Шаблоны D: учебное пособие

Автор: Philippe Sigaud

Перевод: Striver

Оглавление

Введение

Шаблоны являются центральной особенностью D, дающей вам мощные возможности генерации кода времени компиляции, которые сделают ваш код более чистым, более гибким, и даже более эффективным. Они везде используются в Phobos — стандартной библиотеке D — и, следовательно, любой пользователь D должен о них знать. Но, основанные на шаблонах C++, как и они, шаблоны D поначалу могут немного пугать. Документация веб-сайта D Programming Language является хорошим началом, хотя её описание шаблонов разбросано среди многих различных файлов, и (так как это справочник по языку) её материал не столько учит вас использованию шаблонов, сколько показывает вам их синтаксис и семантику.

Этот документ стремится быть своего рода учебником по шаблонам D, чтобы показать начинающему D-программисту, чего можно достичь с их помощью. Когда я использовал C++, я не помню, чтобы когда-нибудь использовал шаблоны, кроме как в контейнерах, и считал, что я никогда не смогу понять код метапрограмм уровня Boost[1], не говоря уже о его создании. Ну, нормальный синтаксис D для шаблонов и замечательные возможности, такие как, например, `static if`, `alias` или кортежи, вылечили меня от этого впечатления. Я надеюсь, что этот документ поможет вам тоже.

Что находится в этом документе

Первая часть имеет дело с самыми основами: как объявлять и создавать экземпляр шаблона, стандартные «строительные блоки», которые вы будете использовать почти во всех ваших шаблонах, вместе с шаблонами функций, структур и классов. Во всем тексте примеры будут представлять применения этих понятий.

Вторая часть о более продвинутых темах, которые пользователи шаблонов D, вероятно, будут использовать, но не ежедневно, такие как ограничения шаблона, mixin-шаблоны или перегрузка операторов.

Третья часть представляет другие инструменты метапрограммирования: строки mixin, вычисление функций времени компиляции (compile-time function evaluation), и __traits. Они рассмотрены с точки зрения шаблонов: как они могут взаимодействовать с шаблонами и что вы можете построить из них в сочетании с шаблонами.

Четвертая часть представляет более развитые примеры того, что можно сделать с помощью шаблонов, на основе реальных потребностей, которые у меня были в определенное время, и которые можно реализовать с помощью шаблонов.

Наконец, приложение о вездесущем выражении `is`, и о других ресурсах для дальнейшего чтения, дополняющих этот документ.

Соглашения

Для того, чтобы сделать этот документ более легкочитаемым, я использую стандартные соглашения книг по программированию, выделяя части текста. Главным образом, так:

1 Библиотека C++ Boost широко использует шаблоны.

Ключевые слова D будут выделены вот так: `int`, **`static if`**, `__traits` (они будут окрашены или нет, в зависимости от того, что было использовано для создания документа).

Идентификаторы и имена, используемые в примерах кода и упоминаемые в тексте, будут написаны примерно так: `myFunc`, `flatten`.

Внутренние ссылки будут выглядеть так.

Внешние ссылки будут выглядеть так.

Примеры кода с подсвеченным синтаксисом показаны так:

```
/**
 * Это многострочный комментарий.
 */
module intro;

import std.stdio;

void main()
{
    int times = 10;
    // Это комментарий
    foreach(i; 0..times)
        writeln("Hello, Word!");
}
```

Нумерация строк используется только при необходимости.

Я иногда отклоняюсь для небольших пояснений, обсуждая небольшой кусочек информации, слишком маленький, чтобы создавать для него отдельный раздел, но тем не менее интересный для читателя. Они обозначаются так:

Semi-Literate Programming. Большинство образцов кода, представленные в этом документе, компилируются с достаточно нестарым D-компилятором. В каталоге `utils` есть небольшой D-скрипт с именем `codesamples.d`, который извлекает образцы кода из файлов mardkdown.

Примеры с объявлением модуля **`module`** `name;` будет извлечено объявление, будет создан файл с именем `name.d` и откомпилирован (возможно с заглушкой `main()`, если ничего нет). Предыдущий пример кода создаёт файл с именем `intro.d` и так далее. Результаты компиляции помещаются в файл с именем `result.txt`. Примеры, которые зависят от других примеров, просто импортируют их (да, для модульности D нет необходимости в специфической разметке, чтобы сплести совместный код). Примеры с именем, заканчивающимся на `_error`, ожидаемо не компилируются: они находятся там, чтобы показать ошибки, заблуждения и подводные камни. Анонимные примеры *не* извлекаются: я использую их, чтобы показать небольшие фрагменты, не предназначенные быть самостоятельными, или просто, чтобы показать D-псевдокод.

В общем, вы можете рассматривать весь этот документ как гигантский D-пакет, описывающий сотни небольших модулей.

Наконец, некоторые разделы в этом документе пока не завершены. Разделы, которые я

считаю незаконченными будут содержать это:

> **Незакончено.** Эй, теперь я добавил раздел Благодарности. Но пока я добавляю новые части в документ, новые добавления, и новый раздел, это вступление не будет закончено.

Наверняка я забыл какие-либо метки 'незакончено', не стесняйтесь сообщать мне об этом.

Как получить этот документ

Этот документ — просто markdown-файл, расположенный на Github. Не стесняйтесь ответвлять его (to fork), или (даже лучше для меня) вносить изменения (pull requests)! Для тех из вас, кто читает это на бумаге, вот адрес:

https://github.com/PhilippeSigaud/D-templates-tutorial

Благодарности

Как только я публично выпустил этот документ, участники сообщества D предложили мне помощь, предложения, корректировки и примеры кода. Это здорово, видеть, как появляются D-сообщества, и как люди участвуют в совместных проектах. Следующие люди помогли мне:

Craig Dillabaugh, Andrej Mitrovic, Justin Whear, Zachary Lund, Jacob Carlborg, Timon Gehr, Simen Kjaeras, Andrei Alexandrescu, Bjorn Lietz-Spendig.

Спасибо, парни!

Основы

Почему Шаблоны?

Вы здесь, читаете документ размером с книгу о шаблонах D. Но почему они вообще должны вас интересовать? Скажем, у вас есть эта замечательная структура, реализующая дерево:

```
module basicTree1;

struct Tree {
    int value;
    Tree[] children;

    size_t size() {
        size_t s = 1;
        foreach(child; children)
            s += child.size();
        return s;
    }

    bool isLeaf() @property {
        return children.length == 0;
    }
}
```

Которая используется как-то так:

```
module usingBasicTree1;

import basicTree1;

void main()
{
    auto tree = Tree(0, [Tree(1), Tree(2, [Tree(3), Tree(4), Tree(5)])]);
    assert(!tree.isLeaf);
    assert(tree.size() == 6);
}
```

Всё хорошо и замечательно, это отличное базовое n-нарное дерево, содержащее `int`'ы. Но что, если через некоторое время вам понадобится такое же для хранения `float`'ов? Без проблем, это можно легко закодировать через копирование-вставку. Сначала мы изменяем `Tree` на `IntTree`, затем создаём узел `FloatTree`:

```
module basicTree2;

struct IntTree {
    int value;
    IntTree[] children;

    size_t size() {
        size_t s = 1;
        foreach(child; children)
            s += child.size();
        return s;
    }

    bool isLeaf() @property {
        return children.length == 0;
    }
```

```
}

struct FloatTree {
    float value;
    FloatTree[] children;

    size_t size() {
        size_t s = 1;
        foreach(child; children)
            s += child.size();
        return s;
    }

    bool isLeaf() @property {
        return children.length == 0;
    }
}
```

Но здесь много дублирования кода: единственные изменения — типы значений `value` и потомков `children`, которые стали `float` и `FloatTree` вместо `int` и `IntTree`. Какое расточительство! А что, если нам понадобится другое дерево, например для хранения функций (дерево обратных вызовов, например)? Должен быть лучший способ.

