителя здесь вместо круглых скобок можно использовать любой символ, не относящийся к числу пробельных символов, букв и цифр.

Переменные

Переменная-массив содержит одно значение в виде списка (нуль или более скалярных значений). Имена переменных-массивов похожи на имена скалярных переменных. Единственное отличие состоит в первом символе — это не знак доллара (\$), а знак @. Например:

```
@fred # массив-переменная @fred
@A_Very_Long_Array_Variable_Name
@A_Very_Long_Array_Variable_Name_that_is_different
```

Отметим здесь, что переменная-массив @fred не имеет никакого отношения к скалярной переменной \$fred. Для разных типов переменных Perl предусматривает отдельные пространства имен.

Переменная-массив, которой еще не присвоено значение, имеет значение (), т.е. пустой список.

Выражение может ссылаться на переменные-массивы в целом, а также проверять и изменять отдельные элементы таких массивов.

Операции над массивами и функции обработки массивов

Операции над массивами и функции обработки массивов манипулируют целыми массивами. Некоторые из них возвращают список, который затем можно использовать как значение в другой функции обработки массивов или присвоить переменной-массиву.

Присваивание

Вероятно, самая важная операция, проводимая над массивами — операция присваивания, посредством которой переменной-массиву присваивается значение. Эта операция, как и скалярная операция присваивания, обозначается знаком равенства. Perl определяет тип присваивания (скалярное или для массива), анализируя, какой переменной присваивается значение — скалярной или массиву. Например:

```
@fred = (1,2,3); # массив fred получает трехэлементное литеральное значение
@barney = @fred; # теперь оно копируется в @barney
```

Если переменной-массиву присваивается скалярное значение, оно становится единственным элементом этого массива:

```
@huh = 1; # 1 автоматически становится списком (1)
```

Имена переменных-массивов могут входить в списочный литерал. При вычислении значений такого списка Perl заменяет имена массивов текущими значениями, например:

```
@fred = qw(one two);
@barney = (4,5,@fred,6,7); # @barney превращается в (4,5,"one","two",6,7)
@barney = (8,@barney);     # в начале списка элементов @barney ставится 8
                           # и "last" в конце.
@barney = (@barney,"last"); # @barney превращается в
                           # (8,4,5,"one","two",6,7,"last")
```

Отметим, что вставленные подобным образом элементы массива находятся на том же уровне иерархии, что и остальная часть литерала: список не может содержать другой список в качестве элемента*.

Если списочный литерал содержит только ссылки на переменные (а не выражения), то этот литерал также можно рассматривать как переменную. Другими словами, такой списочный литерал можно использовать в левой части операции присваивания. Каждая скалярная переменная в списочном литерале принимает соответствующее значение из списка, стоящего в правой части операции присваивания. Например:

```
($a,$b,$c) = (1,2,3); # присвоить 1 переменной $a, 2 — переменной $b,
                       # 3 — переменной $c
($a,$b) = ($b,$a);    # поменять местами $a и $b
($d,@fred) = ($a,$b,$c) # присвоить $d значение $a, а @fred — значение ($b,$c)
($e,@fred) = @fred;    # переместить первый элемент массива @fred
                       # в переменную $e. В результате @fred = ($c),
                       # а $e = $b
```

Если число присваиваемых элементов не соответствует числу переменных, то лишние значения (стоящие справа от знака равенства) просто отбрасываются, а все лишние переменные (слева от знака равенства) получают значение undef.

Переменная-массив, входящая в литеральный список, должна стоять в нем последней, потому что переменные-массивы очень “прожорливы” и поглощают все оставшиеся значения. (Конечно, после нее можно поставить и другие переменные, но всем им будет присвоено значение undef.)

Если переменная-массив присваивается скалярной переменной, то присваиваемое число является *размером* массива, например:

```
@fred = (4,5,6); # инициализация массива @fred
$a = @fred;      # $a получает значение 3, текущий размер массива @fred
```

Размер возвращается и в том случае, если имя переменной-массива используется там, где должно быть скалярное значение. (В разделе “Скалярный и списочный контексты” мы увидим, что это называется использованием имени

* Хотя ссылка на список и может быть элементом списка, на самом деле это не означает использование списка в качестве элемента другого списка. Впрочем, работает это почти так же, что позволяет создавать многомерные массивы. См главу 4 книги *Programming Perl* и map-страницу `perllo(1)`

массива в *скалярном контексте*.) Например, чтобы получить число на единицу меньше, чем размер массива, можно использовать `@fred-1`, так как скалярная операция вычитания требует наличия скаляров в обеих частях. Обратите внимание на следующий пример:

```
$a = @fred;      # переменной $a присваивается размер массива @fred
($a) = @fred;    # переменной $a присваивается первый элемент @fred
```

Первое присваивание — скалярное, и массив `@fred` рассматривается как скаляр, поэтому значение переменной `$a` будет равно размеру массива. Второе присваивание — для массива (несмотря на то, что требуется всего одно значение), поэтому переменной `$a` в качестве значения присваивается первый элемент массива `@fred`, а остальные элементы просто отбрасываются.

В результате выполнения присваивания для массива мы получаем значение, представляющее собой список. Это позволяет делать “каскадирование”. Например:

```
@fred = (@barney = (2,3,4)); # @fred и @barney получают значения (2,3,4)
@fred = @barney = (2,3,4);   # то же самое
```

Обращение к элементам массива

До сих пор мы рассматривали массив в целом, добавляя и удаляя значения с помощью присваивания для массива. Многие полезные программы так и построены — с использованием массивов, но без обращения к их элементам. Perl, однако, предоставляет и традиционный способ обращения к элементам массива по их числовым индексам.

Элементы массива нумеруются последовательными целыми числами с шагом 1, начиная с нуля*. Первый элемент массива `@fred` обозначается как `$fred[0]`. Обратите внимание: при ссылке на элемент вместо знака `@` в имени массива используется знак `$`. Это объясняется тем, что обращение к элементу массива идентифицирует скалярную переменную (часть массива), которой в результате может быть присвоено значение или которая используется в выражении, например:

```
@fred = (7,8,9);
$b = $fred[0];    # присвоить $b значение 7 (первый элемент массива @fred)
$fred[0] = 5;     # теперь @fred = (5,8,9)
```

Точно так же можно обращаться к другим элементам:

```
$c = $fred[1];    # присвоить $c значение 8
$fred[2]++;       # инкрементировать третий элемент массива @fred
$fred[1] += 4;    # прибавить 4 ко второму элементу
($fred[0], $fred[1]) = ($fred[1], $fred[0]);
                  # поменять местами первые два элемента
```

* Значение индекса первого элемента можно изменить на что-нибудь другое (например, на “1”). Это, однако, повлечет за собой очень серьезные последствия: может запутать тех, кто будет сопровождать ваш код, и повредить программы, которые вы заимствуете у других. По этой причине настоятельно рекомендуем считать такую возможность отсутствующей.

Обращение к списку элементов одного массива (как в последнем примере) называется *срезы** и встречается достаточно часто, поэтому для него есть специальное обозначение.

```
@fred[0,1]           # то же, что и ($fred[0],$fred[1])
@fred[0,1] = @fred[1,0] # поменять местами первые два элемента
@fred[0,1,2] = @fred[1,1,1] # сделать все три элемента такими, как второй
@fred[1,2] = (9,10);      # заменить последние два значения на 9 и 10
```

Обратите внимание: в этом срезе используется не \$, а @. Это вызвано тем, что вы создаете переменную-массив (выбирая для этого часть массива), а не скалярную переменную (обращаясь к одному элементу массива).

Срезы работают также с литеральными списками и вообще с любой функцией, которая возвращает список:

```
@who = (qw(fred barney betty wilma))[2,3];
      # как @x = qw(fred barney betty wilma); @who = @x[2,3]
```

Значения индексов в этих примерах — литеральные целые, но индекс может быть и любым выражением, которое возвращает число, используемое затем для выбора соответствующего элемента:

```
@fred = (7,8,9);
$a = 2;
$b = $fred[$a];      # как $fred[2], или 9
$c = $fred[$a-1];    # $c получает значение $fred[1], или 8
($c) = (7,8,9)[$a-1]; # то же самое, но с помощью среза
```

Таким образом, обращение к массивам в Perl-программах может производиться так же, как во многих других языках программирования.

Идея использования выражения в качестве индекса применима и к срезам. Следует помнить, однако, что индекс среза — список значений, поэтому выражение представляет собой выражение-массив, а не скаляр.

```
@fred = (7,8,9);           # как в предыдущем примере
@barney = (2,1,0);
@backfred = @fred[@barney];
# то же, что и @fred[2,1,0], или ($fred[2],$fred[1],$fred[0]), или (9,8,7)
```

Если обратиться к элементу, находящемуся за пределами массива (т.е. задав индекс меньше нуля или больше индекса последнего элемента), то возвращается значение `undef`. Например:

```
@fred = (1,2,3);
$barney = $fred[7];      # $barney теперь имеет значение undef
```

* Это перевод английского термина *slice*, использованный в данной книге. Более точно суть операции можно было бы выразить термином *вырезка*, но мы остановились на слове *срез*, чтобы не вызывать гастрономических ассоциаций у любителей мясных блюд — *Прим ред.*

Присваивание значение элементу, находящемуся за пределами текущего массива, автоматически расширяет его (с присваиванием всем промежуточным значениям, если таковые имеются, значения undef). Например:

```
@fred = (1,2,3);  
fred[3] = "hi";      # @fred теперь имеет значение (1,2,3,"hi")  
$fred[6] = "ho";     # @fred теперь имеет значение  
                    # (1,2,3,"hi",undef,undef,"ho")
```

Присваивание значения элементу массива с индексом меньше нуля является грубой ошибкой, потому что происходит такое, скорее всего, из-за Очень Плохого Стиля Программирования.

Для получения значения индекса последнего элемента массива @fred можно использовать операцию \$#fred. Можно даже задать это значение, чтобы изменить размер массива @fred, но это, как правило, не нужно, потому что размер массива увеличивается и уменьшается автоматически.

Использование отрицательного индекса означает, что следует вести обратный отсчет от последнего элемента массива. Так, последний элемент массива можно получить, указав индекс -1. Предпоследний элемент будет иметь индекс -2 и т.д. Например:

```
@fred = ("fred", "wilma", "pebbles", "dino");  
print $fred[-1];      # выводит "dino"  
print $#fred;         # выводит 3  
print $fred[$#fred];  # выводит "dino"
```

Функции *push* и *pop*

Одним из распространенных вариантов использования массива является создание стека данных, где новые значения вводятся и удаляются с правой стороны списка. Эти операции применяются довольно часто, поэтому для них предусмотрены специальные функции:

```
push(@mylist,$newvalue);  # означает @mylist = (@mylist,$newvalue)  
$oldvalue = pop($mylist); # удаляет последний элемент из @mylist
```

Если в функцию pop введен пустой список, она возвращает undef, не выдавая, в соответствии с принятым в Perl этикетом, никакого предупреждающего сообщения.

Функция push также принимает список значений, подлежащих помещению в стек. Эти значения вводятся в конец списка. Например:

```
@mylist = (1,2,3);  
push(@mylist,4,5,6);  # @mylist = (1,2,3,4,5,6)
```

Отметим, что первый аргумент должен быть именем переменной-массива, потому что для литеральных списков функции push и pop смысла не имеют.

Функции *shift* и *unshift*

Функции `push` и `pop` действуют в “правой” части списка (части со старшими индексами). Функции `unshift` и `shift` выполняют соответствующие действия в “левой” части списка (части с младшими индексами). Вот несколько примеров:

```
unshift(@fred,$a);           # соответствует @fred = ($a,@fred);
unshift(@fred,$a,$b,$c);     # соответствует @fred = ($a,$b,$c,@fred);
$x = shift(@fred);           # соответствует ($x,@fred) = @fred;
                              # с реальными значениями
@fred = (5,6,7);
unshift(@fred,2,3,4);        # @fred теперь имеет значение (2,3,4,5,6,7)
$x = shift(@fred);
                              # $x получает значение 2, @fred теперь имеет значение (3,4,5,6,7)
```

Как и функция `pop`, функция `shift`, если в нее ввести пустую переменную-массив, возвращает значение `undef`.

Функция *reverse*

Функция `reverse` изменяет порядок следования элементов аргумента на противоположный и возвращает список-результат. Например:

```
@a = (7,8,9);
@b = reverse(@a);           # присваивает @b значение (9,8,7)
@b = reverse(7,8,9);       # делает то же самое
```

Обратите внимание: список-аргумент не изменяется, так как функция `reverse` работает с копией. Если вы хотите изменить порядок элементов “на месте”, список-аргумент следует затем присвоить той же переменной:

```
@b = reverse(@b);          # присвоить массиву @b его же значения,
                              # но расположить его элементы в обратном порядке
```

Функция *sort*

Функция `sort` сортирует аргументы так, как будто это отдельные строки, в порядке возрастания их кодов ASCII. Она возвращает отсортированный список, не изменяя оригинал. Например:

```
@x = sort("small","medium","large");
                              # @x получает значение "large", "medium", "small"
@y = (1,2,4,8,16,32,64);
@y = sort (@y);              # @y получает значение 1, 16, 2, 32, 4, 64, 8
```

Отметим, что сортировка чисел производится не по их числовым значениям, а по их строковым представлениям (1, 16, 2, 32 и т.д.). Изучив главу 15, вы научитесь выполнять сортировку по числовым значениям, по убыванию, по третьему символу строки и вообще каким угодно методом.

Функция *chomp*

Функция `chomp` работает не только со скалярной переменной, но и с массивом. У каждого элемента массива удаляется последний пробельный символ. Это удобно, когда вы, прочитав несколько строк как список отдельных элементов массива, хотите одновременно убрать из всех строк символы новой строки. Например:

```
@stuff = ("hello\n", "world\n", "happy days");  
chomp(@stuff); # @stuff теперь имеет значение ("hello", "world", "happy day")
```

Скалярный и списочный контексты

Как видите, каждая операция и функция предназначена для работы с определенной комбинацией скаляров или списков и возвращает скаляр или список. Если операция или функция рассчитывает на получение скалярного операнда, то мы говорим, что операнд или аргумент обрабатывается в *скалярном контексте*. Аналогичным образом, если операнд или аргумент должен быть списочным значением, мы говорим, что он обрабатывается в *списочном контексте*.

Как правило, это особого значения не имеет, но иногда в разных контекстах можно получить совершенно разные результаты. Например, в списочном контексте `@fred` возвращает содержимое массива `@fred`, а в скалярном — размер этого массива. При описании операций и функций мы упоминаем эти тонкости.

Скалярное значение, используемое в списочном контексте, превращается в одноэлементный массив.

<STDIN> как массив

Одна из ранее изученных нами операций, которая в списочном контексте возвращает иное значение, чем в скалярном, — `<STDIN>`. Как упоминалось выше, `<STDIN>` в скалярном контексте возвращает следующую введенную строку. В списочном же контексте эта операция возвращает все строки, оставшиеся до конца файла. Каждая строка при этом возвращается как отдельный элемент списка, например:

```
$a = <STDIN>; # читать стандартный ввод в списочном контексте
```

Если пользователь, выполняющий программу, введет три строки и нажмет `[Ctrl+D]`* (чтобы обозначить конец файла), массив будет состоять из трех элементов. Каждый из них является строкой, заканчивающейся символом новой строки, и соответствует введенной пользователем строке.

* В некоторых системах конец файла обозначается нажатием клавиш `[Ctrl+Z]`, а в других эта комбинация служит для приостановки выполняемого процесса.

Интерполяция массивов

Как и скаляры, значения массивов могут интерполироваться в строку, заключенную в двойные кавычки. Каждый элемент массива заменяется его значением, например:

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

Отметим, что индексное выражение вычисляется как обычное, как будто оно находится вне строки, т.е. оно предварительно не интерполируется.

Если вы хотите поставить после простой ссылки на скалярную переменную литеральную левую квадратную скобку, нужно выделить эту скобку так, чтобы она не считалась частью массива:

```
@fred = ("hello","dolly"); # присвоить массиву @fred значение для проверки
$fred = "right";
                                # мы пытаемся сказать "this is right[1]"
$x = "this is $fred[1]";      # неправильно, дает "this is dolly"
$x = "this is ${fred}[1]";    # правильно (защищено фигурными скобками)
$x = "this is $fred"."[1]";   # правильно (другая строка)
$x = "this is $fred\[1]";     # правильно (скобку защищает обратная косая)
```

Аналогичным образом может интерполироваться список значений переменной-массива. Самая простая интерполяция — интерполяция всего массива, обозначенного именем (с начальным символом @). В этом случае элементы интерполируются по очереди, с разделением их пробелами, например:

```
@fred = ("a","bb","ccc",1,2,3);
$all = "Now for @fred here!";
        # $all получает значение "Now for a bb ccc 1 2 3 here!"
```

Можно также выбрать часть массива с помощью среза:

```
@fred = ("a","bb","ccc",1,2,3);
$all = "Now for @fred[2,3] here!";
        # $all получает значение "Now for ccc 1 here!"
$all = "Now for @fred[@fred[4,5]] here!"; # то же самое
```

Опять-таки, если вы хотите поставить после ссылки на имя массива литеральную левую квадратную скобку, а не индексное выражение, можете использовать любой из описанных выше механизмов.

Упражнения

Ответы к упражнениям приведены в приложении А.

1. Напишите программу, которая читает список строковых значений, стоящих в отдельных строках, и выводит этот список в обратном порядке. Если вы читаете список на экране, то вам, вероятно, нужно будет выделить конец списка, нажав комбинацию клавиш “конец файла” (в UNIX или Plan 9 это, наверное, [Ctrl+D], а в других системах чаще всего [Ctrl+Z]).
2. Напишите программу, которая читает число, затем список строковых значений (находящихся в отдельных строках), после чего выводит одну из строк списка в соответствии с указанным числом.
3. Напишите программу, которая читает список строк, а затем выбирает и выводит на экран случайную строку из этого списка. Чтобы выбрать случайный элемент массива @somearray, поместите в начало программы функцию

```
srand;
```

(она инициализирует генератор случайных чисел), а затем используйте

```
rand(@somearray)
```

там, где требуется случайное значение, меньшее размера массива @somearray.

В этой главе:

- *Блоки операторов*
- *Оператор if/unless*
- *Оператор while/until*
- *Оператор for*
- *Оператор foreach*
- *Упражнения*

Управляющие структуры

Блоки операторов

Блок операторов — это последовательность операторов, заключенная в парные фигурные скобки. Блок операторов выглядит следующим образом:

```
{
    первый_оператор;
    второй_оператор;
    третий_оператор;
    ...
    последний_оператор;
}
```

Perl выполняет операторы по очереди, начиная с первого и кончая последним. (Позднее вы узнаете о том, как можно изменять порядок выполнения в блоке, но пока достаточно и этого.)

Синтаксически блок операторов принимается вместо любого одиночного оператора, но обратное не верно.

Завершающая точка с запятой, стоящая за последним оператором, не обязательна. Таким образом, вы можете разговаривать на языке Perl с С-акцентом (точка с запятой присутствует) или с паскалевским акцентом (точка с запятой отсутствует). Чтобы облегчить последующее добавление операторов, мы обычно рекомендуем опускать точку с запятой лишь в том случае, если блок занимает целую строку. Сравните примеры этих стилей в следующих двух блоках `if`:

```
if ($ready) { $hungry++ }
if ($tired) {
    $sleepy = ($hungry + 1) * 2;
}
```


Оператор *if/unless*

Следующей по сложности управляющей структурой является оператор *if*. Эта конструкция состоит из управляющего выражения (проверяемого на истинность) и блока. В ней также может быть часть, начинающаяся оператором *else*, за которой следует еще один блок операторов. Другими словами, все это выглядит так:

```
if (выражение) {
    оператор_1;
    оператор_2;
    оператор_3;
} else {
    оператор_1;
    оператор_2;
    оператор_3;
}
```

(Если вы любите программировать на С и Java, то для вас должна быть вполне очевидной обязательность фигурных скобок. Они устраняют необходимость в применении правила “путающего зависшего *else*”).

Во время выполнения Perl-программы вычисляется управляющее выражение. Если оно истинно, то выполняется первый блок операторов в приведенном выше примере. Если выражение ложно, то выполняется второй блок.

Что же такое “истина” и “ложь”? В Perl эти правила несколько странны, но, тем не менее, дают ожидаемые результаты. Управляющее выражение вычисляется как *строковая* величина в скалярном контексте (если это уже строка, ничего не изменяется, а если это число, то оно преобразуется в строку*). Если данная строка либо пуста (т.е. имеет нулевую длину), либо состоит из одного символа “0” (цифры ноль), то значение выражения — “ложь”. Все остальное автоматически дает значение “истина”. Почему же в Perl столь забавные правила? А потому, что это облегчает условный переход не только по нулю (в противоположность ненулевому числу), но и по пустой (в противоположность непустой) строке, причем без необходимости создания двух версий интерпретируемых значений “истина” и “ложь”. Вот несколько примеров интерпретации этих значений.

```
0          # преобразуется в "0", поэтому "ложь"
1-1        # дает в результате 0, затем преобразуется в "0", поэтому "ложь"
1          # преобразуется в "1", поэтому "истина"
""         # пустая строка, поэтому "ложь"
"1"        # не "" или "0", поэтому "истина"
"00"       # не "" или "0", поэтому "истина" (это странно, поэтому будьте настороже)
"0.000"    # "истина" — будьте внимательны, по той же причине
undef      # дает в результате "", поэтому "ложь"
```

* Внутренне все трактуется несколько иначе, но внешне все выполняется так, будто именно это и происходит.

Таким образом, интерпретация значений как истинных или ложных достаточно интуитивна, но пусть это вас не пугает.

Вот пример полного оператора `if`:

```
print "how old are you? ";
$a = <STDIN>;
chomp($a) ;
if ($a < 18) {
    print "So, you're not old enough to vote, eh?\n";
} else {
    print "Old enough! Cool! So go vote! \n";
    $voter++; # count the voters for later
}
```

Блок `else` можно опустить, оставив только часть, касающуюся `then`:

```
print "how old are you? ";
$a = <STDIN>;
chomp($a) ;
if ($a < 18) {
    print "So, you're not old enough to vote, eh?\n";
}
```

Иногда бывает удобно часть “`then`” опустить и оставить только `else`, потому что более естественно сказать “сделайте то, если это ложь”, нежели “сделайте то, если это — не истина”. В Perl этот вопрос решается с помощью оператора `unless`:

```
print "how old are you? ";
$a = <STDIN>;
chomp($a) ;
unless ($a < 18) {
    print "Old enough! Cool! So go vote!\n";
    $voter++;
}
```

Заменить `if` на `unless` — это все равно что сказать “Если управляющее выражение ложно, сделать...” (Оператор `unless` может содержать блок `else`, как и оператор `if`.)

Если у вас больше двух возможных вариантов, введите в оператор `if` ветвь `elsif`, например:

```
if (выражение_один) {
    оператор_1_при_истине_один;
    оператор_2_при_истине_один;
    оператор_3_при_истине_один;
} elsif (выражение_два) {
    оператор_1_при_истине_два;
    оператор_2_при_истине_два;
    оператор_3_при_истине_два;
} elsif (выражение_три) {
    оператор_1_при_истине_три;
    оператор_2_при_истине_три;
    оператор_3_при_истине_три;
}
```

```

} else {
    оператор_1_при_всех_ложных;
    оператор_2_при_всех_ложных;
    оператор_3_при_всех_ложных;
}

```

Все управляющие выражения вычисляются по очереди. Если какое-либо выражение истинно, то выполняется соответствующая ветвь, а все остальные управляющие выражения и соответствующие блоки операторов пропускаются. Если все выражения ложны, то выполняется ветвь `else` (если таковая имеется). Присутствие блока `else` не обязательно, но он никогда не помешает. В программе может быть столько ветвей `elsif`, сколько вам необходимо.

Оператор *while/until*

Ни один язык программирования не был бы полным без какой-нибудь формы организации цикла* (повторяющегося выполнения блока операторов). Perl может организовать цикл с помощью оператора `while`:

```

while (выражение) {
    оператор_1;
    оператор_2;
    оператор_3;
}

```

Чтобы выполнить оператор `while`, Perl вычисляет управляющее выражение (в данном примере — *выражение*). Если полученное значение — “истина” (по принятым в Perl правилам установления истинности), то один раз вычисляется тело оператора `while`. Это повторяется до тех пор, пока управляющее выражение не станет ложным. Тогда Perl переходит к оператору, следующему после цикла `while`. Например:

```

print "how old are you? " ;
$a = <STDIN>;
chomp($a);
while ($a > 0) {
    print "At one time, you were $a years old.\n";
    $a--;
}

```

Иногда легче сказать “до тех пор, пока что-то не станет истинным”, чем “пока не это — истина”. Для этого случая у Perl тоже есть ответ. Требуемый эффект дает замена `while` на `until`:

```

until (выражение) {
    оператор_1;
    оператор_2;
    оператор_3;
}

```

* Вот почему HTML — не язык программирования.

Обратите внимание на то, что в обеих формах операторы тела цикла полностью пропускаются, если при анализе управляющего выражения выясняется, что цикл должен быть завершен. Например, если в приведенном выше фрагменте программы пользователь введет возраст меньше нуля, Perl пропустит тело цикла.

Вполне может случиться так, что управляющее выражение не даст циклу завершиться. Это абсолютно допустимо, а иногда и желательно, поэтому ошибкой не считается. Например, вы хотите, чтобы цикл повторялся все время, пока у вас нет ошибок, а после цикла ставите какой-то код обработки ошибок. Эту схему можно использовать для программы-демона, которая должна работать до тех пор, пока система не откажет.

Оператор `do {} while/until`

Оператор `while/until`, который вы видели в предыдущем разделе, проверяет условие в начале каждого цикла, до входа в него. Если результат проверки условия — “ложь”, цикл не будет выполнен вообще.

Иногда возникает необходимость проверять условие не в начале, а в конце цикла. Для этого в Perl есть оператор `do {} while`, который очень похож на обычный оператор `while*`, за исключением того, что он проверяет выражение только после однократного выполнения цикла.

```
do {  
    оператор_1;  
    оператор_2;  
    оператор_3;  
} while выражение;
```

Perl выполняет операторы блока `do`. Дойдя до конца, он вычисляет выражение на предмет истинности. Если выражение ложно, цикл завершается. Если выражение истинно, весь блок выполняется еще раз, а затем выражение проверяется вновь.

Как и в обычном цикле `while`, условие проверки можно инвертировать, заменив `do {} while` на `do {} until`. Выражение все равно проверяется в конце цикла, но на обратное условие. В некоторых случаях, особенно сложных, такой способ записи условия представляется более естественным.

```
$stops = 0;  
do {  
    $stops++;  
    print "Next stop? " ;  
    chomp($location = <STDIN>) ;  
} until $stops > 5 || $location eq 'home';
```

* Ну, не совсем чтобы очень похож; управляющие директивы цикла, описанные в главе 9, в такой форме не работают.

Оператор *for*

Еще одна конструкция Perl, предназначенная для организации цикла, — оператор `for`, который выглядит подозрительно похоже на оператор `for` языков C и Java и работает примерно так же. Вот он:

```
for ( начальное_выражение, проверочное_выражение, возобновляющее_выражение ) {  
    оператор_1,  
    оператор_2,  
    оператор_3,  
}
```

Если преобразовать этот оператор в те формы, которые мы рассмотрели раньше, то он станет таким:

```
начальное_выражение,  
while (проверочное_выражение) {  
    оператор_1,  
    оператор_2,  
    оператор_3,  
    возобновляющее_выражение,  
}
```

В любом случае сначала вычисляется начальное выражение. Это выражение, как правило, присваивает начальное значение переменной цикла, но никаких ограничений относительно того, что оно может содержать, не существует, оно даже может быть пустым (и ничего не делать). Затем вычисляется проверочное выражение. Если полученное значение истинно, выполняется тело цикла, а затем вычисляется возобновляющее выражение (как правило, используемое для инкрементирования переменной цикла — но не только для этого). Затем Perl повторно вычисляет проверочное выражение, повторяя необходимые действия.

Следующий фрагмент программы предназначен для вывода на экран чисел от 1 до 10, после каждого из которых следует пробел.

```
for ($i = 1, $i <= 10, $i++) {  
    print $i " ",
```

Сначала переменная `$i` устанавливается в 1. Затем эта переменная сравнивается с числом 10. Поскольку она меньше или равна 10, то выполняется тело цикла (один оператор `print`), а затем вычисляется возобновляющее выражение (`$i++`), в результате чего значение `$i` изменяется на 2. Поскольку эта переменная все еще меньше или равна 10, процесс повторяется до тех пор, пока в последней итерации значение 10 в `$i` не изменится на 11. Поскольку переменная `$i` уже не меньше и не равна 10, цикл завершается (при этом `$i` становится равным 11).

Оператор *foreach*

Еще одна циклическая конструкция — оператор `foreach`. Этот оператор получает список значений и присваивает их по очереди скалярной переменной, выполняя с каждым последующим присваиванием блок кода. Выглядит это так:

```
foreach $i (@список) {  
    оператор_1;  
    оператор_2;  
    оператор_3;  
}
```

В отличие от C-shell, в Perl исходное значение этой скалярной переменной при выходе из цикла автоматически восстанавливается; другими словами, эта скалярная переменная локальна для данного цикла.

Вот пример использования оператора `foreach`:

```
@a = (1,2,3,4,5);  
foreach $b (reverse @a) {  
    print $b;  
}
```

Эта программа выводит на экран 54321. Отметим, что список, используемый в операторе `foreach`, может быть произвольным списочным литералом, а не просто переменной-массивом. (Это справедливо для всех конструкций Perl, которым требуется список.)

Имя скалярной переменной можно опустить. В этом случае Perl будет действовать так, как будто вы указали имя переменной `$_`. Вы увидите, что переменная `$_` используется по умолчанию во многих операциях языка Perl, поэтому ее можно рассматривать как неявную временную переменную. (Все операции, в которых по умолчанию используется `$_`, могут использовать и обычную скалярную переменную.) Например, функция `print` выводит значение переменной `$_`, если другие значения не указаны, поэтому следующий пример дает такой же результат, как и предыдущий:

```
@a = (1,2,3,4,5);  
foreach (reverse @a) {  
  
    print ;  
}
```

Видите, насколько использование неявной переменной `$_` все упрощает? Когда вы познакомитесь с другими функциями и операциями, которые по умолчанию используют `$_`, то еще выше оцените полезность данной конструкции. Это один из тех случаев, когда короткая конструкция более понятна, чем длинная.

Если список, над которым производятся циклические преобразования, состоит из реальных переменных, а не получен при помощи функции,

возвращающей списочное значение, то используемая в цикле переменная представляет собой псевдоним для каждой переменной этого списка, а не просто копию ее значения. Это значит, что, изменяя скалярную переменную, вы изменяете и конкретный элемент в списке, который ей соответствует. Например:

```
@a = (3,5,7,9);
foreach $one (@a) {
    $one *= 3;
}
# @a теперь равно (9,15,21,27)
```

Обратите внимание на то, что изменение переменной `$one` привело к изменению каждого элемента массива `@a`.

Упражнения

Ответы к упражнениям даны в приложении А.

1. Напишите программу, которая запрашивает температуру окружающего воздуха и выводит на экран слова “too hot”, если температура выше 72 градусов (по Фаренгейту), а в противном случае выводит “too cold”.
2. Модифицируйте программу из предыдущего упражнения так, чтобы она выводила на экран “too hot”, если температура выше 75 градусов, “too cold” при температуре ниже 68 градусов и “just right!” при температуре от 68 до 75 градусов.
3. Напишите программу, которая читает список чисел (каждое из которых записано в отдельной строке), пока не будет прочитано число 999, после чего программа выводит сумму всех этих чисел. (Ни в коем случае не прибавляйте 999!) Например, если вы вводите 1, 2, 3 и 999, программа должна ответить цифрой 6 (1+2+3).
4. Напишите программу, которая читает список строковых значений (каждое из которых занимает отдельную строку) и выводит его на экран в обратном порядке, причем без проведения над списком операции `reverse`. (Вспомните, что `<STDIN>` при использовании в списочном контексте читает список строковых значений, каждое из которых занимает отдельную строку.)
5. Напишите программу, которая выводит на экран таблицу чисел от 0 до 32 и их квадратов. Попробуйте предложить способ, при котором список не обязательно должен содержать все указанные числа, а затем способ, при котором их нужно задавать все. (Чтобы выводимый результат выглядел более привлекательно, используйте функцию

```
printf "%5g %8g\n", $a, $b
```

которая выводит `$a` как пятизначное число, а `$b` — как восьмизначное.

В этой главе:

- *Что такое хеш*
- *Хеш-переменные*
- *Литеральное представление хеша*
- *Хеш-функции*
- *Срезы хешей*
- *Упражнения*

5

Хеши

Что такое хеш

Хеш* похож на массив, который мы рассматривали выше, тем, что представляет собой набор скалярных данных, отдельные элементы которого выбираются по индексному значению. В отличие от массива, индексные значения хеша — не малые неотрицательные целые, а произвольные скаляры. Эти скаляры (называемые *ключами*) используются для выборки значения из массива.

Элементы хеша не стоят в каком-то конкретном порядке. Можете рассматривать их как стопку библиографических карточек. Верхняя половина каждой карточки — это ключ, а нижняя — значение. Каждый раз, когда вы помещаете в хеш значение, создается новая карточка. Когда нужно изменить значение, вы указываете ключ, и Perl находит необходимую карточку. Поэтому порядок карточек, по сути дела, роли не играет. Perl хранит все карточки (т.е. пары ключ-значение) в особом внутреннем порядке, который облегчает поиск конкретной карточки, поэтому при поиске не приходится просматривать все пары. Порядок хранения карточек изменять нельзя, так что даже и не пытайтесь**.

Хеш-переменные

Имя хеш-переменной состоит из знака процента (%) и буквы, за которой могут идти другие буквы, цифры и знаки подчеркивания числом от нуля и больше. Другими словами, часть имени после знака процента похожа на

* В старой документации хеши назывались ассоциативными массивами, но мы настолько устали применять к столь распространенному понятию такой многосложный термин, что решили заменить его гораздо более удачным односложным словом

** Модули типа `IxHash` и `DB_file` обеспечивают некоторую степень упорядочения, но ценой существенного снижения производительности

соответствующую часть имен скалярных переменных и массивов. Кроме того, точно так же, как нет никакой связи между `$fred` и `@fred`, хеш-переменная `%fred` не имеет ничего общего с названными объектами.

Чаще всего хеш создается и используется путем обращения к его элементам, а не ко всему хешу. Каждый элемент хеша — отдельная скалярная переменная, доступная через индекс, представляющий собой строковое значение и называемый ключом. Так, обращение к элементам хеша `%fred` производится путем указания `$fred{$ключ}`, где `$ключ` — любое скалярное выражение. Вновь подчеркнем, что обращение к элементу хеша требует иного синтаксиса, нежели обращение ко всему хешу целиком.

Как и в случае с массивами, новые элементы хеша создаются путем присваивания значения:

```
$fred{"aaa"} = "bbb";      # создает ключ "aaa", значение "bbb"
$fred{234.5} = 456.7;      # создает ключ "234.5", значение 456.7
```

С помощью этих операторов в хеше создаются два элемента. При последующем обращении к элементам (по указанным ключам) возвращаются ранее записанные значения:

```
print $fred{"aaa"};        # выводит на экран "bbb"
$fred{234.5} += 3;         # делает значение равным 459.7
```

При обращении к несуществующему элементу возвращается значение `undef` (как и при обращении к отсутствующему элементу массива или к неопределенной скалярной переменной).

Литеральное представление хеша

У вас может возникнуть необходимость обратиться к хешу целиком — например, чтобы инициализировать его или скопировать в другой хеш. Фактически в Perl никакого литерального формата для хеша не предусмотрено, поэтому он просто представляется в виде списка. Каждая пара элементов этого списка (в котором всегда должно быть четное число элементов) задает ключ и соответствующее значение. Это развернутое представление может быть присвоено другому хешу, который затем воссоздаст тот же самый хеш. Другими словами:

```
@fred_list = %fred;        # @fred_list получает значение
                             # ("aaa","bbb","234.5","456.7")
%barney = @fred_list;      # создать %barney как %fred
%barney = %fred;           # ускоренный метод выполнения этой задачи
%smooth = ("aaa","bbb","234.5","456.7");
# создать %smooth как %fred из литеральных значений
```

Порядок пар ключ-значение в этом развернутом формате произвольный и контролю не поддается. Даже если вы меняете местами какие-то значения или создасте хеш целиком, возвращаемый развернутый список все равно будет стоять в том порядке, который Perl выбрал для обеспечения эффективного доступа к отдельным элементам. Ни на какую конкретную последовательность рассчитывать нельзя.

Одно из применений такого свертывания-развертывания — копирование хеш-значения в другую хеш-переменную:

```
%copy = %original; # копировать из %original в %copy
```

Используя операцию `reverse`, можно создать хеш, в котором ключи и значения поменяются местами:

```
%backwards = reverse %normal;
```

Конечно, если `%normal` имеет два идентичных значения, то в `%backwards` они превратятся в один элемент, поэтому данную операцию лучше всего выполнять только над хешами с уникальными ключами и значениями.

Хеш-функции

В этом разделе перечислены некоторые функции, предназначенные для обработки хешей.

Функция *keys*

Функция `keys(%имя_хеша)` выдает список всех текущих ключей, имеющих в хеше `%имя_хеша`. Другими словами, применение этой функции эквивалентно возвращению всех элементов списка с нечетными номерами (первый, третий, пятый и т.д.) путем развертывания хеша `%имя_хеша` в списочном контексте, причем функция `keys` возвращает их именно в этом порядке. Если элементы в хеше отсутствуют, функция `keys` возвращает пустой список.

Применим эту функцию к хешу из предыдущих примеров:

```
$fred{"aaa"} = "bbb";  
$fred{234.5} = 456.7;  
@list = keys(%fred); # @list получает значение ("aaa", 234.5)  
                     # или (234.5, "aaa")
```

Как и во всех остальных встроенных функциях, круглые скобки не обязательны: функция `keys %fred` полностью идентична `keys(%fred)`.

```
foreach $key (keys(%fred)) { # однократно для каждого значения хеша %fred  
    print "at $key we have $fred{$key}\n"; # показать ключ и значение  
}
```

В этом примере показано также, что отдельные элементы хеша могут интерполироваться в строки в двойных кавычках. Весь хеш, однако, интерполировать таким образом нельзя*.

В скалярном контексте функция `keys` выдает число элементов (пар ключ-значение), содержащихся в хеше. Например, вы можете выяснить, пуст ли хеш, так:

```
if (keys(%хеш)) { # если keys() не равно 0:
    ...;        # массив не пустой
}
# ... или ...
while (keys(%хеш) < 10) {
    ...;        # продолжать цикл, пока меньше 10 элементов
}
```

Для того чтобы узнать, пуст хеш или нет, нужно просто использовать функцию `%хеш` в скалярном контексте:

```
if (%хеш) { # если "истина", в нем что-то есть
    # что-то сделать
}
```

Функция *values*

Функция `values(%имя_массива)` возвращает список всех текущих значений указанного массива в том же порядке, в каком функция `keys(%имя_массива)` возвращает ключи. Как всегда, круглые скобки не обязательны. Например:

```
%lastname = (); # сделать %lastname пустым
$lastname{"fred"} = "flintstone";
$lastname{"barney"} = "rubble";
@lastnames = values(%lastname); # получить значения
```

Массив `@lastnames` будет содержать либо значения ("flintstone", "rubble"), либо ("rubble", "flintstone").

Функция *each*

Для выполнения цикла над всем хешем (т.е. для проверки каждого его элемента) можно использовать функцию `keys` и получать значения по возвращаемым ею ключам. Действительно, этот метод широко используется, но есть и более эффективный способ — функция `each(%имя_хеша)`, которая возвращает пару ключ-значение как двухэлементный список. При каждом вычислении этой функции для одного хеша возвращается очередная пара ключ-значение, пока не будут проверены все элементы. Если пар больше нет, `each` возвращает пустой список.

* Можно, в принципе, с помощью среза, но здесь о срезах мы не говорим

Например, чтобы пройти по хешу `%lastname` из предыдущего примера, нужно использовать нечто такое:

```
while (($first,$last) = each(%lastname)) {  
    print "The last name of $first is $last\n";  
}
```

Присваивание нового значения всему хешу заставляет функцию `each` перейти в его начало. Добавление элементов в хеш и удаление из него элементов в ходе выполнения цикла вполне может “запутать” функцию `each` (и вас, наверное, тоже).

Функция delete

Итак, вы можете добавлять элементы в хеш, но пока не можете удалять их (кроме как путем присваивания нового значения всему хешу). Для удаления элементов хеша в Perl используется функция `delete`. Операнд этой функции — хеш-ссылка, аналогичная той, которая используется для извлечения конкретного значения из хеша. Perl удаляет из хеша соответствующую ссылке пару ключ-значение. Например:

```
%fred = ("aaa","bbb",234.5,34.56); # добавить в %fred два элемента  
delete $fred{"aaa"}; # теперь в хеше %fred только одна пара ключ-значение
```

Срезы хешей

Как и в случае с переменной-массивом (или списочным литералом), можно воспользоваться срезом хеша, что даст возможность обращаться не к одному его элементу, а одновременно к набору элементов. Возьмем, к примеру, результаты игры в кегли:

```
$score{"fred"} = 205;  
$score{"barney"} = 195;  
$score{"dino"} = 30;
```

Все это можно записать одной строкой:

```
($score{"fred"},$score{"barney"},$score{"dino"}) = (205,195,30);
```

Но даже она слишком длинна, поэтому давайте используем *срез хеша*:

```
$score{"fred","barney","dino"} = (205,195,30);
```

Вот так гораздо короче. Можно сочетать использование среза хеша и интерполяции переменных:

```
@players = qw(fred barney dino);  
print "scores are: @score{@players}\n";
```

Срезы хешей можно также использовать для слияния небольшого хеша с более крупным. В этом примере меньший хеш имеет приоритет в том смысле, что при наличии ключей-дубликатов используется значение из меньшего хеша:

```
@league{keys %score} = values %score;
```

Здесь значения хеша `%score` сливаются с хешем `%league`. Это эквивалентно выполнению гораздо более медленной операции:

```
%league = (%league, %score); = # слить %score с %league
```

Упражнения

Ответы к упражнениям даны в приложении А.

- 1. Напишите программу, которая читает строку, а затем выводит эту строку и соответствующее ей значение согласно приведенной ниже таблице:

Ввод	Вывод
red	apple
green	leaves
blue	ocean

- 2. Напишите программу, которая читает ряд слов (по одному в строке) до конца файла, а затем выводит на экран сводку о том, сколько раз встретилось каждое слово. (Дополнительная задача: отсортируйте появляющиеся на экране слова по их ASCII-значениям в порядке возрастания последних.)

В этой главе:

- Ввод из *STDIN*
- Ввод из операции “ромб”
- Вывод в *STDOUT*
- Упражнения

6

Базовые средства ввода-вывода

Ввод из *STDIN*

Чтение со стандартного ввода (через Perl-дескриптор файла *STDIN*) — несложная задача. Мы уже делали это в операции *<STDIN>*. Выполняя эту операцию в скалярном контексте, мы получаем следующую строку ввода* (а если строк больше нет — то значение *undef*):

```
$a = <STDIN>;    # прочитать следующую строку
```

Выполнение в списочном контексте дает все оставшиеся строки в виде списка, каждый элемент которого представляет собой одну строку, включающую завершающий символ новой строки. Напомним:

```
@a = <STDIN>;
```

Как правило, при разработке программы приходится решать задачу чтения всех строк по одной с последующим выполнением над каждой из них каких-либо операций. Вот широко используемый метод решения этой задачи:

```
while (defined($line_ = <STDIN>)) {  
    # здесь обработать $line  
}
```

Пока есть непрочитанные строки, в результате выполнения операции *<STDIN>* получается определенное значение, и выполнение цикла продолжается. Если строк для чтения у *<STDIN>* больше нет, эта операция возвращает значение *undef* и завершает цикл.

* До символа новой строки или того, что вы присвоили переменной *\$/*.

Операция чтения скалярного значения из `<STDIN>` в `$_` и использование этого значения в качестве переменной цикла (как в предыдущем примере) выполняется довольно часто, поэтому в Perl предусмотрена для этого случая специальная сокращенная запись. Если в выражении для проверки цикла указан только оператор чтения со стандартного ввода (нечто вроде `<...>`), то Perl автоматически копирует строку, которая считывается, в переменную `$_`.

```
while (<STDIN>) { # как "while(defined($_ = <STDIN>))"
    chomp;      # как "chomp($_)"
    # здесь выполняются другие операции с $_
}
```

Поскольку переменная `$_` по умолчанию используется во многих операциях, таким способом вы можете значительно сократить объем набираемого текста.

Ввод из операции “ромб”

Другой способ чтения входной информации состоит в использовании операции “ромб” (`<>`). По принципу работы она похожа на `<STDIN>` — тем, что возвращает в скалярном контексте одну строку (или `undef`, если все строки прочитаны), а в списочном контексте — все оставшиеся строки. В отличие, однако, от операции `<STDIN>` операция “ромб” получает данные из файла или файлов, заданных в командной строке при вызове Perl-программы. Пусть, например, у вас есть программа *kitty*:

```
#!/usr/bin/perl
while (<>) {
    print $_;
}
```

и вы вызываете ее так:

```
kitty file1 file2 file3
```

Операция “ромб” читает сначала все строки первого файла, затем все строки второго и, наконец, все строки третьего файла. Значение `undef` возвращается только тогда, когда будут прочитаны все строки всех файлов. Как видите, программа *kitty* немного похожа на UNIX-команду *cat* — тем, что посылает все строки указанных файлов по очереди на стандартный вывод. Если, как в *cat*, имена файлов в командной строке не указаны, операция “ромб” автоматически читает данные со стандартного ввода.

В ходе своего выполнения операция “ромб” не рассматривает аргументы командной строки непосредственно, а работает с массивом `@ARGV`. Это специальный массив, инициализируемый интерпретатором Perl в соответствии с аргументами командной строки. Каждый аргумент заносится в отдельный элемент этого массива. Интерпретировать этот массив можно как

удочно*. Можно даже определить этот массив в своей программе и заставить операцию “ромб” работать с этим новым списком, а не с аргументами командной строки, например:

```
@ARGV = ("aaa", "bbb", "ccc");
while (<>) {    # обработать файлы aaa, bbb и ccc
    print "this line is: $_";
}
```

В главе 10 мы увидим, как можно открывать и закрывать определенные файлы в определенное время. Впрочем, мы уже использовали этот метод в некоторых наших слепленных на скорую руку программах.

Вывод в STDOUT

Для посылки каких-либо данных на стандартный вывод в Perl служат функции `print` и `printf`. Давайте посмотрим, как они работают.

Функция `print` — обычный вывод

Мы уже пользовались функцией `print`, направляя текст на стандартный вывод. Давайте остановимся на этом вопросе несколько подробнее.

Функция `print` получает список строк и посылает их по очереди на стандартный вывод, не добавляя никаких промежуточных или конечных символов. Возможно, не совсем очевидно, что `print` — это просто функция, которая принимает список аргументов и возвращает определенное значение подобно любой другой функции. Другими словами,

```
$a = print("hello ", "world", "\n");
```

— это еще один вариант вывода приветствия `hello, world`. Возвращаемое значение функции `print` — “истина” или “ложь”, говорящее соответственно об успешном и неудачном выполнении. Она почти всегда выполняется успешно, если только не произошла ошибка ввода-вывода, поэтому в рассматриваемом случае `$a` обычно бывает равно 1.

Иногда в `print` нужно вводить круглые скобки, как показано в данном примере, особенно когда первый элемент, который вы хотите вывести, сам начинается с левой скобки:

```
print (2+3),"hello";    # неверно! Выводит 5, игнорируя "hello"
print ((2+3),"hello");  # верно, выводит 5hello
print 2+3,"hello";      # тоже верно, выводит 5hello
```

* Стандартный дистрибутив Perl содержит модули `getopt`-подобного синтаксического анализа аргументов командной строки Perl-программы. Информация об этой библиотеке есть в книге *Programming Perl* и на map-странице `perlmodlib(1)`.

Функция `printf` — форматированный вывод

Вероятно, вы захотите в большей степени контролировать выводимые данные, чем это позволяет функция `print`. Возможно, вы привыкли к форматированному выводу C-функции `printf`. Спешим обрадовать: в Perl есть почти сравнимая операция с таким же именем.

Функция `printf` принимает список аргументов (заключенный в необязательные круглые скобки, как и в случае использования функции `print`). Первый аргумент — строка управления форматом, указывающая, как вывести остальные аргументы. Если вы не знакомы со стандартной функцией `printf`, обратитесь к map-странице `printf(3)` или `perlfunc(1)`, если она у вас есть, или к главе 3 книги *Programming Perl*.

Пример:

```
printf "%15s %5d %10.2f\n", $s, $n, $r;
```

Эта функция выводит `$s` в 15-символьном поле, затем пробел, затем `$n` как десятичное целое в 5-символьном поле, затем еще один пробел, затем `$r` как значение с плавающей запятой с двумя десятичными знаками в 10-символьном поле и, наконец, символ новой строки.

Упражнения

Ответы к упражнениям даны в приложении А.

1. Напишите программу, которая работает аналогично команде `cat`, но изменяет на обратный порядок следования всех строк всех файлов, указанных в командной строке, или всех строк, поступающих со стандартного ввода, если файлы не указаны. (В некоторых системах есть подобная утилита; она называется `tac`.)
2. Измените программу из предыдущего упражнения таким образом, чтобы в каждом файле, указанном в командной строке, порядок строк изменялся на обратный индивидуально. (Да-да, вы можете сделать это, пользуясь тем материалом, который до сих пор изучили, даже не возвращаясь к обзору возможностей языка Perl, приведенному в главе 1.)
3. Напишите программу, которая читает список строковых значений (каждое из которых занимает отдельную строку) и выводит эти строки в 20-символьном столбце с выравниванием справа. Например, при вводе строк `hello` и `good-bye` они выводятся в 20-символьном столбце с выравниванием справа. (Добейтесь, чтобы ваша программа действительно использовала 20-символьный столбец, а не 21-символьный. Это весьма распространенная ошибка.)
4. Измените программу из предыдущего упражнения таким образом, чтобы пользователь мог выбирать ширину столбца. Например, при вводе строк `20, hello` и `good-bye` получатся те же результаты, что и в предыдущем упражнении, а ввод `30, hello, good-bye` обеспечит размещение `hello` и `good-bye` в 30-символьном столбце.

В этой главе:

- Основные понятия
- Основные направления использования регулярных выражений
- Образцы
- Еще об операции сопоставления
- Операция замены
- Функции *split* и *join*
- Упражнения

Регулярные выражения

Основные понятия

Регулярное выражение представляет собой образец — шаблон — который сопоставляется со строкой. Сопоставление регулярного выражения со строкой дает либо успешный результат, либо неудачный. Иногда получение того или иного результата может быть единственной целью использования регулярного выражения, а иногда ставится задача замены совпавшего образца другой строкой.

Регулярные выражения используются многими программами, в частности, UNIX-командами, программами *grep*, *sed*, *awk*, *ed*, *vi*, *emacs* и даже различными shell. В каждой программе используется свой набор метасимволов (большой частью они совпадают). Perl — семантическое надмножество всех этих средств: любое регулярное выражение, которое можно записать в одной из подобных программ, может быть записано и на языке Perl, но не обязательно теми же символами.

Основные направления использования регулярных выражений

Если бы нам нужно было найти в каком-то файле все строки, содержащие строку *abc*, мы могли бы использовать команду *grep*:

```
grep abc somefile >results
```

В этом случае `abc` — регулярное выражение, которое команда *grep* сверяет с каждой входной строкой. Строки, соответствующие этому регулярному выражению, посылаются на стандартный вывод и попадают в файл *results* (так как в командной строке стоит оператор переадресации).

В Perl мы можем превратить строку `abc` в регулярное выражение, заключив ее между косыми:

```
if (/abc/) {  
    print $_;  
}
```

Но что же сверяется с регулярным выражением `abc` в данном случае? Да наша старая подруга, переменная `$_`! Если регулярное выражение заключено между косыми (как в этом примере), то переменная `$_` сверяется с регулярным выражением. Если значение переменной совпадает с регулярным выражением, операция *сопоставления* возвращает значение “истина”. В противном случае она возвращает “ложь”.

В данном примере предполагается, что переменная `$_` содержит какую-то строку текста и выводится, если в любом месте этой строки обнаруживается последовательность символов `abc` (аналогичные действия производит приведенная выше команда *grep*. Однако в отличие от *grep*, которая оперирует всеми строками файла, данный фрагмент Perl-программы просматривает только одну строку). Чтобы обрабатывались все строки, добавьте операцию цикла:

```
while (<>) {  
    if (/abc/) {  
        print $_;  
    }  
}
```

А что, если мы не знаем, сколько символов `b` стоит между `a` и `c`? То есть что нужно делать, если мы хотим вывести на экран строку только в том случае, если она содержит символ `a`, за которым следует ни одного или более символов `b` и символ `c`? Работая с *grep*, мы написали бы так:

```
grep "ab*c" somefile >results
```

(Аргумент, содержащий звездочку, заключен в кавычки, потому что мы не хотим, чтобы shell обработал его так, как будто это метасимвол, встретившийся в имени файла. Чтобы звездочка сработала, ее нужно передать в *grep* как есть.) В Perl мы можем сделать то же самое:

```
while (<>) {  
    if (/ab*c/) {  
        print $_;  
    }  
}
```

Как и в *grep*, такая запись обозначает последовательность, содержащую символ *a*, ни одного или более символов *b* и символ *c*.

Другие варианты сопоставления с образцом мы рассмотрим в разделе “Еще об операции сопоставления” после того, как поговорим обо всех видах регулярных выражений.

Еще одна простая операция, в которой используются регулярные выражения, — операция замены, посредством которой часть строки, соответствующая регулярному выражению, заменяется другой строкой. Операция замены похожа на команду *s* UNIX-утилиты *sed*: она состоит из буквы *s*, косой черты, регулярного выражения, еще одной косой, заменяющей строки и третьей косой черты:

```
s/ab*c/def/;
```

Переменная (в данном случае *_*) сопоставляется с регулярным выражением (*ab*c*). Если сопоставление оказалось успешным, то соответствующая часть строки отбрасывается и заменяется строкой (*def*). Если сопоставление неудачно, ничего не происходит.

Позже, в разделе “Операция замены”, мы рассмотрим множество опций операции замены.

Образцы

Регулярное выражение — это образец. Одни части образца обозначают отдельные символы. Другие части соответствуют группам символов. Сначала мы рассмотрим образцы, соответствующие одному символу, а затем образцы, при помощи которых в регулярном выражении обозначается группа символов.

Образцы, обозначающие один символ

Самый простой и самый распространенный символ, встречающийся в регулярных выражениях, — это одиночный символ, соответствующий самому себе. Другими словами, наличие буквы *a* в регулярном выражении требует наличия соответствующей буквы *a* в строке.

Следующий из самых известных символов сопоставления — точка (“.”). Точка обозначает любой одиночный символ, кроме символа новой строки (*\n*). Например, образцу */a./* соответствует любая двухбуквенная последовательность, которая начинается с буквы *a* и не является последовательностью “*a\n*”.

Класс символов сопоставления задается списком символов, заключенных в квадратные скобки. Чтобы строка считалась совпавшей с образцом, в соответствующей ее части должен присутствовать один и только один из этих символов. Например, образцу

```
/[abode]
```

соответствует строка, содержащая любую из первых пяти строчных букв алфавита, тогда как образцу

`/[aeiouAEIOU]`

соответствует любая из первых пяти гласных, причем как строчных, так и прописных. Если вы хотите вставить в список правую квадратную скобку (`]`), поставьте перед ней обратную косую или же поставьте эту скобку на первое место в списке. Диапазоны символов (например, от `a` до `z`) можно приводить в сокращенной записи, указав конечные точки диапазона через дефис (`-`). Чтобы включить в список дефис как таковой, поставьте перед ним обратную косую или поместите его в конец. Вот еще несколько примеров:

```
[0123456789] # обозначает любую цифру
[0-9]         # то же самое
[0-9\^-]      # обозначает цифры 0-9 или знак минус
[a-z0-9]      # обозначает любую строчную букву или цифру
[a-zA-Z0-9_]  # обозначает любую букву, цифру или знак подчеркивания
```

Существует также такое понятие, как отрицание класса символов: оно обозначается знаком `^`, который ставится сразу же за левой скобкой. Такому классу символов соответствует любой символ, отсутствующий в этом списке. Например:

```
[^0-9]        # обозначает любой нецифровой символ
[^aeiouAEIOU] # обозначает любую негласную букву
[^\\^]        # обозначает любой символ, кроме символа ^
```

Для удобства пользователя некоторые распространенные классы символов определены заранее. Они представлены в таблице 7.1.

