Бесплатная электронная книга

УЧУСЬ

Perl Language

Free unaffiliated eBook created from
Stack Overflow contributors.

#perl

Около

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: perl-language

It is an unofficial and free Perl Language ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official Perl Language.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

глава 1: Начало работы с языком Perl

замечания

Perl - верблюд языков: полезный, но не всегда красивый. Он имеет довольно хорошую документацию, доступ к которой можно получить с помощью команды `perldoc` из командной строки. Он также доступен онлайн на perldoc.perl.org .

Версии

Версия	Примечания к выпуску	Дата выхода
+1,000		1987-12-18
+2,000		1988-06-05
+3,000		1989-10-18
4,000		1991-03-21
+5,000		1994-10-17
5,001		1995-05-13
5,002		1996-02-29
5,003		1996-06-25
5,004	perl5004delta	1997-05-15
5,005	perl5005delta	1998-07-22
5.6.0	perl56delta	2000-03-22
5.8.0	perl58delta	2002-07-18
5.8.8	perl581delta , perl582delta , perl583delta , perl584delta , perl585delta , perl586delta , perl587delta , perl588delta	2006-02-01
5.10.0	perl5100delta	2007-12-18

Версия	Примечания к выпуску	Дата выхода
5.12.0	perl5120delta	2010-04-12
5.14.0	perl5140delta	2011-05-14
5.16.0	perl5160delta	2012-05-20
5.18.0	perl5180delta	2013-05-18
5.20.0	perl5200delta	2014-05-27
5.22.0	perl5220delta	2015-06-01
5.24.0	perl5240delta	2016-05-09
5.26.0	perl5260delta	2017-05-30

Examples

Начало работы с Perl

Perl пытается сделать то, что вы имеете в виду:

```
print "Hello World\n";
```

Два сложных бита - точка с запятой в конце строки и `\n` , которая добавляет новую строку (фид строки). Если у вас относительно новая версия perl, вы можете использовать `say` вместо `print` чтобы автоматически добавить возврат каретки:

5.10.0

```
use feature 'say';
say "Hello World";
```

Функция say также включена автоматически с `use v5.10` (или выше):

```
use v5.10;
say "Hello World";
```

Очень просто использовать perl в командной строке с помощью опции `-e` :

```
$ perl -e 'print "Hello World\n"'
Hello World
```

Добавление опции `-l` является одним из способов автоматического распечатывания новых строк:

```
$ perl -le 'print "Hello World"'
Hello World
```

5.10.0

Если вы хотите включить новые функции , используйте вместо этого параметр `-E` :

```
$ perl -E 'say "Hello World"'
Hello World
```

Вы также можете, конечно, сохранить скрипт в файле. Просто удалите параметр командной строки `-e` и используйте имя файла скрипта: `perl script.pl` . Для программ, длинных, чем линия, целесообразно включить несколько опций:

```
use strict;
use warnings;

print "Hello World\n";
```

Нет никакого реального недостатка, кроме как сделать код немного дольше. В свою очередь, строгая прагма предотвращает использование кода, который потенциально небезопасен, а предупреждения предупреждают вас о многих распространенных ошибках.

Обратите внимание, что конечная точка с запятой является необязательной для последней строки, но это хорошая идея, если позже вы добавите конец кода.

Дополнительные сведения о том, как запустить Perl, см. В perlrun или введите `perldoc perlrun` в командной строке. Для более подробного ознакомления с Perl см. Perlintro или введите `perldoc perlintro` в командной строке. Для причудливого интерактивного учебника, попробуйте Perl .

Прочитайте Начало работы с языком Perl онлайн: https://riptutorial.com/ru/perl/topic/341/начало-работы-с-языком-perl

глава 2: Perl однострочные

Examples

Выполнить некоторый код Perl из командной строки

Простые однострочные могут быть указаны в качестве аргументов командной строки для perl, используя ключ `-e` (подумайте «выполнить»):

```
perl -e'print "Hello, World!\n"'
```

Из-за правил цитирования Windows вы не можете использовать строки с одним кавычком, но должны использовать один из этих вариантов:

```
perl -e"print qq(Hello, World!\n)"
perl -e"print \"Hello, World!\n\""
```

Обратите внимание, что во избежание нарушения старого кода, с `-e` можно использовать только синтаксис, доступный до Perl 5.8.x. Чтобы использовать что-либо новое, версия вашего perl может поддерживать, вместо этого используйте `-E` . Например, использовать значение `say` доступное от 5.10.0, плюс Unicode 6.0 от> = v5.14.0 (также использует `-CO` чтобы убедиться, что `STDOUT` печатает UTF-8):

5.14.0

```
perl -CO -E'say "\N{PILE OF POO}"'
```

Использование строк с двойными кавычками в однострочных окнах Windows

Для использования параметров командной строки Windows использует только двойные кавычки. Чтобы использовать двойные кавычки в perl one-liner (т. Е. Для печати строки с интерполированной переменной), вы должны избегать их с помощью обратных косых черт:

```
perl -e "my $greeting = 'Hello'; print \"$greeting, world!\n\""
```

Чтобы улучшить читаемость, вы можете использовать оператор `qq()` :

```
perl -e "my $greeting = 'Hello'; print qq($greeting, world!\n)"
```

Печатать строки, соответствующие шаблону (PCRE grep)

```
perl -ne'print if /foo/' file.txt
```

Без учета регистра:

```
perl -ne'print if /foo/i' file.txt
```

Замените подстроку другим (PCRE sed)

```
perl -pe"s/foo/bar/g" file.txt
```

Или на месте:

```
perl -i -pe's/foo/bar/g' file.txt
```

В Windows:

```
perl -i.bak -pe"s/foo/bar/g" file.txt
```

Печать только определенных полей

```
perl -lane'print "$F[0] $F[-1]"' data.txt
# prints the first and the last fields of a space delimited record
```

Пример CSV:

```
perl -F, -lane'print "$F[0] $F[-1]"' data.csv
```

Печатать строки от 5 до 10

```
perl -ne'print if 5..10' file.txt
```

Редактировать файл на месте

Без резервной копии (не поддерживается в Windows)

```
perl -i -pe's/foo/bar/g' file.txt
```

С резервной копией `file.txt.bak`

```
perl -i.bak -pe's/foo/bar/g' file.txt
```

С резервной копией `old_file.txt.orig` в подкаталог `backup` (при условии, что последний существует):

```
perl -i'backup/old_*.orig' -pe's/foo/bar/g' file.txt
```

Чтение всего файла в виде строки

```
perl -0777 -ne'print "The whole file as a string: --->$_<---\n"'
```

Примечание: `-0777` - это просто соглашение. Любой `-0400` и выше будет то же самое.

Загрузить файл в mojolicious

```
perl -Mojo -E 'p("http://localhost:3000" => form => {Input_Type => "XML", Input_File => {file
=> "d:/xml/test.xml"}})'
```

Файл `d:/xml/test.xml` будет загружен на сервер, который прослушивает подключения на `localhost:3000` (Source)

В этом примере:

`-Mmodule` выполняет `use module;` перед выполнением вашей программы
`-E commandline` используется для ввода одной строки программы
Если у вас нет модуля `ojo` вы можете использовать `cpanm ojo` для ее установки

Чтобы узнать больше о том, как запустить perl, используйте `perldoc perlrun` или прочитайте здесь

Прочитайте Perl однострочные онлайн: https://riptutorial.com/ru/perl/topic/3696/perl-однострочные

глава 3: Perlbrew

Вступление

Perlbrew - это инструмент для управления несколькими установками perl в каталоге `$HOME` .

замечания

Смотрите также

- Официальная домашняя страница для perlbrew
- Документация CPAN для perlbrew

Examples

Настройка perlbrew в первый раз

Создать скрипт установки `~/.perlbrew.sh` :

```
# Reset any environment variables that could confuse `perlbrew`:
export PERL_LOCAL_LIB_ROOT=
export PERL_MB_OPT=
export PERL_MM_OPT=

# decide where you want to install perlbrew:
export PERLBREW_ROOT=~/perlbrew
[[ -f "$PERLBREW_ROOT/etc/bashrc" ]] && source "$PERLBREW_ROOT/etc/bashrc"
```

Создать установочный скрипт `install_perlbrew.sh` :

```
source ~/.perlbrew.sh
curl -L https://install.perlbrew.pl | bash
source "$PERLBREW_ROOT/etc/bashrc"

# Decide which version you would like to install:
version=perl-5.24.1
perlbrew install "$version"
perlbrew install-cpanm
perlbrew switch  "$version"
```

Запустить сценарий установки:

```
./install_perlbrew.sh
```

Добавить в конец вашего `~/.bashrc`

```
[[ -f ~/.perlbrew.sh ]] && source ~/.perlbrew.sh
```

Источник `~/.bashrc` :

```
source ~/.bashrc
```

Прочитайте Perlbrew онлайн: https://riptutorial.com/ru/perl/topic/9144/perlbrew

глава 4: Unicode

замечания

Предупреждение о кодировке имени файла

Стоит отметить, что Filename Encoding - это не только специфичная для *платформы,* но и специфичная для *файловой системы* .

Он никогда *полностью* безопасно предположить (но часто обычно) , что только потому , что вы можете кодировать и записывать заданное имя файла, что когда вы позже попытаться открыть тот же имя файла для чтения, он все равно будет называться то же самое.

Например, если вы пишете в файловую систему, такую как `FAT16` которая не поддерживает юникод, ваши имена файлов могут быть беззвучно переведены в ASCII-совместимые формы.

Но еще *менее* безопасно предположить, что файл, который вы можете создавать, читать и писать с помощью явного именования, будет называться тем же самым при запросе через другие вызовы, например, `readdir` может возвращать разные байты для вашего имени файла, чем вы указали для `open` ,

В некоторых системах, таких как VAX, вы даже не можете предположить, что `readdir` вернет то же имя файла, которое вы указали с `open` для имен файлов, просто как `foo.bar` , поскольку *расширения* имен файлов могут быть искажены операционной системой.

Кроме того, в UNIX существует очень либеральный набор юридических символов для имен файлов, которые разрешена ОС, исключая только `/` и `\0` , где, как и в Windows, существуют определенные диапазоны символов, которые запрещены в именах файлов и будут вызывать ошибки.

Здесь проявляйте большую осторожность, избегайте причудливых трюков с именами файлов, если у вас есть выбор , и всегда проводите тесты, чтобы убедиться, что любые причудливые трюки, которые *вы* используете, непротиворечивы.

Выполняйте вдвойне осторожность, если вы пишете код, предназначенный для запуска на платформах вне вашего контроля, например, если вы пишете код, предназначенный для `CPAN` , и предположите, что по крайней мере 5% вашей пользовательской базы будут

застревать с помощью некоторых древних или сломанных технологий, либо по выбору, случайно, либо силами вне их контроля, и что они будут сговариваться создавать ошибки для них.

: encoding (utf8) vs: utf8

Поскольку UTF-8 является одним из внутренних форматов представления строк в Perl, шаг кодирования / декодирования часто может быть пропущен. Вместо `:encoding(utf-8)` вы можете просто использовать `:utf8` , если ваши данные уже находятся в UTF-8. `:utf8` можно безопасно использовать с выходными потоками, тогда как для входного потока это может быть опасно, потому что это вызывает внутреннюю несогласованность, когда у вас есть недопустимые последовательности байтов. Кроме того, использование `:utf8` для ввода может привести к нарушениям безопасности, поэтому рекомендуется использовать `:encoding(utf-8)` .

Подробнее: В чем разница между: encoding и: utf8

UTF-8 vs utf8 vs UTF8

Начиная с Perl `v5.8.7` , `"UTF-8"` (с тире) означает UTF-8 в строгом и безопасном виде, тогда как `"utf8"` означает UTF-8 в его либеральной и свободной форме.

Например, `"utf8"` может использоваться для кодовых точек, которые не существуют в Юникоде, например, `0xFFFFFFFF` . Соответственно, недопустимые байтовые последовательности UTF-8, такие как `"\x{FE}\x{83}\x{BF}\x{BF}\x{BF}\x{BF}\x{BF}"` будут декодироваться в недействительный Unicode (но действует Perl) элемент кода (`0xFFFFFFFF`) при использовании `"utf8"` , в то время как `"UTF-8"` кодирования не допустит декодирования для кодовых значений за пределами диапазона допустимых Unicode и даст вам замену символа (`0xFFFD`) вместо этого.

Поскольку имена кодировок нечувствительны к регистру, `"UTF8"` совпадает с `"utf8"` (т.е. *нестрогим* вариантом).

Более подробная информация: UTF-8 vs. utf8 vs. UTF8

Подробнее Чтение

Подробная информация о работе с Unicode в Perl описана более подробно в следующих источниках:

- perlunicode
- perlunitut
- perluniintro
- perlunifaq
- perlunicook
- utf8 прагма
- Функция unicode_strings
- открытая прагма
- PerlIO
- PerlIO :: кодирование
- открытая функция
- шифровать
- perlrun - ключи командной строки
- Глава 6, Программирование Perl

Сообщения из stackoverflow.com (caveat: возможно, не обновлены):

- Почему современный Perl избегает UTF-8 по умолчанию?

YouTube видео:

- Million Billion Squiggly Characters от Ricardo Signes на YAPC NA 2016.

Examples

Создание имен файлов

В следующих примерах используется кодировка UTF-8 для представления имен файлов (и имен каталогов) на диске. Если вы хотите использовать другую кодировку, вы должны использовать `Encode::encode(...)` .

```
use v5.14;
# Make Perl recognize UTF-8 encoded characters in literal strings.
# For this to work: Make sure your text-editor is using UTF-8, so
# that bytes on disk are really UTF-8 encoded.
use utf8;

# Ensure that possible error messages printed to screen are converted to UTF-8.
# For this to work: Check that your terminal emulator is using UTF-8.
binmode STDOUT, ':utf8';
binmode STDERR, ':utf8';

my $filename = 'æ€'; # $filename is now an internally UTF-8 encoded string.

# Note: in the following it is assumed that $filename has the internal UTF-8
#  flag set, if $filename is pure ASCII, it will also work since its encoding
#  overlaps with UTF-8. However, if it has another encoding like extended ASCII,
#  $filename will be written with that encoding and not UTF-8.
# Note: it is not necessary to encode $filename as UTF-8 here
#  since Perl is using UTF-8 as its internal encoding of $filename already
```

```
# Example1  -- using open()
open ( my $fh, '>', $filename ) or die "Could not open '$filename': $!";
close $fh;

# Example2 -- using qx() and touch
qx{touch $filename};

# Example3 -- using system() and touch
system 'touch', $filename;

# Example4 -- using File::Touch
use File::Touch;
eval { touch( $filename ) }; die "Could not create file '$filename': $!" if $@;
```

Чтение имен файлов

Perl не пытается декодировать имена файлов, возвращаемые встроенными функциями или модулями. Такие строки, представляющие имена файлов, всегда должны быть явно декодированы, чтобы Perl распознавал их как Unicode.

```
use v5.14;
use Encode qw(decode_utf8);

# Ensure that possible error messages printed to screen are converted to UTF-8.
# For this to work: Check that you terminal emulator is using UTF-8.
binmode STDOUT, ':utf8';
binmode STDERR, ':utf8';

# Example1  -- using readdir()
my $dir = '.';
opendir(my $dh, $dir) or die "Could not open directory '$dir': $!";
while (my $filename = decode_utf8(readdir $dh)) {
    # Do something with $filename
}
close $dh;

# Example2  -- using getcwd()
use Cwd qw(getcwd);
my $dir = decode_utf8( getcwd() );

# Example3  -- using abs2rel()
use File::Spec;
use utf8;
my $base = 'ø';
my $path = "$base/b/æ";
my $relpath = decode_utf8( File::Spec->abs2rel( $path, $base ) );
# Note: If you omit $base, you need to encode $path first:
use Encode qw(encode_utf8);
my $relpath = decode_utf8( File::Spec->abs2rel( encode_utf8( $path ) ) );

# Example4  -- using File::Find::Rule (part1 matching a filename)
use File::Find::Rule;
use utf8;
use Encode qw(encode_utf8);
my $filename = 'æ';
# File::Find::Rule needs $filename to be encoded
my @files = File::Find::Rule->new->name( encode_utf8($filename) )->in('.');
```

```
$_ = decode_utf8( $_ ) for @files;

# Example5  -- using File::Find::Rule (part2 matching a regular expression)
use File::Find::Rule;
use utf8;
my $pat = '[æ].$'; # Unicode pattern
# Note: In this case:  File::Find::Rule->new->name( qr/$pat/ )->in('.')
#  will not work since $pat is Unicode and filenames are bytes
#  Also encoding $pat first will not work correctly
my @files;
File::Find::Rule->new->exec( sub { wanted( $pat, \@files ) } )->in('.');
$_ = decode_utf8( $_ ) for @files;
sub wanted {
    my ( $pat, $files ) = @_;
    my $name = decode_utf8( $_ );
    my $full_name = decode_utf8( $File::Find::name );
    push @$files, $full_name if $name =~ /$pat/;
}
```

Примечание. Если вы обеспокоены недопустимым UTF-8 в именах файлов, использование `decode_utf8( ... )` в приведенных выше примерах, вероятно, должно быть заменено `decode( 'utf-8', ... )` . Это объясняется тем, что `decode_utf8( ... )` является синонимом `decode( 'utf8', ... )` и существует разница между кодировками `utf-8` и `utf8` (см. Ниже замечания для получения дополнительной информации), где `utf-8` больше строгого, что приемлемо, чем `utf8` .

Переключатели командной строки для однострочных

Включить прагму utf8

Чтобы включить прагму `utf8` в одном `-Mutf8` , интерпретатор perl должен вызываться с параметром `-Mutf8` :

```
perl -Mutf8 -E 'my $□ = "human"; say $□'
```

Работа с Unicode с ключом -C

Флаг командной строки `-C` позволяет вам управлять функциями Unicode. За ним может следовать список букв опций.

Стандартный ввод-вывод

- `I` - `STDIN` будет в *UTF-8*
- `O` - `STDOUT` будет в *UTF-8*
- `E` - `STDERR` будет в *UTF-8*
- `S` - сокращение для `IOE` , стандартные потоки ввода-вывода будут в *UTF-8*

```
echo "Ματαιότης ματαιοτήτων" | perl -CS -Mutf8 -nE 'say "ok" if /Ματαιότης/'
```

Аргументы сценария

- `A` - обрабатывает `@ARGV` как массив строк с *кодировкой UTF-8*

```
perl -CA -Mutf8 -E 'my $arg = shift; say "anteater" if $arg eq "муравьед"' муравьед
```

Уровень по умолчанию PerlIO

- `i` - *UTF-8* - это уровень PerlIO по умолчанию для входных потоков
- `o` - *UTF-8* является уровнем PerlIO по умолчанию для выходных потоков
- `D` - сокращенное `io`

```
perl -CD -Mutf8 -e 'open my $fh, ">", "utf8.txt" or die $!; print $fh "□□ □□□"'
```

`-M` и `-C` могут быть объединены:

```
perl -CASD -Mutf8 -E 'say "Ματαιότης ματαιοτήτων\n"';
```

Стандартный ввод-вывод

Кодировка, используемая для стандартных дескрипторов ввода / вывода (`STDIN` , `STDOUT` и `STDERR`), может быть задана отдельно для каждого дескриптора с использованием `binmode` :

```
binmode STDIN, ':encoding(utf-8)';
binmode STDOUT, ':utf8';
binmode STDERR, ':utf8';
```

Примечание: при чтении, в общем, предпочтительнее `:encoding(utf-8)` over `:utf8` , см. Примечания для получения дополнительной информации.

Кроме того, вы можете использовать `open` прагму.

```
# Setup such that all subsequently opened input streams will use ':encoding(utf-8)'
# and all subsequently opened output streams will use ':utf8'
# by default
use open (IN => ':encoding(utf-8)', OUT => ':utf8');
# Make the (already opened) standard file handles inherit the setting
# given by the IO settings for the open pragma
use open ( :std );
# Now, STDIN has been converted to ':encoding(utf-8)', and
# STDOUT and STDERR have ':utf8'
```

Кроме того, для установки всех дескрипторов файлов (как тех, которые еще предстоит

открыть, так и стандартных) `:encoding(utf-8)` :

```
use open qw( :encoding(utf-8) :std );
```

Ручки файлов

Установка кодировки с помощью open ()

При открытии текстового файла вы можете явно указать его кодировку с помощью трех аргументов `open()` . Этот en- / decoder, прикрепленный к дескриптору файла, называется «слоем ввода / вывода»:

```
my $filename = '/path/to/file';
open my $fh, '<:encoding(utf-8)', $filename or die "Failed to open $filename: $!";
```

См. Замечания для обсуждения различий между `:utf8` и `:encoding(utf-8)` .

Настройка кодировки с помощью binmode ()

Кроме того, вы можете использовать binmode () для установки кодировки для отдельного дескриптора файла:

```
my $filename = '/path/to/file';
open my $fh, '<', $filename or die "Failed to open $filename: $!";
binmode $fh, ':encoding(utf-8)';
```

открытая прагма

Чтобы избежать установки кодировки для каждого дескриптора файла отдельно, вы можете использовать `open` прагму для установки уровня ввода-вывода по умолчанию, используемого всеми последующими вызовами функции `open()` и аналогичных операторов в лексической области этой прагмы:

```
# Set input streams to ':encoding(utf-8)' and output streams to ':utf8'
use open (IN => ':encoding(utf-8)', OUT => ':utf8');
# Or to set all input and output streams to ':encoding(utf-8)'
use open ':encoding(utf-8)';
```

Установка кодировки с помощью

командной строки -C

Наконец, также возможно запустить интерпретатор perl с флагом `-CD` который применяет UTF-8 в качестве уровня ввода-вывода по умолчанию. Однако этого варианта следует избегать, поскольку он зависит от поведения конкретного пользователя, которое невозможно предсказать и не контролировать.

Прагма utf8: использование Unicode в ваших источниках

`utf8` указывает, что исходный код будет интерпретироваться как UTF-8. Конечно, это будет работать, только если ваш текстовый редактор также сохранит исходный код в кодировке UTF-8.

Теперь строковые литералы могут содержать произвольные символы Unicode; идентификаторы могут также содержать Unicode, но только словарные символы (см. perldata и perlrecharclass для получения дополнительной информации):

```
use utf8;
my $var1 = '$я$©';      # works fine
my $я = 4;               # works since я is a word (matches \w) character
my $p$2 = 3;             # does not work since $ is not a word character.
say "ya" if $var1 =~ /я$/; # works fine (prints "ya")
```

Примечание . При печати текста на терминал убедитесь, что он поддерживает UTF-8. *

Могут быть сложные и противоинтуитивные отношения между кодированием вывода и источника. Находясь на терминале UTF-8, вы можете обнаружить, что добавление `utf8` похоже, нарушает:

```
$ perl -e 'print "Møøse\n"'
Møøse
$ perl -Mutf8 -e 'print "Møøse\n"'
M◆◆se
$ perl -Mutf8 -CO -e 'print "Møøse\n"'
Møøse
```

В первом случае Perl обрабатывает строку как необработанные байты и печатает их так. Поскольку эти байты являются действительными UTF-8, они выглядят правильными, хотя Perl действительно не знает, какие символы они (например, `length("Møøse")` вернет 7, а не 5). Когда вы добавляете `-Mutf8` , Perl правильно декодирует исходный код UTF-8 для символов, но выход по умолчанию работает в режиме Latin-1, а печать Latin-1 на терминал UTF-8 не работает. Только при переключении `STDOUT` на UTF-8 с использованием `-CO` выход будет правильным.

