PRAGMATIC PERL

13

03/2014

pragmaticperl.com

Pragmatic Perl 13

pragmaticperl.com

Выпуск 13. Март 2014

Другие выпуски и форматы журнала всегда можно загрузить с `http://pragmaticperl.com`. С вопросами и предложениями пишите на editor@pragmaticperl.com.

Комментарии к каждой статье есть в html-версии. Подписаться на новые выпуски можно по ссылке pragmaticperl.com/subscribe.

Авторы статей: Владимир Леттиев, Дмитрий Шаматрин

Обложка: Марко Иваныk

Корректоры: Андрей Шитов

Выпускающий редактор: Вячеслав Тихановский

Ревизия: 2014-03-07 11:12

Оглавление

1. От редактора: Год журналу

Друзья! Нашему журналу исполняется ровно год с первого выпуска. Поздравляем вас и нас с этим событием. За целый год произошло многое: мы писали статьи, брали интервью, проводили конкурсы; вы писали письма, оставляли комментарии, советы.

Подведем итоги первого года.

За год мы выпустили без перерывов и вовремя 12 номеров, взяли интервью у 12 известных в Perl-сообществе программистов, авторов книг, организаторов конференций. В создании журнала поучаствовали 16 авторов. Совместными усилиями авторов, корректоров и редакторов подготовили 761 страницу текста!

Подписалось на рассылку по email — 875 человек, по rss — 173. Присоединяйтесь!

У сайта журнала было около 50 000 посещений, из которых 20 000 были уникальными. Скачало журнал 20 000 человек.

География (top 10):

- Россия — 29 053
- Украина — 11 143
- Беларусь — 1 392
- Казахстан — 862
- США — 849
- Нидерланды — 726
- Германия — 526
- Великобритания — 397
- Израиль — 321
- Литва — 240

Самые смешные комментарии, письма, анонсы оставленные на сайте журнала и на других интернет-ресурсах:

- *— Pragmatic Perl — первый в мире русскоязычный журнал, посвященный проблемам программирования на языке perl.*
- *— Первые 8 страниц журнала похожи на «плач Ярославны» и попытку откопать Perl.*
- *— Книга мёртвых.*
- *— Посмотрим, протянет ли сей журнал дольше, чем «Практика функционального программирования».*
- *— Вы серьёзно полагаете, что язык с закорючками перед именами переменных — инструмент профессионала?*
- *— Сейчас и посмотрим сколько людей на перле кодят. Пока что 71.*
- *— В хтмл версии очень нехватает выравнивания по ширине.*
- *— Никому не нужны умные люди.*
- *— Дайте Hello World уже для чайников, только чтобы он короче был чем на Пыхе.*
- *— А как в perl с юникодом?*
- *— Зачем Perl6 если уже пилят Perl7?*
- *— Не хватает картинок. И не надо смеяться, я серьезно.*
- *— Может вам накатать статью про хардкорный AnyEvent?*
- *— Маразматик Perl.*

Надеемся, что мы сможем отметить и следующий год журнала. Однако, нам никак не обойтись без поддержки наших читателей, то есть вас. Продолжайте читать, комментировать, участвовать в развитии журнала и мы будем становиться лучше!

Мы продолжаем искать авторов для следующих номеров. Если у вас есть идеи или желание помочь, пожалуйста, свяжитесь с нами.

Приятного чтения.

■ *Вячеслав Тихановский*

2. «Старые» современные возможности Perl

Расмотрены наиболее интересные, по мнению автора, особенности языка Perl

Perl это вообще весьма занятный язык программирования общего назначения. Многие говорят что он очень древний, в нем что-то не поддерживается, что это вообще убогий язык и нужно бы его закопать.

Новые языки программирования появляются постоянно. Вместе с этим добавляются новые аспекты программирования, новые архитектуры, новые модные приемы. А Perl как будто старый и не способен соревноваться.

Где-то давно я находил фразу приблизительно следующего содержания:

> Многие люди когда видят перл чувствуют резкий разрыв шаблонов от того, что язык старый, не распиаренный корпорациями умеет то же самое, что языки, корпорациями распиаренные. А иногда даже и больше. Вот именно это основная тема статьи.

В рамках этой статьи я покажу, как можно делать замечательные и модные вещи на великолепном, хотя и старом языке.

Анонимные функции

Анонимные функции это особый вид функций, которые объявляются в месте использования и не получают уникального идентификатора для доступа к ним. Обычно при создании они либо вызываются напрямую, либо ссылка на функцию присваивается переменной, с помощью которой затем можно косвенно вызывать данную функцию.

Анонимная функция создается проще простого:

```
my $subroutine = sub {
    my $msg = shift;
    printf "I am called wit message: %s\n", $msg;
    return 42;
};
# $subroutine теперь ссылается на анонимную функцию
$subroutine->("Oh, my message!");
```

Анонимные функции хорошо использовать как для создания блоков кода, так и для замыканий, которые, кстати, поддерживаются.

Замыкания

Как говорит нам википедия:

> Замыкание — это особый вид функции. Она определена в теле другой функции и создаётся каждый раз во время её выполнения. В записи это выглядит как функция, находящаяся целиком в теле другой функции. При этом вложенная внутренняя функция содержит ссылки на локальные переменные внешней функции. Каждый раз при выполнении внешней функции происходит создание нового экземпляра внутренней функции, с новыми ссылками на переменные внешней функции.

```
# возвращает ссылку на анонимную функцию
sub adder($) {
    my $x = shift;    # в котором x — свободная
       переменная,
    return sub ($) {
        my $y = shift;    # а y — связанная переменная
        return $x + $y;
    };
}

$add1 = adder(1);   # делаем процедуру для прибавления 1
print $add1->(10);  # печатает 11

$sub1 = adder(-1);  # делаем процедуру для вычитания 1
print $sub1->(10);  # печатает 9
```

Замыкания полезно использовать в той ситуации, когда необходимо получить функцию с уже готовыми параметрами, которые будут в ней сохранены. Или же, например, для генерации функции-парсера.

Неизменяемый объект

Нет ничего проще. Очень часто приходится видеть простыни текстов на форумах, как это реализовать. Многие «современные», «модные» языки не могут похвастаться настолько простым и изящным решением, как то, которое имеет Perl на такой случай. Фактически, `state` работает как `my`, у него одинаковая область видимости, но при этом переменная state НИКОГДА не будет переинициализирована в рамках программы. Это иллюстрируют два примера.

```
use feature 'state';
gimme_another();
gimme_another();
sub gimme_another {
    state $x;
    print ++$x, "\n";
}
```

Эта программа напечатает 1, затем 2.

```
use feature 'state';
gimme_another();
gimme_another();
sub gimme_another {
    my $x;
    print ++$x, "\n";
}
```

А эта 1, затем 1.