Таблица 7.1. Предопределенные классы символов

Конструкция	Эквивалентный класс	Конструкция с отрицанием	Эквивалентный класс с отрицанием
<code>\d</code> (цифра)	<code>[0-9]</code>	<code>\D</code> (нецифровые символы)	<code>[^0-9]</code>
<code>\w</code> (обычный символ)	<code>[a-zA-Z0-9_]</code>	<code>\W</code> (специальные символы)	<code>[^a-zA-Z0-9_]</code>
<code>\s</code> (пробельный символ)	<code>[\r\t\n\f]</code>	<code>\S</code> (непробельный символ)	<code>[^ \r\t\n\f]</code>

Образцу `\d` соответствует одна цифра. Образцу `\w` формально соответствует один обычный символ, но на самом деле ему соответствует любой символ, который допустим в именах переменных Perl. Образцу `\s` соответствует один пробельный символ. К пробельным символам относятся пробел, возврат каретки (редко используемый в UNIX), символ табуляции, символы

перехода на новую строку и на новую страницу. Варианты конструкций с использованием прописных букв соответствуют дополнениям (отрицаниям) этих классов. Так, \w обозначает один специальный символ, \s — один символ, который не является пробельным (т.е. является буквой, знаком препинания, управляющим символом и т.д.), а \D — один нецифровой символ.

Приведенные выше конструкции можно использовать при задании других классов символов:

```
[\da-fA-F] # соответствует одной шестнадцатеричной цифре
```

Образцы, обозначающие группу символов

Свою истинную силу регулярные выражения показывают, когда вам нужно сказать, например, “один и более из этих символов” или “до пяти из этих символов”. Давайте посмотрим, как это делается.

Последовательность

Первый (и, вероятно, самый неочевидный) образец данного вида — *последовательность*. Например, образец `abc` соответствует букве `a`, за которой следует буква `b`, за которой идет буква `c`. Вроде бы просто, но название этому виду образца все равно нужно дать, чтобы в дальнейшем знать, о чем идет речь.

Множители

Мы уже встречались со звездочкой (*) в роли образца, обозначающего группу символов. Звездочка обозначает ни одного или более экземпляров стоящего непосредственно перед ней символа (или класса символов).

Есть еще два образца, работающих подобным образом: знак “плюс” (+), который обозначает один или более экземпляров стоящего непосредственно перед ним символа, и вопросительный знак (?), который обозначает ни одного или один экземпляр стоящего непосредственно перед ним символа. Например, регулярное выражение `/fo+ba?r/` обозначает символ `f`, за которым следует один или более символов `o`, затем символ `b`, затем ни одного или один символ `a` и, наконец, символ `r`.

Однако все описанные выше образцы (множители) характеризуются “прозорливостью”. Например, если множителю может соответствовать 5-10 символов, то каждый раз он будет выбирать десятисимвольную строку. Например,

```
$_ = "fred xxxxxxxxxxxx barney";  
s/x+/boom/;
```

всегда заменяет словом `boom` все символы `x` (что в результате дает `fred boom barney`), а не только один или два, несмотря на то, что более короткий набор `ixсов` соответствовал бы этому же регулярному выражению.

Если нужно сказать “от пяти до десяти” символов x , можно поставить пять иксов, а затем еще пять, дав после каждого из последних пяти вопросительный знак. Это, однако, выглядит уродливо. Есть более простой способ — применение *общего множителя*. Общий множитель состоит из пары фигурных скобок, между которыми заключены одно-два числа, например $/x\{5,10\}$. Необходимо найти символ, стоящий непосредственно перед скобками (в данном случае это буква x), повторяющийся указанное число раз (в рассматриваемом случае — от пяти до десяти)*.

Если второе число не указано (например, $/x\{5, \}/$), это означает “столько или больше” (в данном случае пять и более), а если выпущена и запятая (например, $/x\{5\}/$), это означает “ровно столько” (в данном случае пять символов x). Чтобы получить пять или менее символов x , нужно перед запятой поставить ноль: $/x\{0,5\}/$.

Так, регулярное выражение $/a.\{5\}b/$ соответствует букве a , отделенной от буквы b любыми пятью символами, кроме символов новой строки, и все это может быть в любом месте строки. (Вспомните, что точка соответствует любому символу, кроме символа новой строки, а нам здесь нужно пять таких символов.) Эти пять символов не обязательно должны быть одинаковыми. (В следующем разделе мы увидим, как заставить их быть одинаковыми.)

Можно было бы вполне обойтись без $*$, $+$ и $?$, потому что эти образцы полностью эквивалентны образцам $\{0, \}$, $\{1, \}$ и $\{0,1\}$, но проще ввести один эквивалентный знак препинания, к тому же это более привычно.

Если в одном выражении используются два множителя, то “правило прожорливости” дополняется правилом “чем левее, тем прожорливее”. Например:

```
$ _ = "a xxx c xxxxxxxx c xxx d";
/a.*c.*d/;
```

В этом случае первая комбинация $“.*”$ в регулярном выражении соответствует всем символам до второй буквы c , несмотря на то, что положительный результат был бы достигнут даже при совпадении только символов, стоящих до первой буквы c . Сейчас это никакой роли не играет, но позднее, когда нам потребуются анализировать части, совпавшие с регулярным выражением, это будет очень важно.

Можно заставить любой множитель перестать быть “прожорливым” (т.е. сделать его *ленивым*), поставив после него вопросительный знак:

```
$ _ = "a xxx c xxxxxxxx c xxx d";
/a.*?c.*d/;
```

Здесь $a.*?c$ теперь соответствует минимальному числу символов между a и c , а не максимальному. Это значит, что c образцом совпадает часть

* Конечно, $/\{d\{3\}/$ соответствует не только трехзначным числам, но и любому числу с количеством знаков больше трех. Чтобы задать именно трехзначное число, нужно использовать фиксирующие точки, которые рассматриваются ниже в разделе “Фиксирующие образцы”.

строки до первой буквы *c*, а не до второй. Такой модификатор можно ставить после любого множителя (*?*, *+*, *** и *{m, n}*).

Что, если строка и регулярное выражение несколько изменятся, скажем, так:

```
$ _ = "a xxx ce xxxxxxxx ci xxx d";  
/a.*ce.*d/;
```

Символы *.** в этом случае соответствуют максимально возможному числу символов, стоящих до следующей буквы *c*, но очередной символ регулярного выражения (*e*) не совпадает с очередным символом строки (*i*). В этом случае мы получаем автоматический *поиск с возвратом*: поиск начинается сначала и завершается остановкой в некоторой позиции до выбранной на первом этапе (в нашем случае — в позиции предыдущей *c*, рядом с *e*)*. Сложное регулярное выражение может включать множество уровней поиска с возвратом, в результате чего время выполнения значительно увеличивается. В данном случае превращение множителя в “ленивый” (с помощью вопросительного знака) упрощает задачу, которую должен выполнить Perl, поэтому рекомендуем хорошо изучить этот метод.

Круглые скобки как способ запоминания

Следующая групповая операция — пара круглых скобок, в которую заключается часть образца. При совпадении с образцом никаких изменений не происходит, просто совпавшая часть строки запоминается, и к ней можно впоследствии обращаться. Например, *(a)* продолжает соответствовать букве *a*, а *{[a-z]}* — любой строчной букве.

Чтобы вызвать часть строки, которую программа запомнила, нужно поставить обратную косую и целое число. Образец такой конструкции обозначает последовательность символов, обозначенную ранее в паре круглых скобок под тем же номером (считая с единицы). Например,

```
/fred(.)barney\1/;
```

соответствует строке, состоящей из слова *fred*, любого символа, кроме символа новой строки, слова *barney* и еще одного такого же символа. Таким образом, данному образцу соответствует последовательность символов *fredxbarneyx*, а не *fredxbarneyy*. Сравните это с

```
/fred.barney./;
```

где два обозначенных точками символа могут быть одинаковыми или разными; роли это не играет.

Откуда взялась единица? Она обозначает первую заключенную в круглые скобки часть регулярного выражения. Если таких частей больше, чем одна,

* На самом деле для поиска буквы *c* в первой позиции понадобится больший объем поиска с возвратом в операции ***, но описание этого процесса не представляет интереса, а работает он по такому же принципу.

то вторая часть (считая левые круглые скобки слева направо) обозначается как \2, третья — как \3 и т. д. Например,

```
/a(.)b(.)c\2d\1/;
```

обозначает *a*, какой-то символ (назовем его #1), *b*, еще один символ (назовем его #2), *c*, символ #2, *d* и символ #1. Таким образом, этот образец соответствует, в частности, строке *axbycydx*.

Запоминаемая часть может состоять не только из одного символа. Например,

```
/a(.*?)b\1c/;
```

обозначает *a*, любое количество символов (даже нуль), *b*, ту же последовательность символов и, наконец, *c*. Следовательно, этот образец совпадет со строкой *aFREDbFREDc* и даже со строкой *abc*, но не со строкой *aXXbXXXc*.

Дизъюнкция

Следующая групповая конструкция — *дизъюнкция*, т.е. *a|b|c*. Это значит, что данный образец соответствует только одному из указанных вариантов (в данном случае — *a*, *b* или *c*). Такая конструкция работает даже в том случае, если варианты содержат несколько символов, как в образце */song|blue/*, что соответствует либо *song*, либо *blue*. (Для односимвольных альтернатив определенно лучше будет использовать класс символов, например, */[abc]/*.)

Что, если бы мы хотели найти *songbird* или *bluebird*? Мы могли бы написать */songbird|bluebird/*, но часть *bird* не хотелось бы указывать дважды. Из такой ситуации есть выход, однако вначале нам следует поговорить о приоритете группирующих образцов, который рассматривается ниже, в разделе “Приоритет”.

Фиксирование образцов

Некоторые особые виды записи позволяют фиксировать образец относительно позиции в строке, в которой ищется соответствие. Обычно при сопоставлении образец “перемещается” по строке слева направо; сообщение о совпадении дается при первой же возможности. Фиксирующие точки позволяют гарантировать, что с образцом совпадают определенные части сравниваемой строки.

Первая пара фиксирующих директив требует, чтобы определенная часть символов, соответствующих образцу, была расположена либо на границе слова, либо не на границе слова. Фиксирующая директива *\b* требует, чтобы совпадение с образцом *b* происходило только на границе слова. Граница слова — это место между символами, которые соответствуют предопределенным классам *\w* или *\W*, либо между символами, которые соответствуют классу *\w*, а также начало или окончание строки. Отметим, что все это больше предназначено для работы с *C*, а не с английскими словами, но вполне применимо и к словам. Например:

```
/fred\b/;      # соответствует слову fred, но не Frederick
/\bmo/;       # соответствует словам мое и mole, но не Elmo
/\bFred\b/;   # соответствует слову Fred, но не Frederick или alFred
/\b\+\b/;     # соответствует "x+y", но не "++" или " + "
/abc/bdef/;   # никогда не дает совпадения (границы там быть не может)
```

Аналогичным образом \B требует, чтобы в указанной точке границы слова не было. Например:

```
/\bFred\B/;   # соответствует "Frederick", но не "Fred Flintstone"
```

Две другие фиксирующие точки требуют, чтобы определенная часть образца стояла рядом с концом строки. Символ ^ обозначает начало строки, если стоит в месте, где сопоставление с началом строки имеет смысл. Например, ^a соответствует символу a в том и только в том случае, если a — первый символ в строке. a^ соответствует двум символам, a и ^, стоящим в любом месте строки. Другими словами, символ ^ утратил свое специальное значение. Если вы хотите, чтобы он имел буквальный смысл и в начале строки, поставьте перед ним обратную косую черту.

Символ \$, как и ^, фиксирует образец, но не по началу, а по концу строки. Другими словами, c\$ соответствует символу c только в том случае, если он стоит в конце строки*. Знак доллара в любом другом месте образца, вероятно, будет интерпретироваться как представление скалярного значения, поэтому для того, чтобы использовать его в строке буквально, перед ним следует поставить обратную косую.

Поддерживаются и другие фиксирующие точки, включая \A, \Z и предупреждающие фиксирующие точки, создаваемые с помощью комбинаций (?=...) и (!...). Они подробно описаны в главе 2 книги *Programming Perl* и на map-странице *perlre(1)*.

Приоритет

Что произойдет, если объединить a|b*? Что будет отыскиваться — любое количество символов a или b или один символ a и любое количество b?

Групповые и фиксированные образцы, как и операции, имеют приоритет. Приоритет образцов (от высшего к низшему) приведен в таблице 7.2.

Таблица 7.2. Приоритет групповых регулярных выражений**

Наименование	Обозначение
Круглые скобки	() (? :
Множители	? + * {m,n} ?? +? *? {m,n}?
Последовательность и фиксация	abc ^ \$ \A \Z (?=) (!)
Дизъюнкция	

* Или прямо перед символом новой строки в конце строки.
** Некоторые из этих символов в нашей книге не описываются. См. книгу *Programming Perl* или map-страницу *perlre(1)*.

Согласно этой таблице, специальный символ * имеет более высокий приоритет, чем |. В силу этого /a|b*/ интерпретируется как один символ a или любое число символов b.

Что, если нам понадобится другое — например, “любое число символов a или b”? В этом случае нужно просто использовать пару круглых скобок. В нашем примере в скобки нужно заключить ту часть выражения, к которой должна относиться *, т.е. (a|b)*. Если вы хотите подчеркнуть, какое выражение вычисляется первым, можно дать избыточные круглые скобки: a| (b*) .

Изменение приоритета с помощью круглых скобок одновременно активизирует режим запоминания для данного образца, как мы рассказывали выше. То есть эти круглые скобки учитываются, когда вы определяете, соответствует ли какой-то элемент \2, \3 и т.д. Если вы хотите использовать круглые скобки без включения режима запоминания, применяйте форму (?:...), а не (...). Она тоже позволяет указывать множители, но не изменяет значение счетчика подлежащих запоминанию лексем, используя, например, переменную \$4 и т.п. Например, /(?:Fred|Wilma) Flintstone/ ничего не записывает в переменную \$1; здесь просто предполагается группирование.

Вот еще несколько примеров регулярных выражений и действия круглых скобок:

abc*	# соответствует ab, abc, abcc, abccc, abcccc, и т.д.
(abc)*	# соответствует "", ab, abc, abcabc, abcabcabc, и т.д.
^x y	# соответствует x в начале строки или y в любом месте
^(x y)	# соответствует x или y в начале строки
a bc d	# либо a, либо bc, либо d
(a b)(c d)	# ac, ad, bc или bd
(song blue)bird	# songbird или bluebird

Еще об операции сопоставления

Мы уже рассматривали простейшие варианты использования операции сопоставления (регулярного выражения, заключенного между косыми). Теперь давайте изучим способы, которыми можно заставить эту операцию делать нечто иное.

Выбор другого объекта для сопоставления (операция =~)

Обычно строка, которую нужно сопоставить с образцом, не находится в переменной \$_, и помещать ее туда довольно утомительно. (Может быть, в переменной \$_ уже хранится значение, которое вам не хочется терять.) Ничего страшного — здесь нам поможет операция =~. С ее помощью вы можете назначить для проведения операции сопоставления строку, хранящуюся в переменной, отличной от \$_.

Эта переменная указывается справа от знака операции. Выглядит это так:

```
$a = "hello world";
$a =~ /^he/;           # истина
$a =~ /(.)\1/;         # тоже истина (соответствует двум 1)
if ($a =~ /(.)\1/) {   # истина, поэтому проводятся дальнейшие операции
}
```

Справа от знака операции `=~` может стоять любое выражение, которое дает в результате некоторое скалярное строковое значение. Например, `<STDIN>` при использовании в скалярном контексте дает скалярное строковое значение, поэтому, объединив эту операцию с операцией `=~` и операцией сопоставления с регулярным выражением, мы получим компактную программу проверки входных данных:

```
print "any last request? ";
if (<STDIN> =~ /^[yY]/) { # начинаются ли входные данные с буквы y?
    print "And just what might that request be? ";
                        # и чтобы это мог быть за запрос?
    <STDIN>;           # получить строку со стандартного ввода
    print "Sorry, I'm unable to do that.\n";
                        # прошу прощения, но я не могу этого сделать
}
```

В данном случае при помощи `<STDIN>` берется очередная строка со стандартного ввода, которая затем сразу же используется как строка, сопоставляемая с образцом `^[yY]`. Отметим, что мы не сохраняли входные данные в переменной, поэтому если мы захотим сопоставить эти данные с другим образцом или же вывести их в сообщении об ошибке, то у нас ничего не выйдет. Тем не менее эта форма часто оказывается удобной.

Игнорирование регистра

В предыдущем примере мы указывали образец `[yY]` для обозначения строчной и прописной буквы `y`. Если речь идет об очень коротких строках, например, `y` или `fred`, то данный способ обозначения достаточно удобен, скажем, `[fF][rR][eE][dD]`. А что делать, если сопоставляемая строка — это слово `procedure` в нижнем или верхнем регистре?

В некоторых версиях *grep* флаг `-i` означает “игнорировать регистр”. В Perl тоже есть такая опция. Чтобы ею воспользоваться, нужно добавить строчную `i` к закрывающей косой черте, т.е. написать `/образец/i`. Такая запись говорит о том, что буквы образца будут соответствовать буквам строки в любом регистре. Например, чтобы найти слово `procedure` в любом регистре, стоящее в начале строки, запишите `/^procedure/i`.

Теперь наш предыдущий пример будет выглядеть так:

```
print "any last request? ";
if (<STDIN> =~ /^y/i) { # начинаются ли входные данные с буквы y?
                        # да! выполнить какие-то операции
    ...
}
```

Использование другого разделителя

Чтобы найти строку, которая содержит несколько косых (/), в соответствующем регулярном выражении нужно перед каждой из них поставить обратную косую черту (\). Например, чтобы найти строку, которая начинается с названия директории /usr/etc, нужно записать:

```
$path = <STDIN>; # прочитать путевое имя (вероятно, из find?)
if ($path =~ /\^\/usr\/etc/) {
    # начинается с /usr/etc...
}
```

Как видите, комбинация “обратная косая — косая” создает между элементами текста своеобразные “проходы”. Если косых очень много, это занятие может стать весьма утомительным, поэтому в Perl предусмотрена возможность использования другого разделителя (delimiter). Поставьте перед любым специальным символом* (выбранным вами в качестве разделителя) букву `m`, укажите свой образец и дайте еще один такой же разделитель:

```
/^\/usr\/etc/ # использование стандартного разделителя — косой черты
m@^\/usr/etc@ # использование в качестве разделителя символа @
m#^\/usr/etc# # использование в качестве разделителя символа #
               # (это мой любимый символ)
```

Если хотите, можете опять использовать косые, например, `m/fred/`. Таким образом, `m` — общепринятое обозначение операции сопоставления с регулярным выражением, но если в качестве разделителя выбрана косая черта, то `m` не обязательна.

Использование интерполяции переменных

Перед тем как регулярное выражение рассматривается на предмет наличия специальных символов, в нем производится интерполяция переменных. Следовательно, регулярное выражение можно строить не только из литералов, но и из вычисляемых строк. Например:

```
$what = "bird";
$sentence = "Every good bird does fly.";
if ($sentence =~ /\b$what\b/) {
    print "The sentence contains the word $what'\n";
}
```

Здесь мы использовали ссылку на переменную для построения операции сопоставления с регулярным выражением `\bbird\b/`.

* Если этот разделитель — левый элемент пары (круглая, фигурная, угловая или квадратная скобка), то закрывающим разделителем будет соответствующий правый элемент пары. В остальных случаях первый и второй разделители будут совпадать.

Вот несколько более сложный пример:

```
$sentence = "Every good bird does fly.";
print "What should I look for? ";
$what = <STDIN>;
chomp($what);
if ($sentence =~ /$what/) {    # нашли!
    print "I saw $what in $sentence.\n";
} else {
    print "nope... didn't find it.\n";
}
```

Если вы введете слово `bird`, оно будет найдено, а если слово `scream` — не будет. Если ввести `[bw]ird`, результаты поиска тоже будут успешными. Это говорит о том, что квадратные скобки в данном случае воспринимаются как символы сопоставления с образцом.

Чтобы избежать этого, следует поставить перед этими символами обратную косую, которая превратит их в символы буквального сопоставления. Это кажется сложным, если в вашем распоряжении нет заковычивающей управляющей последовательности `\Q`:

```
$what = "[box]";
foreach (qw(in([box] out[box] white[sox])) {
    if (/\\Q$what\\E/) {
        print "$_ matched'\n";
    }
}
```

Здесь конструкция `\\Q$what\\E` превращается в `\\[box\\]`, в результате чего операция сопоставления ищет пару квадратных скобок, а не рассматривает всю конструкцию как класс символов.

Специальные переменные, защищенные от записи

После успешного сопоставления с образцом переменным `$1`, `$2`, `$3` и т.д. присваиваются те же значения, что и `\1`, `\2`, `\3` и т.д. Это можно использовать для поиска соответствия в последующем коде. Например:

```
$_ = "this is a test";
/(\w+)\W+(\w+)/;    # сопоставление первых двух слов
                    # $1 теперь содержит this, а $2 — is
```

Доступ к тем же значениям (`$1`, `$2`, `$3` и т.д.) можно также получить, используя операцию сопоставления для соответствующих списков. Если результаты сопоставления окажутся положительными, будет получен список значений от `$1` до `$n` (где `n` — количество занесенных в память элементов). В противном случае значения не определены. Запишем последний пример по-другому:

```
$_ = "this is a test";
($first, $second) = /(\w+)\W+(\w+)/; # сопоставление первых двух слов
                    # $first теперь содержит this, а $second — is
```

К другим предопределенным защищенным от записи переменным относятся: `$&` (часть строки, совпавшая с регулярным выражением); `$`` (часть строки, стоящая перед совпавшей частью); `$'` (часть строки, стоящая после совпавшей части). Например:

```
$_ = "this is a sample string";  
/sa.*le/; # соответствует слову sample внутри строки  
# $` теперь содержит "this is a "  
# $& теперь содержит "sample"  
# $' теперь содержит "string"
```

Поскольку значения этим переменным присваиваются при каждом успешном сопоставлении, их нужно где-нибудь сохранить, если они вам впоследствии понадобятся*.

Операция замены

Мы уже говорили о простейшей форме операции замены: `s/регулярное_выражение/новая_строка/`. Пора рассмотреть несколько разновидностей этой операции.

Если вы хотите, чтобы замена выполнялась при всех возможных совпадениях, а не только при первом, добавьте в запись, задающую проведение операции замены, букву `g`, например:

```
$_ = "foot fool buffoon";  
s/foo/bar/g; # $_ теперь содержит "bart barl bufbarn"
```

В заменяющей строке производится интерполяция переменных, что позволяет задавать эту строку во время выполнения:

```
$_ = "hello, world";  
$new = "goodbye";  
s/hello/$new/; # заменяет hello на goodbye
```

Символы сопоставления (метасимволы) в регулярном выражении позволяют выполнять сопоставление с образцом, а не просто с символами, трактуемыми буквально:

```
$_ = "this is a test";  
s/(\w+)/<$1>/g; # $_ теперь содержит "<this> <is> <a> <test>"
```

Вспомните, что в `$1` заносятся данные, полученные при совпадении с первой заключенной в круглые скобки частью образца.

Суффикс `i` (перед буквой `g` или после нее, если она есть) заставляет используемое в операции замены регулярное выражение игнорировать регистр, как и аналогичная опция в ранее рассмотренной нами операции сопоставления.

* О влиянии этих переменных на производительность рассказывается в книге *Mastering Regular Expressions* (издательство O'Reilly).

Как и в операции сопоставления, можно выбрать другой разделитель, если косая черта неудобна. Для этого просто нужно использовать один символ три раза*:

```
s#fred#barney#;      # заменить fred на barney, как в s/fred/barney/
```

Как и при сопоставлении, можно с помощью операции `=~` указать другой объект для проведения замены. В этом случае объект должен быть таким, которому можно присвоить скалярное значение, — например, скалярной переменной или элементом массива:

```
$which = "this is a test";  
$which =~ s/test/quiz/;      # $which теперь содержит "this is a quiz"  
$someplace[$here] =~ s/left/right/; # заменить элемент массива  
$d{"t"} =~ s/^/x /;          # поставить "x " перед элементом массива
```

Функции *split* и *join*

Регулярные выражения можно использовать для разбивки строки на поля. Это делает функция `split`. Функция `join` выполняет противоположное действие — вновь “склеивает” эти кусочки.

Функция *split*

Функция `split` получает регулярное выражение и строку и ищет в этой строке все экземпляры указанного регулярного выражения. Те части строки, которые не совпадают с регулярным выражением, возвращаются по порядку как список значений. Вот, например, код синтаксического анализа разделенных двоеточиями полей, аналогичных тем, которые используются в UNIX-файлах `/etc/passwd`:

```
$line = "merlyn::118:10:Randal:/home/merlyn:/usr/bin/perl";  
@fields = split(/:./,$line); # разбить $line, используя в качестве  
                             # разделителя двоеточие  
# теперь @fields содержит ("merlyn","", "118", "10",  
# "Randal", "/home/merlyn", "/usr/bin/perl")
```

Обратите внимание на то, что второе пустое поле стало пустой строкой. Если вы этого не хотите, задайте сопоставление следующим образом:

```
@fields = split(/:+/ , $line);
```

Здесь при сопоставлении принимаются во внимание одно и более расположенных рядом двоеточий, поэтому пустое поле не образуется.

Очень часто приходится разбивать на поля значение переменной `$_`, поэтому этот случай предлагается по умолчанию:

```
$_ = "some string";  
@words = split(/ /); # то же самое, что и @words = split(/ /, $_);
```

* Или две пары, если используется символ из пары “левая-правая”.

При такой разбивке соседние пробелы в разбиваемой строке вызовут появление пустых полей (пустых строк). Лучше использовать образец / +/, а лучше /\s+/, который соответствует одному и более пробельным символам. Этот образец, по сути дела, используется по умолчанию*, поэтому, если вы разбиваете переменную \$_ по пробельным символам, вы можете использовать все стандартные значения и просто написать :

```
@words = split;    # то же самое, что и @words = split(/\s+/, $_);
```

Завершающие строки пустые поля в список, как правило, не включаются. Особой роли это обычно не играет. Решение вроде

```
$line = "merlyn::118:10:Randal:/home/merlyn:";
($name,$password,$uid,$gid,$gcos,$home,$shell) = split(/:/,$line);
# разбить $line, используя в качестве разделителя двоеточие
```

просто присваивает переменной \$shell нулевое значение (undef), если эта строка недостаточно длинна или содержит в последнем поле пустые значения. (Разбиение выполняется так, что лишние поля просто игнорируются.)

Функция join

Функция join берет список значений и “склеивает” их, ставя между элементами списка строку-связку. Выглядит это так:

```
$bigstring = join($glue,@list);
```

Например, чтобы восстановить строку пароля, попробуйте использовать следующее:

```
$outline = join(":", @fields);
```

Отметим, что строка-связка — это не регулярное выражение, а обычная строка, состоящая из символов общим числом нуль или более.

Если нужно поставить связку не между элементами, а перед каждым элементом, то достаточно такого трюка:

```
$result = (join "+", "", @fields);
```

Здесь пустая строка "" рассматривается как пустой элемент, который должен быть связан с первым элементом данных массива @fields. В результате связка помещается перед каждым элементом. Аналогичным образом можно поставить пустой элемент-связку в конец списка:

```
$output = join ("\n", @data, "");
```

* На самом деле образец по умолчанию — строка "", поэтому начальный пробельный разделитель игнорируется, но для нас вышесказанного пока достаточно

Упражнения

Ответы к упражнениям даны в приложении А.

1. Постройте регулярное выражение, которое соответствует:
 - а) минимум одному символу `a`, за которым следует любое число символов `b`;
 - б) любому числу обратных косых, за которым следует любое число звездочек (любое число может быть и нулем);
 - в) трем стоящим подряд копиям того, что содержится в переменной `$whatever`;
 - г) любым пяти символам, включая символ новой строки;
 - д) одному слову, написанному два или более раз подряд (с возможно изменяющимся пробельным символом), где “слово” определяется как непустая последовательность непробельных символов.
 2. а) Напишите программу, которая принимает список слов из `STDIN` и ищет строку, содержащую все пять гласных (`a`, `e`, `i`, `o` и `u`). Запустите эту программу с `/usr/dict/words*` и посмотрите, что получится. Другими словами, введите

```
$ программа </usr/dict/words
```
 - б) Модифицируйте программу так, чтобы пять гласных должны были стоять по порядку, а промежуточные буквы значения не имели.
 - в) Модифицируйте программу так, чтобы все гласные должны были стоять в порядке возрастания, чтобы все пять гласных должны были присутствовать и чтобы перед буквой “`a`” не стояла буква “`e`”, перед буквой “`e`” не стояла буква “`i`” и т.д.
3. Напишите программу, которая просматривает файл `/etc/passwd**` (из `STDIN`), выводя на экран регистрационное имя и реальное имя каждого пользователя. (Совет: с помощью функции `split` разбейте строку на поля, а затем с помощью `s///` избавьтесь от тех частей поля `comment`, которые стоят после первой запятой.)

* Словарь вашей системы может находиться не в каталоге `/usr/dict/words`; обратитесь к map-странице `spell(1)`.

** Если используется NIS, то файл `/etc/passwd` в вашей системе будет содержать мало данных. Посмотрите, может быть, `ypcat passwd` даст больше информации.

4. Напишите программу, которая просматривает файл */etc/passwd* (из `STDIN`) на предмет наличия двух пользователей с одинаковыми именами и выводит эти имена. (Совет: после извлечения первого имени создайте хеш с этим именем в качестве ключа и числом его экземпляров в качестве значения. Прочитав последнюю строку `STDIN`, ищите в этом хеше счетчики с показанием больше единицы.)
5. Повторите последнее упражнение, но с выдачей имен всех пользователей, зарегистрировавшихся под одинаковыми именами. (Совет: в хеше вместо числа экземпляров сохраните список регистрационных имен, записанных через пробелы. Затем ищите значения, содержащие пробел.)

В этой главе:

- Определение пользовательской функции
- Вызов пользовательской функции
- Возвращаемые значения
- Аргументы
- Локальные переменные в функциях
- Полулокальные переменные, созданные при помощи функции *local*
- Создаваемые операцией *my()* переменные файлового уровня
- Упражнения

Функции

Мы уже знакомы со встроенными пользовательскими функциями, например *chomp*, *print* и другими, и пользовались ими. Теперь давайте рассмотрим функции, которые вы можете определить сами.

Определение пользовательской функции

Пользовательская функция, чаще называемая подпрограммой, определяется в Perl-программе с помощью следующей конструкции:

```
sub имя {
    оператор_1;
    оператор_2;
    оператор_3;
}
```

Здесь *имя* — это имя подпрограммы, которое может быть любым именем вроде *tex*, которые мы давали скалярным переменным, массивам и хешам. Вновь подчеркнем, что эти имена хранятся в другом пространстве имен, поэтому у вас может быть скалярная переменная *\$fred*, массив *@fred*, хеш *%fred*, а теперь и подпрограмма *fred**.

* Более правильно эту подпрограмму следовало бы назвать *&fred*, но пользоваться такого рода именами приходится редко

Блок операторов, следующий за именем подпрограммы, становится ее определением. Когда эта подпрограмма вызывается, то блок операторов, входящих в нее, выполняется, и вызывающему объекту выдается соответствующее возвращаемое значение (как будет описано ниже).

Вот, например, подпрограмма, которая выдает знаменитую фразу:

```
sub say_hello {  
    print "hello, world!\n";  
}
```

Определения подпрограмм могут стоять в любом месте текста программы (при выполнении они пропускаются), но обычно их размещают в конце файла программы, чтобы основная часть программы находилась в начале. (Если вам часто приходилось писать программы на языке Паскаль, можете по привычке поставить свои подпрограммы в начало, а выполняемые операторы — в конец. Это ваше дело.)

Определения подпрограмм глобальны*; локальных подпрограмм не бывает. Если у вас вдруг появились два определения подпрограмм с одинаковым именем, то второе определение заменяет первое без предупреждения**.

В теле подпрограммы вы можете обращаться к переменным, используемым в других частях программы (*глобальным* переменным), и присваивать им значения. По умолчанию любая ссылка на переменную в теле подпрограммы относится к глобальной переменной. Об исключениях из этого правила мы расскажем в разделе “Локальные переменные в функциях”.

В следующем примере:

```
sub say_what {  
    print "hello, $what\n";  
}
```

переменная `$what` является глобальной переменной, которая может использоваться также и в других частях программы.

Вызов пользовательской функции

Для вызова подпрограммы из любого выражения необходимо поставить после ее имени круглые скобки, например:

```
say_hello();           # простое выражение  
$a = 3 + say_hello()   # часть более сложного выражения  
for ($x = start_value(); $x < end_value(); $x += increment()) {  
    ...  
} # вызов трех подпрограмм для определения значений
```

* Точнее, глобальны для текущего пакета, но поскольку в этой книге отдельные пакеты не рассматриваются, вы можете считать определения подпрограмм глобальными для всей программы.

** Если только вы не выполняете программу с ключом `-w`.

Одна подпрограмма может вызывать другую подпрограмму, которая, в свою очередь, может вызывать третью подпрограмму и т.д., пока вся наличная память не будет заполнена адресами возврата и не полностью вычисленными выражениями. (Ведь настоящего программиста вряд ли удовлетворят какие-то 8 или 32 уровня вложенности подпрограмм.)

Возвращаемые значения

Вызов подпрограммы почти всегда является частью некоторого выражения. Результат, полученный после выполнения подпрограммы, называется *возвращаемым значением*. Возвращаемое значение представляет собой результат выполнения оператора `return` или последнего выражения, вычисленного в подпрограмме.

Давайте, например, определим такую подпрограмму:

```
sub sum_of_a_and_b {  
    return $a + $b;  
}
```

Последнее выражение, вычисляемое в теле этой подпрограммы (фактически единственное вычисляемое выражение), — сумма переменных `$a` и `$b`, поэтому эта сумма и будет возвращаемым значением. Вот как все это работает:

```
$a = 3; $b = 4;  
$c = sum_of_a_and_b();      # $c присваивается значение 7  
$d = 3 * sum_of_a_and_b();  # $d содержит значение 21
```

При вычислении в списочном контексте подпрограмма может возвращать список значений. Рассмотрим такую подпрограмму и ее вызов:

```
sub list_of_a_and_b  
{  
    return($a,$b);  
}  
$a = 5;      $b = 6;  
@c = list_of_a_and_b();    # @c присваивается значение (5,6)
```

Последнее вычисленное выражение действительно означает последнее вычисленное выражение, а не последнее выражение, определенное в теле подпрограммы. Например, следующая подпрограмма возвращает `$a`, если `$a > 0`; в противном случае она возвращает `$b`:

```
sub gimme_a_or_b {  
    if ($a > 0) {  
        print "choosing a ($a)\n";  
        returns $a;  
    } else {  
        print "choosing b ($b)\n";  
        returns $b;  
    }  
}
```

Все это довольно тривиальные примеры. Однако будет гораздо лучше, если вместо того, чтобы полагаться на глобальные переменные, мы сможем передавать в подпрограмму значения, разные для каждого вызова. Именно к этому мы сейчас и перейдем.

Аргументы

Несомненно, подпрограммы, выполняющие одно определенное действие, полезны, однако перед вами откроются совершенно новые горизонты, когда вы сможете передавать в подпрограмму *аргументы*.

В Perl после вызова подпрограммы следует список, заключенный в круглые скобки, которые обеспечивают автоматическое присваивание элементов данного списка на период выполнения этой подпрограммы специальной переменной с именем `@_`. Подпрограмма может обратиться к этой переменной и получить число аргументов и их значения. Например:

```
sub say_hello_to {  
    print "hello, $_[0]!\n"      # первый параметр  
}
```

Здесь мы видим ссылку на `$_[0]` — первый элемент массива `@_`. Обратите внимание: несмотря на внешнее сходство, значение `$_[0]` (первый элемент массива `@_`) не имеет ничего общего с переменной `$_` (самостоятельной скалярной переменной). Не путайте их! Из этого кода видно, что подпрограмма приветствует того, чье имя мы указываем в качестве первого параметра. Это значит, что ее можно вызвать так:

```
say_hello_to("world");           # выдает hello, world  
$x = "somebody";  
say_hello_to($x);               # выдает hello, somebody  
say_hello_to("me")+say_hello_to("you"); # а теперь приветствует себя и вас
```

В последней строке возвращаемые значения явно использованы не были, однако для определения суммы Perl сначала должен вычислить все ее слагаемые, поэтому подпрограмма была вызвана дважды.

Вот пример с использованием более одного параметра:

```
sub say {  
    print "$_[0], $_[1]!\n";  
}  
say("hello","world");           # опять hello world  
say("goodbye","cruel world");   # goodbye cruel world - популярная фраза из фильмов
```

Избыточные параметры игнорируются, т.е. если вы никогда не заглядываете в `$_[3]`, языку Perl это абсолютно все равно. Недостающие параметры также игнорируются, и если вы попытаетесь обратиться за пределы массива `@_`, как и любого другого массива, то просто получите в ответ `undef`.

Переменная `@_` является локальной для подпрограммы. Если для `@_` установлено глобальное значение, то оно перед вызовом подпрограммы сохраняется, а после возврата из подпрограммы восстанавливается. Это также означает, что подпрограмма может передавать аргументы в другую подпрограмму, не боясь “потерять” собственную переменную `@_`; вложенный вызов подпрограммы точно так же получает собственную переменную `@_`.

Давайте вернемся к программе сложения `a` и `b` из предыдущего раздела. Вот подпрограмма, которая складывает любые два значения, а именно два значения, передаваемые в нее как параметры:

```
sub add_two {
    return $_[0] + $_[1];
}
print add_two(3,4);      # выводит значение 7
$c = add_two(5,6);      # $c получает значение 11
```

Давайте обобщим эту подпрограмму. Что, если нам нужно сложить 3, 4 или 100 значений? Это можно было бы сделать с помощью цикла, например:

```
sub add {
    $sum = 0;          # инициализировать сумму
    foreach $_ (@_) {
        $sum += $_;    # прибавить все элементы
    }
    return $sum        # последнее вычисленное выражение: сумма всех элементов
}
$a = add(4,5,6);      # складывает 4+5+6=15 и присваивает переменной $a
print add(1,2,3,4,5); # выводит 15
print add (1..5);     # тоже выводит 15, потому что список 1..5 раскрывается
```

Что, если бы переменная с именем `$sum` использовалась у нас раньше? При вызове подпрограммы `add` мы просто потеряли бы ее значение. В следующем разделе мы увидим, как избежать этого.

Локальные переменные в функциях

Мы уже говорили о переменной `@_` и о том, как для каждой подпрограммы, вызываемой с параметрами, создается локальная копия этой переменной. Вы можете создавать собственные скалярные переменные, переменные-массивы и хеш-переменные, которые будут работать точно так же. Это делается с помощью операции `my`, которая получает список имен переменных и создает их локальные версии (или *реализации*, если вам так больше нравятся). Вот снова функция `add`, на этот раз построенная на основе операции `my`:

```
sub add {
    my ($sum);        # сделать $sum локальной переменной
    $sum = 0;         # инициализировать сумму
```



```

    foreach $_ (@_) {
        $sum += $_; # прибавить все элементы
    }
return $sum          # последнее вычисленное выражение: сумма всех элементов
}

```

Когда выполняется первый оператор в теле подпрограммы, текущее значение глобальной переменной `$sum` сохраняется и создается совершенно новая переменная с именем `$sum` (и значением `undef`). При выходе из подпрограммы Perl отбрасывает эту локальную переменную и восстанавливает предыдущее (глобальное) значение. Эта схема работает даже в том случае, если переменная `$sum` в текущий момент является локальной переменной, взятой из другой подпрограммы (той, которая вызывает данную подпрограмму, или той, которая вызывает ту, которая вызывает данную подпрограмму, и т.д.). Переменные могут иметь много вложенных локальных версий, но одновременно разрешается обращаться только к одной из них.

Вот способ создания списка всех элементов массива, значения которых превышают число 100:

```

sub bigger_than_100 {
    my (@result);          # временная для хранения возвращаемого значения
    foreach $_ (@_) {      # проход по списку аргументов
        if ($_ > 100) {     # подходит?
            push(@result, $_); # прибавить
        }
    }
    return @result;        # вернуть окончательный список
}

```

Что, если бы нам понадобились все элементы, значения которых превышают 50, а не 100? Пришлось бы отредактировать программу, заменив 100 на 50. А если бы нам было нужно и то, и другое? В этом случае следовало бы заменить 50 или 100 ссылкой на переменную. Тогда программа выглядела бы так:

```

sub bigger_than {
    my ($n, @values);      # создать локальные переменные
    ($n, @values) = @_;    # выделить из списка аргументов значение предела
                          # и элементы массива
    my (@result);          # временная переменная для хранения возвращаемого
                          # значения
    foreach $_ (@values) { # проход по списку аргументов
        if ($_ > $n) {      # подходит?
            push(@result, $_); # прибавить
        }
    }
    @result;               # вернуть окончательный список
}

# примеры вызова:
@new = bigger_than(100, @list); # @new содержит все значения @list > 100
@this = bigger_than(5, 1, 5, 15, 30); # @this содержит значение (15, 30)

```

Обратите внимание: в этот раз мы использовали еще две локальные переменные, чтобы присвоить имена аргументам. Это — весьма распространенная практика, потому что гораздо удобнее (да и безопаснее) использовать переменные `$n` и `$values`, чем указывать `$_[0]` и `@_[1..$#_]`.

В результате выполнения операции `my` создается список, который можно использовать в левой части операции присваивания. Этому списку можно присваивать начальные значения для каждой из вновь созданных переменных. (Если значения списку не присвоить, то новые переменные вначале получают значение `undef`, как и любая другая переменная.) Это значит, что мы можем объединить первые два оператора этой подпрограммы, т.е. операторы

```
my($n,@values);  
($n,@values) = @_; # выделить из списка аргументов значение предела  
                   # и элементы массива
```

заменить на

```
my($n,@values)= @_;
```

Этот прием очень распространен в Perl. Локальным переменным, не являющимся аргументами, можно таким же способом присваивать литеральные значения, например:

```
my($sum) = 0; # инициализировать локальную переменную
```

Имейте в виду, что, несмотря на наличие объявления, `my` в действительности представляет собой выполняемую операцию. Стратегия правильной работы в Perl состоит в том, что все операции `my` должны быть размещены в начале определения подпрограммы, до того, как начинается реализация основных выполняемых в ней действий.

Полулокальные переменные, созданные при помощи функции `local`

В Perl существует еще один способ создания “частных” локальных переменных — с помощью функции `local`. Важно понять различия между `my` и `local`. Например:

```
$values = "original";  
  
tellme();  
spooft();  
tellme();  
sub spooft {  
    local ($value) = "temporary";  
    tellme();  
}  
sub tellme {  
    print "Current value is $value\n";  
}
```

На выходе получаем следующее:

```
Current value is original
Current value is temporary
Current value is original
```

Если бы вместо `local` мы использовали `my`, то локальный вариант переменной `$value` был бы доступен только в пределах подпрограммы `spoof()`. При использовании функции `local`, как видно по результатам программы, это локальное значение не совсем локальное; оно доступно и во всех остальных подпрограммах, вызываемых из `spoof()`. Общее правило таково: локальные переменные доступны для функций, вызываемых из того блока, в котором эти функции объявлены.

Операцию `my` можно использовать только для объявления простых скалярных переменных, переменных-массивов и хеш-переменных с буквенно-цифровыми именами, тогда как для переменной `local` такие ограничения не установлены. Кроме того, встроенные переменные Perl, такие как `$_`, `#1` и `@ARGV`, с помощью `my` объявлять нельзя, а вот с `local` они работают прекрасно. Поскольку `$_` используется в большинстве Perl-программ, то будет, вероятно, разумным помещать строку

```
local $_;
```

в начало любой подпрограммы, которая использует `$_` для собственных нужд. Это гарантирует сохранность предыдущего значения и его автоматическое восстановление при выходе из подпрограммы.

На более серьезном уровне программирования вам, возможно, нужно будет знать, что переменные, создаваемые функцией `local`, — это, по сути дела, замаскированные глобальные переменные, т.е. значение глобальной переменной сохраняется и временно заменяется локально объявленным значением.

В большинстве случаев рекомендуем использовать не `local`, а `my`, потому что эта операция действует быстрее и надежнее.

Создаваемые операцией `my()` переменные файлового уровня

Операцию `my()` можно также использовать на внешнем уровне программы, вне всех подпрограмм и блоков. Хотя в результате не будет получена “локальная” переменная в описанном выше смысле, это может оказаться достаточно полезным, особенно при использовании в сочетании с *Perl-прагмой**

```
use strict;
```

* Прагма — это директива компилятора. Среди этих директив — директивы задания целочисленных арифметических операций, перегрузки числовых операций, запрашивания дополнительных текстовых предупреждений и сообщений об ошибках. Эти директивы описаны в главе 7 книги *Programming Perl* и на map-странице `perlmodlib(1)`.

Если поставить эту прагму в начало файла, то вы больше не сможете использовать переменные (скалярные, массивы и хеши), сначала не “объявив” их. Объявляются они с помощью операции `my()` следующим образом:

```
use strict;
my $a;                                # сначала значение undef
my @b = qw(fred barney betty);        # присвоить начальное значение
...
push @b, qw(wilma);                  # разве можно забыть про Вильму?
@c = sort @b;                        # НЕ КОМПИЛИРУЕТСЯ
```

Во время компиляции последний оператор будет помечен флагом ошибки, потому что он обратился к переменной, не объявленной ранее с помощью операции `my` (т.е. `@c`). Другими словами, ваша программа даже не начнет работать до тех пор, пока не будут объявлены все используемые переменные.

Преимущества принудительного объявления переменных таковы:

1. Ваши программы будут работать несколько быстрее (обращение к переменным, созданным с помощью `my`, производится несколько быстрее, чем к обычным переменным).*
2. Гораздо быстрее будут выявляться ошибки набора, потому что вы больше не сможете случайно обратиться к несуществующей переменной `$freed`, когда вам будет нужна переменная `$fred`.

По этой причине многие программисты, пишущие на Perl, автоматически начинают каждую новую программу прагмой `use strict`.

Упражнения

Ответы к упражнениям даны в приложении А.

1. Напишите подпрограмму, которая будет принимать в качестве аргумента числовое значение от 1 до 9 и возвращать английское название этого числа (т.е. `one`, `two` и т.д.). Если значение не принадлежит указанному диапазону, подпрограмма должна вернуть вместо имени исходное число. Проверьте работу подпрограммы, введя какие-нибудь данные. Для вызова этой подпрограммы вам, наверное, придется написать какой-то код. (Совет: эта подпрограмма не должна выполнять никакого ввода-вывода.)
2. Взяв подпрограмму из предыдущего упражнения, напишите программу, которая будет брать два числа и складывать их, выдавая результат в формате `Two plus two equals four`. (Не забудьте начать первое слово с заглавной буквы!)
3. Модернизируйте эту подпрограмму так, чтобы она возвращала названия от `negative nine` до `negative one` и `zero` (т.е. принимала числа из диапазона от -9 до -1 и нуль). Проверьте, как она будет работать в программе из упражнения 2.

* “Обычная переменная” в этом случае — переменная пакета (поэтому `$x` — это, по сути дела, `$main::x`). Переменные, созданные с помощью `my()`, ни в один пакет не входят.

В этой главе:

- *Оператор last*
- *Оператор next*
- *Оператор redo*
- *Метки*
- *Модификаторы выражений*
- *Операции && и || как управляющие структуры*
- *Упражнения*

Управляющие структуры

Оператор last

Выполняя некоторые из предыдущих упражнений, вы, возможно, иногда думали: “Если бы в моем распоряжении был С-оператор `break`, все было бы нормально”. Даже если такая мысль и не приходила вам в голову — все равно позвольте мне рассказать о Perl-операторе для преждевременного выхода из цикла `last`.

Оператор `last` обеспечивает выход из самого внутреннего блока цикла, в котором расположен этот оператор*, передавая управление оператору, следующему непосредственно за этим блоком. Например:

```
while (что-то) {
    что-то;
    что-то;
    что-то;
    if (условие) {
        что-то или другое;
        что-то или другое;
        last; # выход из цикла while
    }
    еще что-то;
    еще что-то;
}
# last передает управление в эту точку программы
```

* Конструкция типа `do {} while/until` с точки зрения операторов `next`, `last` и `redo` циклом не считается.

Если *условие* истинно, то выполняются строки *что-то_или_другое*, после чего оператор `last` завершает цикл `while`.

Оператор `last` учитывает только блоки, образующие цикл, а не блоки, необходимые для построения определенных синтаксических конструкций. Это значит, что блоки для операторов `if` и `else`, равно как и для конструкций `do {} while/until`, “не считаются”; учитываются только блоки, которые образуют циклы `for`, `foreach`, `while`, `until` и “голые” блоки. (“Голый” блок — это блок, не являющийся частью более крупной конструкции, например цикла, подпрограммы, оператора `if/then/else`.)

Допустим, мы хотим посмотреть, было ли почтовое сообщение, сохраненное в файле, послано пользователем `merlyn`. Такое сообщение могло выглядеть следующим образом:

```
From: merlyn@stonehenge.com (Randal L. Schwartz)
To: stevet@ora.com
Date: 01-DEC-96 08:16:24 PM -0700
Subject: A sample mail message
```

Here's the body of the mail message. And
here is some more.

Нам нужно было бы найти в этом сообщении строку, которая начинается словом `From:`, а затем посмотреть, не содержит ли эта строка регистрационное имя `merlyn`.

Это можно было бы сделать так:

```
while (<STDIN>) {           # читать входные строки
    if (/^From: /) {        # начинается ли строка со слова From:? Если да...
        if (/merlyn/) {    # то сообщение от merlyn'
            print "Email from Randal' It's about time!\n";
        }
        last;              # дальше искать строки From: не нужно, поэтому выйти
    }                      # конец цикла "if from:"
    if (/^$/ ) {           # пустая строка?
        last;              # если да, больше строки не проверять
    }
}                            # конец цикла while
```

Найдя строку, которая начинается со слова `From:`, мы выходим из цикла, потому что хотим видеть только первую такую строку. Кроме того, поскольку заголовок почтового сообщения заканчивается на первой пустой строке, мы можем выйти и из основного цикла.

Оператор *next*

Как и *last*, оператор *next* изменяет последовательность выполнения программы. Отличие между ними состоит в том, что *next* заставляет программу пропускать оставшуюся часть самого внутреннего блока цикла, не завершая этот цикл*. Используется оператор *next* следующим образом:

```
while (что-то) {
    первая_часть;
    первая_часть;
    первая_часть;
    if (условие) {
        какая-то_часть;
        какая-то_часть;
        next;
    }
    другая_часть;
    другая_часть;
    # next передает управление в эту точку программы
}
```

Если *условие* истинно, то выполняется *какая-то_часть*, а *другая_часть* пропускается.

Как и при использовании оператора *last*, блок оператора *if* не считается блоком, образующим цикл.

Оператор *redo*

Третий способ передачи управления в блоке цикла — оператор *redo*. Эта конструкция обеспечивает переход в начало текущего блока (без повторного вычисления контрольного выражения):

```
while (условие) {
    # redo передает управление в эту точку программы
    что-то;
    что-то;
    что-то;
    if (условие) {
        какие-то_действия;
        какие-то_действия;
        redo;
    }
    еще_что-то;
    еще_что-то;
    еще_что-то;
}
```

* Если в данном цикле есть оператор *continue*, который мы не рассматривали, *next* переходит в начало блока *continue*, а не в конец блока цикла. Это практически одно и то же.

Блок `if` здесь тоже не учитывается; считаются только циклообразующие блоки.

Пользуясь оператором `redo`, оператором `last` и “голым” блоком, можно построить бесконечный цикл, образующийся внутри блока:

```
{
    начальные_действия;
    начальные_действия;
    начальные_действия;
    if (условие) {
        last;
    }
    последующие_действия;
    последующие_действия;
    последующие_действия;
    redo;
}
```

Такая схема годится для `while`-подобного цикла, некоторая часть которого должна выполняться как инициализационная перед первой проверкой. (В разделе “Модификаторы выражений” мы покажем, как можно использовать оператор `if` с меньшим числом знаков препинания.)

Метки

Что делать, если вы хотите выйти из блока, в котором содержится самый внутренний блок, иными словами, выйти сразу из двух вложенных блоков? В С вы могли бы обратиться к оклеветанному всеми оператору `goto`. В Perl такая хитрость не нужна — здесь по отношению к любому охватываемому блоку можно использовать операторы `last`, `next` и `redo`, присвоив этому блоку имя с помощью *метки*.

Метка — это еще один тип имени из еще одного пространства имен, отвечающего тем же правилам, что и имена скаляров, массивов, хешей и подпрограмм. Мы увидим, однако, что в метке нет специального префиксного символа (аналогичного `$` в скалярах, символу `&` в подпрограммах и т.д.), поэтому метка `print`, к примеру, конфликтовала бы с зарезервированным словом `print`, в силу чего ее использование не допускается. По этой причине лучше создавать метки с именами, состоящими только из прописных букв и цифр, которые в будущем никогда не будут использоваться в зарезервированных словах. Кроме того, прописные буквы и цифры хорошо выделяются в тексте программы, который набирается в основном с использованием нижнего регистра.

Выбрав имя метки, введите его прямо перед оператором, содержащим блок, и поставьте двоеточие:

```
МЕТКА: while (условие) {
    оператор;
    оператор;
```

```

оператор;
if (другое_условие) {
    last МЕТКА;
}
}

```

Мы указали метку как параметр в операторе `last`. Это дает языку Perl указание выйти из блока с именем `МЕТКА`, а не только из самого внутреннего блока. В данном случае, однако, у нас есть только один циклический блок. Давайте рассмотрим вариант со вложенными циклами:

```

OUTER: for ($i=1; $i <= 10; $i++) {
    INNER: for ($j=1; $j <= 10; $j++) {
        if ($i * $j == 63) {
            print "$i times '$j' is 63'\n";
            last OUTER;
        }
        if ($j>=$i) {
            next OUTER;
        }
    }
}

```

Этот набор операторов позволяет перебирать все результаты перемножения небольших чисел до тех пор, пока не будет найдена пара, произведение которой равно 63 (7 и 9). После того как эта пара будет найдена, проверка остальных чисел теряет смысл, поэтому первый оператор `if` обеспечивает выход из обоих циклов `for`, используя для этого оператор `last` с меткой. Второй оператор `if` гарантирует, что большим из двух чисел всегда будет первое, переходя к следующей итерации внешнего цикла, как только поставленное условие перестанет быть истинным. Это значит, что числа будут проверяться при значениях переменных $(\$i, \$j) = (1, 1), (2, 1), (2, 2), (3, 1), (3, 2), (3, 3), (4, 1)$ и т.д.

Даже если самый внутренний блок имеет метку, действие операторов `last`, `next` и `redo` без необязательного параметра (метки) все равно распространяется на него. Кроме того, с помощью меток нельзя переходить в блок — можно только выходить из него. Операторы `last`, `next` или `redo` должны находиться внутри блока.

Модификаторы выражений

В качестве еще одного способа сказать “если это, тогда то” Perl позволяет “прикреплять” к выражению, представляющему собой отдельный оператор, модификатор `if`:

выражение if управляющее_выражение;

В данном случае *управляющее_выражение* проверяется на истинность (по тем же правилам, что и всегда). Если оно истинно, то вычисляется *выражение*. Это приблизительно эквивалентно конструкции

```
if (управляющее_выражение) {  
    выражение;  
}
```

Однако в данном случае дополнительные знаки препинания не нужны, оператор читается справа налево, а выражение должно быть простым (т.е. не может быть блоком операторов). Во многих случаях, однако, такая инвертированная запись оказывается наиболее естественным способом задания условия. Например, вот как можно выйти из цикла при возникновении определенного условия:

```
LINE: while (<STDIN>) {  
    last LINE if /^From: /;  
}
```

Видите, насколько проще такая запись? Более того, вы даже можете прочитать ее на нормальном языке: “последняя строка, если она начинается со слова From”.