`use utf8` не влияет на стандартную кодировку ввода-вывода и файлы.

Чтение недопустимого UTF-8

При чтении кодированных данных UTF-8 важно знать, что кодированные данные UTF-8 могут быть недействительными или искаженными. Такие данные обычно не принимаются вашей программой (если вы не знаете, что делаете). При неожиданном столкновении с искаженными данными можно рассмотреть различные действия:

- Распечатайте stacktrace или сообщение об ошибке и прекратите программу изящно, или
- Вставьте символ замещения в том месте, где появилась некорректная последовательность байтов, напечатайте предупреждающее сообщение в STDERR и продолжайте читать, поскольку ничего не произошло.

По умолчанию Perl `warn` вас о сбоях в кодировке, но не прерывает вашу программу. Вы можете сделать свою программу прерванной, сделав предупреждения UTF-8 фатальными, но помните о предостережениях в Fatal Warnings .

Следующий пример записывает 3 байта в кодировке ISO 8859-1 на диск. Затем он пытается прочитать байты снова как кодированные данные UTF-8. Один из байтов, `0xE5` , является недопустимой последовательностью в 1 байт UTF-8:

```
use strict;
use warnings;
use warnings FATAL => 'utf8';

binmode STDOUT, ':utf8';
binmode STDERR, ':utf8';
my $bytes = "\x{61}\x{E5}\x{61}";  # 3 bytes in iso 8859-1: aåa
my $fn = 'test.txt';
open ( my $fh, '>:raw', $fn ) or die "Could not open file '$fn': $!";
print $fh $bytes;
close $fh;
open ( $fh, "<:encoding(utf-8)", $fn ) or die "Could not open file '$fn': $!";
my $str = do { local $/; <$fh> };
close $fh;
print "Read string: '$str'\n";
```

Программа будет прервана с фатальным предупреждением:

```
utf8 "\xE5" does not map to Unicode at ./test.pl line 10.
```

Строка 10 здесь вторая вторая строка, и ошибка возникает в части строки с помощью `<$fh>` при попытке прочитать строку из файла.

Если вы не делаете предупреждений, фатальных в вышеуказанной программе, Perl все

равно будет печатать предупреждение. Однако в этом случае он попытается восстановить из искаженного байта `0xE5` , вставив в поток четыре символа `\xE5` , а затем продолжит следующий байт. В результате программа напечатает:

```
Read string: 'a\xE5a'
```

Прочитайте Unicode онлайн: https://riptutorial.com/ru/perl/topic/4375/unicode

глава 5: Даты и время

Examples

Создать новую дату

Установите `DateTime` на свой компьютер, а затем используйте его в perl-скрипте:

```
use DateTime;
```

Создать новое текущее время

```
$dt = DateTime->now( time_zone => 'Asia/Ho_Chi_Minh');
```

Затем вы можете получить доступ к значениям элементов даты и времени:

```
$year   = $dt->year;
$month  = $dt->month;
$day    = $dt->day;
$hour   = $dt->hour;
$minute = $dt->minute;
$second = $dt->second;
```

Чтобы получить только время:

```
my $time = $dt->hms; #требуемое время в формате hh:mm:ss
```

Чтобы получить только дату:

```
my $date = $dt->ymd; # дата окончания с форматом yyyy-mm-dd
```

Работа с элементами datetime

Установить один элемент:

```
$dt->set( year => 2016 );
```

Задайте множество элементов:

```
$dt->set( year => 2016, 'month' => 8);
```

Добавить продолжительность в datetime

```
$dt->add( hour => 1, month => 2)
```

Вычитание по времени:

```
my $dt1 = DateTime->new(
      year       => 2016,
      month      => 8,
```

```
      day       => 20,
);

my $dt2 = DateTime->new(
      year      => 2016,
      month     => 8,
      day       => 24,
);

my $duration = $dt2->subtract_datetime($dt1);
print $duration->days
```

Результат получится 4 дня

Рассчитать время выполнения кода

```
use Time::HiRes qw( time );

my $start = time();

#Code for which execution time is calculated
sleep(1.2);

my $end = time();

printf("Execution Time: %0.02f s\n", $end - $start);
```

Это будет печатать время выполнения кода в секундах

Прочитайте Даты и время онлайн: https://riptutorial.com/ru/perl/topic/5920/даты-и-время

глава 6: Даты и время

Examples

Форматирование даты

Время :: шт доступно в perl 5 после версии 10

```
use Time::Piece;

my $date = localtime->strftime('%m/%d/%Y');
print $date;
```

```
Output
07/26/2016
```

Прочитайте Даты и время онлайн: https://riptutorial.com/ru/perl/topic/5923/даты-и-время

глава 7: Интерполяция в Perl

Examples

Базовая интерполяция

Интерполяция означает, что интерпретатор Perl будет подставлять значения переменных для их имени и некоторых символов (которые невозможно или трудно вводить напрямую) для специальных последовательностей символов (это также называется экранированием). Самое важное различие заключается в одиночных и двойных кавычках: двойные кавычки интерполируют замкнутую строку, но одинарные кавычки не делают.

```
my $name = 'Paul';
my $age = 64;
print "My name is $name.\nI am $age.\n"; # My name is Paul.
                                         # I am 64.
```

Но:

```
print 'My name is $name.\nI am $age.\n'; # My name is $name.\nI am $age.\n
```

Вы можете использовать `q{}` (с любым разделителем) вместо одиночных кавычек и `qq{}` вместо двойных кавычек. Например, `q{I'm 64}` позволяет использовать апостроф внутри строки без интерполирования (иначе она завершит строку).

Заявления:

```
print qq{$name said: "I'm $age".}; # Paul said: "I'm 64".
print "$name said: \"I'm $age\"."  # Paul said: "I'm 64".
```

делайте то же самое, но в первом вам не нужно избегать двойных кавычек внутри строки.

Если ваше имя переменной сталкивается с окружающим текстом, вы можете использовать синтаксис `${var}` для устранения неоднозначности:

```
my $decade = 80;
print "I like ${decade}s music!"  # I like 80s music!
```

Что интерполировано

Perl интерполирует имена переменных:

```
my $name = 'Paul';
print "Hello, $name!\n"; # Hello, Paul!
```

```
my @char = ('a', 'b', 'c');
print "$char[1]\n"; # b

my %map = (a => 125, b => 1080, c => 11);
print "$map{a}\n"; # 125
```

Массивы могут быть интерполированы как целое, их элементы разделены пробелами:

```
my @char = ('a', 'b', 'c');
print "My chars are @char\n"; # My chars are a b c
```

Perl *не* интерполирует хеши в целом:

```
my %map = (a => 125, b => 1080, c => 11);
print "My map is %map\n"; # My map is %map
```

и вызовы функций (включая константы):

```
use constant {
    PI => '3.1415926'
};
print "I like PI\n";         # I like PI
print "I like " . PI . "\n"; # I like 3.1415926
```

Perl интерполирует *escape-последовательности,* начиная с `\` :

```
\t                  horizontal tab
\n                  newline
\r                  return
\f                  form feed
\b                  backspace
\a                  alarm (bell)
\e                  escape
```

Интерполяция `\n` зависит от системы, в которой работает программа: она будет генерировать символ новой строки в соответствии с текущими системными соглашениями.

Perl *не* интерполирует `\v` , что означает вертикальную вкладку на C и других языках.

Характер может быть адресован с использованием их кодов:

```
\x{1d11e}     𝄞 by hexadecimal code
\o{350436}    𝄞 by octal code
\N{U+1d11e}   𝄞 by Unicode code point
```

или Unicode:

```
\N{MUSICAL SYMBOL G CLEF}
```

Персонаж с кодами от `0x00` до `0xFF` в *нативном* кодировании может быть рассмотрен в

краткой форме:

```
\x0a     hexadecimal
\012     octal
```

Управляющий символ может быть обработан с помощью специальных управляющих последовательностей:

```
\c@      chr(0)
\ca      chr(1)
\cb      chr(2)
...
\cz      chr(26)
\c[      chr(27)
\c\      chr(28) # Cannot be used at the end of a string
                 # since backslash will interpolate the terminating quote
\c]      chr(29)
\c^      chr(30)
\c_      chr(31)
\c?      chr(127)
```

Прописные буквы имеют то же значение: `"\cA" == "\ca"` .

Интерпретация всех управляющих последовательностей, за исключением `\N{...}` может зависеть от платформы, поскольку они используют кодовые и зависимые от платформы коды.

Прочитайте Интерполяция в Perl онлайн: https://riptutorial.com/ru/perl/topic/5284/интерполяция-в-perl

глава 8: Команды Perl для Windows Excel с помощью модуля Win32 :: OLE

Вступление

В этих примерах используются наиболее часто используемые команды Perl для управления Excel через модуль Win32 :: OLE.

Синтаксис

- $ Sheet-> Range (*Cell1* , [*Cell2*]) # Выбрать ячейку или диапазон ячеек
- $ Sheet-> Cells (*rowIndex* , *columnIndex*) #Выберите ячейку по индексу строки и столбца

параметры

параметры	подробности
Cell1 (обязательно)	Название диапазона. Это должно быть ссылка типа A1 в языке макроса. Он может включать оператор диапазона (двоеточие), оператор пересечения (пробел) или оператор объединения (запятая).
Cell2 (необязательно)	Если указано, *Cell1* соответствует верхнему левому углу диапазона, а *Cell2* соответствует нижнему правому углу диапазона

замечания

Ссылка для получения информации о цветах в Excel:
http://dmcritchie.mvps.org/excel/colors.htm

	A	B	C	D
1		1		29
2		2		30
3		3		31
4		4		32
5		5		33
6		6		34
7		7		35
8		8		36
9		9		37
10		10		38
11		11		39
12		12		40
13		13		41
14		14		42
15		15		43
16		16		44
17		17		45
18		18		46
19		19		47
20		20		48
21		21		49
22		22		50
23		23		51
24		24		52
25		25		53
26		26		54
27		27		55
28		28		56

Ссылка для информации о константах Excel: http://msdn.microsoft.com/en-us/library/aa221100%28office.11%29.aspx

Ссылки из модуля Win32 :: OLE: http://search.cpan.org/~jdb/Win32-OLE-0.1712/lib/Win32/OLE.pm#EXAMPLES

Полезную информацию об использовании Excel можно найти по этому адресу

Examples

1. Открытие и сохранение Excel / рабочих книг

```
#Modules to use
use Cwd 'abs_path';
use Win32::OLE;
use Win32::OLE qw(in with);
use Win32::OLE::Const "Microsoft Excel";
$Win32::OLE::Warn = 3;

#Need to use absolute path for Excel files
my $excel_file = abs_path("$Excel_path") or die "Error: the file $Excel_path has not been
found\n";

# Open Excel application
my $Excel = Win32::OLE->GetActiveObject('Excel.Application')
```

```
    || Win32::OLE->new('Excel.Application', 'Quit');

# Open Excel file
my $Book = $Excel->Workbooks->Open($excel_file);

#Make Excel visible
$Excel->{Visible} = 1;

#___ ADD NEW WORKBOOK
my $Book = $Excel->Workbooks->Add;
my $Sheet = $Book->Worksheets("Sheet1");
$Sheet->Activate;

#Save Excel file
$Excel->{DisplayAlerts}=0; # This turns off the "This file already exists" message.
$Book->Save; #Or $Book->SaveAs("C:\\file_name.xls");
$Book->Close; #or $Excel->Quit;
```

2. Манипуляция рабочими листами

```
#Get the active Worksheet
my $Book = $Excel->Activewindow;
my $Sheet = $Book->Activesheet;

#List of Worksheet names
my @list_Sheet = map { $_->{'Name'} } (in $Book->{Worksheets});

#Access a given Worksheet
my $Sheet = $Book->Worksheets($list_Sheet[0]);

#Add new Worksheet
$Book->Worksheets->Add({After => $workbook->Worksheets($workbook->Worksheets->{Count})});

#Change Worksheet Name
$Sheet->{Name} = "Name of Worksheet";

#Freeze Pane
$Excel -> ActiveWindow -> {FreezePanes} = "True";

#Delete Sheet
$Sheet -> Delete;
```

3. Манипуляция клетками

```
#Edit the value of a cell (2 methods)
$Sheet->Range("A1")->{Value} = 1234;
$Sheet->Cells(1,1)->{Value} = 1234;

#Edit the values in a range of cells
$Sheet->Range("A8:C9")->{Value} = [[ undef, 'Xyzzy', 'Plugh' ],
                                  [ 42,    'Perl',  3.1415  ]];

#Edit the formula in a cell (2 types)
$Sheet->Range("A1")->{Formula} = "=A1*9.81";
$Sheet->Range("A3")->{FormulaR1C1} = "=SUM(R[-2]C:R[-1]C)";      # Sum of rows
$Sheet->Range("C1")->{FormulaR1C1} = "=SUM(RC[-2]:RC[-1])";      # Sum of columns
```

```
#Edit the format of the text (font)
$Sheet->Range("G7:H7")->Font->{Bold}       = "True";
$Sheet->Range("G7:H7")->Font->{Italic}     = "True";
$Sheet->Range("G7:H7")->Font->{Underline}  = xlUnderlineStyleSingle;
$Sheet->Range("G7:H7")->Font->{Size}       = 8;
$Sheet->Range("G7:H7")->Font->{Name}       = "Arial";
$Sheet->Range("G7:H7")->Font->{ColorIndex} = 4;

#Edit the number format
$Sheet -> Range("G7:H7") -> {NumberFormat} = "\@";                                  # Text
$Sheet -> Range("A1:H7") -> {NumberFormat} = "\$#,##0.00";                         # Currency
$Sheet -> Range("G7:H7") -> {NumberFormat} = "\$#,##0.00_);[Red](\$#,##0.00)";      # Currency
- red negatives
$Sheet -> Range("G7:H7") -> {NumberFormat} = "0.00_);[Red](0.00)";                  # Numbers
with decimals
$Sheet -> Range("G7:H7") -> {NumberFormat} = "#,##0";                           #      Numbers
with commas
$Sheet -> Range("G7:H7") -> {NumberFormat} = "#,##0_);[Red](#,##0)";                # Numbers
with commas - red negatives
$Sheet -> Range("G7:H7") -> {NumberFormat} = "0.00%";                               # Percents
$Sheet -> Range("G7:H7") -> {NumberFormat} = "m/d/yyyy"                             # Dates

#Align text
$Sheet -> Range("G7:H7") -> {HorizontalAlignment} = xlHAlignCenter;                 # Center
text;
$Sheet -> Range("A1:A2") -> {Orientation} = 90;                                     # Rotate
text

#Activate Cell
$Sheet -> Range("A2") -> Activate;

$Sheet->Hyperlinks->Add({
   Anchor          =>  $range, #Range of cells with the hyperlink; e.g. $Sheet->Range("A1")
   Address         =>  $adr, #File path, http address, etc.
   TextToDisplay   =>  $txt, #Text in the cell
   ScreenTip       =>  $tip, #Tip while hovering the mouse over the hyperlink
});
```

NB: чтобы получить список гиперссылок, просмотрите следующее сообщение. Список гиперссылок из листа Excel с помощью Perl Win32 :: OLE

4. Манипуляция строк / столбцов

```
#Insert a row before/after line 22
$Sheet->Rows("22:22")->Insert(xlUp, xlFormatFromRightOrBelow);
$Sheet->Rows("23:23")->Insert(-4121,0);     #xlDown is -4121 and that xlFormatFromLeftOrAbove
is 0

#Delete a row
$Sheet->Rows("22:22")->Delete();

#Set column width and row height
$Sheet -> Range('A:A') -> {ColumnWidth} = 9.14;
$Sheet -> Range("8:8") -> {RowHeight}   = 30;
$Sheet -> Range("G:H") -> {Columns} -> Autofit;

# Get the last row/column
my $last_row = $Sheet -> UsedRange -> Find({What => "*", SearchDirection => xlPrevious,
```

```
SearchOrder => xlByRows})    -> {Row};
my $last_col = $Sheet -> UsedRange -> Find({What => "*", SearchDirection => xlPrevious,
SearchOrder => xlByColumns}) -> {Column};


#Add borders (method 1)
$Sheet -> Range("A3:H3") -> Borders(xlEdgeBottom)       -> {LineStyle}  = xlDouble;
$Sheet -> Range("A3:H3") -> Borders(xlEdgeBottom)       -> {Weight}     = xlThick;
$Sheet -> Range("A3:H3") -> Borders(xlEdgeBottom)       -> {ColorIndex} = 1;
$Sheet -> Range("A3:H3") -> Borders(xlEdgeLeft)         -> {LineStyle}  = xlContinuous;
$Sheet -> Range("A3:H3") -> Borders(xlEdgeLeft)         -> {Weight}     = xlThin;
$Sheet -> Range("A3:H3") -> Borders(xlEdgeTop)          -> {LineStyle}  = xlContinuous;
$Sheet -> Range("A3:H3") -> Borders(xlEdgeTop)          -> {Weight}     = xlThin;
$Sheet -> Range("A3:H3") -> Borders(xlEdgeBottom)       -> {LineStyle}  = xlContinuous;
$Sheet -> Range("A3:H3") -> Borders(xlEdgeBottom)       -> {Weight}     = xlThin;
$Sheet -> Range("A3:H3") -> Borders(xlEdgeRight)        -> {LineStyle}  = xlContinuous;
$Sheet -> Range("A3:H3") -> Borders(xlEdgeRight)        -> {Weight}     = xlThin;
$Sheet -> Range("A3:H3") -> Borders(xlInsideVertical)   -> {LineStyle}  = xlDashDot
$Sheet -> Range("A3:H3") -> Borders(xlInsideVertical)   -> {Weight}     = xlMedium;
$Sheet -> Range("A3:I3") -> Borders(xlInsideHorizontal) -> {LineStyle}  = xlContinuous;
$Sheet -> Range("A3:I3") -> Borders(xlInsideHorizontal) -> {Weight}     = xlThin;

#Add borders (method 2)
my @edges = qw (xlInsideHorizontal xlInsideVertical xlEdgeBottom xlEdgeTop xlEdgeRight);
foreach my $edge (@edges)
```

Прочитайте Команды Perl для Windows Excel с помощью модуля Win32 :: OLE онлайн: https://riptutorial.com/ru/perl/topic/3420/команды-perl-для-windows-excel-с-помощью-модуля-win32----ole

глава 9: Комментарии

Examples

Однострочные комментарии

Однострочные комментарии начинаются с знака фунта `#` и идут в конец строки:

```
# This is a comment

my $foo = "bar"; # This is also a comment
```

Многострочные комментарии

Многострочные комментарии начинаются с `=` и с инструкцией `=cut` . Это специальные комментарии под названием POD (Обычная старая документация).

Любой текст между маркерами будет прокомментирован:

```
=begin comment

This is another comment.
And it spans multiple lines!

=end comment

=cut
```

Прочитайте Комментарии онлайн: https://riptutorial.com/ru/perl/topic/6815/комментарии

глава 10: Компиляция модуля cpan perl sapnwrfc из исходного кода

Вступление

Я хотел бы описать предварительные условия и шаги по созданию модуля sapnwrfc CPAN Perl с средой Strawberry Perl под Windows 7 x64. Он должен работать и для всех последующих версий Windows, таких как 8, 8.1 и 10.

Я использую Strawberry Perl 5.24.1.1 64 бит, но он также должен работать со старыми версиями.

Мне потребовалось несколько часов, чтобы преуспеть с несколькими попытками (32-битная установка 64-разрядной версии Perl, SAP NW RFC SDK, MinGW и компилятор Microsoft C). Поэтому я надеюсь, что некоторые из моих результатов получат пользу.

замечания

Установите текущий пакет с 64-битным пакетом Strawberry Perl с сайта http://strawberryperl.com . В моем случае это было 5.24.1.1.

Загрузите текущую версию пакета SAP NW RFC SDK x64 с https://launchpad.support.sap.com/#/softwarecenter

Вы можете найти его со следующей трассировкой: **Пакеты поддержки и патчи => По категориям => Дополнительные компоненты => SAP NW RFC SDK => SAP NW RFC SDK 7.20**

В моем случае текущая версия была 7.20 PL42 x64.

Извлеките загруженный файл с помощью `sapcar -xvf NWRFC_42-20004568.SAR`

Я переименовал папку в `C:\nwrfcsdk_x64`

Создайте файлы .def и .a для компилятора / компоновщика MinGW со следующими командами в каталоге C: \ nwrfcsdk_x64:

```
gendef *.dll
dlltool --dllname icuin34.dll --def icuin34.def --output-lib icuin34.a
dlltool --dllname icudt34.dll --def icudt34.def --output-lib icudt34.a
dlltool --dllname icuuc34.dll --def icuuc34.def --output-lib icuuc34.a
dlltool --dllname libsapucum.dll --def libsapucum.def --output-lib libsapucum.a
dlltool --dllname libicudecnumber.dll --def libicudecnumber.def --output-lib libicudecnumber.a
dlltool --dllname sapnwrfc.dll --def sapnwrfc.def --output-lib sapnwrfc.a
```

В dircectory C: \ nwrfcsdk_x64 \ lib должны существовать следующие файлы:

```
icudt34.a
icudt34.def
icudt34.dll
icuin34.a
icuin34.def
icuin34.dll
icuuc34.a
icuuc34.def
icuuc34.dll
libicudecnumber.a
libicudecnumber.def
libicudecnumber.dll
libsapucum.a
libsapucum.def
libsapucum.dll
libsapucum.lib
sapdecfICUlib.lib
sapnwrfc.a
sapnwrfc.def
sapnwrfc.dll
sapnwrfc.lib
```

Запустите командную строку с `cmd.exe` и запустите программу `cpan` .

Запустите команду `get sapnwrfc` чтобы загрузить модуль perl sapnwrfc из CPAN.

Оставьте среду cpan командой `exit` . Измените каталог на `C:\Strawberry\cpan\build\sapnwrfc-0.37-0` .

Создайте Makefile (ы) с помощью следующей команды. Адаптируйте имена папок в соответствии с настройками.

```
perl Makefile.PL --source=C:\nwrfcsdk_x64 --addlibs "C:\nwrfcsdk_x64\lib\sapnwrfc.a
C:\nwrfcsdk_x64\lib\libsapucum.a"
```

Запустите команды `dmake` и `dmake install` для сборки и установки модуля.

Скопируйте файлы из `C:\nwrfcsdk_x64\lib` в `C:\Strawberry\perl\site\lib\auto\SAPNW\Connection` .

Examples

Простой пример для проверки соединения RFC

Простой пример из http://search.cpan.org/dist/sapnwrfc/sapnwrfc-cookbook.pod

```
use strict;
use warnings;
use utf8;
use sapnwrfc;
```

```
SAPNW::Rfc->load_config('sap.yml');
my $conn = SAPNW::Rfc->rfc_connect;

my $rd = $conn->function_lookup("RPY_PROGRAM_READ");
my $rc = $rd->create_function_call;
$rc->PROGRAM_NAME("SAPLGRFC");

eval {
$rc->invoke;
};
if ($@) {
    die "RFC Error: $@\n";
}

print "Program name: ".$rc->PROG_INF->{'PROGNAME'}."\n";
my $cnt_lines_with_text = scalar grep(/LGRFCUXX/, map { $_->{LINE} } @{$rc->SOURCE_EXTENDED});
$conn->disconnect;
```

Прочитайте Компиляция модуля cpan perl sapnwrfc из исходного кода онлайн: https://riptutorial.com/ru/perl/topic/9775/компиляция-модуля-cpan-perl-sapnwrfc-из-исходного-кода

глава 11: Контрольные заявления

Examples

Conditionals

Perl поддерживает множество видов условных операторов (операторов, базирующихся на логических результатах). Наиболее распространенными условными утверждениями являются if-else, if и trernary statements. `given` операторы вводятся в виде коммутационной конструкции из C-производных языков и доступны в версиях Perl 5.10 и выше.