Давайте изучим предыдущий код. Что нам здесь нужно — это способ производить код, генерирующий различные типы дерева, вставляя тип для `value` как задаваемый пользователем ввод. Это немного похоже на функцию: ввод параметров и получение результата. Давайте представим себе некоторый код с заполнителем, давайте назовём его `Type` (Тип), чтобы представлять тип значения `value`.

```
struct Tree {
    Type value;
    Tree[] children;

    size_t size() {
        size_t s = 1;
        foreach(child; children)
            s += child.size();
        return s;
    }

    bool isLeaf() @property {
        return children.length == 0;
    }
}
```

Но здесь `Type` является идентификатором, введенным без объявления в пределах структуры. Если вы попытаетесь скомпилировать этот код, компилятор по праву будет жаловаться и спрашивать Вас «откуда `Type` берётся?». У функций параметры вводятся в списке параметров (!), что выталкивает их в следующую область, тело функции. Это то, что нам здесь нужно. Мы должны сообщить компилятору, что `Type` является нашим заполнителем. Давайте продолжим наш воображаемый синтаксис:

```
module genericTree;

struct Tree(Type) {
    Type value;
    Tree[] children;
```

```
    size_t size() {
        size_t s = 1;
        foreach(child; children)
            s += child.size();
        return s;
    }

    bool isLeaf() @property {
        return children.length == 0;
    }
}
```

Смотрите, как я ввел `Type` в (одно-элементный) список параметров, справа после `Tree`? В идеале, это то, как должно выглядеть абстрагирование определения структуры от типа `value`, не так ли? Своего рода... рецепт, используемый для генерации нужного нам дерева `Tree`.

Не ищите дальше, предыдущее определение является стандартом D для шаблона структуры! Вы можете скомпилировать его и использовать сколько душе угодно (смотрите следующие главы, чтобы сделать это).

Вот суть шаблонов: написание некоего кода, рассматривая способ абстрагировать его по каким-либо элементам (типам, идентификаторам, числам, возможностей много) и определение такого универсального рецепта.

Что такое Шаблон?

В следующих главах вы увидите как определять шаблоны функций, структур и классов. Изящный синтаксис, продемонстрированный непосредственно перед этим разделом, представляет собой частный случай для этих конструкций. Это упрощенная версия полного синтаксиса декларации шаблона, который мы увидим в следующем разделе.

Но перед этим, я хотел бы представить, что такое шаблон на самом деле, потому что его определение является наиболее фундаментальным во всём документе. Как я сказал, шаблон является способом определить проект генерации некоего кода, будь то определение класса, функции или... чего? Что могло бы быть наиболее абстрактной единицей кода?

Скажем, вы имеете замечательный кусок кода, полный определений функций, структур и их методов, новых идентификаторов, и так далее. У этой части кода может быть несколько точек входа для будущей параметризации: какие-то типы, какие-то идентификаторы можно абстрагировать оттуда и поместить в список параметров шаблона.

```
// Может быть параметризовано по типу, как видели раньше
struct Tree { ... }

// Кроме того, может быть параметризовано по типу дерева
// И, может быть, по f?
Tree mapOnTree(Tree input, Tree delegate(Tree) f) { ... }

void printTree(Tree input) { ... }

// Что по поводу определения разных массивов TreeArray для всех возможных
деревьев?
alias TreeArray = Tree[];
```

Но где основная единица, содержащая этот код? Хорошо, это блок кода, конечно, или

область видимости. В идеале, мы хотели бы один из способов группировать все предыдущие определения в один блок, с тем же списком параметров:

```
// область видимости?
(Type)
{
    struct Tree { /* здесь используем Type */}
    Tree mapOnTree(Tree input, Tree delegate(Tree) f) { /* здесь тоже */ }
    void printTree(Tree input) { /* то же самое*/ }
    alias TreeArray = Tree[]; // Ты понял
}
```

Так как нам нужно будет «вызвать» это для производства некоторого кода (немного похоже на то, как вы вызываете функцию), этому блоку кода нужно имя. И затем, нам просто нужно сообщить компилятору: «то, что здесь находится, это — проект». Ключевое слово D для этого `template`:

```
template MyDefs(Type)
{
    struct Tree {...}
    Tree mapOnTree(Tree input, Tree delegate(Tree) f) {...}
    void printTree(Tree input) {...}
    alias TreeArray = Tree[];
}
```

Вот, пожалуйста. Это то, чем является шаблон по своей сути: именованный, параметризованный блок кода, готовый для реализации специально для вас.

Объявления Шаблонов

Вот синтаксис для объявления шаблона:

```
template templateName(list, of, parameters)
{
  // Здесь несколько синтаксически корректных объявлений
  // Аргументы доступны в области видимости шаблона.
}
```

templateName — ваш обычный D-идентификатор, а список параметров (list of parameters) — разделенный запятыми список с нулём или более параметров шаблона. Это могут быть:

Типы (`identifier`)

Идентификатор `identifier` сам по себе считается именем типа. Согласно общепринятому стилю D нужно использовать идентификаторы, начинаемые с большой буквы (`Range`, `Rest`), как для любых определенных пользователем типов. Многие шаблоны D используют традиции C++, в соответствии с которыми для имени типа используется одна большая буква, начиная с `T` (`U`, `V`,...). Не считайте себя ограниченными таким образом, используйте имена, которые делают ваши шаблоны более легкими для понимания.

Псевдонимы (`alias identifier`)

Вы объявляете их с идентификатором псевдонима `alias identifier`. Они принимают не типы, а идентификаторы: имена переменных, имена классов, даже имена других шаблонов. Они будут также принимать множество литералов времени компиляции: строки, массивы, литералы функций, ... По большей части,

если вам нужен широко-принимающий шаблон, используйте параметр-псевдоним. Заметьте однако, что они **не** примут встроенных типов в качестве аргументов, так как `int` не является допустимым идентификатором в D (это — ключевое слово).

Значения литералов (`typeName identifier`)

Они все объявляются подобно этому: `typeName identifier`. Значениями литералов могут быть целые величины (`int`, `ulong`, ...), основанные на перечислениях `enum`, строки string, символы `char`, значения с плавающей точкой или логические величины. Подойдёт любое выражение, которое может быть вычислено во время компиляции. Например: `int` depth или `string` name.

Кортежи параметров Шаблона (`identifier...`)

Синтаксис с `identifier...` (да, с тремя точками) и кортеж должен быть последним параметром шаблона. Кортежи параметров шаблона будут захватывать одним идентификатором **целый список параметров шаблона** (типы, имена, литералы, ...). Эти кортежи сохранят любой аргумент шаблона, который вы передадите в него. Если не передаётся никакого аргумента, вы просто получите пустой, нулевой-длины, кортеж. Действительно, так как они могут иметь дело с типами также, как с идентификаторами, эти кортежи являются некоей смесью типов, но они — удивительно мощны и просты в использовании, как вы увидите в разделе Кортежи.

Из всего этого, типы и псевдонимы наиболее часто используются, тогда как значения с плавающей точкой довольно редки: их использование в качестве аргументов для вычислений времени компиляции заменены на вычисление функций времени компиляции в D (Compile-Time Function Evaluation), также известный под именем **CTFE**. Вы увидите различные примеры использования этих параметров в этом документе.