Кроме описанной выше, существуют и другие формы с модификаторами:

```
выраж2 unless выраж1;    # как unless (выраж1) { выраж2; }  
выраж2 while выраж1;     # как while (выраж1) { выраж2; }  
выраж2 until выраж1;     # как until (выраж1) { выраж2; }
```

Во всех этих формах сначала вычисляется *выраж1*, а затем в зависимости от результата что-то делается или не делается с *выраж2*.

Например, вот как можно найти первую по порядку степень числа 2, которая больше заданного числа:

```
chomp($n = <STDIN>);  
$i = 1;                      # исходное предположение  
$i *= 2 until $i > $n;       # вычислять, пока не будет найдена
```

Как видите, мы опять добиваемся ясности и убираем лишнее.

Эти формы нельзя вкладывать одну в другую: нельзя сказать *выраж3 while выраж2 if выраж1*. Это объясняется тем, что форма *выраж2 if выраж1* — уже не выражение, а полноценный оператор; после оператора модификатор ставить нельзя.

Операции `&&` и `||` как управляющие структуры

Операции `&&` и `||` в тексте программы похожи на знаки препинания или компоненты выражения. Можно ли считать их управляющими структурами? В Perl возможно почти все, поэтому давайте разберемся, что же здесь имеется в виду.

Вам часто приходится сталкиваться с концепцией “если это, тогда то”. Мы уже видели две подобные формы:

```
if (это) { то; }      # один способ
то if это;            # другой способ
```

Вот третья (хотите верьте, хотите нет — но есть и другие):

```
это && то;
```

Почему она работает? Разве это не операция “логическое И”? Давайте посмотрим, что происходит, когда *это* принимает значения “истина” или “ложь”.

- Если *это* — истина, то значение всего выражения все равно не известно, потому что оно зависит от значения элемента *то*. Поэтому нужно вычислить *то*.
- Если *это* — ложь, то все выражение будет ложным и нет никакого смысла смотреть на *то*. Поскольку вычислять *то* смысла нет, его можно пропустить.

Как раз это и делает Perl — вычисляет *то* лишь в том случае, если *это* истинно, что делает данную форму эквивалентной двум предыдущим.

Аналогичным образом “логическое ИЛИ” работает, как оператор `unless` (или модификатор `unless`). Вы можете заменить

```
unless (это) { то; }
```

на

```
это || то;
```

Если вы знаете, как использовать эти операции в shell для управления командами условного выполнения, то поймете, что в Perl они работают похоже.

Какую же из этих форм использовать? Это зависит от вашего настроения (иногда), от того, насколько велики части выражения, наконец, от того, нужно ли вам заключать выражения в круглые скобки из-за конфликтов приоритетов. Возьмите программы своих коллег и посмотрите, что они делают. В них, вероятно, вы обнаружите влияние каждого из этих факторов. Ларри предлагает ставить самую важную часть выражения первой, чтобы акцентировать на ней внимание.

Упражнения

Ответы к упражнениям приведены в приложении А.

1. Модифицируйте упражнение из предыдущей главы так, чтобы операция повторялась до тех пор, пока в качестве одного из значений не будет введено слово `end`. (Совет: используйте бесконечный цикл, а затем выполните оператор `last`, если любое из значений равно `end`.)
2. Перепишите упражнение из главы 4 (суммирование чисел до появления числа 999), используя цикл с выходом из середины. (Совет: используйте “голый” блок с оператором `redo` в конце, чтобы получить бесконечный цикл, и оператор `last` в середине, выполняющийся при выполнении определенных условий.)

В этой главе:

- *Что такое дескриптор файла*
- *Открытие и закрытие дескриптора файла*
- *Небольшое отступление: функция `die`*
- *Использование дескрипторов файлов*
- *Операции для проверки файлов*
- *Функции `stat` и `lstat`*
- *Упражнения*

10

Дескрипторы файлов и проверка файлов

Что такое дескриптор файла

Дескриптор файла в Perl-программе — это имя соединения для ввода-вывода между вашим Perl-процессом и внешним миром. Мы уже видели дескрипторы файлов и пользовались ими, сами того не зная: `STDIN` — это дескриптор, которым именуется соединение между Perl-процессом и стандартным вводом UNIX. Аналогичным образом в Perl существует `STDOUT` (для стандартного вывода) и `STDERR` (для стандартного вывода ошибок). Это те же самые имена, что и используемые библиотекой стандартного ввода-вывода в C и C++, которую Perl задействует в большинстве операций ввода-вывода.

Имена дескрипторов файлов похожи на имена помеченных блоков, но они берутся из другого пространства имен (поэтому у вас может быть скаляр `$fred`, массив `@fred`, хеш `%fred`, подпрограмма `&fred`, метка `fred`, а теперь и дескриптор файла `fred`). Как и метки блоков, дескрипторы файлов используются без специального префиксного символа, поэтому их можно спутать с существующими или возможными в будущем зарезервированными словами (для команд, подпрограмм и др.). Рекомендуем составлять дескрипторы файлов ТОЛЬКО ИЗ ПРОПИСНЫХ БУКВ. Во-первых, они будут хорошо выделяться в тексте программы, и, во-вторых, благодаря этому программа не даст сбой при введении нового зарезервированного слова.

Открытие и закрытие дескриптора файла

В Perl есть три дескриптора файлов, `STDIN`, `STDOUT` и `STDERR`, которые автоматически открываются для файлов или устройств, установленных родительским процессом программы (вероятно, `shell`). Для открытия дополнительных дескрипторов используется функция `open`. Она имеет следующий синтаксис:

```
open (ДЕСКРИПТОР, "имя" );
```

где *ДЕСКРИПТОР* — новый дескриптор файла, а *имя* — имя файла (или устройства), которое будет связано с новым дескриптором. Этот вызов открывает файл для чтения. Чтобы открыть файл для записи, используйте ту же функцию `open`, но поставьте перед именем файла знак “больше” (как в `shell`):

```
open (OUT, ">выходной_файл" );
```

Мы увидим, как использовать этот дескриптор, в разделе “Использование дескрипторов файлов”. Как и в `shell`, файл можно открыть для добавления, поставив перед именем два знака “больше чем”, т.е.

```
open (LOGFILE, ">>мой_файл_регистрации" );
```

Все формы функции `open` в случае успешного выполнения возвращают значение “истина”, а в случае неудачи — “ложь”. (Например, при попытке открытия файла для чтения выдается значение “ложь”, если файла нет или доступ к нему запрещен; при открытии файла для вывода возвращается значение “ложь”, если файл защищен от записи или если невозможна запись в каталог либо доступ к этому каталогу.)

Закончив работать с дескриптором файла, вы можете закрыть его, воспользовавшись операцией `close`:

```
close (LOGFILE) ;
```

Попытка повторного открытия дескриптора файла приводит к автоматическому закрытию ранее открытого файла. Это же происходит и при выходе из программы. По этой причине многие Perl-программы не обременяют себя выполнением операции `close`. Если же вы хотите, чтобы все было сделано аккуратно или чтобы все данные записывались на диск незадолго до завершения программы, эта операция необходима. Вызов `close` может закончиться неудачей, если диск переполнен, если удаленный сервер, на котором находился файл, стал недоступен либо если возникла какая-нибудь иная проблема. Рекомендуем всегда проверять возвращаемые значения всех системных вызовов.

Небольшое отступление: функция *die*

Считайте этот раздел большой сноской, сделанной посреди страницы.

Дескриптор файла, который не удалось успешно открыть, все равно может использоваться в программе, причем без каких-либо предупреждений*. Если вы читаете данные из дескриптора файла, вы сразу же получите признак конца файла. Если вы записываете данные в дескриптор, эти данные попросту выбрасываются (как обещания участников прошлогодней избирательной кампании).

Скорее всего, вы захотите проверить результат выполнения функции `open` и получить сообщение об ошибке, если этот результат не оправдал ваши ожидания. Естественно, вы можете приправить свою программу разными штучками вроде

```
unless (open (DATAPLACE,">/tmp/dataplace")) {
    print "Sorry, I couldn't create /tmp/dataplace\n";
} else {
    # остальная часть программы
}
```

Но это очень объемная задача, и встречается она достаточно часто, поэтому в Perl для нее предусмотрено специальное сокращение. Функция `die` получает список, который может быть заключен в круглые скобки, выводит этот список (как это делает `print`) на стандартное устройство вывода ошибок, а затем завершает Perl-процесс (тот, который выполняет Perl-программу) с ненулевым кодом выхода (который, как правило, означает, что произошло нечто необычное**). Используя эту функцию, приведенный выше код можно переписать так:

```
unless (open DATAPLACE,">/tmp/dataplace") {
    die "Sorry, I couldn't create /tmp/dataplace\n";
}
# остальная часть программы
```

Можно пойти еще дальше. Вспомним, что для сокращения записи можно использовать операцию `||` (логическое ИЛИ):

```
open(DATAPLACE,">/tmp/dataplace") ||
    die "Sorry, I couldn't create /tmp/dataplace\n";
```

Таким образом, `die` выполняется только в том случае, если значение, получаемое в результате выполнения функции `open`, — “ложь”. Читать это нужно так: “открыть этот файл или умереть!” Это простой способ запомнить, какую логическую операцию использовать — И либо ИЛИ.

* Если вы не выполняете программу с ключом `-w`.

** Фактически `die` просто генерирует исключение, но поскольку мы не показываем вам, как обрабатывать исключения, она ведет себя так, как здесь написано. Подробности см. в главе 3 книги *Programming Perl* (функция `eval`) или на map-странице *perlfunc(1)*.

К сообщению, выдаваемому в случае “смерти” (оно строится на основе аргумента функции `die`) автоматически присоединяется имя Perl-программы и номер строки, поэтому вы можете легко определить, какая именно функция `die` несет ответственность за преждевременный выход из программы. Если же вы не хотите указывать номер строки или имя файла, поставьте в конец “предсмертного” текста символ новой строки. Например:

```
die "you gravy-sucking pigs";
```

выводит файл и номер строки, а

```
die "you gravy-sucking pigs\n";
```

не выводит.

Еще одна удобная штука внутри `die`-строк — переменная `$!`, которая содержит строку с описанием самой последней ошибки операционной системы. Используется она так:

```
open(LOG,">>logfile") || die "cannot append: $!";
```

Например, в результате может быть выдано сообщение "cannot append: Permission denied".

Имеется также функция “вызова при закрытии”, которую большинство пользователей знают как `warn`. Она делает все, что делает `die`, только “не умирает”. Используйте ее для выдачи сообщений об ошибках на стандартный вывод:

```
open(LOG,">>log") || warn "discarding logfile output\n";
```

Использование дескрипторов файлов

После того как дескриптор файла открыт для чтения, из него можно читать строки точно так же, как со стандартного ввода `STDIN`. Например, для чтения строк из файла паролей используется такой код:

```
open (EP, "/etc/passwd");
while (<EP>) {
    chomp;
    print "I saw $_ in the password file'\n";
}
```

Обратите внимание: вновь открытый дескриптор помещен в угловые скобки, аналогично тому как ранее мы использовали `STDIN`.

Если вы открыли дескриптор файла для записи или добавления и хотите ввести в него что-либо (с помощью функции `print`), этот дескриптор нужно

поставить сразу за ключевым словом `print`, перед остальными аргументами. Запятой между дескриптором и остальными аргументами быть не должно:

```
print LOGFILE "Finished item $n of $max\n";
print STDOUT "hi, world'\n"; # как print "hi, world'\n"
```

В этом случае сообщение, начинающееся со слова `Finished`, посылается в дескриптор файла `LOGFILE`, который, предположительно, был открыт в программе ранее. Сообщение `hi, world` направляется на стандартный вывод, как и раньше, когда вы не указывали дескриптор. Мы говорим, что `STDOUT` — это *дескриптор файла по умолчанию* для оператора `print`.

Предлагаем вам способ копирования данных из файла, заданного в переменной `$a`, в файл, указанный в переменной `$b`. Он иллюстрирует почти все, о чем мы рассказывали на последних нескольких страницах.*

```
open(IN,$a) || die "cannot open $a for reading: $!";
open(OUT,">$b") || die "cannot create $b: $!";
while (<IN>) {          # прочитать строку из файла $a в $_
    print OUT $_;       # вывести эту строку в файл $b
}
close(IN) || die "can't close $a: $!";
close(OUT) || die "can't close $b: $!";
```

Операции для проверки файлов

Теперь вы знаете, как открыть дескриптор файла для вывода, уничтожив существующий файл с таким же именем. Предположим, вы хотите удостовериться, что файла с таким именем не существует (чтобы избежать случайного уничтожения своей электронной таблицы или очень важного календаря дней рождений). Если бы вы писали сценарий `shell`, вы использовали бы для проверки существования файла нечто вроде `-e имя_файла`. Аналогичным образом в `Perl` применяется операция `-e $filevar`, которая проверяет факт существования файла, заданного в скалярной переменной `$filevar`. Если этот файл существует, результат — “истина”; в противном случае операция дает “ложь”***. Например:

```
$name = "index.html";
if (-e $name) {
    print "I see you already have a file named $name\n";
} else {
    print "Perhaps you'd like to make a file called $name\n";
}
```

* Хотя при наличии модуля `File::Copy` этот способ оказывается лишним

** Это не совсем хорошо, если вы работаете с `lock`-файлами или если файлы часто появляются и исчезают. В этом случае вам нужно обратиться к функциям `sysopen` и `flock`, которые описаны в книге *Programming Perl*, или изучить примеры, приведенные в главе 19

Операнд операции `-e` — любое скалярное выражение, вычисление которого дает некоторую строку, включая строковый литерал. Вот пример, в котором проверяется наличие файлов `index.html` и `index.cgi` в текущем каталоге:

```
if (-e "index.html" && -e "index.cgi") {  
    print "You have both styles of index files here.\n";  
}
```

Существуют и другие операции. Например, `-r $filevar` возвращает значение “истина”, если заданный в `$filevar` файл существует и может быть прочитан. Операция `-w $filevar` проверяет возможность записи в файл. В следующем примере файл с заданным пользователем именем проверяется на возможность чтения и записи:

```
print "where? ";  
$filename = <STDIN>;  
chomp $filename; # выбросить этот надоедливый символ новой строки  
if (-r $filename && -w $filename) {  
    # файл существует, я могу читать его и записывать в него  
    ...  
}
```

Есть много других операций для проверки файлов. Полный перечень их приведен в таблице 10.1.

Таблица 10.1 Операции для проверки файлов и их описание

Обозначение	Описание
<code>-r</code>	Файл или каталог доступен для чтения
<code>-w</code>	Файл или каталог доступен для записи
<code>-x</code>	Файл или каталог доступен для выполнения
<code>-o</code>	Файл или каталог принадлежит владельцу
<code>-R</code>	Файл или каталог доступен для чтения реальным пользователем, но не “эффективным” пользователем (отличается от <code>-r</code> для программ с установленным битом смены идентификатора пользователя)
<code>-W</code>	Файл или каталог доступен для записи реальным пользователем, но не “эффективным” пользователем (отличается от <code>-w</code> для программ с установленным битом смены идентификатора пользователя)
<code>-X</code>	Файл или каталог доступен для выполнения реальным пользователем, но не “эффективным” пользователем (отличается от <code>-x</code> для программ с установленным битом смены идентификатора пользователя)
<code>-O</code>	Файл или каталог принадлежит реальному пользователю, но не “эффективному” пользователю (отличается от <code>-o</code> для программ с установленным битом смены идентификатора пользователя)

Обозначение	Описание
-e	Файл или каталог существует
-z	Файл существует и имеет нулевой размер (каталоги пустыми не бывают)
-s	Файл или каталог существует и имеет ненулевой размер (значение — размер в байтах)
-f	Данный элемент — обычный файл
-d	Данный элемент — каталог
-l	Данный элемент — символическая ссылка
-S	Данный элемент — порт
-p	Данный элемент — именованный канал (FIFO-файл)
-b	Данный элемент — блок-ориентированный файл (например, монтируемый диск)
-c	Данный элемент — байт-ориентированный файл (например, файл устройства ввода-вывода)
-u	У файла или каталога установлен идентификатор пользователя
-g	У файла или каталога установлен идентификатор группы
-k	У файла или каталога установлен бит-липучка
-t	Выполнение операции isatty() над дескриптором файла дало значение “истина”
-T	Файл — текстовый
-B	Файл — двоичный
-M	Время с момента последнего изменения (в днях)
-A	Время с момента последнего доступа (в днях)
-C	Время с момента последнего изменения индексного дескриптора (в днях)

Большинство этих проверок возвращает просто значение “истина” или “ложь”. О тех, которые этого не делают, мы сейчас поговорим.

Операция `-s` возвращает значение “истина”, если файл непустой, но это значение особого вида. Это длина файла в байтах, которая интерпретируется как “истина” при ненулевом значении.

Операции `-M`, `-A` и `-C` (да-да, в верхнем регистре) возвращают количество дней соответственно с момента последнего изменения файла, доступа к нему и изменения его индексного дескриптора*. (Индексный дескриптор содержит всю информацию о файле; подробности см. на map-странице, посвященной системному вызову `stat`.) Возвращаемое значение — десятичное число,

* Эти значения определяются относительно времени запуска программы, занесенного в системном формате времени в переменную `$^T`. Если запрашиваемое значение относится к событию, которое произошло после начала работы программы, оно может быть отрицательным.

соответствующее прошедшему времени с точностью до 1 секунды: 36 часов возвращается как 1,5 дня. Если при отборе файлов будет выполнено сравнение этого показателя с целым числом (например, с 3), то вы получите только те файлы, которые были изменены ровно столько дней назад, ни секундой раньше и ни секундой позже. Это значит, что для получения всех файлов, значение определенного показателя для которых находится в диапазоне от трех до четырех дней, вам, вероятно, нужно будет использовать операцию сравнения диапазонов*, а не операцию сравнения значений.

Эти операции могут работать не только с именами файлов, но и с дескрипторами. Для этого нужно просто указать в качестве операнда дескриптор файла. Так, чтобы проверить, доступен ли для выполнения файл, открытый как `SOMEFILE`, можно сделать следующее:

```
if (-x SOMEFILE) {  
    # файл, открытый как SOMEFILE, доступен для выполнения  
}
```

Если имя или дескриптор файла не указаны (т.е. даны только операции `-r` или `-s`), то по умолчанию в качестве операнда берется файл, указанный в переменной `$_` (опять эта переменная!). Так, чтобы проверить список имен файлов и установить, какие из них доступны для чтения, нужно просто-напросто написать следующее:

```
foreach (@some_list_of_filenames) {  
    print "$_ is readable\n" if -r;    # то же, что и -r $_  
}
```

Функции *stat* и *lstat*

Вышеупомянутые операции весьма эффективны при проверке различных атрибутов конкретного файла или дескриптора файла, но полную информацию с их помощью получить нельзя. Например, не предусмотрена операция проверки, которая возвращала бы число ссылок на файл. Чтобы добраться до остальных сведений о файле, вызовите функцию `stat`, которая возвращает практически все, что возвращает системный вызов `stat` в POSIX (надеюсь, мы сказали больше, чем вы хотите знать).

Операнд функции `stat` — дескриптор файла или выражение, посредством которого определяется имя файла. Возвращаемое значение — либо `undef`, если вызов неудачен, либо 13-элементный список**, который легче всего описать с помощью такого списка скалярных переменных:

```
($dev,$ino,$mode,$nlink,$uid,$gid,$rdev,  
$size,$atime,$mtime,$ctime,$blksize,$blocks) = stat(...)
```

* Или операцию `int`.

** Если вам тяжело запомнить порядок значений, возвращаемых функцией `stat`, можете обратиться к модулю `File::stat`, впервые введенному в выпуске 5.004. Он обеспечивает доступ к этим значениям следующим образом

```
$file_owner = stat($filename)->uid
```

Имена здесь соответствуют частям структуры `stat`, подробно описанной на man-странице `stat(2)`. Рекомендуем изучить приведенные там подробные пояснения.

Например, чтобы получить идентификаторы пользователя и группы из файла паролей, нужно записать:

```
($uid,$gid) = (stat("/etc/passwd"))[4,5];
```

и этого окажется достаточно.

Вызов функции `stat` с именем символической ссылки возвращает информацию о том, на что указывает эта ссылка, а не сведения о самой ссылке (если только она не указывает на что-то в текущий момент недоступное). Если вам нужна информация о самой символической ссылке (большой частью бесполезная), используйте вместо `stat` функцию `lstat` (которая возвращает те же данные в том же порядке). С элементами, которые не являются символическими ссылками, функция `lstat` работает аналогично `stat`.

Как и в операциях проверки файлов, операнд функций `stat` и `lstat` по умолчанию — `$_`. Это значит, что операция `stat` будет выполняться над файлом, заданным скалярной переменной `$_`.

Упражнения

Ответы приведены в приложении А.

1. Напишите программу чтения имени файла из `STDIN`, открытия этого файла и выдачи его содержимого с предварением каждой строки именем файла и двоеточием. Например, если считано имя `fred`, а файл `fred` состоял из трех строк, `aaa`, `bbb` и `ccc`, вы должны увидеть `fred: aaa`, `fred: bbb` и `fred: ccc`.
2. Напишите программу, которая приглашает ввести имя входного файла, имя выходного файла, образец поиска и заменяющую строку, после чего копирует входной файл в выходной, заменяя все экземпляры образца этой строкой. Попробуйте выполнить эту программу с несколькими файлами. Можете ли вы заменить существующий файл (не проводите эту операцию над чем-то важным!)? Можете ли вы использовать символы регулярных выражений в искомой строке? Можете ли вы использовать `$1` в заменяющей строке?
3. Напишите программу чтения списка имен файлов и выдачи информации о том, какие файлы доступны для чтения, для записи и (или) для выполнения, а какие файлы не существуют. (Каждую проверку можно выполнять для каждого имени файла по мере их чтения или для всей совокупности имен после прочтения их всех. Не забудьте удалять символ новой строки в конце каждого прочитанного имени файла.)
4. Напишите программу чтения списка имен файлов и поиска среди них самого старого. Выведите на экран имя файла и его возраст в днях.

В этой главе:

- Что такое формат
- Определение формата
- Вызов формата
- Еще о поледержателях
- Формат начала страницы
- Изменение в форматах установок по умолчанию
- Упрощения



Форматы

Что такое формат

Помимо всего прочего, Perl — это, как мы уже говорили, “практический язык извлечений и отчетов”. Теперь самое время узнать, почему его называют языком отчетов.

В Perl существует понятие шаблона для написания отчета, который называется *форматом*. В формате определяется постоянная часть (заголовки столбцов, метки, неизменяемый текст и т.д.) и переменная часть (текущие данные, которые вы указываете в отчете). Структура формата близка собственно к структуре вывода и подобна форматированному выводу в Коболе или выводу при помощи клаузы `print using` в некоторых реализациях Бейсика.

Использование формата предполагает выполнение трех операций:

1. Определение формата.
2. Загрузка данных, подлежащих печати, в переменные части формата (поля).
3. Вызов формата.

Чаще всего первый этап выполняется один раз (в тексте программы, чтобы формат определялся во время компиляции)*, а два остальных этапа — многократно.

* Форматы можно создавать и во время выполнения, пользуясь функцией `eval` (см. книгу *Programming Perl*) и map-страницу `perlform(1)`.

Определение формата

Формат задается с помощью специальной конструкции, которая называется определением формата. Определение формата, как и подпрограмма, может стоять в любом месте программы. Выглядит оно так:

```
format имя_формата =
строка_полей
значение_один, значение_два, значение_три
строка_полей
значение_один, значение_два
строка_полей
значение_один, значение_два, значение_три
```

Первая строка содержит зарезервированное слово `format`, за которым следует имя формата и знак равенства (=). Имя формата выбирается из отдельного пространства имен в соответствии с теми же правилами, что и все прочие имена. Поскольку имена форматов в теле программы никогда не используются (кроме как в строковых значениях), вы спокойно можете брать имена, совпадающие с зарезервированными словами. Как вы увидите в следующем разделе, “Вызов формата”, большинство имен форматов будут, вероятно, совпадать с именами дескрипторов файлов (что, конечно, делает их не идентичными зарезервированным словам, и это хорошо).

За первой строкой идет сам *шаблон*, который может включать несколько текстовых строк или быть “пустым”, т.е. не содержать ни одной строки. Конец шаблона обозначается точкой, которая ставится в отдельной строке*. В шаблонах разные пробельные символы интерпретируются по-разному; это один из тех немногих случаев, когда вид пробельных символов (пробел, символ новой строки или знак табуляции) и их количество имеют значение для текста Perl-программы.

Определение шаблона содержит ряд *строк полей*. Каждая строка полей может содержать неизменяемый текст — текст, который будет выведен буквально при вызове формата. Вот пример строки полей с неизменяемым текстом:

```
Hello, my name is Fred Flintstone.
```

Строки полей могут содержать *поледержатели* (*fieldholder*, или переменные полей) для изменяемого текста. Если строка содержит поледержатели, то следующая строка шаблона (которая называется *строкой значений*) содержит ряд скалярных значений — по одному на каждый поледержатель — которые будут вставляться в эти поля. Вот пример строки полей с одним поледержателем и строки значений:

```
Hello, my name is @<<<<<<<<<
$name
```

* В текстовых файлах последняя строка должна заканчиваться символом новой строки.

Поледержатель здесь — @<<<<<<<<. Этот поледержатель задает текстовое поле, которое состоит из 11 символов и выравнивается по левому краю. Подробно поледержатели описаны в разделе “Еще о поледержателях”.

Если в строке полей имеется несколько полейдержателей, то в следующей строке должны быть указаны несколько значений. Эти значения отделяются друг от друга запятыми:

Hello, my name is @<<<<<<<< and I'm @<< years old.
\$name, \$age

Собрав все вместе, мы создадим простой формат для вывода адреса:

```
format ADDRESSLABEL =  
===== ! @<<<<<<<<<<<<<<<<<<<<<< |  
$name | @<<<<<<<<<<<<<<<<<<<<<< |  
$ address | @<<<<<<<<<<<<<<<<<<<<<< |  
| @<<<<<<<<<<<<<<, @< @<<<< |  
$city, $state, $zip  
=====
```

Обратите внимание: строки, состоящие из знаков равенства (в начале и конце формата) полей не содержат, поэтому строк значений после них нет. (Если поставить после такой строки строку значений, она будет интерпретирована как строка полей и поставленную перед ней задачу не выполнит.)

Пробельные символы в строке значений игнорируются. Некоторые предпочитают вставлять в строку значений дополнительные пробельные символы, чтобы выравнивать переменные по поледержателям в предыдущей строке (например, в только что рассмотренном примере переменная `$zip` поставлена таким образом под третьим полем предыдущей строки), но это делается только для красоты. Perl на подобное не реагирует, и на выводимую информацию это никак не влияет.

Текст, стоящий в строке значений после первого символа новой строки, отбрасывается (за исключением особого случая с многострочными поледержателями, который будет рассмотрен ниже).

Определение формата похоже на определение подпрограммы. Оно не содержит немедленно выполняемого кода и может поэтому располагаться в любом месте файла программы. Мы предпочитаем ставить определение формата ближе к концу файла, перед определениями подпрограмм.

Вызов формата

Вызов формата производится с помощью функции `write`. Эта функция получает имя дескриптора файла и генерирует для этого дескриптора текст, используя текущий для данного дескриптора формат. По умолчанию таким форматом является формат с тем же именем, что и у дескриптора файла (так, для дескриптора `STDOUT` используется формат `STDOUT`). Скоро вы узнаете, как этот порядок можно изменить.

Давайте еще раз обратимся к формату адресной этикетки и создадим файл, в котором содержится несколько таких этикеток. Вот фрагмент программы:

```
format ADDRESSLABEL =
=====
| @<<<<<<<<<<<<<<<<<<< |
$name
| @<<<<<<<<<<<<<<<<<<< |
$address
| @<<<<<<<<<<<<<, @< @<<<< |
$city, $state, $zip
=====
```

```
open(ADDRESSLABEL,">labels-to-print") || die "can't create";
open (ADDRESSES,"addresses") || die "cannot open addresses";
while (<ADDRESSES>) {
    chomp; # удалить символ новой строки
    ($name,$address,$city,$state,$zip) = split(/:/);
    # загрузить глобальные переменные
    write (ADDRESSLABEL); # send the output
}
```

Здесь мы видим уже знакомое определение формата, но теперь у нас есть и выполняемый код. Сначала мы открываем дескриптор для выходного файла, который называется `labels-to-print`. Отметим, что имя дескриптора файла (`ADDRESSLABEL`) совпадает с именем формата. Это важно. Затем мы открываем дескриптор для файла, содержащего список адресов. Предполагается, что этот список имеет следующий формат:

Stonehenge:4470 SW Hall Suite 107 :Beaverton:OR:97005
Fred Flintstone:3737 Hard Rock Lane:Bedrock:OZ:999bc

Другими словами, список содержит пять разделенных двоеточиями полей, которые наша программа анализирует описанным ниже образом.

Цикл `while` в программе считывает строку из файла адресов, избавляется от символа новой строки, а затем разбивает то, что осталось, на пять переменных. Обратите внимание: имена переменных — те же самые, которые мы использовали, определяя формат. Это тоже важно.

После загрузки значений всех необходимых для работы с форматом переменных функция `write` вызывает формат. Отметим, что параметром функции `write` является дескриптор файла, в который будет осуществляться запись; кроме того, по умолчанию используется формат с тем же именем.

Каждое поле в формате заменяется соответствующим значением из следующей строки формата. После обработки двух приведенных в примере записей файл `labels-to-print` будет содержать следующее:

```
=====
| Stonehenge                               |
| 4470 SW Hall Suite 107                   |
| Beaverton      , OR 97005               |
=====
| Fred Flintstone                          |
| 3737 Hard Rock Lane                     |
| Bedrock      , OZ 999bc                 |
=====
```

Еще о полевых держателях

Из примеров вы уже поняли, что полевой держатель `@<<<<` обозначает выровненное по левому краю поле, которое содержит пять символов, а `@<<<<<<<<<<` — выровненное по левому краю поле, содержащее одиннадцать символов. Как мы и обещали, опишем полевые держатели более подробно.

Текстовые поля

Большинство полевых держателей начинается со знака `@`. Символы, стоящие после него, обозначают тип поля, а число этих символов (включая `@`) соответствует ширине поля.

Если после знака `@` стоят левые угловые скобки (`<<<<`), поле выровнено по левому краю, т.е. в случае, если значение окажется короче, чем отведенное для него поле, оно будет дополняться справа символами пробела. (Если значение слишком длинное, оно автоматически усекается; структура формата всегда сохраняется.)

Если после знака `@` стоят правые угловые скобки (`>>>>`), поле выровнено по правому краю, т.е. в случае, если значение окажется короче, чем отведенное для него поле, оно будет заполняться пробелами слева.

Наконец, если символы, стоящие после знака `@`, — повторяющаяся несколько раз вертикальная черта (`||||`), то поле центрировано. Это означает, что если значение слишком короткое, производится дополнение пробелами с обеих сторон, в результате значение будет расположено по центру поля.

Числовые поля

Следующий тип полейдержателя — числовое поле с фиксированной десятичной запятой, полезное для больших финансовых отчетов. Это поле также начинается со знака @; за ним следует один или более знаков # с необязательной точкой (она обозначает десятичную запятую). Опять-таки, знак @ считается одним из символов поля. Например:

```
format MONEY =
Assets: @#####.## Liabilities: @#####.## Net:  @#####.##
$assets, $liabilities, $assets-$liabilities
.
```

Эти три числовых поля предусматривают шесть знаков слева от десятичной запятой и два справа (в расчете на суммы в долларах и центах). Обратите внимание на то, что в формате используется выражение — это абсолютно допустимо и встречается очень часто.

Ничего оригинального в Perl больше нет: вам не удастся установить формат с плавающей запятой для вывода значений денежных сумм, нельзя также заключить отрицательные значения или что-нибудь подобное в квадратные скобки. Для этого придется писать отдельную программу, например:

```
format MONEY =
Assets: @<<<<<<<< Liabilities: @<<<<<<<< Net:  @<<<<<<<<
&cool($assets,10), &cool($liab,9), &cool($assets-$liab,10)
.

sub pretty {
    my($n,$width) = @_ ;
    $width -= 2; # учтем отрицательные числа
    $n = sprintf("%.2f",$n); # sprintf описывается в одной из следующих глав
    if ($n < 0) {
        return sprintf ("[%$width.2f]", -$n); # отрицательные числа
                                                # заключаются в квадратные скобки
    } else {
        return sprintf (" %$width.2f ", $n);  # положительные числа выделяются
                                                # пробелами
    }
}

## body of program:
$assets = 32125.12;
$liab = 45212.15;
write (MONEY) ;
```

Многостроковые поля

Как уже упоминалось выше, при подстановке значения, содержащегося в строке значений, Perl обычно заканчивает обработку строки, встретив в этом значении_символ новой строки. Многостроковые полейдержатели позволяют использовать значения, которые представляют собой несколько

строк информации. Эти поледержатели обозначаются комбинацией @*, которая ставится в отдельной строке. Как всегда, следующая строка определяет значение, подставляемое в это поле. В данном случае это может быть выражение, которое дает в результате значение, содержащее несколько строк.

Подставленное значение будет выглядеть так же, как исходный текст: четыре строки значения становятся четырьмя строками выходной информации. Например, приведенный ниже фрагмент программы

```
format STDOUT =
Text Before.
@*
$long_string
Text After.
.

$long_string = "Fred\nBarney\nBetty\nWilma\n";
write;
```

позволяет получить такие выходные данные:

```
Text Before.
Fred
Barney
Betty
Wilma
Text After.
```

Заполненные поля

Следующий вид поледержателя — заполненное поле. Этот поледержатель позволяет создавать целый текстовый абзац, разбиение на строки которого выполнено по границам слов. Для этого используются несколько элементов формата, которые работают в совокупности. Но давайте рассмотрим их по отдельности.

Во-первых, заполненное поле обозначается путем замены маркера @ в текстовом поледержателе на символ ^ (например, ^<<<). Соответствующее значение для заполненного поля (в следующей строке формата) должно быть скалярной переменной*, содержащей текст, а не выражением, которое возвращает скалярное значение. Это объясняется тем, что при заполнении этого поледержателя Perl изменяет значение переменной, а значение выражения изменить очень трудно.

Когда Perl заполняет данный поледержатель, он берет значение этой переменной и “захватывает” из него столько слов (подразумевая разумное определение термина “слово”)**, сколько поместится в этом поле. Эти слова фактически “выдираются” из переменной; после заполнения поля значение переменной представляет собой то, что осталось после удаления слов. Почему делается именно так, вы скоро увидите.

* А также отдельным скалярным элементом массива или хеша, например, \$a[3] или \$h{"fred"}.

** Разделители слов задаются переменной \$:.

Пока что особой разницы между заполненным полем и обычным текстовым полем не видно; мы выводим в поле ровно столько слов, сколько в нем умещается (за исключением того, что мы соблюдаем границу последнего слова, а не просто обрезаем текст по ширине поля). Удобство использования заполненного поля проявляется, когда в одном формате делается несколько ссылок на одну переменную. Взгляните на этот пример:

Обратите внимание: переменная `$comment` появляется четыре раза. Первая строка (строка с полем имени) выводит имя и первые несколько слов значения этой переменной. В процессе вычисления этой строки `$comment` изменяется так, что эти слова исчезают. Вторая строка опять ссылается на ту же переменную (`$comment`) и поэтому получает из нее следующие несколько слов. Это справедливо и для третьей, и для четвертой строк. По сути дела, мы создали прямоугольник, который будет максимально заполнен словами из переменной `$comment`, расположенным в четырех строках.

Чтобы решить эту проблему, нужно использовать признак подавления. Строка, содержащая тильду (~), подавляется (не выводится), если при печати она окажется пустой (т.е. будет содержать только пробельные символы). Сама тильда всегда печатается как пробел и может ставиться в любом месте строки, в котором можно было бы поставить пробел. Перепишем последний пример так:

Теперь, если комментарий занимает всего две строки, третья и четвертая строки будут автоматически подавлены.

Что будет, если комментарий занимает больше четырех строк? Мы могли бы сделать около двадцати копий последних двух строк этого формата, надеясь, что двадцати строк хватит. Это, однако, противоречит идее о том, что Perl помогает лентяям, поэтому специально для них предусмотрено следующее: любая строка, которая содержит две рядом стоящие тильды, автоматически повторяется до тех пор, пока результат не будет полностью пустой строкой. (Эта пустая строка подавляется.) В итоге наше определение формата приобретает такой вид:

```
format PEOPLE =  
Name: @<<<<<<<<< Comment: ^<<<<<<<<<<<<<<<<<<<<  
      $name,                      $comment  
~~                                ^<<<<<<<<<<<<<<<<<<<<  
                                 $comment
```

Таким образом, все будет нормально, сколько бы строк ни занимал комментарий — одну, две или двадцать.

Отметим, что данный критерий прекращения повторения строки требует, чтобы в некоторый момент данная строка стала пустой. Это значит, что в этой строке не нужно размещать постоянный текст (кроме символов пробела и тильды), иначе она никогда не станет пустой.

Формат начала страницы

Многие отчеты в конечном итоге выводятся на печатающее устройство, например на принтер. Принтерная бумага обычно обрезается по размеру страницы, потому что в большинстве своем мы уже давно не пользуемся рулонами. Поэтому в тексте, поступающем на печать, должны, как правило, отмечаться границы страниц, для чего нужно вставлять пустые строки или символы перехода на новую страницу. В принципе, можно было бы взять результат работы Perl-программы и обработать его какой-либо утилитой (может быть, даже написанной на Perl), которая выполняет такую разбивку на страницы. Существует, однако, более легкий способ.

Perl позволяет определять специальный формат начала страницы, с помощью которого запускается режим постраничной обработки. Perl подсчитывает все выходные строки, генерируемые при вызове формата для конкретного дескриптора файла. Если следующая строка не умещается на оставшейся части текущей страницы, Perl выдает символ перехода на новую страницу, за которым автоматически следует вызов формата начала страницы и текст, который выводится с использованием формата для конкретного дескриптора файла. Благодаря этому текст, полученный в результате одного вызова функции `write`, никогда не разбивается на несколько страниц (если только он не настолько велик, что не помещается на одной странице).

Формат начала страницы определяется так же, как и всякий другой формат. Имя формата начала страницы для конкретного дескриптора файла по умолчанию состоит из имени этого дескриптора и символов `_TOP` (обязательно прописных).

Переменная `%` в Perl определяется как количество вызовов формата начала страницы для конкретного дескриптора файла, что позволяет использовать эту переменную в составе формата начала страницы для нумерации страниц. Например, добавление следующего определения формата в предыдущий фрагмент программы предотвращает разрыв адресной этикетки на границах страниц и обеспечивает указание текущего номера страницы:

```
format ADDRESSLABEL_TOP =  
My Addresses -- Page @<  
                %%
```

Длина страницы по умолчанию — 60 строк. Этот параметр можно изменить, присвоив значение специальной переменной, о которой вы вскоре узнаете.

Perl не замечает, если вы выполняете для этого же дескриптора файла функцию `print` из другого места в программе, вследствие чего число строк, которые можно разместить на текущей странице, уменьшается. Вам следует либо переписать свой код, чтобы с помощью одних и тех же форматов выводить на печать всю информацию, либо после выполнения `print` изменить переменную “число строк на текущей странице”. Через минуту мы увидим, как можно изменить это значение.

Изменение в форматах установок по умолчанию

Мы часто говорим об использовании в тех или иных ситуациях значений “по умолчанию”. В Perl имеется способ отмены этих “умолчаний” практически для всех случаев. Давайте поговорим об этом.

Изменение дескриптора файла с помощью функции `select()`

Когда мы говорили о функции `print` в главе 6, я упомянул, что `print` и `print STDOUT` идентичны, потому что `STDOUT` — это установка по умолчанию для `print`. Это не совсем так. Настоящая установка по умолчанию для `print` (а также для `write` и еще нескольких операций, до которых мы скоро доберемся) — это *выбранный в текущий момент дескриптор файла*.

Вначале выбранный в текущий момент дескриптор файла — это `STDOUT`, благодаря чему упрощается отправка данных на стандартный вывод. Изменить выбранный в текущий момент дескриптор файла можно с помощью

функции `select`. В качестве аргумента эта функция принимает отдельный дескриптор файла (или скалярную переменную, которая содержит имя дескриптора файла). Изменение выбранного в текущий момент дескриптора файла влияет на все последующие операции, которые от него зависят. Например:

```
print "hello world\n";      # аналогично print STDOUT "hello world\n"
select (LOGFILE);          # выбрать новый дескриптор файла
print "howdy, world\n";    # аналогично print LOGFILE "howdy, world\n"
print "more for the log\n"; # еще в LOGFILE
select (STDOUT);           # вновь выбрать STDOUT
print "back to stdout\n";   # это идет на стандартный вывод
```

Отметим, что операция `select` — “липкая”; после выбора нового дескриптора он остается “выбранным в текущий момент” до проведения следующей операции `select`.

Таким образом, более удачное определение `STDOUT` по отношению к функциям `print` и `write` будет звучать так: `STDOUT` — выбранный в текущий момент дескриптор по умолчанию, или просто дескриптор *по умолчанию*.

В подпрограммах может возникать необходимость смены выбранного в текущий момент дескриптора файла. Представьте, какое неприятное чувство можно испытать, вызвав подпрограмму и обнаружив, что все тщательно проверенные строки текста уходили куда-то “налево”, потому что подпрограмма, оказывается, изменила выбранный в текущий момент дескриптор файла и не восстановила его! Что же должна делать “хорошо воспитанная” подпрограмма? Если она “знает”, что текущий дескриптор — `STDOUT`, она может восстановить выбранный дескриптор с помощью кода, похожего на приведенный выше. А если программа, которая вызвала подпрограмму, уже изменила выбранный дескриптор файла — что тогда?

Оказывается, значение, возвращаемое функцией `select`, — это строка, которая содержит имя ранее выбранного дескриптора. Получив данное значение, можно впоследствии восстановить этот дескриптор, используя такой код:

```
$oldhandle = select LOGFILE;
print "this goes to LOGFILE\n";
select ($oldhandle);    # восстановить предыдущий дескриптор
```

Действительно, в приведенных выше примерах гораздо проще в качестве дескриптора файла для `print` явным образом указать `LOGFILE`, но некоторые операции, как мы вскоре увидим, требуют именно изменения выбранного в текущий момент дескриптора файла.

Изменение имени формата

Имя формата по умолчанию для конкретного дескриптора файла совпадает с именем этого дескриптора. Для выбранного в текущий момент дескриптора файла этот порядок можно изменить, присвоив новое имя формата специальной переменной `$~`. Можно также проверить значение этой переменной и посмотреть, каков текущий формат для выбранного в текущий момент дескриптора файла.

Например, чтобы использовать формат `ADDRESSLABEL` с дескриптором `STDOUT`, следует просто записать:

```
$~ = "ADDRESSLABEL";
```

А что, если нужно установить для дескриптора `REPORT` формат `SUMMARY`? Для этого необходимо сделать всего лишь следующее:

```
$oldhandle = select REPORT;  
$~ = "SUMMARY";  
select ($oldhandle);
```

Когда в следующий раз мы напишем

```
write (REPORT);
```

то тем самым передадим текст на дескриптор `REPORT`, но в формате `SUMMARY*`.

Обратите внимание на то, что вы сохранили предыдущий дескриптор в скалярной переменной, а затем восстановили его. Этот прием — признак хорошего стиля программирования. В коде реальной программы мы, вероятно, решили бы предыдущий однострочный пример таким же способом, а не предполагали бы, что `STDOUT` — дескриптор по умолчанию.

Изменяя текущий формат для конкретного дескриптора файла, вы можете чередовать в одном отчете много разных форматов.

Изменение имени формата начала страницы

Точно так же, как путем установки переменной `$~` мы можем изменять имя формата для конкретного дескриптора файла, так путем установки переменной `$^` мы можем менять формат начала страницы. Эта переменная содержит имя формата начала страницы для выбранного в текущий момент дескриптора файла и доступна для чтения и записи, т.е. вы можете проверить ее значение и узнать текущее имя формата, а также изменить его, присвоив этой переменной новое значение.

* Объектно-ориентированный модуль `FileHandle`, входящий в состав стандартного дистрибутива Perl, обеспечивает выполнение этой задачи более простым способом.

Изменение длины страницы

Если определен формат начала страницы, длина страницы приобретает особое значение. По умолчанию длина страницы равна 60 строкам, т.е. если результаты работы функции `write` не умещаются до конца 60-й строки, то перед дальнейшим выводом текста автоматически вызывается формат начала страницы.

Иногда 60 строк — не то, что нужно. Этот параметр можно изменить, установив переменную `$=`. Данная переменная содержит текущую длину страницы для выбранного в текущий момент дескриптора файла. Опять-таки, для замены дескриптора `STDOUT` (выбранного в текущий момент дескриптора файла по умолчанию) на другой нужно использовать операцию `select`. Вот как следует изменить дескриптор файла `LOGFILE` так, чтобы страница содержала 30 строк:

```
$old = select LOGFILE;    # выбрать LOGFILE и сохранить старый дескриптор
$= = 30;
select $old;
```

Изменение длины страницы вступает в силу только при следующем вызове формата начала страницы. Если вы установили новую длину перед выводом текста в дескриптор файла в каком-то формате, то все будет работать как надо, потому что формат начала страницы вызывается при первом же вызове функции `write`.

Изменение положения на странице

Если вы выводите свой текст в дескриптор файла с помощью функции `print`, то значение счетчика строк будет неправильным, потому что Perl проводит подсчет строк только для функции `write`. Если вы хотите дать Perl знать, что выводите несколько дополнительных строк, можно настроить внутренний счетчик строк Perl, изменив значение переменной `$-`. Эта переменная содержит число строк, оставшихся на текущей странице для выбранного в текущий момент дескриптора файла. Каждая функция `write` уменьшает число оставшихся строк на число фактически выведенных строк. Когда значение этого счетчика достигает нуля, вызывается формат начала страницы и из переменной `$=` (задающей длину страницы) копируется значение `$-`.

Например, чтобы сообщить Perl, что вы послали в `STDOUT` дополнительную строку, нужно сделать следующее:

```
write; # вызвать формат STDOUT для STDOUT
...;
print "An extra line... oops!\n"; # это идет в STDOUT
$- --; # декрементировать $-, чтобы показать, что в STDOUT пошла строка не из write
...;
write; # сработает, учтя дополнительную строку
```

В начале программы `$-` устанавливается в нуль для каждого дескриптора файла. Это позволяет гарантировать, что формат начала страницы будет первым элементом, вызываемым для каждого дескриптора файла при выполнении первой операции `write`.

Упражнения

Ответы см. в приложении А.

1. Напишите программу, которая открывает файл */etc/passwd* по имени и выводит на экран имя пользователя, идентификатор (номер) пользователя и его реальное имя в форматированных столбцах. Используйте функции *format* и *write*.
2. Добавьте в предыдущую программу формат начала страницы. (Если ваш файл паролей относительно короткий, то, возможно, придется установить длину страницы где-то в 10 строк, чтобы можно было использовать несколько экземпляров начала страницы.)
3. Добавьте в начало страницы номер страницы, чтобы при их выводе указывалось *page 1*, *page 2* и т.д.

В этой главе:

- Перемещение по дереву каталогов
- Развертывание
- Дескрипторы каталогов
- Открытие и закрытие дескриптора каталога
- Чтение дескриптора каталога
- Упражнения

12

Доступ к каталогам

Перемещение по дереву каталогов

Вы уже, вероятно, знакомы с понятием “текущий каталог” и с тем, как использовать команду `cd` shell. В системном программировании для изменения текущего каталога процесса вы производили бы системный вызов `chdir`. В Perl тоже используется это имя.

Функция `chdir` в Perl принимает один аргумент — выражение; при его вычислении определяется имя каталога, который становится текущим. Как и в большинстве других системных вызовов, при успешном изменении текущего каталога на затребованный `chdir` возвращает значение “истина”, а при неудачном исходе — “ложь”. Вот пример:

```
chdir("/etc") || die "cannot cd to /etc ($!)";
```

Круглые скобки не обязательны, поэтому можно обойтись и такой записью:

```
print "where do you want to go? ";
chomp($where = <STDIN>);
if (chdir $where) {
    # получилось
} else {
    # не получилось
}
```

Вы не сможете определить, где вы находитесь, не запустив команду `pwd`*. О запуске команд мы расскажем в главе 14.

* Или не используя функцию `getcwd()` из модуля `Cwd`.

Для каждого процесса* назначается свой текущий каталог. Когда запускается новый процесс, он наследует текущий каталог своего родительского процесса, но на этом вся связь заканчивается. Если Perl-программа меняет свой каталог, это не влияет на родительский shell (или иной объект), который запустил этот Perl-процесс. Точно так же процессы, создаваемые Perl-программой, не влияют на текущий каталог самой этой программы. Текущие каталоги для новых процессов наследуются от текущего каталога Perl-программы.

По умолчанию функция `chdir` без параметра делает текущим начальный каталог, почти так же, как команда `cd` shell.

Развертывание

Если в качестве аргумента в командной строке стоит звездочка (*), то shell (или тот интерпретатор командной строки, которым вы пользуетесь) интерпретирует ее как список имен всех файлов, находящихся в текущем каталоге. Так, если вы дали команду `rm *`, то она удалит все файлы из текущего каталога. (Не делайте этого, если не хотите потом долго приставать к системному администратору с просьбами о восстановлении файлов.) Еще примеры: аргумент `[a-m]*.c` превращается в список имен файлов, находящихся в текущем каталоге, которые начинаются с одной из букв первой половины алфавита и заканчиваются на `.c`, а аргумент `/etc/host*` — в список имен файлов каталога `/etc`, которые начинаются со слова `host`. (Если для вас это ново, то перед дальнейшей работой рекомендуем почитать дополнительную литературу о сценариях shell.)

Преобразование аргументов вроде * или `/etc/host*` в список соответствующих имен файлов называется *развертыванием* (*globbing*). Perl обеспечивает развертывание с помощью очень простого механизма: подлежащий развертыванию образец заключается в угловые скобки или же используется функция `glob`.

```
@a = </etc/host*>
@a = glob("/de/host*");
```

В списочном контексте, как показано выше, результатом развертывания является список всех имен, которые совпадают с образцом (как если бы shell раскрыл указанные аргументы), или пустой список, если совпадения не обнаружено. В скалярном контексте возвращается следующее совпадающее имя, а если совпадений нет — возвращается `undef`; это очень похоже на чтение из дескриптора файла. Например, просмотр имен по одному реализуется с помощью следующего кода:

```
while (defined($nextname = </etc/host*>)) {
    print "one of the files is $nextname\n";
}
```

* Это справедливо для UNIX и большинства других современных операционных систем.

Здесь возвращенные имена файлов начинаются с префикса, соответствующего пути доступа к ним (*/etc/host*), поэтому, если вам нужна только последняя часть имени, придется выделять ее самостоятельно, например:

```
while ($nextname = </etc/host*>) {  
    $nextname =~ s#.#/##;    # удалить часть до последней косой черты  
    print "one of the files is $nextname\n";  
}
```

Внутри аргумента допускается указывать несколько образцов; эти списки развертываются отдельно, а затем конкатенируются так, как будто это один большой список:

```
@fred_barney_files = <fred* barney*>;
```

Другими словами, эта операция развертывания выдает те же значения, которые возвратила бы эквивалентная ей команда *echo* с такими же параметрами.*

Хотя развертывание списка файлов и сопоставление с регулярным выражением осуществляются почти одинаково, специальные символы имеют совершенно разный смысл. Не путайте эти механизмы, иначе будете удивляться, почему вдруг `<\.c$>` не находит все файлы, имена которых заканчиваются на *.c*!

В аргументе функции *glob* перед развертыванием производится интерполяция переменных. С помощью Perl-переменных можно выбирать файлы, задаваемые строкой, вычисляемой во время выполнения:

```
if (-d "/usr/etc") {  
    $where = "/usr/etc";  
} else {  
    $where = "/etc";  
}  
@files = <$where/*>;
```

Здесь переменной *\$where* присваивается одно из двух имен каталогов в зависимости от того, существует каталог */usr/etc* или нет. Затем мы получаем список файлов, находящихся в выбранном каталоге. Обратите внимание: переменная *\$where* является развертываемой, т.е. подлежащими развертыванию символами являются */etc/** или */usr/etc/**.

Есть одно исключение из этого правила: образец `<$var>` (который означает использование в качестве развертываемого выражения переменной *\$var*) должен записываться как `<${var}>`. В причины появления этого исключения сейчас лучше не вдаваться.**

* Для вас не будет сюрпризом узнать о том, что для выполнения развертывания Perl просто запускает C-shell, который раскрывает указанный список аргументов, и производит синтаксический анализ того, что получает обратно.

** Конструкция `<$fred>` читает строку из дескриптора файла, заданного содержимым скалярной переменной *\$fred*. Наряду с некоторыми другими особенностями, не упомянутыми в нашей книге, эта конструкция позволяет использовать косвенные дескрипторы файлов, где имя дескриптора передается и обрабатывается так, как будто это данные.

Дескрипторы каталогов

Если в вашей конкретной разновидности операционной системы имеется библиотечная функция *readdir* или ее функциональный эквивалент, Perl обеспечивает доступ к этой программе (и ее спутникам) с помощью *дескрипторов каталогов*. Дескриптор каталога — это имя из отдельного пространства имен, и предостережения и рекомендации, которые касаются дескрипторов файлов, касаются также и дескрипторов каталогов (нельзя использовать зарезервированные слова, рекомендуется использовать верхний регистр). Дескриптор файла FRED и дескриптор каталога FRED не связаны между собой.

Дескриптор каталога представляет собой соединение с конкретным каталогом. Вместо чтения данных (как из дескриптора файла) вы используете дескриптор каталога для чтения списка имен файлов в этом каталоге. Дескрипторы каталогов всегда открываются только для чтения; нельзя использовать дескриптор каталога для изменения имени файла или удаления файла.

Если функции *readdir()* и ее аналогов в библиотеке нет (и при инсталляции языка Perl никакую замену вы не предусмотрели), то использование любой из этих программ приведет к фатальной ошибке и ваша программа компиляцию не пройдет: она аварийно завершится до выполнения первой строки кода. Perl всегда старается изолировать вас от влияния рабочей среды, но такие чудеса ему не подвластны.

Открытие и закрытие дескриптора каталога

Функция *opendir* работает аналогично библиотечному вызову с тем же именем в C и C++. Ей дается имя нового дескриптора каталога и строковое значение, задающее имя открываемого каталога. Если каталог может быть открыт, *opendir* возвращает значение “истина”; в противном случае она возвращает “ложь”. Вот пример:

```
opendir(ETC, "/etc") || die "Cannot opendir /etc: $!";
```

После этого обычно следуют разного рода манипуляции с дескриптором каталога ETC, но сначала, наверное, нужно разобраться, как закрывать дескриптор каталога. Это делается с помощью функции *closedir*, которая весьма похожа на *close*:

```
closedir(ETC);
```

Как и *close*, *closedir* часто оказывается ненужной, потому что все дескрипторы каталогов автоматически закрываются перед повторным открытием либо в конце программы.

Чтение дескриптора каталога

Открыв дескриптор каталога, мы можем прочитать список имен с помощью функции `readdir`, которая принимает единственный параметр — дескриптор каталога. Каждый вызов `readdir` в скалярном контексте возвращает следующее имя файла (только *основное имя*: в возвращаемом значении никаких косых нет) в порядке, который на первый взгляд кажется случайным*. Если больше имен нет, `readdir` возвращает `undef`** . Вызов `readdir` в списочном контексте возвращает все оставшиеся имена файлов в виде списка с одним именем, приходящимся на каждый элемент. Вот пример, в котором выдается перечень всех имен файлов, содержащихся в каталоге `/etc`:

```
opendir(ETC,"/etc") || die "no etc?: $!";
while ($name = readdir(ETC)) { # скалярный контекст, по одному на цикл
    print "$name\n"; # выводит ., .., passwd, group и т.д.
}
closedir(ETC);
```

А вот как можно получить все имена в алфавитном порядке с помощью функции `sort`:

```
opendir(ETC,"/etc") || die "no etc?: $!";
foreach $name (sort readdir(ETC)) { # списочный контекст с сортировкой
    print "$name\n"; # выводит ., .., passwd, group и т.д.
}
closedir(ETC);
```

В этот список включены имена файлов, которые начинаются с точки. Это не похоже на результат развертывания, выполненного с использованием `<*>`, при котором имена, начинающиеся с точки, не возвращаются. С другой стороны, это похоже на результат работы команды `echo* shell`.