If-Else Statementments

Базовая структура if-утверждения выглядит так:

```
if (EXPR) BLOCK
if (EXPR) BLOCK else BLOCK
if (EXPR) BLOCK elsif (EXPR) BLOCK ...
if (EXPR) BLOCK elsif (EXPR) BLOCK ... else BLOCK
```

Для простых if-утверждений if может предшествовать или преуспеть в выполнении кода.

```
$number = 7;
if ($number > 4) { print "$number is greater than four!"; }

# Can also be written this way
print "$number is greater than four!" if $number > 4;
```

Loops

Perl поддерживает множество типов конструкций циклов: for / foreach, while / do-while и до.

```
@numbers = 1..42;
for (my $i=0; $i <= $#numbers; $i++) {
    print "$numbers[$i]\n";
}

#Can also be written as
foreach my $num (@numbers) {
    print "$num\n";
}
```

Цикл while вычисляет условное выражение *перед* выполнением соответствующего блока. Таким образом, иногда блок никогда не выполняется. Например, следующий код никогда не будет выполняться, если filehandle `$fh` был файловым дескриптором для пустого файла, или если он был уже исчерпан до условного.

```
while (my $line = readline $fh) {
    say $line;
}
```

С другой стороны, циклы `do` / `while` и `do` / `until` оценивают условное значение *после* каждого выполнения блока. Таким образом, цикл `do` / `while` или `do` / `until` всегда выполняется хотя бы один раз.

```
my $greeting_count = 0;
do {
    say "Hello";
    $greeting_count++;
} until ( $greeting_count > 1)

# Hello
# Hello
```

Прочитайте Контрольные заявления онлайн: https://riptutorial.com/ru/perl/topic/4896/контрольные-заявления

глава 12: Лучшие практики

Examples

Использование Perl :: Критика

Если вы хотите начать применять лучшие практики, для себя или своей команды, то Perl :: Critic - лучшее место для начала. Модуль основан на книге *Perl Best Practices* Дэмиена Конвей и довольно неплохо работает над воплощением предложений, сделанных в ней.

> **Примечание:** *я должен упомянуть (и сам Конвей говорит в книге), что это предложения. Я обнаружил, что в большинстве случаев книга содержит убедительные аргументы, хотя я, конечно, не согласен со всеми из них. Важно помнить, что, независимо от того, какую практику вы принимаете, вы остаетесь последовательными. Чем более предсказуемым будет ваш код, тем легче будет его поддерживать.*

Вы также можете попробовать Perl :: Critic через свой браузер на perlcritic.com .

Монтаж

```
cpan Perl::Critic
```

Это установит базовый набор правил и **perlcritic** скрипт, который можно вызвать из командной строки.

Основное использование

Документ CPAN для perlcritic содержит полную документацию, поэтому я буду использовать только самые распространенные варианты использования, чтобы вы начали. Основное использование - просто вызвать perlcritic в файле:

```
perlcritic -1 /path/to/script.pl
```

perlcritic работает как на скриптах, так и на модулях. **-1** относится к уровню серьезности правил, которые вы хотите запустить против скрипта. Существует пять уровней, которые соответствуют тому, насколько Perl :: Critic будет выделять ваш код.

-5 является самым нежным и будет предупреждать только о потенциально опасных проблемах, которые могут вызвать неожиданные результаты. **-1** является самым жестоким и будет жаловаться на вещи размером с ваш код, аккуратный или нет. По моему опыту,

соблюдение кода, соответствующего уровню 3, является достаточно хорошим, чтобы избежать опасности, не слишком мучаясь.

По умолчанию в любых отказах указывается причина и степень серьезности триггера правила:

```
perlcritic -3 --verbose 8 /path/to/script.pl

Debugging module loaded at line 16, column 1.  You've loaded Data::Dumper, which probably
shouln't be loaded in production.  (Severity: 4)
Private subroutine/method '_sub_name' declared but not used at line 58, column 1.  Eliminate
dead code.  (Severity: 3)
Backtick operator used at line 230, column 37.  Use IPC::Open3 instead.  (Severity: 3)
Backtick operator used at line 327, column 22.  Use IPC::Open3 instead.  (Severity: 3)
```

Просмотр политик

Вы можете быстро увидеть, какие правила запускаются и почему, используя опцию perlcritic **--verbose** :

Установка уровня до 8 покажет вам правило, вызвавшее предупреждение:

```
perlcritic -3 --verbose 8 /path/to/script.pl

[Bangs::ProhibitDebuggingModules] Debugging module loaded at line 16, column 1.  (Severity: 4)
[Subroutines::ProhibitUnusedPrivateSubroutines] Private subroutine/method '_sub_name' declared
but not used at line 58, column 1.  (Severity: 3)
[InputOutput::ProhibitBacktickOperators] Backtick operator used at line 230, column 37.
(Severity: 3)
[InputOutput::ProhibitBacktickOperators] Backtick operator used at line 327, column 22.
(Severity: 3)
```

Хотя уровень 11 покажет конкретные причины, по которым правило существует:

```
perlcritic -3 --verbose 11 /path/to/script.pl

Debugging module loaded at line 16, near 'use Data::Dumper;'.
  Bangs::ProhibitDebuggingModules (Severity: 4)
    This policy prohibits loading common debugging modules like the
    Data::Dumper manpage.

    While such modules are incredibly useful during development and
    debugging, they should probably not be loaded in production use. If this
    policy is violated, it probably means you forgot to remove a `use
    Data::Dumper;' line that you had added when you were debugging.
Private subroutine/method '_svn_revisions_differ' declared but not used at line 58, near 'sub
_sub_name {'.
  Subroutines::ProhibitUnusedPrivateSubroutines (Severity: 3)
    By convention Perl authors (like authors in many other languages)
    indicate private methods and variables by inserting a leading underscore
    before the identifier. This policy catches such subroutines which are
    not used in the file which declares them.
```

```
    This module defines a 'use' of a subroutine as a subroutine or method
    call to it (other than from inside the subroutine itself), a reference
    to it (i.e. `my $foo = \&_foo'), a `goto' to it outside the subroutine
    itself (i.e. `goto &_foo'), or the use of the subroutine's name as an
    even-numbered argument to `use overload'.
Backtick operator used at line 230, near 'my $filesystem_diff = join q{}, `diff
$trunk_checkout $staging_checkout`;'.
  InputOutput::ProhibitBacktickOperators (Severity: 3)
    Backticks are super-convenient, especially for CGI programs, but I find
    that they make a lot of noise by filling up STDERR with messages when
    they fail. I think its better to use IPC::Open3 to trap all the output
    and let the application decide what to do with it.

        use IPC::Open3 'open3';
        $SIG{CHLD} = 'IGNORE';

        @output = `some_command`;                       #not ok

        my ($writer, $reader, $err);
        open3($writer, $reader, $err, 'some_command'); #ok;
        @output = <$reader>;  #Output here
        @errors = <$err>;     #Errors here, instead of the console
Backtick operator used at line 327, near 'my $output = `$cmd`;'.
  InputOutput::ProhibitBacktickOperators (Severity: 3)
    Backticks are super-convenient, especially for CGI programs, but I find
    that they make a lot of noise by filling up STDERR with messages when
    they fail. I think its better to use IPC::Open3 to trap all the output
    and let the application decide what to do with it.

        use IPC::Open3 'open3';
        $SIG{CHLD} = 'IGNORE';

        @output = `some_command`;                       #not ok

        my ($writer, $reader, $err);
        open3($writer, $reader, $err, 'some_command'); #ok;
        @output = <$reader>;  #Output here
        @errors = <$err>;     #Errors here, instead of the console
```

Игнорирование кода

Будут случаи, когда вы не можете выполнить политику Perl :: Critic. В таких случаях вы можете добавить специальные комментарии, « ## **use crit ()** » и « ## **no crit** », вокруг вашего кода, чтобы заставить Perl :: Critic игнорировать их. Просто добавьте правила, которые вы хотите игнорировать в круглых скобках (кратные могут быть разделены запятой).

```
##no critic qw(InputOutput::ProhibitBacktickOperator)
my $filesystem_diff = join q{}, `diff $trunk_checkout $staging_checkout`;
## use critic
```

Обязательно оберните весь блок кода, или критик может не распознать инструкцию ignore.

```
## no critic (Subroutines::ProhibitExcessComplexity)
sub no_time_to_refactor_this {
    ...
}
## use critic
```

Обратите внимание, что существуют определенные политики, которые выполняются на уровне документа и не могут быть освобождены таким образом. Однако их можно отключить ...

Создание постоянных исключений

Использование ## no crit () хорошо, но по мере того, как вы начинаете применять стандарты кодирования, вы, вероятно, захотите сделать перманентные исключения для определенных правил. Вы можете сделать это, создав конфигурационный файл **.perlcriticrc** .

Этот файл позволит вам настроить не только те политики, которые выполняются, но и то, как они выполняются. Использование его так же просто, как размещение файла в вашем домашнем каталоге (в Linux, неуверенном, если это то же самое место в Windows). Или вы можете указать файл конфигурации при запуске команды с помощью опции **-profile** :

```
perlcritic -1 --profile=/path/to/.perlcriticrc /path/to/script.pl
```

Опять же, страница Perlcritic CPAN имеет полный список этих параметров. Я приведу несколько примеров из моего собственного файла конфигурации:

Применить основные настройки:

```
#very very harsh
severity = 1
color-severity-medium = bold yellow
color-severity-low = yellow
color-severity-lowest = bold blue
```

Отключите правило (обратите внимание на тире перед именем политики):

```
# do not require version control numbers
[-Miscellanea::RequireRcsKeywords]

# pod spelling is too over-zealous, disabling
[-Documentation::PodSpelling]
```

Изменение правила:

```
# do not require checking for print failure ( false positives for printing to stdout, not
filehandle )
[InputOutput::RequireCheckedSyscalls]
```

```
    functions = open close

# Allow specific unused subroutines for moose builders
[Subroutines::ProhibitUnusedPrivateSubroutines]
private_name_regex = _(?!build_)\w+
```

Заключение

Правильно используемая Perl :: Critic может быть бесценным инструментом, помогающим командам сохранять свою кодировку согласованной и легко поддерживаемой, независимо от того, какую политику вы используете.

Прочитайте Лучшие практики онлайн: https://riptutorial.com/ru/perl/topic/5919/лучшие-практики

глава 13: Обработка исключений

Examples

eval и die

Это встроенный способ справиться с «исключениями», не полагаясь на сторонние библиотеки, такие как Try :: Tiny .

```
my $ret;

eval {
  $ret = some_function_that_might_die();
  1;
} or do {
  my $eval_error = $@ || "Zombie error!";
  handle_error($eval_error);
};

# use $ret
```

Мы «злоупотребляем» тем фактом, что `die` имеет ложное возвращаемое значение, а возвращаемое значение общего кодового блока - это значение последнего выражения в кодовом блоке:

- если `$ret` назначено успешно, то `1;` выражение - последнее, что происходит в блоке `eval` code. Таким образом, блок кода `eval` имеет истинное значение, поэтому блок `or do` блок не запускается.
- если `some_function_that_might_die()` действительно `die` , то последнее, что происходит в блоке `eval` code, является `die` . Таким образом, блок `eval` кода имеет ложное значение, и блок `or do` блок выполняет его.
- Первое, что вы *должны* сделать в блоке `or do` блоке, - читать `$@` . Эта глобальная переменная будет содержать любой аргумент, который передается `die` . `|| "Zombie Error"` популярен, но не нужен в общем случае.

Это важно понять, потому что некоторые не все коды не срабатывают, вызывая die, но та же структура может использоваться независимо. Рассмотрим функцию базы данных, которая возвращает:

- количество строк, повлиявших на успех
- `'0 but true'` если запрос успешный, но ни одна строка не была затронута
- `0` если запрос не был успешным.

В этом случае вы все равно можете использовать ту же идиому, но вам *нужно* пропустить последний `1;` , И эта функция *должна* быть последней вещью в Eval. Что-то вроде этого:

```
eval {
  my $value = My::Database::retrieve($my_thing); # dies on fail
  $value->set_status("Completed");
  $value->set_completed_timestamp(time());
  $value->update(); # returns false value on fail
} or do { # handles both the die and the 0 return value
  my $eval_error = $@ || "Zombie error!";
  handle_error($eval_error);
};
```

Прочитайте Обработка исключений онлайн: https://riptutorial.com/ru/perl/topic/1894/обработка-исключений

глава 14: Объектно-ориентированный Perl

Examples

Создание объектов

В отличие от многих других языков, Perl не имеет конструкторов, которые выделяют память для ваших объектов. Вместо этого следует написать метод класса, который создает структуру данных и заполняет их данными (вы можете знать это как шаблон проектирования Factory Method).

```
package Point;
use strict;

sub new {
    my ($class, $x, $y) = @_;
    my $self = { x => $x, y => $y }; # store object data in a hash
    bless $self, $class;             # bind the hash to the class
    return $self;
}
```

Этот метод можно использовать следующим образом:

```
my $point = Point->new(1, 2.5);
```

Всякий раз, когда оператор стрелки `->` используется с методами, его левый операнд добавляется к данному списку аргументов. Итак, `@_` в `new` будет содержать значения `('Point', 1, 2.5)` .

Там нет ничего особенного во имя `new` . Вы можете вызывать заводские методы, как вы предпочитаете.

В хешах нет ничего особенного. Вы можете сделать то же самое следующим образом:

```
package Point;
use strict;

sub new {
    my ($class, @coord) = @_;
    my $self = \@coord;
    bless $self, $class;
    return $self;
}
```

В общем, любая ссылка может быть объектом, даже скалярной ссылкой. Но чаще всего хеши являются наиболее удобным способом представления данных объекта.

Определение классов

В общем, классы в Perl - это просто пакеты. Они могут содержать данные и методы, как обычные пакеты.

```
package Point;
use strict;

my $CANVAS_SIZE = [1000, 1000];

sub new {
    ...
}

sub polar_coordinates {
    ...
}

1;
```

Важно отметить, что переменные, объявленные в пакете, являются переменными класса, а не переменными объекта (экземпляра). Изменение переменной уровня пакета влияет на все объекты класса. Как хранить данные, специфичные для объекта, см. В разделе «Создание объектов».

Что делает пакеты классов конкретными, это оператор стрелки `->` . Он может использоваться после простого слова:

```
Point->new(...);
```

или после скалярной переменной (обычно содержащей ссылку):

```
my @polar = $point->polar_coordinates;
```

То, что слева от стрелки, добавляется к данному списку аргументов метода. Например, после вызова

```
Point->new(1, 2);
```

array `@_` в `new` будет содержать три аргумента: `('Point', 1, 2)` .

Пакеты, представляющие классы, должны учитывать это соглашение и ожидать, что все их методы будут иметь один дополнительный аргумент.

Разрешение наследования и методы

Чтобы сделать класс подклассом другого класса, используйте `parent` прагму:

```
package Point;
use strict;
...
1;

package Point2D;
use strict;
use parent qw(Point);
...
1;

package Point3D;
use strict;
use parent qw(Point);
...
1;
```

Perl допускает множественное наследование:

```
package Point2D;
use strict;
use parent qw(Point PlanarObject);
...
1;
```

Наследование - это все о разрешении, метод которого следует вызывать в конкретной ситуации. Поскольку чистый Perl не устанавливает никаких правил о структуре данных, используемых для хранения данных объекта, наследование не имеет к этому никакого отношения.

Рассмотрим следующую иерархию классов:

```
package GeometryObject;
use strict;

sub transpose { ...}

1;

package Point;
use strict;
use parent qw(GeometryObject);

sub new { ... };

1;

package PlanarObject;
use strict;
use parent qw(GeometryObject);

sub transpose { ... }

1;

package Point2D;
use strict;
```

```
use parent qw(Point PlanarObject);

sub new { ... }

sub polar_coordinates { ... }

1;
```

Разрешение метода работает следующим образом:

1. Начальная точка определяется левым операндом оператора стрелки.

    - Если это голое слово:

      ```
      Point2D->new(...);
      ```

      ... или скалярная переменная, содержащая строку:

      ```
      my $class = 'Point2D';
      $class->new(...);
      ```

      ... тогда отправной точкой является пакет с соответствующим именем (`Point2D` в обоих примерах).

    - Если левый операнд является скалярной переменной, содержащей *блаженную* ссылку:

      ```
      my $point = {...};
      bless $point, 'Point2D'; # typically, it is encapsulated into class methods
      my @coord = $point->polar_coordinates;
      ```

      то отправной точкой является класс ссылки (опять же, `Point2D`). Оператор стрелки не может использоваться для вызова методов для *беспроблемных* ссылок.

2. Если начальная точка содержит требуемый метод, она просто называется.

    Таким образом, поскольку `Point2D::new` существует,

    ```
    Point2D->new(...);
    ```

    просто назовут его.

3. Если начальная точка не содержит требуемого метода, выполняется поиск по глубине в `parent` классах. В приведенном выше примере порядок поиска будет выглядеть следующим образом:

    - `Point2D`
    - `Point` (первый родитель `Point2D`)

- `GeometryObject` (родительский элемент `Point`)
- `PlanarObject` (второй родитель `Point2D`)

Например, в следующем коде:

```
my $point = Point2D->new(...);
$point->transpose(...);
```

метод, который будет называться, это `GeometryObject::transpose` , хотя он будет переопределен в `PlanarObject::transpose` .

4. Вы можете установить начальную точку явно.

   В предыдущем примере вы можете явно вызвать `PlanarObject::transpose` следующим образом:

```
my $point = Point2D->new(...);
$point->PlanarObject::transpose(...);
```

5. Аналогичным образом `SUPER::` выполняет поиск метода в родительских классах текущего класса.

   Например,

```
package Point2D;
use strict;
use parent qw(Point PlanarObject);

sub new {
    (my $class, $x, $y) = @_;
    my $self = $class->SUPER::new;
    ...
}

1;
```

вызовет `Point::new` в ходе выполнения `Point2D::new` .

Методы класса и объекта

В Perl разница между методами class (static) и object (instance) не так сильна, как на некоторых других языках, но она все еще существует.

Левый операнд оператора стрелки `->` становится первым аргументом вызываемого метода. Это может быть строка:

```
# the first argument of new is string 'Point' in both cases
Point->new(...);

my $class = 'Point';
```

```
$class->new(...);
```

или ссылку на объект:

```
# reference contained in $point is the first argument of polar_coordinates
my $point = Point->new(...);
my @coord = $point->polar_coordinates;
```

Методы класса - это только те, которые ожидают, что их первый аргумент будет строкой, а объектные методы - это те, которые ожидают, что их первый аргумент будет ссылкой на объект.

Обычно методы класса не делают ничего с их первым аргументом, который является просто именем класса. Как правило, он используется только самим Perl для разрешения метода. Поэтому типичный метод класса можно вызвать и для объекта:

```
my $width = Point->canvas_width;

my $point = Point->new(...);
my $width = $point->canvas_width;
```

Хотя этот синтаксис разрешен, он часто вводит в заблуждение, поэтому лучше его избегать.

Методы объектов получают ссылку на объект в качестве первого аргумента, поэтому они могут обращаться к данным объекта (в отличие от методов класса):

```
package Point;
use strict;

sub polar_coordinates {
    my ($point) = @_;
    my $x = $point->{x};
    my $y = $point->{y};
    return (sqrt($x * $x + $y * $y), atan2($y, $x));
}

1;
```

Тот же метод может отслеживать оба случая: когда он вызывается как класс или метод объекта:

```
sub universal_method {
    my $self = shift;
    if (ref $self) {
        # object logic
        ...
    }
    else {
        # class logic
        ...
    }
```

```
}
```

Определение классов в современном Perl

Хотя доступный, определение класса с нуля не рекомендуется в современном Perl. Используйте одну из вспомогательных OO-систем, которые предоставляют больше возможностей и удобства. Среди этих систем:

- `Moose` - вдохновленный дизайном Perl 6 OO
- `Class::Accessor` - легкая альтернатива Moose
- `Class::Tiny` - действительно минимальный класс строитель

лось

```
package Foo;
use Moose;

has bar => (is => 'ro');                 # a read-only property
has baz => (is => 'rw', isa => 'Bool');  # a read-write boolean property

sub qux {
    my $self = shift;
    my $barIsBaz = $self->bar eq 'baz';  # property getter
    $self->baz($barIsBaz);               # property setter
}
```

Класс :: Accessor (синтаксис Moose)

```
package Foo;
use Class::Accessor 'antlers';

has bar => (is => 'ro');                 # a read-only property
has baz => (is => 'rw', isa => 'Bool');  # a read-write property (only 'is' supported, the
type is ignored)
```

Класс :: Accessor (собственный синтаксис)

```
package Foo;
use base qw(Class::Accessor);

Foo->mk_accessors(qw(bar baz));  # some read-write properties
Foo->mk_accessors(qw(qux));      # a read-only property
```

Класс :: Крошечный

```
package Foo;
use Class::Tiny qw(bar baz);  # just props
```

Роли

Роль в Perl в основном

- набор методов и атрибутов, которые
- вводится в класс напрямую.

Роль обеспечивает часть функциональности , которая может *состоять* в (или *приложенной* к) любой класс (который , как говорят *потреблять* роль). Роль не может быть унаследована, но может быть использована другой ролью.

Роль может также *потребовать,* чтобы классы потребления реализовали некоторые методы вместо реализации самих методов (как и интерфейсы на Java или C #).

Perl не имеет встроенной поддержки ролей, но есть классы CPAN, которые предоставляют такую поддержку.

Moose :: Роль

```
package Chatty;
use Moose::Role;

requires 'introduce';  # a method consuming classes must implement

sub greet {            # a method already implemented in the role
    print "Hi!\n";
}


package Parrot;
use Moose;

with 'Chatty';

sub introduce {
    print "I'm Buddy.\n";
}
```

Роль :: Крошечный

Используйте, если ваша система ОО не обеспечивает поддержку ролей (например, `Class::Accessor` или `Class::Tiny`). Не поддерживает атрибуты.

```
package Chatty;
use Role::Tiny;

requires 'introduce';  # a method consuming classes must implement

sub greet {            # a method already implemented in the role
    print "Hi!\n";
}

package Parrot;
use Class::Tiny;
use Role::Tiny::With;
```

```
with 'Chatty';

sub introduce {
    print "I'm Buddy.\n";
}
```

Прочитайте Объектно-ориентированный Perl онлайн: https://riptutorial.com/ru/perl/topic/2920/объектно-ориентированный-perl

глава 15: Оптимизация использования памяти

Examples

Чтение файлов: foreach и while

При чтении потенциально большой файл, будет в `while` цикл имеет значительное преимущество над памятью `foreach` . Следующее будет читать запись файла по записи (по умолчанию «запись» означает «строка», как указано в `$/`), присваивая каждому значение `$_` мере чтения:

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

Алмазный оператор делает здесь немного волшебства, чтобы убедиться, что цикл завершается только в конце файла, а не на линиях, содержащих только символ «0».

Следующий цикл, похоже, работает одинаково, однако он оценивает оператора алмаза в контексте списка, заставляя весь файл читать за один раз:

```
foreach(<$fh>) {
    print;
}
```

Если вы работаете в одной записи одновременно, это может привести к огромной трате памяти, и поэтому ее следует избегать.