Бесскобочные функции

Я думаю, что это правильный перевод parentheses less.

Очень часто `print`, например, пишется и вызывается без скобок. Возникает вопрос, а можем ли мы тоже создавать такие функции?

Можем. Для этого у Perl есть даже специальная прагма — `subs`. Синтетический пример, в котором мы хотим сделать красивую проверку. Допустим, мы хотим напечатать OK, если определенная переменная `true`. Я позволю себе напомнить, что в Perl нет такого типа данных, как Boolean, также, как и в C нет такого типа данных, тогда как в место него выступает не ложное значение — например, 1.

```
use strict;
use subs qw/checkflag/;
my $flag = 1;
print "OK" if checkflag;

sub checkflag {
    return $flag;
}
```

Данная программа напечатает OK. Но это не единственный способ. Perl — довольно умен, а потому, если мы реструктуризируем нашу программу и приведем ее к такому виду:

```
use strict;
sub checkflag {
    return $flag;
}

my $flag = 1;
print "OK" if checkflag;
```

То результат будет тот же. Закономерность здесь слудеющая. Мы можем вызывать функцию без скобок в нескольких случаях:

- Используя прагму subs.
- Написав функцию ПЕРЕД ее вызовом.
- Использовать прототипы функций.

Рассмотрим, что же такое прототипы.

Прототипы функций

Для начала стоит выяснить, как вообще функции работают в Perl.

Есть, например, какая-то абстрактная себе функция, которая называется, допустим `my_sub`:

```
sub my_sub {
    print join ', ', @_;
}
```

Например, мы ее вызываем следующим образом:

```
my_sub(1, 2, 3, 4, 5);
```

Функция напечатает следующее:

```
1, 2, 3, 4, 5,
```

Таким образом мы можем передавать в ЛЮБУЮ функцию Perl любое количество аргументов. И пусть сама функция разбирается, что мы от нее хотели.

В функцию передается «текущий массив», контекстная переменная. Поэтому, запись вида:

```
sub ms {
    my $data = shift;
    print $data;
}
```

Эквивалентна:

```
sub ms {
    my $data = shift, @_;
    print $data;
}
```

Логично предположить, что должен быть механизм по ограничению количества переданных аргументов в функцию. Этот механизм и называется прототипами.

Функция Perl с прототипами будет выглядеть так:

```
sub my_sub($$;$) {
    my ($v1, $v2, $v3) = @_;
    $v3 ||= 'empty';
    printf("v1: %s, v2: %s, v3: %s\n", $v1, $v2, $v3);
}
```

Прототипы функций записываются после имени функции в круглых скобках. Прототип `$$;$` означает, что в качестве параметров ОБЯЗАНЫ присутствовать два скаляра и третий по желанию, `;` — отделяет обязательные параметры от возможных.

Теперь, если мы ее попробуем вызвать вот так:

```
my_sub();
```

То мы получим ошибку вида:

```
Not enough arguments for main::my_sub at pragmaticperl.pl
    line 7, near "()"
Execution of pragmaticperl.pl aborted due to compilation
   errors.
```

Однако, если вызвать вот так:

```
&my_sub();
```

То проверка прототипов не будет происходить.

Важный момент. Прототипы будут работать в следующих случаях:

- Если функция вызывается без знака амперсанда (`&`). Perlcritic, кстати, ругается на запись вызова функции через амперсанд.
- Если функция написана ПЕРЕД вызовом. Если мы сначала вызовем функцию, а потом ее напишем, при включенных `warnings` получим следующее предупреждение:

  main::my_sub() called too early to check prototype at pragmaticperl.pl line 4.

Правильная программа с прототипами выглядит так:

```
use strict;
use warnings;
use subs qw/my_sub/;

sub my_sub($$;$) {
    my ($v1, $v2, $v3) = @_;
    $v3 ||= 'empty';
    printf("v1: %s, v2: %s, v3: %s\n", $v1, $v2, $v3);
}
my_sub();
```

ООП

ООП у Perl странное. Кто-то говорит, что его нет, а кто-то говорит, что это лучшая реализация, т.к. она не мешает жить программисту, а позволяет делать сложные вещи возможными. Я для себя, например, еще не определился.

Что такое ООП в случае Perl? Это, так называемый, blessed hashref. Инкапсуляции, как таковой, в Perl нет. Есть сторонние модули, например, `Moose`, `Mouse`, которые предоставляю все возможности ООП Perl программисту ценой потери скорости.

Итак, простейший пакет.

```
use strict;
use Data::Dumper;
my $obj = Top->new();
print Dumper $obj;

$obj->top_sub();

package Top;
use strict;
sub new {
    my $class = shift;
    my $self = {};
    bless $self, $class;
    return $self;
}

sub top_sub {
    print "I am TOP sub\n";
```

```
}
1;
```

Данная программа выведет:

```
$VAR1 = bless( {}, 'Top' );
I am TOP sub
```

Собственно, ООП сводится к тому, что есть объект и он может выступать хранилищем состояний и вызывать методы пакета. Через стрелочку (`$obj->top_sub`).

Множественное наследование

Увы, как показыват практика и очень умные книги очень умных людей — если у вас в приложении появилась необходимость множественного наследования, то это приложение можно описать примерно следующим стишком.

> ваш код сегодняшний коллега напоминает даунхилл
> среди деревьев и говнища велосипед и костыли

Но, Perl нам позволяет сделать и это. Далее приведен пример программы, которая наследует два класса:

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

my $obj = Bottom->new();
print Dumper $obj;
# вызываем функции всех пакетов
$obj->top_sub();
$obj->top2_sub();
$obj->bottom_sub();

package Top;
use strict;
sub new {
    my $class = shift;
    my $self = {};
    bless $self, $class;
```

```
17      return $self;
18  }
19
20  sub top_sub {
21      print "I am TOP sub\n";
22  }
23
24  package Top2 {
25      sub new {
26          my $class = shift;
27          my $self = {};
28          bless $self, $class;
29          return $self;
30      }
31
32      sub top2_sub {
33          print "I am TOP2 sub\n";
34      }
35  }
36
37  package Bottom;
38  use strict;
39  # наследуем Top и Top2
40  use base qw/Top Top2/;
41
42  sub new {
43      my $class = shift;
44      my $self = {};
45      bless $self, $class;
46      return $self;
47  }
48
49  sub bottom_sub {
50      print "I am BOTTOM sub\n";
51  }
52
53  1;
```

В данном случае программа выведет:

```
1  I am TOP sub
2  I am TOP2 sub
3  I am BOTTOM sub
```

Оверрайд методов

Оверрайд довольно часто хорошая штука. Например, у нас есть модуль, который писал некий N. И все в нем хорошо, а вот один метод, допустим, call_me, должен всегда возвращать 1, иначе беда, а метод из базовой поставки модуля возвращает всегда 0. Код модуля трогать нельзя.

Пусть программа выглядит следующим образом:

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

my $obj = Top->new();
if ($obj->call_me()) {
    print "Purrrrfect\n";
}
else {
    print "OKAY :(\n";
}

package Top;
use strict;
sub new {
    my $class = shift;
    my $self = {};
    bless $self, $class;
    return $self;
}

sub call_me {
    print "call_me from TOP called!\n";
    return 0;
}

1;
```

Которая выведет:

```
call_me from TOP called!
OKAY :(
```

И снова у нас есть решение:

Допишем перед вызовом $obj->call_me() следующую вещь:

```
*Top::call_me = sub {
    print "Overrided subroutine called!\n";
    return 1;
};
```

И теперь наш вывод будет выглядеть примерно следующим образом:

```
Overrided subroutine called!
Purrrrfect
```

Код модуля не меняли, функция теперь делает то, что нам надо.

Программисту на заметку: Если приходится часто использовать данные прием в работе — налицо архитектурный косяк и вообще, я уже описывал эту ситуацию стишком. Хороший пример использование — добавление дебаг информации в функции.

Локальные переменные или динамическая область видимости:

Допустим, у нас есть скрипт, который что-то считает. Вдруг нам в функции, например, понадобилась локальная переменная, которая, пусть, должна быть копией глобальной.

Мы можем это сделать следующим образом:

```
use strict;
use warnings;
my $x = 1;
print "x: $x\n";
do_with_x();
print "x: $x\n";

sub do_with_x {
    my $y = $x;
    $y++;
    print "y: $y\n";
}
```

Вывод ожидаемый:

```
x: 1
y: 2
x: 1
```

Однако, в данном случае мы можем обойтись без y. Решение выглядит так:

```
use strict;
use warnings;
use vars '$x';
$x = 1;
print "x: $x\n";
do_with_x();
print "x: $x\n";

sub do_with_x {
    local $x = $x;
    $x++;
    print "x: $x\n";
}
```

Эта штука и называется динамической областью видимости. Очень хорошо этот пример помогает понять представление переменной в виде карточек `local` — это когда мы закрываем то, что написано на карточке другой карточкой, а как только выходим из блока, все возвращается на круги своя. Также очень часто эта конструкция используется внутри блоков кода в которых необходим автосброс буфера, тогда можно сделать:

```
local $| = 1;
```

Или, если лениво писать в конце каждого print `\n`:

```
local $\ = "\n";
```

Атрибуты функций

В Python есть такое понятие, как декоратор. Это такая штуковина, которая позволяет «добавить объекту дополнительное поведение».

Да, в Perl декораторов нет, зато есть атрибуты функций. Если мы откроем `perldoc perlsub` и посмотрим на описание функции, там

есть такая любопытная запись:

```
sub NAME(PROTO) : ATTRS BLOCK
```

Таким образом, функция с атрибутами может выглядеть так:

```
sub mySub($$;$) : MyAttr {
    print "Hello, I am sub with attributes and prototypes
       !";
}
```

Многие программисты, которые пишут на Perl ни разу не сталкивались ни с атрибутами, ни с прототипами. Причина этого кроется в литературе по языку для быстрого старта довольно низкого качества и сомнительного содержания. Но мы работаем над этим.

И скоро даже что-нибудь покажем. Все вышесказанное не касается книги с Верблюдом, Альпакой и Ламой, это прекрасная литература и тем кто ее не читал — я настоятельно рекомендую. Вернемся к нашим атрибутам.

Работа с атрибутами в Perl дело нетривиальное, потому уже довольно давно в стандартную поставку Perl входит модуль `Attribute::Handlers`.

Дело в том, что атрибуты из коробки имеют довольно много ограничений и нюансов работы, так что если кому интересно — мой почтовый адрес есть в статье и мне можно всегда написать.

Допустим, у нас есть функция, которая может быть вызвана только в том случае, если пользователь авторизован. За то, что пользователь авторизован отвечает переменная `$auth`, которая равна 1, если пользователь авторизован и 0, соответственно, нет. Мы можем сделать следующим образом:

```
my $auth = 1;

sub my_sub {
    if ($auth) {
        print "Okay!\n";
        return 1;
    }
    print "YOU SHALL NOT PASS!!!1111";
    return 0;
```

```
10 }
```

И это нормальное решение.

Но может возникнуть такая ситуация, что функций будет становиться больше и больше. А в каждой делать проверку будет становиться накладно. Проблему можно решить на атрибутах.

```
1  use strict;
2  use warnings;
3  use Attribute::Handlers;
4  use Data::Dumper;
5
6  my_sub();
7
8  sub new {
9      return bless {}, shift;
10 }
11
12 sub isAuth : ATTR(CODE) {
13     my ($package, $symbol, $referent, $attr, $data,
          $phase, $filename, $linenum) = @_;
14     no warnings 'redefine';
15     unless (is_auth()) {
16         *{$symbol} = sub {
17             require Carp;
18             Carp::croak "YOU SHALL NOT PASS\n";
19             goto &$referent;
20         };
21     }
22 }
23
24 sub my_sub : isAuth {
25     print "I am called only for auth users!\n";
26 }
27
28 sub is_auth {
29     return 0;
30 }
```

В данном примере вызов программы будет выглядеть так:

```
1  YOU SHALL NOT PASS at myattr.pl line 18. main::__ANON__()
       called at myattr.pl line 6
```

А если мы заменим return 0 на return 1 в `is_auth`, то:

```
I am called only for auth users!
```

Я не зря написал про атрибуты в конце статьи. Для того, чтобы написать этот пример мы воспользовались:

- Анонимными функциями.
- Оверрайдом функций.
- Специальной формой оператора goto.

Не смотря на довольно громоздкий синтаксис атрибуты успешно применяются в `Catalyst`, например.

К тому же, не стоит забывать, что они, все-таки, являются экспериментальной фичей Perl, а потому их синтаксис может меняться.

О чем хотелось бы еще написать и я напишу:

- Фазы исполнения Perl программы.
- Более пристальный взгляд на модули.
- AUTOLOAD и DESTROY.
- Особая форма оператора goto.
- Прагмы Perl.

До новых встреч.

■ *Дмитрий Шаматрин*

3. Сигнатура функции в Perl 5.20

Случилось то, что многие так долго ждали. 6 февраля 2014 г. реализация сигнатуры функции была добавлена в основную ветку разработки Perl и стала доступна для использования всем желающим. Эта экспериментальная возможность включена в релиз для разработчиков 5.19.9, и, если не будет выявлено серьёзных проблем, войдёт в грядущий стабильный релиз Perl 5.20.