Заметьте, что указатели, массивы, объекты (экземпляры классов), структуры или функции не являются частью этого списка. Но, как я уже говорил, параметры-псевдонимы позволяют вам захватывать и использовать ***имена*** массива, класса, функции или структуры, и получать доступ к их возможностям.

Псевдонимы, Идентификаторы и Имена. Существует большое различие между встроенными типами, такими, как `int` или `double[3]`, и типами, определяемыми пользователем. Тип, определяемый пользователем, скажем, класс с именем `MyClass` — это имя типа. Таким образом, это и тип (класс `MyClass`, принимаемый как аргумент шаблона тип) и имя, идентификатор (`MyClass`, принимаемый как параметр-псевдоним шаблона `alias`). С другой стороны, `int`, будучи ключевым словом D — не является ни идентификатором, ни именем. Это просто тип. Вы не можете передать его в параметр-псевдоним шаблона.

Тело шаблона может содержать любые стандартные объявления D: переменные, функции, классы, интерфейсы, другие шаблоны, объявления псевдонима, ... Единственное исключение, которое я смог придумать — это объявление модуля `module`, так как оно делается в области видимости верхнего уровня.

Синтаксис и Семантика. Код внутри объявления шаблона должен быть только

синтаксически правильным D-кодом (то есть: код, который выглядит похожим на код D). Семантика не проверяется до создания экземпляра. Это означает, что ваш код может казаться правильным, при написании шаблонов самих по себе, и компилятор не моргнет глазом, если вы не используете ваши шаблоны для создания их экземпляров.

В теле шаблона все параметры доступны как заменители для будущих аргументов. Также, собственное имя шаблона относится к его текущему экземпляру при генерации кода. Это, по большей части, используется в шаблонах структур и классов.

Вот некоторые примеры объявления шаблона:

```
1   module declaration;
2
3   template ArrayOf(T) // T — это тип
4   {
5       alias ArrayType   = T[];
6       alias ElementType = T;
7   }
8
9   template Transformer(From, To) // From (от) и To (до) — это тоже типы
10  {
11      To transform(From from)
12      {
13          import std.conv;
14          return to!(To)(from);
15      }
16
17      class Modificator
18      {
19          From f;
20          To t;
21          this(From f) { /*...*/ }
22      }
23  }
24
25  template nameOf(alias a)
26  {
27      enum string name = a.stringof; // enum: Объявление константы,
28                                     // определяемой во время компиляции.
29                                     // Смотрите ниже.
30  }
31
32  template ComplicatedOne(T, string s, alias a, bool b, int i)
33  { /* некоторый код, использующий T, s, a, b и i */ }
34
35  template Minimalist() {} // Ноль параметров в объявлении шаблона.
36
37  template OneOrMore(FirstType, Rest...) // Rest (остальное) — это кортеж.
38  { /*...*/ }
39
40  template ZeroOrMore(Types...) // Types — это кортеж.
41  { /*...*/ }
42
43  template Multiple(T)     { /*...*/ } // Версия с одним аргументом.
44  template Multiple(T,U)   { /*...*/ } // Два аргумента,
45  template Multiple(T,U,V) { /*...*/ } // и три.
```

Полный синтаксис для деклараций шаблона немного сложнее, я покажу больше в следующих секциях. Вы увидите, например, ограничения типа в разделе Специализации

Шаблонов, значения по умолчанию в разделе Значения по умолчанию, ограничения экземпляра в Ограничениях Шаблона, и дополнительно о кортежах в соответствующем разделе.

Существует ограничение, которое стоит иметь в виду: шаблоны могут быть объявлены почти в любой области видимости, кроме как внутри (обычной) функции.

> **enum**. В предыдущем коде, видите строку 27? Она определяет строку типа `string` с именем `name` как член шаблона `nameOf`. **`enum`**, расположенный прямо перед идентификатором, означает, что `name` является константой времени компиляции. Вы можете рассматривать его как тип класса памяти, наряду с **`immutable`** или **`const`**, и он означает, что значение полностью задано и зафиксировано во время выполнения. Вы увидите множество примеров **`enum`** в этом документе.

Создание экземпляра Шаблона

Синтаксис

Для того, чтобы создать экземпляр шаблона, используйте следующий синтаксис:

```
templateName!(list, of, arguments)
//  templateName — имя шаблона
//  list, of, arguments — список аргументов
```

Обратите внимание на восклицательный знак (!) перед списком аргументов, разделённым запятыми. Таким образом отличаются списки аргументов шаблона от обычных списков аргументов (функции). Если присутствует и то, и другое (для шаблонов функций), мы используем:

```
templateName!(template, argument, list)(runtime, agument, list)
//  template, argument, list — список аргументов шаблона
//  runtime, agument, list — список аргументов времени выполнения
```

Есть небольшая хитрость, позволяющая использовать укороченный синтаксис создания экземпляра: если список аргументов содержит только один аргумент с длиной в один литерал, вы можете опустить скобки, вот так:

```
templateName!argument
```

Таким образом, вот примеры допустимых экземпляров шаблонов:

```
Template!(int)
Template!int

Template!("string arg")
Template!"string arg"

map!(foo)(range); // foo — это идентификатор, захватывается
                  // с помощью псевдонима alias.
                  // range — это аргумент времени выполнения.
map!foo(range);   // то же самое

// Однако:
Multiple!(int, double)
// И нельзя:
// Multiple!int, double //??
```

Шаблоны как Аргументы шаблона

Аргументы сами могут являться результатом другого экземпляра шаблона. Если шаблон возвращает тип после создания экземпляра, он вполне подходит для использования его в другом списке аргументов шаблона. В этом документе вы будете регулярно видеть Матрёшку, вызываемую так:

```
firstTemp!(secondTempl!(Arguments), OtherArguments).
//  firstTemp - первый шаблон,  secondTempl — второй шаблон
//  Arguments — аргументы, OtherArguments — другие аргументы
```

Выбор среди объявлений

Компилятор просматривает объявления (если с вызываемым именем было объявлено больше одного шаблона) и выбирает один с корректным количеством аргументов и корректными типами для создания экземпляра. Если создать экземпляр можно у более, чем одного шаблона, он пожалуется и остановится в этом месте (хотя, взгляните на разделы специализации шаблона и ограничения шаблона).

Результат создания экземпляра

Когда вы создаёте экземпляр шаблона, глобальным эффектом будет появление новой именованной области видимости (блока кода), созданного в области видимости объявления шаблона. Имя этой новой области видимости является именем шаблона с его списком аргументов: `templateName!(args)`. Теперь параметры внутри этого блока «заменены» соответствующими аргументами (классы памяти применяются, переменные инициализируются, ...). Вот как возможные экземпляры предыдущих шаблонов могут выглядеть:

```
module instantiation;
import declaration;

void main()
{
    ArrayOf!(int).ArrayType myArray;

     // От псевдонима для типа double
     // К псевдониму для типа int
    alias transfo = Transformer!(double,int);

    struct MyStruct { /*...*/ }

    // "MyStruct" — это идентификатор -> захватывается псевдонимом alias
    auto name = nameOf!(MyStruct).name;

    alias complicatedExample =
    ComplicatedOne!( int[]   // тип
                   , "Hello" // строковый литерал
                   , ArrayOf // имя
                   , true    // логический литерал
                   , 1+2     // вычисляется до результата '3'.
                   );

    alias min1 = Minimalist!(); // Нет аргументов

    // FirstType - это 'int'
    // Rest - это 'double, string, "abc"'
    alias oneOrMore =
```

```
        OneOrMore!( int
                  , double, string, "abc"
                  );

    // Types - это одноэлементный кортеж: (int)
    alias zero1 = ZeroOrMore!(int);
    // Types - это (int, double, string)
    alias zero2 = ZeroOrMore!(int, double, string);
    // Types — это пустой кортеж: ()
    alias zero3 = ZeroOrMore!();

    // Выбирается одноаргументная версия
    alias mult1 = Multiple!(int);
    // Трёхаргументная версия
    alias mult2 = Multiple!(int,double,string);
    // Ошибка! Нет 0-аргументной версии
    //alias mult3 = Multiple!();
}
```