Обработка длинных списков

Если у вас уже есть список в памяти, простой и обычно достаточный способ его обработки - простой цикл `foreach` :

```
foreach my $item (@items) {
    ...
}
```

Это нормально, например, для обычного случая выполнения некоторой обработки на `$item` а затем записывать его в файл без сохранения данных. Тем не менее, если вы строите некоторые другую структуру данных из элементов, в `while` цикл меньше памяти:

```
my @result;
while(@items) {
```

```
    my $item = shift @items;
    push @result, process_item($item);
}
```

Если ссылка на `$item` напрямую не попадает в ваш список результатов, элементы, которые вы `@items` массивом `@items` могут быть освобождены, а память будет повторно использована интерпретатором при вводе следующей итерации цикла.

Прочитайте Оптимизация использования памяти онлайн:
https://riptutorial.com/ru/perl/topic/6327/оптимизация-использования-памяти

глава 16: Отладка скриптов Perl

Examples

Запустить сценарий в режиме отладки

Чтобы запустить сценарий в режиме отладки, вы должны добавить параметр `-d` в командной строке:

```
$perl -d script.pl
```

Если `t` указано, это указывает отладчику, что потоки будут использоваться в отлаживаемом коде:

```
$perl -dt script.pl
```

Дополнительная информация на `perldoc` PerlRun

Используйте нестандартный отладчик

`$perl -d:MOD script.pl` запускает программу под управлением модуля отладки, профилирования или трассировки, установленного как `Devel::MOD` .

Например, `-d:NYTProf` выполняет программу с использованием профилировщика `Devel::NYTProf` .

Просмотреть все доступные модули Devel здесь

Рекомендуемые модули:

- `Devel::NYTProf` - Мощный быстрый многофункциональный Perl-исходник
- `Devel::Trepan` - модульный gdb-подобный отладчик Perl
- `Devel::MAT` - инструмент анализа памяти Perl
- `Devel::hdb` - отладчик Perl как веб-страница и служба REST
- `Devel::DebugHooks::KillPrint` - позволяет забыть об отладке с помощью инструкции `print`
- `Devel::REPL` - современная интерактивная оболочка perl
- `Devel::Cover` - Показатели покрытия кода для Perl

Прочитайте Отладка скриптов Perl онлайн: https://riptutorial.com/ru/perl/topic/6769/отладка-скриптов-perl

глава 17: Отладочный вывод

Examples

Демпинговые данные-структуры

```
use Data::Dumper;

my $data_structure = { foo => 'bar' };
print Dumper $data_structure;
```

Использование Data :: Dumper - это простой способ взглянуть на структуры данных или содержимое переменной во время выполнения. Он поставляется с Perl, и вы можете легко загрузить его. Функция `Dumper` возвращает структуру данных, сериализованную таким образом, чтобы она выглядела как Perl-код.

```
$VAR1 = {
        'foo' => 'bar',
}
```

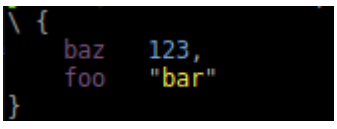
Это очень полезно для быстрого просмотра некоторых значений в вашем коде. Это один из самых полезных инструментов в вашем арсенале. Прочтите полную документацию по metacpan .

Сбрасывание со стилем

Иногда Data :: Dumper недостаточно. Получил объект лося, который вы хотите проверить? Огромные номера той же структуры? Хотите, чтобы материал отсортировался? Цветные? Данные :: Принтер - ваш друг.

```
use Data::Printer;

p $data_structure;
```



Данные :: Принтер записывает в STDERR, как `warn` . Это облегчает поиск выходных данных. По умолчанию он сортирует хеш-ключи и смотрит на объекты.

```
use Data::Printer;
use LWP::UserAgent;

my $ua = LWP::UserAgent->new;
p $ua;
```

Он будет рассматривать все методы объекта, а также список внутренних элементов.

```
LWP::UserAgent  {
    Parents       LWP::MemberMixin
    public methods (45) : add_handler, agent, clone, conn_cache, cookie_jar, credentials,
default_header, default_headers, delete, env_proxy, from, get, get_basic_credentials,
get_my_handler, handlers, head, is_online, is_protocol_supported, local_address, max_redirect,
max_size, mirror, new, no_proxy, parse_head, post, prepare_request, progress,
protocols_allowed, protocols_forbidden, proxy, put, redirect_ok, remove_handler, request,
requests_redirectable, run_handlers, send_request, set_my_handler, show_progress,
simple_request, ssl_opts, timeout, use_alarm, use_eval
    private methods (4) : _agent, _need_proxy, _new_response, _process_colonic_headers
    internals: {
        def_headers             HTTP::Headers,
        handlers                {
            response_header   HTTP::Config
        },
        local_address           undef,
        max_redirect            7,
        max_size                undef,
        no_proxy                [],
        protocols_allowed       undef,
        protocols_forbidden     undef,
        proxy                   {},
        requests_redirectable   [
            [0] "GET",
            [1] "HEAD"
        ],
        show_progress           undef,
        ssl_opts                {
            verify_hostname   1
        },
        timeout                 180,
        use_eval                1
    }
}
```

Вы можете настроить его дальше, поэтому он сериализует определенные объекты определенным образом или включает объекты до произвольной глубины. Полная конфигурация доступна в документации .

К сожалению, Data :: Printer не поставляется с Perl, поэтому вам нужно установить его из CPAN или через систему управления пакетами.

Список удаленных массивов

```
my @data_array = (123, 456, 789, 'poi', 'uyt', "rew", "qas");
print Dumper @data_array;
```

Использование Data :: Dumper дает легкий доступ к значениям списка извлечения. `Dumper` возвращает значения списка, сериализованные таким образом, который выглядит как код Perl.

Выход:

```
$VAR1 = 123;
$VAR2 = 456;
$VAR3 = 789;
$VAR4 = 'poi';
$VAR5 = 'uyt';
$VAR6 = 'rew';
$VAR7 = 'qas';
```

Как было предложено пользователем @dgw При сбрасывании массивов или хешей лучше использовать ссылку на массив или хеш-ссылку, они будут лучше отображаться на входе.

```
$ref_data = [23,45,67,'mnb','vcx'];
print Dumper $ref_data;
```

Выход:

```
$VAR1 = [
          23,
          45,
          67,
          'mnb',
          'vcx'
        ];
```

Вы можете также ссылаться на массив при печати.

```
my @data_array = (23,45,67,'mnb','vcx');
print Dumper \@data_array;
```

Выход:

```
$VAR1 = [
          23,
          45,
          67,
          'mnb',
          'vcx'
        ];
```

Данные показывают

Функция `show` автоматически экспортируется при `use Data::Show;` выполняется. Эта функция принимает переменную в качестве единственного аргумента и выводит:

1. имя переменной
2. содержимое этой переменной (в читаемом формате)
3. строка `show` файла запускается из
4. `show` файла запускается из

Предполагая, что следующий код из файла `example.pl` :

```
use strict;
use warnings;
use Data::Show;

my @array = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

my %hash  = ( foo => 1, bar => { baz => 10, qux => 20 } );

my $href = \%hash;

show @array;
show %hash;
show $href;
```

`perl example.pl` дает следующий результат:

```
======(  @array  )======================[ 'example.pl', line 11 ]======

    [1 .. 10]


======(  %hash  )=======================[ 'example.pl', line 12 ]======

    { bar => { baz => 10, qux => 20 }, foo => 1 }


======(  $href  )=======================[ 'example.pl', line 13 ]======

    { bar => { baz => 10, qux => 20 }, foo => 1 }
```

См. Документацию для `Data::Show` .

Прочитайте Отладочный вывод онлайн: https://riptutorial.com/ru/perl/topic/983/отладочный-вывод

глава 18: Пакеты и модули

Синтаксис

- требуется Module :: Name; # Требовать по имени от @INC
- требуется «путь / в / файл.pm»; # Требовать относительный путь от @INC
- используйте Module :: Name; # require и импорт по умолчанию на BEGIN
- используйте Module :: Name (); # требуется и не импортировать в BEGIN
- используйте Module :: Name (@ARGS); # требуется и импортировать с помощью args в BEGIN
- use Module :: Name VERSION; # require, проверка версии и импорт по умолчанию в BEGIN
- используйте Module :: Name VERSION (); # require, проверка версии и отсутствие импорта в BEGIN
- используйте Module :: Name VERSION (@ARGS); # require, проверка версии, импорт с args в BEGIN
- do "path / to / file.pl"; # загрузить и eval данный файл

Examples

Выполнение содержимого другого файла

```
do './config.pl';
```

Это прочитает содержимое файла config.pl и выполнит его. (См. Также: `perldoc -f do` .)

NB: Избегайте `do` если `do` не играете в гольф или что-то еще, поскольку проверка ошибок отсутствует. Для включения библиотеки модулей, использование `require` или `use` .

Загрузка модуля во время выполнения

```
require Exporter;
```

Это обеспечит загрузку модуля `Exporter` во время выполнения, если он еще не был импортирован. (См. Также: `perldoc -f require` .)

NB: Большинство пользователей должны `use` модули, а не `require` их. В отличие от `use` , `require` не вызывает метод импорта модуля и выполняется во время выполнения, а не во время компиляции.

Этот способ загрузки модулей полезен, если вы не можете решить, какие модули вам нужны до запуска, например, с помощью системы плагинов:

```
package My::Module;
my @plugins = qw( One Two );
foreach my $plugin (@plugins) {
    my $module = __PACKAGE__ . "::Plugins::$plugin";
    $module =~ s!::!/!g;
    require "$module.pm";
}
```

Это попытается загрузить `My::Package::Plugins::One` и `My::Package::Plugins::Two` . `@plugins` должен, конечно, поступать из некоторого пользовательского ввода или конфигурационного файла, чтобы это имело смысл. Обратите внимание на оператор подстановки `s!::!/!g` который заменяет каждую пару двоеточий косой чертой. Это связано с тем, что вы можете загружать модули с использованием синтаксиса знакомого имени модуля из `use` только если имя модуля является делом. Если вы передаете строку или переменную, она должна содержать имя файла.

Использование модуля

```
use Cwd;
```

Это будет импортировать модуль `Cwd` во время компиляции и импортировать его символы по умолчанию, то есть сделать некоторые из переменных и функций модуля доступными для кода, используя его. (См. Также: `perldoc -f use` .)

Как правило, это будет правильно. Иногда, однако, вам нужно будет контролировать, какие символы импортированы. Добавьте список символов после имени модуля для экспорта:

```
use Cwd 'abs_path';
```

Если вы это сделаете, будут импортированы только указанные вами символы (т. Е. Набор по умолчанию не будет импортирован).

При импорте нескольких символов идиоматично использовать конструкцию построения `qw()` :

```
use Cwd qw(abs_path realpath);
```

Некоторые модули экспортируют подмножество своих символов, но можно сказать, что они экспортируют все с помощью `:all` :

```
use Benchmark ':all';
```

(Обратите внимание, что не все модули распознают или используют тег `:all`).

Использование модуля внутри каталога

```
use lib 'includes';
use MySuperCoolModule;
```

`use lib 'includes';` добавляет относительный каталог, `includes/` как другой путь поиска модуля в `@INC` . Предположим, что у вас есть файл модуля `MySyperCoolModule.pm` внутри `includes/` , который содержит:

```
package MySuperCoolModule;
```

Если вы хотите, вы можете сгруппировать столько модулей из своего собственного внутри одного каталога и сделать их доступными с помощью одного оператора `use lib` .

На этом этапе использование подпрограмм в модуле потребует префикса имени подпрограммы с именем пакета:

```
MySuperCoolModule::SuperCoolSub_1("Super Cool String");
```

Чтобы иметь возможность использовать подпрограммы без префикса, вам нужно экспортировать имена подпрограмм, чтобы они были распознаны программой, вызывающей их. Экспортирование может быть настроено так, чтобы быть автоматическим, таким образом:

```
package MySuperCoolModule;
use base 'Exporter';
our @EXPORT = ('SuperCoolSub_1', 'SuperCoolSub_2');
```

Затем в файле, который `use` этот модуль, эти подпрограммы будут автоматически доступны:

```
use MySuperCoolModule;
SuperCoolSub_1("Super Cool String");
```

Или вы можете настроить модуль для условного экспорта подпрограмм, таким образом:

```
package MySuperCoolModule;
use base 'Exporter';
our @EXPORT_OK = ('SuperCoolSub_1', 'SuperCoolSub_2');
```

В этом случае вам нужно явно запросить нужные подпрограммы для экспорта в скрипт, который `use` модуль:

```
use MySuperCoolModule 'SuperCoolSub_1';
SuperCoolSub_1("Super Cool String");
```

CPAN.pm

`CPAN.pm` - это модуль Perl, который позволяет запрашивать и устанавливать модули с сайтов

CPAN.

Он поддерживает интерактивный режим, вызываемый с помощью

```
cpan
```

или же

```
perl -MCPAN -e shell
```

Запросы модулей

По имени:

```
cpan> m MooseX::YAML
```

Режимом по имени модуля:

```
cpan> m /^XML::/
```

Примечание: для включения пейджера или перенаправления на файл используйте `|` *или* `>` *перенаправление оболочки (пробелы являются обязательными в пределах* `|` *и* `>` *), например:* `m /^XML::/ | less` *.*

По распределению:

```
cpan> d LMC/Net-Squid-Auth-Engine-0.04.tar.gz
```

Установка модулей

По имени:

```
cpan> install MooseX::YAML
```

По распределению:

```
cpan> install LMC/Net-Squid-Auth-Engine-0.04.tar.gz
```

Список всех установленных модулей

Из командной строки:

```
cpan -l
```

Из сценария Perl:

```
use ExtUtils::Installed;
my $inst = ExtUtils::Installed->new();
my @modules = $inst->modules();
```

Прочитайте Пакеты и модули онлайн: https://riptutorial.com/ru/perl/topic/451/пакеты-и-модули

глава 19: переменные

Синтаксис

- моя # Лексическая декларация
- наша глобальная декларация #
- $ foo # Скаляр
- @foo # Массив
- $ # foo # Массив Last-Index
- % foo # Хеш
- $ {$ foo} # Scalar De-Reference
- @ {$ foo} # Array De-Reference
- $ # {$ foo} # Array-DeRef Last-Index
- % {$ foo} # Hash De-Reference
- $ foo [$ index] # Массив индексируется
- $ {$ foo} [$ index] # Array De-Reference и индексируется.
- $ foo -> [$ index] # Array De-Reference и индексироваться (упрощено)
- $ foo {$ key} # Хеш получить значение для ключа
- $ {$ foo} {$ key} # Hash Dereference и получить значение для ключа
- $ foo -> {$ key} # Hash Dereference и получить значение для ключа (упрощенного)
- \ $ x # Ссылка на скаляр
- \ @x # Ссылка на массив
- \% x # Ссылка на хэш
- = [] # Ссылка на анонимный массив (Inline)
- = {} # Ссылка на анонимный хэш (Inline)

Examples

Скаляры

Скаляры - это самый базовый тип данных Perl. Они отмечены сигилой `$` и содержат одно значение одного из трех типов:

- **число** (`3` , `42` , `3.141` и т. д.),
- **строка** (`'hi'` , `"abc"` и т. д.)
- **ссылка** на переменную (см. другие примеры).

```
my $integer = 3;                    # number
my $string = "Hello World";         # string
my $reference = \$string;           # reference to $string
```

Perl конвертирует между числами и строками «на лету» , исходя из того, что ожидает

конкретный оператор.

```
my $number = '41';                    # string '41'
my $meaning = $number + 1;            # number  42
my $sadness = '20 apples';            # string '20 apples'
my $danger = $sadness * 2;            # number '40', raises warning
```

При преобразовании строки в число Perl берет столько цифр от строки, сколько может, поэтому `20 apples` преобразуются в `20` в последней строке.

Исходя из того, хотите ли вы обрабатывать содержимое скаляра в виде строки или числа, вам нужно использовать разные операторы. Не смешивайте их.

```
# String comparison                   # Number comparison
'Potato' eq 'Potato';                 42 == 42;
'Potato' ne 'Pomato';                 42 != 24;
'Camel'  lt 'Potato';                 41 < 42;
'Zombie' gt 'Potato';                 43 > 42;

# String concatenation                # Number summation
'Banana' . 'phone';                   23 + 19;

# String repetition                   # Number multiplication
'nan' x 3;                            6 * 7;
```

Попытка использования строковых операций над номерами не вызывает предупреждений; попытка использовать числовые операции для нечисловых строк. Имейте в виду, что некоторые нецифровые строки, такие как `'inf'` , `'nan'` , `'0 but true'` считаются цифрами.

Массивы

Массивы хранят упорядоченную последовательность значений. Вы можете получить доступ к содержимому по индексу или перебрать их. Значения будут оставаться в том порядке, в котором вы их заполняли.

```
my @numbers_to_ten = (1,2,3,4,5,6,7,8,9,10); # More conveniently: (1..10)
my @chars_of_hello = ('h','e','l','l','o');
my @word_list = ('Hello','World');

# Note the sigil: access an @array item with $array[index]
my $second_char_of_hello = $chars_of_hello[1]; # 'e'

# Use negative indices to count from the end (with -1 being last)
my $last_char_of_hello = $chars_of_hello[-1];

# Assign an array to a scalar to get the length of the array
my $length_of_array = @chars_of_hello; # 5

# You can use $# to get the last index of an array, and confuse Stack Overflow
my $last_index_of_array = $#chars_of_hello; # 4

# You can also access multiple elements of an array at the same time
# This is called "array slice"
```

```
# Since this returns multiple values, the sigil to use here on the RHS is @
my @some_chars_of_hello = @chars_of_hello[1..3]; # ('H', 'e', 'l')
my @out_of_order_chars = @chars_of_hello[1,4,2]; # ('e', 'o', 'l')

# In Python you can say array[1:-1] to get all elements but first and last
# Not so in Perl: (1..-1) is an empty list. Use $# instead
my @empty_list = @chars_of_hello[1..-1];                          # ()
my @inner_chars_of_hello = @chars_of_hello[1..$#chars_of_hello-1]; # ('e','l','l')

# Access beyond the end of the array yields undef, not an error
my $undef = $chars_of_hello[6]; # undef
```

Массивы изменяемы:

```
use utf8; # necessary because this snippet is utf-8
$chars_of_hello[1] = 'u';              #     ('h','u','l','l','o')
push @chars_of_hello, ('!', '!');      #     ('h','u','l','l','o','!','!')
pop @chars_of_hello;                   #     ('h','u','l','l','o','!')
shift @chars_of_hello;                 #         ('u','l','l','o','!')
unshift @chars_of_hello, ('¡', 'H');   # ('¡','H','u','l','l','o','!')
@chars_of_hello[2..5] = ('O','L','A'); # ('¡','H','O','L','A',undef,'!') whoops!
delete $chars_of_hello[-2];            # ('¡','H','O','L','A',      '!')

# Setting elements beyond the end of an array does not result in an error
# The array is extended with undef's as necessary. This is "autovivification."
my @array;           # ()
my @array[3] = 'x';  # (undef, undef, undef, 'x')
```

Наконец, вы можете перебрать содержимое массива:

```
use v5.10; # necessary for 'say'
for my $number (@numbers_to_ten) {
  say $number ** 2;
}
```

При использовании в качестве булевых элементов массивы являются истинными, если они не пусты.

Хэш

Хэши можно понимать как таблицы поиска. Вы можете получить доступ к его содержимому, указав ключ для каждого из них. Ключи должны быть строками. Если это не так, они будут преобразованы в строки.

Если вы дадите хэш просто известный ключ, он будет служить вам вашей ценности.

```
# Elements are in (key, value, key, value) sequence
my %inhabitants_of = ("London", 8674000, "Paris", 2244000);

# You can save some typing and gain in clarity by using the "fat comma"
# syntactical sugar. It behaves like a comma and quotes what's on the left.
my %translations_of_hello = (spanish => 'Hola', german => 'Hallo', swedish => 'Hej');
```

В следующем примере обратите внимание на скобки и сигилы: вы получаете доступ к элементу `%hash` с помощью `$hash{key}` потому что *значение, которое* вы хотите, является скаляром. Некоторые считают хорошей практикой процитировать ключ, в то время как другие находят этот стиль визуально шумным. Цитирование требуется только для ключей, которые могут быть ошибочно приняты за выражения типа `$hash{'some-key'}`

```
my $greeting = $translations_of_hello{'spanish'};
```

В то время как Perl по умолчанию будет пытаться использовать просто слова как строки, `+` модификатор также может использоваться для указания Perl, что ключ не должен быть интерполирован, а выполнен с результатом выполнения, используемым в качестве ключа:

```
my %employee = ( name => 'John Doe', shift => 'night' );
# this example will print 'night'
print $employee{shift};

# but this one will execute [shift][1], extracting first element from @_,
# and use result as a key
print $employee{+shift};
```

Подобно массивам, вы можете одновременно получить доступ к нескольким элементам хэша. Это называется *хэш-срезом* . Результирующее значение - это список, поэтому используйте `@` sigil:

```
my @words = @translations_of_hello{'spanish', 'german'}; # ('Hola', 'Hallo')
```

Итерации по клавишам хэша с `keys keys` возвращают элементы в произвольном порядке. Совместимый с `sort` , если вы хотите.

```
for my $lang (sort keys %translations_of_hello) {
  say $translations_of_hello{$lang};
}
```

Если вам действительно не нужны такие ключи, как в предыдущем примере, `values` возвращают `values` хеша напрямую:

```
for my $translation (values %translations_of_hello) {
  say $translation;
}
```

Вы также можете использовать цикл while, `each` должен перебирать хэш. Таким образом, вы получите как ключ, так и значение в одно и то же время без отдельного поиска значений. Его использование, однако, обескураживает, так как `each` может нарушить путаницу.

```
# DISCOURAGED
while (my ($lang, $translation) = each %translations_of_hello) {
  say $translation;
```

```
}
```

Доступ к неустановленным элементам возвращает undef, а не ошибку:

```
my $italian = $translations_of_hello{'italian'}; # undef
```

`map` и list flattening можно использовать для создания хэшей из массивов. Это популярный способ создания «набора» значений, например, чтобы быстро проверить, находится ли значение в `@elems` . Эта операция обычно занимает время O (n) (т. Е. Пропорционально количеству элементов), но может выполняться в постоянное время (O (1)) путем превращения списка в хэш:

```
@elems = qw(x y x z t);
my %set = map { $_ => 1 } @elems;   # (x, 1, y, 1, t, 1)
my $y_membership = $set{'y'};       # 1
my $w_membership = $set{'w'};       # undef
```

Это требует некоторого объяснения. Содержимое `@elems` считывается в список, который обрабатывается `map` . `map` принимает блок кода, который вызывается для каждого значения его входного списка; значение элемента доступно для использования в `$_` . Наш кодовый блок возвращает *два* элемента списка для каждого элемента ввода: `$_` , входной элемент и `1` , только некоторое значение. Как только вы учитываете сглаживание списка, результатом является то, что `map { $_ => 1 } @elems` превращает `qw(xyxzt)` в `(x => 1, y => 1, x => 1, z => 1, t => 1)` .

Поскольку эти элементы назначаются в хеш, нечетные элементы становятся хеш-ключами, а даже элементы становятся хеш-значениями. Когда ключ указан несколько раз в списке, который должен быть присвоен хэшу, выигрывает *последнее* значение. Это эффективно отменяет дубликаты.