Пурпурные сигнатуры

Попытки добавить сигнатуру функции в ядро Perl предпринимаются уже достаточно давно. *Peter Martini* демонстрировал сообществу наработки ещё до релиза 5.18, но тогда эта реализация была достаточно сырой и имела множество недостатков. Работа была продолжена и, например, в версии 5.19.5 появилась альтернативная возможность для указания прототипа функции через специальный атрибут функции `prototype`. Это был небольшой шаг к продвижению реализации сигнатур в ядре Perl.

В середине января 2014 г. по результатам дискуссии в списке рассылки p5p стало казаться, что сигнатуры функции опять не попадут в ядро и будут отложены на неопределённое время. Zefram выступил с разгромной критикой на реализацию Питера, доказывая, что предложенное изменение API крайне нежелательно. Но, как известно, разговоры ничего не стоят, поэтому Zefram продемонстрировал свой вариант реализации сигнатур, который вообще не меняет существующее API, лишь расширяя существующий синтаксис прототипов в случае если эта возможность явно включена.

Первоначально реализация получила название `simple_signatures` (простые сигнатуры), но поскольку обсуждение выявило неподдельный интерес к реализации сигнатур с возможностью для расширения синтаксиса, то новая реализация получила кодовое название `purple_signatures` (пурпурные сигнатуры): с одной стороны из-за того, что Zefram'у трудно было подобрать подходящее прилагательное, с другой стороны, возможно потому, что пурпурный

цвет часто относят к роскоши (цвет одежды римских императоров, церковных служителей и т.д.), таким образом закладывается база для построения совершенной и законченной системы сигнатур функции, которой так не хватало в Perl.

Ricardo Signes, как текущий ответственный за выпуск стабильных версий Perl, одобрил включение пурпурных сигнатур в основную ветку, тем самым закрепив реализацию сигнатур Zefram'а для будущих версий Perl. Ну а реализация Питера осталась не у дел: несмотря на его попытки переделать решение с исправлением всех высказанных замечаний, ему достаточно твёрдо сказали, что он может и дальше присылать свои патчи для Perl, но уже в рамках какой-либо другой темы.

Использование сигнатуры функции

Для того, чтобы поэкспериментировать с новыми сигнатурами, достаточно установить Perl 5.19.9 или *blead*:

```
$ perlbrew install blead
```

Для использования этой сборки Perl в текущей шел-сессии выполните:

```
$ perlbrew use blead
```

Функционал сигнатур является на данный момент экспериментальным и активируется путём подключения возможности `signatures`. Таким образом, каждая программа, использующая эту возможность должна начинаться со следующих строк:

```
use feature 'signatures';
no warnings 'experimental::signatures';
```

Простейший пример сигнатуры функции

```
sub sum ($x, $y) {
    return $x + $y
}
```

```
4
5 my $z = sum(2,2); # 4
```

Определяется, что функция `sum()` имеет два аргумента: переменные `$x` и `$y`. Функция не может быть вызвана с меньшим или большим числом аргументов — это приведёт к ошибке во время исполнения.

Это становится наглядно, если посмотреть на код под призмой `B::Deparse`:

```
1 sub sum {
2     use feature 'signatures';
3     die 'Too many arguments for subroutine' unless @_ <=
        2;
4     die 'Too few arguments for subroutine' unless @_ >=
        2;
5     my $x = $_[0];
6     my $y = $_[1];
7     ();
8     return $x + $y;
9 }
```

Как видно, переменные `$x`, `$y` являются **копиями** переданных в функцию аргументов. В то время как массив `@_` содержит алиасы на аргументы функции.

Необязательные аргументы

Существует возможность задавать необязательные аргументы, которым, в случае отсутствии явного значения при вызове, присваивается значение по умолчанию:

```
1 sub sum ( $x, $y = 1 ) {
2     return $x + $y
3 }
4
5 my $z = sum(1);  # $z = 2
```

Таким образом, если не указан второй аргумент, то он принимает значение по-умолчанию: 1.

Важно, что все аргументы, для которых указано значения по умолчанию, должны идти в конце списка. Например, такой код

приведёт к ошибке компиляции:

```
# Ошибка! Обязательный аргумент ёидт после
    необязательного
sub sum ( $x = 0, $y ) {
     return $x + $y
}
```

Кроме того, значение undef имеет приоритет над значением по умолчанию:

```
sub sum ( $x, $y = 10) {
    return $x + $y
}

my $z = sum(5, undef);  # $z = 5, а не 15
```

Для задания значения необязательных аргументов можно использовать произвольные выражения, включая переменные, определённые в сигнатуре:

```
sub sum ($x, $y = $x + 5) {
    return $x + $y

my $z = sum(10); # $z = 10 + (10 + 5) = 25
```

Следует лишь не запутаться в порядке определения переменных:

```
use strict;

# Ошибка! $z ёещ не определена при первом использовании
sub sum ($x, $y = $x+$z+1, $z = 10) {
     return $x + $y + $z
}
```

Как видно в примере, при задании значения по умолчанию для переменной y, z ещё не определена.

Переменное число аргументов

Если у функции может быть один или несколько необязательных аргументов, их можно присвоить массиву в конце списка сигнатуры:

```
sub sum ($x, $y, @other) {
```

```
    my $z = 0;
    $z += $_ for @other;
    return $x + $y + $z
}

my $value = sum(1,2,3,4,5,6,7)   # $value = 28
```

Также возможно использование хеша:

```
sub method ($self, %opts) {
    ...
}
```

Обязательно, чтобы такой аргумент был последним в списке.

Игнорирование некоторых аргументов

Иногда может потребоваться пропустить часть аргументов (неважных для данной функции) и не выделять под их хранение локальную переменную. Для этого можно использовать обычный сигил без имени переменной в сигнатуре:

```
sub sum_1_4( $x, $, $, $y, @) {
    return $x + $y
}

sum_1_4( 1, "not important", "not important", 2, "bla bla
    ", "foo bar"); # 3
```

В данном примере функцию интересуют только первый и четвёртый аргументы. Для остальных аргументов указаны заполнители позиций: сигилы `$` и `@`, они не будут определяться внутри функции.

Прототип функции

Если используются сигнатуры, то для описания прототипа функции должен использоваться атрибут `prototype`. Например:

```
sub foo :prototype($$) ($x, $y) {
    ...
```

```
}
```

Прототип должен быть указан **до** сигнатуры, т.к. после сигнатуры функции может идти только тело функции.

Важно помнить, что проверка прототипов работает на этапе компиляции, в то время как проверка сигнатуры происходит во время исполнения программы. По этой причине прототипы не работают для методов класса, но при этом проверка сигнатуры работает.