За пределами области видимости (то есть там, где вы поместили экземпляр шаблона в ваш собственный код), внутренние объявления доступны посредством их полной квалификации:

```
module internaldeclarations1;
import declaration;

// ArrayType доступен (это int[])
// array — это совершенно стандартный динамический массив целых.
ArrayOf!(int).ArrayType array;
ArrayOf!(int).ElementType element; // точно также, element — это int.

void main()
{
    // Функция transform доступна. Экземпляр, созданный таким образом —
    // это функция из double в string.
    auto s = Transformer!(double, string).transform(3.14159);
    assert(is(typeof(s) == string)); // s — это string
}
```

Очевидно, использование шаблонов таким образом, с их полным именем, сильно раздражает (*в оригинале* ***is a pain*** *— это боль — прим. пер.*). Изящные объявления псевдонимов D `alias` являются вашими друзьями:

```
module internaldeclarations2;
import declaration;

alias DtoS = Transformer!(double, string);

void main()
{
    auto s = DtoS.transform(3.14159);
    auto m = new DtoS.Modificator(1.618); // DtoS.Modificator — это класс,
                                          // сохраняющий double и string.
}
```

Вы должны иметь в виду, что создание экземпляра шаблона означает генерацию кода. Используя различные аргументы в разных местах вашего кода, вы будете создавать экземпляры как *множество различных именованных областей видимости*. Это основное отличие от дженериков (generics) в таких языках, как Java или C#, где обобщённый код создается только один раз, и используется тип erasure (стирание), чтобы связать всё это вместе. С другой стороны, создание экземпляра того же шаблона, с теми же аргументами,

создаст только одну часть кода. (*Честно говоря, никогда не пользовался дженериками в Java, а C# вообще не знаю, поэтому мог и накосячить в переводе этого места — прим. пер.*)

```
module differentinstantiations;
import declaration;

alias StoD = Transformer!(string, double);
alias DtoS = Transformer!(double, string);
alias StoI = Transformer!(string, int);
// Теперь мы можем использовать три различные функции и три различных класса.
```

void. Заметьте, что `void` — допустимый тип D и, как таковой, является допустимым аргументом шаблона для параметра типа. Будьте внимательны: многие шаблоны не имеют смысла, когда `void` используется в качестве типа. В следующих разделах и в приложении вы увидите способы ограничения аргументов только определенными типами.

Строительные блоки Шаблона

До этого момента, шаблоны могут показаться вам неинтересными, даже с простым синтаксисом объявления и создания экземпляра. Но подождите! D вводит несколько искусных трюков, которые одновременно упрощают и существенно расширяют использование шаблона. Этот раздел представит вам ваших будущих наилучших друзей, основания, на которых будут строиться ваши шаблоны.

Одноимённый (Eponymous) Трюк

Если в шаблоне объявляется идентификатор **с таким же именем** (по-гречески: eponymous), как и объемлющий шаблон, этот идентификатор допускается соотнести с ним в момент создания экземпляра шаблона. Это неплохо очищает ваш код:

```
module pair;

template pair(T)
{
    // шаблон 'pair' объявляет только член 'pair'
    T[] pair(T t) { return [t,t];}
}

auto array = pair!(int)(1);  // не требуется писать pair!(int).pair(1)
```

или:

```
module nameof1;

template nameOf(alias name)
{
    enum string nameOf = name.stringof;
}

struct Example { int i;}

void main()
{
    Example example;

    auto s1 = nameOf!(Example);
```

```
    auto s2 = nameOf!(example);

    assert(s1 == "Example");
    assert(s2 == "example");
}
```

Раньше здесь было ограничение, в соответствии с которым одноимённый трюк работал *только* в том случае, если вы определяете один (и только один) идентификатор. Даже если остальные идентификаторы были закрытыми (private), они ломали одноимённую подстановку. Недавно (осенью 2012 года) это изменилось, и теперь мы можем делать:

```
module record;

template Record(T, U, V)
{
    import std.typecons: Tuple, tuple;

    // Реальная работа выполняется здесь
    // Используйте столько идентификаторов, сколько вам нужно.
    alias Tuple!(T,U) Pair;
    alias Pair[V] AssocArray;
    alias AssocArray[] Record;
}
```

Заметьте, что в этом случае одноимённый член является псевдонимом, тогда как он был функцией или объявлением константы (`enum`) в предыдущих примерах. Как уже было сказано, действует любой член с тем же именем.

И затем, использование `Record`:

```
module using_record;
import record;

Record!(int, string, double[]) recordslist;
/* ... */
```

Хотя, в этом случае, одноимённый член прячет остальные члены: `Pair` и `AssocArray` больше не доступны. Это кажется логичным, поскольку, если вы используете одноимённый трюк, он должен предоставить упрощенный интерфейс вашим пользователям.

Внутренний псевдоним

Обычное использование шаблонов выполняет своего рода магию с типами: выведение типов, сборка их новым способом, и т.п. Типы не являются сущностями первого класса в D (нет типа `type`), но ими легко можно манипулировать, как любым другим идентификатором, через его псевдоним. Так, когда шаблон должен вывести наружу тип, это делается через придание ему нового имени.

```
module allarrays;

template AllArraysOf(T)
{
    alias T    Element;           // Элемент
    alias T*   PointerTo;         // Указатель на него
    alias T[]  DynamicArray;      // Динамический массив
    alias T[1] StaticArray;       // Статический массив
    alias T[T] AssociativeArray;  // Ассоциативный массив
}
```

Выведение наружу параметров Шаблона. Хотя они являются частью имени шаблона, параметры не являются непосредственно доступными извне. Имейте в виду, что имя шаблона — просто имя области видимости. Как только создан его экземпляр, все эти `T` и `U` сами по себе больше не существуют. Если они вам нужны извне, выведите их наружу через член шаблона, как это сделано с `AllArraysOf.Element`. Вы найдёте другие подобные примеры в разделах Шаблоны структур и Шаблоны классов.

static if

Синтаксис

Конструкция **`static if`**[2] позволяет вам выбирать между двумя частями кода во время компиляции. Это не специфично для шаблонов (вы можете использовать её и в других частях вашего кода), но она невероятно полезна, чтобы ваши шаблоны приспосабливались к аргументам. Таким образом, используя вычисленные во время компиляции предикаты, основанные на аргументах шаблона, вы сгенерируете различный код и настроите шаблон под ваши потребности.

Синтаксис:

```
static if (compileTimeExpression)
{
     /* Код, созданный, если результат compileTimeExpression - true */
}
else /* опционально */
{
     /* Код, созданный в случае false */
}
```

Что здесь действительно важно — это небольшая магия компилятора: как только путь кода будет выбран, результирующий код воплощается в теле шаблона, но без фигурных скобок. В противном случае создалась бы локальная область видимости, скрывающая всё, что происходит внутри, и она бы резко ограничила мощность выражения **`static if`**. Так что фигурные скобки там только для того, чтобы сгруппировать утверждения вместе.