* Точнее говоря — это порядок, в котором имена файлов расположены в каталоге, т.е. тот же “беспорядочный порядок”, в котором вы получаете файлы в ОС UNIX в результате вызова команды `find` или `ls -f`

** Это означает, что при работе с опцией `-w` вам придется использовать цикл `while (defined ($name = readdir (...)))`

Упражнения

Ответы см. в приложении А.

1. Напишите программу, которая открывает каталог, заданный ее параметром, а затем выдает список имен файлов этого каталога в алфавитном порядке. (В случае, если переход в каталог не может быть выполнен, программа должна предупредить об этом пользователя.)
2. Модифицируйте программу так, чтобы она выдавала имена всех файлов, а не только те, имена которых не начинаются с точки. Попробуйте сделать это как с помощью операции разворачивания, так и посредством использования дескриптора каталога.

В этой главе:

- Удаление файла
- Переименование файла
- Создание для файла альтернативных имен: связывание ссылками
- Создание и удаление каталогов
- Изменение прав доступа
- Изменение принадлежности
- Изменение меток времени
- Упражнения

13

Манипулирование файлами и каталогами

В этой главе мы покажем, как можно манипулировать самими файлами, а не только содержащимися в них данными. При демонстрации процедуры доступа к файлам и каталогам мы будем пользоваться семантикой UNIX (а также POSIX и Linux). Есть и другие механизмы доступа к файловым системам, но описываемые здесь средства являются стандартными для современных файловых систем.

Удаление файла

Вы уже научились создавать в Perl файл, открывая его для вывода через дескриптор файла. Сейчас мы освоим опасную процедуру удаления файлов (очень кстати для тринадцатой главы, не правда ли?).

Perl-функция `unlink` (названная по имени системного вызова POSIX) удаляет одно из имен файла (который может иметь и другие имена). Когда удаляется последнее имя файла и ни в одном процессе он не открыт, удаляется и сам файл. Это в точности соответствует тому, что делает UNIX-команда `rm`. Поскольку у файла часто бывает только одно имя (если вы не создавали жесткие ссылки), то в большинстве случаев удаление имени можно считать удалением файла. Приняв это допущение, покажем, как удалить файл *fred*, а затем удалить файл, имя которого вводится во время выполнения программы:

```
unlink ("fred");      # спрашиваем с файлом fred
print "what file do you want to delete? ";
```

```
chomp($name = <STDIN>);
unlink ($name);
```

Функция `unlink` может принимать также список имен, подлежащих удалению:

```
unlink ("cowbird","starling");    # убьем двух зайцев
unlink <*.o>; # как "rm *.o" в shell
```

Операция `<*.o>` выполняется в списочном контексте и создает список имен файлов, которые совпадают с образцом. Это именно то, что нужно передать в `unlink`.

Функция `unlink` возвращает количество успешно удаленных файлов. Если указан только один аргумент и соответствующий ему файл удаляется, то возвращается единица; в противном случае возвращается нуль. Если заданы имена трех файлов, но удалить можно только два, то возвращается два. Установить, какие именно файлы были удалены, на основании возвращенного значения невозможно, поэтому если вам нужно определить, какой файл не удален, удаляйте их по одному. Вот как производится удаление всех объектных файлов (имена которых заканчиваются на `.o`) с выдачей сообщения об ошибке для каждого файла, который удалить нельзя:

```
foreach $file (<*.o>) { # пройти по списку .o-файлов
    unlink($file) || warn "having trouble deleting $file: $!";
}
```

Если `unlink` возвращает 1 (это означает, что единственный указанный файл был удален), то функция `warn` пропускается. Если имя файла не может быть удалено, результат “0” означает “ложь”, поэтому `warn` выполняется. Абстрактно это можно прочитать так: “удали этот файл или сообщи мне о нем”.

Если функция `unlink` дается без аргументов, то по умолчанию вновь используется переменная `$_`. Так, приведенный выше цикл можно записать следующим образом:

```
foreach (<*.o>) { # пройти по списку .o-файлов
    unlink || warn "having trouble deleting $_: $!";
}
```

Переименование файла

В shell UNIX имя файла изменяется с помощью команды `mv`. В Perl эта операция обозначается как `rename($старое,$новое)`. Вот как следует переименовать файл `fred` в `barney`:

```
rename("fred","barney") || die "Can't rename fred to barney: $!";
```

Как и большинство других функций, при успешном выполнении `rename` возвращает значение “истина”, поэтому, чтобы узнать, сработала ли функция `rename`, нужно проверить этот результат.

Когда пользователь вводит *mv файл какой-то_каталог*, команда *mv* делает пару закулисных фокусов и создает полное путевое имя (или, другими словами, полное описание пути к месту размещения файла). Функция *rename* этого делать не умеет. В Perl эквивалентная операция выглядит так:

```
rename("файл", "какой-то_каталог/файл");
```

Обратите внимание: в Perl нужно указать имя файла в новом каталоге явно. Кроме того, команда *mv копирует* файл, когда он переименовывается с одного смонтированного устройства на другое (если вы используете одну из достаточно совершенных операционных систем). Функция *rename* не настолько умна, поэтому вы получите сообщение об ошибке, означающее, что эту проблему нужно решать каким-то иным способом (возможно, посредством вызова команды *mv* с теми же именами). Модуль *File::Find* поддерживает функцию *move*.

Создание для файла альтернативных имен: связывание ссылками

Иногда пользователю нужно, чтобы у файла было два, три, а то и дюжина имен (как будто одного имени файлу не хватает!). Операция присвоения файлу альтернативных имен называется *создание ссылок*. Две основных формы создания ссылок — это создание жестких ссылок и создание символических (или мягких) ссылок. Не все виды файловых систем поддерживают оба типа ссылок или хотя бы один из них. В этом разделе описаны файловые системы, соответствующие стандарту POSIX.

Жесткие и символические ссылки

Жесткая ссылка на файл неотличима от его исходного имени; ни одна из жестких ссылок не является более “реальным именем” для файла, чем любая другая.

Операционная система следит за тем, сколько жестких ссылок обозначают файл в каждый данный момент времени. Когда файл впервые создается, у него имеется одна ссылка. Каждая новая жесткая ссылка увеличивает это число, а каждая удаленная — уменьшает. Когда исчезает последняя ссылка на файл и файл закрывается, он прекращает свое существование.

Каждая жесткая ссылка на файл должна располагаться в той же файловой системе (обычно это диск или часть диска). По этой причине нельзя создать новую жесткую ссылку на файл, находящийся в другой файловой системе.

В большинстве систем применение жестких ссылок для каталогов ограничено. Чтобы структура каталогов имела вид дерева, а не произвольную форму, для каталога допускается наличие только одного имени от корня, ссылки из “точечного” файла на данный каталог, и семейства жестких ссылок “точка-точка” из каждого из его подкаталогов. Если вы попытаетесь

создать еще одну жесткую ссылку на каталог, то получите сообщение об ошибке (если только вы не привилегированный пользователь — тогда вам придется провести всю ночь за восстановлением своей поврежденной файловой системы).

Символическая ссылка — это файл особого вида, который содержит в качестве данных путевое имя. Когда этот файл открывается, операционная система рассматривает его содержимое как заменяющие символы для данного путевого имени и заставляет ядро еще немного порыскать по дереву каталогов, используя новое имя.

Например, если символическая ссылка *fred* содержит имя *barney*, то указание открыть файл *fred* — это, на самом деле, указание открыть файл *barney*. Если *barney* — каталог, то *fred/wilma* обозначает *barney/wilma*.

Содержимое символической ссылки (т.е. имя, на которое указывает символическая ссылка) не обязательно должно обозначать существующий файл или каталог. В момент, когда создается *fred*, существование *barney* вовсе не обязательно. Более того, *barney* может никогда и не появиться! Содержимое символической ссылки может указывать на путь, который ведет за пределы текущей файловой системы, поэтому можно создавать символическую ссылку на файл, находящийся в другой смонтированной файловой системе.

Отслеживая новое имя, ядро может натолкнуться на другую символическую ссылку. Эта новая ссылка содержит новые элементы отслеживаемого пути. Одни символические ссылки могут указывать на другие символические ссылки. Как правило, допускается до восьми уровней символических ссылок, но на практике такое встречается редко.

Жесткая ссылка защищает содержимое файла от уничтожения (потому что она считается одним из имен файла). Символическая же ссылка не может уберечь содержимое файла от исчезновения. Символическая ссылка может указывать на другие смонтированные файловые системы, а жесткая — не может. Для каталога может быть создана только символическая ссылка.

Создание жестких и символических ссылок в Perl

В ОС UNIX жесткие ссылки создают с помощью команды *ln*. Например, команда

```
ln fred bigdumbguy
```

позволяет создать жесткую ссылку из файла *fred* (который должен существовать) на *bigdumbguy*. В Perl это выражается так:

```
link("fred","bigdumbguy") || die "cannot link fred to bigdumbguy";
```

Функция *link* принимает два параметра — старое имя файла и новый псевдоним для этого файла. Если ссылка создана успешно, *link* возвращает значение “истина”. Как и команда *mv*, UNIX-команда *ln* позволяет указывать в качестве псевдонима только каталог (без имени файла). Функция *link*

(как и функция `rename`) не настолько умна, поэтому вы должны указывать полное имя файла явно.

При создании жесткой ссылки старое имя не может быть именем каталога*, а новый псевдоним должен указывать на ту же файловую систему. (Эти ограничения частично обусловили необходимость создания символических ссылок.)

В системах, которые поддерживают символические ссылки, в команде `ln` может использоваться опция `-s`, которая создает символическую ссылку. Например, если необходимо создать символическую ссылку из *barney* на *neighbor* (чтобы обращение к *neighbor* фактически было обращением к *barney*), следует использовать команду

```
ln -s barney neighbor
```

В Perl для этого применяется функция `symlink`:

```
symlink("barney","neighbor") || die "cannot symlink to neighbor";
```

Отметим, что *barney* не обязательно должен существовать — ни сейчас, ни в будущем. В этом случае обращение к *neighbor* возвратит нечто туманное вроде `No such file or directory`.

Когда вы вызываете `ls -l` для каталога, содержащего символическую ссылку, вы получаете как имя этой ссылки, так и информацию о том, куда она указывает. В Perl эту же информацию можно получить с помощью функции `readlink`, которая по принципу работы удивительно похожа на системный вызов с тем же именем: она возвращает имя, на которое указывает заданная символическая ссылка. Так, в результате выполнения операции

```
if (defined($x = readlink("neighbor"))) {  
    print "neighbor points at '$x'\n";  
}
```

вы получите сведения о *barney*, если все нормально. Если выбранная символическая ссылка не существует, не может быть прочитана или вообще не является таковой, `readlink` возвращает `undef` (т.е. в данном случае “ложь”) — именно поэтому мы ее здесь и проверяем.

В системах, не поддерживающих символические ссылки, обе функции — и `symlink`, и `readlink` — потерпят неудачу и выдадут сообщения об ошибке. Perl может “спрятать” от вас некоторые зависящие от конкретной системы особенности, но некоторые все равно проявляются. Это как раз одна из них.

* Если только вы не привилегированный пользователь и не любите забавляться с командой `fsck`, восстанавливая поврежденную файловую систему.

Создание и удаление каталогов

Вы не смогли бы выполнить указанные операции (во всяком случае, в UNIX-системе), не зная о команде *mkdir(1)*, которая создает каталоги, содержащие файлы и другие каталоги. В Perl есть эквивалент этой команды — функция *mkdir*, которая в качестве аргументов принимает имя нового каталога и некое число, определяющее права доступа к созданному каталогу. Права доступа задаются как число, интерпретируемое во внутреннем формате прав доступа. Если вы не знакомы с внутренним форматом прав доступа, обратитесь к man-странице *chmod(2)*. Если вам некогда с этим разбираться, просто укажите права доступа как 0777, и все будет нормально*. Вот пример создания каталога с именем *gravelpit*:

```
mkdir("gravelpit",0777) || die "cannot mkdir gravelpit: $!";
```

UNIX-команда *rmdir(1)* удаляет пустые каталоги. В Perl есть ее эквивалент с тем же именем. Вот как можно сделать Фреда безработным:

```
rmdir("gravelpit") || die "cannot rmdir gravelpit: $!";
```

Хотя эти Perl-операции используют преимущества системных вызовов с такими же именами, они будут выполняться (хотя и чуточку медленнее) даже в системах, не поддерживающих такие вызовы. Perl вызывает утилиты *mkdir* и *rmdir* (или как там они называются у вас в системе) автоматически.

Изменение прав доступа

Права доступа к файлу или каталогу определяют, кто (в широком смысле слова) и что может делать с этим файлом или каталогом. В UNIX общепринятый метод изменения прав доступа к файлу — применение команды *chmod(1)*. (Если вы не знакомы с этой операцией, обратитесь к ее man-странице.) В Perl права доступа изменяются с помощью функции *chmod*. Эта функция получает в качестве аргументов заданный восьмеричным числом режим доступа и список имен файлов и пытается изменить права доступа ко всем этим файлам в соответствии с указанным режимом. Чтобы сделать файлы *fred* и *barney* доступными в режимах чтения и записи для всех пользователей, нужно выполнить такую операцию:

```
chmod(0666,"fred","barney");
```

Режим 0666 обозначает чтение и запись для владельца, группы и прочих пользователей, т.е. как раз то, что нам нужно.

* В данном случае вы не создаете каталог с самыми широкими правами доступа. Определить права доступа вам также поможет текущая маска доступа *umask* вашего процесса. В UNIX-системах см. описание команды shell *umask* или man-страницу *umask(2)*.

Функция `chmod` возвращает число файлов, для которых были успешно изменены права доступа (даже если в результате фактически ничего не изменилось). Таким образом, в отношении контроля ошибок она работает аналогично функции `unlink`. Поэтому, чтобы изменить права доступа к файлам *fred* и *barney* и выполнить контроль ошибок в каждом случае, необходимо использовать следующую конструкцию:

```
foreach $file ("fred","barney") {  
    unless chmod (0666,$file) {  
        warn "hmm... couldn't chmod $file.\$!";  
    }  
}
```

Изменение принадлежности

Каждый файл в файловой системе (обычный, каталог, файл устройства и т.д.) имеет владельца и группу. Эти параметры определяют, кому принадлежат права доступа, установленные для файла по категориям “владелец” и “группа” (чтение, запись и (или) выполнение). Владелец и группа определяются в момент создания файла, но при определенных обстоятельствах вы можете изменить их. (Эти обстоятельства зависят от конкретной разновидности UNIX, с которой вы работаете; подробности см. на map-странице *chown*.)

Функция `chown` получает идентификатор пользователя (UID), идентификатор группы (GID) и список имен файлов и пытается изменить принадлежность каждого из перечисленных файлов в соответствии с указанными идентификаторами. Успешному результату соответствует ненулевое значение, равное числу файлов, принадлежность которых изменена (как в функциях `chmod` и `unlink`). Обратите внимание: вы одновременно меняете и владельца, и группу. Если какой-то из этих идентификаторов вы менять не хотите, поставьте вместо него -1. Помните, что нужно использовать числовые UID и GID, а не соответствующие символические имена (хотя команда *chmod* и принимает такие имена). Например, если UID файла *fred* — 1234, а идентификатор группы *stoners*, которой этот файл принадлежит по умолчанию, — 35, то в результате применения следующей команды файлы *slate* и *granite* переходят к пользователю *fred* и его группе:

```
chown(1234, 35, "slate", "granite"); # то же, что и  
# chown fred slate granite  
# chgrp stoners slate granite
```

В главе 16 вы узнаете, как преобразовать *fred* в 1234 и *stoners* в 35.

Изменение меток времени

С каждым файлом связан набор из трех меток времени. Мы вкратце упоминали о них, когда говорили о том, как получить информацию о файле: это время последнего доступа, время последнего изменения и время последнего изменения индексного дескриптора. Первым двум меткам времени можно присвоить произвольные значения с помощью функции `utime` (которая соответствует системному вызову `utime` в ОС UNIX). При установке двух этих значений третья метка автоматически устанавливается в текущее время, поэтому отдельного способа для ее установки не предусмотрено.

Эти значения устанавливаются во внутреннем формате времени, а именно в количестве секунд, прошедших после полуночи 1 января 1970 года по среднегринвичскому времени. Когда мы писали нашу книгу, эта цифра достигла 800 миллионов с небольшим. (Во внутреннем формате она представляется как 32-разрядное число без знака, и если все мы не перейдем на 64-разрядные (и более) машины, то переполнение наступит где-то в следующем столетии. У нас будут гораздо более серьезные проблемы в 2000-м году*.)

Функция `utime` работает аналогично функциям `chmod` и `unlink`. Она получает список имен файлов и возвращает число файлов, параметры времени которых были изменены. Вот что нужно сделать, чтобы файлы *fred* и *barney* выглядели так, будто они изменялись в недавнем прошлом:

```
$atime = $mtime = 700_000_000; # некоторое время назад
utime($atime, $mtime, "fred", "barney")
```

Никакого “разумного” значения для меток времени нет: можно сделать так, чтобы файл выглядел сколь угодно старым, или чтобы казалось, будто он был изменен в далеком будущем (это полезно, если вы пишете научно-фантастические рассказы). Вот как, например с помощью функции `time` (которая возвращает текущее время как метку времени UNIX) можно сделать так, чтобы казалось, будто файл *max_headroom* был изменен спустя 20 минут после текущего момента времени:

```
$when = time() + 20*60; # 20 минут с текущего момента
utime($when, $when, "max_headroom");
```

* Perl-функции `localtime` и `gmtime` работают так, как в C они возвращают год, из которого вычтена цифра 1900. В 2003-м году `localtime` выдаст год как 103.

Упражнения

Ответы к упражнениям см. в приложении А.

1. Напишите программу, которая работает как утилита *rm*, удаляя файлы, имена которых были заданы как аргументы командной строки при вызове программы. (Никакие опции команды *rm* вам для этого не понадобятся.) Обязательно проверьте эту программу с почти пустым каталогом, чтобы случайно не удалить нужные файлы! Помните, что аргументы командной строки при запуске программы извлекаются из массива `@ARGV`.
2. Напишите программу, которая работает как утилита *mv*, переименовывая первый аргумент командной строки во второй аргумент. (Никакие опции команды *mv* вам для этого не нужны, и аргументов нужно всего два.) Можете также рассмотреть случай переименования, когда целевым объектом является каталог.
3. Напишите программу, которая работает, как *ln*, создавая жесткую ссылку из первого аргумента командной строки на второй аргумент. (Никакие опции команды *ln* вам для этого не нужны, и аргументов нужно всего два.)
4. Если у вас есть символические ссылки, модифицируйте программу из предыдущего упражнения так, чтобы она работала с необязательным ключом `-s`.
5. Если у вас есть символические ссылки, напишите программу, которая ищет в текущем каталоге все файлы, на которые есть такие ссылки, и выводит их имена и значения ссылок так, как это делает *ls -l* (имя \rightarrow значение). Создайте в текущем каталоге несколько символических ссылок и проверьте программу.

В этой главе:

- Использование функций *system* и *exec*
- Использование обратных кавычек
- Использование процессов как дескрипторов файлов
- Использование функции *fork*
- Сводка операций, проводимых над процессами
- Передача и прием сигналов
- Упражнения

Использование функций *system* и *exec*

Когда вы из командной строки shell задаете выполнение какой-либо команды, он обычно создает новый процесс. Этот новый процесс становится порожденным процессом shell, выполняется независимо, но в координации с последним.

Аналогичным образом Perl-программа в состоянии запускать новые процессы и может делать это, как и большинство других операций, несколькими способами.

Самый простой способ запуска нового процесса — использовать для этого функцию *system*. В простейшей форме эта функция передает в совершенно новый shell */bin/sh* одну строку, которая будет выполняться как команда. По выполнении команды функция *system* возвращает код завершения данной команды (если все было нормально — это, как правило, 0). Вот пример того, как Perl-программа выполняет команду *date* с помощью shell*:

```
system("date");
```

Здесь мы не проверяем возвращаемое значение, но неудачный исход выполнения команды *date* вряд ли возможен.

Куда идет результат этой команды? Откуда поступают исходные данные, если они нужны команде? Хорошие вопросы, и ответы на них позволяют узнать, чем отличаются различные способы создания процессов.

* В данном случае shell фактически не используется, поскольку Perl сам выполняет операции shell, если командная строка достаточно проста, как в данном случае

Три стандартных файла для функции `system` (стандартный ввод, стандартный вывод и стандартный вывод ошибок) наследуются от Perl-процесса. Таким образом, результат выполнения команды `date` в приведенном выше примере направляется туда же, куда поступает результат выполнения функции `print STDOUT` — скорее всего, на дисплей вызвавшего ее пользователя. Поскольку вы запускаете `shell`, то можете переадресовать стандартный вывод, пользуясь обычными для `/bin/sh` операциями переадресации. Например, чтобы направить результаты работы команды `date` в файл `right_now`, нужно сделать что-то вроде этого:

```
system("date >right_now") && die "cannot create right_now";
```

На этот раз мы не только посылаем результат команды `date` в файл, выполняя переадресацию в `shell`, но и проверяем статус возврата. Если статус возврата — значение “истина” (не ноль), это значит, что с командой `shell` что-то произошло, и функция `die` выполнит свою миссию. Данное правило обратно обычным правилам выполнения операций в Perl: ненулевое возвращаемое значение операции `system`, как правило, указывает на какую-то ошибку.

Аргументом функции `system` может быть все, что пригодно для передачи в `/bin/sh`, поэтому можно задавать сразу несколько команд, разделяя их точками с запятой или символами новой строки. Процессы, после которых указан символ `&`, запускаются, но программа не ждет их завершения, т.е. в данном случае все происходит аналогично тому, как если бы вы ввели в `shell` строку, которая заканчивается символом `&`.

Вот пример задания команд `date` и `who` в `shell` с передачей результатов в файл, заданный Perl-переменной. Все это выполняется в фоновом режиме, чтобы для продолжения выполнения Perl-сценария не нужно было ждать завершения данного процесса.

```
$where = "who_out." ++$1; # получить новое имя файла
system "(date; who) >$where &";
```

В этом случае функция `system` возвращает код выхода `shell` и показывает таким образом, успешно ли был запущен фоновый процесс, но не сообщает, были ли успешно выполнены команды `date` и `who`. В этой заключенной в двойные кавычки строке производится интерполяция переменных, поэтому переменная `$where` заменяется своим значением (это делает Perl, а не `shell`). Если бы вы хотели обратиться к переменной `shell` с именем `$where`, вам нужно было бы поставить перед знаком доллара обратную косую или использовать строку в одинарных кавычках.

Помимо стандартных дескрипторов файлов, порожденный процесс наследует от родительского процесса много других вещей. Это текущее значение, заданное командой `umask`, текущий каталог и, конечно, идентификатор пользователя.

Кроме того, порожденный процесс наследует все переменные среды. Эти переменные обычно изменяются командой `ssh setenv` или же соответствующим

присваиванием и командой *export shell (/bin/sh)*. Переменные среды используются многими утилитами, включая сам *shell*, для изменения порядка работы этих утилит и управления ими.

В Perl предусмотрена возможность проверки и изменения текущих переменных среды посредством специального хеша, который называется `%ENV`. Каждый ключ этого хеша соответствует имени переменной среды, а соответствующее значение — значению переменной. Содержимое данного хеша отражает параметры среды, переданные Perl родительским *shell*; изменение хеша изменяет параметры среды, которую использует Perl и порожденные им процессы, но не среды, используемой родительскими процессами.

Вот простая программа, которая работает, как *printenv*:

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

Обратите внимание: знак равенства здесь — это не символ операции присваивания, а просто текстовый символ, с помощью которого функция *print* выдает сообщения вида `TERM=xterm` или `USER=merlyn`.

Вот фрагмент программы, с помощью которого значение переменной `PATH` изменяется таким образом, чтобы поиск команды *grep*, запущенной функцией *system*, производился только в “обычных” местах:

```
$oldPATH = $ENV{"PATH"};           # сохранить предыдущий путь  
$ENV{"PATH"} = "/bin:/usr/bin:/usr/ucb"; # ввести известный путь  
system("grep fred bedrock >output"); # запустить команду  
$ENV{"PATH"} = $oldPATH;           # восстановить предыдущий путь
```

Как много текста придется набирать! Гораздо быстрее будет просто установить локальное значение для этого элемента хеша.

Несмотря на наличие некоторых недостатков, операция *local* может делать одну вещь, которая не под силу операции *my*: она способна присваивать временное значение одному элементу массива или хеша.

```
{  
    local $ENV{"PATH"} = "/bin:/usr/bin:/usr/ucb";  
    system("grep fred bedrock >output");  
}
```

Функция *system* может принимать не один аргумент, а список аргументов. В этом случае Perl не передает список аргументов в *shell*, а рассматривает первый аргумент как подлежащую выполнению команду (при необходимости производится ее поиск согласно переменной `PATH`), а остальные аргументы — как аргументы команды без обычной для *shell* интерпретации. Другими словами, вам не нужно заключать в кавычки пробельные символы и беспокоиться об аргументах, которые содержат угловые скобки, потому что все это — просто символы, передаваемые в программу. Таким образом, следующие две команды эквивалентны:

```
system "grep 'fred flintstone' buffaloes"; # с использованием shell  
system "grep","fred flintstone","buffaloes"; # без использования shell
```


Применение в функции `system` списка, а не одной строки, экономит также один процесс `shell`, поэтому поступайте так при любой возможности. (Если форма функции `system` с одним аргументом достаточно проста, Perl сам оптимизирует код, полностью убирая вызов `shell` и обращаясь к соответствующей программе непосредственно, как если бы вы использовали вызов функции с несколькими аргументами.)

Вот еще один пример эквивалентных форм:

```
@cfiles = ("fred.c", "barney.c");           # что компилировать
@options = ("-DHARD", "-DGRANITE");         # опции
system "cc -o slate @options @cfiles";      # c shell
system "cc", "-o", "slate", @options, @cfiles; # без shell
```

Использование обратных кавычек

Еще один способ запуска процесса — заключить командную строку для `/bin/sh` в обратные кавычки. Как и в `shell`, этот механизм запускает команду и ожидает ее завершения, получая данные со стандартного вывода по мере их поступления:

```
$now = "the time is now".`date`;           # получает текст и дату
```

Значение переменной `$now` теперь представляет собой текст `the time is now` и результат выполнения команды `date(1)` (включая конечный символ новой строки):

```
the time is now Fri Aug 13 23:59:59 PDT 1996
```

Если взятая в обратные кавычки команда используется не в скалярном, а в списочном контексте, то возвращается список строковых значений, каждое из которых представляет собой строку (оканчивающуюся символом новой строки*) из результата выполнения команды. В примере с командой `date` у нас был бы всего один элемент, потому что она выдала всего одну строку текста. Результат работы команды `who` выглядит так:

```
merlyn    tty42    Dec      7      19:41
fred      tty1A    Aug     31      07:02
barney    tty1F    Sep      1      09:22
```

Вот как можно получить этот результат в списочном контексте:

```
foreach $_ (`who`) {      # один раз для каждой строки текста из who
    ($who, $where, $when) = /(\\S+)\\s+(\\S+)\\s+(.*)/;
    print "$who on $where at $when\\n";
}
```

* Или символом, который у вас занесен в переменную `$/`

При каждом выполнении этого цикла используется одна строка выходных данных команды *who*, потому что взятая в обратные кавычки команда интерпретируется в списочном контексте.

Стандартный ввод и стандартный вывод ошибок команды, взятой в обратные кавычки, наследуются от Perl-процесса*. Это значит, что обычно стандартный вывод таких команд вы можете получить как значение строки, заключенной в обратные кавычки. Одна из распространенных операций — объединение стандартного вывода ошибок со стандартным выводом, чтобы команда в обратных кавычках “подбирала” их оба. Для этого используется конструкция `shell 2>&1`:

```
die "rm spoke!" if rm fred 2>&1 ;
```

Здесь Perl-процесс завершается, если *rm* посылает какое-нибудь сообщение — либо на стандартный вывод, либо на стандартный вывод ошибок, потому что результат больше не будет пустой строкой (пустая строка соответствовала бы значению “ложь”).

Использование процессов как дескрипторов файлов

Следующий способ запуска процесса — создание процесса, который выглядит как дескриптор файла (аналогично библиотечной подпрограмме *open*(3), если вы с ней знакомы). Мы можем создать для процесса дескриптор файла, который либо получает результат работы процесса, либо подает в него входные данные**. Ниже приведен пример создания дескриптора файла для процесса *who*(1). Поскольку этот процесс выдает результат, который мы хотим прочитать, мы создаем дескриптор файла, открытый для чтения:

```
open(WHOPROC, "who|");      # открыть who для чтения
```

Обратите внимание на вертикальную черту справа от *who*. Эта черта информирует Perl о том, что данная операция `open` относится не к имени файла, а к команде, которую необходимо запустить. Поскольку черта стоит справа от команды, данный дескриптор файла открывается для чтения. Это означает, что предполагается прием данных со стандартного вывода команды *who*. (Стандартный ввод и стандартный вывод ошибок продолжают использоваться совместно с Perl-процессом.) Для остальной части программы дескриптор `WHOPROC` — это просто дескриптор файла, который открыт для

* На самом деле все не так просто. См. соответствующий ответ в разделе 8 сборника часто задаваемых вопросов по Perl (“Как перехватить `STDERR` из внешней команды?”). Если у вас Perl версии 5.004, этот сборник распространяется как обычная man-страница — в данном случае *perlfaq8*(1).

** Но не одновременно. Примеры двунаправленной связи приведены в главе 6 книги *Programming Perl* и на man-странице *perlipc*(1).

чтения, что означает возможность выполнения над файлом всех обычных операций ввода-вывода. Вот как можно прочитать данные из команды *who* в массив:

```
@whosaid = <WHOPROC>;
```

Аналогичным образом для запуска команды, которой необходимы входные данные, мы можем открыть дескриптор файла процесса для записи, поставив вертикальную черту слева от команды, например:

```
open(LPR,"|lpr -Pslatewriter");
print LPR @rockreport;
close(LPR);
```

В этом случае после открытия LPR мы записываем в него данные и закрываем данный файл. Открытие процесса с дескриптором файла позволяет выполнять команду параллельно с Perl-программой. Задание для дескриптора файла команды *close* заставляет Perl-программу ожидать завершения процесса. Если дескриптор не будет закрыт, процесс может продолжаться даже после завершения Perl-программы.

Открытие процесса для записи предполагает, что стандартный ввод команды будет получен из дескриптора файла. Стандартный вывод и стандартный вывод ошибок используются этим процессом совместно с Perl. Как и прежде, вы можете использовать переадресацию ввода-вывода в стиле */bin/sh*. Вот как в нашем последнем примере можно отбрасывать сообщения об ошибках команды *lpr*:

```
open(LPR,"|lpr -Pslatewriter >/dev/null 2>&1");
```

С помощью операции *>/dev/null* обеспечивается отбрасывание стандартного вывода путем переадресации его на нулевое устройство. Операция *2>&1* обеспечивает передачу стандартного вывода ошибок туда, куда направляется стандартный вывод, поэтому сообщения об ошибках также отбрасываются.

Можно даже объединить все эти фрагменты программы и в результате получить отчет обо всех зарегистрированных пользователях, кроме Фреда:

```
open (WHO,"who|");
open (LPR,"|lpr - Pslatewriter");
while (<WHO>) {
    unless (/fred/) {      # не показывать имя Фред
        print LPR $_;
    }
}
close WHO;
close LPR;
```

Считывая из дескриптора *WHO* по одной строке, этот фрагмент кода выводит в дескриптор *LPR* все строки, которые не содержат строкового значения *fred*. В результате на принтер выводятся только те строки, которые не содержат имени *fred*.

Вовсе не обязательно указывать в команде `open` только по одной команде за один прием. В ней можно задать сразу весь конвейер. Например, следующая строка запускает процесс `ls(1)`, который передает свои результаты по каналу в процесс `tail(1)`, который, в свою очередь, передает свои результаты в дескриптор файла `WHOPR`:

```
open(WHOPR, "ls | tail -r |");
```

Использование функции *fork*

Еще один способ создания нового процесса — клонирование текущего Perl-процесса с помощью UNIX-функции `fork`. Функция `fork` делает то же самое, что и системный вызов `fork(2)`: создает клон текущего процесса. Этот клон (он называется порожденным процессом, а оригинал — родительским) использует тот же выполняемый код, те же переменные и даже те же открытые файлы. Различаются эти два процесса по возвращаемому значению функции `fork`: для порожденного процесса оно равно нулю, а для родительского — ненулевое (или `undef`, если этот системный вызов окажется неудачным). Ненулевое значение, получаемое родительским процессом, — это не что иное как идентификатор порожденного процесса. Вы можете проверить возвращаемое значение и действовать соответственно:

```
if (!defined($child_pid = fork())) {  
    die "cannot fork: $!";  
} elsif ($pid) {  
    # я — родительский процесс  
} else {  
    # я — порожденный процесс  
}
```

Чтобы максимально эффективно использовать этот клон, нам нужно изучить еще несколько функций, которые весьма похожи на своих UNIX-тезок: это функции `wait`, `exit` и `exec`.

Самая простая из них — функция `exec`. Это почти то же самое, что и функция `system`, за тем исключением, что вместо запуска нового процесса для выполнения shell-команды Perl заменяет текущий процесс на shell. После успешного выполнения `exec` Perl-программа исчезает, поскольку вместо нее выполняется затребованная программа. Например,

```
exec "date";
```

заменяет текущую Perl-программу командой `date`, направляя результат этой команды на стандартный вывод Perl-программы. После завершения команды `date` делать больше нечего, потому что Perl-программа давно исчезла.

Все это можно рассматривать и по-другому: функция `system` похожа на комбинацию функции `fork` с функцией `exec`, например:

```
# МЕТОД 1... использование system:
system("date");
# МЕТОД 2... использование fork/exec:
unless (fork) {
    # fork выдала нуль, поэтому я — порожденный процесс и я выполняю:
    exec("date");    # порожденный процесс становится командой date
}
```

Использовать `fork` и `exec` таким способом — не совсем правильно, потому что команда `date` и родительский процесс “пыхтят” одновременно, их результаты могут перемешаться и испортить все дело. Как дать родительскому процессу указание подождать, пока не завершится порожденный процесс? Именно это и делает функция `wait`; она ждет завершения данного (да и любого, если быть точным) порожденного процесса. Функция `waitpid` более разборчива: она ждет завершения не любого, а определенного порожденного процесса:

```
if (defined($kidpid = fork())) {
    # fork возвратила undef, т.е. неудача
    die "cannot fork: $!";
} elsif ($pid == 0) {
    # fork возвратила 0, поэтому данная ветвь — порожденный процесс
    exec("date");
    # если exec терпит неудачу, перейти к следующему оператору
    die "can't exec date: $!";
} else {
    # fork возвратила не 0 и не undef,
    # поэтому данная ветвь — родительский процесс
    waitpid($kidpid, 0);
}
```

Если все это кажется вам слишком сложным, изучите системные вызовы `fork(2)` и `exec(2)`, отыскав материалы о них в каком-нибудь руководстве по ОС UNIX, потому что Perl просто передает вызовы этих функций прямо в системные вызовы UNIX.

Функция `exit` обеспечивает немедленный выход из текущего Perl-процесса. Она используется для прерывания Perl-программы где-нибудь посередине или — вместе с функцией `fork` — для выполнения Perl-кода в процессе с последующим выходом. Вот пример удаления нескольких файлов из каталога `/tmp` в фоновом режиме с помощью порожденного Perl-процесса:

```
unless (defined ($pid = fork)) {
    die "cannot fork: $!";
}
unless ($pid) {
    unlink </tmp/badrock.*>;    # удалить эти файлы
    exit;                        # порожденный процесс останавливается здесь
}
                                # родительский процесс продолжается здесь
waitpid($pid, 0);    # после уничтожения порожденного процесса нужно все убрать
```

Без использования функции `exit` порожденный процесс продолжал бы выполнять Perl-код (со строки “# родительский процесс продолжается здесь”) — а как раз этого нам и не нужно.

Функция `exit` может иметь необязательный параметр, служащий числовым кодом выхода, который воспринимается родительским процессом. По умолчанию выход производится с нулевым кодом, показывающим, что все прошло нормально.

Сводка операций, проводимых над процессами

Операции, служащие для запуска процессов, перечислены в таблице 14.1.

Таблица 14.1 Операции запуска процессов

Операция	Стандартный ввод	Стандартный вывод	Стандартный вывод ошибок	Нужно ли ожидать завершения процесса
<code>System()</code>	Наследуется от программы	Наследуется от программы	Наследуется от программы	Да
Строка в обратных кавычках	Наследуется от программы	Принимается как строковое значение	Наследуется от программы	Да
Запуск процесса как дескриптора файла для вывода при помощи команды <code>open()</code>	Соединен с дескриптором файла	Наследуется от программы	Наследуется от программы	Только во время выполнения <code>close()</code>
Запуск процесса как дескриптора файла для ввода при помощи команды <code>open()</code>	Наследуется от программы	Соединен с дескриптором файла	Наследуется от программы	Только во время выполнения <code>close()</code>
<code>fork</code> , <code>exec</code> , <code>wait</code> , <code>waitpid</code>	Выбирается пользователем	Выбирается пользователем	Выбирается пользователем	Выбирается пользователем

Самый простой способ создать процесс — использовать для этого функцию `system`. На стандартный ввод, вывод и вывод ошибок это не влияет (они наследуются от Perl-процесса). Строка в обратных кавычках создает процесс и передает данные со стандартного вывода этого процесса как строковое значение для Perl-программы. Стандартный ввод и стандартный вывод ошибок не изменяются. Оба эти метода требуют завершения процесса до выполнения другого кода.

Простой способ получить асинхронный процесс (процесс, который позволяет продолжать выполнение Perl-программы до своего завершения) — открыть команду как дескриптор файла с созданием канала для стандартного ввода или стандартного вывода этой команды. Команда, открытая как дескриптор файла для чтения, наследует стандартный ввод и стандартный вывод ошибок от Perl-программы, команда, открытая как дескриптор файла для записи, наследует от Perl-программы стандартный вывод и стандартный вывод ошибок.

Самый гибкий способ запустить процесс — заставить программу вызвать функции `fork`, `exec` и `wait` или `waitpid`, которые полностью соответствуют своим UNIX-тезкам. С помощью этих функций вы можете запустить какой-либо процесс синхронно или асинхронно, а также конфигурировать по своему усмотрению стандартный ввод, стандартный вывод и стандартный вывод ошибок*.

Передача и прием сигналов

Один из вариантов организации межпроцессного взаимодействия основан на передаче и приеме сигналов. Сигнал — это одноразрядное сообщение (означающее, что “произошло данное событие”), которое посылается в процесс из другого процесса или из ядра. Сигналам присваиваются номера, обычно из диапазона от единицы до небольшого числа, например 15 или 31. Одни сигналы (фиксированные) имеют предопределенное значение и посылаются в процесс автоматически при возникновении определенных обстоятельств (например, при сбоях памяти или в исключительных ситуациях, возникающих при выполнении операций с плавающей запятой). Другие сигналы генерируются исключительно пользователем из других процессов, но не из всех, а только из тех, которые имеют разрешение на передачу сигналов. Передача сигнала разрешается только в том случае, если вы являетесь привилегированным пользователем или если передающий сигнал процесс имеет тот же идентификатор пользователя, что и принимающий.

Ответ на сигнал называется *действием сигнала*. Фиксированные сигналы выполняют определенные действия по умолчанию, например, осуществляют прерывание или приостановку процесса. Остальные сигналы по умолчанию

* Полезно также знать о формах типа `open(STDERR, ">&STDOUT")` используемых для точной настройки дескрипторов файлов. См. пункт `open` в главе 3 книги *Programming Perl* или на той же странице *perlfunc(1)*.

полностью игнорируются. Почти для любого сигнала действие по умолчанию может быть переопределено, с тем чтобы данный сигнал либо игнорировался, либо перехватывался (с автоматическим вызовом указанной пользователем части кода).

Все это аналогично тому, что делается и в других языках программирования, но сейчас излагаемый материал приобретет специфический Perl-оттенок. Когда Perl-процесс перехватывает сигнал, асинхронно и автоматически вызывается указанная вами подпрограмма, моментально прерывая выполнявшийся до нее код. Когда эта подпрограмма завершается, выполнение прерванного кода возобновляется, как будто ничего не случилось (за исключением появления результатов действий, выполненных этой подпрограммой, — если она вообще что-нибудь делала).

Обычно подпрограмма-обработчик сигнала делает одно из двух: прерывает программу, выполнив “очистку”, или устанавливает какой-то флаг (например, глобальную переменную), которую данная программа затем проверяет*.

Для того чтобы зарегистрировать подпрограммы-обработчики сигналов в Perl, нужно знать имена сигналов. После регистрации обработчика сигнала Perl при получении этого сигнала будет вызывать выбранную подпрограмму.

Имена сигналов определяются на man-странице *signal(2)*, а также, как правило, в подключаемом C-файле */usr/include/sys/signal.h*. Эти имена обычно начинаются с букв SIG, например SIGINT, SIGQUIT и SIGKILL. Чтобы объявить подпрограмму `my_sigint_catcher()` обработчиком сигнала SIGINT, мы должны установить соответствующее значение в специальном хеше %SIG. В этом хеше в качестве значения ключа INT (это SIGINT без SIG) следует указать имя подпрограммы, которая будет перехватывать сигнал SIGINT:

```
$SIG{'INT'} = 'my_sigint_catcher';
```

Но нам понадобится также определение этой подпрограммы. Вот пример простого определения:

```
sub my_sigint_catcher {  
    $saw_sigint = 1;    # установить флаг  
}
```

Данный перехватчик сигналов устанавливает глобальную переменную и сразу же возвращает управление. Выполнение программы продолжается с той позиции, в которой оно было прервано. Обычно сначала обнуляется флаг `$saw_sigint`, соответствующая подпрограмма определяется как перехватчик сигнала SIGINT, а затем следует код основной программы, например:

```
$saw_sigint = 0;          # очистить флаг  
$SIG{'INT'} = 'my_sigint_catcher';    # зарегистрировать перехватчик
```

* Попытка выполнения действий более сложных, чем вышеописанные, вероятнее всего, запутает ситуацию; большинство внутренних механизмов Perl “не любят”, когда их вызывают одновременно в основной программе и из подпрограммы. Ваши системные библиотеки тоже этого “не любят”.


```
foreach (@huge_array) {
    # что-нибудь сделать
    # еще что-нибудь сделать
    # и еще что-нибудь
    if ($saw_sigint) { # прерывание нужно?
        # здесь "очистка"
        last;
    }
}
$SIG{'INT'} = 'DEFAULT'; # восстановить действие по умолчанию
```

Особенность использования сигнала в данном фрагменте программы состоит том, что значение флага проверяется в важнейших точках процесса вычисления и используется для преждевременного выхода из цикла; при этом выполняется и необходимая “очистка”. Обратите внимание на последний оператор в приведенном выше коде: установка действия в значение `DEFAULT` восстанавливает действие конкретного сигнала по умолчанию (следующий сигнал `SIGINT` немедленно прервет выполнение программы). Еще одно полезное специальное значение вроде этого — `IGNORE`, т.е. “игнорировать сигнал” (если действие по умолчанию — не игнорировать сигнал, как у `SIGINT`). Для сигнала можно установить действие `IGNORE`, если не нужно выполнять никакой “очистки” и вы не хотите преждевременно завершать выполнение основной программы.

Один из способов генерирования сигнала `SIGINT` — заставить пользователя нажать на клавиатуре терминала соответствующие прерыванию клавиши (например, `[Ctrl+C]`). Процесс тоже может генерировать сигнал `SIGINT`, используя для этого функцию `kill`. Данная функция получает номер или имя сигнала и посылает соответствующий сигнал в процессы (обозначенные идентификаторами) согласно списку, указанному после сигнала. Следовательно, для передачи сигнала из программы необходимо определить идентификаторы процессов-получателей. (Идентификаторы процессов возвращаются некоторыми функциями, например функцией `fork`, и при открытии программы как дескриптора файла функцией `open`). Предположим, вы хотите послать сигнал 2 (известный также как `SIGINT`) в процессы 234 и 237. Это делается очень просто:

```
kill(2,234,237); # послать SIGINT в 234 и 237
kill ('INT', 234, 237); # то же самое
```

Более подробно вопросы обработки сигналов описаны в главе 6 книги *Programming Perl* и на man-странице *perlipc(1)*.

Упражнения

Ответы к упражнениям см. в приложении А.

1. Напишите программу, которая получает результат команды *date* и вычисляет текущий день недели. Если день недели — рабочий день, выводить *get to work*, в противном случае выводить *go play*.
2. Напишите программу, которая получает все реальные имена пользователей из файла */etc/passwd*, а затем трансформирует результат команды *who*, заменяя регистрационное имя (первая колонка) реальным именем. (Совет: создайте хеш, где ключ — регистрационное имя, а значение — реальное имя.) Попробуйте выполнить эту задачу с использованием команды *who* как в обратных кавычках, так и открытой как канал. Что легче?
3. Модифицируйте предыдущую программу так, чтобы ее результат автоматически поступал на принтер. (Если у вас нет доступа к принтеру, то, вероятно, вы можете послать самому себе сообщение электронной почты.)
4. Предположим, функция *mkdir* перестала работать. Напишите подпрограмму, которая не использует *mkdir*, а вызывает */bin/mkdir* с помощью функции *system*. (Убедитесь в том, что она работает с каталогами, в именах которых есть пробел.)
5. Расширьте программу из предыдущего упражнения так, чтобы в ней устанавливались права доступа (с помощью функции *chmod*).

Другие операции преобразования данных

В этой главе:

- Поиск подстроки
- Извлечение и замена подстроки
- Форматирование данных с помощью функции `sprintf()`
- Сортировка по заданным критериям
- Транслитерация
- Упражнения

Поиск подстроки

Успех поиска подстроки зависит от того, где вы ее потеряли. Если вы потеряли ее в большей строке — вам повезло, потому что в таком случае может помочь операция `index`. Вот как ею можно воспользоваться:

```
$x = index($строка,$подстрока)
```

Perl находит первый экземпляр указанной подстроки в заданной строке и возвращает целочисленный индекс первого символа. Возвращаемый индекс отсчитывается от нуля, т.е. если *подстрока* найдена в начале указанной строки, вы получаете 0. Если она найдена на символ дальше, вы получаете 1 и т.д. Если в указанной строке нет подстроки, вы получаете -1.

Вот несколько примеров:

```
$where = index("hello","e"); # $where получает 1
$person = "barney";
$where = index("fred barney",$person); # $where получает 5
$rockers = ("fred","barney");
$where = index(join(" ",@rockers),$person); # то же самое
```

Отметим, что и строка, в которой производится поиск, и строка, которая ищется, может быть литеральной строкой, скалярной переменной, содержащей строку, и даже выражением, которое имеет строковое значение. Вот еще несколько примеров:

```
$which = index("a very long string", "long");      # $which получает 7
$which = index("a very long string", "lame");      # $which получает -1
```

Если строка содержит искомую подстроку в нескольких местах, то функция `index` возвращает первый относительно начала строки индекс. Чтобы найти другие экземпляры подстроки, можно указать для этой функции третий параметр — минимальное значение позиции, которое она будет возвращать при поиске экземпляра подстроки. Выглядит это так:

```
$x = index($большая_строка, $маленькая_строка, $пропуск)
```

Вот несколько примеров того, как работает этот третий параметр:

```
$where = index("hello world", "l");      # возвращает 2 (первая буква l)
$where = index("hello world", "l", 0);   # то же самое
$where = index("hello world", "l", 1);   # опять то же самое
$where = index("hello world", "l", 3);   # теперь возвращает 3
# (3 — первая позиция, которая больше или равна 3)
$where = index("hello world", "o", 5);   # возвращает 7 (вторая o)
$where = index("hello world", "o", 8);   # возвращает -1 (ни одной после 8)
```

Вы можете пойти другим путем и просматривать строку справа налево с помощью функции `rindex`, чтобы получить правый крайний экземпляр. Эта функция тоже возвращает количество символов между левым концом строки и началом подстроки, как и раньше, но вы получите крайний правый экземпляр, а не первый слева, если их несколько. Функция `rindex` тоже принимает третий параметр, как и функция `index`, чтобы вы могли получить целочисленный индекс первого символа экземпляра подстроки, который меньше или равен адресу выбранной позиции. Вот несколько примеров того, что можно таким образом получить:

```
$w = rindex("hello world", "he");      # $w принимает значение 0
$w = rindex("hello world", "l");      # $w принимает значение 9 (крайняя правая l)
$w = rindex("hello world", "o");      # $w принимает значение 7
$w = rindex("hello world", "o ");     # теперь $w принимает значение 4
$w = rindex("hello world", "xx");     # $w принимает значение -1 (не найдена)
$w = rindex("hello world", "o", 6);   # $w принимает значение 4 (первая до 6)
$w = rindex("hello world", "o", 3);   # $w принимает значение -1 (не найдена до 3)
```

Извлечение и замена подстроки

Извлечь фрагмент строки можно путем осторожного применения регулярных выражений, но если этот фрагмент всегда находится на известной позиции, такой метод неэффективен. В этом случае удобнее использовать функцию `substr`. Эта функция принимает три аргумента: строковое значение, начальную позицию (определяемую так же, как в функции `index`) и длину, т.е.

```
$s = substr($строка, $начало, $длина);
```

Начальная позиция определяется так же, как в функции `index`: первый символ — ноль, второй символ — единица и т.д. Длина — это число символов, которые необходимо извлечь, начиная от данной позиции: нулевая длина означает, что символы не извлекаются, единица означает получение первого символа, двойка — двух символов и т.д. (Больше символов, чем имеется в строке, извлечь нельзя, поэтому если вы запросите слишком много, ничего страшного не произойдет.) Выглядит это так:

```
$hello = "hello, world!";  
$grab = substr($hello, 3, 2);      # $grab получает "lo"  
$grab = substr($hello, 7, 100);    # 7 до конца, или "world!"
```

Можно даже выполнять подобным образом операцию “десять в степени n ” для небольших целочисленных степеней, например:

```
$big = substr("10000000000",0,$power+1);    # 10 ** $power
```

Если количество символов равно нулю, то возвращается пустая строка. Если либо начальная, либо конечная позиция меньше нуля, то такая позиция отсчитывается на соответствующее число символов, начиная с конца строки. Так, начальная позиция -1 и длина 1 (или более) дает последний символ. Аналогичным образом начальная позиция -2 отсчитывается от второго символа относительно конца строки:

```
$stuff = substr("a very long string",-3,3);    # последние три символа  
$stuff = substr("a very long string",-3,1);    # буква л
```

Если начальная позиция указана так, что находится “левее” начала строки (например, задана большим отрицательным числом, превышающим длину строки), то в качестве начальной позиции берется начало строки (как если бы вы указали начальную позицию 0). Если начальная позиция — большое положительное число, то всегда возвращается пустая строка. Другими словами, эта функция всегда возвращает нечто, отличное от сообщения об ошибке.

Отсутствие аргумента “длина” эквивалентно взятию в качестве этого аргумента большого числа — в этом случае извлекается все от выбранной позиции до конца строки*.

Если первый аргумент функции `substr` — скалярная переменная (другими словами, она может стоять в левой части операции присваивания), то сама эта функция может стоять в левой части операции присваивания. Если вы перешли к программированию на Perl из C, вам это может показаться странным, но для тех, кто когда-нибудь имел дело с некоторыми диалектами Basic, это вполне нормально.

* В очень старых версиях Perl пропуск третьего аргумента не допускался, поэтому первые Perl-программисты использовали в качестве этого аргумента большие числа. Вы, возможно, столкнетесь с этим в своих археологических исследованиях программ, написанных Perl.

В результате такого присваивания изменяется та часть строки, которая была бы возвращена, будь `substr` использована не в левой, а в правой части выражения. Например, `substr($var, 3, 2)` возвращает четвертый и пятый символы (начиная с 3 в количестве 2), поэтому присваивание изменяет указанные два символа в `$var` подобно тому, как это приведено ниже:

```
$hw = "hello world";  
substr($hw, 0, 5) = "howdy";    # $hw теперь равна "howdy world"
```

Длина заменяющего текста (который присваивается функции `substr`) не обязательно должна быть равна длине заменяемого текста, как в этом примере. Строка автоматически увеличивается или уменьшается в соответствии с длиной текста. Вот пример, в котором строка укорачивается:

```
substr($hw, 0, 5) = "hi";        # $hw теперь равна "hi world"
```

В следующем примере эта строка удлиняется:

```
substr($hw, -6, 5) = "nationwide news";    # заменяет "world"
```

Процедуры укорачивания и удлинения заменяемой строки выполняются достаточно быстро, поэтому не бойтесь их использовать — хотя лучше все же заменять строку строкой той же длины.

Форматирование данных с помощью функции `sprintf()`

Функция `printf` оказывается удобной, когда нужно взять список значений и создать выходную строку, в которой эти значения отображались бы в заданном виде. Функция `sprintf` использует такие же аргументы, как и функция `printf`, но возвращает то, что выдала бы `printf`, в виде одной строки. (Можете считать ее “строковой функцией `printf`”.) Например, чтобы создать строку, состоящую из буквы `X` и значения переменной `$y`, дополненного нулями до пяти разрядов, нужно записать:

```
$result = sprintf("X%05d", $y);
```

Описание аргументов функции `sprintf` вы найдете в разделе `sprintf` главы 3 книги *Programming Perl* и на `man`-странице `printf(3)` (если она у вас есть).

Сортировка по заданным критериям

Вы уже знаете, что с помощью встроенной функции `sort` можно получить какой-либо список и отсортировать его по возрастанию кодов ASCII. Что, если вы хотите отсортировать список не по возрастанию кодов ASCII, а, скажем, с учетом числовых значений? В Perl есть инструменты, которые позволят вам решить и эту задачу. Вы увидите, что Perl-функция `sort` может выполнять сортировку в любом четко установленном порядке.

Чтобы задать порядок сортировки, следует определить программу сравнения, которая задает способ сравнения двух элементов. Для чего она нужна? Нетрудно понять, что сортировка — это размещение множества элементов в определенном порядке путем их сравнения между собой. Поскольку сравнить сразу все элементы нельзя, нужно сравнивать их по два и, используя результаты этих попарных сравнений, расставить все их по местам.

Программа сравнения определяется как обычная подпрограмма. Она будет вызываться многократно, и каждый раз ей будут передаваться два аргумента сортируемого списка. Данная подпрограмма должна определить, как первое значение соотносится со вторым (меньше, равно или больше), и возвратить закодированное значение (которое мы опишем чуть ниже). Этот процесс повторяется до тех пор, пока не будет рассортирован весь список.

Чтобы повысить скорость выполнения, эти два значения передаются в подпрограмму не в массиве, а как значения глобальных переменных `$a` и `$b`. (Не волнуйтесь: исходные значения `$a` и `$b` надежно защищены.) Эта подпрограмма должна возвратить любое отрицательное число, если `$a` меньше `$b`, нуль, если `$a` равно `$b`, и любое положительное число, если `$a` больше `$b`. Теперь учтите, что “меньше чем” соответствует вашему пониманию этого результата в данном конкретном случае; это может быть сравнение чисел, сравнение по третьему символу строки, наконец, сравнение по значениям какого-то хеша с использованием передаваемых значений как ключей — в общем, это очень гибкий механизм.

Вот пример подпрограммы сортировки в числовом порядке:

```
sub by_number {  
    if ($a < $b) {  
        return -1;  
    } elsif ($a == $b) {  
        return 0;  
    } elsif ($a > $b) {  
        return 1;  
    }  
}
```

Обратите внимание на имя `by_number`. На первый взгляд, в имени этой подпрограммы нет ничего особенного, но скоро вы поймете, почему нам нравятся имена, которые начинаются с префикса `by_`.

Давайте разберем эту подпрограмму. Если значение `$a` меньше (в данном случае в числовом смысле), чем значение `$b`, мы возвращаем значение `-1`.

Если значения численно равны, мы возвращаем ноль, а в противном случае возвращаем 1. Таким образом, в соответствии с нашей спецификацией программы сравнения для сортировки этот код должен работать.