Зачем могут потребоваться прототипы, если есть сигнатуры функции? Рассмотрим простой пример, где эти отличия проявляются:

```
sub sum_np ($x, $y = 40) {
    return $x + $y
}

sub sum_wp :prototype($;$) ($x, $y = 40) {
    return $x + $y
}

my @arr = (10, 14);
say sum_np(@arr);  # 24
say sum_wp(@arr);  # 42
```

Две одинаковые функции отличаются прототипом. В случае без прототипа мы передаём значения `$x` и `$y` в массиве `@arr` — это числа `10` и `14`, их сумма равна `24`. В случае с прототипом мы ожидаем первым параметром скаляр, таким образом массив `@arr` интерпретируется в скалярном контексте и возвращает `2` — число элементов массива. Второе значение не передаётся, поэтому переменная `$y` получает значение по умолчанию `40` и итоговая сумма становится `42`.

Переменная `@_`

Большую дискуссию вызвал вопрос, что делать с массивом `@_`, содержащим переданные в функцию параметры. Одной из предпосылок создания сигнатуры функции была идея оптимизации,

чтобы не инициализировать переменную @_. Однако текущая реализация никак не затрагивает переменную @_, более того выносит отказ от @_ за рамки реализации сигнатуры функции.

Высказывалось несколько идей оптимизации, например, если функция не использует переменную @_, то и не надо инициализировать её. Но, к сожалению, парсер не может выявить использование @_ в подпрограммах на этапе компиляции, т.к. некоторые части программы могут определяться только на этапе исполнения. Например, при выполнении eval кода, содержащегося в строке:

```
sub foo {
    eval 'say for @_';
}
```

Dave Mitchell также привёл пример с выполнением кода в регулярных выражениях:

```
$pat = qr/(?{ print "[$_]\n" for @_ })/;
sub f($str, $pat) { $str =~ $pat }
```

Поэтому пока переменная @_ по-прежнему доступна в функциях с сигнатурой.

Заключение

Сигнатура функции — одна из самых ожидаемых возможностей языка Perl 5. Приживётся ли существующая реализация и насколько эффективно будет её использование покажет время. На данный момент одним из самых заметных недостатков реализации — это дополнительная проверка количества элементов функции во время выполнения, что добавляет оверхед в уже и без того не слишком быструю реализацию вызова функций.

■ *Владимир Леттиев*

4. Обзор CPAN за февраль 2014 г.

Рубрика с обзором интересных новинок CPAN за прошедший месяц.

В этом месяце замечена активность по исправлению некоторых базовых модулей для корректной работы на платформе **Android** (`IO::Compress`, `DateTime`, `Module::Build` и некоторые другие). Новый релиз Perl 5.19.9 для разработчиков включил поддержку нативных сигнатур функций. Известный модуль `CGI` ищет нового мейнтейнера.

Статистика

- Новых дистрибутивов — 200
- Новых выпусков — 855

Новые модули

- warnings::MaybeFatal

Модуль `warnings::MaybeFatal` позволяет сделать предупреждения фатальными только на этапе компиляции. Это может быть полезно, если требуется поймать возможные ошибки при запуске, но не прерывать работу программы, когда она уже запущена.

```
1     use strict;
2     use warnings qw(all);
3     use warnings::MaybeFatal;
4
5     # Неинициализированное значение при конкатенации
         строк
6     # Предупреждение во время исполнения, поэтому
         нефатальное
7
8     print undef . "string";
9
```

```
10      # Бесполезное использование константы в пустом
           контексте
11      # Предупреждение во время компиляции, поэтому
           фатальное
12
13      42;
```

- Data::DynamicValidator

В полку валидаторов структур прибавление — модуль `Data::DynamicValidator` позволяет выполнять сложные проверки структур данных. Благодаря использованию динамических правил, возможен обход всех листьев ветки структуры или только части, удовлетворяющих определённым условиям.

- Text::Nimble

Появился модуль для обработки и конвертирования языка разметки Nimble в html. Структурированный язык разметки Nimble в чём-то схож с Wiki и Markdown, но имеет ряд интересных возможностей. Например, макросы для создания сложных повторяющихся фрагментов, и метаданные документа, доступные как хеш с набором атрибутов на выходе парсера.

- Exobrain

По аналогии с экзоскелетом, `Exobrain`, вероятно, означает внешний мозг. Данный проект является свободным аналогом сервиса IFTTT, позволяя автоматически реагировать на различные события (например, слать твиты друзьям, которые живут в городе, в который вы сейчас прибыли, используя данные Foursquare и Twitter).

- Localizer

Модуль `Localizer` — это новый фреймворк для создания локализации, отличающийся простотой интерфейса и реализации.

- Raisin

`Raisin` — это микро веб-фреймворк, ориентированный на создание RESTful API сервисов. По синтаксису DSL он напоминает фреймворк Grape из мира Ruby.

Обновлённые модули

- XML::LibXML 2.0110

Небольшое косметическое изменение появилось в дистрибутиве `XML-LibXML` — все модули в его составе обзавелись прагмами `use strict` + `use warnings`. Лучше поздно, чем никогда.

- Pod::Markdown 2.000

Вышел новый мажорный релиз модуля `Pod::Markdown` для конвертирования документов формата POD в Markdown. В новой версии произошёл переход на `Pod::Simple` вместо `Pod::Parser`. Появились некоторые несовместимые изменения, в частности, встречаемые теги `E<>` теперь просто конвертируются в соответствующие UTF-8 символы, символы табуляции заменяются пробелами.

- Pod::Parser 1.62

Документация новой версии включает упоминание о том, что модуль является устаревшим и рекомендует использовать `Pod::Simple`.

- Pod::Perldoc 3.23

В новом perldoc был восстановлен флаг `–U`, который позволяет не сбрасывать привилегии программы в случае запуска от `root`. Опасно ли не сбрасывать привилегии программе, которая только читает код, вопрос отдельный, но иметь такую возможность иногда требуется.

- Mail::SpamAssassin 3.4.0

Почти три года прошло с последнего стабильного релиза антиспам системы SpamAssassin. В новой версии появилась нативная поддержка IPv6, улучшена технология блокировки по DNS-именам и добавлена поддержка работы Байесовых фильтров с использованием бэкенда на Redis.

- Elasticsearch 1.0.4

Обновился официальный клиент системы поиска и анализа `Elasticsearch`. Новая мажорная версия поддерживает новое API сервиса версии 1.0 и использует его по умолчанию. Для использования старой версии API 0.90.x необходимо явно указывать это в параметрах подключения.