Если присутствует только одно утверждение, можно полностью избавиться от скобок, вы это часто увидите в D-коде. Например, предположим, вам нужен шаблон, который «возвращает» истину, если переданный тип — это динамический массив, и ложь в противном случае (этот своего рода шаблон-предикат разработан немного глубже в разделе **Шаблоны-Предикаты**).

```
module isdynamicarray;

template isDynamicArray(T)
{
    static if (is(T t == U[], U))
        enum isDynamicArray = true;
    else
        enum isDynamicArray = false;
}
```

Как вы можете видеть, без фигурных скобок после **`static if`** и с одноимённым трюком (`isDynamicArray` — идентификатор, определенный шаблоном и тип, автоматически

2 Это одновременно утверждение и объявление, так что я назову это конструкция.

выведенный компилятором), результатом стал очень чистый синтаксис. Выражение `is()` является способом получить интроспекцию времени компиляции, которое идет рука об руку со `static if`. В конце этого документа есть интенсивный курс по нему (смотрите **Приложение - Выражение is**).

Необязательный код

Обычное использование `static if` — включать или выключать код: единичный `static if` без пункта `else` сгенерирует код только тогда, когда условие является истиной. Вы можете найти множество примеров этой идиомы в модуле библиотеки std.range, где диапазоны высокого уровня (диапазоны, обёртывающие другие диапазоны), активизируют некоторую функциональность тогда и только тогда, когда обёрнутый (wrapped) диапазон поддерживает её, подобно такому:

```
/* Мы - внутри шаблона структуры MyRange, обёртки для R. */

    R innerRange;

/* Код, который присутствует во всех экземплярах MyRange */
(...)

/* необязательный (опциональный) код */
static if (hasLength!R) // действительно innerRange имеет метод .length()?
    auto length()       // Тогда MyRange тоже его имеет.
    {
        return innerRange.length;
    }

static if (isInfinite!R)     // innerRange является бесконечным диапазоном?
    enum bool empty = false; // Тогда MyRange тоже бесконечен.
// И так далее…
```

Вложенные `static if`

Выражения `static if` могут быть вложенными: просто поместите другой `static if` после `else`. Вот шаблон, выбирающий псевдоним:

```
module selector;
import std.traits: isIntegral, isFloatingPoint;

template selector(T, alias intFoo, alias floatFoo, alias defaultFoo)
{
    static if (isIntegral!T)
       alias intFoo selector;
    else static if (isFloatingPoint!T)
       alias floatFoo selector;
    else // в остальных случаях
        alias defaultFoo selector;
}
```

Если Вам нужна своего рода конструкция `static switch`, посмотрите раздел [**примеры-staticswitch**].

Рекурсия со `static if`

Ранг:

Теперь, давайте используем `static if` для чего-то немного более сложного, чем

просто диспетчеризация между путями кода: рекурсия. Что, если вы знаете, что вы получите n-мерные массивы (простые массивы, массивы массивов, массивы массивов массивов, ...), и захотите использовать самую быструю, супер-оптимизированную числовую функцию для 1-мерного массива, другую для 2D-массива и ещё одну для более высокоуровневых массивов? Абстрагируясь от этого, нам нужен шаблон, выполняющий некоторую интроспекцию на типах, которая возвращает 0 для элемента (что-нибудь, что не является массивом), 1 для 1-мерного массива (`T[]`, для некоторого T), 2 для 2-мерного массива (`T[][]`), и так далее. Математики называют это *ранг* массива, так что мы воспользуемся этим. Определение вполне рекурсивное:

```
1   module rank1;
2
3   template rank(T)
4   {
5       static if (is(T t == U[], U))         // Является ли T массивом для
6                                             // некоторого типа U?
7           enum size_t rank = 1 + rank!(U); // тогда давайте выполним рекурсию вниз.
8       else
9           enum size_t rank = 0;             // Базовый случай, конец рекурсии.
10  }
```

Строки 5 и 7 - наиболее интересны: с некоторой магией выражения **`is`**, `U` был выведен компилятором и стал доступен в одной из веток **`static if`**. Мы используем его, чтобы снять один уровень `[]` с типа и выполнить рекурсию вниз, используя `U` как новый тип для установления ранга. Или `U` сам является массивом (в этом случае рекурсия продолжится), или она попадёт в базовый случай и остановится там. Поскольку шаблон определяет элемент с именем, таким же, как его собственное, результат доступен непосредственно: любой экземпляр шаблона `rank` будет величиной типа `size_t`.

Давайте используем его:

```
module using_rank1;
import rank1;

static assert(rank!(int)        == 0);
static assert(rank!(int[])      == 1);
static assert(rank!(int[][])    == 2);
static assert(rank!(int[][][]) == 3);

/* Это будет работать для любого типа, очевидно */
struct S {}

static assert(rank!(S)  == 0);
static assert(rank!(S[])== 1);
static assert(rank!(S*) == 0);
```

static assert. Устанавливая **`static`** перед **`assert`** мы заставляем **`assert`** выполниться во время компиляции. Использование выражения **`is`** как пункт теста дает утверждение о типах. Один из обычных вариантов использования **`static assert`** — это остановить компиляцию, например, если мы попали на плохую ветку кода, используя `static assert(false, someString)` (or **`static assert`**`(0, someString)`). После этого строка будет выдана в качестве сообщения ошибки компилятора.

Ранг для Диапазонов:

D имеет интересную концепцию последовательности под названием диапазон (*range*). Стандартная библиотека Phobos поставляется со встроенными тестирующими шаблонами в std.range. Почему бы не расширить `rank` так, чтобы он имел дело с диапазонами и посмотреть, является ли что-то диапазоном диапазонов или более? Тип может быть протестирован, является ли он дипазоном с помощью `isInputRange` и его тип элемента получается применением `ElementType` к типу дипазона. Оба шаблона находятся в std.range. Кроме того, поскольку массивы входят в понятие диапазона, мы можем полностью выбросить вариант для массивов и использовать только диапазоны. Вот немного модифицированная версия `rank`:

```
module rank2;
import std.range;

template rank(T)
{
    static if (isInputRange!T)                  // T — это диапазон?
        enum size_t rank = 1 + rank!(ElementType!T); // если да, то рекурсия
    else
        enum size_t rank = 0;                   // базовый случай, остановка здесь
}

unittest
{
    auto c = cycle([[0,1],[2,3]]); // == [[0,1],[2,3],[0,1],[2,3],[0,1]...
    assert(rank!(typeof(c)) == 2); // диапазон диапазонов
}
```

Тип базового элемента:

С шаблоном `rank` у нас теперь есть возможность получить число скобок [] в типе массива (T[][][]) или уровень вложенности в диапазоне диапазонов. Дополнительный запрос может быть в получении типа базового элемента, T, из любого массива массивов... T или соответствующий эквивалент для диапазона. Здесь это:

```
1  module baseelementtype;
2  import std.range;
3  import rank2;
4
5  template BaseElementType(T)
6  {
7      static if (rank!T == 0)       // не диапазон
8          static assert(0, T.stringof ~ " - не диапазон.");
9      else static if (rank!T == 1) // простой диапазон
10         alias ElementType!T                         BaseElementType;
11     else                          // по крайней мере, диапазон диапазонов
12         alias BaseElementType!(ElementType!(T)) BaseElementType;
13 }
```

Строка 8 - пример остановки компиляции с помощью **`static assert`**, если мы когда-либо попадём в плохую ветку кода. Строка 12 - пример вызова Матрёшки: шаблон использует вызов другого шаблона как параметр.