Как использовать данную программу? Давайте попробуем рассортировать такой список:

```
@somelist = (1,2,4,8,16,32,64,128,256);
```

Если использовать с этим списком обычную функцию `sort` без всяких “украшений”, числа будут рассортированы так, как будто это строки, причем сортировка будет выполнена с учетом кодов ASCII, т.е.:

```
@wronglist = sort @somelist;      # @wronglist теперь содержит  
(1,128,16,2,256,32,4,64,8)
```

Конечно, это не совсем числовой порядок. Давайте используем в функции `sort` нашу только что определенную программу сортировки. Имя этой программы ставится сразу после ключевого слова `sort`:

```
@rightlist = sort_by_number @wronglist;  
# @rightlist теперь содержит (1,2,4,8,16,32,64,128,256)
```

Задача решена. Обратите внимание: функцию `sort` можно прочитать вместе с ее спутницей, программой сортировки, на человеческом языке, т.е. “рассортировать по числовым значениям”. Вот почему мы использовали в имени подпрограммы префикс `by_` (“по”).

Такое тройное значение (-1, 0, +1), отражающее результаты сравнения числовых значений, встречается в программах сортировки достаточно часто, поэтому в Perl есть специальная операция, которая позволяет сделать все это за один раз. Эту операцию часто называют “челноком” (или “космическим кораблем”, как следует из дословного перевода английского *spaceship*), потому что ее знак — `<=>`.

Используя “космический корабль”, можно заменить предыдущую подпрограмму сортировки следующим кодом:

```
sub by_number {  
    $a <=> $b;  
}
```

Обратите внимание на знак операции между двумя переменными. Да, он действительно состоит из трех символов. Эта операция возвращает те же значения, что и цепочка `if/elsif` из предыдущего определения этой программы. Теперь все записано очень кратко, но этот вызов можно сократить и дальше, заменив имя подпрограммы сортировки самой подпрограммой, записанной в той же строке:

```
@rightlist = sort { $a <=> $b } @wronglist;
```

Некоторые считают, что такая запись снижает удобочитаемость. Мы с ними не согласны. Некоторые говорят, что благодаря этому в программе исчезает необходимость выполнять переход к определению подпрограммы. Но языку Perl все равно. Наше собственное правило гласит: если код не умещается в одной строке или должен использоваться более чем однажды, он оформляется как подпрограмма.

Для операции сравнения числовых значений “челнок” есть соответствующая строковая операция — `cmp`*. Эта операция возвращает одно из трех значений в зависимости от результата сравнения двух аргументов по строковым значениям. Вот как можно по-другому записать порядок сортировки, который установлен по умолчанию:

```
@result = sort { $a cmp $b } @somelist;
```

Вам, вероятно, никогда не придется писать именно такую подпрограмму (имитирующую встроенную функцию стандартной сортировки) — если только вы не пишете книгу о Perl. Тем не менее, операция `cmp` все же находит применение в каскадных схемах упорядочивания. Например, вам необходимо расставить элементы по численным значениям, если они численно не равны; при равенстве они должны идти быть упорядочены по строковым значениям. (По умолчанию приведенная выше подпрограмма `by_number` просто ставит нечисловые строки в случайном порядке, потому что при сравнении двух нулевых значений числовое упорядочение провести нельзя.) Вот как можно сказать “числовые, если они численно не равны, иначе строковые”:

```
sub by_mostly_numeric {  
    ($a <=> $b) || ($a cmp $b);  
}
```

Этот код работает следующим образом. Если результат работы “челнока” равен -1 или 1, то остальная часть выражения пропускается и возвращается -1 или 1. Если “челнок” дает нуль, то выполняется операция `cmp`, которая возвращает соответствующее значение, сравнивая сортируемые значения как строки.

Сравниваются не обязательно те значения, которые передаются в программу. Пусть, например, у вас есть хеш, ключи которого — регистрационные имена, а значения — реальные имена пользователей. Предположим, вы хотите напечатать таблицу, в которой регистрационные и реальные имена будут рассортированы по порядку реальных имен.

Сделать это довольно легко. Давайте предположим, что значения находятся в массиве `%names`. Регистрационные имена, таким образом, представляют собой список `keys(%names)`. Нам нужно получить список регистрационных имен, рассортированных по соответствующим значениям, поэтому для любого конкретного ключа `$a` мы должны проверить значение `$names{$a}` и провести

* Не вполне соответствующая. Встроенная функция `sort` отбрасывает элементы `undef`, а эта функция — нет.

сортировку соответственно данного значения. Если следовать этой логике, то программа практически напишется сама:

```
@sortedkeys = sort_by_name keys (%names);
sub by_names {
    return $names{$a} cmp $names{$b};
}
foreach (@sortedkeys) {
    print "$_ has a real name of $names{$_}\n";
}
```

К этому нужно еще добавить “аварийное” сравнение. Предположим, что реальные имена двух пользователей совпадают. Из-за капризной природы программы `sort` мы в первый раз можем получить эти значения в одном порядке, а во второй раз — в другом. Это плохо, если данный результат придется, например, вводить в программу сравнения для формирования отчета, поэтому следует избегать таких вещей. Задача легко решается с помощью операции `cmp`:

```
sub by_names {
    ($names{$a} cmp $names{$b}) || ($a cmp $b);
}
```

Если реальные имена совпадают, то сортировка здесь производится на основании регистрационного имени. Поскольку регистрационные имена уникальны (ведь, помимо всего прочего, они являются ключами хеша, а ключи совпадать не могут), мы можем добиться нужного нам результата. Если вы не хотите, чтобы поздно вечером раздался звонок от системного администратора, удивленно спрашивающего, почему включается аварийная сигнализация — пишите хорошие программы днем!

Транслитерация

Если вам необходимо взять строку и заменить все экземпляры какого-нибудь символа другим символом или удалить их, это можно сделать, как вы уже знаете, с помощью тщательно подобранных команд `s///`. Предположим, однако, вам нужно превратить все буквы `a` в буквы `b`, а все буквы `b` — в буквы `a`. Это нельзя сделать посредством двух команд `s///`, потому что вторая команда отменит все изменения, сделанные первой.

Такое преобразование данных очень просто выполняется в `shell` с помощью стандартной команды `tr(1)`:

```
tr ab ba <indata >outdata
```

(Если вы ничего не знаете о команде `tr`, загляните на `man`-страницу `tr(1)`; это полезный инструмент.) В `Perl` тоже применяется операция `tr`, которая работает в основном так же:

```
tr/ab/ba;
```

Операция `tr` принимает два аргумента: *старая_строка* и *новая_строка*. Они используются так же, как аргументы команды `s///`; другими словами, имеется некий разделитель, который стоит сразу же за ключевым словом `tr` и разделяет и завершает аргументы (в данном случае это косая черта, но в этой роли могут выступать почти все символы).

Аргументы операции `tr` похожи на аргументы команды `tr(1)`. Операция `tr` изменяет содержимое переменной `$_` (совсем как `s///`), отыскивая в ней символы старой строки и заменяя найденные символы соответствующими символами новой строки. Вот несколько примеров:

```
$_ = "fred and barney";
tr/fb/bf;          # $_ теперь содержит "bred and farney"
tr/abcde/ABCDE/;   # $_ теперь содержит "BrED AnD fArnEy"
tr/a-z/A-Z/;       # $_ теперь содержит "BRED AND FARNEY"
```

Обратите внимание на то, что диапазон символов можно обозначить двумя символами, разделенными дефисом. Если вам нужен в строке дефис как таковой, поставьте перед ним обратную косую.

Если новая строка короче старой, то последний символ новой строки повторяется столько раз, сколько нужно для того, чтобы строки имели одинаковую длину, например:

```
$_ = "fred and barney";
tr/a-z/x/;         # $_ теперь содержит "xxxx xxx xxxxxx"
```

Чтобы такое не происходило, поставьте в конце операции `tr///` букву `d`, которая означает *delete* ("удалить"). В данном случае последний символ не повторяется. Все символы старой строки, для которых нет соответствующих символов в новой строке, просто удаляются:

```
$_ = "fred and barney";
tr/a-z/ABCDE/d;    # $_ теперь содержит "ED AD BAЕ"
```

Обратите внимание на то, что все буквы, стоящие после буквы `e`, исчезают, потому что в новом списке соответствующей буквы нет, и на то, что на пробелы это не влияет, потому что их нет в старом списке. По принципу работы это эквивалентно команде `tr` с опцией `-d`.

Если новый список пуст и опция `d` не используется, то новый список будет совпадать со старым. Это может показаться глупым — зачем заменять `I` на `I` и `2` на `2`? — но на самом деле в этом есть довольно глубокий смысл. Операция `tr///` возвращает количество символов, совпавших со старой строкой, и путем замены символов на самих себя вы можете получить число таких символов, содержащихся в новой строке*. Например:

* Это справедливо только для одиночных символов. Для подсчета строк в операции сопоставления с образцом используйте флаг `/g`:

```
while (/образец/g) {
    $count++;
}
```

```
$_ = "fred and barney";
$count = tr/a-z//;      # $_ не изменилась, но $count = 13
$count2 = tr/a-z/A-Z//; # $_ переведена в верхний регистр, и $count2 = 13
```

Если в конце операции добавить букву `c` (как мы добавляли букву `d`), символы старой строки будут рассматриваться как исключение из набора всех 256 символов. Каждый символ, указанный в старой строке, удаляется из совокупности всех возможных символов; оставшиеся символы, взятые по порядку от младшего к старшему, образуют новую строку-результат. Например, подсчитать или изменить в нашей строке все символы, не являющиеся буквами, можно так:

```
$_ = "fred and barney";
$count = tr/a-z//c;      # $_ не изменилась, но $count = 2
tr/a-z/_/c;             # $_ теперь содержит "fred_and_barney" (символы-небуквы => _)
tr/a-z//cd;             # $_ теперь содержит "fredandbarney" (символы-небуквы
удалены)
```

Отметим, что эти операции можно объединять, как показано в последнем примере, где мы сначала меняем совокупность символов на “дополняющую” (список букв становится списком всех символов-небукв), а затем с помощью опции `d` удаляем все символы этой совокупности.

Последняя опция операции `tr///` — `s`, которая заменяет множество экземпляров одной преобразованной буквы одним. Например:

```
$_ = "aaabbbcccdefghi";
tr/defghi/abcd/s;      # $_ теперь содержит "aaabbbcccabcd"
```

Обратите внимание: буквы `def` заменены на `abc`, а `ghi` (которые без опции `s` превратились бы в `ddd`) стали одной буквой `d`. Отметим также, что стоящие друг за другом буквы в первой части строки “не сжимаются”, потому что для них не задано преобразование. Вот еще несколько примеров:

```
$_ = "fred and barney, wilma and betty";
tr/a-z/X/s;      # $_ теперь содержит "X X X, X X X"
$_ = "fred and barney, wilma and betty";
tr/a-z/_/cs;     # $_ теперь содержит "fred_and_barney_wilma_and_betty"
```

В первом из этих примеров каждое слово (стоящие друг за другом буквы) было заменено одной буквой `x`. Во втором примере все группы стоящих друг за другом символов-небукв стали одиночными знаками подчеркивания.

Как и команда `s///`, операция `tr` может быть проведена над другой строкой, а не только над строкой, хранящейся в переменной `$_`. Это достигается с помощью операции `=~`:

```
$names = "fred and barney";
$names =~ tr/aeiou/X/;      # $names теперь содержит "frXd Xnd bXrnXy"
```

Упражнения

Ответы к упражнениям см. в приложении А.

1. Напишите программу, которая читает список имен файлов и разбивает каждое имя на начальный и конечный компоненты. (Все, что стоит в имени файла до последней косой черты — начальный компонент, а все, что за ней — конечный компонент. Если косой нет, то все имя является конечным компонентом.) Попробуйте выполнить эту программу с именами вроде */fred, barney, fred/barney*. Имеют ли результаты смысл?
2. Напишите программу, которая читает список чисел, стоящих в отдельных строках, и сортирует их по числовым значениям, выводя список-результат в столбец с выравниванием справа. (Совет: для вывода столбца с выравниванием справа нужно использовать формат наподобие `%20g`.)
3. Напишите программу вывода реальных и регистрационных имен пользователей из файла */etc/passwd* с сортировкой по фамилиям пользователей. Работоспособно ли ваше решение в случае, если у двух пользователей одинаковые фамилии?
4. Создайте файл, состоящий из предложений, каждое из которых стоит в отдельной строке. Напишите программу, которая переводит первый символ каждого предложения в верхний регистр, а остальную часть предложения — в нижний. (Работает ли эта программа в случае, если первый символ — небуква? Как решить эту задачу, если предложения не стоят в отдельных строках?)

В этой главе:

- *Получение информации о паролях и группах*
- *Упаковка и распаковка двоичных данных*
- *Получение информации о сети*
- *Упражнение*

Доступ к системным базам данных

Получение информации о паролях и группах

Информация о вашем пользовательском имени и идентификаторе, которая имеется в системе UNIX, практически открыта. По сути дела, любая программа, которая не сочтет за труд заглянуть в файл `/etc/passwd`, поможет вам увидеть почти все, кроме незашифрованного пароля. Этот файл имеет особый формат, определяемый в `passwd(5)`, и выглядит приблизительно так:

```
name:passwd:uid:gid:gcoss:dir:shell
```

Поля определены следующим образом:

`name`

Регистрационное имя пользователя

`passwd`

Зашифрованный пароль или что-нибудь простое, если используется теневой файл паролей

`uid`

Идентификатор пользователя (для пользователя `root` — 0, для обычных пользователей — ненулевое число)

`gid`

Регистрационная группа по умолчанию (группа 0 может быть привилегированной, но не обязательно)

`gcos`

Как правило, содержит полное имя пользователя, за которым через запятую следует и другая информация

`dir`

Начальный каталог (каталог, в который вы переходите, когда даете команду `cd` без аргументов, и в котором хранится большинство ваших файлов, имена которых начинаются с точки)

`shell`

Ваш регистрационный shell, как правило, `/bin/sh` или `/bin/csh` (а, может быть, даже `/usr/bin/perl`, если вы большой оригинал)

Типичные элементы файла паролей выглядят так:

```
fred*:123:15:Fred Flintstone,,,:/home/fred:/bin/csh
barney*:125:15:Barney Rubble,,,:/home/barney:/bin/csh
```

Сейчас в Perl достаточно инструментов для того, чтобы можно было легко выполнить разбор такой строки (например, с помощью функции `split`), не прибегая к специальным программам. Тем не менее в библиотеке UNIX все же есть набор специальных программ: `getpwent(3)`, `getpwuid(3)`, `getpwnam(3)` и т.д. Эти программы доступны в Perl под теми же именами, с похожими аргументами и возвращаемыми значениями.

Например, программа `getpwnam` в Perl становится функцией `getpwnam`. Ее единственный аргумент — пользовательское имя (например, `fred` или `barney`), а возвращаемое значение — строка файла `/etc/passwd`, преобразованная в массив со следующими значениями:

```
($name, $passwd, $uid, $gid, $quota, $comment, $gcos, $dir, $shell)
```

Обратите внимание: здесь несколько больше значений, чем в файле паролей. Обычно в UNIX-системах, по крайней мере в тех, которые мы видели, поле `$quota` всегда пусто, а поля `$comment` и `$gcos` часто оба содержат персональную информацию о пользователе (поле GCOS). Так, для старины Фреда мы получаем

```
("fred", "*", 123, 15, "", "Fred Flintstone,,,", "Fred Flintstone,,,",
"/home/gred", " /bin/csh")
```

посредством любого из следующих вызовов:

```
getpwuid(123)
getpwnam("fred")
```

Отметим, что в качестве аргумента функция `getpwuid` принимает идентификатор пользователя, а `getpwnam` — регистрационное имя.

При вызове в скалярном контексте функции `getpwnam` и `getpwuid` также имеют возвращаемое значение — данные, которые вы запросили с их помощью. Например:

```
$idnum = getpwuid("daemon");
$logn = getpwnam(25);
```

Возможно, вам захочется получить эти результаты по отдельности, используя некоторые из уже знакомых вам операций, проводимых над списками. Один способ — получить часть списка, используя для этого срез списка, например получить для Фреда только начальный каталог:

```
($fred_home) = (getpwnam ("fred"))[7];      # начальный каталог Фреда
```

Как просмотреть весь файл паролей? Для этого можно было бы поступить, к примеру, так:

```
for($id = 0; $id <= 10_000; $id++) {
    @stuff = getpwuid $id;
} ### не рекомендуется'
```

Это, однако, неверный путь. Наличие нескольких способов само по себе еще не означает, что все они в равной степени эффективны.

Функции `getpwuid` и `getpwnam` можно считать функциями произвольного доступа; они извлекают конкретный элемент по его ключу, поэтому для начала у вас должен быть ключ. Другой метод доступа к файлу паролей — последовательный, т.е. поочередное получение его записей.

Программами последовательного доступа к файлу паролей являются функции `setpwent`, `getpwent` и `endpwent`. В совокупности эти три функции выполняют последовательный проход по всем записям файла паролей. Функция `setpwent` инициализирует просмотр. После инициализации каждый вызов `getpwent` возвращает следующую запись файла паролей. Если данных для обработки больше нет, `getpwent` возвращает пустой список. Наконец, вызов `endpwent` освобождает ресурсы, используемые программой просмотра; это делается автоматически и при выходе из программы.

Приведенное описание может оказаться не совсем понятным без примера, поэтому дадим его:

```
setpwent();                                # инициализировать просмотр
while (@list = getpwent()) {               # выбрать следующий элемент
    ($login,$home) = @list[0,7];           # получить регистрационное имя
                                           # и начальный каталог
    print "Home directory for $login is $home\n";    # сообщить это
}
endpwent();                                # все сделано
```

Эта программа сообщает имена начальных каталогов всех пользователей, перечисленные в файле паролей. А если вы хотите расставить начальные каталоги в алфавитном порядке? В предыдущей главе мы изучили функцию `sort`, давайте воспользуемся ею:

```
setpwent();                                # инициализировать просмотр
while (@list = getpwent()) {               # выбрать следующий элемент
    ($login,$home) = @list[0,7];           # получить регистрационное имя
                                           # и начальный каталог
    $home{$login} = $home;                 # сохранить их
}
endpwent();                                # все сделано
@keys = sort { $home{$a} cmp $home{$b} } keys %home;
foreach $login (@keys) {                   # пройти по рассортированным именам
    print "home of $login is $home{$login}\n";
}
```

Этот несколько более длинный фрагмент иллюстрирует важную особенность последовательного просмотра файла паролей: вы можете сохранять соответствующие фрагменты данных в структурах данных, выбираемых по своему усмотрению. Первая часть примера — это код просмотра всего файла паролей с созданием хеша, в котором ключ — регистрационное имя, а значение — начальный каталог, соответствующий этому регистрационному имени. Строка `sort` получает ключи хеша и сортирует их в соответствии со строковым значением. Завершающий цикл — это проход по рассортированным ключам и поочередный вывод всех значений.

В общем случае для просмотра небольшого количества значений рекомендуется использовать программы произвольного доступа (`getpwuid` и `getpwnam`). Если значений много или необходим просмотр всех значений, проще выполнить проход с последовательным доступом (с помощью функций `setpwent`, `getpwent` и `endpwent`) и поместить конкретные значения, которые вы будете искать, в хеш*.

Доступ к файлу */etc/group* осуществляется аналогичным образом. Последовательный доступ обеспечивается вызовами функций `setgrent`, `getgrent` и `endgrent`. Вызов `getgrent` возвращает значения в следующем формате:

```
($name, $passwd, $gid, $members)
```

Эти четыре значения примерно соответствуют четырем полям файла */etc/group*, поэтому за подробной информацией обращайтесь к описаниям, приведенным на map-страницах, относящихся к формату этого файла. Соответствующие функции произвольного доступа — `getgrgid` (по идентификатору группы) и `getgrnam` (по имени группы).

* Если у вас узел с большой NIS-картой, то по соображениям производительности такой способ предобработки файла паролей лучше не использовать

Упаковка и распаковка двоичных данных

Данные о паролях и группах удобно использовать в текстовом виде. Информацию в других системных базах данных более естественно представлять иначе. Например, IP-адрес интерфейса обрабатывается внутренними механизмами как четырехбайтовое число. Хотя его часто представляют в текстовом виде (как четыре небольших целых числа, разделенных точками), такое преобразование — пустая трата времени, если эти данные в промежутке между преобразованиями не выводятся на экран пользователя.

По этой причине написанные на Perl сетевые программы, ожидающие или возвращающие IP-адрес, используют четырехбайтовую строку, одному символу которой соответствует один байт в памяти. Хотя конструирование и интерпретация такой байтовой строки — довольно простая задача, решаемая с помощью функций `chr` и `ord` (здесь не представленных), в Perl используется более эффективное решение, которое в равной степени применимо и к более сложным структурам.

Функция `pack` по принципу работы немного похожа на функцию `sprintf`. Она получает строку, задающую формат, и список значений и упаковывает значения в одну строку. Однако в `pack` строка, задающая формат, предназначена для создания двоичной структуры данных. Например, вот как можно взять четыре небольших целых числа и упаковать их в виде последовательности байтов без знака в строке:

```
$buf = pack("CCCC", 140, 186, 65, 25);
```

Здесь строка формата `pack` — четыре буквы `C`. Каждая `C` соответствует отдельному значению, взятому из приведенного следом списка (подобно тому, что делает спецификация `%` в функции `sprintf`). Формат `C` (согласно ман-страницам Perl, краткому справочнику, книге *Programming Perl*, HTML-файлам и даже видеоролику Perl: The Motion Picture) обозначает один байт, вычисляемый из символьного значения без знака (короткого целого). Строка-результат в переменной `$buf` представляет собой четырехсимвольную строку, в которой каждый символ задан одним байтом. Эти байты имеют значения 140, 186, 65 и 25 соответственно.

Аналогичным образом формат `l` генерирует длинное значение со знаком. На многих машинах это четырехбайтовое число, хотя этот формат зависит от конкретной машины. На четырехбайтовой “длинной” машине оператор

```
$buf = pack("l", 0x41424344);
```

генерирует четырехсимвольную строку, состоящую из символов `ABCD` или `DCBA` — в зависимости от того, какой порядок хранения байтов используется на данной машине: “младший в младшем” или “старший в младшем” (либо что-то совершенно иное, если эта машина “не говорит” на ASCII). Это объясняется тем, что мы упаковываем одно значение в четыре символа (для представления длинного целого отводится четыре байта), а это одно значение

как раз состоит из байтов, представляющих коды ASCII первых четырех букв алфавита. Аналогичным образом,

```
$buf = pack("ll", 0x41424344, 0x45464748);
```

создает восьмибайтовую строку, состоящую из букв ABCDEFGH или DCBANGFE, опять-таки в зависимости от того, какой порядок хранения байтов используется в данной машине — “младший в младшем” или “старший в младшем”.

Полный перечень различных форматов, используемых для упаковки, приведен в справочной документации (*perlfunc(1)* или *Programming Perl*). Мы приведем некоторые из них как примеры, но все, конечно, давать не будем.

Допустим, вам дали восьмибайтовую строку ABCDEFGH и сказали, что она является представлением хранящихся в памяти (один символ — один байт) двух длинных (четыребайтовых) значений со знаком. Как ее интерпретировать? Нужно воспользоваться функцией, обратной функции `pack`, — функцией `unpack`. Она берет строку управления форматом (как правило, идентичную той, которую вы указывали в функции `pack`) и строку данных и возвращает список значений, которые хранятся в соответствующих ячейках памяти. Давайте, например, распакуем такую строку:

```
($val1,$val2) = unpack("ll","ABCDEFGH");
```

Это даст нам в переменной `$val1` нечто вроде 0x41424344, а может быть, и 0x44434241 (в зависимости от порядка хранения байтов). По сути дела, по возвращаемым значениям мы можем определить, на какой машине работаем — с порядком “младший в младшем” или “старший в младшем”.

Пробельные символы в строке, задающей формат, игнорируются и используются лишь для удобочитаемости. Число в этой строке, как правило, задает повторение предыдущей спецификации соответствующее количество раз. Например, `CCCC` можно записать как `C4` или `C2C2`, смысл от этого не изменится. (Однако в некоторых спецификациях число, указанное после символа, задающего формат, является частью спецификации, поэтому их подобным образом записывать нельзя.)

После символа формата может стоять также звездочка, которая задает повторное применение данного формата до тех пор, пока не обработана остальная часть списка значений или пока не создана остальная часть строки, содержащей двоичное представление (в зависимости от того, что выполняется — упаковка или распаковка). Вот еще один способ упаковки четырех символов без знака в одну строку:

```
$buf = pack("C*", 140, 186, 65, 25);
```

Здесь указанные четыре значения полностью обрабатываются одной спецификацией формата. Если бы вам требовались два коротких целых и “максимально возможное количество символов без знака”, то можно было бы написать примерно так:

```
$buf = pack("s2 C*", 3141, 5926, 5, 3, 5, 8, 9, 7, 9, 3, 2);
```

Здесь мы получаем первые два значения как короткие (и генерируем, вероятно, четыре или восемь символов), а остальные девять — как символы без знака (и генерируем, почти наверняка, девять символов).

Функция `unpack` со звездочкой в качестве спецификации может формировать список элементов, длина которых заранее не определена. Например, при распаковке с использованием формата `C*` создается один элемент списка (число) для каждого символа строки. Так, оператор

```
@values = unpack("C*", "hello, world!\n");
```

позволяет сформировать список из 14 элементов, по одному для каждого символа строки.

Получение информации о сети

Perl поддерживает сетевое программирование средствами, которые хорошо знакомы тем, кто писал программы для сетевых приложений на C. По сути дела, большинство функций Perl, обеспечивающих доступ к сети, имеют и те же имена, что их C-коллеги, и похожие параметры. В этой главе мы не можем привести полную информацию по сетевому программированию, поэтому просто рассмотрим фрагмент сетевого приложения.

Один из параметров, который вам приходится часто определять,— это IP-адрес, соответствующий сетевому имени (или наоборот). В C вы преобразуете сетевое имя в сетевой адрес с помощью программы `gethostbyname(3)`. Затем, используя полученный адрес, вы устанавливаете связь между своей программой и другой программой, которая работает где-то в другом месте.

В Perl функция, преобразующая хост-имя в адрес, имеет то же имя, что и C-программа, и похожие параметры. Выглядит она так:

```
($name, $aliases, $addrtype, $length, @addrs) = gethostbyname($name);  
# основная форма функции gethostbyname
```

Параметр этой функции — имя хоста, например, `slate.bedrock.com`, а возвращаемое значение — список из четырех и более параметров (в зависимости от того, сколько адресов связано с данным именем). Если имя хоста недействительно, функция возвращает пустой список.

Когда `gethostbyname` вызывается в скалярном контексте, возвращается только первый адрес.

Если `gethostbyname` завершается успешно, то переменной `$name` в качестве значения присваивается каноническое имя, которое, если входное имя — псевдоним, отличается от входного имени. Значение переменной `$aliases` — это список разделенных пробелами имен, под которыми данный хост известен в сети. Переменная `$addrtype` содержит кодовое обозначение формата представления адреса. Для имени `slate.bedrock.com` мы можем предположить, что это значение указывает на IP-адрес, обычно представляемый как четыре числа из диапазона от 1 до 256, разделенных точками.

Переменная `$length` содержит количество адресов. Это лишняя информация, так как в любом случае можно посмотреть на размер массива `@addrs`.

Наиболее полезная часть возвращаемого значения — массив `@addrs`. Каждый элемент данного массива — это отдельный IP-адрес, представленный во внутреннем формате и обрабатываемый в Perl как четырехсимвольная строка*. Эта четырехсимвольная строка представлена в форме, понятной для других сетевых Perl-функций. Однако предположим, что нам требуется вывести результат в виде, удобном для пользователя. В данном случае нам нужно с помощью функции `unpack` и еще нескольких операций преобразовать возвращаемое значение в удобочитаемый формат. Вот код, который обеспечивает вывод одного из IP-адресов хоста `slate.bedrock.com`:

```
($addr) = (gethostbyname("slate.bedrock.com"))[4];
print "Slate's address is ",
    join(".",unpack("C4", $addr)), "\n";
```

Функция `unpack` получает четырехбайтовую строку и возвращает четыре числа. Оказывается, они стоят именно в том порядке, который нужен функции `join` для того, чтобы она вставила между каждой парой чисел точку и представила таким образом все это в удобочитаемой форме. Информация о простых программах-клиентах приведена в приложении В.

Упражнение

Ответ см. в приложении А.

1. Напишите программу, которая создает таблицу соответствия идентификаторов пользователей и реальных имен из записей файла паролей, а затем с помощью этой таблицы выводит список реальных имен, принадлежащих каждой группе, упомянутой в файле групп. (Включает ли ваш список тех пользователей, у которых в записи файла паролей стоит группа по умолчанию, но в записи файла групп явного упоминания этой группы нет? Если не включает, как это сделать?)

* По крайней мере до IPv6.

В этой главе:

- DBM-базы данных и DBM-хеши
- Открытие и закрытие DBM-хешей
- Использование DBM-хеши
- Базы данных произвольного доступа с записями фиксированной длины
- Базы данных с записями переменной длины (текстовые)
- Упражнения

Работа с пользовательскими базами данных

DBM-базы данных и DBM-хеши

В большинстве UNIX-систем есть стандартная библиотека, которая называется DBM. Эта библиотека представляет собой простую систему управления базами данных, которая позволяет программам записывать набор пар ключ-значение в пару файлов. В этих файлах хранятся значения базы данных в промежутках между вызовами программ, использующих ее, и эти программы могут вводить в базы данных новые значения, обновлять существующие и удалять старые.

Библиотека DBM довольно проста, но, учитывая ее доступность, некоторые системные программы активно используют эту библиотеку для своих довольно скромных нужд. Например, *sendmail* (а также ее варианты и производные) хранит базу данных *aliases* (соответствие адресов электронной почты и имен получателей) как DBM-базу данных. Самое популярное ПО телеконференций *Usenet* использует DBM-базу данных для хранения информации о текущих и недавно просмотренных статьях. Главные файлы базы данных Sun NIS (урожденной YP) также хранятся в формате DBM.

Perl обеспечивает доступ к такому же механизму DBM довольно умным способом: посредством процесса, похожего на открытие файла, с DBM-базой данных можно связать хеш. Этот хеш (называемый DBM-массивом) используется для доступа к DBM-базе данных и внесения в нее изменений.

Создание нового элемента в этом массиве влечет за собой немедленное изменение в базе данных. Удаление элемента приводит к удалению значения из DBM-базы данных и т.д.*

Размер, количество и вид ключей и значений в DBM-базе данных ограничены. В зависимости от того, какой версией библиотеки DBM вы пользуетесь, эти же ограничения могут иметь место и для DBM-массива. Подробности см. на map-странице *AnyDBM_File*. В общем, если вы сумеете сделать так, чтобы и ключи, и значения упаковывались не больше чем в 1000 символов с произвольными двоичными значениями, то все будет нормально.

Открытие и закрытие DBM-хешей

Чтобы связать DBM-базу данных с DBM-массивом, применяется функция `dbmopen`, которая используется следующим образом:

```
dbmopen(%ИМЯ_МАССИВА, "имя_DBM-файла", $режим)
```

Параметр `%ИМЯ_МАССИВА` — это имя Perl-хеша. (Если в данном хеше уже есть значения, они выбрасываются.) Хеш соединяется с DBM-базой данных, заданной параметром `имя_DBM-файла`. Она обычно хранится на диске в виде пары файлов с именами `имя_DBM-файла.dir` и `имя_DBM-файла.pag`.

Параметр `$режим` — это число, которое соответствует битам прав доступа к названным двум файлам, если файлы создаются заново. Обычно оно указывается в восьмеричном формате; часто используемое значение 0644 предоставляет право доступа только для чтения всем, кроме владельца, который имеет право на чтение и запись. Если эти файлы существуют, данный параметр не действует. Например:

```
dbmopen(%FRED, "mydatabase", 0644);    # открыть %FRED на mydatabase
```

Этот вызов связывает хеш `%FRED` с файлами `mydatabase.dir` и `mydatabase.pag`, расположенными в текущем каталоге. Если эти файлы не существуют, они создаются с правами доступа 0644, которые модифицируются с учетом текущего значения, установленного командой `umask`.

Функция `dbmopen` возвращает значение “истина”, если базу данных можно открыть или создать; в противном случае возвращается “ложь” — точно так же, как при вызове функции `open`. Если вы не хотите создавать файлы, используйте вместо параметра `$режим` значение `undef`. Например:

```
dbmopen(%A, "/etc/xx", undef) || die "cannot open DBM /etc/xx";
```

* Это, по сути дела, просто особый случай использования общего механизма `tie`. Если вам понадобится что-нибудь более гибкое, обратитесь к map-страницам *AnyDBM_File(3)*, *DB_File(3)* и *perltie(1)*.

Если, как в данном случае, файлы `/etc/xx.dir` и `/etc/xx.pag` открыть нельзя, то вызов `dbmopen` возвращает значение “ложь” без попытки создать эти файлы.

DBM-массив остается открытым в течение выполнения всей программы. Когда программа завершается, разрывается и связь с DBM-базой данных. Эту связь можно разорвать и способом, близким к закрытию дескриптора файла — с помощью функции `dbmclose`:

```
dbmclose(%A);
```

Как и функция `close`, `dbmclose` возвращает значение “ложь”, если что-нибудь происходит не так, как надо.

Использование DBM-хеша

После открытия базы данных обращения к DBM-хешу преобразуются в обращения к базе данных. Изменение значения в хеше или ввод в него нового значения вызывает немедленную запись соответствующих элементов в файлы на диске. Например, после открытия массива `%FRED` из предыдущего примера мы можем обращаться к элементам базы данных, вводить в нее новые элементы и удалять существующие:

```
$FRED{"fred"} = "bedrock";      # создать (или обновить) элемент
delete $FRED{"barney"};         # удалить элемент базы данных
foreach $key (keys %FRED) {     # пройти по всем значениям
    print "$key has value of $FRED{$key}\n";
}
```

Последний цикл должен просмотреть весь файл на диске дважды: один раз для выборки ключей, а второй — для поиска значений, соответствующих этим ключам. Если вы просматриваете DBM-хеш, то более эффективным способом с точки зрения эксплуатации диска является использование операции `each`, которая делает всего один проход:

```
while (($key, $value) = each(%FRED)) {
    print "$key has value of $value\n";
}
```

Если вы обращаетесь к системным DBM-базам данных, например к базам данных, созданным системами *sendmail* и *NIS*, вы должны иметь в виду, что в плохо написанных C-программах в конце строк иногда стоит символ NUL (`\0`). Программам библиотеки DBM этот NUL не нужен (они обрабатывают двоичные данные с помощью счетчика байтов, а не строки с символом NUL на конце), поэтому он хранится как часть данных. В таком случае вы должны добавлять символ NUL в конец своих ключей и отбрасывать NUL, стоящий в конце возвращаемых значений, иначе данные не будут иметь смысла.

Например, чтобы найти имя `merlyn` в базе данных псевдонимов, можно сделать так:

```
dbmopen(%ALI, "/etc/aliases", undef) || die "no aliases?";
$value = $ALI{"merlyn\0"};      # обратите внимание на добавленный NUL
chop($value);                  # удалить добавленный NUL
print "Randal's mail is headed for: $value\n";      # показать результат
```

В вашей версии UNIX база данных псевдонимов может храниться не в каталоге `/etc`, а в каталоге `/usr/lib`. Чтобы выяснить, где именно она хранится, придется провести маленькое расследование. Новые версии *sendmail* этим NUL-дефектом не страдают.

Базы данных произвольного доступа с записями фиксированной длины

Еще одна форма хранения данных — файл на диске, предназначенный для записей фиксированной длины. В этой схеме данные состоят из ряда записей одинаковой длины. Нумерация этих записей либо не имеет значения, либо определяется по какой-нибудь схеме индексации.

Например, у нас может быть ряд записей со следующими данными: 40 символов — имя, один символ — инициал, 40 символов — фамилия и двухбайтовое целое — возраст. Таким образом, длина каждой записи составляет 83 байта. Если бы мы читали все эти данные в базе данных, то делали бы это порциями по 83 байта до тех пор, пока не добрались до конца. Если бы мы хотели перейти к пятой записи, то мы пропустили бы четыре раза по 83 байта (332 байта) и прочитали бы непосредственно пятую запись.

Perl поддерживает программы, которые используют файл с подобными записями. Помимо того, что вы уже знаете, понадобятся еще несколько операций:

1. Открытие файла на диске для чтения и записи.
2. Переход в этом файле на произвольную позицию.
3. Выборка данных фиксированной длины, а не до следующего символа новой строки.
4. Запись данных блоками фиксированной длины.

В функции `open` перед спецификацией, задающей способ открытия файла (для чтения или записи), необходимо записать знак плюс, указав таким образом, что данный файл в действительности открывается и для чтения, и для записи. Например:

```
open(A, "+<b");      # открыть файл b для чтения-записи (ошибка, если файл отсутствует)
open(C, "+>d");      # создать файл d с доступом для чтения-записи
open(E, "+>>f");     # открыть или создать файл f с доступом для чтения-записи
```

Отметим, что все, что мы сделали — это добавили знак плюс к спецификации, задающей направление ввода-вывода данных в файл.

Открыв файл, мы должны перейти на определенную позицию в нем. Это делается с помощью функции `seek`, которая принимает те же три параметра, что и библиотечная программа `fseek(3)`. Первый параметр — это дескриптор файла, а второй параметр задает смещение, которое интерпретируется в совокупности с третьим параметром. Как правило, в качестве третьего параметра ставится нуль, чтобы второй параметр задавал абсолютную позицию для следующего чтения из файла или записи в файл. Например, чтобы перейти к пятой записи в дескрипторе файла `NAMES` (как описано выше), можно сделать так:

```
seek(NAMES, 4*83, 0);
```

После перемещения указателя в файле на нужную позицию следующая операция ввода или вывода будет начинаться с этой позиции. Для вывода используйте функцию `print`, но не забудьте, что записываемые данные должны иметь строго определенную длину. Чтобы сформировать запись правильной длины, можно воспользоваться функцией `pack`:

```
print NAMES pack("A40 A A40 s", $first, $middle, $last, $age);
```

В данном случае `pack` задает 40 символов для `$first`, один символ — для `$middle`, еще 40 символов — для `$last` и короткое целое (два байта) для `$age`. Определенная таким образом запись будет иметь в длину 83 байта и начинаться с текущей позиции в файле.

Наконец, нам нужно узнать, как выбрать конкретную запись. Конструкция `<NAMES>` возвращает все данные, начиная с текущей позиции и до следующего символа новой строки, однако в нашем случае предполагается, что данные занимают 83 байта и, вероятно, символ новой строки непосредственно в записи отсутствует. Поэтому вместо нее мы используем функцию `read`, которая по внешнему виду и принципу работы очень похожа на свою UNIX-коллегу:

```
$count = read(NAMES, $buf, 83);
```

Первый параметр функции `read` — дескриптор файла. Второй параметр — это скалярная переменная, в которую будут записаны прочитанные данные. Третий параметр задает количество байтов, которые нужно прочитать. Возвращает функция `read` количество фактически прочитанных байтов; как правило, оно равно затребованному количеству байтов, если только дескриптор файла открыт и если вы не находитесь слишком близко к концу файла.

Получив эти 83-символьные данные, разбейте их на компоненты с помощью функции `unpack`:

```
($first, $middle, $last, $age) = unpack("A40 A A40 s", $buf);
```

Как видно, строки, определяющие формат, в функциях `pack` и `unpack` — одинаковы. В большинстве программ эту строку заносят в переменную, указываемую в начале программы, и даже вычисляют с помощью функции `pack` длину записей, а не используют везде константу 83:

```
$names = "A40 A A40 s";  
$names_length = length(pack($names));    # вероятно, 83
```

Базы данных с записями переменной длины (текстовые)

Многие системные базы данных ОС UNIX (и довольно большое число пользовательских баз данных) представляют собой наборы понятных человеку текстовых строк, каждая из которых образует одну запись. Например, каждая строка файла паролей соответствует одному пользователю системы, а строка файла хостов — одному хост-имени.

Корректируются эти базы данных в основном с помощью простых текстовых редакторов. Процедура обновления базы данных состоит из чтения ее в какую-то временную область (память или другой дисковый файл), внесения необходимых изменений и либо записи результата обратно в исходный файл, либо создания нового файла с тем же именем, с одновременным удалением или переименованием старой версии. Этот процесс можно рассматривать как разновидность копирования: данные копируются из исходной базы данных в новую ее версию с внесением изменений в процессе копирования.

`Perl` поддерживает редактирование такого типа в строчно-ориентированных базах данных методом *редактирования на месте*. Редактирование на месте — это модификация способа, посредством которого операция “ромб” (<>) считывает данные из списка файлов, указанного в командной строке. Чаще всего этот режим редактирования включается путем установки аргумента командной строки `-i`, но его можно запустить и прямо из программы, как показано в приведенных ниже примерах.

Чтобы запустить режим редактирования на месте, присвойте значение скалярной переменной `$^I`. Оно играет важную роль и будет сейчас рассмотрено.

Когда используется конструкция `<>` и переменная `$^I` имеет значение, отличное от `undef`, к списку неявных действий, которые выполняет операция “ромб”, добавляются шаги, отмеченные в приведенном ниже коде комментарием `## INPLACE ##`:

```
$ARGV = shift @ARGV;  
open (ARGV, "<$ARGV");  
rename ($ARGV, "$ARGV$^I");    ## INPLACE ##  
unlink ($ARGV);                ## INPLACE ##
```

```
open (ARGVOUT, ">$ARGV");      ## INPLACE ##
select (ARGVOUT);              ## INPLACE ##
```

В результате в операции “ромб” при чтении используется старый файл, а запись в дескриптор файла по умолчанию осуществляется в новую копию этого файла. Старый файл остается в резервной копии, суффикс имени файла которой равен значению переменной `$^I`. (При этом биты прав доступа копируются из старого файла в новый.) Эти шаги повторяются каждый раз, когда новый файл берется из массива `@ARGV`.

Типичные значения переменной `$^I` — `.bak` или `~`, т.е. резервные файлы создаются почти так же, как это делается в текстовом редакторе. Странное и полезное значение `$^I` — пустая строка (“”), благодаря которой старый файл после редактирования аккуратно удаляется. К сожалению, если система при выполнении вашей программы откажет, то вы потеряете все свои старые данные, поэтому значение “” рекомендуется использовать только храбрецам, дуракам и излишне доверчивым.

Вот как можно путем редактирования файла паролей заменить регистрационный shell всех пользователей на `/bin/sh`:

```
@ARGV = ("/etc/passwd"); # снабдить информацией операцию "ромб"
$^I = ".bak";             # для надежности записать /etc/passwd.bak
while (<>) {               # основной цикл, по разу для каждой строки файла
    # /etc/passwd
    s#:[^:]*$#:/bin/sh#; # заменить shell на /bin/sh
    print;               # послать выходную информацию в ARGVOUT: новый
    # /etc/passwd
}
```

Как видите, эта программа довольно проста. Однако ее можно заменить всего лишь одной командой с несколькими аргументами командной строки, например:

```
perl -p -i.bak -e 's#:[^:]*$#:/bin/sh#' /etc/passwd
```

Ключ `-p` охватывает вашу программу циклом `while`, который включает оператор `print`. Ключ `-i` устанавливает значение переменной `$^I`. Ключ `-e` определяет следующий аргумент как фрагмент Perl-кода для тела цикла, а последний аргумент задает начальное значение массива `@ARGV`.

Более подробно аргументы командной строки рассматриваются в книге *Programming Perl* и на map-странице *perlrun*.

Упражнения

Ответы см. в приложении А.

1. Создайте программу, которая открывает базу данных псевдонимов *send-mail* и выводит на экран все ее элементы.
2. Создайте две программы: одну для чтения данных с помощью операции `<>`, разбивки их на слова и обновления DBM-файла с запоминанием числа экземпляров каждого слова, а вторую для открытия этого DBM-файла и отображения результатов, рассортированных по количеству экземпляров каждого слова в убывающем порядке. Выполните первую программу по отношению к нескольким файлам и посмотрите, осуществляет ли вторая программа правильную сортировку.

В этой главе:

- *Преобразование awk-программ в Perl-программы*
- *Преобразование sed-программ в Perl-программы*
- *Преобразование shell-сценариев в Perl-программы*
- *Упрямление*

18

Преобразование других программ в Perl-программы

Преобразование *awk*-программ в *Perl*-программы

Одна из самых замечательных особенностей Perl состоит в том, что он представляет собой семантическое надмножество (как минимум) языка *awk*. С практической точки зрения это значит, что если вы можете сделать что-либо в *awk*, вы сможете сделать это и в Perl. При этом, однако, Perl не обладает синтаксической совместимостью с *awk*. Например, переменная NR (номер входной записи) *awk* представляется в Perl как \$.

Если у вас есть *awk*-программа и вы хотите выполнить ее Perl-вариант, можно осуществить механическое преобразование этой программы с помощью утилиты *a2p*, которая есть в дистрибутиве Perl. Эта утилита конвертирует синтаксис *awk* в синтаксис Perl и может создать непосредственно выполняемый Perl-сценарий для подавляющего большинства *awk*-программ.

Чтобы воспользоваться утилитой *a2p*, поместите свою *awk*-программу в отдельный файл и вызовите *a2p* с именем этого файла в качестве аргумента или переадресуйте стандартный ввод *a2p* на ввод из этого файла. В результате на стандартном выводе *a2p* вы получите нормальную Perl-программу. Например:

```
$ cat myawkprog
BEGIN { sum = 0 }
/llama/ { sum += $2 }
END { print "The llama count is " sum }
```



```
$ a2p <myawkprog >myperlprog
$ perl myperlprog somefile
The llama count is 15
$
```

Можно также направить стандартный вывод *a2p* прямо в Perl, потому что интерпретатор Perl принимает программу со стандартного ввода, если получает такое указание:

```
$ a2p <myawkprog | perl - somefile
The llama count is 15
$
```

Преобразованный для использования в Perl *awk*-сценарий, как правило, выполняет идентичную функцию, часто с большей скоростью и, конечно, без каких-либо присущих *awk* ограничений на длину строки, количество параметров и т.д. Некоторые преобразованные Perl-программы могут выполняться медленнее; Perl-действие, эквивалентное данной *awk*-операции, не обязательно будет самым эффективным Perl-кодом, по сравнению с написанным вручную.

Вы, возможно, захотите оптимизировать преобразованный Perl-код или добавить в Perl-версию программы новые функциональные возможности. Это сделать довольно просто, поскольку полученный Perl-код читается достаточно легко (учитывая, что перевод выполняется автоматически, следует отметить это как большое достижение).

В некоторых случаях перевод не выполняется механически. Например, сравнение “меньше чем” и для чисел, и для строк в *awk* выражается операцией `<`. В Perl для строк используется `lt`, а для чисел — операция `<`. В большинстве случаев *awk*, как и утилита *a2p*, делает разумное предположение относительно числового или строкового характера двух сравниваемых значений. Однако вполне возможно, что о каких-нибудь двух значениях будет известно недостаточно много для того, чтобы определить, какое должно выполняться сравнение — числовое или строковое, поэтому *a2p* использует наиболее вероятную операцию и помечает возможно ошибочную строку знаками `###` (Perl-комментарием) и пояснением. После преобразования обязательно просмотрите результат на предмет наличия таких комментариев и проверьте сделанные предположения. Более подробно о работе утилиты *a2p* рассказывается на ее `man`-странице. Если этой утилиты в том каталоге, откуда вы вызываете Perl, нет, громко пожалуйте тому, кто устанавливал вам Perl.

Преобразование *sed*-программ в *Perl*-программы

Может быть, вам покажется, что мы повторяемся, но угадайте, что мы сейчас скажем? А вот что: *Perl* — семантическое надмножество не только *awk*, но и *sed*.

С дистрибутивом поставляется конвертор *sed-Perl*, который называется *s2p*. Как и *a2p*, *s2p* получает со стандартного ввода *sed*-сценарий и направляет на стандартный вывод *Perl*-программу. В отличие от результата работы *a2p*, преобразованная программа редко ведет себя не так, как нужно, поэтому вы вполне можете рассчитывать на ее нормальную работу (при отсутствии дефектов в самой *s2p* или *Perl*).

Конвертированные *sed*-программы могут работать быстрее или медленнее оригинала. Как правило, они работают значительно быстрее (благодаря хорошо оптимизированным *Perl*-программам обработки регулярных выражений).

Конвертированный *sed*-сценарий может работать с опцией *-n* или без нее. Опция *-n* имеет тот же смысл, что и соответствующий ключ для *sed*. Чтобы воспользоваться этой опцией, конвертированный сценарий должен направить сам себя в препроцессор *C*, а это несколько замедляет запуск. Если вы знаете, что всегда будете вызывать конвертированный *sed*-сценарий с опцией *-n* или без нее (например, при преобразовании *sed*-сценария, используемого в больших *shell*-программах с известными аргументами), вы можете информировать утилиту *s2p* об этом (посредством ключей *-n* и *-p*), и она оптимизирует сценарий под выбранный вами ключ.

В качестве примера, демонстрирующего высокую универсальность и мощь языка *Perl*, отметим тот факт, что транслятор *s2p* написан на *Perl*. Если вы хотите увидеть, как Ларри программирует на *Perl*, взгляните на этот транслятор — только сначала сядьте, чтобы не упасть (даже с учетом того, что это очень древний код, относительно не изменившийся с версии 2).

Преобразование *shell*-сценариев в *Perl*-программы

Вы, наверное, подумали, что речь пойдет о конверторе, осуществляющем преобразование *shell*-сценариев в *Perl*-программы?

А вот и нет. Многие спрашивают о такой программе, но проблема в том, что большинство из того, что делает сценарий, делается отнюдь не им самим. Большинство сценариев *shell* тратят практически все свое время на вызовы отдельных программ для извлечения фрагментов строк, сравнения чисел, конкатенации файлов, удаления каталогов и т.д. Преобразование такого сценария в *Perl* требовало бы понимания принципа работы каждой из

вызываемых утилит или переключив вызов каждой из них на плечи Perl, что абсолютно ничего не дает.

Поэтому лучшее, что вы можете сделать, — это посмотреть на сценарий shell, определить, что он делает, и начать все с нуля, но уже на Perl. Конечно, вы можете провести на скорую руку транслитерацию — поместив основные части исходного сценария в вызовы `system()` или заключив их в обратные кавычки. Возможно, вам удастся заменить некоторые операции командами Perl: например, заменить `system(rm fred)` на `unlink(fred)` или цикл *for* shell на такой же цикл Perl. В большинстве случаев, однако, вы увидите, что это несколько напоминает конвертирование программы, написанной на Коболе, в С (почти с тем же уменьшением количества символов и, как следствие, повышением степени неразборчивости текста программы).

Упражнение

Ответ см. в приложении А.

1. Преобразуйте следующий сценарий shell в Perl-программу:

```
cat /etc/passwd |
awk -F: '{print $1, $6}' |
while read user home
do
    newsrc="$home/.newsrc"
    if [ -r $newsrc ]
    then
        if grep -s `^comp\.lang\.perl\.announce:` $newsrc
        then
            echo -n "$user is a good person, ";
            echo "and reads comp.lang.perl.announce!"
        fi
    fi
done
```

В этой главе:

- *Модуль CGI.pm*
- *Ваша CGI-программа в контексте*
- *Простейшая CGI-программа*
- *Передача параметров через CGI*
- *Как сократить объем вводимого текста*
- *Генерирование формы*
- *Другие компоненты формы*
- *Создание CGI-программы гостевой книги*
- *Поиск и устранение ошибок в CGI-программах*
- *Perl и Web: не только CGI-программирование*
- *Дополнительная литература*
- *Упражнения*

CGI- программирование

Если в течение последних нескольких лет вы не сидели взаперти в деревянной хижине без электричества, то вы наверняка слышали о World Wide Web. Web-адреса (больше известные как URL) сейчас можно найти везде: на рекламных плакатах и в титрах кинофильмов, на обложках журналов и на страницах других изданий, от газет до правительственных отчетов.

Многие из самых интересных Web-страниц включают разного рода формы, предназначенные для ввода данных пользователем. Вы вводите данные в такую форму и щелкаете на кнопке или рисунке. Это действие запускает некую программу на Web-сервере, которая изучает введенные вами данные и генерирует новую выходную информацию. Иногда эта программа (широко известная как программа общего шлюзового интерфейса, или CGI-программа) представляет собой просто интерфейс к существующей базе данных; она преобразует введенные вами данные в нечто понятное для этой базы данных, а выходную информацию базы данных — в нечто понятное для Web-браузера (обычно в HTML-форме).

CGI-программы не просто обрабатывают данные, введенные в форму. Они вызываются и тогда, когда вы щелкаете на графическом изображении, и фактически могут использоваться для отображения всего того, что “видит” ваш браузер. Web-страницы с CGI-поддержкой — это не безликие и нудные документы, а удивительно живые страницы с динамически изменяющимся содержанием. Именно динамическая информация делает Web интересным и интерактивным источником информации, а не просто средством, предназначенным для чтения книги с терминала.

Независимо от того, во что вас, возможно, заставят поверить все эти отскакивающие шарики и прыгающие объявления, Web содержит также большой объем текста. Поскольку мы имеем дело с текстом, файлами, сетевыми коммуникациями и в некоторой степени с двоичными данными, то Perl идеально подходит для Web-программирования.

В этой главе мы не только изучим основы CGI-программирования, но и параллельно получим определенные начальные сведения о гипертекстовых ссылках, библиотечных модулях и объектно-ориентированном программировании на Perl. В конце главы мы дадим начальные сведения о применении Perl для других видов Web-программирования.

Если рассматривать эту главу как отдельное пособие, следует отметить, что ее (как и любого другого документа объемом меньше пары сотен страниц) недостаточно для изучения более сложных тем, затронутых здесь, в частности объектно-ориентированного программирования и методов использования различных типов ссылок и запросов в WWW. Однако, будучи средством всего лишь предварительного ознакомления с тем, что вас ждет, представленные здесь примеры и пояснения к ним, возможно, побудят вас подробнее ознакомиться с затронутыми темами и дадут вам ориентиры для выбора соответствующих учебников. А если вы любите учиться на практике, они помогут вам сразу написать некоторые полезные программы на основе тех моделей, которые здесь представлены.

Мы предполагаем, что вы уже в основном знакомы с HTML.

Модуль CGI.pm

Начиная с версии 5.004, в состав стандартного дистрибутива Perl включается модуль CGI.pm, который все знает и все умеет*.

Этот модуль, который написал Линкольн Штейн, автор хорошо известной книги *How to Setup and Maintain Your Web Site*, превращает процедуру создания CGI-программ на Perl в легкую прогулку. Как и сам Perl, CGI.pm является платформо-независимым, поэтому его можно использовать практически с любой ОС, от UNIX и Linux до VMS; он работает даже в таких системах, как Windows и MacOS.

* Если у вас установлена одна из более ранних версий Perl (но как минимум 5.001) и вы еще не собрались переходить на новую, просто получите CGI.pm из CPAN.

Если CGI.pm уже установлен у вас в системе, вы можете прочесть его полную документацию, воспользовавшись любым из способов, которые вы используете для чтения map-страниц Perl, например с помощью команд *man(1)* или *perldoc(1)* либо обратившись к HTML-варианту документации. Если ничего не получается, прочтите файл *CGI.pm*: документация на модуль встроена в сам модуль, представленный в простом формате *pod* *.

Разрабатывая CGI-программы, держите экземпляр map-страницы модуля CGI.pm под рукой. Она не только содержит описание функций этого модуля, но и загружается вместе со всевозможными примерами и советами.

Ваша CGI-программа в контексте

На рис. 19.1 показаны взаимосвязи между Web-браузером, Web-сервером и CGI-программой. Когда вы, работая со своим браузером, щелкаете на какой-либо ссылке, помните, что с этой ссылкой связан универсальный локалатор ресурса, URL (Uniform Resource Locator). Этот URL указывает на Web-сервер и ресурс, доступный через данный сервер. Таким образом, браузер взаимодействует с сервером, запрашивая указанный ресурс. Если, скажем, ресурс представляет собой HTML-форму, предназначенную для заполнения, то Web-сервер загружает эту форму в браузер, который затем выводит ее на экран, чтобы вы могли ввести требуемые данные.