- Catalyst 5.90060

Новый стабильный релиз веб-фреймворка Catalyst содержит ряд заметных изменений. Прежде всего это изменение в способе передаче данных ответа Plack-серверу. Раньше для этого всегда использовался потоковый интерфейс, теперь же, если тело ответа это простой скаляр, объект или файловый дескриптор, то он передаётся в Plack-сервер как есть. В некоторых случаях это может повысить скорость отдачи данных, корректно заработает chunked encoding, если сервер её поддерживает. Начата миграция части

кода Catalyst на использование аналогичного по функционалу Plack Middleware.

- Module::Runtime 0.014

В новом релизе `Module::Runtime` исправлено несколько ошибок. Одно из исправлений меняет логику работы `use_module()`, передавая значение version для проверки версии модуля, даже если оно имеет значение `undef`. Некоторые существующие системы могут быть завязаны на старом поведении, что неизбежно приведёт к их поломке при обновлении данного модуля. В частности, так оказался сломан `Moose`.

- Capture::Tiny 0.24

Данный релиз исправляет ошибку безопасности при работе с временными файлами. При открытии временного файла на запись существовала возможность переписать существующий файл пользователя при использовании атаки *symlink race*.

- Moose 2.1204

`Moose` был исправлен в связи с изменением поведения нового `Module::Runtime`.

- mop 0.03

Очередной экспериментальный релиз реализации метаобъектного протокола для базового Perl. В новом релизе присутствует множество внутренних улучшений. Также включены конфигурационные файлы для корректной подсветки синтаксиса в редакторах vim, emacs и textmate.

- perlsecret 1.008

Новый релиз документации по секретным операторам в Perl содержит два новых оператора: *Эббот и Костелло* и *Наклонившийся Эббот и Костелло*.

■ *Владимир Леттиев*

5. Интервью с Брено де Оливейра (Breno G. de Oliveira)

Брено де Оливейра (garu) — бразильский Perl-программист, автор многих модулей на CPAN, активный участник и организатор Perl-мероприятий, обладатель награды White Camel.

Когда и как начал программировать?

В 1990 году, когда мне было 8 лет, у меня появился отчим, который в качестве хобби программировал на BASIC. Я быстро этим заинтересовался и начал приставать к нему с распросами о том, как сделать простые и незамысловатые штуки типа напечатать имя пользователя сто раз или как издавать звуки динамиком. Через некоторое время мы с братом перешли на продвинутые вещи типа ascii-анимации, мелодии через системный динамик, и крутую (но я был всего лишь ребенком!) систему “шифрования”, которая использовала динамическую смену шифра, но достаточную для защиты наших сверхсекретных данных. И нам очень нравилось редактировать DOS-игры через hex-редактор, меняя там тексты, но это было больше хакерством и желанием разобраться в том, как работают программы, чем собственно программированием.

Какой редактор используешь?

VIM НАВСЕГДА!! :D

Мне очень нравится писать в vim, несмотря на то, что у нас с ним было сложное знакомство в раннее время использования Linux. Сейчас мне всегда не по себе, когда передо мной открыт другой редактор, и я часто нажимаю ESC (или `:w`) после того как что-то напечатал, иногда даже в браузере.

Я также много работал над Padre, IDE для Perl, и думаю, что это хороший редактор, особенно (но не только) для новичков. Несмотря на это, я не участвую в холиварах, и когда кто-то спрашивает у меня совета, я рекомендую попробовать несколько редакторов и выбрать наиболее комфортный.

```
:w
```

Когда и как познакомился с Perl?

В начале 2000-х в университете я загорелся информационной безопасностью. В то время время Perl был большим игроком в этой сфере, и я решил его выучить, так как хотел применить свои навыки программирования в будущей профессии. Но я все откладывал этот момент, пока на одной локальной конференции свободного ПО не выиграл книгу Perl Cookbook во время викторины после доклада по информационной безопасности. Я начал просматривать книгу, цепляясь за всякие интересные идеи, которые я бы мог реализовать, и был поражен, насколько просто можно было писать программы, просто подключая уже готовые модули, делая не только то, что я хотел, но и делая это правильно. К тому времени я профессионально работал с C, и синтаксис Perl, его портируемость и интуитивность позволили мне программировать без забот о выделении памяти, указателях, а больше фокусироваться на решении самой проблемы. Это невероятно освобождало, и я никогда не хотел вернуться.

С какими другими языками интересно работать?

У меня всегда будет слабость к C в моем сердце, несмотря на то, что я уже давно ничего не писал на нем. С недавнего времени я также приобщился с JavaScript-сообществу, также немного игрался со Scala и Lisp — я даже написал статью “Scala для Perl-программистов”, показывая сходства и различия между двумя языками. Ruby также неплох, хотя я особо не пишу на нем, никаких серьезных проектов.

Я также основал локальную группу по программированию, где мы пробуем разные языки и парадигмы, это тоже весело. О! И также время от времени я загружаю последнюю версию Rakudo и пробую немного Perl 6, в основном для развлечения и чтобы посмотреть, достаточно ли он быстрый для прототипирования некритичного кода. С появлением rakudobrew это стало довольно просто.

Что, по-твоему, является самым большим преимуществом Perl?

Должен сказать, что это сообщество. Но сообщество не только в виде людей, но и в виде культуры этого сообщества. Когда я начал

участвовать в списках рассылки и мероприятиях, я быстро понял, что за редким исключением большинство Perl-программистов умные, вежливые, знающие свое дело и умеющие аргументировать свою точку зрения без FUD, предрассудков и фанатизма. Эта культура делает стоящим само участие в сообществе. Также она поощряет обмен информацией, так как мы стараемся помогать друг другу, будь то ответы новичкам, оформление отчета об ошибке или написание полезного модуля и выкладывание его совершенно бесплатно.

Благодаря такому сообществу в 90% случаев на CPAN можно найти уже готовое решение, протестированное на нескольких платформах, оцененное и осмотренное лучшими программистами, и все это бесплатно. Благодаря такому сообществу мы практичестки мгновенно можем получить помощь, когда сталкиваемся с какой-то проблемой. Благодаря такому сообществу мы можем на полную использовать Perl 5, видеть как он развивается каждый день, как улучшается качество кода, появляется новый фунционал, эволюционируют лучшие практики и много другое.

Какими, по-твоему, свойствами должны обладать языки будущего?

Должен признаться, что об этом серьезно никогда не задумывался, но скорее тут есть два больших направления.

Со стороны железа, процессоры последнее время не сильно увеличивают свою производительность, и кажется, что многоядерный подход прочно закрепился. Но тем не менее, большинство языков, Perl 5 в том числе, работают либо на одном ядре, либо имеют очень ограниченную, сложную и полностью не интегрированную поддержку сопрограмм и многоядерности. Для меня, языки будущего должны поддерживать сопрограммы и многоядерность элегантным и интуитивным способом, в то же время простым и прогрочным, чтобы приложения могли полностью реализовывать этот потенциал при наличии такой возможности. Я слышал, что Perl 6 двигается в этом направлении, что очень интересно, но, как я уже сказал, у меня не было возможности с этим разобраться поближе.