Более быстрый способ превратить список в хэш использует назначение хэш-фрагмента. Он использует оператор `x` чтобы умножить один элементный список `(1)` на размер `@elems` , поэтому для каждого из ключей в срезе с левой стороны есть `1` значение:

```
@elems = qw(x y x z t);
my %set;
@set{@elems} = (1) x @elems;
```

Следующее приложение хешей также использует тот факт, что хэши и списки часто могут быть взаимозаменяемы для реализации названных функций args:

```
sub hash_args {
  my %args = @_;
  my %defaults = (foo => 1, bar => 0);
  my %overrides = (__unsafe => 0);
  my %settings = (%defaults, %args, %overrides);
}
```

```
# This function can then be called like this:
hash_args(foo => 5, bar => 3); # (foo => 5, bar => 3, __unsafe ==> 0)
hash_args();                   # (foo => 1, bar => 0, __unsafe ==> 0)
hash_args(__unsafe => 1)       # (foo => 1, bar => 0, __unsafe ==> 0)
```

При использовании в качестве булевых хешей истинны, если они не пусты.

Скалярные ссылки

Ссылка - это скалярная переменная (одна префикс `$`), которая «ссылается» на некоторые другие данные.

```
my $value     = "Hello";
my $reference = \$value;
print $value;     # => Hello
print $reference; # => SCALAR(0x2683310)
```

Чтобы получить указанные данные, вы **удалите ссылку на** него.

```
say ${$reference};                  # Explicit prefix syntax
say $$reference;                    # The braces can be left out (confusing)
```

5.24.0

Новый синтаксис разыменования постфикса, доступный по умолчанию из v5.24

```
use v5.24;
say $reference->$*; # New postfix notation
```

Это «де-ссылочное значение» может быть затем изменено, как и исходная переменная.

```
${$reference} =~ s/Hello/World/;
print ${$reference};  # => World
print $value;         # => World
```

Ссылка всегда **правдивая** - даже если значение, на которое оно ссылается, является ложным (например, `0` или `""`).

Возможно, вам понадобится Scalar Reference If:

- Вы хотите передать строку в функцию и изменить ее для этой строки, не возвращая ее.
- Вы хотите явно запретить Perl неявно копировать содержимое большой строки в какой-то момент передачи вашей функции (что особенно важно для старых Perls без строк копирования на запись)
- Вы хотите устранить неоднозначные значения строк с определенным значением из

строк, передающих контент, например:

    - Disambiguate имя файла из содержимого файла
    - Disambiguate возвратил содержимое из строки возвращаемой ошибки

- Вы хотите реализовать облегченную внутреннюю объектную модель, где объекты, переданные вызывающему коду, не имеют видимых пользователем метаданных:

```
our %objects;
my $next_id = 0;
sub new {
   my $object_id = $next_id++;
   $objects{ $object_id } = { ... }; # Assign data for object
   my $ref = \$object_id;
   return bless( $ref, "MyClass" );
}
```

Ссылки на массивы

Массивные ссылки - это скаляры (`$`), которые относятся к массивам.

```
my @array = ("Hello"); # Creating array, assigning value from a list
my $array_reference = \@array;
```

Их можно создать более коротко:

```
my $other_array_reference = ["Hello"];
```

Изменение / использование ссылок на массивы требует разглашения их в первую очередь.

```
my @contents = @{ $array_reference };               # Prefix notation
my @contents = @$array_reference;                   # Braces can be left out
```

5.24.0

Новый синтаксис разыменования постфикса, доступный по умолчанию из v5.24

```
use v5.24;
my @contents = $array_reference->@*; # New postfix notation
```

При доступе к содержимому arrayref по индексу вы можете использовать синтаксический сахар `->` .

```
my @array = qw(one two three);      my $arrayref = [ qw(one two three) ]
my $one = $array[0];                my $one = $arrayref->[0];
```

В отличие от массивов, arrayrefs может быть вложенным:

```
my @array = ( (1, 0), (0, 1) )  # ONE array of FOUR elements: (1, 0, 0, 1)
```

```
my @matrix = ( [1, 0], [0, 1] ) # an array of two arrayrefs
my $matrix = [ [0, 1], [1, 0] ] # an arrayref of arrayrefs
# There is no namespace conflict between scalars, arrays and hashes
# so @matrix and $matrix _both_ exist at this point and hold different values.

my @diagonal_1 = ($matrix[0]->[1], $matrix[1]->[0])     # uses @matrix
my @diagonal_2 = ($matrix->[0]->[1], $matrix->[1]->[0]) # uses $matrix
# Since chained []- and {}-access can only happen on references, you can
# omit some of those arrows.
my $corner_1 = $matrix[0][1];   # uses @matrix;
my $corner_2 = $matrix->[0][1]; # uses $matrix;
```

При использовании в качестве булева ссылки всегда верны.

Хэш-ссылки

Хеш-ссылки - это скаляры, содержащие указатель на ячейку памяти, содержащую данные хэша. Поскольку скаляр указывает непосредственно на сам хэш, когда он передается подпрограмме, изменения, сделанные в хэше, не являются локальными для подпрограммы, как при регулярном хэше, но вместо этого являются глобальными.

Сначала давайте рассмотрим, что происходит, когда вы передаете обычный хеш подпрограмме и изменяете ее внутри:

```
use strict;
use warnings;
use Data::Dumper;

sub modify
{
    my %hash = @_;

    $hash{new_value} = 2;

    print Dumper("Within the subroutine");
    print Dumper(\%hash);

    return;
}

my %example_hash = (
    old_value   => 1,
);

modify(%example_hash);

print Dumper("After exiting the subroutine");
print Dumper(\%example_hash);
```

Результат:

```
$VAR1 = 'Within the subroutine';
$VAR1 = {
          'new_value' => 2,
          'old_value' => 1
```

```
        };
$VAR1 = 'After exiting the subroutine';
$VAR1 = {
          'old_value' => 1
        };
```

Обратите внимание, что после выхода из подпрограммы хэш остается неизменным; все изменения в нем были локальными для подпрограммы изменения, потому что мы передали копию хэша, а не самого хэша.

Для сравнения, когда вы передаете hashref, вы передаете адрес исходному хешу, поэтому любые изменения, сделанные в подпрограмме, будут сделаны в исходном хэше:

```
use strict;
use warnings;
use Data::Dumper;

sub modify
{
    my $hashref = shift;

    # De-reference the hash to add a new value
    $hashref->{new_value} = 2;

    print Dumper("Within the subroutine");
    print Dumper($hashref);

    return;
}

# Create a hashref
my $example_ref = {
    old_value   => 1,
};

# Pass a hashref to a subroutine
modify($example_ref);

print Dumper("After exiting the subroutine");
print Dumper($example_ref);
```

Это приведет к:

```
$VAR1 = 'Within the subroutine';
$VAR1 = {
          'new_value' => 2,
          'old_value' => 1
        };
$VAR1 = 'After exiting the subroutine';
$VAR1 = {
          'new_value' => 2,
          'old_value' => 1
        };
```

Typeglobs, typeglob refs, дескрипторы файлов и константы

Типglob `*foo` содержит ссылки на содержимое *глобальных* переменных с таким именем: `$foo` , `@foo` , `$foo` , `&foo` и т. Д. Вы можете получить к нему доступ как хэш и назначить непосредственно манипулировать таблицами символов (зло!).

```
use v5.10; # necessary for say
our $foo = "foo";
our $bar;
say ref *foo{SCALAR};     # SCALAR
say ${ *foo{SCALAR} };    # bar
*bar = *foo;
say $bar;                 # bar
$bar = 'egg';
say $foo;                 # egg
```

Typeglobs чаще обрабатываются при работе с файлами. `open` , например, создает ссылку на типglob, когда его просят создать неглобальный дескриптор файла:

```
use v5.10; # necessary for say
open(my $log, '> utf-8', '/tmp/log') or die $!; # open for writing with encoding
say $log 'Log opened';

# You can dereference this globref, but it's not very useful.
say ref $log;                   # GLOB
say (*{$log}->{IO} // 'undef'); # undef

close $log or die $!;
```

Typeglobs также могут использоваться для создания глобальных переменных только для чтения, хотя `use constant` используется более широко.

```
# Global constant creation
*TRUE = \('1');
our $TRUE;
say $TRUE;  # 1
$TRUE = ''; # dies, "Modification of a read-only value attempted"

# use constant instead defines a parameterless function, therefore it's not global,
# can be used without sigils, can be imported, but does not interpolate easily.
use constant (FALSE => 0);
say FALSE;        # 0
say &FALSE;       # 0
say "${\FALSE}";  # 0 (ugh)
say *FALSE{CODE}; # CODE(0xMA1DBABE)

# Of course, neither is truly constant when you can manipulate the symbol table...
*TRUE = \('');
use constant (EVIL => 1);
*FALSE = *EVIL;
```

Sigils

Perl имеет ряд сигил:

```
$scalar = 1; # individual value
```

```
@array = ( 1, 2, 3, 4, 5 ); # sequence of values
%hash = ('it', 'ciao', 'en', 'hello', 'fr', 'salut'); # unordered key-value pairs
&function('arguments'); # subroutine
*typeglob; # symbol table entry
```

Они выглядят как сигилы, но не являются:

```
\@array; # \ returns the reference of what's on the right (so, a reference to @array)
$#array; # this is the index of the last element of @array
```

Вы можете использовать фигурные скобки после сигилы, если вы так склонны. Иногда это улучшает читаемость.

```
say ${value} = 5;
```

Хотя вы используете разные сигилы для определения переменных разных типов, к одной и той же переменной можно обращаться по-разному в зависимости от того, какие символы вы используете.

```
%hash;            # we use % because we are looking at an entire hash
$hash{it};        # we want a single value, however, that's singular, so we use $
$array[0];        # likewise for an array. notice the change in brackets.
@array[0,3];      # we want multiple values of an array, so we instead use @
@hash{'it','en'}; # similarly for hashes (this gives the values: 'ciao', 'hello')
%hash{'it','fr'}; # we want an hash with just some of the keys, so we use %
                  # (this gives key-value pairs: 'it', 'ciao', 'fr', 'salut')
```

Это особенно касается ссылок. Чтобы использовать ссылочное значение, вы можете комбинировать сигилы вместе.

```
my @array = 1..5;                     # This is an array
my $reference_to_an_array = \@array; # A reference to an array is a singular value
push @array, 6;                       # push expects an array
push @$reference_to_an_array, 7;      # the @ sigil means what's on the right is an array
                                      # and what's on the right is $reference_to_an_array
                                      # hence: first a @, then a $
```

Вот, возможно, менее запутанный способ подумать об этом. Как мы видели ранее, вы можете использовать фигурные скобки, чтобы обернуть то, что находится справа от сигилы. Поэтому вы можете думать о `@{}` как о чем-то, что берет ссылку на массив и дает вам ссылочный массив.

```
# pop does not like array references
pop $reference_to_an_array; # ERROR in Perl 5.20+
# but if we use @{}, then...
pop @{ $reference_to_an_array }; # this works!
```

Как оказалось, `@{}` фактически принимает выражение:

```
my $values = undef;
```

```
say pop @{ $values };       # ERROR: can't use undef as an array reference
say pop @{ $values // [5] } # undef // [5] gives [5], so this prints 5
```

... и тот же трюк работает и для других сигил.

```
# This is not an example of good Perl. It is merely a demonstration of this language feature
my $hashref = undef;
for my $key ( %{ $hashref // {} } ) {
  "This doesn't crash";
}
```

... но если «аргумент» для сигилы прост, вы можете оставить фигурные скобки.

```
say $$scalar_reference;
say pop @$array_reference;
for keys (%$hash_reference) { ... };
```

Все может стать чрезмерно экстравагантным. Это работает, но, пожалуйста, Perl ответственно.

```
my %hash = (it => 'ciao', en => 'hi', fr => 'salut');
my $reference = \%hash;
my $reference_to_a_reference = \$reference;

my $italian = $hash{it};                              # Direct access
my @greets = @$reference{'it', 'en'};                 # Dereference, then access as array
my %subhash = %$$reference_to_a_reference{'en', 'fr'} # Dereference ×2 then access as hash
```

Для большинства нормальных применений вы можете просто использовать имена подпрограмм без сигилы. (Переменные без сигилы обычно называются «barewords»). `&` Sigil полезен только в ограниченном числе случаев.

- Составление ссылки на подпрограмму:

  ```
  sub many_bars { 'bar' x $_[0] }
  my $reference = \&many_bars;
  say $reference->(3); # barbarbar
  ```

- Вызов функции, игнорирующей ее прототип.
- В сочетании с goto, как немного странный вызов функции, у которого текущий кадр вызова заменен вызывающим. Подумайте о вызове API linux `exec()` , но для функций.

Прочитайте переменные онлайн: https://riptutorial.com/ru/perl/topic/1566/переменные

глава 20: подпрограммы

замечания

Подпрограммы получают свои аргументы магической переменной `@_` . Хотя его не нужно распаковывать, рекомендуется, так как он помогает читать и предотвращает случайные изменения, поскольку аргументы `@_` передаются по ссылке (могут быть изменены).

Examples

Создание подпрограмм

Подпрограммы создаются с использованием ключевого слова `sub` за которым следует идентификатор и блок кода, заключенный в фигурные скобки.

Вы можете получить доступ к аргументам, используя специальную переменную `@_` , которая содержит все аргументы в виде массива.

```
sub function_name {
    my ($arg1, $arg2, @more_args) = @_;
    # ...
}
```

Поскольку по умолчанию `shift` функции сдвигается `@_` при использовании внутри подпрограммы, это обычный шаблон, который последовательно извлекает аргументы в локальные переменные в начале подпрограммы:

```
sub function_name {
    my $arg1 = shift;
    my $arg2 = shift;
    my @more_args = @_;
    # ...
}

# emulate named parameters (instead of positional)
sub function_name {
    my %args = (arg1 => 'default', @_);
    my $arg1 = delete $args{arg1};
    my $arg2 = delete $args{arg2};
    # ...
}

sub {
    my $arg1 = shift;
    # ...
}->($arg);
```

5.20.0

Альтернативно, экспериментальная особенность `"signatures"` может использоваться для распаковки параметров, которые передаются по значению (*не* по ссылке).

```
use feature "signatures";

sub function_name($arg1, $arg2, @more_args) {
    # ...
}
```

Значения по умолчанию могут использоваться для параметров.

```
use feature "signatures";

sub function_name($arg1=1, $arg2=2) {
    # ...
}
```

Вы можете использовать любое выражение, чтобы дать значение по умолчанию для параметра, включая другие параметры.

```
sub function_name($arg1=1, $arg2=$arg1+1) {
    # ...
}
```

Обратите внимание, что вы не можете ссылаться на параметры, которые определены после текущего параметра - следовательно, следующий код работает не так, как ожидалось.

```
sub function_name($arg1=$arg2, $arg2=1) {
    print $arg1;  # => <nothing>
    print $arg2;  # => 1
}
```

Аргументы подпрограммы передаются по ссылке (кроме записей в подписи)

Аргументы подпрограммы в Perl передаются по ссылке, если они не находятся в подписи. Это означает, что члены массива `@_` внутри sub являются просто *псевдонимами* для фактических аргументов. В следующем примере `$text` в основной программе остается модифицированным после вызова подпрограммы, потому что `$_[0]` внутри субфайла фактически является просто другим именем для той же переменной. Второй вызов вызывает ошибку, потому что строковый литерал не является переменной и поэтому не может быть изменен.

```
use feature 'say';

sub edit {
    $_[0] =~ s/world/sub/;
}
```

```
my $text = "Hello, world!";
edit($text);
say $text;      # Hello, sub!

edit("Hello, world!"); # Error: Modification of a read-only value attempted
```

Поэтому, чтобы `@_` переменных вашего вызывающего абонента, важно скопировать `@_` на локально измененные переменные (`my ...`), как описано в разделе «Создание подпрограмм».

подпрограммы

Подпрограммы содержат код. Если не указано иное, они определяются глобально.

```
# Functions do not (have to) specify their argument list
sub returns_one {
  # Functions return the value of the last expression by default
  # The return keyword here is unnecessary, but helps readability.
  return 1;
}

# Its arguments are available in @_, however
sub sum {
  my $ret = 0;
  for my $value (@_) {
    $ret += $value
  }
  return $ret;
}

# Perl makes an effort to make parens around argument list optional
say sum 1..3;     # 6

# If you treat functions as variables, the & sigil is mandatory.
say defined &sum; # 1
```

Некоторые встроенные функции, такие как `print` или `say` являются ключевыми словами, а не функциями, поэтому, например, `&say` undefined. Это также означает, что вы можете определить их, но вам нужно будет указать имя пакета, чтобы на самом деле вызвать их

```
# This defines the function under the default package, 'main'
sub say {
  # This is instead the say keyword
  say "I say, @_";
}

# ...so you can call it like this:
main::say('wow'); # I say, wow.
```

5.18.0

Начиная с Perl 5.18, вы также можете иметь неглобальные функции:

```
use feature 'lexical_subs';
my $value;
{
  # Nasty code ahead
  my sub prod {
    my $ret = 1;
    $ret *= $_ for @_;
    $ret;
  }
  $value = prod 1..6; # 720
  say defined &prod; # 1
}
say defined &prod; # 0
```

5.20.0

Начиная с 5.20, вы также можете иметь именованные параметры.

```
use feature 'signatures';
sub greet($name) {
  say "Hello, $name";
}
```

Это *не* следует путать с прототипами, средство Perl должно позволить вам определять функции, которые ведут себя как встроенные. Прототипы функций должны быть видны во время компиляции, и его эффекты можно игнорировать, указав `&` sigil. Прототипы, как правило, считаются расширенной функцией, которую лучше всего использовать с большой осторожностью.

```
# This prototype makes it a compilation error to call this function with anything
# that isn't an array. Additionally, arrays are automatically turned into arrayrefs
sub receives_arrayrefs(\@\@) {
   my $x = shift;
   my $y = shift;
}

my @a = (1..3);
my @b = (1..4);
receives_arrayrefs(@a, @b);    # okay,   $x = \@a, $y = \@b, @_ = ();
receives_arrayrefs(\@a, \@b);  # compilation error, "Type … must be array …"
BEGIN { receives_arrayrefs(\@a, \@b); }

# Specify the sigil to ignore the prototypes.
&receives_arrayrefs(\@a, \@b); # okay,   $x = \@a, $y = \@b, @_ = ();
&receives_arrayrefs(@a, @b);   # ok, but $x = 1,   $y = 2,   @_ = (3,1,2,3,4);
```

Прочитайте подпрограммы онлайн: https://riptutorial.com/ru/perl/topic/711/подпрограммы

глава 21: Правда и ложь

Синтаксис

- undef # False
- '# Определено, False
- 0 # Определено, имеет длину, ложь
- '0' # Определено, имеет длину, ложь

замечания

Perl не имеет логического типа данных и не имеет никаких `true` и `false` ключевых слов, как многие другие языки. Однако каждое скалярное значение будет оцениваться как истинное или ложное при оценке в булевом контексте (например, условие в случае if или цикл while).

Следующие значения считаются ложными:

- `''` , пустая строка. Это то, что возвращают встроенные операторы сравнения (например, `0 == 1`)
- `0` , число 0, даже если вы пишете его как 000 или 0.0
- `'0'` , строка, содержащая одну цифру 0
- `undef` , неопределенное значение
- Объекты, которые используют перегрузку для нумерации / стягивания в ложные значения, такие как `JSON::false`

Все остальные значения верны:

- любое ненулевое число, такое как `1` , `3.14` , `'NaN'` ИЛИ `'Inf'`
- любая строка, численная 0, но не буквально строка `'0'` , такая как `'00'` , `'0e0'` , `"0\n"` и `"abc"` .
  Если вы *намеренно* возвращаете истинное числовое значение 0, предпочитаете `'0E0'` (используется известными модулями) или `'0 but true'` (используется функциями Perl)
- любая другая строка, которая не является пустой, например, `' '` , `'false'`
- все ссылки, даже если они ссылаются на ложные значения, такие как `\''` , `[]` или `{}`
- массив или хэш ложных значений

Следующие операторы обычно обрабатываются для возврата булевского в скалярном контексте:

- `@a` возвращает ли массив пустым или нет

- `%h` возвращает ли хэш пустой или нет
- `grep` возвращает ли найденные совпадающие элементы или нет
- `@a = LIST` и `(LIST) = LIST` возвращать, будет ли в правой стороне СПИСИЛИТЬ любые скаляры или нет

Examples

Список истинных и ложных значений

```
use feature qw( say );

# Numbers are true if they're not equal to 0.
say 0             ? 'true' : 'false'; # false
say 1             ? 'true' : 'false'; # true
say 2             ? 'true' : 'false'; # true
say -1            ? 'true' : 'false'; # true
say 1-1           ? 'true' : 'false'; # false
say 0e7           ? 'true' : 'false'; # false
say -0.00         ? 'true' : 'false'; # false

# Strings are true if they're not empty.
say 'a'           ? 'true' : 'false'; # true
say 'false'       ? 'true' : 'false'; # true
say ''            ? 'true' : 'false'; # false

# Even if a string would be treated as 0 in numeric context, it's true if nonempty.
# The only exception is the string "0", which is false.
# To force numeric context add 0 to the string
say '0'           ? 'true' : 'false'; # false
say '0.0'         ? 'true' : 'false'; # true
say '0e0'         ? 'true' : 'false'; # true
say '0 but true'  ? 'true' : 'false'; # true
say '0 whargarbl' ? 'true' : 'false'; # true
say 0+'0 argarbl' ? 'true' : 'false'; # false

# Things that become numbers in scalar context are treated as numbers.
my @c = ();
my @d = (0);
say @c            ? 'true' : 'false'; # false
say @d            ? 'true' : 'false'; # true

# Anything undefined is false.
say undef         ? 'true' : 'false'; # false

# References are always true, even if they point at something false
my @c = ();
my $d = 0;
say \@c            ? 'true' : 'false'; # true
say \$d            ? 'true' : 'false'; # true
say \0             ? 'true' : 'false'; # true
say \''            ? 'true' : 'false'; # true
```

Прочитайте Правда и ложь онлайн: https://riptutorial.com/ru/perl/topic/649/правда-и-ложь

глава 22: Приложения GUI в Perl

замечания

Tk является одним из наиболее часто используемых инструментов GUI для Perl. Другими распространенными инструментами являются GTK + 2 и 3, WxWidgets и виджеты Win32. Менее часто используются Qt4, XUL, Prima и FLTK.

Tk, GTK + 3, Wx, Win32, Prima, FLTK и XUL активно обновляются. Qt4 и GTK + 2 больше не разрабатываются активно, но могут иметь версии технического обслуживания.

Examples

Приложение GTK

```
use strict;
use warnings;

use Gtk2 -init;

my $window = Gtk2::Window->new();
$window->show();

Gtk2->main();

0;
```

Прочитайте Приложения GUI в Perl онлайн: https://riptutorial.com/ru/perl/topic/5924/приложения-gui-в-perl

глава 23: Простое взаимодействие с базой данных через модуль DBI

параметры

колонка	колонка
`$driver`	Драйвер для DB, «Pg» для Postgresql и «mysql» для MySQL
`$database`	имя вашей базы данных
`$userid`	идентификатор вашей базы данных
`$password`	пароль вашей базы данных
`$query`	поставьте свой запрос здесь, например: «выберите * из $ your_table»

Examples

Модуль DBI

Вы должны убедиться, что модуль DBI установлен на вашем компьютере, а затем выполните следующие шаги:

1. использовать DBI-модуль в вашем скрипте perl

```
use DBI;
```

2. Объявить некоторые первичные параметры

```
my $driver = "MyDriver";

my $database = "DB_name";

my $dsn = "DBI:$driver:dbname=$database";

my $userid = "your_user_ID";

my $password = "your_password";

my $tablename = "your_table";
```

3. Подключение к базе данных

```
my $dbh = DBI->connect($dsn, $userid, $password);
```

4. Подготовьте свой запрос

```
my $query = $dbh->prepare("Your DB query");
```

Пример:

```
$my_query = qq/SELECT * FROM table WHERE column1 = 2/;

my $query = $dbh->prepare($my_query);
```

Мы также можем использовать переменную в запросе, как показано ниже:

```
my $table_name = "table";

my $filter_value = 2;

$my_query = qq/SELECT * FROM $table_name WHERE column1 = $filter_value/;
```

5. Выполнение запроса

```
$query->execute();
```

* Примечание. Чтобы избежать инъекции, вы должны использовать заполнители `?` вместо того, чтобы поместить вашу переменную в запрос.