Каждое предназначенное для ввода текста поле в этой форме имеет имя (указанное в HTML-коде формы) и соответствующее значение, которым является все, что вы вводите в этом поле. Сама форма связана (через HTML-директиву `<FORM>`) с CGI-программой, которая обрабатывает данные, введенные в форму. Когда вы, заполнив форму, щелкаете на кнопке Submit, браузер обращается к URL CGI-программы. Перед этим он добавляет в конец URL так называемую *строку запроса*, которая состоит из одной или более пар *имя=значение*; каждое имя — это имя поля, предназначенного для ввода текста, а каждое значение — данные, которые вы ввели. Таким образом, URL, в который браузер передает данные, введенные вами в форму, выглядит приблизительно так (строкой запроса считается все, что стоит после вопросительного знака):

```
http://www.SOMEWHERE.org/cgi-bin/some_cgi_prog?flavor=vanilla&size=double
```

Вы видите здесь две пары *имя=значение*. Такие пары отделяются друг от друга амперсандом (&). Работая с модулем CGI.pm, об этой детали можете не беспокоиться. Компонент `/cgi-bin/some_cgi_prog/` мы рассмотрим немного позднее; на данный момент важно лишь то, что он содержит путь к CGI-программе, которая будет обрабатывать данные, введенные в HTML-форму.

* Pod сокращенно обозначает plain old documentation (“обычная старая документация”). Это упрощенная форма представления, используемая для всей документации на Perl. Принцип работы этого формата изложен на map-странице *perlpod(1)*, а некоторые pod-трансляторы описаны на map-страницах *pod2man(1)*, *pod2html(1)* и *pod2text(1)*.

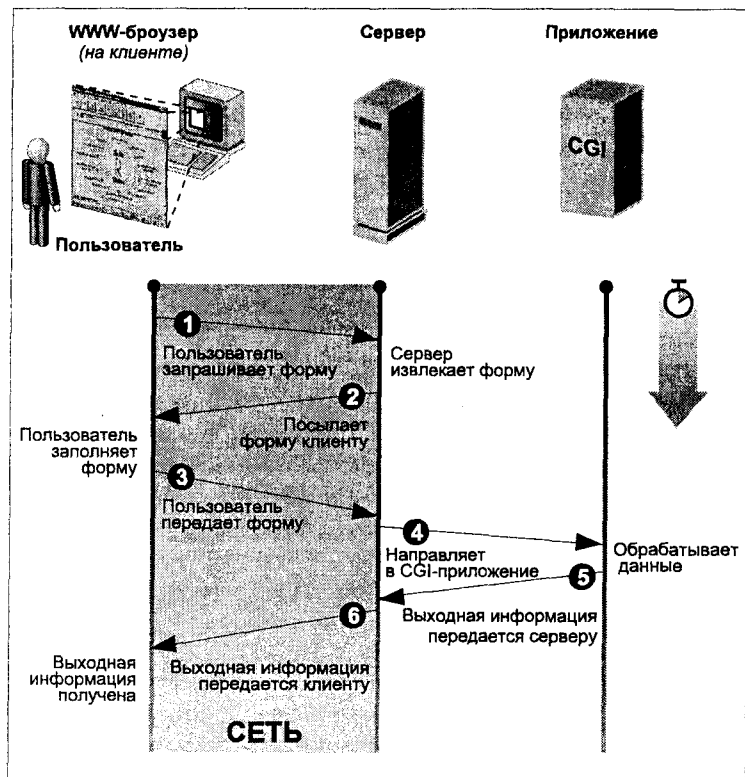


Рис. 19.1. Заполнение формы с привлечением CGI

Когда Web-сервер (в данном случае *www.SOMEWHERE.org*) получает URL от вашего браузера, он вызывает указанную CGI-программу и передает в нее в качестве аргументов пары имя=значение. Программа затем делает то, что должна делать, и (как правило) возвращает HTML-код серверу, который, в свою очередь, загружает его в браузер для представления пользователю.

“Переговоры” между браузером и сервером, а также между сервером и CGI-программой ведутся в соответствии с протоколом, известным как HTTP. При написании CGI-программы об этом беспокоиться не нужно, т.к. модуль CGI.pm сам решает все вопросы, связанные с использованием протокола.

Способ получения CGI-программой ее аргументов (и другой информации) от браузера через сервер описывается спецификацией Common Gateway Interface. Об этом тоже не нужно беспокоиться; как вы вскоре увидите, CGI.pm автоматически обрабатывает эти аргументы, не требуя от вас каких-либо дополнительных действий.

Наконец, вам следует знать, что CGI-программы могут работать с любым HTML-документом, а не только с формой. Например, вы могли бы написать такой HTML-код:

```
Click <a href="http://www.SOMEWHERE.org/cgi-bin/fortune.cgi">here</a> to
receive your fortune.
```

Здесь *fortune.cgi* — программа, которая просто вызывает программу *fortune* (в UNIX-системах). В данном случае в CGI-программу не вводятся никакие аргументы. Другой вариант: HTML-документ мог бы содержать две ссылки для пользователя — одну для получения предсказания судьбы, вторую для выяснения текущей даты. Обе ссылки могли бы указывать на одну и ту же программу — в одном случае с аргументом *fortune*, поставленным в URL после вопросительного знака, а в другом случае — с аргументом *date*. Эти HTML-ссылки выглядели бы так:

```
<a href="http://www.SOMEWHERE.org/cgi-bin/fortune_or_date?fortune">
<a href="http://www.SOMEWHERE.org/cgi-bin/fortune_or_date?date">
```

CGI-программа (в данном случае *fortune_or_date*) определила бы, какой из двух возможных аргументов получен, и выполнила бы соответственно программу *fortune* или программу *date*.

Как видите, аргументы вовсе не должны иметь формат имя=значение, характерный для HTML-форм. Вы можете написать CGI-программу, которая будет делать практически все, что вы пожелаете, и можете передавать ей практически любые аргументы.

В этой главе мы будем, главным образом, рассматривать применение HTML-форм. При этом мы предполагаем, что вы уже знакомы с простыми HTML-кодами.

Простейшая CGI-программа

Вот исходный код вашей первой CGI-программы. Она настолько проста, что в ней даже не пришлось использовать модуль CGI.pm:

```
#!/usr/bin/perl5 -w
#самая легкая из CGI-программ
print <<END of Multiple Text;
Content-type: text/html

<HTML>
  <HEAD>
    <TITLE>Hello World</TITLE>
  </HEAD>
  <BODY>
    <H1>Greetings, Terrans!</H1>
  </BODY>
</HTML>

END_of_Multiline_Text
```

Каждый раз, когда эта программа вызывается, она выдает на экран одно и то же. Это, конечно, не особенно интересно, но позднее мы сделаем ее более занимательной.

Эта программка содержит всего один оператор: вызов функции `print`. Несколько забавно выглядящий аргумент — это так называемый *here-документ*. Он состоит из двух знаков “меньше чем” и слова, которое мы назовем конечной лексемой. Для программиста, работающего с `shell`, написанное, возможно, будет похоже на переадресацию ввода-вывода, но на самом деле это просто удобный способ взятия в кавычки строкового значения, занимающего несколько строк. Это строковое значение начинается на следующей строке программы и продолжается до строки, содержащей конечную лексему, которая должна стоять в самом начале этой строки; ничего другого в этой строке быть не должно. Неге-документы особенно полезны для создания HTML-документов.

Первая часть этого строкового значения — определенно самая важная: строка `Content-Type` задает тип генерируемой выходной информации. Сразу за ней идет пустая строка, которая не должна содержать пробелов и знаков табуляции.

У большинства новичков первые CGI-программы отказываются работать, потому что пользователи забывают об этой пустой строке, отделяющей заголовок (нечто вроде заголовка сообщения электронной почты) от следующего за ним необязательного тела*. После пустой строки следует HTML-документ, посылаемый в браузер пользователя, где он форматируется и отображается.

Сначала добейтесь, чтобы ваша программа правильно выполнялась при вызове ее из командной строки. Это необходимый, но не достаточный шаг для того, чтобы обеспечить функционирование вашей программы как сценария, работающего на сервере. Ошибки могут возникать и в других местах программы; см. ниже раздел “Поиск и устранение ошибок в CGI-программах”.

Если программа должным образом работает при вызове ее из командной строки, необходимо инсталлировать ее на компьютере-сервере. Приемлемые места размещения зависят от сервера, хотя для CGI-сценариев часто используется каталог `/usr/etc/httpd/cgi-bin/` и его подкаталоги. Обсудите этот вопрос с Web-мастером или системным администратором.

После завершения инсталляции вашей программы в CGI-каталоге ее можно выполнять, указывая браузеру ее путевое имя в составе URL. Например, если ваша программа называется *howdy*, URL будет выглядеть так: `http://www.SOMEWHERE.org/cgi-bin/howdy`.

Серверы обычно позволяют использовать вместо длинных путевых имен псевдонимы. Сервер, имеющий адрес `www.SOMEWHERE.org`, может запросто перевести `cgi-bin/howdy`, содержащийся в этом URL, в нечто вроде `usr/etc/httpd/cgi-bin/howdy`. Ваш системный администратор или Web-мастер может подсказать, какой псевдоним следует использовать при обращении к вашей программе.

* Этот заголовок необходим для протокола HTTP, о котором мы упоминали выше.

Передача параметров через CGI

Для передачи параметров в CGI-программы (точнее, в большинство CGI-программ) никакие формы не нужны. Чтобы убедиться в этом, замените URL на http://www.SOMEWHERE.org/cgi-bin/ice_cream?flavor=mint.

Когда вы “нацеливаете” свой браузер на этот URL, браузер не только просит Web-сервер вызвать программу `ice_cream`, но и передает в нее строку `flavor=mint`. Теперь дело программы — прочитать данную строку-аргумент и разобрать ее. Эта задача не так проста, как кажется. Многие программы пытаются решить ее и разобрать запрос самостоятельно, но большинство “самодельных” алгоритмов время от времени отказывают. Учитывая то, насколько сложно найти правильное решение такой задачи для всех возможных случаев, вам, наверное, не следует писать код самим, особенно при наличии отличных готовых модулей, которые выполняют этот хитрый синтаксический анализ за вас.

К вашим услугам — модуль `CGI.pm`, который всегда разбирает входящий CGI-запрос правильно. Чтобы вставить этот модуль в свою программу, просто напишите

```
use CGI;
```

где-нибудь в начале программы*.

Оператор `use` похож на оператор `#include` языка C тем, что в процессе компиляции извлекает код из другого файла. Но он допускает также использование необязательных аргументов, показывающих, к каким функциям и переменным из этого модуля вы хотели бы обращаться. Поместите их в список, следующий за именем модуля в операторе `use`, — и вы сможете обращаться к указанным функциям и переменным так, как будто они ваши собственные.

В данном случае все, что нам нужно использовать из модуля `CGI.pm` — это функция `param()`**.

Если аргументы не указаны, функция `param()` возвращает список всех полей, имевшихся в HTML-форме, на которую отвечает данный CGI-сценарий. (В текущем примере это поле `flavor`, а в общем случае — список всех имен, содержащихся в строках `имя=значение` переданной формы.) Если указан аргумент, обозначающий поле, то `param()` возвращает значение (или значения), связанные с этим полем. Следовательно, `param("flavor")` возвращает `"mint"`, потому что в конце URL мы передали `?flavor=mint`.

* Имена всех Perl-модулей имеют расширение `.pm`. Более того, оператор `use` подразумевает это расширение. О том, как создавать свои собственные модули, вы можете узнать в главе 5 книги *Programming Perl* или на map-странице `perlmod(1)`.

** Некоторые модули автоматически экспортируют все свои функции, но, поскольку `CGI.pm` — это на самом деле объектный модуль, замаскированный под обычный, мы должны запрашивать его функции явно.

Несмотря на то что в нашем списке для оператора `use` имеется всего один элемент, мы будем использовать запись `qw()`. Благодаря этому нам будет легче впоследствии раскрыть этот список.

```
#!/usr/local/bin/perl5 -w
# программа ответа на форму о любимом сорте мороженого (версия 1)
use CGI qw(param);

print <<END_of_Start;
Content-type: text/html

<HTML>
  <HEAD>
  <TITLE>Hello World</TITLE>
  </HEAD>
  <BODY>
  <H1>Greetings, Terrans!</H1>
END_of_Start

my $favorite = param("flavor");
print "<P>Your favorite flavor is $favorite.";
print <<All_Done;
  </BODY>
  </HTML>
All_Done
```

Как сократить объем вводимого текста

Вводить все равно приходится очень много, но в `CGI.pm` есть множество удобных функций, упрощающих набор. Каждая из этих функций возвращает строковое значение, которое вы будете выводить. Например, `header()` возвращает строковое значение, содержащее строку `Content-type` с последующей пустой строкой, `start_html(строка)` возвращает указанную строку как HTML-титул (название документа), `h1(строка)` возвращает указанную строку как HTML-заголовок первого уровня, а `p(строка)` возвращает указанную строку как новый HTML-абзац.

Мы могли бы перечислить все эти функции в списке, прилагаемом к оператору `use`, но такой список разросся бы до небывалых размеров. В `CGI.pm`, как и во многих других модулях, имеются так называемые *директивы импорта* — метки, которые обозначают группы импортируемых функций. Вам нужно лишь поставить желаемые директивы (каждая из которых начинается двоеточием) в начале своего списка импорта. В модуле `CGI.pm` имеются такие директивы:

```
:cgi
```

Импортировать все методы обработки аргументов, например `param()`.

```
:form
    Импортировать все методы создания заполняемых форм, например text-
    field().

:html2
    Импортировать все методы, которые генерируют стандартные элементы
    HTML 2.0.

:html3
    Импортировать все методы, которые генерируют элементы, предложен-
    ные в HTML 3.0 (такие как <table>, <sup> и <sub>).

:netscape
    Импортировать все методы, которые генерируют расширения HTML,
    характерные для Netscape.

:shortcuts
    Импортировать все сокращения, генерируемые HTML (т.е. "html2" +
    "html3" + "netscape").

:standard
    Импортировать "стандартные" возможности: "html2", "form" и "cgi".

:all
    Импортировать все имеющиеся методы. Полный список приведен в
    модуле CGI.pm, где определяется переменная %TAGS.
```

Мы будем использовать только директиву `:standard`. (Подробная информация об импортировании функций и переменных из модулей приведена в главе 7 книги *Programming Perl*, а также на man-странице *Exporter(3)*.)

Вот как выглядит наша программа со всеми сокращениями, которые используются в CGI.pm:

```
#!/usr/local/bin/perl5 -w
# cgi-bin/ice_cream # программа ответа на форму о любимом
# сорте мороженого (версия 2)

use CGI qw(:standard);
print header();
print start_html("Hello World"), h1("Hello World");
my $favorite = param("flavor");
print p("Your favorite flavor is $favorite.");
print end_html();
```

Видите, насколько это проще? Вам не нужно беспокоиться о декодировании данных формы, о заголовках и HTML-тексте, если вы этого не хотите.

Генерирование формы

Если вам надоело вводить параметры своей программы в браузер — создайте заполняемую форму. К таким формам привыкли большинство пользователей. Компоненты формы, которые принимают вводимые пользователем данные, иногда называются *widgets*; считается, что этот термин гораздо удобнее, чем “устройства графического ввода”. Такие компоненты форм включают одно- и многостроковые текстовые поля, всплывающие меню, прокручиваемые списки, различные виды кнопок и отмечаемых блоков.

Создайте следующую HTML-страницу, которая включает форму с одним компонентом “текстовое поле” и кнопкой передачи. Когда пользователь щелкает на кнопке передачи*, вызывается сценарий *ice_cream*, заданный атрибутом ACTION.

```
<'-- ice_cream.html -->
<HTML>
  <HEAD>
    <TITLE>Hello Ice Cream</TITLE>
  </HEAD>
  <BODY>
    <H1>Hello Ice Cream!</H1>
    <FORM ACTION="http://www.SOMEWHERE.org/cgi-bin/ice_cream">
      What's your flavor? <INPUT NAME="favorite" VALUE="mint">
    <P>
    <INPUT TYPE="submit">
  </FORM>
</BODY>
</HTML>
```

Помните, что CGI-программа может выдавать ту выходную HTML-информацию, которую вы ей укажете. Эта информация будет затем передаваться в тот браузер, который обратится к URL данной программы. CGI-программа может, таким образом, не только реагировать на данные, введенные пользователем в форму, но и генерировать HTML-страницу с формой. Более того, одна программа может выполнять одну за другой обе эти задачи. Все, что вам нужно сделать, — это разделить программу на две части, которые делают разные вещи в зависимости от того, была ли программа вызвана с аргументами или нет. Если аргументов не было, программа посылает в браузер пустую форму; в противном случае аргументы содержат данные, введенные пользователем в ранее переданную форму, и программа возвращает в браузер ответ на основании этих данных.

* Некоторые браузеры позволяют обходиться без кнопки передачи, если форма содержит только одно поле для ввода текста. Если курсор находится в этом поле и пользователь нажимает клавишу [Enter], это считается запросом на передачу. Однако лучше здесь использовать традиционный способ.

При размещении всех компонентов программы в одном CGI-файле упрощается ее сопровождение. Цена — незначительное увеличение времени обработки при загрузке исходной страницы. Вот как все это выглядит:

```
#!/usr/local/bin/perl5 -w
# программа ответа на форму о любимом сорте мороженого
# *и генерирования этой формы* (версия 3)
use CGI qw(:standard);
my $favorite = param("flavor");
print header;
print start_html("Hello Ice Cream"), h1("Hello Ice Cream");
if ($favorite) {
    print p("Your favorite flavor is $favorite.");
} else {
    print hr, start_form;
    print p("Please select a flavor: ", textfield("flavor","mint"));
    print end_form, hr;
}
```

Если во время работы с браузером вы щелкнете на ссылке, которая указывает на эту программу (и если ссылка в конце URL не содержит ?whatever), то увидите экран, подобный изображенному на рис. 19.2. Текстовое поле изначально содержит значение по умолчанию, но это значение заменяется данными, введенными пользователями (если они есть).

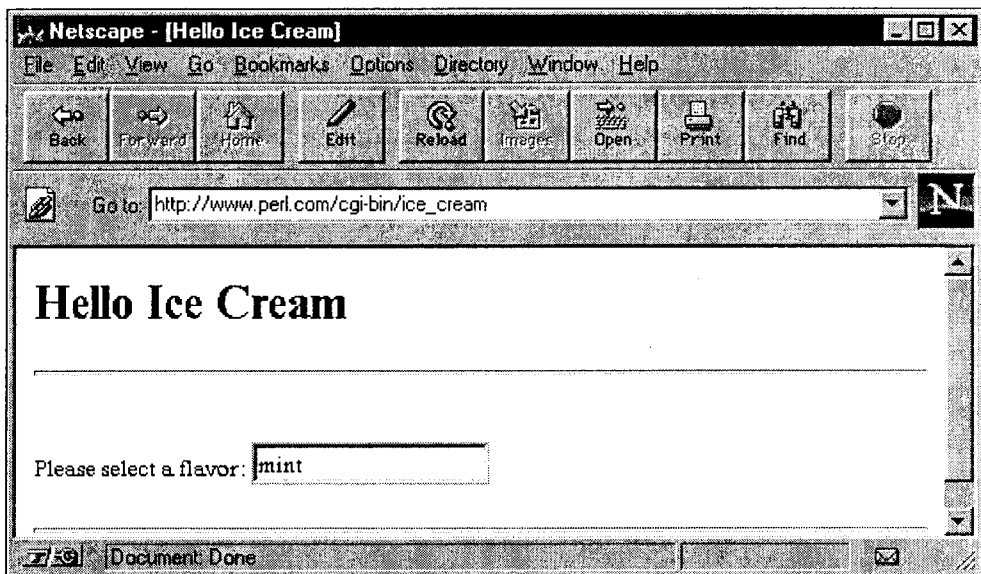


Рис. 19.2. Исходная заполняемая форма

Теперь заполните поле Please select a flavor, нажмите клавишу [Enter], и вы увидите то, что показано на рис. 19.3.

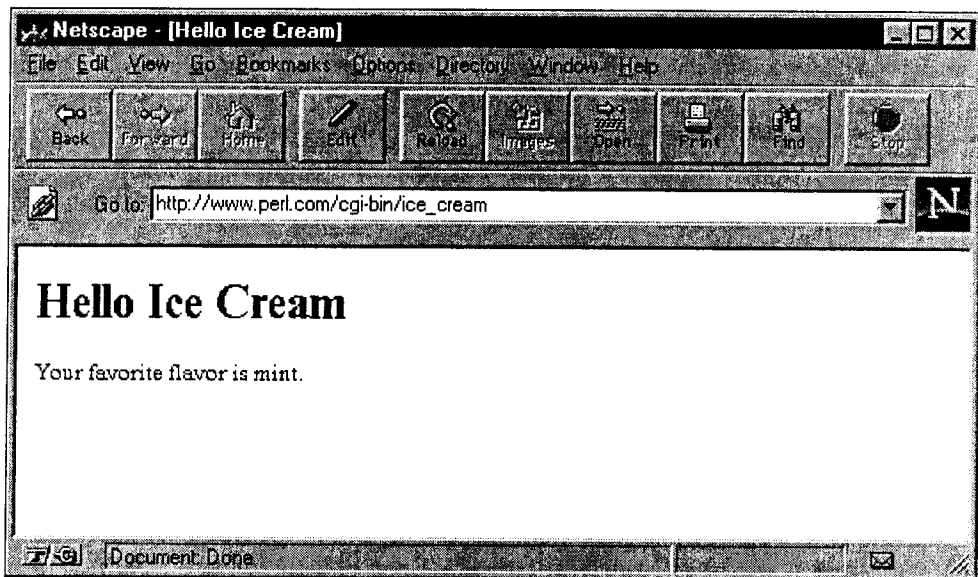


Рис. 19.3. Результат обработки переданного с использованием формы запроса

Другие компоненты формы

Теперь, когда вы знаете, как создавать в форме простые текстовые поля и заполнять их, вам, наверное, интересно будет узнать, как создавать компоненты формы других типов — кнопки, отмечаемые блоки и меню.

Сейчас мы рассмотрим более развитую версию нашей программы. В частности, мы включили в нее новые компоненты формы: всплывающие меню, кнопку передачи (которая называется `order`) и кнопку очистки полей формы, позволяющую стереть все данные, введенные пользователем. Всплывающие меню делают именно то, о чем говорят их имена, но аргументы, указанные в `popup_menu`, могут озадачить вас — пока вы не прочитаете следующий раздел, “Ссылки”. Функция `textfield()` создает поле для ввода текста с указанным именем. Подробнее об этой функции мы расскажем ниже, когда будем описывать программу гостевой книги.

```
#!/usr/local/bin/perl5 -w
# программа ответа на форму заказа мороженого и генерирования этой формы (версия 4)
use strict;
# ввести объявления переменных и выполнить заключение в кавычки
use CGI qw(:standard);

print header;
```

```

print start_html("Ice Cream Stand"), h1("Ice Cream Stand");
if (param()) { # форма уже заполнена
    my $who = param("name");
    my $flavor = param("flavor");
    my $scoops = param("scoops");
    my $taxrate = 1.0743;
    my $cost = sprintf("%.2f", $taxrate * (1.00 + $scoops * 0.25));
    print p("Ok, $who, have $scoops scoops of $flavor for \$$cost.");
} else { # первый проход, представить незаполненную форму
    print hr();
    print start_form();
    print p("What's your name? ",textfield("name"));
    print p("What flavor: ", popup_menu("flavor",
                                       ['mint','cherry','mocha']));
    print p("How many scoops? ", popup_menu("scoops", [1..3]));
    print p(submit("order"), reset("clear"));
    print end_form(), hr();
}
print end_html;

```

На рис. 19.4 представлено изображение начальной формы, которую создает рассматриваемая программа.

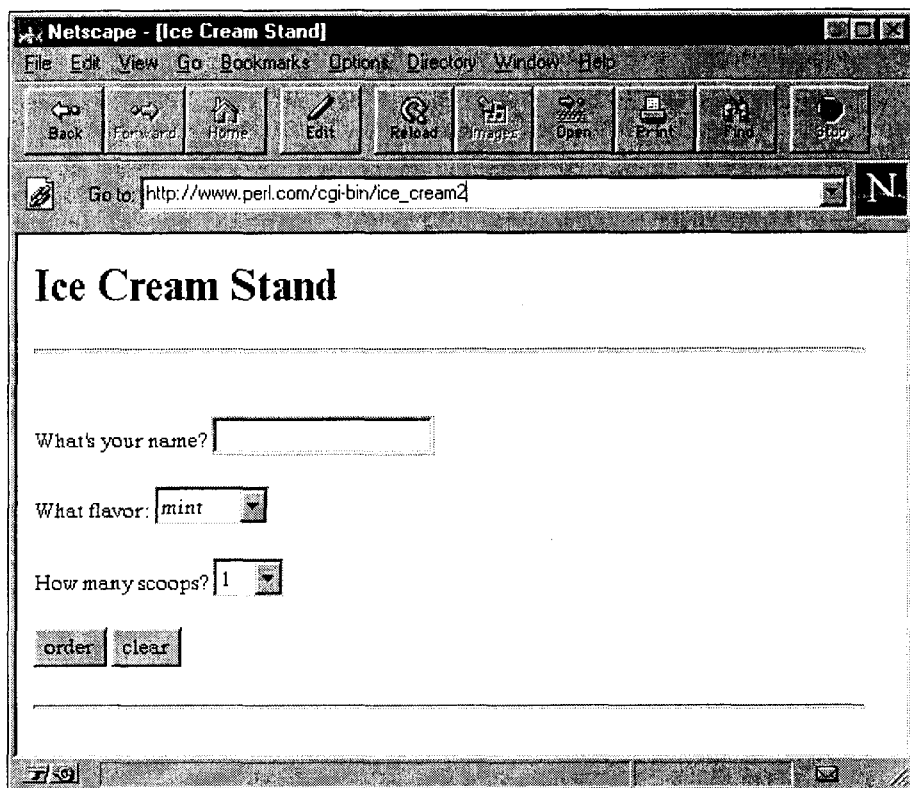


Рис. 19.4. Более сложная форма

Как вы помните, функция `param()` при вызове ее без аргументов возвращает имена всех полей формы, которые были заполнены. Таким образом вы можете узнать, была ли заполнена форма перед вызовом программы. Если у вас есть параметры, это значит, что пользователь заполнил некоторые поля существующей формы, поэтому на них нужно ответить. В противном случае следует генерировать новую форму с расчетом на вторичный вызов той же самой программы.

Ссылки

Вы, возможно, заметили, что обе функции `popup_menu()` в предыдущем примере имеют весьма странные аргументы. Что означают `['mint', 'cherry', 'mocha']` и `[1..3]`? Квадратные скобки создают нечто такое, с чем вы раньше не встречались: ссылку на анонимный массив. Это обусловлено тем, что функция `popup_menu()` в качестве аргумента рассчитывает получить именно ссылку на массив. Другой способ создания ссылки на массив — использовать перед именованным массивом обратную косую черту, например `\@choices`. Так, следующий фрагмент кода:

```
@choices = ('mint', 'cherry', 'mocha');
print p("What flavor: ", popup_menu("flavor", \@choices));
```

работает так же хорошо, как этот:

```
print p("What flavor: ", popup_menu("flavor",
['mint', 'cherry', 'mocha']));
```

Ссылки функционируют примерно так, как указатели в других языках, но с меньшей вероятностью появления ошибок. Они представляют собой значения, которые указывают на другие значения (или переменные). Ссылки Perl строго делятся на типы (без возможности приведения типов) и никогда не вызывают вывода дампов ядра операционной системы. Более того, если область памяти, на которую указывают ссылки, больше не используется, она автоматически возвращается в использование. Ссылки играют центральную роль в объектно-ориентированном программировании. Они применяются и в традиционном программировании, являясь основой для создания структур данных, более сложных, нежели простые одномерные массивы и хеши. Язык Perl поддерживает ссылки как на именованные, так и на анонимные скаляры, массивы, хеши и функции.

Так же, как методом `\@массив` можно создавать ссылки на именованные массивы и посредством указания `[список]` — на анонимные хеши, можно методом `%хеш` создавать ссылки на именованные хеши, а методом

```
{ ключ1, значение1, ключ2, значение2, ... }
```

— на анонимные*.

* Да, фигурные скобки теперь используются в Perl с различными целями. Их функцию определяет контекст, в котором используются фигурные скобки.

Подробнее о ссылках вы прочитаете в главе 4 книги *Programming Perl* и на ман-странице *perlref(1)*.

Более сложные вызывающие последовательности

Мы закончим наш рассказ о компонентах форм созданием одного очень полезного компонента, который позволяет пользователю выбирать любое число элементов этого компонента. Функция `scrolling_list()` модуля `CGI.pm` может принимать произвольное число пар аргументов, каждая из которых состоит из именованного параметра (начинающегося со знака `-`) и значения этого параметра.

Чтобы ввести в форму прокручиваемый список, нужно сделать следующее:

```
print scrolling_list(
    -NAME => "flavors",
    -VALUES => [ qw(mint chocolate cherry vanilla peach) ],
    -LABELS => {
        mint => "Mighty Mint",
        chocolate => "Cherished Chocolate",
        cherry => "Cherry Cherry",
        vanilla => "Very Vanilla",
        peach => "Perfectly Peachy",
    },
    -SIZE => 3,
    -MULTIPLE => 1, #1 for true , 0 for false
);
```

Значения параметров имеют следующий смысл:

-NAME

Имя компонента формы. Значение этого параметра можно использовать позже для выборки пользовательских данных из формы с помощью функции `param()`.

-LABELS

Ссылка на анонимный хеш. Значения хеша — это метки (элементы списка), которые видит пользователь формы. Когда пользователь выбирает ту или иную метку, в CGI-программу возвращается соответствующий ключ хеша. Например, если пользователь выбирает элемент, заданный как `Perfectly Peachy`, CGI-программа получает аргумент `peach`.

-VALUES

Ссылка на анонимный массив. Этот массив состоит из ключей хеша, на которые ссылается `-LABELS`.

-SIZE

Число, определяющее, сколько элементов списка пользователь будет видеть одновременно.

-MULTIPLE

Истинное или ложное значение (в том смысле, который принят для этих понятий в Perl), показывающее, можно ли будет пользователю формы выбирать более одного элемента списка.

Если -MULTIPLE установлена в значение “истина”, вы можете присвоить список, возвращаемый функцией `param()`, массиву:

```
@choices = param("flavors");
```

Вот другой способ создания этого прокручиваемого списка — с передачей ссылки на существующий хеш вместо создания такого хеша “на ходу”:

```
%flavors = (
    "mint", "Mighty Mint",
    "chocolate", "Cherished Chocolate",
    "cherry", "Cherry Cherry",
    "vanilla", "Very Vanilla",
    "peach", "Perfectly Peachy",
);
print scrolling_list(
    -NAME => "flavors",
    -LABELS => \%flavors,
    -VALUES => [ keys %flavors ],
    -SIZE => 3,
    -MULTIPLE => 1, #1 for true , 0 for false
);
```

На этот раз мы передаем в функцию значения, вычисленные по ключам хеша `%flavors`, ссылка на который выполняется с помощью операции `\`. Обратите внимание: параметр `-VALUES` здесь тоже взят в квадратные скобки. Простая передача результата операции `keys` в виде списка не сработает, потому что в соответствии с правилом вызова функции `scrolling_list()` должна быть сделана ссылка на массив, которую как раз и создают квадратные скобки. Считайте квадратные скобки удобным способом представления нескольких значений как одного.

Создание CGI-программы гостевой книги

Если вы внимательно изучили примеры, приведенные выше, то уже должны быть способны заставить работать простые CGI-программы. А как насчет более сложных? Одна из распространенных задач — создание CGI-программы для управления гостевой книгой, чтобы посетители вашего Web-узла могли записывать в нее свои собственные сообщения*.

* Как мы отметим ниже, это приложение можно было бы назвать программой *Webchat* (переговоров через Web).

Форма, используемая для создания гостевой книги, довольно проста, она даже проще, чем некоторые из наших форм, посвященных мороженому. Она, правда, имеет некоторые особенности, но не беспокойтесь, по мере продвижения к финишу мы преодолеем все трудности.

Вероятно, вы хотите, чтобы сообщения в гостевой книге сохранялись и по завершении посещения вашего узла тем или иным пользователем, поэтому вам нужен файл, куда они будут записываться. Гостевая CGI-программа (вероятно) работает не под вашим управлением, поэтому у нее, как правило, не будет права на обновление вашего файла. Следовательно, первое, что необходимо сделать, — это создать для нее файл с широкими правами доступа. Если вы работаете в UNIX-системе, то можете сделать (из своего shell) для инициализации файла программы гостевой книги следующее:

```
touch /usr/tmp/chatfile
chmod 0666 /usr/tmp/chatfile
```

Отлично, но как обеспечить одновременную работу с программой гостевой книги нескольких пользователей? Операционная система не блокирует попытки одновременного доступа к файлам, поэтому если вы будете недостаточно осторожны, то получите файл, в который записывают сообщения все пользователи одновременно. Чтобы избежать этого, мы используем Perl-функцию `flock`, позволяющую пользователю получить монополярный доступ к файлу, который мы разрешаем обновить. Это будет выглядеть примерно так:

```
use Fcntl qw(:flock);      # импортирует LOCK_EX, LOCK_SH, LOCK_NB
....
flock(CHANDLE, LOCK_EX) || bail("cannot flock $CHATNAME: $!");
```

Аргумент `LOCK_EX` функции `flock` — вот что позволяет нам обеспечить монополярный доступ к файлу*.

Функция `flock` представляет собой простой, но универсальный механизм блокировки, несмотря на то, что его базовая реализация существенно изменяется от системы к системе. Она не возвращает управление до тех пор, пока файл не будет разблокирован. Отметим, что блокировки файлов носят чисто рекомендательный характер: они работают только тогда, когда все процессы, обращающиеся к файлу, соблюдают эти блокировки одинаково. Если три процесса соблюдают блокировки, а один не соблюдает, то не функционирует ни одна из блокировок.

* В версиях Perl до 5.004 вы должны превратить в комментарий `use Fcntl` и в качестве аргумента функции `flock` использовать просто 2.

Объектно-ориентированное программирование на Perl

Наконец пришло время научить вас пользоваться объектами и классами — и это важнее всего. Хотя решение задачи построения вашего собственного объектного модуля выходит за рамки данной книги, это еще не повод для того, чтобы вы не могли использовать существующие объектно-ориентированные библиотечные модули. Подробная информация об использовании и создании объектных модулей приведена в главе 5 книги *Programming Perl* и на man-странице *perltoot(1)*.

Мы не будем углубляться здесь в теорию объектов, и вы можете просто считать их пакетами (чем они и являются!) удивительных, великолепных вещей, вызываемых косвенно. Объекты содержат подпрограммы, которые делают все, что вам нужно делать с объектами.

Пусть, например, модуль `CGI.pm` возвращает объект `$query`, который представляет собой входные данные пользователя. Если вы хотите получить параметр из этого запроса, вызовите подпрограмму `param()`:

```
$query->param("answer");
```

Данная запись означает: “Выполнить подпрограмму `param()` с объектом `$query`, используя “`answer`” как аргумент”. Такой вызов в точности соответствует вызову любой другой подпрограммы, за исключением того что вы используете имя объекта, за которым следует синтаксическая конструкция `->`. Кстати, подпрограммы, связанные с объектами, называются методами.

Если вы хотите получить значение, возвращенное подпрограммой `param()`, воспользуйтесь обычным оператором присваивания и сохраните это значение в обычной переменной `$he_said`:

```
$he_said = $query->param("answer");
```

Объекты выглядят как скаляры; они хранятся в скалярных переменных (таких как переменная `$query` в нашем примере), и из них можно составлять массивы и хеши. Тем не менее их не следует рассматривать как строки и числа. По сути дела, это особый вид ссылок, их нельзя рассматривать как обычные ссылки. Объекты следует трактовать как особый, определяемый пользователем тип данных.

Тип конкретного объекта известен как его класс. Имя класса обычно состоит из имени модуля без расширения `pm`, и к тому же термины “класс” и “модуль” часто используются как эквиваленты. Таким образом, мы можем говорить о CGI-модуле или о CGI-классе. Объекты конкретного класса создает и контролирует модуль, реализующий этот класс.

Доступ к классам осуществляется путем загрузки модуля, который выглядит точно так же, как любой другой модуль, за исключением того что объектно-ориентированные модули обычно ничего не экспортируют. Вы можете рассматривать класс как фабрику, которая производит совершенно новые объекты. Чтобы класс выдал один из таких объектов, нужно вызвать специальный метод, который называется конструктор.

Вот пример:

```
$query = CGI->new();      # вызвать метод new() в классе "CGI"
```

Здесь мы имеем дело с вызовом *метода класса*. Метод класса выглядит точно так же, как *метод объекта* (о котором мы говорили секунду назад), за исключением того что вместо использования объекта для вызова метода мы используем имя класса, как будто он сам — объект. Метод объекта говорит “вызвать функцию с этим именем, которая относится к данному объекту”, тогда как метод класса говорит “вызвать функцию с этим именем, которая относится к данному классу”.

Иногда то же самое записывается так:

```
$query = new CGI;         # то же самое
```

Вторая форма по принципу действия идентична первой. Здесь меньше знаков препинания, благодаря чему в некоторых случаях она более предпочтительна. Однако она менее удобна в качестве компонента большого выражения, поэтому в нашей книге мы будем использовать исключительно первую форму.

С точки зрения разработчика объектных модулей, объект — это ссылка на определяемую пользователем структуру данных, часто на анонимный хеш. Внутри этой структуры хранится всевозможная интересная информация. Воспитанный пользователь, однако, должен добираться до этой информации (с целью ее изучения или изменения), не рассматривая объект как ссылку и не обращаясь непосредственно к данным, на которые он указывает, а используя только имеющиеся методы объектов и классов. Изменение данных объекта другими средствами — это нечестная игра, после которой о вас обязательно станут говорить и думать плохо. Чтобы узнать о том, что представляют собой и как работают вышеупомянутые методы, достаточно прочитать документацию на объектный модуль, которая обычно прилагается в *pod*-формате.

Объекты в модуле CGI.pm

CGI-модуль необычен в том смысле, что его можно рассматривать либо как традиционный модуль с экспортируемыми функциями, либо как объектный модуль. Некоторые программы пишутся гораздо легче с помощью объектного интерфейса к модулю CGI.pm, нежели с помощью процедурного интерфейса к данному модулю. Наша гостевая книга — одна из таких программ. Мы получаем доступ к входной информации, которую пользователь ввел в форму, через CGI-объект и можем, при желании, с помощью этого же объекта генерировать новый HTML-код для отправки обратно пользователю.

Сначала, однако, нам нужно создать этот объект явно. Для CGI.pm, как и для многих других классов, метод, который позволяет создавать объекты, — это метод класса `new()`*.

* В отличие от C++ Perl не считает `new` ключевым словом; вы совершенно свободно можете использовать такие методы-конструкторы, как `gimme_another()` или `fred()`. Тем не менее большинство пользователей в итоге приходят к тому, что называют свои конструкторы во всех случаях `new()`.

Данный метод конструирует и возвращает новый CGI-объект, соответствующий заполненной форме. Этот объект содержит все данные, введенные пользователем в форму. Будучи вызванным без аргументов, метод `new()` создает объект путем чтения данных, переданных удаленным браузером. Если в качестве аргумента указан дескриптор файла, он читает этот дескриптор, надеясь найти в нем данные, введенные в форму в предыдущем сеансе работы с браузером.

Через минуту мы покажем вам эту программу и поясним ее работу. Давайте предположим, что она называется *guestbook* и находится в каталоге *cgi-bin*. Хотя она и не похожа ни на один из тех сценариев, которые мы рассмотрели выше (в которых одна часть выводит HTML-форму, а вторая читает данные, введенные в форму пользователем, и отвечает на них), вы увидите, что она, тем не менее, выполняет обе эти функции. Поэтому отдельный HTML-документ, содержащий форму гостевой книги, нам не нужен. Пользователь мог бы сначала запустить нашу программу, просто щелкнув мышкой на такой ссылке:

```
Please sign our
<A HREF="http://www.SOMEWHERE.org/cgi-bin/guestbook">guestb ook</A>.
```

Затем программа загружает в браузер HTML-форму и, на всякий случай, предыдущие сообщения гостей (в ограниченном количестве), чтобы пользователь мог их просмотреть. Пользователь заполняет форму, передает ее, и программа читает то, что передано. Эта информация добавляется в список предыдущих сообщений (хранящийся в файле), который затем вновь выводится в браузер, вместе со свежей формой. Пользователь может продолжать чтение текущего набора сообщений и передавать новые сообщения, заполняя предлагаемые формы, столько раз, сколько сочтет необходимым.

Вот наша программа. Перед тем как разбирать ее поэтапно, вы, возможно, захотите просмотреть программу целиком.

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

use 5.004;
use strict;                # установить объявления и взятие в кавычки
use CGI qw(:standard);     # импортировать сокращения согласно :standard
use Fcntl qw(:flock);      # импортирует LOCK_EX, LOCK_SH, LOCK_NB

sub bail {                 # функция обработки ошибок
    my $error = "@_";
    print hl("Unexpected Error"), p($error), end_html;
    die $error;
}

my(
    $CHATNAME,             # имя файла гостевой книги
    $MAXSAVE,              # какое количество хранить
    $TITLE,                # название и заголовок страницы
    @cur,                  # все текущие записи
```

```

$entry,          # одна конкретная запись
);

$TITLE = "Simple Guestbook";
$CHATNAME = "/usr/tmp/chatfile";  # где все это в системе находится
$MAXSAVE = 10;

print header, start_html($TITLE), h1($TITLE);

$cur = CGI->new();          # текущий запрос
if ($cur->param("message")) {    # хорошо, мы получили сообщение
    $cur->param("date", scalar localtime);  # установить текущее время
    @entries = ($cur);          # записать сообщение в массив
}

# открыть файл для чтения и записи (с сохранением предыдущего содержимого)
open(CHANDLE, "<+ $CHATNAME") || bail("cannot open $CHATNAME: $!");

# получить эксклюзивную блокировку на гостевую книгу
# (LOCK_EX == exclusive lock)
flock(CHANDLE, LOCK_EX) || bail("cannot flock $CHATNAME: $!");

# занести в $MAXSAVE старые записи (первой – самую новую)
while (!eof(CHANDLE) && @entries < $MAXSAVE) {
    $entry = CGI->new(\*CHANDLE);  # передать дескриптор файла по ссылке
    push @entries, $entry;
}
seek(CHANDLE, 0, 0) || bail("cannot rewind $CHATNAME: $!");
foreach $entry (@entries) {
    $entry->save(\*CHANDLE);      # передать дескриптор файла по ссылке
}
truncate(CHANDLE, tell(CHANDLE)) || bail("cannot truncate $CHATNAME: $!");
close(CHANDLE) || bail("cannot close $CHATNAME: $!");

print hr, start_form;          # hr() проводит горизонтальную линию: <HR>
print p("Name:", $cur->textfield(
    -NAME => "name"));
print p("Message:" $cur->textfield(
    -NAME => "message",
    -OVERRIDE => 1,             # стирает предыдущее сообщение
    -SIZE => 50));
print p(submit("send"), reset("clear"));
print end_form, hr;

print h2("Prior Messages");
foreach $entry (@entries) {
    printf("%s [%s]: %s",
        $entry->param("date"),
        $entry->param("name"),
        $entry->param("message"));
    print br();
}
print end_html;

```


На рис. 19.5 вы видите изображение, которое появляется на экране после запуска этой программы.

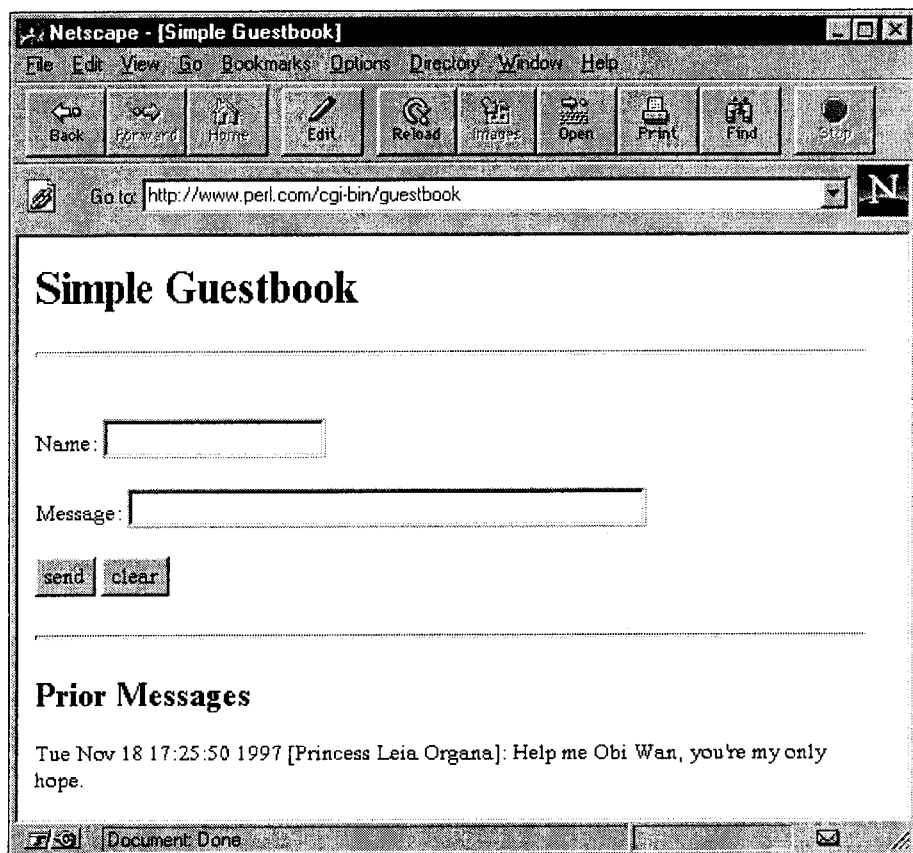


Рис. 19.5. Форма простой гостевой книги

Обратите внимание на то, что программа начинается с оператора

```
use 5.004;
```

Если вы хотите запускать ее с помощью более ранние версии Perl 5, то нужно превратить в комментарий строку

```
use Fcntl qw(:flock)
```

и заменить `LOCK_EX` в первом вызове `flock` на `Z`.

Поскольку каждое выполнение программы приводит к возврату HTML-формы в браузер, который обратился к программе, то программа начинается с задания HTML-кода:

```
print header, start_html($TITLE), h1($TITLE);
```

Затем создается новый CGI-объект:

```
$cur = CGI->new(); # текущий запрос
if ($cur->param("message")) { # хорошо, мы получили сообщение
$cur->param("date", scalar localtime); # установить текущее время
@entries = ($cur); # записать сообщение в массив
}
```

Если нас вызывают посредством заполнения и передачи формы, то объект `$cur` должен содержать информацию о тексте, введенном в форму. Форма, которую мы предлагаем (см. ниже), содержит два поля ввода: *поле имени* для ввода имени пользователя и *поле сообщения* для ввода сообщения. Кроме того, приведенный выше код ставит на введенные в форму данные (после их получения) метку даты. Передача в метод `param()` двух аргументов — это способ присваивания параметру, заданному первым аргументом, значения, указанного во втором аргументе.

Если нас вызывают не посредством передачи формы, а выполняя щелчок мышью на ссылке `Please sign our guestbook`, то объект запроса, который мы создаем, будет пуст. Проверка `if` даст значение “ложь”, и в массив `@entries` никакой элемент занесен не будет.

В любом случае мы переходим к проверке наличия записей, созданных ранее в нашем сохраняемом файле. Эти записи мы будем считывать в массив `@entries`. (Вспомните о том, что мы только что сделали текущие данные, если они имеются в форме, первым элементом этого массива.) Но сначала мы должны открыть сохраняемый файл:

```
open(CHANDLE, "<+ $CHATNAME") || bail("cannot open $CHATNAME. $!");
```

Эта функция открывает файл в режиме неразрушающего чтения-записи. Вместо `open` можно использовать `sysopen()`. При таком способе посредством единственного вызова открывается старый файл, если он существует (без уничтожения его содержимого), а в противном случае создается новый файл:

```
# нужно импортировать две "константы" из модуля Fcntl для sysopen
use Fcntl qw( O_RDWR O_CREAT );
sysopen(CHANDLE, $CHATFILE, O_RDWR|O_CREAT, 0666) || bail "can't open
$CHATFILE: $!";
```

Затем мы блокируем файл, как описывалось выше, и переходим к считыванию текущих записей из `$MAXSAVE` в `@entries`:

```
flock(CHANDLE, LOCK_EX) || bail("cannot flock $CHATNAME: $!");
while (!eof(CHANDLE) && @entries < $MAXSAVE) {
    $entry = CGI->new(\*CHANDLE); # передать дескриптор файла по ссылке
    push @entries, $entry;
}
```

Функция `eof` — встроенная Perl-функция, которая сообщает о достижении конца файла. Многократно передавая в метод `new()` ссылку на дескриптор сохраняемого файла*, мы выбираем старые записи, по одной при каждом вызове. Затем мы обновляем файл так, чтобы он включал новую запись, которую мы (возможно) только что получили:

```
seek(CHANDLE, 0, 0) || bail("cannot rewind $CHATNAME: $!");
foreach $entry (@entries) {
    $entry->save(*CHANDLE);    # передать дескриптор файла по ссылке
}
truncate(CHANDLE, tell(CHANDLE)) || bail("cannot truncate $CHATNAME: $!");
close(CHANDLE) || bail("cannot close $CHATNAME: $!");
```

Функции `seek`, `truncate` и `tell` — встроенные Perl-функции, описания которых вы найдете в любом справочнике по языку Perl. Здесь `seek` переставляет указатель позиции в начало файла, `truncate` усекает указанный файл до заданной длины, а `tell` возвращает текущее смещение указателя позиции в файле по отношению к началу файла. Назначение этих строк программы — сохранить в файле только самые последние записи `$MAXSAVE`, начиная с той, которая была сделана только что.

Метод `save()` обеспечивает собственно создание записей. Его можно вызвать здесь как `$entry->save`, поскольку `$entry` — это CGI-объект, созданный с помощью `CGI->new()`.

Формат записи сохраняемого файла выглядит следующим образом (запись завершается знаком "=", стоящим в отдельной строке):

```
ИМЯ1=ЗНАЧЕНИЕ1
ИМЯ2=ЗНАЧЕНИЕ2
ИМЯ3=ЗНАЧЕНИЕ3
=
```

Теперь пора вернуть обновленную форму браузеру и его пользователю. (Это будет, конечно, первая форма, которую он видит, в том случае, если он щелкнул на ссылке `Please sign our guestbook.`) Сначала некоторые предварительные действия:

```
print hr, start_form;    # hr() проводит горизонтальную линию. <HR>
```

Как мы уже упоминали, CGI.pm позволяет нам использовать либо прямые вызовы функций, либо вызовы методов через CGI-объект. В нашей программе для создания базового HTML-кода мы обратились к простым вызовам функций, а для задания полей ввода формы продолжаем пользоваться методами объектов:

```
print p("Name:", $cur->textfield(
    -NAME => "name"));
print p("Message:" $cur->textfield(
```

* Фактически она представляет собой glob-ссылку, а не ссылку на дескриптор файла, но в данном случае это почти то же самое

```

-NAME => "message",
-OVERRIDE => 1,      # стирает предыдущее сообщение
-SIZE => 50));
print p(submit("send"), reset("clear"));
print end_form, hr;

```

Метод `textfield()` возвращает поле ввода текста для нашей формы. Первый из двух приведенных выше вызовов генерирует HTML-код поля ввода текста с HTML-атрибутом, `NAME="name"`, а второй — создает поле с атрибутом `NAME="message"`.

Компоненты формы, создаваемые модулем `CGI.pm`, по умолчанию устойчивы: они сохраняют свои значения до следующего вызова. (Но лишь в течение одного “сеанса” работы с формой, считая с того момента, когда пользователь щелкнул на ссылке *Please sign our guestbook.*) Это значит, что поле `NAME="name"`, созданное в результате первого вызова `textfield()`, будет содержать значение имени пользователя, если он уже хотя бы один раз в этом сеансе заполнял и передавал форму. Таким образом, поле ввода, которое мы сейчас создаем, будет иметь следующие HTML-атрибуты:

```
NAME="name" VALUE="Sam Smith"
```

Совсем другое дело — второй вызов `textfield()`. Мы не хотим, чтобы поле сообщения содержало значение старого сообщения. Поэтому пара аргументов `-OVERRIDE => 1` указывает: “Выбросить предыдущее значение этого текстового поля и восстановить значение по умолчанию”. Пара аргументов `-SIZE => 50` задает размер (в символах) отображаемого поля ввода. Помимо показанных здесь, могут быть и другие необязательные пары аргументов: `-DEFAULT => 'начальное значение'` и `-MAXLENGTH => n`, где `n` — максимальное число символов, которое может принять данное поле.

Наконец, к удовольствию пользователя, мы выводим текущий перечень сохраняемых сообщений, включающий, естественно, то, которое он только что передал:

```

print h2("Prior Messages");
foreach $entry (@entries) {
    printf("%s [%s]: %s",
        $entry->param("date"),
        $entry->param("name"),
        $entry->param("message"));
    print br();
}
print end_html;

```

Как вы, без сомнения, догадываетесь, функция `h2` задает HTML-заголовок второго уровня. В остальной части кода мы просто последовательно формируем текущий список сохраняемых записей (это тот же список, который мы ранее записали в сохраняемый файл), выводя из каждой дату, имя и сообщение.

Пользователи могут работать с этой формой гостевой книги, непрерывно набирая сообщения и нажимая кнопку передачи. Это имитирует электронную доску объявлений, позволяя пользователям видеть новые сообщения друг друга сразу же после их передачи. Общаясь друг с другом подобным образом, пользователи многократно вызывают одну и ту же CGI-программу; предыдущие значения компонентов формы автоматически сохраняются до следующего вызова. Это особенно удобно при создании многоступенчатых форм — например, тех, которые используются в так называемых приложениях “тележек для покупок”, когда вы, перемещаясь по виртуальному магазину, последовательно “делаете покупки” и форма все их запоминает.

Поиск и устранение ошибок в CGI-программах

CGI-программы, запускаемые с Web-сервера, функционируют в совершенно иной среде, нежели та, в которой они работают при вызове из командной строки. Конечно, вы всегда должны добиваться, чтобы ваша CGI-программа нормально работала при вызове ее из командной строки*, но это не гарантирует нормальную работу программы при вызове с Web-сервера.

Помогут вам разобраться в этом сборник часто задаваемых вопросов по CGI и хорошая книга по CGI-программированию. Некоторые из источников перечислены в конце главы.

Вот краткий перечень проблем, часто встречающихся в CGI-программировании. При возникновении почти каждой из них выдается раздражающе-бесполезное сообщение 500 Server Error, с которым вы вскоре познакомитесь и которое возненавидите.

- Если, посылая HTML-код в браузер, вы забыли о пустой строке между HTTP-заголовком (т.е. строкой Content-type) и телом, этот HTML-код работать не будет. Помните, что перед тем, как делать все остальное, нужно ввести соответствующую строку Content-Type (и, возможно, другие HTTP-заголовки) и совершенно пустую строку.
- Серверу необходим доступ к сценарию с правом чтения и выполнения, поэтому он, как правило, должен иметь режим 0555, а лучше 0755. (Это характерно для UNIX.)
- Каталог, в котором находится сценарий, сам должен быть выполняемым, поэтому присвойте ему режим доступа 0111, а лучше 0755. (Это характерно для UNIX.)
- Сценарий должен быть установлен в соответствующем конфигурации вашего сервера каталоге. В некоторых системах, например, это может быть `/usr/etc/httpd/cgi-bin`.

* Советы по отладке в режиме командной строки приведены в документации на CGI.pm.

- Возможно, в имя файла вашего сценария понадобится вводить определенное расширение, например *cgi* или *pl*. Мы возражаем против подобной настройки WWW-сервера, предпочитая устанавливать разрешение на выполнение CGI для каждого каталога отдельно, но в некоторых конфигурациях она может быть необходима. Автоматически позволять, чтобы все заканчивающиеся на *.cgi* файлы были исполняемыми, рискованно, если FTP-клиентам разрешается производить запись во всех каталогах, а также при “зеркальном” копировании чужой структуры каталогов. В обоих случаях выполняемые программы могут внезапно появиться у вас на сервере без ведома и разрешения Web-мастера. Это означает также, что все файлы, имена которых имеют расширение *cgi* или *pl*, нельзя будет вызывать через обычный URL. Данный эффект может иметь самые разные последствия — от нежелательных до катастрофических.
- Помните: расширение *pl* означает, что это Perl-библиотека, а не исполняемый Perl-файл. Не путайте эти понятия, иначе вашей судьбе не позавидуешь. Если у вас безусловно *должно быть* уникальное расширение для разрешения выполнения Perl-программ (потому что ваша операционная система просто не настолько умна, чтобы использовать нечто вроде записи `#!/usr/bin/perl`), мы предлагаем использовать расширение *plx*. Это, однако, не избавит вас от других только что упомянутых нами проблем.
- Конфигурация вашего сервера требует особого разрешения на выполнение CGI для каталога, в который вы поместили свой CGI-сценарий. Убедитесь в том, что разрешены и GET, и POST. (Ваш Web-мастер знает, что это значит.)
- Web-сервер не выполняет ваш сценарий при запуске с вашим идентификатором пользователя. Убедитесь в том, что файлы и каталоги, к которым обращается сценарий, открыты для всех пользователей, для которых Web-сервер выполняет сценарии, таких как, например, *nobody*, *wwwuser* или *httpd*. Возможно, вам понадобится заранее создать такие файлы и каталоги и присвоить им самые широкие права доступа для записи. При работе в среде UNIX это делается посредством команды `chmod a+w`. Предоставляя широкий доступ к файлам, всегда будьте настороже.
- Всегда запускайте свой сценарий с Perl-флагом `-w`, чтобы иметь возможность получать предупреждения. Они направляются в файл регистрации ошибок Web-сервера, который содержит сообщения об ошибках и предупреждения, выдаваемые вашим сценарием. Узнайте у своего Web-мастера путь к этому файлу и проверяйте его на предмет наличия проблем. О том, как лучше обрабатывать ошибки, можно узнать и из описания стандартного модуля CGI:Carp.
- Убедитесь, что версии программ и пути к каталогам с Perl и всем используемым вами библиотекам (вроде CGI.pm) на компьютере, где работает Web-сервер, соответствуют ожидаемым.

- В начале своего сценария включите режим autoflush для дескриптора файла STDOUT, присвоив переменной \$| значение “истина”, например 1. Если вы применили модуль FileHandle или любой из модулей ввода-вывода (скажем, IO::File, IO::Socket и т.д.), то можете использовать с этим дескриптором файла метод, имя которого легко запомнить: autoflush():