Генерация массивов:

Теперь, что насчёт того, чтобы стать более производительными при переворачивании

процесса? Дан тип `T` и ранг `r` (типа `size_t`), мы хотим получить `T[][]...[]`, с `r` уровнями скобок `[]`. Ранг 0 означает создание `T`, как типа результата.

```
1  module ndim;
2
3  template NDimArray(T, size_t r)
4  {
5      static if (r == 0)
6          alias T NDimArray;
7      else
8          alias NDimArray!(T, r-1)[] NDimArray;
9  }
```

Здесь, рекурсия выполняется в строке 8: мы создаём экземпляр `NDimArray!(T,r-1)`, который является типом, затем создаем массив из него, устанавливая `[]` в конце, и выводим это наружу через псевдоним. Это также хороший пример использования целой величины, `r`, как параметра шаблона.

```
module using_ndim;
import ndim;

alias NDimArray!(double, 8) Level8;
static assert(is(Level8 == double[][][][][][][][]));
static assert(is(NDimArray!(double, 0) == double));
```

Повторяющаяся композиция:

Как последний пример, мы используем псевдоним параметра шаблона в сочетании с некоторой рекурсией `static if`, чтобы определить шаблон, который создает «экспоненциальную» функцию, также известную под именем повторяющейся композиции. Вот что я имею в виду:

```
module using_power;
import repeatedcomposition;

// стандартная функция
string foo(string s) { return s ~ s;}

// Шаблоны функции. Вы увидите их скоро.
Arr[] makeArray(Arr)(Arr array) { return [array,array];}

void main()
{
   // power!(foo, n) — это функция.
   assert(power!(foo, 0)("a") == "a");                 // функция идентичности
   assert(power!(foo, 1)("a") == foo("a"));            // "aa"
   assert(power!(foo, 2)("a") == foo(foo("a")));       // "aaaa"
   assert(power!(foo, 3)("a") == foo(foo(foo("a")))); // "aaaaaaaa"

   // Это даже лучше с шаблонами функций:
   assert(power!(makeArray, 0)(1) == 1);
   assert(power!(makeArray, 1)(1) == [1,1]);
   assert(power!(makeArray, 2)(1) == [[1,1],[1,1]]);
   assert(power!(makeArray, 3)(1) == [[[1,1],[1,1]],[[1,1],[1,1]]]);
}
```

Во-первых, это шаблон, который «возвращает» функцию (скорее, становится ею). Это легко выполнить через одноимённый трюк: просто определить в шаблоне функцию с тем же именем. Во-вторых, он чисто рекурсивен в своем определении, с двумя базовыми случаями:

если показатель является нулем, тогда мы должны произвести функцию идентичности (тождества), а если показатель — единица, мы должны просто вернуть саму входную функцию. Тут, как говорится, `power` (степень) пишет сама себя:

```
1   module repeatedcomposition;
2
3   template power(alias fun, uint exponent)
4   {
5       static if (exponent == 0) // вырожденный случай -> функция идентичности
6           auto power(Args)(Args args) { return args; }
7       else static if (exponent == 1) // случай конца рекурсии -> fun
8           alias fun power;
9       else
10          auto power(Args...)(Args args)
11          {
12              return .power!(fun, exponent-1)(fun(args));
13          }
14  }
```

> **.power** Если вы удивляетесь, что такое с синтаксисом `.power` в строке 12, то он таков, потому что, определяя одноимённый шаблон, мы прячем имя родительского шаблона. Так, внутри `power(Args...)`, `power` имеет отношение к `power(Args...)`, а не к `power(alias fun, uint exponent)`. Здесь нам требуется новый `power`, который нужно сгенерировать, так что мы вызываем глобальный шаблон `power` с оператором «глобальной области видимости» (`.`).

Во всех трех ветках `static if`, `power` выводит наружу член `power`, активизируя одноимённый трюк шаблона и позволяя легкое использование клиентом. Заметьте, что этот шаблон будет работать не только для одноаргументных функций, но также для функций с n-аргументами[3], для делегатов и для структур или классов, в которых определён оператор `()` (т.е., `opCall`), и для шаблонов функций... [4]

> Ну а теперь вы начинаете видеть силу шаблонов?

> *Здесь ещё была врезка с заголовком «**Curried Templates?**». Примеров кода там не было, и в конце присутствует фраза «это выходит за рамки данного документа». Раньше я не встречался с так называемым каррированием, и моих знаний оказалось недостаточно для достоверного перевода этой врезки. Т.к. лучше не переводить вообще, чем переводить с возможными ошибками, то я эту врезку пропускаю — прим. пер.*

Специализации Шаблонов

Вплоть до этого момента, при написании T в списке параметров шаблона, не было никаких ограничений на тип, которым может быть T при создании экземпляра. Специализация Шаблона — это небольшой «подсинтаксис», ограничивающий экземпляры шаблонов в подмножество всех возможных типов, и направляющий компилятор при создании экземпляра к конкретной версии шаблона из нескольких. Если вы читали

3 За исключением вырожденного случая, n=0, так как функция идентичности, определенная выше, принимает только один аргумент. Версия с более, чем одним аргументом, возможна, но она должна возвращать кортеж.

4 Я тут немного обманул, поскольку результирующая функция принимает любое количество аргументов любого типа, все-же стандартные проверки параметров функции остановят что-нибудь неблагоприятное, что может случиться. Более чистая (но более длинная, и, для шаблона функции, более сложная) реализация должна размножать начальный параметр функции с помощью кортежа.

[**Приложение — выражение is**] о выражении `is`(), вы уже знаете как это записать. Если ещё не читали, пожалуйста сделайте это теперь, так как это на самом деле тот же синтаксис. Эти специализации являются прямыми наследниками шаблонов C++, вплоть до способа их написания, и они существовали в D с самого начала, задолго до того, как были добавлены конструкции `static if` или ограничения шаблонов.

Специализации добавляются в список параметров шаблона, в части `(T, U, V)` определения шаблона. `Type : OtherType` ограничивает неявное преобразование `Type` в `OtherType`.

```
module specialization1;

template ElementType(T : U[], U) // может создавать экземпляр только с массивами
{
    alias U ElementType;
}

template ElementType(T : U[n], U, size_t n) // только со статическими массивами
{
    alias U ElementType;
}

class Array { alias int ElementType;}

template ElementType(T : Array)
{
    alias Array.ElementType ElementType;
}
```

Теперь, вся идея может показаться вам странной: если вы знаете, что вы хотите ограничить `Type`, чтобы он был `AnotherType`, зачем делать его параметром шаблона? В этом основное применение специализаций шаблонов: вы можете написать различные реализации шаблона (очевидно, с одинаковым именем), и когда потребуется создать экземпляр, компилятор автоматически решит, какой из них использовать, выбрав «наиболее приспособленный» к предоставленным аргументам. Этот выбор «наиболее приспособленного» подчиняется несколько усложнённым правилам, которые вы можете найти на веб-сайте языка программирования D, но они действуют естественным образом большую часть времени. Аккуратный подход такой: вы можете определить общий шаблон *и* некоторую специализацию. Специализированные версии будут выбраны по-возможности.

```
module specialization2;

template InnerType(T : U*, U) // Специализация для указателей
{
    alias U InnerType;
}

template InnerType(T : U[], U) // Специализация для динамических массивов
{ /*...*/ }

template InnerType(T) // Стандарт, случай по-умолчанию
{ /*...*/ }

void main()
{
    int* p;
    int i;
```

```
    alias InnerType!(typeof(p)) Pointer; // выбрана версия для указателей
    alias InnerType!(typeof(i)) Default; // выбран стандартный шаблон
}
```

Эта идиома часто используется в C++, где нет (встроенной) конструкции **`static if`** или ограничений шаблонов. В самых старых D-шаблонах это тоже часто использовалось, но с тех пор в течение нескольких лет появились другие способы, последний D-код кажется более ориентированным на ограничения: взгляните на сильно шаблонизированные модули Phobos, например std.algorithm или std.range.