Пример: вы хотите показать все данные из «таблицы», где column1 = $ value1 и column2 = $ value2:

```
my $query = $dbh->prepare("SELECT * FROM table WHERE column1 = ? AND column2 = ?;");

$query->execute($value1, $value2);
```

6. Fletch ваши данные

`my @row = $query->fetchrow_array();` хранить данные как массив

или же

`my $ref = $sth->fetchrow_hashref();` хранить данные как хеш-ссылку

7. Завершить и отключить БД

```
$sth->finish;

$dbh->disconnect();
```

Прочитайте Простое взаимодействие с базой данных через модуль DBI онлайн: https://riptutorial.com/ru/perl/topic/5917/простое-взаимодействие-с-базой-данных-через-модуль-dbi

глава 24: Простой способ проверить установленные модули на Mac и Ubuntu

Examples

Проверить установленные модули perl через терминал

Введите команду ниже:

```
instmodsh
```

Он покажет вам гильдию, как показано ниже:

```
Available commands are:
   l            - List all installed modules
   m <module>   - Select a module
   q            - Quit the program
cmd?
```

Затем введите `l` чтобы перечислить все установленные модули, вы также можете использовать команду `m <module>` чтобы выбрать модуль и получить его информацию.

После окончания просто введите `q` чтобы выйти.

Использовать perldoc для проверки пути установки пакета Perl

```
$ perldoc -l Time::Local
```

Как проверить Perel-совместимые модули.

```
$ corelist -v v5.23.1
```

Как проверить версию установленного модуля?

```
$> perl -MFoo::Bar\ 9999
$> Foo::Bar version 9999 required--this is only version 1.1.
```

Прочитайте Простой способ проверить установленные модули на Mac и Ubuntu онлайн: https://riptutorial.com/ru/perl/topic/5925/простой-способ-проверить-установленные-модули-на-mac-и-ubuntu

глава 25: Разбор XML

Examples

Анализ с помощью XML :: Twig

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

use strict;
use warnings 'all';

use XML::Twig;

my $twig = XML::Twig->parse( \*DATA );

#we can use the 'root' method to find the root of the XML.
my $root = $twig->root;

#first_child finds the first child element matching a value.
my $title = $root->first_child('title');

#text reads the text of the element.
my $title_text = $title->text;

print "Title is: ", $title_text, "\n";

#The above could be combined:
print $twig ->root->first_child_text('title'), "\n";

## You can use the 'children' method to iterate multiple items:
my $list = $twig->root->first_child('list');

#children can optionally take an element 'tag' - otherwise it just returns all of them.
foreach my $element ( $list->children ) {

   #the 'att' method reads an attribute
   print "Element with ID: ", $element->att('id') // 'none here', " is ", $element->text,
     "\n";
}

#And if we need to do something more complicated, we an use 'xpath'.
#get_xpath or findnodes do the same thing:
#return a list of matches, or if you specify a second numeric argument, just that numbered
match.

#xpath syntax is fairly extensive, but in this one - we search:
# anywhere in the tree: //
#nodes called 'item'
#with an id attribute [@id]
#and with that id attribute equal to "1000".
#by specifying '0' we say 'return just the first match'.

print "Item 1000 is: ", $twig->get_xpath( '//item[@id="1000"]', 0 )->text, "\n";

#this combines quite well with `map` to e.g. do the same thing on multiple items
print "All IDs:\n", join ( "\n", map { $_ -> att('id') } $twig -> get_xpath('//item'));
```

```
#note how this also finds the item under 'summary', because of //

__DATA__
<?xml version="1.0" encoding="utf-8"?>
<root>
  <title>some sample xml</title>
  <first key="value" key2="value2">
    <second>Some text</second>
  </first>
  <third>
    <fourth key3="value">Text here too</fourth>
  </third>
  <list>
     <item id="1">Item1</item>
     <item id="2">Item2</item>
     <item id="3">Item3</item>
     <item id="66">Item66</item>
     <item id="88">Item88</item>
     <item id="100">Item100</item>
     <item id="1000">Item1000</item>
     <notanitem>Not an item at all really.</notanitem>
  </list>
  <summary>
     <item id="no_id">Test</item>
  </summary>
</root>
```

Потребление XML с помощью XML :: Rabbit

С `XML::Rabbit` можно легко использовать XML-файлы. Вы *определяете* декларативным образом и с синтаксисом XPath, который вы ищете в XML и `XML::Rabbit` , возвращают объекты в соответствии с данным определением.

Определение:

```
package Bookstore;
use XML::Rabbit::Root;
has_xpath_object_list books => './book' => 'Bookstore::Book';
finalize_class();

package Bookstore::Book;
use XML::Rabbit;
has_xpath_value bookid => './@id';
has_xpath_value author => './author';
has_xpath_value title => './title';
has_xpath_value genre => './genre';
has_xpath_value price => './price';
has_xpath_value publish_date => './publish_date';
has_xpath_value description => './description';
has_xpath_object purchase_data => './purchase_data' => 'Bookstore::Purchase';
finalize_class();

package Bookstore::Purchase;
use XML::Rabbit;
has_xpath_value price => './price';
has_xpath_value date => './date';
finalize_class();
```

Потребление XML:

```
use strict;
use warnings;
use utf8;

package Library;
use feature qw(say);
use Carp;
use autodie;

say "Showing data information";
my $bookstore = Bookstore->new( file => './sample.xml' );

foreach my $book( @{$bookstore->books} ) {
    say "ID: " . $book->bookid;
    say "Title: " . $book->title;
    say "Author: " . $book->author, "\n";
}
```

Заметки:

Будьте осторожны со следующим:

1. Первый класс должен быть `XML::Rabbit::Root` . Он поместит вас в основной тег XML-документа. В нашем случае он разместит нас внутри `<catalog>`

2. Вложенные классы, которые являются необязательными. Эти классы должны быть доступны через блок try / catch (или `eval / $@` check). Необязательные поля просто возвращают значение `null` . Например, для `purchase_data` цикл будет:

```
foreach my $book( @{$bookstore->books} ) {
    say "ID: " . $book->bookid;
    say "Title: " . $book->title;
    say "Author: " . $book->author;
    try {
        say "Purchase price: ". $book->purchase_data->price, "\n";
    } catch {
        say "No purchase price available\n";
    }
}
```

sample.xml

```
<?xml version="1.0"?>
<catalog>
   <book id="bk101">
      <author>Gambardella, Matthew</author>
      <title>XML Developer's Guide</title>
      <genre>Computer</genre>
      <price>44.95</price>
      <publish_date>2000-10-01</publish_date>
      <description>An in-depth look at creating applications
      with XML.</description>
   </book>
```

```
   <book id="bk102">
      <author>Ralls, Kim</author>
      <title>Midnight Rain</title>
      <genre>Fantasy</genre>
      <price>5.95</price>
      <publish_date>2000-12-16</publish_date>
      <description>A former architect battles corporate zombies,
      an evil sorceress, and her own childhood to become queen
      of the world.</description>
   </book>
   <book id="bk103">
      <author>Corets, Eva</author>
      <title>Maeve Ascendant</title>
      <genre>Fantasy</genre>
      <price>5.95</price>
      <publish_date>2000-11-17</publish_date>
      <description>After the collapse of a nanotechnology
      society in England, the young survivors lay the
      foundation for a new society.</description>
   </book>
   <book id="bk104">
      <author>Corets, Eva</author>
      <title>Oberon's Legacy</title>
      <genre>Fantasy</genre>
      <price>5.95</price>
      <publish_date>2001-03-10</publish_date>
      <description>In post-apocalypse England, the mysterious
      agent known only as Oberon helps to create a new life
      for the inhabitants of London. Sequel to Maeve
      Ascendant.</description>
      <purchase_data>
        <date>2001-12-21</date>
        <price>20</price>
      </purchase_data>
   </book>
</catalog>
```

Анализ с помощью XML :: LibXML

```
# This uses the 'sample.xml' given in the XML::Twig example.

# Module requirements (1.70 and above for use of load_xml)
use XML::LibXML '1.70';

# let's be a good perl dev
use strict;
use warnings 'all';

# Create the LibXML Document Object
my $xml = XML::LibXML->new();

# Where we are retrieving the XML from
my $file = 'sample.xml';

# Load the XML from the file
my $dom = XML::LibXML->load_xml(
    location => $file
);
```

```
# get the docroot
my $root = $dom->getDocumentElement;

# if the document has children
if($root->hasChildNodes) {

    # getElementsByLocalName returns a node list of all elements who's
    # localname matches 'title', and we want the first occurrence
    # (via get_node(1))
    my $title = $root->getElementsByLocalName('title');

    if(defined $title) {
        # Get the first matched node out of the nodeList
        my $node = $title->get_node(1);

        # Get the text of the target node
        my $title_text = $node->textContent;

        print "The first node with name 'title' contains: $title_text\n";
    }

    # The above calls can be combined, but is possibly prone to errors
    # (if the getElementsByLocalName() failed to match a node).
    #
    # my $title_text = $root->getElementsByLocalName('title')->get_node(1)->textContent;
}

# Using Xpath, get the price of the book with id 'bk104'
#

# Set our xpath
my $xpath = q!/catalog/book[@id='bk104']/price!;

# Does that xpath exist?
if($root->exists($xpath)) {

    # Pull in the twig
    my $match = $root->find($xpath);

    if(defined $match) {
        # Get the first matched node out of the nodeList
        my $node = $match->get_node(1);

        # pull in the text of that node
        my $match_text = $node->textContent;

        print "The price of the book with id bk104 is: $match_text\n";
    }
}
```

Прочитайте Разбор XML онлайн: https://riptutorial.com/ru/perl/topic/3590/разбор-xml

глава 26: Разделить строку на неупорядоченные разделители

Examples

parse_line ()

Использование `parse_line()` Text :: ParseWords :

```
use 5.010;
use Text::ParseWords;

my $line = q{"a quoted, comma", word1, word2};
my @parsed = parse_line(',', 1, $line);
say for @parsed;
```

Выход:

```
"a quoted, comma"
 word1
 word2
```

Текст :: CSV или текст :: CSV_XS

```
use Text::CSV; # Can use Text::CSV which will switch to _XS if installed
$sep_char = ",";
my $csv = Text::CSV->new({sep_char => $sep_char});
my $line = q{"a quoted, comma", word1, word2};
$csv->parse($line);
my @fields = $csv->fields();
print join("\n", @fields)."\n";
```

Выход:

```
a quoted, comma
 word1
 word2
```

ЗАМЕТКИ

- По умолчанию Text :: CSV не `Text::ParseWords` пробелы вокруг символа разделителя, как `Text::ParseWords` делает `Text::ParseWords` . Однако добавление `allow_whitespace=>1` в атрибуты конструктора достигает такого эффекта.

```
my $csv = Text::CSV_XS->new({sep_char => $sep_char, allow_whitespace=>1});
```

Выход:

```
a quoted, comma
word1
word2
```

- Библиотека поддерживает экранирование специальных символов (кавычек, разделителей)
- Библиотека поддерживает настраиваемый символ разделителя, символ кавычки и escape-символ

Документация: http://search.cpan.org/perldoc/Text::CSV

Прочитайте Разделить строку на неупорядоченные разделители онлайн: https://riptutorial.com/ru/perl/topic/2115/разделить-строку-на-неупорядоченные-разделители

глава 27: Регулярные выражения

Examples

Соответствующие строки

Оператор `=~` пытается сопоставить регулярное выражение (выделенное `/`) в строку:

```
my $str = "hello world";
print "Hi, yourself!\n" if $str =~ /^hello/;
```

`/^hello/` - действительное регулярное выражение. `^` - специальный символ, который сообщает регулярному выражению начинать с начала строки и не встречаться где-то посередине. Затем регулярное выражение пытается найти следующие буквы в порядке `h` , `e` , `l` , `l` и `o` .

Регулярные выражения пытаются сопоставить переменную по умолчанию (`$_`), если она голая:

```
$_ = "hello world";

print "Ahoy!\n" if /^hello/;
```

Вы также можете использовать разные разделители, если вы предшествуете регулярному выражению с помощью оператора `m` :

```
m~^hello~;
m{^hello};
m|^hello|;
```

Это полезно при сопоставлении строк, которые включают символ `/` :

```
print "user directory" if m|^/usr|;
```

Использование \ Q и \ E в сопоставлении с образцом

Что между \ Q и \ E рассматривается как обычные символы

```
#!/usr/bin/perl

my $str = "hello.it's.me";

my @test = (
```

```
  "hello.it's.me",
    "hello/it's!me",
    );

sub ismatched($) { $_[0] ? "MATCHED!" : "DID NOT MATCH!" }

my @match = (
      [ general_match=> sub { ismatched /$str/ } ],
      [ qe_match    => sub { ismatched /\Q$str\E/ } ],
      );

for (@test) {
    print "\String = '$_':\n";

foreach my $method (@match) {
    my($name,$match) = @$method;
    print "  - $name: ", $match->(), "\n";
}
```

}

Выход

```
String = 'hello.it's.me':
  - general_match: MATCHED!
  - qe_match: MATCHED!
String = 'hello/it's!me':
  - general_match: MATCHED!
  - qe_match: DID NOT MATCH!
```

Разбор строки с регулярным выражением

Как правило, не рекомендуется использовать регулярное выражение для синтаксического анализа сложной структуры . Но это может быть сделано. Например, вы можете загрузить данные в таблицу hive, а поля разделяются запятой, но сложные типы, такие как массив, разделяются символом «|». Файлы содержат записи со всеми полями, разделенными запятой и сложным типом, находятся внутри квадратной скобки. В этом случае этот бит одноразового Perl может быть достаточным:

```
echo "1,2,[3,4,5],5,6,[7,8],[1,2,34],5" | \
    perl -ne \
        'while( /\[[^,\]]+\,.*\]/ ){
            if( /\[([^\]\|]+)\]/){
                $text = $1;
                $text_to_replace = $text;
                $text =~ s/\,/\|/g;
                s/$text_to_replace/$text/;
            }
        } print'
```

Вы хотите определить результат:

1,2, [3 | 4 | 5], 5,6, [7 | 8], [1 | 2 | 34], 5

Заменить строку, используя регулярные выражения

```
s/foo/bar/;         # replace "foo" with "bar" in $_
my $foo = "foo";
$foo =~ s/foo/bar/; # do the above on a different variable using the binding operator =~
s~ foo ~ bar ~;     # using ~ as a delimiter
$foo = s/foo/bar/r; # non-destructive r flag: returns the replacement string without modifying
the variable it's bound to
s/foo/bar/g;        # replace all instances
```

Прочитайте Регулярные выражения онлайн: https://riptutorial.com/ru/perl/topic/3108/регулярные-выражения

глава 28: Сортировка

Вступление

Для сортировки списков вещей Perl имеет только одну функцию, неудивительно называемую `sort` . Он достаточно гибкий, чтобы сортировать все виды элементов: числа, строки в любом количестве кодировок, вложенных структур данных или объектов. Однако из-за его гибкости существует множество трюков и идиом, которые нужно изучить для его использования.

Синтаксис

- сортировать SUBNAME LIST
- сортировать БЛОК-СПИСОК
- сортировать СПИСОК

Examples

Базовая лексическая сортировка

```
@sorted = sort @list;

@sorted = sort { $a cmp $b } @list;

sub compare { $a cmp $b }
@sorted = sort compare @list;
```

Три приведенных выше примера делают то же самое. Если вы не предоставляете какую-либо функцию или блок-компаратор, `sort` предполагает, что вы хотите, чтобы список был правильно отсортирован лексически. Обычно это форма, которую вы хотите, если вам просто нужны ваши данные в определенном предсказуемом порядке и не заботятся о лингвистической правильности.

`sort` проходит пар элементов в `@list` к функции компаратора, который говорит `sort` , какой элемент больше. Оператор `cmp` делает это для строк, а `<=>` делает то же самое для чисел. Сравнитель вызывается довольно часто, в среднем *n* * log (*n*) раз, когда *n* - это количество элементов для сортировки, поэтому важно быстро. В этом случае `sort` использует предопределенные глобальные переменные пакета (`$a` и `$b`) для передачи элементов, которые будут сравниваться с блоком или функцией, вместо правильных параметров функции.

Если вы `use locale` , `cmp` учитывает специфический порядок сортировки по языку, например, он будет сортировать `Å` подобно `A` в датском языке, но после `Z` под английским или

немецким. Однако он не учитывает более сложные правила сортировки Юникода и не обеспечивает какого-либо контроля над заказом - например, телефонные книги часто сортируются иначе, чем словари. В этих случаях рекомендуется использовать `Unicode::Collate` и особенно `Unicode::Collate::Locale` .

Числовая сортировка

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

Сравнение `$a` и `$b` с оператором `<=>` гарантирует, что они сравниваются численно, а не текстовыми по умолчанию.

Обратная сортировка

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

Сортировка элементов в порядке убывания может быть просто достигнута путем замены `$a` и `$b` в блоке компаратора. Тем не менее, некоторые люди предпочитают четкость отдельного `reverse` даже если он немного медленнее.

Трансформация Шварца

Это, пожалуй, самый известный пример оптимизации сортировки с использованием возможностей программирования Perl, которые будут использоваться там, где порядок сортировки элементов зависит от дорогостоящей функции.

```
# What you would usually do
@sorted = sort { slow($a) <=> slow($b) } @list;

# What you do to make it faster
@sorted =
map { $_->[0] }
sort { $a->[1] <=> $b->[1] }
map { [ $_, slow($_) ] }
@list;
```

Проблема с первым примером заключается в том, что компаратор вызывается очень часто и продолжает пересчитывать значения, используя медленную функцию снова и снова. Типичным примером будет сортировка имен файлов по их размеру файла:

```
use File::stat;
@sorted = sort { stat($a)->size <=> stat($b)->size } glob "*";
```

Это работает, но в лучшем случае это наносит накладные расходы на два системных вызова за сравнение, а в худшем случае приходится дважды переходить на диск для каждого отдельного сравнения, и этот диск может находиться в перегруженном файловом

сервере с другой стороны планета.

Введите трюк Рэндалла Шварца.

`@list` преобразование в основном `@list` через три функции, снизу вверх. Первая `map` превращает каждую запись в двухэлементный список исходного элемента, а результат медленной функции - в качестве ключа сортировки, поэтому в конце этого мы называем `slow()` ровно один раз для каждого элемента. Следующий `sort` может затем просто открыть ключ сортировки, просмотрев список. Поскольку нам не нужны ключи сортировки, но нужны только исходные элементы в отсортированном порядке, окончательная `map` выбрасывает списки из двух элементов из уже отсортированного списка, который он получает от `@sort` и возвращает список только своих первых членов ,

Нечувствительный к регистру сортировка

Традиционная техника для выполнения `sort` игнорирует регистр строк `lc` или `uc` для сравнения:

```
@sorted = sort { lc($a) cmp lc($b) } @list;
```

Это работает во всех версиях Perl 5 и вполне достаточно для английского языка; неважно, используете ли вы `uc` или `lc` . Тем не менее, это представляет проблему для таких языков, как греческий или турецкий, где нет соответствия 1: 1 между буквами верхнего и нижнего регистра, чтобы вы получали разные результаты в зависимости от того, используете ли вы `uc` или `lc` . Поэтому Perl 5.16 и выше имеют функцию *фальцовки* `fc` , называемую `fc` которая позволяет избежать этой проблемы, поэтому современная многоязычная сортировка должна использовать это:

```
@sorted = sort { fc($a) cmp fc($b) } @list;
```

Прочитайте Сортировка онлайн: https://riptutorial.com/ru/perl/topic/8958/сортировка

глава 29: Специальные переменные

замечания

ДЕЛАТЬ: Добавить больше содержимого.

Examples

Специальные переменные в perl:

1. `$_` : пространство ввода по умолчанию и шаблонное пространство.

Пример 1:

```
my @array_variable = (1 2 3 4);
foreach (@array_variable){
    print $_."\n";    # $_ will get the value 1,2,3,4 in loop, if no other variable is
supplied.
}
```

Пример 2:

```
while (<FH>){
    chomp($_);    # $_ refers to the iterating lines in the loop.
}
```

Следующие функции используют `$_` в качестве аргумента по умолчанию:

```
abs, alarm, chomp, chop, chr, chroot, cos, defined, eval,
evalbytes, exp, fc, glob, hex, int, lc, lcfirst, length, log,
lstat, mkdir, oct, ord, pos, print, printf, quotemeta, readlink,
readpipe, ref, require, reverse (in scalar context only), rmdir,
say, sin, split (for its second argument), sqrt, stat, study,
uc, ucfirst, unlink, unpack.
```

2. `@_` : Этот массив содержит аргументы, переданные подпрограмме.

Пример 1:

```
example_sub( $test1, $test2, $test3 );

sub example_sub {
    my ( $test1, $test2, $test3 ) = @_;
}
```

Внутри подпрограммы массив `@_` содержит **аргументы,** переданные этой подпрограмме. Внутри подпрограммы `@_` является массивом по `default` для операторов массива `pop` и `shift` .

Прочитайте Специальные переменные онлайн: https://riptutorial.com/ru/perl/topic/7962/специальные-переменные

глава 30: Списки

Examples

Массив в виде списка

Массив является одним из основных типов переменных Perl. Он содержит список, который представляет собой упорядоченную последовательность из нуля или более скаляров. Массив - это перенос переменной (и предоставление доступа) к данным списка, как это описано в perldata .

Вы можете назначить список массиву:

```
my @foo = ( 4, 5, 6 );
```

Вы можете использовать массив везде, где ожидается список:

```
join '-', ( 4, 5, 6 );
join '-', @foo;
```

Некоторые операторы работают только с массивами, поскольку они мутируют список, содержащий массив:

```
shift @array;
unshift @array, ( 1, 2, 3 );
pop @array;
push @array, ( 7, 8, 9 );
```

Присвоение списка хешу

Списки также могут быть присвоены хэш-переменным. При создании списка, который будет назначен хеш-переменной, рекомендуется использовать ***жирную запятую*** `=>` между клавишами и значениями, чтобы показать их взаимосвязь:

```
my %hash = ( foo => 42, bar => 43, baz => 44 );
```

The `=>` - действительно только специальная запятая, которая автоматически цитирует операнд слева. Итак, вы *можете* использовать обычные запятые, но отношения не так ясны:

```
my %hash = ( 'foo', 42, 'bar', 43, 'baz', 44 );
```

Вы также можете использовать цитированные строки для левого операнда жирной запятой `=>` , что особенно полезно для ключей, содержащих пробелы.