```
use FileHandle;  
STDOUT->autoflush(1);
```

- Проверяйте возвращаемое значение каждого системного вызова, который производит ваша программа, и в случае неудачного исхода выполняйте соответствующее действие.

Perl и Web: не только CGI-программирование

Perl используется не только в CGI-программировании. Среди других направлений его применения — анализ файлов регистрации, управление встроенными функциями и паролями, “активными” изображениями, манипулирование изображениями*. И все это — лишь верхушка айсберга.

Специализированные издательские системы

Коммерческие издательские Web-системы могут сделать трудные вещи легкими, особенно для непрограммистов, но они не столь гибки, как настоящие языки программирования. Без исходного кода в руках вы всегда ограничены чьими-то решениями: если что-то работает не так, как вам хотелось бы, ничего сделать уже нельзя. Независимо от того, сколько великолепных программ предлагается потребителю на рынке, программист всегда будет нужен для решения тех особых задач, которые выходят за рамки стандартных требований. И, конечно, прежде всего кто-то должен писать ПО издательских систем.

Perl — идеальный язык для создания специализированных издательских систем, приспособленных под ваши уникальные потребности. С его помощью можно одним махом преобразовать необработанные данные в мириады HTML-страниц. Perl применяется для формирования и сопровождения узлов по всей World Wide Web. *The Perl Journal* (www.tpj.com) использует Perl для создания всех своих страниц. Perl Language Home Page (www.perl.com) содержит около 10000 Web-страниц, которые автоматически сопровождаются и обновляются различными Perl-программами.

* Perl-интерфейс к графической библиотеке gd Томаса Баутелла содержится в модуле GD.pm, который можно найти в CPAN.

Встроенный Perl

Самый быстрый, самый дешевый (дешевле бесплатного уже ничего быть не может) и самый популярный Web-сервер в Internet, Apache, может работать с встроенным в него Perl, используя модуль `mod_perl` из CPAN. С этим модулем Perl становится языком программирования для вашего Web-сервера. Вы можете писать маленькие Perl-программы для обработки запросов проверки полномочий, обработки сообщений об ошибках, проведения регистрации и решения любых других задач. Они не требуют запуска нового процесса, потому что Perl теперь встроен в Web-сервер. Еще более привлекателен для многих тот факт, что при работе с Apache вам не нужно запускать новый процесс всякий раз, когда поступает CGI-запрос. Вместо этого создается новый поток, который и выполняет предкомпилированную Perl-программу. Это значительно ускоряет выполнение ваших CGI-программ; обычно работа замедляется из-за вызовов `fork/exec`, а не из-за большого объема самой программы.

Другой способ ускорить выполнение CGI — использовать стандартный модуль `CGI::Fast`. В отличие от описанного выше встроенного интерпретатора Perl, такая схема выполнения CGI не требует наличия Web-сервера Apache. Подробности см. на map-странице модуля `CGI::Fast`.

Если Web-сервер у вас работает под Windows NT, вам определенно следует посетить Web-сервер ActiveWare, www.activeware.com. Там можно найти не только готовые двоичные файлы Perl для Windows-платформ*, но и PerlScript и PerlIS. Пакет PerlScript — это механизм разработки сценариев ActiveX, который позволяет встраивать Perl-код в ваши Web-страницы так, как это делается средствами JavaScript и VBScript. Пакет PerlIS — это динамически связываемая библиотека интерфейса ISAPI, которая выполняет Perl-сценарии непосредственно из IIS и других ISAPI-совместимых Web-серверов, давая значительный выигрыш в производительности.

Автоматизация работы в Web с помощью LWP

Ставили ли вы когда-нибудь перед собой задачу проверить Web-документ на предмет наличия “мертвых” ссылок, найти его название или выяснить, какие из его ссылок обновлялись с прошлого четверга? Может быть, вы хотели загрузить изображения, которые содержатся в каком-либо документе, или зеркально скопировать целый каталог? Что будет, если вам придется проходить через проху-сервер или проводить переадресацию?

* Стандартный дистрибутив Perl версии 5.004 предусматривает установку под Windows, при этом предполагается, что у вас есть компилятор C (и это предположение, как правило, оправдывается).

Сейчас вы *можли* бы сделать все это вручную с помощью броузера, но, поскольку графические интерфейсы совершенно не приспособлены для автоматизации программирования, это был бы медленный и утомительный процесс, требующий большого терпения и меньшей лени*, чем присущи большинству из нас.

Модули LWP (Library for WWW access in Perl — библиотека для доступа к WWW на Perl) из CPAN решают за вас все эти задачи — и даже больше. Например, обращение в сценарии к Web-документу с помощью этих модулей осуществляется настолько просто, что его можно выполнить с помощью одностроковой программы. Чтобы, к примеру, получить документ `/perl/index.html` с узла `www.perl.com`, введите следующую строку в свой shell или интерпретатор команд:

```
perl -MLWP::Simple -e "getprint 'http://www.perl.com/perl/index.html'"
```

За исключением модуля `LWP::Simple`, большинство модулей комплекта LWP в значительной степени объектно-ориентированы. Вот, например, крошечная программа, которая получает URL как аргументы и выдает их названия:

```
#!/usr/local/bin/perl
use LWP;
$browser = LWP::UserAgent->new(); # создать виртуальный браузер
$browser->agent("Mothra/126-Paladium:"); # дать ему имя
foreach $url (@ARGV) { # ожидать URL как аргументы
    # сделать GET-запрос по URL через виртуальный браузер
    $webdoc = $browser->request(HTTP::Request->new(GET => $url));
    if($webdoc->is_success) { # нашли
        print STDOUT "$url: :, $result->title, "\n";
    } else { # что-то не так
        print STDERR "$0: Couldn't fetch $url\n";
    }
}
```

Как видите, усилия, потраченные на изучение объектов Perl, не пропали даром. Но не забывайте, что, как и модуль `CGI.pm`, модули LWP скрывают большую часть сложной работы.

Этот сценарий работает так. Сначала создается объект — пользовательский агент (нечто вроде автоматизированного виртуального браузера). Этот объект используется для выдачи запросов на удаленные серверы. Дадим нашему виртуальному браузеру какое-нибудь глупое имя, просто чтобы сделать файлы регистрации пользователей более интересными. Затем получим удаленный документ, направив HTTP-запрос GET на удаленный сервер. Если результат успешный, выведем на экран URL и имя сервера; в противном случае немножко поплачем.

* Помните, что по Ларри Уоллу три главных достоинства программиста есть Лениость, Нетерпение и Гордость.

Вот программа, которая выводит рассортированный список уникальных ссылок и изображений, содержащихся в URL, переданных в виде аргументов командной строки.

```
#!/usr/local/bin/perl -w
use strict;
use LWP 5.000;
use URI::URL;
use HTML::LinkExor;

my($url, $browser, %saw);
$browser = LPW::UserAgent->new(); # создать виртуальный браузер
foreach $url ( @ARGV ) {
    # выбрать документ через виртуальный браузер
    my $webdoc = $browser->request(HTTP::Request->new(GET => $url));
    next unless $webdoc->is_success;
    next unless $webdoc->content_type eq 'text/html';
    # не могу разобрать GIF-файлы

    my $base = $webdoc->base;

    # теперь извлечь все ссылки типа <A ...> и <IMG ...>
    foreach (HTML::LinkExor->new->parse($webdoc->content)->eof->links){
        my($tag, %links) = @$_;
        next unless $tag eq "a" or $tag eq "img";
        my $link;
        foreach $link (values %links) {
            $saw{ url($link,$base)->abs->as_string }++;
        }
    }
}
print join("\n",sort keys %saw), "\n";
```

На первый взгляд все кажется очень сложным, но вызвано это, скорее всего, недостаточно четким пониманием того, как работают различные объекты и их методы. Мы не собираемся здесь давать пояснения по этим вопросам, потому что книга и так получилась уже достаточно объемной. К счастью, в LWP можно найти обширную документацию и примеры.

Дополнительная литература

Естественно, о модулях, ссылках, объектах и Web-программировании можно рассказать гораздо больше, чем вы узнали из этой маленькой главы. О CGI-программировании можно написать отдельную книгу — и таких книг уже написаны десятки. Приведенный ниже перечень поможет вам продолжить свои исследования в этой области.

- Файлы документации CGI.pm.
- Библиотека LWP из CPAN.
- *CGI Programming on the World Wide Web* by Shishir Gundavaram (O'Reilly & Associates).
- *Web Client Programming with Perl* by Clinton Wong (O'Reilly & Associates).
- *HTML: The Definitive Guide* by Chuck Musciano and Bill Kennedy (O'Reilly & Associates).
- *How to Setup and Maintain a Web Site* by Lincoln Stein (Addison-Wesley).
- *CGI Programming in C and Perl* by Thomas Boutell (Addison-Wesley).
- Сборник FAQ по CGI Ника Кью.
- Map-страницы: *perltoot*, *perlref*, *perlmod*, *perlobj*.

Упражнения

1. Напишите программу для создания формы, содержащей два поля ввода, которые при передаче данных формы объединяются.
2. Напишите CGI-сценарий, который определяет тип браузера, делающего запрос, и сообщает что-нибудь в ответ. (Совет: воспользуйтесь переменной среды `HTTP_USER_AGENT`.)

Ответы к упражнениям

В этом приложении даны ответы к упражнениям, помещенным в конце каждой главы.

Глава 2 “Скалярные данные”

1. Вот один из способов решения этой задачи:

```
$p1 = 3.141592654;  
$result = 2 * $p1 * 12.5;  
print "radius 12.5 is circumference $result\n";
```

Сначала мы присваиваем константу (число π) скалярной переменной `$p1`. Затем мы вычисляем длину окружности, используя значение `$p1` в выражении, и, наконец, выводим результат, применяя строку, содержащую ссылку на него.

2. Вот один из способов решения этой задачи:

```
print "What is the radius: ";  
chomp($radius = <STDIN>);  
$p1 = 3.141592654;  
$result = 2 * $p1 * $radius;  
print "radius $radius is circumference $result\n";
```

Это похоже на предыдущий пример, но здесь мы попросили пользователя, выполняющего программу (применив для выдачи приглашения оператор `print`), ввести значение. Считывание строки с терминала осуществляется посредством операции `<STDIN>`.

Если бы мы забыли применить функцию `chomp`, то получили бы посреди выведенной строки символ новой строки. Важно как можно быстрее выбросить этот символ из строки.

3. Вот один из способов решения этой задачи:

```
print "First number: "; chomp($a = <STDIN>);  
print "Second number: "; chomp($b = <STDIN>);  
$c = $a * $b; print "Answer is $c.\n";
```

Первая строка делает три вещи: приглашает вас ввести число, считывает строку со стандартного ввода, а затем избавляется от неизбежного символа новой строки. Поскольку мы используем значение `$a` строго как число, функцию `chomp` здесь можно опустить, потому что в числовом контексте `45\n` — это 45. Однако столь небрежное программирование может впоследствии обернуться неприятностями (например, если нужно будет включить `$a` в сообщение).

Вторая строка делает то же самое со вторым числом и помещает его в скалярную переменную `$b`.

Третья строка перемножает эти два числа и выводит результат. Отметьте здесь наличие символа новой строки в конце строки (тогда как в первых двух строках он отсутствует). Первые два сообщения — это приглашения, в ответ на которые пользователь должен ввести число в той же строке. Последнее сообщение — это оператор; если бы мы выбросили символ новой строки, то сразу же за сообщением появилось бы приглашение `shell`. Не очень-то хорошо.

4. Вот один из способов решения этой задачи:

```
print "String: "; $a = <STDIN>;  
print "Number of times: "; chomp($b = <STDIN>);  
$c = $a x $b; print "The result is:\n$c";
```

Как в предыдущем упражнении, первые две строки запрашивают значения двух переменных и принимают их. Однако здесь мы не выбрасываем символ новой строки, потому что он нам нужен! Третья строка получает введенные значения и выполняет над ними операцию многократного повторения строк, а затем выводит ответ. Обратите внимание на то, что за вычисляемой переменной `$c` в операторе `print` нет символа новой строки, поскольку мы считаем, что `$c` в любом случае заканчивается этим символом.

Глава 3 “Массивы и списочные данные”

1. Вот один из способов решения этой задачи:

```
print "Enter the list of strings:\n";
@list = <STDIN>;
@reverselist = reverse @list;
print @reverselist;
```

Первая строка приглашает ввести строки. Вторая строка считывает эти строки в переменную-массив. Третья строка формирует список с обратным порядком расположения элементов и заносит его в другую переменную. Последняя строка выводит результат.

Последние три строки можно объединить:

```
print "Enter the list of strings:\n";
print reverse <STDIN>;
```

Этот код работает аналогично предыдущему, так как операция `print` ожидает ввода списка, а операция `reverse` возвращает список. Далее операции `reverse` нужен список значений для реверсирования, а операция `<STDIN>`, применяемая в списочном контексте, возвращает список строк — и они получают необходимое!

2. Вот один из способов решения этой задачи:

```
print "Enter the line number: "; chomp($a = <STDIN>);
print "Enter the lines, end with ^D:\n"; @b = <STDIN>;
print "Answer: $b[$a-1]";
```

Первая строка приглашает ввести число, считывает его со стандартного ввода и удаляет назойливый символ новой строки. Вторая строка запрашивает список строк, а затем с помощью операции `<STDIN>` в списочном контексте считывает все эти строки (до появления признака конца файла) в переменную-массив. Последний оператор выводит ответ, используя для выбора соответствующей строки ссылку на массив. Обратите внимание: нам не нужно добавлять символ новой строки в конце, потому что строка, выбранная из массива `@b`, уже заканчивается таким символом.

Если вы попытаете запустить эту программу с терминала, конфигурированного самым обычным образом, вам нужно будет нажать клавиши `[Ctrl+D]`, чтобы обозначить конец файла.

3. Вот один из способов решения этой задачи:

```
srand;
print "List of strings: "; @b = <STDIN>;
print "Answer: $b[rand(@b)]";
```

Первая строка запускает генератор случайных чисел. Вторая строка считывает группу строк. Третья строка выбирает случайный элемент из этой группы и выводит его на экран.

Глава 4 “Управляющие структуры”

1. Вот один из способов решения этой задачи:

```
print "What temperature is it? ";
chomp($temperature = <STDIN>);
if ($temperature > 72) {
    print "Too hot'\n";
} else {
    print "Too cold'\n";
}
```

Первая строка приглашает ввести температуру. Вторая строка принимает введенное значение температуры. Оператор `if` в последних пяти строках выбирает для вывода одно из двух сообщений в зависимости от значения переменной `$temperature`.

2. Вот один из способов решения этой задачи:

```
print "What temperature is it? ";
chomp($temperature = <STDIN>);
if ($temperature > 75) {
    print "Too hot'\n";
} elsif ($temperature < 68) {
    print "Too cold'\n";
} else {
    print "Just right'\n";
}
```

Здесь мы модифицировали программу, введя трехвариантный выбор. Сначала температура сравнивается со значением 75, затем со значением 68. Обратите внимание: при каждом запуске программы будет выполняться только один из трех вариантов.

3. Вот один из способов решения этой задачи:

```
print "Enter a number (999 to quit): ";
chomp($n = <STDIN>);
while ($n != 999) {
    $sum += $n;
    print "Enter another number (999 to quit): ";
    chomp($n = <STDIN>);
}
print "the sum is $sum\n";
```

Первая строка приглашает ввести первое число. Вторая строка считывает это число с терминала. Цикл `while` продолжает выполняться до тех пор, пока число не станет равным 999.

Операция `+=` накапливает числа в переменной `$sum`. Обратите внимание: начальное значение этой переменной — `undef`, что очень хорошо для сумматора, потому что первое прибавляемое значение будет фактически прибавляться к нулю (помните, что при использовании в качестве числа `undef` равно нулю).

В этом цикле мы должны запрашивать и принимать еще одно число, чтобы проверка в начале цикла производилась по вновь введенному числу.

После выхода из цикла программа выводит накопленные результаты.

Если сразу же ввести 999, то значение переменной `$sum` будет равно не нулю, а пустой строке — т.е. значению `undef` в строковом контексте. Если вы хотите, чтобы программа в этом случае выводила нуль, нужно в начале программы инициализировать значение `$sum` операцией `$sum = 0`.

4. Вот один из способов решения этой задачи:

```
print "Enter some strings, end with ^D:\n";
@strings = <STDIN>;
while (@strings) {
    print pop @strings;
}
```

Сначала программа запрашивает строки. Эти строки сохраняются в переменной-массиве `@strings` (по одной на элемент).

Управляющее выражение цикла `while` — `@strings`. Это управляющее выражение ищет только одно значение (“истина” или “ложь”), поэтому вычисляет выражение в скалярном контексте. Имя массива (такое как `@strings`) при использовании в скалярном контексте представляет собой количество элементов, находящихся в массиве в текущий момент. Поскольку массив не пуст, это число не равно нулю и, следовательно, имеет значение “истина”. Эта идиома очень широко используется в Perl, она соответствует указанию “делать это, пока массив не пуст”.

Тело цикла выводит значение, полученное путем “выталкивания” крайнего справа элемента массива. Следовательно, поскольку этот элемент выводится, при каждом выполнении цикла массив становится на один элемент короче.

Возможно, вам пришла в голову мысль использовать для решения данной задачи индексы. Действительно, эту задачу можно решить несколькими способами, однако в программах настоящих Perl-хакеров индексы встречаются редко, ибо почти всегда находится лучший метод.

5. Вот один из способов решения этой задачи без использования списка:

```
for ($number = 0; $number <= 32; $number++) {  
    $square = $number * $number;  
    printf "%5g %8g\n", $number, $square;  
}
```

А вот как можно решить задачу с помощью списка:

```
foreach $number (0..32) {  
    $square = $number * $number;  
    printf "%5g %8g\n", $number, $square;  
}
```

В обоих решениях применяются циклы с использованием операторов `for` и `foreach`. Тела этих циклов идентичны, потому что в обоих решениях значение переменной `$number` при каждой итерации изменяется от 0 до 32.

В первом решении использован традиционный C-подобный оператор `for`. Первое выражение устанавливает переменную `$number` в 0, второе проверяет, меньше ли `$number`, чем 32, а третье инкрементирует `$number` при каждой итерации.

Во втором решении использован оператор `foreach`, подобный аналогичному оператору C-shell. С помощью конструктора списка создается список из 33 элементов (от 0 до 32). Затем переменной `$number` поочередно присваиваются значения, равные этим элементам.

Глава 5 “Хеши”

1. Вот один из способов решения этой задачи:

```
%map = qw(red apple green leaves blue ocean);  
print "A string please: "; chomp($some_string = <STDIN>);  
print "The value for $some_string is $map{$some_string}\n";
```

Первая строка создает хеш из требуемых пар ключ-значение. Вторая строка выбирает строку, удаляя символ новой строки. Третья строка выводит на экран введенную строку и соответствующее ей значение.

Этот хеш можно создать и с помощью серии отдельных операций присваивания:

```
$map{'red'} = 'apple';  
$map{'green'} = 'leaves';  
$map{'blue'} = 'ocean';
```

2. Вот один из способов решения этой задачи:

```
chomp(@words = <STDIN>);      # читать слова минус символы новой строки
foreach $word (@words) {
    $count{$word} = $count{$word} + 1;  # или $count{$word}++
}
foreach $word (keys %count) {
    print "$word was seen $count{$word} times\n";
}
```

Первая строка считывает строки в массив @words. Вспомните: в результате выполнения этой операции каждая строка становится отдельным элементом массива, причем символ новой строки останется нетронутым.

В следующих четырех строках осуществляется обход массива, при этом \$word приравнивается по очереди каждой строке. Функция `chomp` отсекает символ новой строки, а потом начинается волшебство. Каждое слово используется как ключ хеша. Значение элемента, выбранного по этому ключу (слову), представляет собой значение счетчика повторений данного слова до текущего момента. Сначала в хеше элементов нет, поэтому если слово `wild` встречается в первой строке, то `$count{"wild"}` будет содержать `undef`. Это значение `undef` плюс единица оказывается равным нулю плюс единица, то есть единице. (Напомним, что при использовании в качестве числа `undef` означает ноль.) При следующем проходе у нас будет единица плюс единица, или два, и т.д.

Другой распространенный способ задания этой операции инкрементирования приведен в комментарии. Опытные Perl-программисты обычно отличаются ленью (мы называем это “краткостью”) и никогда не пишут одну и ту же ссылку на хеш в обеих частях операции присваивания, если можно обойтись автоинкрементированием.

После подсчета слов в последних нескольких строках программы осуществляется просмотр хеша и поочередное получение всех его ключей. После вычисления строкового значения сам ключ и соответствующее ему значение выводятся на экран.

Есть и другое решение, отличающееся от описанного только тем, что перед словом `keys` в третьей с конца строке вставлена операция `sort`. Без проведения операции сортировки выводимый результат кажется случайным и непредсказуемым. После сортировки все упорядочивается и становится предсказуемым. (Лично я редко использую операцию `keys` без сортировки; при наличии операции `sort` непосредственно перед `keys` повторные просмотры одних и тех же или похожих данных дают сопоставимые результаты.)

Глава 6 “Базовые средства ввода-вывода”

1. Вот один из способов решения этой задачи:

```
print reverse <>;
```

Вас, может быть, удивит краткость этого ответа, но он, тем не менее, верен. Вот как работает этот механизм:

- а) Сначала функция `reverse` ищет список своих аргументов. Это значит, что операция “ромб” (`<>`) выполняется в списочном контексте. Следовательно, все строки файлов, указанных как аргументы командной строки (или данные, поступающие со стандартного ввода, если аргументов нет), считываются и преобразуются в список, каждый элемент которого состоит из одной строки.
 - б) Затем функция `reverse` меняет порядок следования элементов списка на обратный.
 - в) Наконец, функция `print` получает список-результат и выводит его
2. Вот один из способов решения этой задачи:

```
print "List of strings:\n";
chomp(@strings = <STDIN>);
foreach (@strings) {
    printf "%20s\n", $_;
}
```

Первая строка приглашает ввести список строк.

Следующая строка считывает все строки в один массив и избавляется от символов новой строки.

В цикле `foreach` осуществляется проход по этому массиву с присвоением переменной `$_` значения каждой строки.

Функция `printf` получает два аргумента. Первый аргумент определяет формат `"%20s\n"`, который означает 20-символьный столбец с выравниванием справа и символ новой строки.

3. Вот один из способов решения этой задачи:

```
print "Field width: ";
chomp($width = <STDIN>);
print "List of strings:\n";
chomp(@strings = <STDIN>);
foreach (@strings) {
    printf "%${width}s\n", $_;
}
```

В решение, данное к предыдущей задаче, мы добавили приглашение ввести ширину поля и ответ на него.

Есть еще одно изменение: строка формата `printf` теперь содержит ссылку на переменную. Значение переменной `$width` включается в эту строку до того, как `printf` использует данный формат. Отметим, что мы не можем записать эту строку как

```
printf "%$widths\n", $_; #WRONG
```

потому что тогда Perl искал бы переменную с именем `$widths`, а не переменную с именем `$width`, к которой мы прибавляем букву `s`. По-другому это можно записать так:

```
printf "%$width"."s\n", $_; # RIGHT
```

потому что символ конца строки завершает также имя переменной, защищая следующий символ от присоединения к имени.

Глава 7 “Регулярные выражения”

1. Вот несколько возможных ответов:

- а) `/a+b*/`
- б) `/***/` (Вспомним, что обратная косая черта отменяет значение следующего за ней специального символа.)
- в) `/(whatever){3}/` (Не забудьте про круглые скобки, иначе множитель будет действовать только на последний символ `whatever`; этот вариант не проходит также в том случае, если `whatever` содержит специальные символы.)
- г) `/[\\000-\\377]{5}/` или `/(.\\n){5}/` (Использовать точку без дополнительных знаков здесь нельзя, потому что она не соответствует символу новой строки.)
- д) `/(^|\\s)(\\s+)(\\s+\\2)+(\\s|$)/` (`\\s` — это не пробельный символ, а `\\2` — ссылка на все, что есть “слово”; знак `^` или альтернативный пробельный символ гарантирует, что `\\s+` начинается на границе пробельного символа.)

2. а) Вот один из способов решения этой задачи:

```
while (<STDIN>) {  
    if (/a/i && /e/i && /i/i && /o/i && /u/i) {  
        print;  
    }  
}
```

Здесь у нас приведено выражение, состоящее из пяти операций сопоставления. Все эти операции проверяют содержимое переменной `$_`, куда управляющее выражение цикла `while` помещает каждую строку. Выражение даст значение “истина” лишь в том случае, если будут найдены все пять гласных.

Обратите внимание: если любая из пяти гласных не обнаруживается, остальная часть выражения пропускается, потому что операция `&&` не вычисляет свой правый аргумент, если значение левого аргумента — “ложь”.

б) Еще один способ:

```
while (<STDIN>) {
    if (/a.*e.*i.*o.*u/i) {
        print;
    }
}
```

Этот ответ, как оказывается, проще. Здесь у нас в операторе `if` используется более простое регулярное выражение, которое ищет пять гласных в указанной последовательности, разделенных любым количеством символов.

в) Вот один из способов решения этой задачи:

```
while (<STDIN>) {
    if (/^[eiou]*a^[iou]*e^[aou]*i^[aeu]*o^[aei]*u^[aeio]*$/i) {
        print;
    }
}
```

Выглядит уродливо, но работает. Чтобы написать такое, просто подумайте: “Что может стоять между началом строки и первой буквой а?”, а затем “Что может стоять между первой буквой а и первой буквой е?”. В конце концов все решится само, от вас потребуется минимум усилий.

3. Вот один из способов решения этой задачи:

```
while (<STDIN>) {
    chomp;
    ($user, $gcos) = (split /:/)[0,4];
    ($real) = split(/,/,$gcos);
    print "$user is $real\n";
}
```

Во внешнем цикле `while` производится считывание по одной строке из файла паролей в переменную `$_`. По достижении последней строки цикл завершается.

Вторая строка тела цикла `while` означает разбиение строки на отдельные переменные с использованием в качестве разделителя двоеточия. Два из этих семи значений заносятся в отдельные скалярные переменные с имеющими смысл (мы надеемся) именами.

Поле `GCOs` (пятое поле) затем разбивается на части с использованием в качестве разделителя символа запятой, и список-результат присваивается

одной скалярной переменной, заключенной в круглые скобки. Эти скобки играют важную роль — они указывают, что операция присваивания должна быть не скалярной, а для массива. Скалярная переменная `$real` получает первый элемент списка-результата, а остальные элементы отбрасываются.

Оператор `print` затем выводит результаты на экран.

4. Вот один из способов решения этой задачи:

```
while (<STDIN>) {
    chomp;
    ($gcos) = (split /\:/)[4];
    ($real) = split /\./, $gcos;
    ($first) = split /\s+/, $real;
    $seen{$first}++;
}
foreach (keys %seen) {
    if ($seen{$_} > 1) {
        print "$_ was seen $seen{$_} times\n";
    }
}
```

Цикл `while` работает во многом так же, как цикл `while` из предыдущего упражнения. Помимо разбивки строки на поля и поля `GCOS` на реальное имя (и другие компоненты), в этом цикле осуществляется также разбиение реального имени на собственно имя и остальную часть. После определения имени элемент хеша в `%seen` инкрементируется, отмечая тот факт, что мы нашли конкретное имя. Обратите внимание: оператор `print` в этом цикле не используется.

В цикле `foreach` осуществляется проход по всем ключам хеша `%seen` (именам из файла паролей) с присваиванием каждого из них по очереди переменной `$_`. Если значение, записанное в `%seen` по данному ключу, больше 1, значит, это имя уже встречалось. Оператор `if` проверяет, так ли это, и при необходимости выводит соответствующее сообщение.

5. Вот один из способов решения этой задачи:

```
while (<STDIN>) {
    chomp;
    ($user, $gcos) = (split /\:/)[0,4];
    ($real) = split /\./, $gcos;
    ($first) = split /\s+/, $real;
    $seen{$first} .= " $user";
}
foreach (keys %names) {
    $this = $names{$_};
    if ($this =~ /\./) {
        print "$_ is used by:$this\n";
    }
}
```

Эта программа похожа на ответ к предыдущему упражнению, но вместо того чтобы просто подсчитывать, сколько раз у нас встречалось определенное имя, мы присоединяем регистрационное имя пользователя к элементу хеша `%names`, указывая имя в качестве ключа. Так, для Фреда Роджерса (регистрационное имя `mrrogers`) `$names{"Fred"}` становится равным `" mrrogers"`, а когда появится Фред Флинтстоун (регистрационное имя `fred`), `$names{"Fred"}` становится равным `" mrrogers fred"`. После завершения цикла у нас имеется список имен и список регистрационных имен всех имеющих данное имя пользователей.

В цикле `foreach`, как и в ответе к предыдущему упражнению, осуществляется проход по полученному в результате хешу, но вместо того, чтобы проверять, больше ли единицы значение элемента хеша, мы должны проверить, есть ли в этом значении более одного регистрационного имени. Для этого нужно занести значение в скалярную переменную `$this` и посмотреть, есть ли в нем после какого-нибудь символа пробел. Если есть, то одному реальному имени соответствуют несколько регистрационных имен, которые и указываются в выдаваемом в результате сообщении.

Глава 8 “Функции”

1. Вот один из способов решения этой задачи:

```
sub card {
    my %card_map;
    @card_map{1..9} = qw(
        one two three four five six seven eight nine
    );

    my($num) = @_;
    if ($card_map{$num}) {
        return $card_map{$num};
    } else {
        return $num;
    }
}

# driver routine:
while (<>) {
    chomp;
    print "card of $_ is ", &card($_), "\n";
}
```

Подпрограмма `&card` (названная так потому, что она возвращает название на английском языке для данного числа) начинается с инициализации хеша-константы, который называется `%card_map`. Значения ему присваиваются так, что, например, `$card_map{6}` равно `six`; это делает преобразование достаточно простым.

С помощью оператора `if` мы определяем, принадлежит ли значение заданному диапазону, отыскивая это число в хеше: если в хеше имеется соответствующий элемент, проверка дает значение “истина”, и данный элемент, являющийся соответствующим именем числительным, возвращается. Если соответствующего элемента нет (например, когда `$num` равно 11 или -4), то поиск в хеше возвращает значение `undef` и выполняется ветвь `else` оператора `if`, возвращая исходное число. Весь цикл, задаваемый оператором `if`, можно заменить одним выражением:

```
$card_map{$num} || $num;
```

Если значение слева от `||` истинно, то это — значение всего выражения, которое затем и возвращается. Если оно ложно (например, когда значение переменной `$num` выпадает из диапазона), то вычисляется правая часть операции `||`, возвращая значение `$num`.

Подпрограмма-драйвер последовательно получает строки, отсекает символы новой строки и передает их по одной в программу `&card`, выводя результат.

2. Вот один из способов решения этой задачи:

```
sub card { ...; } # из предыдущего ответа
print "Enter first number: ";
chomp($first = <STDIN>);
print "Enter second number: ";
chomp($second = <STDIN>);
$message = card($first) . " plus " .
    card($second) . " equals " .
    card($first+$second) . ".\n";
print "\u$message";
```

Первые два оператора `print` приглашают ввести два числа, а операторы, следующие сразу же за ними, считывают эти значения в `$first` и `$second`.

Затем путем троекратного вызова `&card` — по одному разу для каждого значения и один раз для суммы — формируется строка `$message`.

После формирования сообщения его первый символ с помощью операции `\u` переводится в верхний регистр. Затем сообщение выводится на экран.

3. Вот один из способов решения этой задачи:

```
sub card {
    my %card_map;
    @card_map{0..9} = qw(
        zero one two three four five six seven eight nine
    );

    my($num) = @_;
    my($negative);
```



```

if ($num < 0) {
    $negative = "negative ";
    $num = - $num;
}
if ($card_map{$num}) {
    return $negative . $card_map{$num};
} else {
    return $negative . $num;
}
}

```

Здесь мы объявили массив `%card_map`, чтобы обнулить его значения.

Первый оператор `if` инвертирует знак переменной `$num` и присваивает переменной `$negative` в качестве значения слово `negative`, если задаваемое в качестве аргумента число меньше нуля. После действия оператора `if` значение `$num` всегда неотрицательное, но при этом в переменную `$negative` записывается строка `negative`, которая в дальнейшем используется как префикс.

Второй оператор `if` определяет, находится ли значение переменной `$num` (теперь положительное) в хеше. Если да, то полученное в результате значение хеша присоединяется к префиксу, который хранится в `$negative`, и возвращается. Если нет, то значение, содержащееся в `$negative`, присоединяется к исходному числу.

Последний оператор `if` можно заменить выражением:

```
$negative . ($card_map{$num} || $num);
```

Глава 9 “Разнообразные управляющие структуры”

1. Вот один из способов решения этой задачи:

```

sub card {} # из предыдущего упражнения

while () { ## НОВОЕ ##
    print "Enter first number: ";
    chomp($first = <STDIN>);
    last if $first eq "end"; ## НОВОЕ ##

    print "Enter second number: ";
    chomp($second = <STDIN>);
    last if $second eq "end"; ## НОВОЕ ##

    $message = &card($first) . " plus " .
        card($second) . " equals " .
        card($first+$second) . ".\n";
    print "\u$message";
} ## НОВОЕ ##

```

Обратите внимание на появление цикла `while` и двух операций `last`. Вот так-то!

Глава 10 “Дескрипторы файлов и проверка файлов”

1. Вот один из способов решения этой задачи:

```
print "What file? ";
chomp($filename = <STDIN>);
open(THATFILE, "$filename") ||
    die "cannot open $filename: $!";
while (<THATFILE>) {
    print "$filename: $_"; # предполагается, что $_ заканчивается \n
}
```

В первых двух строках дается приглашение ввести имя файла, который затем открывается с дескриптором `THATFILE`. Содержимое этого файла считывается с помощью дескриптора и выводится в `STDOUT`.

2. Вот один из способов решения этой задачи:

```
print "Input file name: ";
chomp($infilename = <STDIN>);
print "Output file name: ";
chomp($outfilename = <STDIN>);
print "Search string: ";
chomp($search = <STDIN>);
print "Replacement string: ";
chomp($replace = <STDIN>);
open(IN, $infilename) ||
    die "cannot open $infilename for reading: $!";
## необязательная проверка существования файла
## $outfilename
die "will not overwrite $outfilename" if -e $outfilename;
open(OUT, "$outfilename") ||
    die "cannot create $outfilename: $!";
while (<IN>) {    # читать строку из файла IN в $_
    s/$search/$replace/g; # change the lines
    print OUT $_; # вывести эту строку в файл OUT
}
close(IN);
close(OUT);
```

Эта программа основана на программе копирования файлов, описанной выше в этой главе. К новым особенностям здесь относятся приглашение вводить строки и команда подстановки в середине цикла `while`, а также проверка возможности уничтожения уже существующего файла.

Обратите внимание на то, что обратные ссылки в регулярном выражении работают, а вот обращение к памяти в заменяющей строке — нет.

3. Вот один из способов решения этой задачи:

```
while (<>) {
    chomp; # удалить символ новой строки
    print "$_ is readable\n" if -r;
    print "$_ is writable\n" if -w;
    print "$_ is executable\n" if -x;
    print "$_ does not exist\n" unless -e;
}
```

При каждом выполнении цикла `while` читается имя файла. После удаления символа новой строки файл проверяется (с помощью остальных операторов) на наличие различных прав доступа.

4 Вот один из способов решения этой задачи:

```
while (< >) {
    chomp;
    $age = -M;
    if ($oldest_age < $age) {
        $oldest_name = $_;
        $oldest_age = $age;
    }
}

print "The oldest file is $oldest_name ",
      "and is $oldest age days old.\n";
```

Сначала мы выполняем цикл для каждого считываемого имени файла. Символ новой строки отбрасывается, а затем с помощью операции `-m` вычисляется возраст файла в днях. Если возраст превышает возраст самого старого из файлов, которые мы до сих пор видели, мы запоминаем имя файла и его возраст. Первоначально `$oldest_age = 0`, поэтому мы рассчитываем на то, что имеется хотя бы один файл, возраст которого больше 0 дней.

По завершении цикла оператор `print` выдает отчет.

Глава 11 “Форматы”

1. Вот один из способов решения этой задачи:

[illegible]

Первая строка открывает файл паролей. В цикле `while` этот файл обрабатывается построчно. Для того чтобы можно было загрузить скалярные переменные, каждая строка разбивается на части; в качестве разделителя используется двоеточие. Реальное имя пользователя выбирается из поля `GCOS`. Последний оператор цикла `while` вызывает функцию `write` для вывода всех данных.

Формат дескриптора файла `STDOUT` определяет простую строку с тремя полями. Их значения берутся из трех скалярных переменных, значения которым присваиваются в цикле `while`.

2. Вот один из способов решения этой задачи:

```
# прибавить к программе из первой задачи...
format STDOUT_TOP =
Username User ID Real Name
=====
.
```

Все, что нужно для добавления к предыдущей программе заголовков страниц,— это добавить формат начала страницы. Указанным выражением мы помещаем заголовки в столбцы.

Чтобы выровнять столбцы, мы скопировали текст формата `STDOUT` и, используя в нашем текстовом редакторе режим замены, заменили поля `@<<<` линиями `====`. Это можно сделать благодаря существованию по-символьного соответствия между форматом и получаемым результатом.

3. Вот один из способов решения этой задачи:

```
# прибавить к программе из первой задачи...
format STDOUT_TOP =
Page @<<<
$%

Username User ID Real Name
=====
.
```

Здесь для получения заголовков страниц мы опять-таки ввели формат начала страницы. Этот формат содержит также ссылку на переменную `$%`, которая автоматически присваивает странице номер.

Глава 12 “Доступ к каталогам”

1. Вот один из способов решения этой задачи:

```
print "Where to? ";
chomp($newdir = <STDIN>);
chdir($newdir) || die "Cannot chdir to $newdir: $!";
```

```
foreach (<*>) {
    print "$_\n";
}
```

В первых двух строках запрашивается и считывается имя каталога.

С помощью третьей строки мы пытаемся перейти в каталог с указанным именем. В случае неудачи программа аварийно завершается.

В цикле `foreach` осуществляется проход по списку. Но что это за список? Это разворачивание в списочном контексте, в результате которого мы получим список всех имен файлов, совпадающих с образцом (в данном случае он задан как `*`).

- Вот один из способов решения этой задачи — с помощью дескриптора каталога:

```
print "Where to? ";
chomp($newdir = <STDIN>);
chdir($newdir) ||
    die "Cannot chdir to $newdir: $!";
opendir(DOT, ".") ||
    die "Cannot opendir . (serious dainbramage): $!";
foreach (sort readdir(DOT)) {
    print "$_\n";
}
closedir(DOT);
```

Так же, как в предыдущей программе, мы запрашиваем новый каталог. Перейдя в него посредством команды `chdir`, мы открываем каталог, создавая дескриптор каталога `DOT`. В цикле `foreach` список, возвращенный функцией `readdir` (в списочном контексте) сортируется, а затем просматривается с присваиванием переменной `$_` значения каждого элемента.

А вот как сделать это с помощью разворачивания:

```
print "Where to? ";
chomp($newdir = <STDIN>);
chdir($newdir) || die "Cannot chdir to $newdir: $!";
foreach (sort <*.*>) {
    print "$_\n";
}
```

Да, это, по сути дела, еще одна программа из предыдущего упражнения, но мы вставили перед операцией разворачивания операцию `sort` и дополнили образец символами `.*`, чтобы найти файлы, имена которых начинаются с точки. Операция `sort` нам нужна по той причине, что файл `!fred` должен стоять перед точечными файлами, а `barney` — после них, но простого способа, позволяющего расставить их в таком порядке при помощи `shell`, нет.

Глава 13 “Манипулирование файлами и каталогами”

1. Вот один из способов решения этой задачи:

```
unlink @ARGV;
```

Да, именно так. Массив @ARGV — это список имен, подлежащих удалению. Операция `unlink` получает список имен, поэтому нам нужно лишь соединить два этих компонента, и дело сделано.

Конечно, здесь не реализован ни механизм уведомления об ошибках, ни опции `-f` и `-i`, ни другие подобные вещи, но это было бы уже слишком серьезно. Если вы это сделали — отлично!

2. Вот один из способов решения этой задачи:

```
($old, $new) = @ARGV; # назвать их
if (-d $new) { # новое имя — каталог, его нужно откорректировать
    ($basename = $old) =~ s#.*##s; # получить название собственно
                                # каталога $old
    $new .= "/$basename"; # и добавить его к новому имени
}
rename($old,$new) || die "Cannot rename $old to $new: $!";
```

Рабочая лошадка в этой программе — последняя строка, но все остальное тоже нужно на тот случай, если новое имя принадлежит каталогу.

Сначала мы даем наглядные имена двум элементам массива @ARGV. Затем, если имя \$new — каталог, нам нужно откорректировать его, добавив в конец нового имени собственно имя каталога \$old. Это значит, что переименование `/usr/src/fred` в `/etc` фактически приводит к переименованию `/usr/src/fred` в `/etc/fred`.

Наконец, после добавления собственно имени каталога мы завершаем задачу вызовом `rename`.

3. Вот один из способов решения этой задачи:

```
($old, $new) = @ARGV; # назвать их
if (-d $new) { # новое имя — каталог, его нужно откорректировать
    ($basename = $old) =~ s#.*##s; # получить название собственно
                                # каталога $old
    $new .= "/$basename"; # и добавить его к новому имени
}
link($old,$new) || die "Cannot link $old to $new: $!";
```

Эта программа идентична предыдущей программе за исключением самой последней строки, потому что мы создаем ссылку, а не выполняем переименование.

4. Вот один из способов решения этой задачи:

```
if ($ARGV[0] eq "-s") { # нужна символическая ссылка
    $symlink++; # запомнить
    shift(@ARGV); # и отбросить флаг -s
}
($old, $new) = @ARGV; # назвать их
if (-d $new) { # новое имя — каталог, его нужно откорректировать
    ($basename = $old) =~ s#.*/##s; # получить название собственно
    # каталога $old
    $new .= "/$basename"; # и добавить его к новому имени
}
if ($symlink) { # wants a symlink
    symlink($old, $new);
} else { # нужна жесткая ссылка
    link($old, $new);
}
```

Средняя часть этой программы — такая же, как в предыдущих двух упражнениях. Новые здесь — несколько первых и несколько последних строк.

В первых строках осуществляется проверка первого аргумента программы. Если этот аргумент равен `-s`, то скалярная переменная `$symlink` инкрементируется, получая в результате значение 1. Затем выполняется сдвиг массива `@ARGV`, в результате чего удаляется флаг `-s`. Если флаг `-s` отсутствует, то ничего не делается и `$symlink` остается равной `undef`. Сдвиг массива `@ARGV` выполняется достаточно часто, поэтому имя массива `@ARGV` является аргументом функции `shift` по умолчанию; то есть вместо

```
shift(@ARGV);
```

мы могли бы написать

```
shift;
```

Последние несколько строк проверяют значение `$symlink`. Оно будет равно либо 1, либо `undef`, и на основании этого для файлов выполняется либо операция `symlink`, либо `link`.

5. Вот один из способов решения этой задачи:

```
foreach $f (<*>) {
    print "$f -> $where\n" if defined($where = readlink($f));
}
```

Скалярной переменной `$f` присваивается по очереди каждое из имен файлов текущего каталога. Для каждого имени переменной `$where` присваивается значение, полученное в результате выполнения функции `readlink()` для данного имени. Если имя — не символическая ссылка, то операция `readlink` возвращает `undef`, давая значение “ложь” в

проверке `if`, а `print` пропускается. Если же операция `readlink` возвращает какое-то значение, то `print` выводит название символической ссылки и имя файла или директории, на которую она указывает.

Глава 14 “Управление процессами”

1 Вот один из способов решения этой задачи:

```
if ( date =~ /^S/) {
    print "Go play'\n";
} else {
    print "Get to work'\n";
}
```

Оказывается, команда `date` выводит букву `S` в качестве первого символа только по выходным (`Sat` или `Sun`), что делает задачу тривиальной. Мы вызываем `date`, затем с помощью регулярного выражения смотрим, является ли первый символ буквой `S`. На основании результата мы выводим одно сообщение или другое.

2 Вот один из способов решения этой задачи:

```
open(PW, "/etc/passwd");
while (<PW>) {
    chomp,
    ($user,$gcos) = (split /:/)[0,4];
    ($real) = split(/,/, $gcos);
    $real{$user} = $real;
}
close(PW),

open(WHO, "who|") || die "cannot open who pipe";
while (<WHO>) {
    ($login, $rest) = /^(\\S+)\s+(.*)/;
    $login = $real{$login} if $real{$login};
    printf "%-30s %s\n", $login, $rest,
}
```

В первом цикле создается хеш `%real`, ключи которого — регистрационные имена, а значения — соответствующие реальные имена. Этот хеш используется в следующем цикле для замены регистрационного имени на реальное.

Во втором цикле осуществляется просмотр результата выполнения команды `who`, которая вызвана при помощи дескриптора файла. Каждая строка полученного в результате выражения разбивается путем сопоставления с регулярным выражением в списочном контексте. Первое слово строки (регистрационное имя) заменяется реальным именем из хеша, но только если оно существует. После всего этого с помощью функции `printf` результат помещается в `STDOUT`.

Операции открытия дескриптора файла и начала цикла можно заменить строкой

```
foreach $_ ( who ) {
```

с тем же результатом. Единственное различие состоит в том, что вариант программы с использованием дескриптора файла может приступить к работе, как только *who* начнет выдавать символы, тогда как в варианте с функцией *who* в обратных кавычках придется ждать завершения выполнения функции *who*.

3. Вот один из способов решения этой задачи:

```
open(PW,"/etc/passwd");
while (<PW>) {
    chomp;
    ($user,$gcos) = (split /:/){0,4};
    ($real) = split(/,/,$gcos);
    $real{$user} = $real;
}
close(PW),

open(LPR,"|lpr") || die "cannot open LPR pipe";
open(WHO,"who|") || die "cannot open who pipe";
while (<WHO>) {
# или заменить предыдущие две строки на foreach $_ (`who`) {
($login, $rest) = /^(S+)\s+(.*)/;
    $login = $real{$login} if $real{$login};
    printf LPR "%-30s %s\n",$login,$rest;
}
```

Разница между этой программой и программой из предыдущего упражнения состоит в том, что мы добавили дескриптор файла LPR, открытый для процесса *lpr*, и модифицировали оператор *printf* так, чтобы он посылал данные не в STDOUT, а в этот дескриптор.

4. Вот один из способов решения этой задачи:

```
sub mkdir {
    'system "/bin/mkdir", @_';
}
```

Здесь команда *mkdir* получает аргументы прямо из аргументов подпрограммы. Однако возвращаемое значение должно подвергнуться операции логического отрицания, потому что ненулевой код выхода из *system* должен быть преобразован в значение “ложь” для вызывающей Perl-программы.

5. Вот один из способов решения этой задачи:

```
sub mkdir {
    my($dir, $mode) = @_;
    ('system "/bin/mkdir", $dir) && chmod($mode, $dir),
}
```

Сначала мы описываем локальные аргументы этой подпрограммы — `$dir` и `$mode`. Затем мы вызываем `mkdir` для каталога с именем `$dir`. В случае успеха операция `chmod` присваивает этому каталогу соответствующие права доступа.

Глава 15 “Другие операции преобразования данных”

1. Вот один из способов решения этой задачи:

```
while (<>) {
    chomp;
    $slash = rindex($_, "/");
    if ($slash > -1) {
        $head = substr($_, 0, $slash);
        $tail = substr($_, $slash+1);
    } else {
        ($head, $tail) = ("", $_);
    }
    print "head = '$head', tail = '$tail'\n";
}
```

Каждая строка, прочитанная операцией “ромб”, сначала пропускается через операцию `chomp`, которая удаляет символ новой строки. Затем с помощью `rindex()` мы ищем в этой строке крайнюю справа косую черту. Следующие две строки разбивают строку на части, используя `substr()`. Если косой черты нет, то результат `rindex` равен `-1`, и этот вариант мы не рассматриваем. Последняя строка цикла выводит результат.

2. Вот один из способов решения данной задачи:

```
chomp(@nums = <STDIN>); # обратите внимание на особый случай
                        # использования chomp
@nums = sort { $a <=> $b } @nums;
foreach (@nums) {
    printf "%30g\n", $_;
}
```

В первой строке в массив `@nums` вводятся все числа. Во второй строке этот массив сортируется в числовом порядке, для чего используется встроенный оператор. Цикл `foreach` обеспечивает вывод результатов.

3. Вот один из способов решения этой задачи:

```
open(PW, "/etc/passwd") || die "How did you get logged in?";
while (<PW>) {
    chomp;
    ($user, $gcos) = (split /\:/)[0,4];
    ($real) = split /\:/, $gcos;
    $real{$user} = $real;
    ($last) = (split /\s+/, $real)[-1];
```

```

    $last{$user} = "\L$last";
}
close(PW);

for (sort by_last keys %last) {
    printf "%30s %8s\n", $real{$_}, $_;
}

sub by_last { ($last{$a} cmp $last{$b}) || ($a cmp $b) }

```

В первом цикле создается хеш `%last`, ключи которого — регистрационные имена, а соответствующие им значения — фамилии пользователей, и хеш `%real`, содержащий полные реальные имена пользователей. Все символы переводятся в нижний регистр, чтобы, к примеру, FLINTSTONE, Flintstone и flintstone стояли рядом друг с другом.

Во втором цикле выводится `%real`, упорядоченный по значениям `%last`. Это делается с использованием определения сортировки, предоставленного подпрограммой `by_last`.

4. Вот один из способов решения этой задачи:

```

while (<>) {
    substr($_,0,1) =~ tr/a-z/A-Z/;
    substr($_,1) =~ tr/A-Z/a-z/;
    print;
}

```

Для каждой строки, прочитанной операцией “ромб”, мы используем две операции `tr` — по одной для разных частей строки. Первая операция `tr` переводит первый символ строки в верхний регистр, а вторая переводит все остальные символы в нижний регистр. Результат выводится.

Вот другое решение, с использованием только строковых операций с двойными кавычками:

```

while (<>) {
    print "\u\L$_";
}

```

Если вы самостоятельно нашли это решение, поставьте себе дополнительно пять баллов.

Глава 16 “Доступ к системным базам данных”

1. Вот один из способов решения этой задачи:

```

$_ = " ";
while (@pw = getpwent) {

```


Первая строка открывает DBM псевдонимов. (В вашей системе DBM псевдонимов может находиться в каталоге `/usr/lib/aliases` — попробуйте этот вариант, если наш не сработает.) В цикле `while` осуществляется проход по DBM-массиву. Первая строка цикла служит для удаления символа NUL, которым завершались ключ и значение. Последняя строка цикла обеспечивает вывод результата.

2. Вот один из способов решения этой задачи:

```
# program 1:
dbmopen(%WORDS, "words", 0644);
while (<>) {
    foreach $word (split(/\W+/)) {
        $WORDS{$word}++;
    }
}
dbmclose(%WORDS);
```

Первая программа (записывающая) открывает DBM `words` в текущем каталоге, создавая файлы `words.dir` и `words.pag`. Цикл `while` получает каждую строку, используя операцию “ромб”. Эта строка разбивается с помощью операции `split` и разделителя `/\W+/`, который обозначает нетекстовые символы. Затем производится подсчет всех слов, имеющих в DBM-массиве, причем для осуществления прохода в цикле по всем словам используется оператор `foreach`.