Со стороны окружения, я думаю, что преимущество над другими

у языков, которые могут исполняться в браузере. Конечно, я сейчас говорю о JavaScript, однако я все думаю, когда же какой-нибудь другой язык получит нативную поддержку в браузерах. С другой стороны я нахожу забавным, что в то время как многие языки из серверной части пытаются перебраться на клиентскую, JavaScript-разработчики делают обратное и пытаются перестать быть просто языком веб-браузера с помощью таких проектов как node. Разработка под браузер означает, что ваше приложение может быть запущено на любом устройстве, где есть браузер, и быть там, где люди проводят и так большую часть своего времени (если только вы не пишете чего-нибудь специфического или не для конечных пользователей).

В заключение, также про окружения, JVM давно уже стало де-факто стандартом для виртуальных машин, улучшив скорость исполнения, портируемость и требовательность к ресурсам в течение последних лет. Большим корпорациям гораздо проще внедрять новые продукты, уже имея JVM-наработки в каждом ИТ-департаменте, учитывая также то, что разработка под одно окружение сильно упрощает жизнь, перекладывая вопросы портируемости на виртуальную машину, а не флаги компиляции, позволяя языку быть проще в поддержке и эволюционировать. Мне кажется, что языки будущего должны стремиться иметь в своей поставке хотя бы одну имплементацию, работающую поверх JVM.

Насколько Data::Printer отличается от Data::Dumper?

Во многом! Data::Dumper решает проблему сериализации структур данных в Perl таким образом, чтобы можно было использовать их в коде, т.е. переменные выводятся, но очень читабельно. Data::Printer (или DDP), с другой стороны, не сильно заботится о возможности использования вывода в коде и разрабатывался, ориентируясь на читабельность. Он предоставляет цветной вывод, отсортированные ключи, фильтры и много другое! Например, при вызове Data::Dumper на объекте вы делаете что-то вроде:

```
1 use Data::Dumper; warn Dumper($obj);
```

и печатается внутренняя структура bless-переменной и имя класса, что часто не совсем то, что нужно. Если использовать Data::Printer, например:

```
use DDP; p $obj;
```

что не только короче, но также напечатает, что это объект такого-то класса, список его родителей (`@ISA`), публичные/приватные/унаследованные методы, *и* его внутреннюю структуру, и все это в цвете и удобно отформатированное. Также сообщается и много дополнительной информации, которой нет в Data::Dumper, например является ли переменная опасной (taint), доступной только для чтения (readonly) или слабой ссылкой (weak reference), что очень помогает при отладке. Data::Printer может показать ссылки, которые не поддерживаются (пока что) Data::Dumper, например lvalue, а также гибко настраивается в зависимости от требований просмотра или отладки.

Так что если вы хотите видеть и инспектировать содержимое переменных, вам стоит использовать Data::Printer. Вы не вернетесь к Data::Dumper, я гарантирую :)

Когда будет закончен App::Rad?

Эй, смотри! Трехголовая обезьяна!

Ох, это очень хороший вопрос. Я написал App::Rad, потому что одно время я писал множество консольных утилит и мне не очень нравилось каждый раз писать одно и тоже, когда я приступал к новому проекту. Уже был модуль App::Cmd авторства rjbs, но тогда он мне казался (да и иногда сейчас кажется) перегибом для моих задач. Это как, например, если бы App::Cmd был как Catalyst, а мне нужно было что-то вроде Mojolicious или Dancer. Оказывается, что многие разработчики считали так же, и у моего модуля быстро появились пользователи, люди начали просить новых фишек и исправления багов. Я даже слышал, что он используется в CERN в нескольких внутренних приложениях, которые интегрируют данные из разных лабораторий для LHC, что очень круто.

Когда я написал App::Rad, я сделал это для себя, для решения определенных задач, поэтому сожалению о многих внутренних архитектурных решениях, которые не позволяют ему быть расширяемым настолько, насколько хотелось бы. По мере того как проект становился больше и мощнее, новый функционал не был обратно

совместим, поэтому я встал перед дилеммой начать писать что-то новое или выпустить версию, которая бы ломала написанный код. Поэтому я решил все переписать с нуля, стараясь больше внимания уделять расшиямости и простоте поддержки. К сожалению, это заняло времени больше, чем у меня было, и к тому времени я уже перестал работать над консольными приложениями и так и не вернулся к этому проекту :(

Я все еще планирую выпустить его в будущем, но к сожалению у меня сейчас другие интересы.

Что такое QA-хакатоны? Кого приглашают?

Как должно быть уже знают читателя журнала, Perl-сообщество уделяет много внимания контролю качества (или сокращенно QA от quality assurance). Мы приветствовали тестирование задолго до того, как это стало модным. У нас кучи модулей для тестирования, утилит, рекомендаций. Инфраструктура CPAN позволяет быть уверенным, что добавленный модуль работает на каждой версии и платформе, на которой поддерживается Perl. Чтобы добиться этого, мы должно быть уверены, что все инструменты могут правильно взаимодействовать между собой. Возьмем, к примеру, сервис CPAN Testers. Он должен взаимодействовать с CPAN-клиентами по всему миру, которые отправляют результаты тестирования конкретного модуля на конкретной системе, и в то же время предоставлять данные для обработки и отображения на MetaCPAN. Это много систем, которые должы быть идеально синхронизированы, и это только *один* пример!

Несмотря на то, что у нас получается вести онлайн-дискуссии, принимать архитекрурные решения и работать над всеми этими инструментами в течение года в наше свободное от работы, семьи, друзей или других проектов время (помните, что это волонтерская работа!), это никогда не будет настолько же продуктивным, как если бы мы сидели в одной комнате — и это именно то, что мы делаем.

Раз в год все, кто вовлечен в инфраструктуру контроля качества, собираются вместе на выходные (обычно с пятницы по воскресенье) где-нибудь и работают исключительно над такими проектами.

Среди нас есть разработчики ядра Perl, которые чинят долго откладываемые баги, авторы модулей для тестирования, обсуждающие решения и выпускающие новые версии и инструменты, любители поработать с данными и их визуализацией для разработчика, администраторы, работающие над тем, чтобы все пересылалось как можно быстрее и надежнее и так далее. Мне повезло поучаствовать в двух последних QA-хакатонах, в Париже и Ланкастере соотвественно, и могу сказать, что это поразительно. Каждый сфокусирован и продуктивен, и мы много успеваем все лишь за один уикенд. Кроме того, никому не платят за это, мы делаем это по своему желанию для развития яызыка и его сообщества. Поэтому очень важно, что мы получаем пожертвования от сообщества и спонсирующих компаний на еду и проживание во время нашей работы.