```
my %hash = ( 'foo bar' => 42, 'baz qux' => 43 );
```

Подробнее см. Comma-оператор `perldoc perlop` .

Списки могут быть переданы в подпрограммы

Чтобы передать список в подпрограмму, вы указываете имя подпрограммы и затем указываете ей список:

```
test_subroutine( 'item1', 'item2' );
test_subroutine  'item1', 'item2';     # same
```

Внутренне Perl делает *псевдонимы* для этих аргументов и помещает их в массив `@_` который доступен в подпрограмме:

```
@_ =  ( 'item1', 'item2' ); # Done internally by perl
```

Вы получаете такие аргументы подпрограммы:

```
sub test_subroutine {
    print $_[0]; # item1
    print $_[1]; # item2
}
```

Алиасирование дает вам возможность изменить исходное значение аргумента, переданного подпрограмме:

```
sub test_subroutine {
    $_[0] +=  2;
}

my $x =  7;
test_subroutine( $x );
print $x; # 9
```

Чтобы предотвратить непреднамеренные изменения исходных значений, переданных в вашу подпрограмму, вы должны скопировать их:

```
sub test_subroutine {
    my( $copy_arg1, $copy_arg2 ) =  @_;
    $copy_arg1 += 2;
}

my $x =  7;
test_subroutine $x; # in this case $copy_arg2 will have `undef` value
print $x; # 7
```

Чтобы проверить, сколько аргументов было передано в подпрограмму, проверьте размер `@_`

```
sub test_subroutine {
    print scalar @_, ' argument(s) passed into subroutine';
}
```

Если вы передадите аргументы массива в подпрограмму, все они будут *сплющены* :

```
my @x =  ( 1, 2, 3 );
my @y =  qw/ a b c /; # ( 'a', 'b', 'c' )
test_some_subroutine @x, 'hi', @y; # 7 argument(s) passed into subroutine
# @_ =  ( 1, 2, 3, 'hi', 'a', 'b', 'c' ) # Done internally for this call
```

Если ваш `test_some_subroutine` содержит оператор `$_[4] = 'd'` , для вышеупомянутого вызова это приведет к тому, что `$y[0]` получит значение `d` :

```
print "@y"; # d b c
```

Возвратный список из подпрограммы

Разумеется, вы можете возвращать списки из субсайтов:

```
sub foo {
    my @list1 =  ( 1, 2, 3 );
    my @list2 =  ( 4, 5 );

    return    ( @list1, @list2 );
}

my @list =  foo();
print @list;          # 12345
```

Но это не рекомендуется, если вы не знаете, что делаете.

Хотя это нормально, когда результат находится в контексте **LIST** , в контексте **SCALAR** все неясно. Давайте посмотрим на следующую строку:

```
print scalar foo();  # 2
```

Почему `2` ? Что здесь происходит?

1. Поскольку `foo()` оцененный в контексте *SCALAR* , этот список `( @list1, @list2 )` *list1* `( @list1, @list2 )` также оценивается в контексте *SCALAR*
2. В контексте *SCALAR* LIST возвращает свой последний элемент. Здесь `@list2`
3. Снова в контексте *SCALAR* **массив** `@list2` возвращает количество его элементов. Здесь `2` .

В большинстве случаев **правильная стратегия будет возвращать ссылки на структуры данных** .
Поэтому в нашем случае мы должны сделать следующее:

```
return    ( \@list1, \@list2 );
```

Затем вызывающий абонент делает что-то вроде этого, чтобы получить два возвращенных параметра *arrayrefs* :

```
my ($list1, $list2) = foo(...);
```

Использование arrayref для передачи массива в sub

Массивref для `@foo` - `\@foo` . Это удобно, если вам нужно передать массив и другие вещи подпрограмме. Передача `@foo` подобна передаче нескольких скаляров. Но передача `\@foo` является единственным скаляром. Внутри подпрограммы:

```
xyz(\@foo, 123);
...
sub xyz {
    my ($arr, $etc) = @_;
    print $arr->[0]; # using the first item in $arr. It is like $foo[0]
```

Хеш как список

В контексте списка хеш сглажен.

```
my @bar =  ( %hash, %hash );
```

Массив `@bar` инициализируется списком двух `%hash` хешей

- оба `%hash` сегмента сглажены
- новый список создается из сплющенных элементов
- `@bar` array инициализируется этим списком

Гарантируется, что пары ключ-значение объединяются. Ключи всегда индексируются, значения - нечетные. Не гарантируется, что пары ключ-значение всегда сплющены в одном порядке:

```
my %hash =  ( a => 1, b => 2 );
print %hash; # Maybe 'a1b2' or 'b2a1'
```

Прочитайте Списки онлайн: https://riptutorial.com/ru/perl/topic/4553/списки

глава 31: Строки и методы цитирования

замечания

Синтаксис версии не позволяет нам защищать версии, которые еще не существуют, поэтому это напоминание о том, что кто-то может вернуться и отредактировать их, когда он приземлится (RE: Perl 5.26). У охранников версии скорее должна быть «будущая» классификация для предварительных функций, которые могут быть доступны для людей, достаточно храбрых, чтобы выполнять проверку исходного кода.

Examples

Строковое литературное цитирование

Строковые литералы не подразумевают экранирования или интерполяции (за исключением цитирования строковых терминаторов)

```
print 'This is a string literal\n'; # emits a literal \ and n to terminal

print 'This literal contains a \'postraphe '; # emits the ' but not its preceding \
```

Вы можете использовать альтернативные механизмы цитирования, чтобы избежать столкновений:

```
print q/This is is a literal \' <-- 2 characters /;  # prints both \ and '
print q^This is is a literal \' <-- 2 characters ^;  # also
```

Некоторые выбранные символы кавычек являются «сбалансированными»,

```
print q{ This is a literal and I contain { parens! } }; # prints inner { }
```

Дважды процитировать

Строки с двойными кавычками используют **интерполяцию** и **экранирование** - в отличие от строк с одной кавычкой. Для того, чтобы двойные кавычки строка, либо использовать двойные кавычки `"` или `qq` оператор.

```
my $greeting = "Hello!\n";
print $greeting;
# => Hello! (followed by a linefeed)

my $bush = "They misunderestimated me."
print qq/As Bush once said: "$bush"\n/;
# => As Bush once said: "They misunderestimated me." (with linefeed)
```

`qq` полезен здесь, чтобы избежать необходимости избегать кавычек. Без этого нам пришлось бы писать ...

```
print "As Bush once said: \"$bush\"\n";
```

... который просто не так хорош.

Perl не ограничивает вас использованием косой черты `/` с `qq` ; вы можете использовать любой (видимый) символ.

```
use feature 'say';

say qq/You can use slashes.../;
say qq{...or braces...};
say qq^...or hats...^;
say qq|...or pipes...|;
# say qq ...but not whitespace. ;
```

Вы также можете интерполировать массивы в строки.

```
use feature 'say';

my @letters = ('a', 'b', 'c');
say "I like these letters: @letters.";
# => I like these letters: a b c.
```

По умолчанию значения разделяются пробелами - поскольку специальная переменная `$"` умолчанию имеет одно пространство. Это, конечно, может быть изменено.

```
use feature 'say';

my @letters = ('a', 'b', 'c');
{local $" = ", "; say "@letters"; }    # a, b, c
```

Если вы предпочитаете, вы можете `use English` и вместо этого изменить `$LIST_SEPARATOR` :

```
use v5.18; # English should be avoided on older Perls
use English;

my @letters = ('a', 'b', 'c');
{ local $LIST_SEPARATOR = "\n"; say "My favourite letters:\n\n@letters" }
```

Для чего-то более сложного, чем это, вы должны использовать цикл.

```
say "My favourite letters:";
say;
for my $letter (@letters) {
  say " - $letter";
}
```

Интерполяция *не* работает с хешей.

```
use feature 'say';

my %hash = ('a', 'b', 'c', 'd');
say "This doesn't work: %hash"         # This doesn't work: %hash
```

Некоторый код злоупотребляет интерполяцией ссылок - **избегайте этого** .

```
use feature 'say';

say "2 + 2 == @{[ 2 + 2 ]}";           # 2 + 2 = 4 (avoid this)
say "2 + 2 == ${\( 2 + 2 )}";          # 2 + 2 = 4 (avoid this)
```

Так называемый «оператор тележки» вызывает perl для разыменования `@{ ... }` ссылки на массив `[ ... ]` которая содержит выражение, которое вы хотите интерполировать, `2 + 2` . Когда вы используете этот трюк, Perl создает анонимный массив, затем разыгрывает его и отбрасывает.

Версия `${\( ... )}` несколько менее расточительна, но по-прежнему требует выделения памяти, и ее еще труднее читать.

Вместо этого подумайте о написании:

- `say "2 + 2 == " . 2 + 2;`
- `my $result = 2 + 2; say "2 + 2 == $result"`

Heredocs

Большие многострочные строки обременительны для записи.

```
my $variable = <<'EOF';
this block of text is interpreted literally,
no \'quotes matter, they're just text
only the trailing left-aligned EOF matters.
EOF
```

NB: Убедитесь, что вы игнорируете подсветку синтаксиса синтаксиса стека: это очень неправильно.

И Interpolated Heredocs работают одинаково.

```
my $variable = <<"I Want it to End";
this block of text is interpreted.
quotes\nare interpreted, and $interpolations
get interpolated...
but still, left-aligned "I Want it to End" matters.
I Want it to End
```

Ожидается в 5.26.0 * - это синтаксис с отступом Heredoc, который выравнивает слева от вас

5.26.0

```
my $variable = <<~"MuchNicer";
    this block of text is interpreted.
    quotes\nare interpreted, and $interpolations
    get interpolated...
    but still, left-aligned "I Want it to End" matters.
MuchNicer
```

Удаление завершающих строк

Функция `chomp` удалит *один* символ новой строки, если присутствует, от каждого переданного к нему скаляра. `chomp` будет мутировать исходную строку и вернет количество удаленных символов

```
my $str = "Hello World\n\n";
my $removed = chomp($str);
print $str;     # "Hello World\n"
print $removed; # 1

# chomp again, removing another newline
$removed = chomp $str;
print $str;     # "Hello World"
print $removed; # 1

# chomp again, but no newline to remove
$removed = chomp $str;
print $str;     # "Hello World"
print $removed; # 0
```

Вы также можете `chomp` несколько строк за один раз:

```
my @strs = ("Hello\n", "World!\n\n"); # one newline in first string, two in second

my $removed = chomp(@strs); # @strs is now  ("Hello", "World!\n")
print $removed;             # 2

$removed = chomp(@strs); # @strs is now ("Hello", "World!")
print $removed;          # 1

$removed = chomp(@strs); # @strs is still ("Hello", "World!")
print $removed;          # 0
```

Но, как правило, никто не беспокоится о том, сколько новых строк было удалено, поэтому `chomp` обычно рассматривается в контексте void и обычно из-за чтения строк из файла:

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

    # now do something with $line
}

my @lines = readline $fh2;
```

```
chomp (@lines); # remove newline from end of each line
```

Прочитайте Строки и методы цитирования онлайн: https://riptutorial.com/ru/perl/topic/1984/строки-и-методы-цитирования

глава 32: танцовщик

Вступление

Около:

Dancer2 (преемник Dancer) - простая, но мощная структура веб-приложений для Perl.

Он вдохновлен Синатрой и написан Алексисом Сукрием.

Основные возможности: ••• Dead Simple - интуитивный, минималистический и очень выразительный синтаксис. ••• Гибкость - поддержка PSGI, плагины и модульная конструкция обеспечивают высокую масштабируемость. ••• Несколько зависимостей. Танцовщица зависит от максимально возможного количества модулей CPAN, что упрощает их установку.

Examples

Самый простой пример

```
#!/usr/bin/env perl
use Dancer2;

get '/' => sub {
    "Hello World!"
};

dance;
```

Прочитайте танцовщик онлайн: https://riptutorial.com/ru/perl/topic/5921/танцовщик

глава 33: Текст атрибута

Examples

Печать цветного текста

```
#!/usr/bin/perl

use Term::ANSIColor;

print color("cyan"), "Hello", color("red"), "\tWorld", color("green"), "\tIt's Me!\n",
color("reset");
```

Прочитайте Текст атрибута онлайн: https://riptutorial.com/ru/perl/topic/5922/текст-атрибута

глава 34: Тестирование Perl

Examples

Пример тестирования модуля Perl

Ниже приведен простой пример тестового сценария Perl, который дает некоторую структуру, позволяющую тестировать другие методы в тестируемом классе / пакете. Сценарий создает стандартный вывод с простым текстом «ok» / «not ok», который называется TAP (Test Anything Protocol).

Как правило, команда подтверждения запускает скрипт (сценарии) и суммирует результаты теста.

```
#!/bin/env perl
# CPAN
use Modern::Perl;
use Carp;
use Test::More;
use Test::Exception;
use Const::Fast;

# Custom
BEGIN { use_ok('Local::MyPackage'); }

const my $PACKAGE_UNDER_TEST => 'Local::MyPackage';

# Example test of method 'file_type_build'
sub test_file_type_build {
    my %arg    = @_;
    my $label  = 'file_type_build';
    my $got_file_type;
    my $filename = '/etc/passwd';

    # Check the method call lives
    lives_ok(
        sub {
            $got_file_type = $PACKAGE_UNDER_TEST->file_type_build(
                filename => $filename
            );
        },
        "$label - lives"
    );

    # Check the result of the method call matches our expected result.
    like( $got_file_type, qr{ASCII[ ]text}ix, "$label - result" );
    return;
} ## end sub test_file_type_build

# More tests can be added here for method 'file_type_build', or other methods.


MAIN: {
```

```
    subtest 'file_type_build' => sub {
       test_file_type_build();
       # More tests of the method can be added here.
       done_testing();
    };

    # Tests of other methods can be added here, just like above.


    done_testing();
} ## end MAIN:
```

Лучшая практика

Тест-скрипт должен тестировать только один пакет / класс, но для проверки пакета / класса может использоваться множество скриптов.

Дальнейшее чтение

- Test :: More - Основные тестовые операции.
- Test :: Exception - тестирование исключений.
- Test :: Differences - Сравнение результатов тестов с сложными структурами данных.
- Test :: Class - Class, а не скрипт. Сходство с Юнит.
- Учебники по тестированию Perl - Дальнейшее чтение.

Прочитайте Тестирование Perl онлайн: https://riptutorial.com/ru/perl/topic/5918/тестирование-perl

глава 35: Упаковка и распаковка

Examples

Вручную преобразует синтаксис C Structs to Pack

Если вы когда-либо сталкиваетесь с C Binary API от Perl Code, используя функции `syscall` , `ioctl` или `fcntl` , вам нужно знать, как построить память на совместимом с C пути.

Например, если вы когда-либо имели дело с какой-либо функцией, которая ожидала `timespec` , вы бы заглянули в `/usr/include/time.h` и находили:

```
struct timespec
{
    __time_t tv_sec;            /* Seconds.  */
    __syscall_slong_t tv_nsec;  /* Nanoseconds.  */
};
```

Вы танцуете с `cpp` чтобы найти, что это на самом деле означает:

```
cpp -E /usr/include/time.h -o /dev/stdout | grep __time_t
# typedef long int __time_t;
cpp -E /usr/include/time.h -o /dev/stdout | grep __syscall_slong_t
# typedef long int __syscall_slong_t
```

Таким образом, его (подписанный) int

```
echo 'void main(){ printf("%#lx\n", sizeof(__syscall_slong_t)); }' |
  gcc -x c -include stdio.h -include time.h - -o /tmp/a.out && /tmp/a.out
# 0x8
```

И это занимает 8 байтов. Итак, 64-битный подписанный int. И я на 64-битном процессоре. знак равно

Perldoc `pack` говорит

```
            q  A signed quad (64-bit) value.
```

Поэтому, чтобы упаковать спецификацию времени:

```
sub packtime {
    my ( $config ) = @_;
    return pack 'qq', @{$config}{qw( tv_sec tv_nsec )};
}
```

И для распаковки времени:

```
sub unpacktime {
   my ( $buf ) = @_;
   my $out = {};
   @{$out}{qw( tv_sec tv_nsec )} = unpack 'qq', $buf;
   return $out;
}
```

Теперь вы можете просто использовать эти функции.

```
my $timespec = packtime({ tv_sec => 0, tv_nsec => 0 });
syscall(  ..., $timespec ); # some syscall that reads timespec

later ...
syscall( ..., $timespec ); # some syscall that writes timespec
print Dumper( unpacktime( $timespec ));
```

Построение заголовка IPv4

Иногда вам приходится иметь дело со структурами, определенными в терминах типов данных C из Perl. Одним из таких приложений является создание сырых сетевых пакетов, если вы хотите сделать что-то более интересное, чем предлагать обычный API сокетов. Это как раз то, что для `pack()` (и для `unpack()`).

Обязательная часть IP-заголовка - 20 октетов (AKA «байты»). Как вы можете видеть позади этой ссылки, IP-адрес источника и получателя составляет два последних 32-битных значения в заголовке. Среди других полей есть некоторые из них с 16 битами, некоторые с 8 битами и несколько меньших фрагментов между 2 и 13 бит.

Предполагая, что в наш заголовок есть следующие переменные:

```
my ($dscp, $ecn, $length,
    $id, $flags, $frag_off,
    $ttl, $proto,
    $src_ip,
    $dst_ip);
```

Обратите внимание, что отсутствуют три поля из заголовка:

- Версия всегда 4 (в конце концов, это IPv4)
- В нашем примере IHL составляет 5, поскольку у нас нет поля *опций* ; длина указана в единицах 4 октета, поэтому 20 октетов дают длину 5.
- Контрольную сумму можно оставить равной 0. На самом деле нам придется ее вычислять, но код для этого здесь не касается.

Мы могли бы попытаться использовать битовые операции для построения, например, первых 32 бит:

```
my $hdr = 4 << 28 | 5 << 24 | $dscp << 18 | $ecn << 16 | $length;
```

Этот подход работает только до размера целого числа, хотя обычно это 64 бита, но может быть как минимум 32. Хуже того, это зависит от консистенции процессора, поэтому он будет работать на некоторых процессорах и терпеть неудачу на других. Попробуем `pack()` :

```
my $hdr = pack('H2B8n', '45', sprintf("%06b%02b", $dscp, $ecn), $length);
```

Сначала в шаблоне задается `H2` , *двухсимвольная строка* с *двумя символами, с высокой степенью nybble* . Соответствующим аргументом для пакета является «45» -версия 4, длина 5. Следующий шаблон - это `B8` , *8-разрядная битовая строка, нисходящий бит-порядок внутри каждого байта* . Нам нужно использовать битовые строки для управления компоновкой до кусков, меньших, чем nybble (4 бит), поэтому `sprintf()` используется для построения такой битовой строки из 6 бит из `$dscp` и 2 из `$ecn` . Последним является `n` , *16-разрядное значение без знака в байтах сети* , т. Е. Всегда big-endian, независимо от того, какой у вас собственный целочисленный формат вашего процессора, и он заполняется из `$length` .

Это первые 32 бита заголовка. Остальное можно построить аналогично:

шаблон	аргументация	замечания
`n`	`$id`	
`B16`	`sprintf("%03b%013b", $flags, $frag_off)`	То же, что и DSCP / ECN
`C2`	`$ttl, $proto`	Два последовательных беззнаковых октета
`n`	`0` / `$checksum`	`x` может использоваться для вставки нулевого байта, но `n` позволяет нам указать аргумент, если мы выберем для вычисления контрольной суммы
`N2`	`$src_ip, $dst_ip`	используйте `a4a4` чтобы упаковать результат двух вызовов `gethostbyname()` поскольку он уже находится в сетевом байтовом порядке!

Таким образом, полный вызов пакета заголовка IPv4 будет следующим:

```
my $hdr = pack('H2B8n2B16C2nN2',
    '45', sprintf("%06b%02b", $dscp, $ecn), $length,
    $id, sprintf("%03b%013b", $flags, $frag_off),
    $ttl, $proto, 0,
    $src_ip, $dst_ip
);
```

Прочитайте Упаковка и распаковка онлайн: https://riptutorial.com/ru/perl/topic/1983/упаковка-и-распаковка

глава 36: Установка Perl

Вступление

Я начну с процесса в Ubuntu, затем в OS X и, наконец, в Windows. Я не тестировал его на всех версиях perl, но это должен быть аналогичный процесс.

Используйте Perlbrew, если вы хотите легко переключаться между разными версиями Perl.

Я хочу сказать, что этот учебник посвящен Perl в его версии с открытым исходным кодом. Существуют и другие версии, такие как `activeperl` которые имеют свои преимущества и недостатки, которые не входят в этот учебник.

Examples

Linux

Существует несколько способов сделать это:

- Использование диспетчера пакетов:

```
sudo apt install perl
```

- Установка из источника:

```
wget http://www.cpan.org/src/5.0/perl-version.tar.gz
tar -xzf perl-version.tar.gz
cd perl-version
./Configure -de
make
make test
make install
```

- Установка в вашем домашнем каталоге $ (не требуется sudo) с Perlbrew :

```
wget -O - https://install.perlbrew.pl | bash
```

  См. Также Perlbrew

OS X

Существует несколько вариантов:

- Перлбрю :

```
# You need to install Command Line Tools for Xcode
curl -L https://install.perlbrew.pl | bash
```

- Perlbrew с поддержкой резьбы:

```
# You need to install Command Line Tools for Xcode
curl -L https://install.perlbrew.pl | bash
```

После установки perlbrew, если вы хотите установить Perl с поддержкой потоков, просто запустите:

```
perlbrew install -v perl-5.26.0 -Dusethreads
```

- Из источника:

```
tar -xzf perl-version.tar.gz
cd perl-version
./Configure -de
make
make test
make install
```

Windows

- Как мы уже говорили, мы идем с версией с открытым исходным кодом. Для Windows вы можете выбрать `strawberry` или `DWIM` . Здесь мы рассмотрим версию `strawberry` , поскольку на ней основан `DWIM` . Легкий путь здесь устанавливается с официального исполняемого файла .

См. Также berrybrew - perlbrew для Windows Strawberry Perl

Прочитайте Установка Perl онлайн: https://riptutorial.com/ru/perl/topic/9317/установка-perl

глава 37: Установка модулей Perl через CPAN

Examples

Запустите Perl CPAN в вашем терминале (Mac и Linux) или в командной строке (Windows)

Командная строка

Вы можете использовать `cpan` для установки модулей непосредственно из командной строки:

```
cpan install DBI
```

За этим последует, возможно, много страниц вывода, описывающих, что именно он делает для установки модуля. В зависимости от установленных модулей он может приостановить и задать вам вопросы.

Интерактивная оболочка

Вы также можете ввести «оболочку», таким образом:

```
perl -MCPAN -e "shell"
```

Он будет производить выход, как показано ниже:

```
Terminal does not support AddHistory.

cpan shell -- CPAN exploration and modules installation (v2.00)
Enter 'h' for help.

cpan[1]>
```

Затем вы можете установить модули, которые вы хотите, с помощью простой команды `install <module>` .

Пример: `cpan[1]> install DBI`

После успешной установки введите `exit` для выхода.

Установка модулей вручную

Если у вас нет прав на установку модулей perl, вы все равно можете их установить вручную, указав собственный путь, на котором у вас есть права на запись.

Fist, скачать и распаковать архив модулей:

```
wget module.tar.gz
tar -xzf module.tar.gz
cd module
```

Затем, если дистрибутив модуля содержит файл `Makefile.PL` , запустите:

```
perl Makefile.PL INSTALL_BASE=$HOME/perl
make
make test
make install
```

или если у вас есть файл `Build.PL` вместо `Makefile.PL` :

```
perl Build.PL --install_base $HOME/perl
perl Build
perl Build test
perl Build install
```

Вы также должны включить путь модуля в переменную окружения `PERL5LIB` , чтобы использовать его в своем коде:

```
export PERL5LIB=$HOME/perl
```

cpanminus, облегченная замена без замены для cpan

использование

Чтобы установить модуль (при условии, что `cpanm` уже установлен):

```
cpanm Data::Section
```

`cpanm` («cpanminus») стремится быть менее подробным, чем `cpan` но при этом `cpan` всю информацию об установке в файле журнала в случае необходимости. Он также обрабатывает многие «интерактивные вопросы» для вас, в то время как `cpan` - нет.