```
# program 2:
dbmopen(%WORDS, "words", undef);
foreach $word (sort { $WORDS{$b} <=> $WORDS{$a} } keys %WORDS) {
    print "$word $WORDS{$word}\n";
}
dbmclose(%WORDS);
```

Вторая программа также открывает DBM `words` в текущем каталоге. Сложная на первый взгляд строка `foreach` делает почти всю “грязную” работу. При каждом выполнении цикла переменной `$word` в качестве значения будет присваиваться следующий элемент списка. Этот список состоит из ключей хеша `%WORDS`, рассортированных по их значениям (т.е. количеству повторений) в убывающем порядке. Для каждого элемента списка мы выводим слово и количество его повторений.

Глава 18 “Преобразование других программ в Perl-программы”

1. Вот один из способов решения этой задачи:

```
for (;;) {
    ($user, $home) = (getpwent){0,7};
    last unless $user;
    next unless open(N, "$home/.newsrc");
```

```

next unless -M N < 30; ## added value :-)
while (<N>) {
    if (/^comp\.lang\.perl\.announce:/) {
        print "$user is a good person, ",
            "and reads comp.lang.perl.announce'\n";
        last;
    }
}
}

```

Самый внешний цикл — это цикл `for`, который выполняется “вечно”, выход из этого цикла осуществляется посредством операции `last`, стоящей внутри него. При каждом выполнении цикла операция `getpwent` выбирает новое значение переменной `$user` (имя пользователя) и переменной `$home` (его начальный каталог).

Если значение `$user` пусто, осуществляется выход из цикла `for`. Следующие две строки ищут последний файл `.newsrsc` в начальном каталоге пользователя. Если этот файл открыть нельзя или если он изменялся слишком давно, запускается следующая итерация цикла `for`.

В цикле `while` осуществляется чтение по одной строке из файла `.newsrsc`. Если строка начинается с `comp.lang.perl.announce:`, то оператор `print` сообщает об этом, и осуществляется преждевременный выход из цикла `while`.

Глава 19 “CGI-программирование”

1. Вот один из способов решения этой задачи:

```

use strict;
use CGI qw(:standard);

print header(), start_html("Add Me");
print h1("Add Me");
if(param()) {
    my $n1 = param('field1');
    my $n2 = param('field2');
    my $n3 = $n2 + $n1;
    print p("$n1 + $n2 = <strong>$n3</strong>\n");
} else {
    print hr(), start_form();
    print p("First Number:", textfield("field1"));
    print p("Second Number:", textfield("field2"));
    print p(submit("add"), reset("clear"));
    print end_form(), hr();
}
print end_html();

```

Если входных данных нет, то просто создается форма с двумя текстовыми полями (при помощи метода `textfield`). При наличии входной информации мы объединяем эти поля и выводим результат.

2. Вот один из способов решения этой задачи:

```
use strict;
use CGI qw(:standard);

print header(), start_html("Browser Detective");
print h1("Browser Detective"), hr();
my $browser = $ENV{'HTTP_USER_AGENT'};
$_ = $browser;

BROWSER:{
    if (/msie/1) {
        msie($_);
    } elsif (/mozilla/1) {
        netscape($_);
    } elsif (/lynx/1) {
        lynx($_);
    } else {
        default($_);
    }
}

print end_html();

sub msie{
    print p("Internet Explorer: @_. Good Choice\n");
}

sub netscape {
    print p("Netscape: @_. Good Choice\n");
}

sub lynx {
    print p("Lynx: @_. Shudder...");
}

sub default {
    print p("What the heck is a @_?");
}
```

Главный момент здесь — проверка переменной среды `HTTP_USER_AGENT` на предмет наличия одного из значений, определяющих известный тип броузера (MS Internet Explorer, Netscape Navigator, Lynx). Такая операция проводится не на всех серверах, но на большинстве из них. Это хороший способ генерировать содержимое возвращаемого документа, ориентированное на возможности конкретного броузера. Обратите внимание: чтобы посмотреть, какой именно броузер применяется пользователем, мы выполняем только тривиальное строковое сопоставление (без учета регистра).

Библиотеки и модули

Для простых программ вы уже теперь можете свободно писать собственные Perl-подпрограммы. Когда же задачи, для решения которых вы применяете Perl, станут более сложными, вам иногда будет приходить в голову мысль: “Кто-то, должно быть, это уже делал”. И в подавляющем большинстве случаев вы окажетесь правы.

Действительно, другие люди уже написали коды для решения большинства распространенных задач. Более того, они поместили их либо в стандартный дистрибутив Perl, либо в бесплатно загружаемый архив CPAN. Чтобы использовать этот код (и сэкономить немного времени), вам придется разобраться в том, как пользоваться Perl-библиотекой. Этот вопрос вкратце освещался в главе 19.

Одно из преимуществ использования модулей из стандартного дистрибутива состоит в том, что потом вы можете предоставлять свою программу другим пользователям, при этом не придется предпринимать никаких специальных мер. Это объясняется тем, что одна и та же стандартная библиотека доступна Perl-программам практически везде.

Если вы решите обратиться к стандартной библиотеке, то в конечном итоге сэкономите свое время. Нет никакого смысла вновь изобретать велосипед. Следует понимать, однако, что эта библиотека содержит очень много материала. Одни модули могут быть исключительно полезны, тогда как другие совершенно не подходят для решения ваших задач. Например, некоторые модули полезны лишь в том случае, если вы создаете дополнения к языку Perl.

Чтобы прочитать документацию, относящуюся к стандартному модулю, воспользуйтесь программой `man` или `perldoc` (если они у вас есть) либо своим Web-браузером, если речь идет о HTML-версиях этой документации. Если ничего не получается, поищите в файлах самого модуля: документация включена в состав каждого модуля (в `pod`-формате). Чтобы найти модуль у себя в системе, попробуйте выполнить из командной строки следующую Perl-программу:


```
# для (большинства) Unix-подобных shell
perl -e 'print "@INC\n"'
# для (некоторых) других интерпретаторов команд
perl -e "print join(' ', "@INC"),\n"
Вы должны найти модуль в одном из каталогов, перечисленных этой командой.
```

Терминология

Перед тем как дать перечень всех стандартных модулей, давайте разберемся в терминах.

Пакет

Пакет — это простое устройство управления пространством имен, позволяющее в каждой из двух разных частей Perl-программы иметь свою переменную с именем `$fred`. Этими пространствами имен управляет объявление *package*, описанное в главе 5 книги *Programming Perl*.

Библиотека

Библиотека — это набор подпрограмм определенного назначения. Часто библиотека объявляет себя отдельным пакетом; это позволяет держать в одном месте соответствующие переменные и подпрограммы, чтобы они не мешали другим переменным в вашей программе. Как правило, библиотека старого стиля размещалась в отдельном файле, часто под именем с расширением `pl`. Библиотечные программы включались в основную программу посредством функции *require*. Не так давно этот подход был заменен использованием *модулей* (см. следующий абзац), и термин *библиотека* теперь часто обозначает всю систему модулей, которые поставляются с Perl.

Модуль

Модуль — это библиотека, соответствующая конкретным соглашениям, которая позволяет включать библиотечные подпрограммы в основную программу во время компиляции с помощью директивы *use*. Имена файлов модулей имеют расширение `pm`, потому что это необходимо для корректного использования директивы *use*. Подробно Perl-модули описаны в главе 5 книги *Programming Perl*.

Прага

Прага — это модуль, который воздействует не только на фазу выполнения программы, но и на фазу ее компиляции. Считайте, что прага содержит подсказки компилятору. В отличие от других модулей, праги часто (но не всегда) ограничивают сферу своего влияния самым внутренним охватывающим блоком вашей программы (т.е. блоком, охватывающим вызов праги). По соглашению имена праг состоят из символов нижнего регистра.

Стандартные модули

Ниже приведен перечень всех Perl-прагм и модулей, входящих в текущий дистрибутив языка (версия 5.004). Классификация модулей произвольная.

Таблица Б.1. Общее программирование: разное

Модуль	Функция
autouse	Задерживает загрузку модуля до его использования
constant	Создает константы периода компиляции
Benchmark	Проверяет и сравнивает временные параметры выполнения кода
Config	Позволяет получить информацию о конфигурации Perl
Env	Импортирует переменные среды
English	Для пунктуационных переменных использует английские имена или имена на языке <i>awk</i>
FindBin	Находит путь к выполняемой в данный момент программе
Getopt::Long	Осуществляет расширенную обработку опций командной строки
Getopt::Std	Обрабатывает односимвольные ключи и осуществляет их кластеризацию
lib	Манипулирует массивом @INC во время компиляции
Shell	Запускает команды shell прозрачно для Perl
strict	Ограничивает использование небезопасных конструкций
Symbol	Генерирует анонимное разворачивание (glob); уточняет имена переменных
subs	Предопределяет имена подпрограмм
vars	Предопределяет имена глобальных переменных

Таблица Б.2. Общее программирование: обработка ошибок и регистрация

Модуль	Функция
Carp	Выдает сообщения об ошибках
diagnostics	Включает режим диагностики с выдачей предупреждений
sigtrap	Разрешает обратное прослеживание стека для неожиданных сигналов
Sys::Syslog	Perl-интерфейс к UNIX-вызовам <i>syslog(3)</i>

Таблица Б.3. Общее программирование: доступ к файлам и их обработка

Модуль	Функция
Cwd	Получает путевое имя текущего рабочего каталога
DirHandle	Выдает методы объектов для работы с дескрипторами каталогов
Fcntl	Загружает C-определения <i>Fcntl.h</i>
File::Basename	Разбирает спецификации файлов
File::CheckTree	Выполняет всевозможные проверки для набора файлов
File::Copy	Копирует файлы или дескрипторы файлов
File::Find	Обеспечивает просмотр дерева файлов
File::Path	Создает и удаляет ряд каталогов
FileCache	Позволяет одновременно открывать больше файлов, чем разрешает система
FileHandle	Выдает методы объектов для работы с дескрипторами файлов
SelectSaver	Сохраняет и восстанавливает выбранный дескриптор файла

Таблица Б.4. Общее программирование: классы для операций ввода-вывода

Модуль	Функция
IO	Интерфейс верхнего уровня к классам IO::*
IO::File	Методы объектов для работы с дескрипторами файлов
IO::Handle	Методы объектов для дескрипторов ввода-вывода
IO::Pipe	Методы объектов для каналов
IO::Seekable	Методы для объектов ввода-вывода на базе поиска
IO::Select	Объектный интерфейс для выбора
IO::Socket	Объектный интерфейс для портов

Таблица Б.5. Общее программирование: обработка текста и экранные интерфейсы

Модуль	Функция
locale	Использует локализацию POSIX для встроенных операций
Pod::HTML	Конвертирует pod-данные в HTML
Pod::Text	Конвертирует pod-данные в форматированный ASCII-текст
Search::Dict	Ищет ключ в файле словаря
Term::Cap	Интерфейс termcap

Модуль	Функция
Term::Complete	Модуль завершения слов
Text::Abbrev	Создает из списка таблицу сокращений
Text::ParseWords	Разбирает текст на лексемы и создает из них массив
Text::Soundex	Реализует алгоритм Soundex, разработанный Кнудом
Text::Tabs	Раскрывает и сворачивает знаки табуляции
Text::Wrap	Выделяет текст в абзац

Таблица Б.6. Интерфейсы к базам данных

Модуль	Функция
AnyDBM_File	Создает основу для множества DBM
DB_File	Доступ к Berkeley DB
GDBM_File	Связанный доступ к библиотеке GDBM
NDBM_File	Связанный доступ к файлам NDBM
ODBM_File	Связанный доступ к файлам ODBM
SDBM_File	Связанный доступ к файлам SDBM

Таблица Б.7. Математика

Модуль	Функция
Integer	Выполняет арифметические операции в целочисленном формате, а не в формате с двойной точностью
Math::BigFloat	Пакет математических операций для чисел с плавающей запятой произвольной длины
Math::BigInt	Пакет математических операций для целых чисел произвольной длины
Math::Complex	Пакет для комплексных чисел

Таблица Б.8. World Wide Web

Модуль	Функция
CGI	Интерфейс Web-сервера (Common Gateway Interface)
CGI::Apache	Поддержка Perl-модуля сервера Apache
CGI::Carp	Ошибки сервера регистрации с полезной информацией
CGI::Fast	Поддержка FastCGI (устойчивый серверный процесс)
CGI::Push	Поддержка “выталкивания” со стороны сервера
CGI::Switch	Простой интерфейс для многих типов серверов

Таблица Б.9. Сети и межпроцессное взаимодействие

Модуль	Функция
IPC::Open2	Открывает процесс для чтения и записи
IPC::Open3	Открывает процесс для чтения, записи и обработки ошибок
Net::Ping	Проверяет, есть ли данный хост в сети
Socket	Загружает С-определения <i>socket.h</i> и манипуляторы структур
Sys::Hostname	Пытается получить хост-имя всеми возможными способами

Таблица Б.10. Автоматизированный доступ к Comprehensive Perl Archive Network

Модуль	Функция
CPAN	Простой интерфейс к CPAN
CPAN::FirstTime	Утилита для создания файла конфигурации CPAN
CPAN::Nox	Запускает CPAN, избегая компилированных расширений

Таблица Б.11. Время и локализация

Модуль	Функция
Time::Local	Эффективно определяет местное и среднее гринвичское время
I18N::Collate	Сравнивает восьмибитовые скалярные данные

Таблица Б.12. Объектные интерфейсы к встроенным функциям

Модуль	Функция
Class::Struct	Объявляет struct-подобные типы данных как Perl-классы
File::stat	Объектный интерфейс к функции stat
Net::hostent	Объектный интерфейс к функциям gethost*
Net::netent	Объектный интерфейс к функциям getnet*
Net::protoent	Объектный интерфейс к функциям getproto*
Net::servent	Объектный интерфейс к функциям getserv*
Time::gmtime	Объектный интерфейс к функции gmtime
Time::localtime	Объектный интерфейс к функции localtime
Time::tm	Внутренний объект для Time::{gm,local}time
User::grent	Объектный интерфейс к функциям getgr*
User::pwent	Объектный интерфейс к функциям getpw*

Таблица Б.13. Для разработчиков: автозагрузка и динамическая загрузка

Модуль	Функция
AutoLoader	Загружает функции только по требованию
AutoSplit	Разбивает пакет для автозагрузки
Devel::SelfStubber	Генерирует заглушки для модуля SelfLoading
DynaLoader	Автоматическая динамическая загрузка Perl-модулей
SelfLoader	Загружает функции только по требованию

Таблица Б.14. Для разработчиков: расширения языка и поддержка разработки платформ

Модуль	Функция
blib	Определяет структуру каталогов <i>blib</i> во время построения модулей
ExtUtils::Embed	Утилиты для встраивания Perl в С-программы
ExtUtils::Install	Инсталлирует файлы
ExtUtils::Liblist	Определяет библиотеки для использования и порядок их использования
ExtUtils::MakeMaker	Создает <i>Makefile</i> для расширения Perl
ExtUtils::Manifest	Утилиты для написания и проверки файла <i>MANIFEST</i>
ExtUtils::Miniperl	Создает С-код для <i>perlmain.c</i>
ExtUtils::Mkbootstrap	Создает файл самозагрузки для использования модулем DynaLoader
ExtUtils::Mksymlists	Пишет файлы опций компоновщика для динамического расширения
ExtUtils::MM_OS2	Методы для отмены UNIX-режима в ExtUtils::MakeMaker
ExtUtils::MM_Unix	Методы, используемые модулем ExtUtils::MakeMaker
ExtUtils::MM_VMS	Методы для отмены UNIX-режима в ExtUtils::MakeMaker
ExtUtils::testlib	Исправляет @INC для использования только что созданного расширения
Opcode	Блокирует коды операций при компиляции Perl-кода
ops	Прагма для использования с модулем Opcode
POSIX	Интерфейс к стандарту IEEE 1003.1
Safe	Создает защищенные пространства имен для оценки Perl-кода
Test::Harness	Выполняет стандартные тестовые Perl-сценарии со сбором статистических данных
vmsish	Обеспечивает возможности, характерные для VMS

Таблица Б.15. Для разработчиков: поддержка объектно-ориентированного программирования

Модуль	Функция
Exporter	Стандартный метод импорта для модулей
overload	Перегружает математические операции Perl
Tie::RefHash	Базовый класс для связанных хешей со ссылками в качестве ключей
Tie::Hash	Содержит определения базового класса для связанных хешей
Tie::Scalar	Содержит определения базового класса для связанных скаляров
Tie::StdHash	Содержит определения базового класса для связанных хешей
Tie::StdScalar	Содержит определения базового класса для связанных скаляров
Tie::SubstrHash	Обеспечивает хеширование с фиксированным размером таблицы и фиксированной длиной ключей
UNIVERSAL	Базовый класс для всех классов

CPAN: не только стандартная библиотека

Если вы не можете найти в стандартной библиотеке модуль, соответствующий вашим потребностям, все равно существует вероятность, что кто-то уже написал код, который будет вам полезен. Есть много превосходных библиотечных модулей, которые не включены в стандартный дистрибутив — по различным причинам практического, политического и вздорного характера. Чтобы выяснить, что есть в наличии, можно заглянуть в Comprehensive Perl Archive Network (CPAN). О CPAN мы говорили в предисловии.

- Вот основные категории модулей, которые можно получить из CPAN:
- Модуль формата листинга.
 - Базовые модули Perl, расширения языка и средства документирования.
 - Модули, обеспечивающие поддержку разработки.
 - Интерфейсы операционных систем.
 - Организация сетей, управление устройствами (модемами) и межпроцессное взаимодействие.
 - Типы данных и утилиты для типов данных.
 - Интерфейсы баз данных.
 - Пользовательские интерфейсы.

- Интерфейсы к другим языкам программирования и средства эмуляции этих языков.
- Имена файлов, файловые системы и блокировки файлов (см. также дескрипторы файлов).
- Обработка строк, обработка текстов, синтаксический анализ и поиск.
- Обработка опций, аргументов, параметров и файлов конфигурации.
- Интернационализация и локализация.
- Аутентификация, защита и шифрование.
- World Wide Web, HTML, HTTP, CGI, MIME.
- Серверные утилиты и демоны.
- Архивирование, сжатие и преобразование.
- Изображения, манипулирование картами пикселей и растрами, рисование и построение графиков.
- Электронная почта и телеконференции Usenet.
- Утилиты управления потоком (обратные вызовы и исключительные ситуации).
- Утилиты для работы с дескрипторами файлов, дескрипторами каталогов и потоками ввода-вывода.
- Модули для Microsoft Windows.
- Прочие модули.

Сетевые клиенты

Немногие компьютеры (и, соответственно, работающие на них пользователи) остаются в изоляции от остального компьютерного мира. Сети, когда-то бывшие достоянием в основном государственных научно-исследовательских лабораторий и факультетов вычислительной техники крупнейших университетов, сейчас доступны практически каждому — даже пользователю домашнего компьютера с модемом, устанавливающему по коммутируемой линии соединение с провайдером с помощью протоколов SLIP или PPP. Сейчас сети больше чем когда-либо используются в повседневной работе организациями и пользователями всех слоев общества — для обмена электронной почтой, планирования встреч, управления распределенными базами данных, доступа к информации предприятий, получения прогнозов погоды, чтения текущих новостей, разговоров с собеседниками из другого полушария, рекламирования продуктов и фирм через Web и т.д.

У всех этих приложений есть одна общая черта: они работают на основе TCP, фундаментального протокола, который обеспечивает взаимодействие между собой всех сетей, входящих в Internet*. И мы имеем в виду не только Internet. Не считая брандмауэров, базовая технология везде одна, независимо от того, устанавливается соединение по Internet, соединение между офисами компании или соединение между кухней и подвалом вашего дома. Это удобно: для того чтобы ориентироваться во всех применениях Internet/intranet, вы должны изучить только одну технологию.

Как же с помощью сети позволить приложению, работающему на одной машине, общаться с другим приложением, которое может функционировать на совершенно другой машине? Средствами Perl это сделать очень легко, но сначала вам, наверное, нужно немного узнать о том, как работает сеть на базе протокола TCP.

* На самом деле коммуникации в Internet обеспечиваются протоколом IP (Internet Protocol), а протокол TCP (Transmission Control Protocol) является протоколом более высокого уровня.

Даже если вы еще ни разу не работали в компьютерной сети, вы уже знаете о системе с установлением соединений: это — телефонная сеть. И пусть вас не смущают причудливые словосочетания вроде “программирование систем клиент/сервер”. Видя слово “клиент”, читайте “вызывающий абонент”, а видя слово “сервер” — читайте “отвечающий абонент”.

Звоня кому-то по телефону, вы выступаете в роли клиента. Тот, кто поднимает трубку на другом конце линии, является сервером.

Программисты, имеющие опыт работы на С, возможно, знакомы с гнездами (sockets). Гнездо — это интерфейс к сети в том же самом смысле, что и дескриптор файла — это интерфейс к файлам в файловой системе. В частности, для тех простых программ, которые мы продемонстрируем ниже, вы можете пользоваться дескриптором гнезда так же, как дескриптором файла*.

Вы можете читать данные из гнезда, записывать в него данные, а также выполнять обе эти операции. Это объясняется тем, что гнездо — особый вид двунаправленного дескриптора файла, представляющего сетевое соединение. В отличие от обычных файлов, созданных посредством функции `open`, гнезда создаются с помощью низкоуровневой функции `socket`.

Давайте выжмем еще немного из нашей телефонной модели. Звоня на коммутатор большой компании, вы можете попросить соединить вас с конкретным отделом по названию (например, с отделом кадров) или по номеру (например, “дополнительный 213”). Представьте, что каждый сервис, работающий на компьютере,— это отдел большой корпорации. Иногда сервис имеет несколько имен, например `http` и `www`, но только один номер, например 80. Этот номер, связанный с именем сервиса, называется его портом. С помощью Perl-функций `getservbyname` и `getservbyport` можно найти имя сервиса по номеру его порта и наоборот. Вот некоторые стандартные TCP-сервисы и номера их портов:

Сервис	Порт	Назначение
echo	7	Принимает все вводимые данные и воспроизводит их
discard	9	Принимает все, но ничего не возвращает
daytime	13	Возвращает текущую дату и местное время
ftp	21	Сервер для обработки запросов пересылки файлов
telnet	23	Сервер для интерактивных telnet-сеансов
smtp	25	Простой протокол пересылки почты; демон-почтальон
time	37	Возвращает число секунд, прошедших с начала 1900-го года (в двоичном формате)
http	80	Сервер World Wide Web
nntp	119	Сервер телеконференций

* Почти так же; поиск по гнезду невозможен.

Хотя гнезда изначально разрабатывались для Berkeley UNIX, все возрастающая популярность Internet побудила практически всех поставщиков операционных систем включить в свои продукты поддержку гнезд для программирования систем клиент/сервер. Функция `socket` является относительно низкоуровневой, а мы в нашей книге рассматриваем в основном высокоуровневые средства Perl. Рекомендуем вам пользоваться более удобным модулем `IO::Socket*`, который мы будем применять во всех наших примерах. Это значит, что мы также будем применять некоторые объектно-ориентированные конструкции Perl. Краткое введение в эти конструкции дано в главе 19. Более подробное введение в объектно-ориентированное программирование на Perl приведено на map-странице *perltoot(1)* и в главе 5 книги *Programming Perl*.

Подробное рассмотрение TCP/IP выходит за рамки нашей книги, но мы можем представить хотя бы несколько простых клиентов. О серверах, которые более сложны, вы можете узнать в главе 6 книги *Programming Perl* и на map-странице *perlipc(1)*.

Простой клиент

Для нашего простейшего клиента мы выберем довольно скучный сервис, который называется `daytime` (“время суток”). Сервер времени суток посылает установившему соединение клиенту одну строку данных, содержащую значение времени суток на этом удаленном сервере, а затем закрывает соединение.

Вот клиент:

```
#!/usr/bin/perl -w
use IO::Socket;
$remote = IO::Socket::INET->new(
    Proto => "tcp",
    PeerAddr => "localhost",
    PeerPort => "daytime(13)",
)
    or die "cannot connect to daytime port at localhost";
while ( <$remote> ) { print }
```

Запустив эту программу, вы должны получить с сервера примерно следующее:

```
Thu May 8 11:57:15 1997
```

* `IO::Socket` входит в состав стандартного дистрибутива Perl версии 5.004. Если у вас более ранняя версия, получите этот модуль из CPAN, где вы найдете модули с простыми интерфейсами к следующим сервисам: DNS, `ftp`, `Ident`(RFC 931), NIS и NISPlus, NNTP, `ping`, POP3, SMTP, SNMP, `SSL`еay, `telnet`, `time` и др.

Вот что означают параметры конструктора `new`

`Proto`

Протокол, который следует использовать. В данном случае возвращенный дескриптор гнезда будет подключен к TCP-гнезду, потому что нам нужно потоковое соединение, т.е. соединение, которое работает почти так же, как старый добрый файл. Не все гнезда такие. Например, с помощью протокола UDP можно создать дейтаграммное гнездо, используемое для передачи сообщений.

`PeerAddr`

Имя или Internet-адрес удаленного хоста, на котором работает сервер. Мы могли бы указать имя подлиннее, например `www.perl.com`, или адрес вроде `204.148.40.9`. Однако для демонстрационных целей мы использовали специальное хост-имя `localhost`, которое всегда должно обозначать машину, на которой вы работаете. Имени `localhost` соответствует Internet-адрес `127.0.0.1`.

`PeerPort`

Имя или номер порта сервиса, с которым мы хотим соединиться. В системах с нормально конфигурированным системным файлом сервисов* мы могли бы обойтись просто именем `daytime`, но на всякий случай мы все же дали в круглых скобках номер порта (13). Использование одного номера без имени тоже сработало бы, но осторожные программисты стараются не использовать числа вместо констант.

Мы обратили внимание на то, как возвращаемое значение конструктора `new` используется в роли дескриптора файла в цикле `while`? Это то, что называется косвенным дескриптором файла — скалярная переменная, содержащая дескриптор файла. Его можно использовать точно так же, как обычный дескриптор. Например, так из него можно прочитать одну строку:

```
$line = <$handle>,
```

так — все остальные строки

```
@lines = <$handle>,
```

а так — послать в него строку данных

```
print $handle "some data\n",
```

* Системный файл сервисов в UNIX находится в каталоге `/etc/services`

Клиент *webget*

Вот простой клиент, который устанавливает соединение с удаленным сервером и получает с него список документов. Этот клиент интереснее предыдущего, потому что перед получением ответа сервера он посылает на него строку данных.

```
#!/usr/bin/perl -w
use IO::Socket;
unless (@ARGV > 1) { die "usage: $0 host document " }
$host = shift(@ARGV);
foreach $document ( @ARGV ) {
    $remote = IO::Socket::INET->new( Proto => "tcp",
        PeerAddr => $host,
        PeerPort => "http(80)",
    );
    unless ($remote) { die "cannot connect to http daemon on $host" }
    $remote->autoflush(1);
    print $remote "GET $document HTTP/1.0\n\n";
    while ( <$remote> ) { print }
    -close $remote;
}
```

Подразумевается, что Web-сервер, на котором работает сервис http, использует свой стандартный порт (номер 80). Если сервер, с которым вы пытаетесь установить соединение, использует другой порт (скажем, 8080), то в качестве третьего аргумента конструктора new() нужно указать PeerPort => 8080. При работе с этим гнездом применяется метод autoflush, потому что в противном случае система буферизировала бы выходную информацию, которую мы ей передали. (Если у вас компьютер Macintosh, то нужно заменить все \n в коде, предназначенном для передачи данные по сети, на \015\012.)

Соединение с сервером — это лишь первый этап процесса: установив соединение, вы должны начать говорить на языке этого сервера. Каждый сервер сети использует свой собственный маленький командный язык, и входные данные, подаваемые на сервер, должны быть сформулированы именно на этом языке. Начинаясь словом GET строка, которую мы послали серверу, соответствует синтаксису протокола HTTP. В данном случае мы просто запрашиваем каждый из указанных документов. Да, мы действительно создаем новое соединение для каждого документа, несмотря на то, что это тот же самый хост. Именно так функционирует HTTP-сервер. (Последние версии Web-браузеров могут требовать, чтобы удаленный сервер оставлял соединение открытым на некоторое время, но сервер не обязан удовлетворять такой запрос.)

Мы назовем нашу программу *webget*. Вот как ее можно было бы выполнить:

```
shell_prompt$ webget www.perl.com /guanaco.html
HTTP/1.1 404 File Not Found
Date: Thu, 08 May 1997 18:02:32 GMT
Server: Apache/1.2b6
Connection: close
Content-type: text/html
<HEAD><TITLE>404 File Not Found</TITLE></HEAD>
<BODY><H1>File Not Found </H1>
The request URL /guanaco.html was not found on this server. <P>
</BODY>
```

Это, конечно, не очень интересно, потому что программа не нашла конкретный документ, однако длинный ответ не поместился бы на этой странице.

Чтобы ознакомиться с более развитой версией данной программы, вам нужно найти программу *lwp-request*, входящую в состав модулей LWP из CPAN. (LWP мы вкратце рассмотрели в конце главы 19.)

Интерактивный клиент

Создать программу-клиент, которая просто читает все с сервера или посылает одну команду, получает один ответ, а затем завершает свою работу, очень легко. А как насчет создания чего-нибудь полностью интерактивного, вроде *telnet*? Мы имеем в виду приложение, которое позволяло бы вам набрать строку, получить ответ, набрать еще одну строку, вновь получить ответ и т.д. (В принципе, *telnet* обычно работает в символьном, а не в строковом режиме, но идею вы поняли.)

Этот клиент — более сложный, чем те два, с которыми мы имели дело до сих пор, но если вы работаете в системе, которая поддерживает мощный вызов *fork*, решение получится не слишком сложным. Установив соединение с тем сервисом, с которым вы хотите пообщаться, клонируйте свой процесс вызовом *fork*. Каждый из созданных идентичных процессов должен выполнить очень простое задание: родительский копирует все из гнезда на стандартный вывод, а порожденный одновременно копирует все со стандартного ввода в гнездо. Реализовать это с помощью только одного процесса было бы гораздо труднее, потому что легче написать два процесса для выполнения одной задачи, чем один процесс — для выполнения двух задач*.

Вот наш код:

```
#'/usr/bin/perl -w
use strict;
use IO::Socket;
```

* Принцип сохранения простоты — один из краеугольных камней не только философии UNIX, но и высококачественного проектирования программного обеспечения. Наверное, именно поэтому он распространился и на другие системы

```

my ($host, $port, $kpid, $handle, $line);
unless (@ARGV == 2 ) { die "usage: $0 host port" }
($host, $port) = @ARGV;
# создать tcp-соединение с указанным хостом и портом
$handle = IO::Socket::INET->new(Proto => "tcp",
    PeerAddr => $host,
    PeerPort => $port)
    or die "can't connect to port $port on $host: $!";
$handle->autoflush(1); # и результат сразу же попадает туда
print STDERR "[Connected to $host:$port]\n";
# разбить программу на два процесса-близнеца
die "can't fork: $!" unless defined($kpid = fork());
# блок if{} выполняется только в родительском процессе
if($kpid) {
    # копировать данные из гнезда на стандартный вывод
    while (defined ($line = <$handle> )) {
        print STDOUT $line;
    }
    kill ("TERM", $kpid); # послать в порожденный процесс сигнал SIGTERM
}
# блок else{} выполняется только в порожденном процессе
else {
    # копировать данные со стандартного ввода в гнездо
    while (defined ($line = <STDIN>)) {
        print $handle $line;
    }
}
}

```

Функция `kill` в блоке `if` родительского процесса пошлет сигнал в наш порожденный процесс (в текущий момент работающий в блоке `else`), как только удаленный сервер закроет свою сторону соединения.

Что еще почитать о сетях

О сетях можно говорить и говорить, но мы дадим ссылки на источники, которые помогут вам перейти от разговоров к делу. В главе 6 книги *Programming Perl* и на [map-странице `perlipc\(1\)`](#) описано межпроцессное взаимодействие в общем; на [map-странице `IO::Socket\(3\)`](#) — объектная библиотека; на [map-странице `Socket\(3\)`](#) — низкоуровневый интерфейс к гнездам. Более опытным программистам рекомендуем книгу *Unix Network Programming* (by Richard Stevens, Addison-Wesley), в которой очень хорошо освещены все вопросы данной темы. Будьте, однако, осторожны: большинство книг по программированию гнезд написаны С-программистами.

Темы, которых мы не коснулись

Как ни странно, даже при таком объеме книги некоторые вопросы все равно остались незатронутыми. Данное приложение содержит полезную дополнительную информацию.

Назначение этого раздела — не обучить вас тем вещам, которые перечислены здесь, а просто дать их перечень. За дальнейшей информацией обращайтесь к книге *Programming Perl*, на man-страницы *perl(1)* и *perlfaq(1)*, к HTML-документам, имеющимся в каталоге *doc* архива CPAN, и к материалам телеконференций Usenet.

Полное межпроцессное взаимодействие

Да, Perl может оказать значительную помощь в создании сети. Кроме потоковых портов TCP/IP, которые мы рассматривали в приложении В, Perl поддерживает также (если ваша система готова к этому) доменные порты UNIX, передачу сообщений по протоколу UDP, совместно используемую память, семафоры, именованные и неименованные каналы и обработку сигналов. Стандартные модули перечислены в главе 6 книги *Programming Perl* и на man-странице *perlipc(1)*, а модули разработки третьих фирм — в разделе каталога модулей CPAN, посвященном сетям.

Да, на основе Perl можно организовать сетевой обмен с использованием портов TCP/IP, доменных портов UNIX и обеспечить работу совместно используемой памяти и семафоров в системах, которые поддерживают эти возможности. Дальнейшие сведения см. на man-странице *perlipc(1)*.

Отладчик

В Perl есть чудесный отладчик, работающий на уровне исходного кода. О нем рассказывается на map-странице *perldebug(1)*.

Командная строка

При запуске интерпретатора Perl можно указывать множество ключей командной строки. См. map-страницу *perlrun(1)*.

Другие операции

Помимо упомянутых в книге, используются и другие операции. Например, операция “запятая”. Есть также операции манипулирования битами `&`, `|`, `^` и `~`, трехместная операция `?\:`, операции `..` и `...` и другие.

Имеются также вариации операций, например, допускается использование модификатора `g` в операции сопоставления. Об этом и многом другом рассказывается на map-странице *perlop(1)*.

Другие функции

В Perl очень много функций. Мы не собираемся здесь их все перечислять, потому что самый простой способ узнать о них — это прочитать раздел о функциях в книге *Programming Perl* и map-страницу *perlfunc(1)*. Вот некоторые из наиболее интересных функций.

Функции *grep* и *map*

Функция `grep` выбирает элементы из списка аргументов на основании результата выражения, которое многократно проверяется на истинность, при этом переменная `$_` последовательно устанавливается равной каждому элементу списка:

```
@bigpowers = grep $_ > 6, 1, 2, 4, 8, 16;    # получает (8, 16)
@b_names = grep /^b/, qw(fred barney betty wilma);
@textfiles = grep -T, <*>;
```

Функция `map` похожа на `grep`, но вместо выбора и отклонения элементов из списка аргументов она просто выдает результаты вычисления выражения (вычисляемого в списочном контексте):

```
@more = map $_ + 3, 3, 5, 7;                # получает 6, 8, 10
@square s = map $_ * $_, 1..10;             # квадраты первых 10 чисел
@that = map "$_\n", @this;                  # как unchop
```

```
@triangle = map 1..$_, 1..5;      # 1,1,2,1,2,3,1,2,3,4,1,2,3,4,5
%sizes = map { $_, -s } <*>;      # хеш файлов и размеров
```

Операция *eval* (*u s///e*)

Да, вы можете создать фрагмент кода во время работы программы и выполнить его при помощи функции `eval` точно так же, как в `shell`. Это по-настоящему полезная возможность, благодаря которой можно выполнить оптимизацию периода компиляции (например, получить скомпилированные регулярные выражения) во время выполнения. С ее помощью можно также вылавливать во фрагменте кода ошибки, которые в противном случае будут фатальными: фатальная ошибка внутри `eval` просто приводит к выходу из этой функции и выдает пользователю код ошибки.

Вот, например, программа, которая читает строку Perl-кода, вводимую пользователем, а затем выполняет ее так, как будто это часть Perl-программы:

```
print "code line: ";
chop($code = <STDIN>);
eval $code; die "eval: $@" if $@;
```

Perl-код можно вставить в заменяющую строку при выполнении операции замены с помощью флага `e`. Это удобно, если вы хотите включить в заменяющую строку нечто сложное, например, вызов подпрограммы, которая возвращала бы результаты поиска в базе данных. Вот цикл, который инкрементирует значение первой колонки для ряда строк:

```
while (<>) {
    s/^(\\S+)/$1+1/e;    # $1+1 is Perl code, not a string
    printf;
}
```

Функция `eval` используется также как механизм обработки исключений:

```
eval {
    some_hairy_routine_that_might_die(@args);
};
if ($@) {
    print "oops some_hairy died with $@";
}
```

Здесь массив `$@` пуст до тех пор, пока блок `eval` работает нормально, а в противном случае этот блок выдает “сообщение о смерти”.

Из этих трех конструкций (`eval "строка"`, `eval { БЛОК }` и `s///e`) только первая действительно соответствует вашим представлениям об `eval` из `shell`. Остальные две компилируются “на ходу”, что влечет за собой незначительное снижение производительности.

Предопределенные переменные

Вы уже знакомы с несколькими предопределенными переменными, например, с переменной `$_`. Их гораздо больше. В дело их обозначения вовлечены почти все знаки препинания. Здесь вам поможет map-страница `perlvar(1)`, а также модуль `English` на странице `perlmod(1)`.

Обработка таблицы символов с помощью *FRED

Вы можете сделать `b` псевдонимом для `a` с помощью операции `*b = *a`. Это значит, что `$a` и `$b` обозначают одну и ту же переменную, равно как `@a` и `@b`, и даже дескрипторы файлов и форматы `a` и `b`. Вы можете также объявить `*b` локальной для данного блока с помощью операции `local(*b)`, что позволит вам иметь локальные дескрипторы файлов, форматы и другие вещи. Это весьма экстравагантно, но иногда бывает весьма полезно.

Дополнительные возможности регулярных выражений

Регулярные выражения могут иметь “расширенный” синтаксис (в котором пробельные символы не обязательны, поэтому регулярное выражение может разбиваться на несколько строк и содержать обычные Perl-комментарии), а также иметь положительное и отрицательное “упреждение”. Этот синтаксис кажется несколько уродливым, но не пугайтесь и обратитесь к книге *Programming Perl* или map-странице `perlre(1)`. Все это и многое другое разъясняется в книге Фридла (Friedl) *Mastering Regular Expressions* (издательство O'Reilly & Associates).

Пакеты

Если над вашим проектом работает много людей или если вы — большой оригинал, вы можете создать пространство имен переменных с помощью пакетов. Пакет — это просто скрытый префикс, который ставится перед именами большинства переменных (кроме тех, которые создаются операцией `my`). Изменяя этот префикс, вы получаете разные переменные. Вот небольшой пример:

```
$a = 123;          # это на самом деле $main::a
$main::a++;        # та же переменная, теперь 124
package fred;      # теперь префикс — fred
$a = 456;          # это $fred::a
```

```
print $a - $main::a;      # выводит 456-124
package main;             # возврат к первоначальному значению по умолчанию
print $a + $fred::a;      # выводит 124+456
```

Таким образом, любое имя с явным именем пакета используется как есть, а все остальные имена считаются принадлежащими текущему пакету, который принят по умолчанию. Пакеты локальны для текущего файла или блока, и вы всегда можете использовать пакет `main` в начале файла. Подробности см. на `man`-странице *perlsub*(1).

Встраиваемость и расширяемость

“Внутренности” Perl определены достаточно хорошо для того, чтобы встраивание компилятора-интерпретатора Perl в другое приложение (так, как это уже сделано с Web-сервером Apache и текстовым редактором *vi*) или расширение Perl путем добавления к нему произвольного кода, написанного на C (или имеющего C-подобный интерфейс), стало относительно несложной задачей. Более того, почти третья часть диалоговой документации на Perl посвящена именно вопросам встраивания и расширения этого языка. Подробно эти аспекты освещены на `man`-страницах *perlembed*(1), *perlapi*(1), *perlxs*(1), *perlxtut*(1), *perlguts*(1) и *perlcall*(1).

Поскольку Perl можно получить бесплатно, вы можете написать собственную программу создания электронных таблиц, используя встроенный Perl для вычисления выражений в ее ячейках и не платя ни цента за всю эту мощь. Радуйтесь.

Вопросы безопасности

Perl создавался с учетом требований безопасности. Посмотрите в главе 6 книги *Programming Perl* или на `man`-странице *perlsec*(1) материал о “проверке пороков” (taint checking). “Проверка пороков” — это такой вид защиты, когда вы доверяете автору программы, а не лицу, которое ее выполняет — как это часто бывает в UNIX с программами, в которых применяется идентификатор пользователя, и во всех системах с программами, запускаемыми сервером. Модуль *Safe*, который описан на `man`-странице *Safe*(3) и в главе 7 книги *Programming Perl*, обеспечивает нечто совершенно иное — защиту, необходимую при выполнении (как в случае с *eval*) непроверенного кода.

Операторы *switch* и *case*

Нет, таких операторов в Perl нет, но их легко создать с помощью базовых конструкций. См. главу 2 книги *Programming Perl* и `man`-страницу *perlfunc*(1).

Прямой ввод-вывод: *sysopen*, *sysread*, *syswrite*, *sysseek*

Иногда средства ввода-вывода Perl оказываются слишком высокоуровневыми для стоящей перед вами задачи. В главе 3 книги *Programming Perl* и на ман-странице *perlfunc*(1) рассматривается вопрос непосредственного использования базовых системных вызовов для ввода-вывода.

Компилятор Perl

Хотя мы говорим, что Perl компилирует ваш код перед тем, как его выполнять, эта скомпилированная форма не является “родным” объектным кодом. Компилятор Perl, написанный Малькольмом Бити, может создать из вашего Perl-сценария независимый байтовый код или компилируемый C-код. Ожидается, что в версию Perl 5.005 будет включена возможность генерации собственного кода. Подробности см. в материале, представленном на ман-странице *perlfaq*(3).

Поддержка баз данных

Да, Perl может обеспечить непосредственное взаимодействие с коммерческими серверами баз данных, включая Oracle, Sybase, Informix и ODBC, не считая многих других. Соответствующие модули расширения вы найдете в разделе баз данных в каталоге модулей CPAN.

Сложные структуры данных

Используя ссылки, вы можете создавать структуры данных произвольной степени сложности. Вопросы их разработки рассматриваются в главе 4 книги *Programming Perl* и на ман-страницах *perllool*(1), *perldsc*(1) и *perlref*(1). Если вы предпочитаете объектно-ориентированную структуру данных, обратитесь к главе 5 вышеупомянутой книги и ман-страницам *perltoot*(1) и *perlobj*(1).

Указатели на функции

Perl может сохранять и передавать указатели на функции посредством записи вида `&funcname`, а также вызывать их косвенно посредством записи `&$funcptr($args)`. Можно даже писать функции, которые создают и возвращают новые анонимные функции, как в языках Lisp и Scheme. Такие анонимные функции часто называют *замыканиями* (*closures*). Подробности см. в главе 2 книги *Programming Perl* и на ман-страницах *perlsub*(1) и *perlfaq*7(1).

И прочее

Perl с каждым днем становится все более мощным и полезным, поэтому оперативное обновление посвященной ему документации — довольно сложная задача. (Кто знает, может быть, к дню появления этой книги на полках магазинов уже будет создан Visual Perl?) В любом случае — спасибо, Ларри!

Предметный указатель

А

Аргументы	128
Ассоциативные массивы	97

Б

Базовые средства ввода-вывода	103
Базы данных	
— пользовательские	217
— произвольного доступа с записями фиксированной длины	220
— с записями переменной длины (текстовые)	222
Библиотека	
— DBM	217
— LWP	258
Библиотеки	292
Блоки операторов	89

В

Возвращаемые значения	40, 127
Выбранный в текущий момент дескриптор файла	162

Г

Глобальные переменные	126
Гнезда	302

Д

Действия	192
Деление с остатком	64
Дескриптор по умолчанию	163
Дескриптор файла по умолчанию	147
Дескрипторы	
— каталогов	170
— файлов	43, 143
заккрытие	144
использование	146
открытие	144
Директивы импорта	236

И

Игнорирование регистра	117
Извлечение и замена подстроки	198
Изменение меток времени	180
Изменение прав доступа	178
Изменение принадлежности файла	179
Индексные ссылки	34
Интерполяция переменных	63, 118

К

Каталоги	167
— создание	178
— удаление	178
Классы	246
Ключи	97
Компоненты формы	238
Конечная лексема	234
Константы	60
Конструкторы	246
Круглые скобки в образцах	113

Л

Лексемы	29
Литералы	60
Литеральные строки	61
— строки в двойных кавычках	62
— строки в одинарных кавычках	61
Лицензия	
— открытая	27
— художественная	27
Локальные переменные в функциях	129

М

Маркер границы слова	37
Массивы	77
Метки	138
Методы	246
— классов	247
— объектов	247
Множители	111
— ленивые	113
— прожорливые	113
Модификаторы выражений	140
Модули	292
Модуль CGI.pm	231

О

Образцы	109
— классы символов	109
— обозначающие один символ	109
— приоритет	115
— фиксированные	114
Общий множитель	112

Общий шлюзовой интерфейс (CGI).... 229

Объектно-ориентированное программирование	246
Объекты	246
Оператор	
— do {} while/until	93
— for	94
— foreach	95
— if/unless	90
— last	135
— next	137
— redo	137
— use	235
— while/until	92
Операторы	89
Операции над массивами	79
Операция	
— замены	109, 120
— конструктора списка	78
— перевода	38
— повторения строки	66
— подстановки	38
— присваивания	69
— ромб	104
— сопоставления	37, 108, 116
— игнорирования регистра	37
— космический корабль	202
— cmp	204
— each	219
— tr	204
— =~	117
Основная программа	29
Отступы	29

П

Пакеты	292
Параметры	40
Передача параметров	235
Переменные среды	185
Переменные-массивы	79
Подпрограммы	40, 125
Поиск	
— ошибок в CGI-программах	254
— подстроки	197
— с возвратом	113

Поледержатели	154	Ссылки	242
— заполненные поля	159	— жесткие	175
— многостроковые	158	— символические	176
— текстовые	157	Стандартный ввод	103
— числовые	158	Стандартный вывод	105
Полулокальные переменные	131	Строка	
Пользовательские функции		— запроса	231
— вызов	126	— значений полей	50
— определение	125	— определения полей	50
Помеченные блоки	138	Строки	61
Порты	302	— значений	154
Последовательности	111	— полей	154
Права доступа	178	— связки	122
Прагмы	292		
Практический язык извлечений и отчетов	25	T	
Пробельные символы	28	Транслитерация	204
Процессы	183		
Путающее зависшее else	90	У	
		Универсальные локаторы ресурсов (URL)	231
P		Управляющие структуры	89
Рабочая область	95	Усовершенствованная сортировка	201
Разбивка	121	Устройства графического ввода	238
Развертывание	168		
Реализации	129	Ф	
Регулярные выражения	107	Файлы	
Редактирование на месте	222	— переименование	174
		— удаление	173
C		Фиксирующие точки	114
Связывание ссылками	175	— упреждающие	115
Сервисы	302	Формат начала страницы	51, 161
Сетевые клиенты	303	Форматы	153
Сети	301	— вызов	156
Сигналы	192	— определение	154
Скалярные данные	59	Функции обработки массивов	79
Скалярные операции	63	Функция	
Скалярные переменные	31, 69	— chdir	167
Скалярный контекст	85	— chmod	178
Служебный язык систем	29	— chown	179
Сохраняемый файл	252	— closedir	170
Списки	77	— dbmclose	219
Списочные литералы	77	— dbmopen	218
Списочный контекст	85	— die	145
Срез	82	— exit	190
Ссылка на список	80	— flock	245

— fork	189	C	
— glob	49, 168		
— glue	122	CGI-программирование	229
— index	197	CGI-программы	229
— link	176	CPAN (Comprehensive Perl Archive Network)	17
— mkdir	178		
— open	144, 221		
— opendir	170		
— pack	221	D	
— param	235	DBM-базы данных	217
— print	221	DBM-массивы	218
— printf	200	DBM-хеши	218
— read	221		
— readdir	171	H	
— readlink	177	Here-документы	234
— rename	174	HTML-формы	232, 238
— rmdir	178	HTTP	232
— seek	221		
— select	163	P	
— sort	201	Perl	
— split	122	— доступность	27
— substr	198	— и awk	225
— system	183	— и sed	227
— unlink	173	— и shell	228
— utime	180	— история создания	25
— wait	189	— назначение	26
— write	156		
		X	
Хеш-переменные	98	World Wide Web	229
Хеш-функции	99		
Хеши	97		
		Ч	
Числа	59		
		Ш	
Шаблоны	154		

Содержание

<i>Предисловие</i>	<i>5</i>
<i>Введение</i>	<i>9</i>
<i>1. Введение</i>	<i>25</i>
История создания языка Perl	25
Назначение языка Perl	26
Доступность	27
Основные понятия	28
Прогулка по стране Perl	30
Упражнение	57
<i>2. Скалярные данные</i>	<i>59</i>
Что такое скалярные данные	59
Числа	59
Строки	61
Скалярные операции	63
Скалярные переменные	69
Скалярные операции и функции	69
<STDIN> как скалярное значение	74
Вывод с помощью функции print	75
Значение undef	75
Упражнения	76
<i>3. Массивы и списочные данные</i>	<i>77</i>
Список и массив	77
Литеральное представление	77
Переменные	79
Операции над массивами и функции обработки массивов	79
Скалярный и списочный контексты	85
<STDIN> как массив	85
Интерполяция массивов	86
Упражнения	87
<i>4. Управляющие структуры</i>	<i>89</i>
Блоки операторов	89
Оператор if/unless	90

Оператор while/until	92
Оператор for	94
Оператор foreach	95
Упражнения	96
5. Хеш	97
Что такое хеш	97
Хеш-переменные	97
Литеральное представление хеша	98
Хеш-функции	99
Срезы хешей	101
Упражнения	102
6. Базовые средства ввода-вывода	103
Ввод из STDIN	103
Ввод из операции “ромб”	104
Вывод в STDOUT	105
Упражнения	106
7. Регулярные выражения	107
Основные понятия	107
Основные направления использования регулярных выражений	107
Образцы	109
Еще об операции сопоставления	116
Операция замены	120
Функции split и join	121
Упражнения	123
8. Функции	125
Определение пользовательской функции	125
Вызов пользовательской функции	126
Возвращаемые значения	127
Аргументы	128
Локальные переменные в функциях	129
Полулокальные переменные, созданные при помощи функции local	131
Создаваемые операцией my() переменные файлового уровня	132
Упражнения	133
9. Управляющие структуры	135
Оператор last	135
Оператор next	137
Оператор redo	137
Метки	138
Модификаторы выражений	139
Операции && и как управляющие структуры	141
Упражнения	142
10. Дескрипторы файлов и проверка файлов	143
Что такое дескриптор файла	143

Открытие и закрытие дескриптора файла	144
Небольшое отступление: функция <code>die</code>	145
Использование дескрипторов файлов	146
Операции для проверки файлов	147
Функции <code>stat</code> и <code>lstat</code>	150
Упражнения	151
11. Форматы	153
Что такое формат	153
Определение формата	154
Вызов формата	156
Еще о поледержателях	157
Формат начала страницы	161
Изменение в форматах установок по умолчанию	162
Упражнения	166
12. Доступ к каталогам	167
Перемещение по дереву каталогов	167
Развертывание	168
Дескрипторы каталогов	170
Открытие и закрытие дескриптора каталога	170
Чтение дескриптора каталога	171
Упражнения	172
13. Манипулирование файлами и каталогами	173
Удаление файла	173
Переименование файла	174
Создание для файла альтернативных имен: связывание ссылками	175
Создание и удаление каталогов	178
Изменение прав доступа	178
Изменение принадлежности	179
Изменение меток времени	180
Упражнения	180
14. Управление процессами	183
Использование функций <code>system</code> и <code>exec</code>	183
Использование обратных кавычек	186
Использование процессов как дескрипторов файлов	187
Использование функции <code>fork</code>	189
Сводка операций, проводимых над процессами	191
Передача и прием сигналов	192
Упражнения	195
15. Другие операции преобразования данных	197
Поиск подстроки	197
Извлечение и замена подстроки	198
Форматирование данных с помощью функции <code>sprintf()</code>	200
Сортировка по заданным критериям	201

Транслитерация	204
Упражнения	207
16. Доступ к системным базам данных	209
Получение информации о паролях и группах	209
Упаковка и распаковка двоичных данных	213
Получение информации о сети	215
Упражнение	216
17. Работа с пользовательскими базами данных	217
DBM-базы данных и DBM-хеши	217
Открытие и закрытие DBM-хешей	218
Использование DBM-хеша	219
Базы данных произвольного доступа с записями фиксированной длины	220
Базы данных с записями переменной длины (текстовые)	222
Упражнения	224
18. Преобразование других программ в Perl-программы	225
Преобразование awk-программ в Perl-программы	225
Преобразование sed-программ в Perl-программы	227
Преобразование shell-сценариев в Perl-программы	227
Упражнение	228
19. CGI-программирование	229
Модуль CGI.pm	230
Ваша CGI-программа в контексте	231
Простейшая CGI-программа	233
Передача параметров через CGI	235
Как сократить объем вводимого текста	236
Генерирование формы	238
Другие компоненты формы	240
Создание CGI-программы гостевой книги	244
Поиск и устранение ошибок в CGI-программах	254
Perl и Web: не только CGI-программирование	256
Дополнительная литература	260
Упражнения	260
Приложение А. Ответы к упражнениям	261
Приложение Б. Библиотеки и модули	289
Приложение В. Сетевые клиенты	298
Приложение Г. Темы, которых мы не коснулись	305
Предметный указатель	311

Об авторах

Рэндал Л. Шварц — торговец и предприниматель, зарабатывающий на жизнь разработкой программного обеспечения, написанием технической литературы, преподаванием, системным администрированием, консультациями по безопасности и производством видеопродукции. Он известен всему миру своими обильными, смешными, изредка неверными посланиями в Usenet — особенно своей подписью "Just another perl hacker" ("Еще один Perl-хакер") в телеконференции *comp.lang.perl*.

Рэндал отшлифовал свои многочисленные навыки за семь лет работы в фирмах Tektronix, ServioLogic и Sequent. С 1985 года он владеет и управляет фирмой Stonehenge Consulting Services в своем родном городе Портленд (штат Орегон).

Том Кристиансен — внештатный консультант, специализирующийся на преподавании Perl и пишущий о нем. После нескольких лет работы в фирме TSR Hobbies (знаменитой своей игрой Dungeons and Dragons) он отправился в колледж, провел год в Испании и пять в Соединенных Штатах, занимаясь музыкой, лингвистикой, программированием и полудюжиной разговорных языков. В конечном итоге он получил в университете штата Висконсин степень бакалавра по испанскому языку и степень магистра по вычислительной технике.

Затем Том провел пять лет в фирме Convex в роли мастера на все руки: он занимался системным администрированием, разработкой утилит и ядра, поддержкой клиентов, обучением и множеством других вопросов. Помимо этого, он два срока был членом совета директоров ассоциации USENIX.

Имея за плечами более чем пятнадцатилетний опыт администрирования и программирования UNIX-систем, Том ведет семинары на международном уровне. Живет он в предгорьях недалеко от города Боулдер (штат Колорадо) в окружении оленей, скунсов, пум и черных медведей, проводя лето в пеших и верховых прогулках, ловле птиц, музицировании и играх.

На нашей обложке

Животное, изображенное на обложке нашей книги, — это лама, одомашненная представительница семейства южноамериканских верблюдов, обитающих в Андах. В эту группу входят также домашняя альпака и ее дикие предки — гуанако и викунья. Дикие гуанако могут бегать со скоростью до 40 миль в час и в случае опасности заходят в воду.

Кости, найденные в поселениях древнего человека, позволяют сделать вывод о том, что приручать альпак и лам начали 4500 лет назад. В 1531 году, когда испанские конкистадоры завоевали империю инков в Андах, они обнаружили большое поголовье этих животных. Южноамериканские верблюды приспособлены для жизни высоко в горах; их гемоглобин может поглощать больше кислорода, чем у других млекопитающих.

Вес лам достигает 300 фунтов, и они используются главным образом как выючные животные. Караван может состоять из нескольких сотен животных и двигаться со скоростью до двадцати миль в день. Лама выдерживает груз весом до 50 фунтов, но очень вспыльчива и, показывая свое недовольство, плюется и кусается. Для жителей Анд ламы — это еще и источник мяса, шерсти (превосходную шерсть дает меньшая по размерам альпака), шкур и жира для свечей. Из шерсти ламы можно вить веревки и ткать ковры, а сушеный навоз используется как топливо.

Изучаем Perl



Изучаем Perl



Предлагаемая вашему вниманию книга стала в США бестселлером. Она написана двумя ведущими специалистами по программированию на языке Perl, которые научат и вас пользоваться этим языком. Предваряемая предисловием Ларри Уолла, создателя Perl, эта тщательно продуманная книга приобрела статус “официального” учебника, предназначенного как для занятий с преподавателем, так и для самостоятельной работы. В данном издании рассматривается версия Perl 5.004.

Книга **“Изучаем Perl”** является учебным пособием, с помощью которого вы быстро научитесь писать Perl-программы. В каждой главе даются упражнения, а в конце книги - подробные решения. Большая глава посвящена CGI-программированию: вы узнаете, как пользоваться библиотечными модулями, применять ссылки и объектно-ориентированные конструкции языка Perl.

Язык Perl предназначен в первую очередь для эффективной работы с текстами, файлами и процессами. Он становится стандартным для большинства UNIX-платформ; существуют его диалекты для большинства современных ОС, и все они могут быть получены бесплатно.

В книге вы найдете:

- Обзор основ программирования на языке Perl
- Систематизированное описание возможностей языка
- Множество примеров небольших программ
- Задачи по программированию и их полные решения
- Методики использования системных команд в Perl-программах
- Способы создания баз данных DBM с помощью Perl
- Введение в CGI-программирование для Web

Рэндал Л. Шварц является соавтором (вместе с Ларри Уоллом и Томом Кристиансеном) книги “Programming Perl”, также выпущенной издательством O'Reilly & Associates. Он хорошо известен как преподаватель Perl, консультант и создатель программного обеспечения. Рэндал - один из основателей Perl Institute. С 1985 г. он владеет и управляет фирмой Stonehenge Consulting Services (г. Портленд, штат Орегон).

Том Кристиансен - внештатный консультант, занимающийся преподаванием Perl и пишущий о нем. Том участвует в выпуске сборников FAQ, посвященных Perl, и является одним из основателей Perl Institute. Живет он поблизости от города Боулдер, штат Колорадо.