Специализации, static if или Ограничения шаблонов? Да, действительно вопрос. Давайте подождём с этой дискуссией до момента, когда мы увидим все три подсистемы.

Значения по-умолчанию

Подобно параметрам функций, параметры шаблонов могут иметь значения по-умолчанию. Синтаксис тот же: `Param = defaultValue`. Значением по-умолчанию может быть что-то, что имеет смысл для соответствующего параметра: тип, литерал, идентификатор или другой параметр шаблона.

```
module def;

template Default(T = int, bool flag = false)
{
    static if (flag)
        alias T Default;
    else
        alias void Default;
}

alias Default!(double) D1;        // Создаётся Default!(double, false)
alias Default!(double, true) D2; // Создаётся Default!(double, true) (Doh!)
alias Default!() D3;              // Создаётся Default!(int, false)
```

В отличие от параметров функции, благодаря Специализациям шаблонов или IFTI, некоторые параметры шаблона могут быть автоматически вычислены компилятором. Так, параметрам шаблона по-умолчанию не требуется быть последними параметрами в списке:

```
module deduced;
import std.typecons: Tuple;

template Deduced(T : U[], V = T, U)
{
    alias Tuple!(T,U,V) Deduced;
}

alias Deduced!(int[], double) D1; // U вычислено в int. V заставили быть double.
alias Deduced!(int[]) D2;         // U вычислено в int. V тоже int.
```

> **Специализация и Значение по умолчанию?** Да, вы это можете. Разместите сначала специализацию, затем значение по умолчанию. Как-то так: `(T : U[] = int[], U)`. Все-же, этим обычно не пользуются.

Как и для функций, хорошо выбранное значение по-умолчанию может существенно упростить стандартные вызовы. Посмотрите, например, std.algorithm.sorting.sort. Его параметрами являются предикат и swapping-стратегия, но и тот, и другой адаптированы к значениям, которые нужны большинству людей при сортировке. Таким образом,

большинство клиентов будут использовать шаблон в коротком и чистом виде, но настроить его при необходимости всё ещё возможно.

TODO Maybe something on template dummy parameters.

Шаблоны функций

Синтаксис

Если вы пришли из языков с дженериками, возможно, вы думали о D-шаблонах, как о параметризованных классах и функциях, и не видели никакого интереса в предыдущих разделах (действующих на типах?). Не бойтесь, вы также можете создавать такие обобщённые по типу функции в D, с дополнительной силой шаблонов.

Как мы видели в разделе Одноимённый трюк, если вы определяете функцию внутри шаблона и используете для неё собственное имя шаблона, вы можете легко её вызывать:

```
module function_declaration1;
import std.conv : to;

// объявление:
template myFunc(T, int n)
{
    auto myFunc(T t) { return to!int(t) * n;}
}

void main()
{
    // вызов:
    auto result = myFunc!(double, 3)(3.1415);

    assert(result == to!int(3.1415)*3);
}
```

Это хорошо, но полная история даже лучше. Для начала, в D есть простой способ объявления шаблона функции: просто поместите список параметров шаблона перед списком аргументов:

```
module function_declaration2;
import std.conv:to;

string concatenate(A,B)(A a, B b)
{
    return to!string(a) ~ to!string(b);
}

Arg select(string how = "max", Arg)(Arg arg0, Arg arg1)
{
    static if (how == "max")
        return (arg0 < arg1) ? arg1 : arg0;
    else static if (how == "min")
        return (arg0 < arg1) ? arg0 : arg1;
    else
        static assert(0,
        "select: строка 'how' должна быть \"max\" или \"min\".");
}
```

Красиво и чисто, да? Обратите внимание, что возвращаемый тип тоже может быть шаблонизирован, используя `Arg` как возвращаемый тип в функции `select`.

auto return

Поскольку вы можете выбирать среди веток кода, возвращаемый тип функции может сильно меняться, в зависимости от параметров шаблона, переданных вами. Используйте `auto` для упрощения кода:

```
module morph;

// Что Morph будет возвращать, будет сильно зависеть от T и U
auto morph(alias f, T, U)(U arg)
{
    static if (is(T == class))
        return new T(f(arg));
    else static if (is(T == struct))
        return T(f(arg));
    else
        return; // функция, возвращающая void.
}
```

auto ref. Шаблон Функции может иметь возвращаемый тип `auto ref`. Это означает, что для шаблонов, где возвращаемые величины являются lvalues (присваиваемые), шаблон получит `ref`-версию. И не-`ref`-версию, если нет. Я должен добавить несколько примеров для этого поведения.

IFTI

Даже лучше — **Неявное создание экземпляра шаблона функции** (Implicit Function Template Instantiation, IFTI), что означает, что компилятор обычно будет способен автоматически определять параметры шаблона, изучая аргументы функции. Если несколько аргументов шаблона являются чистыми параметрами времени компиляции, просто предоставьте их непосредственно:

```
module ifti;
import function_declaration2;

struct Foo {}

void main()
{
    string res1 = concatenate(1, 3.14); // A - это int, а B - это double
    string res2 = concatenate("abc", Foo()); // A — это string, B - это Foo

    auto res3 = select(3, 4); // how стало "max", Arg - это int.
    auto res4 = select!"min"(3.1416, 2.718); // how стало "min", Arg - это double.
}
```

Как вы можете видеть, это приводит к очень простому коду вызова. Итак, мы можем объявлять и вызывать шаблоны функций с одинаково чистым синтаксисом. То же можно делать со структурами или классами, как вы увидите в следующих разделах. Фактически, синтаксис настолько чист, что вы, как и я иногда, можете забывать время от времени, что вы манипулируете **не** функцией (или структурой, и т.п.): вы манипулируете шаблоном, параметризованной частью кода.

Мантра. Шаблоны `XXX` — это не `XXX`, это шаблоны. `XXX` — это любое из (функция, структура, класс, интерфейс, объединение union). Шаблоны — это параметризованные области видимости, и они не являются объектами первого класса в D: у них нет типа, их **нельзя** присвоить переменной, их **нельзя**

возвращать из функций. Это означает, например, что вы не можете возвращать шаблоны функций, вы не можете унаследовать от шаблонов класса, и так далее. Конечно, *созданные экземпляры* шаблонов — отличные примеры функций, классов, и тому подобное. Т.е. вы можете их унаследовать, возвращать…

Мы можем столкнуться с **Мантрой** снова в этом учебном пособии.

Пример: Сглаженные массивы и диапазоны

Давайте используем то, что мы только что видели, для чего-то конкретного. В D вы можете манипулировать 2D и 3D массивами, но иногда нужно обрабатывать их линейно. На момент написания статьи, ни std.algorithm ни std.range не предоставляли функции `flatten` (разглаживание). На примере простых массивов, вот то, что нам нужно:

```
module using_flatten1;
import flatten1;

void main()
{
    assert( flatten([[0,1],[2,3],[4]]) == [0,1,2,3,4] );
    assert( flatten([[0,1]]) == [0,1] );
    assert( flatten([0,1]) == [0,1] );
    assert( flatten(0) == 0 );

    assert( flatten([[[0,1],[]], [[2]], [[3], [4,5]], [[]], [[6,7,8]]])
            == [0,1,2,3,4,5,6,7,8] );
}
```

Итак, рассмотрев примеры, нам в качестве аргументов нужен простой массив (ранг == 1) или не-массив (ранг == 0), чтобы `flatten` никак не влияла на них: она их просто вернёт. Для массивов ранга 2 или выше, она обрушивает элементы вплоть до массива ранга 1. Она классически рекурсивна: мы применим `flatten` на все под-массивы с помощью std.algorithm.map и объединим элементы с помощью std.algorithm.reduce:

```
module flatten1;
import std.algorithm;
import rank2;

auto flatten(Arr)(Arr array)
{
    static if (rank!Arr <= 1)
        return array;
    else
    {
        auto children = map!(.flatten)(array);
        return reduce!"a~b"(children); // объединение children
    }
}
```

Как я понимаю, сейчас, на момент перевода (в декабре 2015 года) вместо `import std.algorithm;` *нужно писать* `import std.algorithm.iteration;` *— прим. пер.*

Мы эффективно используем здесь ключевое слово D `auto` для параметра возврата у функций. Фактически, единственный вызов `flatten` создаст один экземпляр на уровень, все с различным типом возврата.