Если говорить об участниках, как я уже сказал, приглашен любой, кто достаточно заботится о тестировании в экосистеме Perl, чтобы приехать в другой город за свои деньги и провести несколько дней, улучшая эту экосистему, вплоть до максимального количества людей для помещения — обычно это около сорока человек. Большинство участников работают над проектами и вне рамках хакатона, но это не является правлом. Любой, кто хочет помочь, заходите на канал #qa сервера irc.perl.org и сообщите о себе. В ином случае, если вы используете любой из инструментов или модулей для тестирования (а вы используте, поверьте мне!), пожалуйста, рассмотрите возможность пожертвования. Мы бы не делали и одной десятой доли тех вещей для Perl-экосистемы без QA-хакатонов, и это возможно только благодаря вашим пожертвованиям. Даже если этого всего лишь $1. Что? Вы уже пожертвовали в этом году? СПАСИБО!

За что получил награду White Camel?

Как за что, за красивое лицо, конечно!

А если серьезно, то за последние десять лет я делал много для Perl-сообщества, в особенности, организовывал YAPC::Brasil и другие Perl-мероприятия южнее экватора. Также я стараюсь содействовать обмену информацией между бразильским и мировым Perl-сообществами, путешествуя по разным странам и знакомя людей со всеми клевыми штуками, которыми занимаются бразильские Perl-

разработчики, переводя документацию и посты с одного языка на другой и тому подобное. Очень мало бразильцев, даже среди разработчиков, хорошо владеют анлглийским. Я стараюсь понизить языковой барьер и выступаю посредником между сообществами, которым есть что почерпнуть друг у друга. Поэтому в 2012 году во время YAPC::NA я бы удостоен награды White Camel. Это было неожиданно, и я был первым человеком из Латинской Америки, получившим ее. Я был очень рад, и сейчас награда гордо стоит в моей гостинной :)

Где сейчас работаешь? Сколько времени проводишь за написанием Perl-кода?

Я работаю техническим директором в Estante Virtual, электронном аукционе подержаных книг, полностью написанном на Perl. У нас довольно успешный бизнес здесь в Бразилии, мы продаем одну книги каждые три секунды. Мне нравится здесь работать, и несмотря на то, что у меня не так много возможностей писать код на работе, ко мне все еще обращаются за советом, когда кто-то из моей команды сталкивается с проблемой. Также я занимаюсь тренингами по новому функционалу, отладке, шаблонам проектирования и лучшим практиками, особенно для младших специалистов, которые приходят в нашу компанию.

Дома, когда у меня есть время и работа позволяет, я занимаюсь своими открыми проектами, большинство на Perl.

Стоит ли советовать молодым программистам учить Perl?

Вне всяких сомнений! Прямо сейчас! Идем за моей машиной? :D

Здоровое сообщество постоянно обновляется, постоянно эволюционирует и постоянно нуждается в новой силе. Каждому языку нужны посвященные молодые люди, которые ставят под сомнение старые подходы и вносят что-то новое, и Perl здесь не исключение. Я действительно думаю, что для некоторых людей это прекрасный язык, с его гибкостью, портируемостью, практичностью, мульти парадигменностью, общим назначением. Он позволяет новичкам писать код в том стиле, в котором они хотят, а не в том, в каком их кто-то принуждает писать. Также у него есть продвинутое и

дружелюбное по отношению к новичкам сообщество, которое это понимает и поощряет людей учиться, без обычного эгоистичного “рок-зведного” отношения, так распространенного в других сообществах динамических языков.

Вопросы от читателей

Что ты делал на YAPC::Europe 2013 в Киеве? Тебе понравилось?

Мне очень понравилось! Серьезно! Как я уже говорил, я люблю путешествовать и общаться с людьми из разных Perl-сообществ, делиться информацией и знакомиться с локальными проектами и разработчиками. Это была моя первая YAPC::Europe, и я отлично провел время. Даже круче, я остановился у одного из организаторов, и в конце концов начал помогать им на конференции, и у меня даже был специальный бэйдж! :)

Все было великолепно, и я чувствовал себя как дома. Я встретил много друзей из Европы и приобрел новых из Украины, России, Норвегии, Амстердама и Румынии.

Ты поддерживаешь несколько модулей на CPAN от других авторов. Тебя заставили?

Только модули авторства “vti” ;)

Вообще нет. Одной из особенностей Perl-сообщества является то, что как только ты становишься известным как надежный и компенентый человек, который достаточно заботиться о модулях, авторы обычно передают свои модули. Это происходит естественно: ты начинаешь отправлять отчеты об ошибках и патчи, начинаешь писать в список рассылки и irc-каналы, и в конце концов, если автор слишком занят или хочет поработать над чем-то другим, он дает тебе ключи от квартиры и просит поливать цветы.

Возьмем, например, Clone.pm. У меня были проблема с этим модулем, я отправлял патчи, но их никогда не применяли. Через несколько недель, когда я все еще не получил ответа от автора, я попытался связаться с ним через email, спросил что происходит, могу ли помочь с модулем, если автор слишком занят. Автор

быстро мне ответил и дал права на сопровождение, я применил свои патчи (и много других) и загрузил новую версию на CPAN.

Это правда, что ты возишь с собой два теплых пальто, независимо от того, куда едешь и какая там погода?

Конечно! Ведь никогда не знаешь когда вернется ледниковый период :)

Вообще-то, это очень смешная история. Я из очень жаркой страны, и здесь в Рио температура обычно между 20C и 45C в течение всего года, поэтому я не привык к холодной погоде (серьезно, я носил теплую одежду в Орландо, США) и обычно боюсь низких температур. Также я всегда слышал истории, что на Украине очень холодно.

Поэтому, когда я планировал приехать в Киев в 2013 г. на YAPC::Europe, я естественно спросил моего местного друга Вячеслава (Тихановского — *прим. ред.*), какая будет погода, и он сказал, что будет снег и чтобы я взял два теплых пальто с собой. Вместо того, чтобы открыть браузер и проверить чертову погоду, я поверил своему инсайдеру и упаковал два тяжелых пальто в сумку. Было лето в Киеве, 30C, и он, наверное, до сих пор смеется надо мной. Но ничего, я до него еще доберусь :P

Любишь лук?

Ха-ха-ха, вообще-то нет :P

Я назвал свой блог “the onion stand” (onion — лук, — *прим. ред.*), потому что у Perl логотип в виде лука, и из-за (не очень хорошей) игры слов, “stand” означает и стэнд, место где происходят презентации, и “making a stand”, т.е. крепко стоять на своем.

Но, по-правде, я ненавижу лук и капусту.

Здорово получилось!

■ *Вячеслав Тихановский*