`cpanm` также популярен для установки зависимостей проекта, например, от GitHub. Типичным использованием является первый `cd` в корне проекта, затем выполняется

```
cpanm --installdeps .
```

С `--installdeps` он будет:

1. Сканирование и установка зависимостей *configure_requires* от
    - META.json
    - META.yml (если отсутствует META.json)
2. Создайте проект (эквивалент `perl Build.PL`), создав файлы MYMETA
3. Для сканирования и установки *требуются* зависимости от
    - MYMETA.json
    - MYMETA.yml (если отсутствует MYMETA.json)

Чтобы указать файл «some.cpanfile», содержащий зависимости, запустите:

```
cpanm --installdeps --cpanfile some.cpanfile .
```

Установка `cpanm`

Существует несколько способов установить его . Вот установка через `cpan` :

```
cpan App::cpanminus
```

Конфигурация `cpanm`

Там **нет** файла конфигурации для `cpanm` . Скорее, он полагается на следующие переменные среды для своей конфигурации:

- `PERL_CPANM_OPT` (общие параметры командной строки cpanm)
    - `export PERL_CPANM_OPT="--prompt"` # в .bashrc, чтобы включить подсказку, например
    - `setenv PERL_CPANM_OPT "--prompt"` # в .tcshrc
- `PERL_MM_OPT` (ExtUtils :: Параметры командной строки MakeMaker, влияет на установку установки модуля)
- `PERL_MB_OPT` (Module :: Build параметры командной строки, влияет на цель установки модуля)

Прочитайте Установка модулей Perl через CPAN онлайн:
https://riptutorial.com/ru/perl/topic/3542/установка-модулей-perl-через-cpan

глава 38: Файловый ввод-вывод (чтение и запись файлов)

параметры

Режим	Explaination
`>`	**Напишите (trunc)** . Перезапишет существующие файлы. Создает новый файл, если файл не найден
`>>`	**Напишите (добавьте)** . Не перезаписывайте файлы, а добавьте новый контент в конце. Будет также создан файл, если он используется для открытия не существующего файла.
`<`	**Прочитайте** . Открывает файл в режиме только для чтения.
`+<`	**Чтение** / **запись** . Не будет создавать или обрезать файл.
`+>`	**Чтение** / **запись (trunc)** . Будет создавать и усекать файл.
`+>>`	**Чтение** / **запись (добавление)** . Будет создавать, но не обрезать файл.

замечания

`chomp` часто используется при чтении из файла. По умолчанию он обрезает символ новой строки, хотя его полная функциональность относится к perldocs .

Остерегайтесь разницы между символами и байтами: не все кодировки, особенно UTF-8, используют 1-байтовые символы. Несмотря на то, что PerlIO обрабатывается почти безупречно, есть одна потенциальная ошибка:

- `read` использует *символы* для параметров *длины* и *смещения*
- `seek` и `tell` *всегда* использовать *байты* для позиционирования

Поэтому не используйте арифметику на основе этих смешанных значений. Вместо этого используйте, например, `Encode::encode('utf8',$value_by_read)` чтобы получить октеты (байты) из результата `read` , счет которых вы можете использовать с `tell` и `seek` .

Examples

Чтение из файла

```
my $filename = '/path/to/file';

open my $fh, '<', $filename or die "Failed to open file: $filename";

# You can then either read the file one line at a time...
while(chomp(my $line = <$fh>)) {
    print $line . "\n";
}

# ...or read whole file into an array in one go
chomp(my @fileArray = <$fh>);
```

Если вы знаете, что ваш входной файл UTF-8, вы можете указать кодировку:

```
open my $fh, '<:encoding(utf8)', $filename or die "Failed to open file: $filename";
```

После завершения чтения из файла дескриптор файла должен быть закрыт:

```
close $fh or warn "close failed: $!";
```

См. Также: Чтение файла в переменную

Другим и **более быстрым** способом чтения файла является использование модуля File :: Slurper. Это полезно, если вы работаете со многими файлами.

```
use File::Slurper;
my $file = read_text("path/to/file"); # utf8 without CRLF transforms by default
print $file; #Contains the file body
```

См. Также: [Чтение файла с помощью slurp]

Запись в файл

Этот код открывает файл для записи. Возвращает ошибку, если файл не может быть открыт. Также закрывает файл в конце.

```
#!/usr/bin/perl
use strict;
use warnings;
use open qw( :encoding(UTF-8) :std ); # Make UTF-8 default encoding

# Open "output.txt" for writing (">") and from now on, refer to it as the variable $fh.
open(my $fh, ">", "output.txt")
# In case the action failed, print error message and quit.
or die "Can't open > output.txt: $!";
```

Теперь у нас есть открытый файл, готовый для записи, к которому мы обращаемся через `$fh` (эта переменная называется *дескриптором файла*). Затем мы можем направлять вывод в этот файл с помощью оператора `print` :

```
# Print "Hello" to $fh ("output.txt").
print $fh "Hello";
# Don't forget to close the file once we're done!
close $fh or warn "Close failed: $!";
```

Оператор `open` имеет в качестве первого параметра скалярную переменную (`$fh` в этом случае). Поскольку он определен в операторе `open` он рассматривается как *дескриптор файла* . Второй параметр `">"` (больше) определяет, что файл открыт для записи. Последний параметр - это путь к файлу для записи данных.

Чтобы записать данные в файл, используется оператор `print` вместе с *дескриптором файла* . Обратите внимание, что в операторе `print` нет запятой между *дескриптором файла* и самим заявлением, просто пробелы.

Открытие файла FileHandle для чтения

Открытие общих текстовых файлов ASCII

5.6.0

```
open my $filehandle, '<', $name_of_file or die "Can't open $name_of_file, $!";
```

Это основная идиома для «default» File IO и делает `$filehandle` доступным для чтения входным потоком `bytes` , отфильтрованный стандартным системным декодером, который может быть локально установлен с `open` прагмой

Сам Perl не обрабатывает ошибки при открытии файла, поэтому вы должны сами обрабатывать их, проверяя условие выхода `open` . `$!` заполняется сообщением об ошибке, которое вызвало отказ.

В Windows декодер по умолчанию является фильтром «CRLF», который отображает последовательности «\ r \ n» на входе в «\ n»,

Открытие двоичных файлов

5.8.0

```
open my $filehandle, '<:raw', 'path/to/file' or die "Can't open $name_of_file, $!";
```

Это указывает, что Perl *не* должен выполнять трансляцию `CRLF` в Windows.

Открытие текстовых файлов UTF8

5.8.0

```
open my $filehandle, '<:raw:encoding(utf-8)', 'path/to/file'
   or die "Can't open $name_of_file, $!";
```

Это указывает, что Perl должен избегать `CRLF` перевода, а затем декодировать полученные байты в строки *символов* (внутренне реализуемые как массивы целых чисел, которые могут превышать 255) вместо строк *байтов*

Чтение и запись в файл

Прежде чем читать и писать текстовые файлы, вы должны знать, какую кодировку использовать. См. Документацию Perl Unicode для более подробной информации о кодировании . Здесь мы показываем настройку UTF-8 как кодировку и декодирование по умолчанию для `open` функции. Это делается с использованием `open` прагмы в верхней части вашего кода (сразу после `use strict;` режима и `use warnings;` было бы уместно):

```
use strict;
use warnings;
use open qw( :encoding(UTF-8) :std );
```

Функция `open` создает дескриптор файла, который используется для чтения и / или записи в файл. `open` функция имеет подпись

`open(FILEHANDLE, MODE, FILEPATH)` и возвращает ложное значение, если операция `open(FILEHANDLE, MODE, FILEPATH)` с ошибкой. Затем описание ошибки сохраняется в `$!` ,

чтение

```
#!/usr/bin/perl
use strict;
use warnings;
use open qw( :encoding(UTF-8) :std ); # Make UTF-8 default encoding

my $file_path = "/path/to/file";
open(my $file_handle, '<', $file_path) or die "Could not open file! $!";

while(my $row = <$file_handle>) {
    print chomp($row), "\n";
}

close $file_handle
      or warn "Close failed!";
```

Пишу

```
#!/usr/bin/perl
use strict;
use warnings;
use open qw( :encoding(UTF-8) :std ); # Make UTF-8 default encoding

my $file_path = "/path/to/file";
open(my $file_handle, '>', $file_path) or die "Could not open file! $!";
```

```
print $file_handle "Writing to a file";

close $file_handle
      or warn "Close failed!";
```

Чтение кусков

Открытие и чтение больших файлов может занять некоторое время и ресурсы. Если требуется только небольшая часть содержимого, может быть хорошей идеей прочитать содержимое в кусках, используя функцию `read` которая имеет подпись

```
read(FILEHANDLE, SCALAR, LENGTH, OFFSET)
```

`FILEHANDLE` должен быть открытым дескриптором файла, `SCALAR` будет удерживать прочитанные данные после операции. `LENGTH` определяет количество символов для чтения, начиная с `OFFSET` . Функция возвращает количество прочитанных символов, `0` если конец файла был достигнут, а `undef` в случае ошибки.

```
read($file_handle, $data, 16, 0);
```

Считывает 16 символов из начала файла в `$data` .

«использовать autodie», и вам не нужно будет проверять ошибки открытия / закрытия файла

`autodie` позволяет работать с файлами без явной проверки ошибок при открытии / закрытии.

Начиная с Perl 5.10.1, прагма `autodie` была доступна в ядре Perl. При использовании Perl автоматически проверяет наличие ошибок при открытии и закрытии файлов.

Ниже приведен пример, в котором все строки одного файла считываются, а затем записываются в конец файла журнала.

```
use 5.010;    # 5.010 and later enable "say", which prints arguments, then a newline
use strict;   # require declaring variables (avoid silent errors due to typos)
use warnings; # enable helpful syntax-related warnings
use open qw( :encoding(UTF-8) :std ); # Make UTF-8 default encoding
use autodie;  # Automatically handle errors in opening and closing files

open(my $fh_in, '<', "input.txt"); # check for failure is automatic

# open a file for appending (i.e. using ">>")
open( my $fh_log, '>>', "output.log"); # check for failure is automatic

while (my $line = readline $fh_in) # also works: while (my $line = <$fh_in>)
{
     # remove newline
```

```
    chomp $line;

    # write to log file
    say $fh_log $line or die "failed to print '$line'"; # autodie doesn't check print
}

# Close the file handles (check for failure is automatic)
close $fh_in;
close $fh_log;
```

Кстати, вы должны технически всегда проверять утверждения `print` . Многие люди этого не делают, но `perl` (интерпретатор Perl) не делает этого автоматически, и не делает `autodie` .

Перемотайте файл-дескриптор

Иногда бывает необходимо отступить после прочтения.

```
# identify current position in file, in case the first line isn't a comment
my $current_pos = tell;

while (my $line = readline $fh)
{
    if ($line =~ /$START_OF_COMMENT_LINE/)
    {
        push @names, get_name_from_comment($line);
    }
    else {
        last; # break out of the while loop
    }
    $current_pos = tell; # keep track of current position, in case we need to rewind the next
line read
}

# Step back a line so that it can be processed later as the first data line
seek $fh, $current_pos, 0;
```

Чтение и запись сжатых файлов gzip

Запись gzip-файла

Чтобы написать gzip-файл, `use` модуль `IO::Compress::Gzip` и создайте дескриптор файла, создав новый экземпляр `IO::Compress::Gzip` для желаемого выходного файла:

```
use strict;
use warnings;
use open qw( :encoding(UTF-8) :std ); # Make UTF-8 default encoding

use IO::Compress::Gzip;

my $fh_out = IO::Compress::Gzip->new("hello.txt.gz");
```

```
print $fh_out "Hello World!\n";

close $fh_out;

use IO::Compress::Gzip;
```

Чтение из gzip-файла

Чтобы прочитать из gzip-файла, `use` модуль `IO::Uncompress::Gunzip` а затем создайте дескриптор файла, создав новый экземпляр `IO::Uncompress::Gunzip` для входного файла:

```
#!/bin/env perl
use strict;
use warnings;
use open qw( :encoding(UTF-8) :std ); # Make UTF-8 default encoding

use IO::Uncompress::Gunzip;

my $fh_in = IO::Uncompress::Gunzip->new("hello.txt.gz");

my $line = readline $fh_in;

print $line;
```

Установка кодировки по умолчанию для ввода-вывода

```
# encode/decode UTF-8 for files and standard input/output
use open qw( :encoding(UTF-8) :std );
```

Эта `pragma` изменяет режим чтения и записи текста (файлы, стандартный ввод, стандартный вывод и стандартную ошибку) по умолчанию в UTF-8, что обычно требуется при написании новых приложений.

ASCII - это подмножество UTF-8, поэтому это не вызовет проблем с устаревшими файлами ASCII и поможет защитить вас от случайного повреждения файлов, которое может произойти при обработке файлов UTF-8 как ASCII.

Тем не менее, важно знать, что такое кодирование ваших файлов, с которыми вы имеете дело, и обрабатывать их соответственно. (Причины того, что мы не должны игнорировать Unicode.) Для более глубокого изучения Unicode обратитесь к теме Unicode Perl .

Прочитайте Файловый ввод-вывод (чтение и запись файлов) онлайн: https://riptutorial.com/ru/perl/topic/1604/файловый-ввод-вывод--чтение-и-запись-файлов-

глава 39: хаотичность

замечания

Документация для функции rand () perl: http://perldoc.perl.org/functions/rand.html

Examples

Создайте случайное число от 0 до 100

Передайте верхний предел в качестве аргумента функции rand ().

Входные данные:

```
my $upper_limit = 100;
my $random = rand($upper_limit);

print $random . "\n";
```

Выход:

Случайное число с плавающей запятой, например ...

```
45.8733038119139
```

Генерировать случайное целое число от 0 до 9

Передайте произвольное число с плавающей запятой как int.

Входные данные:

```
my $range = 10;

# create random integer as low as 0 and as high as 9
my $random = int(rand($range));   # max value is up to but not equal to $range

print $random . "\n";
```

Выход:

Случайное целое число, например ...

```
0
```

См. Также perldoc для rand .

Доступ к элементу массива случайным образом

```
my @letters = ( 'a' .. 'z' );                 # English ascii-bet

print $letters[ rand @letters ] for 1 .. 5;  # prints 5 letters at random
```

Как это устроено

- `rand EXPR` ожидает скалярное значение, поэтому `@letters` вычисляется в скалярном контексте
- Массив в скалярном контексте возвращает количество содержащихся в нем элементов (26 в этом случае)
- `rand 26` возвращает случайное дробное число в интервале `0 ≤ VALUE < 26` . (Это никогда не может быть `26`)
- Индексы массива всегда целые числа, поэтому `$letters[rand @letters]` ≡ `$letters[int rand @letters]`
- Массивы Perl индексируются с нулевым значением, поэтому `$array[rand @array]` возвращает `$array[0]` , `$array[$#array]` или элемент между

(Тот же принцип применяется к хэшам)

```
my %colors = ( red   => 0xFF0000,
               green => 0x00FF00,
               blue  => 0x0000FF,
             );

print ( values %colors )[rand keys %colors];
```

Прочитайте хаотичность онлайн: https://riptutorial.com/ru/perl/topic/6905/хаотичность

глава 40: Чтение содержимого файла в переменную

Examples

Ручной способ

```
open my $fh, '<', $filename
    or die "Could not open $filename for reading: $!";
my $contents = do { local $/; <$fh> };
```

После открытия файла (прочитайте `man perlio` если вы хотите читать определенные кодировки файлов вместо необработанных байтов), трюк находится в блоке `do` : `<$fh>` , дескриптор файла в алмазном операторе, возвращает одну запись из файла , Переменная «input record separator» `$/` указывает, что такое «запись» - по умолчанию она установлена на символ новой строки, поэтому «запись» означает «одна строка». Поскольку `$/` - глобальная переменная, `local` выполняет две вещи: создает временную локальную копию `$/` которая будет исчезать в конце блока, и дает ему (не) значение `undef` (значение, которое Perl дает к неинициализированным переменным). Когда разделитель входных данных имеет это (не) значение, оператор алмаза вернет весь файл. (Он рассматривает весь файл как одну строку.)

Используя `do` , вы можете даже вручную открыть файл. Для повторного чтения файлов,

```
sub readfile { do { local(@ARGV,$/) = $_[0]; <> } }
my $content = readfile($filename);
```

может быть использован. Здесь другая глобальная переменная (`@ARGV`) локализована для моделирования того же процесса, который используется при запуске скрипта perl с параметрами. `$/` все еще `undef` , так как массив перед ним «ест» все входящие аргументы. Затем оператор алмаза `<>` снова поставляет одну запись, определяемую `$/` (весь файл), и возвращает из блока `do` , которые, в свою очередь, возвращаются из суб.

Sub не имеет явной обработки ошибок, что является плохой практикой! Если во время чтения файла возникает ошибка, вы получите `undef` как возвращаемое значение, а не пустую строку из пустого файла.

Другим недостатком последнего кода является тот факт, что вы не можете использовать PerlIO для разных кодировок файлов - вы всегда получаете необработанные байты.

Путь :: Крошечный

Использование идиомы из Manual Way несколько раз в сценарии скоро становится утомительным, поэтому вы можете попробовать модуль.

```
use Path::Tiny;
my $contents = path($filename)->slurp;
```

Вы можете передать опцию `binmode` если вам нужно контролировать кодировки файлов, окончания строк и т. Д. - см. `man perlio` :

```
my $contents = path($filename)->slurp( {binmode => ":encoding(UTF-8)"} );
```

`Path::Tiny` также имеет множество других функций для работы с файлами, поэтому он может быть хорошим выбором.

Файл :: Slurper

Это минималистский модуль, который только вставляет файлы в переменные, и ничего больше.

```
use File::Slurper 'read_text';
my $contents = read_text($filename);
```

`read_text()` принимает два необязательных параметра для указания кодировки файла и должны ли переводы строк быть переведены между стандартами UNIX Unixish или DOSish CRLF:

```
my $contents = read_text($filename, 'UTF-8', 1);
```

Файл :: Slurp

Не используйте его. Хотя он существует уже давно и по-прежнему является модулем, который, по мнению большинства программистов, будет нарушен и вряд ли будет исправлен .

Разбиение файла на переменную массива

```
open(my $fh, '<', "/some/path") or die $!;
my @ary = <$fh>;
```

При оценке в контексте списка оператор алмаза возвращает список, состоящий из всех строк в файле (в этом случае присваивание результата контексту списка источников массива). Терминатор линии сохраняется и может быть удален путем прерывания:

```
chomp(@ary); #removes line terminators from all the array elements.
```

Файл Slurp в одном слое

Разделитель входных данных может быть задан с помощью ключа `-0` (*ноль* , а не *капитала O*). В качестве значения берется восьмеричное или шестнадцатеричное число. Любое значение `0400` или выше приведет к тому, что Perl будет клонировать файлы, но, по договоренности, значение, используемое для этой цели, составляет `0777` .

```
perl -0777 -e 'my $file = <>; print length($file)' input.txt
```

Идя далее с минимализмом, указание `-n` switch заставляет Perl автоматически читать каждую строку (в нашем случае - весь файл) в переменную `$_` .

```
perl -0777 -ne 'print length($_)' input.txt
```

Прочитайте Чтение содержимого файла в переменную онлайн:
https://riptutorial.com/ru/perl/topic/1779/чтение-содержимого-файла-в-переменную

кредиты

S. No	Главы	Contributors
1	Начало работы с языком Perl	Alan Haggai Alavi, choroba, Christopher Bottoms, Community, datageist, Denis Ibaev, eddy85br, Eugen Konkov, Jon Ericson, Leon Timmermans, oals, Pro Q, rlandster, xfix
2	Perl однострочные	Dmitry Egorov, Eugen Konkov, Kemi, mbethke, zb226
3	Perlbrew	Håkon Hægland
4	Unicode	Håkon Hægland, Kemi, Kent Fredric, mbethke
5	Даты и время	Ngoan Tran, waghso
6	Интерполяция в Perl	oals, Ruslan Batdalov
7	Команды Perl для Windows Excel с помощью модуля Win32 :: OLE	Jean-Francois T.
8	Комментарии	4444, Christopher Bottoms, lanti, Rebecca Close
9	Компиляция модуля cpan perl sapnwrfc из исходного кода	flotux
10	Контрольные заявления	callyalater, Christopher Bottoms, oals, Stephen Leppik
11	Лучшие практики	fifaltra, interduo
12	Обработка исключений	badp, simbabque
13	Объектно-ориентированный Perl	badp, Dmitry Egorov, Ruslan Batdalov, simbabque
14	Оптимизация использования памяти	mbethke

15	Отладка скриптов Perl	4444, Eugen Konkov
16	Отладочный вывод	Ataul Haque, Christopher Bottoms, Joe, simbabque, waghso
17	Пакеты и модули	AntonH, Christopher Bottoms, John Hart, Jon Ericson, Kemi, Kent Fredric, lepe, mbethke
18	переменные	Ataul Haque, badp, digitalis_, dmvrtx, Eugen Konkov, Håkon Hægland, interduo, Jon Ericson, Kent Fredric, mbethke, Mik, nfanta, oals, Otterbein, zb226
19	подпрограммы	badp, Christopher Bottoms, dave, digitalis_, interduo, mbethke, Michael Carman, msh210, Wolf, xfix, xtreak
20	Правда и ложь	badp, Bill the Lizard, Christopher Bottoms, ikegami, Kent Fredric , mbethke, msh210, Ole Tange, xfix
21	Приложения GUI в Perl	oldtechaa
22	Простое взаимодействие с базой данных через модуль DBI	Ngoan Tran
23	Простой способ проверить установленные модули на Mac и Ubuntu	fanlim, Ngoan Tran,
24	Разбор XML	cbmckay, Drav Sloan, eballes, Sobrique
25	Разделить строку на неупорядоченные разделители	DVK, Ian Praxil, serenesat
26	Регулярные выражения	Al.G., Jon Ericson, rlandster, SajithP, Sarwesh Suman, Stephen Leppik
27	Сортировка	Jon Ericson, kjpires, mbethke
28	Специальные переменные	AbhiNickz, Denis Ibaev, oals
29	Списки	brian d foy, Christopher Bottoms, David Mertens, Denis Ibaev,

		DVK, Eugen Konkov, Muaaz Rafi, pwes, reflective_mind, Rick James, Wolf
30	Строки и методы цитирования	badp, Christopher Bottoms, Denis Ibaev, digitalis_, Kent Fredric, mbethke, svarog
31	танцовщик	Chankey Pathak, vanHoesel
32	Текст атрибута	SajithP
33	Тестирование Perl	nslntmnx
34	Упаковка и распаковка	Denis Ibaev, Kent Fredric, mbethke
35	Установка Perl	fanlim, flamey, Håkon Hægland, Iván Rodríguez Torres, luistm
36	Установка модулей Perl через CPAN	Christopher Bottoms, Kemi, luistm, Ngoan Tran, Peter Mortensen, Randall
37	Файловый ввод-вывод (чтение и запись файлов)	Christopher Bottoms, Denis Ibaev, Håkon Hægland, Kemi, Kent Fredric, matt freake, Nagaraju, rbennett485, SajithP, Sebi, SREagle, Tim Hallyburton, yonyon100
38	хаотичность	Christopher Bottoms, Rebecca Close, Zaid
39	Чтение содержимого файла в переменную	Alien Life Form, Christopher Bottoms, digitalis_, Jeff Y, Kemi, mbethke, mob, pwes, rlandster, SREagle