Заметьте, что `flatten` прекрасно работает также на диапазонах, но не лениво (not

lazy): она охотно сцепляет все элементы вплоть до самого последнего во внутреннем диапазоне. Диапазоны ленивы, хорошая реализация `flatten` для них должна сама быть диапазоном, который подаёт элементы по одному, вычисляет следующий только тогда, когда просят (и таким образом сможет работать на бесконечных или очень длинных диапазонах тоже, чего предшествующая простая реализация не может делать). Такая реализация означает создание шаблона структуры с фабричной функцией. Вы найдете её здесь в качестве примера.

Отталкиваясь от нашей текущей реализации `flatten`, вот интересное упражнение на добавление другого параметра: количество уровней, которое вы хотите выровнять. Только первые три уровня или последние два внутренних, например. Просто добавьте целый параметр шаблону, который увеличивается (или уменьшается) при входе в рекурсию, и другой случай остановки для рекурсии. Положительные уровни могли бы означать самые крайние уровни, тогда как отрицательный аргумент должен действовать на внутренние. Возможное использование должно выглядеть как-то так:

```
flatten!1([[[0,1],[]], [[2]], [[3], [4,5]], [[]], [[6,7,8]]]);
     // == [[[0,1],[],[2],[3], [4,5],[],[6,7,8]]
flatten!2([[[0,1],[]], [[2]], [[3], [4,5]], [[]], [[6,7,8]]]);
     // == [0,1,2,3,4,5,6,7,8]
flatten!0([[[0,1],[]], [[2]], [[3], [4,5]], [[]], [[6,7,8]]]);
     // ==[[[0,1],[]], [[2]], [[3], [4,5]], [[]], [[6,7,8]]]
flatten!(-1)([[[0,1],[]], [[2]], [[3], [4,5]], [[]], [[6,7,8]]]);
     // ==[[[0,1]], [[2]], [[3,4,5]], [[]], [[6,7,8]]]
```

Шаблоны анонимных функций

В D вы можете определять анонимные функции (даже делегаты, а именно: замыкания):

```
module anonymous_function1;

auto adder(int a)
{
    return (int b) { return a+b;};
}

unittest
{
    auto add1 = adder(1); // add1 - это int delegate(int)
    assert(add1(2) == 3);
}
```

В предыдущем коде `adder` возвращает анонимного делегата. Может `adder` быть шаблоном? Ха! Вспомните **Мантру**: шаблоны функций являются шаблонами и их нельзя возвращать. Для этой конкретной проблемы есть два возможных решения. Либо вам не нужно нового типа и просто используете `T`:

```
module anonymous_function2;

auto adder(T)(T a)
{
    return (T b) { return a+b;};
}

unittest
{
    auto add1f = adder(1.0); // add1f - это float delegate(float)
```

```
    assert(add1f(2.0) == 3.0);

    import std.bigint;

    // addBigOne принимает BigInt и возвращает BigInt
    auto addBigOne = adder(BigInt("1000000000000000"));
    assert(addBigOne(BigInt("1")) == BigInt("1000000000000001"));

    // Но:
    // auto error = add1(3.14); // Ошибка! Ожидается int, получен double.
}
```

В предыдущем примере возвращаемый анонимный делегат **не** является шаблоном. Здесь просто используется `T`, которое будет вполне определенным, как только экземпляр создан. Если вам действительно нужно вернуть что-то, что может быть вызвано с любым типом, используйте внутреннюю структуру (смотрите раздел о внутренних структурах).

Теперь, это может стать сюрпризом для вас, что D **имеет** шаблоны анонимных функций. Синтаксис является очищенной версией анонимных функций:

```
(a,b) { return a+b;}
```

Да, предыдущий скелет функции — это анонимный шаблон. Но, помните **Мантру**: вы не можете его возвращать. И из-за (на моих глазах) ошибки в грамматике псевдонима, вы не можете записать в идентификатор его псевдоним:

```
alias (a){ return a;} Id; // Ошибка!
```

Так в чем от них польза? Вы можете использовать их с помощью псевдонимов параметров шаблона, когда *these stand for* функций и шаблонов функций:

```
module calltwice;

template callTwice(alias fun)
{
    auto callTwice(T)(T t)
    {
        return fun(fun(t));
    }
}

unittest
{
    alias callTwice!( (a){ return a+1;}) addTwo;
    assert(addTwo(2) == 4);
}
```

Так как они являются делегатами, они могут захватывать локальные идентификаторы:

```
module using_calltwice;
import calltwice;

unittest
{
    enum b = 3; // Объявление константы, инициализирована в 3
    alias callTwice!( (a){ return a+b;}) addTwoB;
    assert(addTwoB(2) == 2 + 3 + 3);
}
```

Замыкания — Объекты для бедных

Замыкания D могут обернуть среду выполнения и хранить её возле своих горячих

маленьких сердец. Мы, конечно, можем создать их с помощью шаблона. Для того, чтобы получить эквивалент объекта, давайте создадим функцию, которая возвращает кортеж (`std.typecons.Tuple` с именованными аргументами).

```
module makecounter;
import std.traits;
import std.typecons;

auto makeCounter(T)(T _counter = T.init) if (isNumeric!T)
{
    bool sense = true;
    auto changeSense = () { sense = !sense;};
    auto inc = (T increment)
               { _counter += (sense ? increment : -increment); };
    auto dec = (T decrement)
               { _counter += (sense ? -decrement : decrement); };
    auto counter = (){ return _counter;};

    return Tuple!( typeof(changeSense), "changeSense"
                 , typeof(inc), "inc"
                 , typeof(dec), "dec"
                 , typeof(counter), "counter")
                 (changeSense, inc, dec, counter);
}
```

Часть `if` после списка аргументов является просто разумным ограничением шаблона (они описаны в разделе об ограничениях). Возвращаемый `Tuple` (кортеж) немного тяжеловат на мой вкус, но использование именованных полей кортежа дает нам хороший объекто-подобный синтаксис вызова[5]. Вместо этого можно было бы использовать просто `return tuple(changeSense, inc, dec, counter);`, но тогда внутренние замыкания были бы доступны по их индексу, а не по имени.

Вот как это использовать:

```
module using_makecounter;
import makecounter;

void main()
{
    auto c = makeCounter(0); // T - это int.
    auto c2 = makeCounter!int; // Тоже самое.

    c.inc(5);
    assert(c.counter() == 5);
    c.inc(10);
    assert(c.counter() == 15);
    c.changeSense(); // теперь каждый inc (приращение) фактически
                     // будет вычитаться
    c.inc(5);
    assert(c.counter() == 10);
}
```

Function Overloading

Незакончено. Ок, мне нужно что-то написать об этом.

Классы памяти

Как мы видели в разделе Создание экземпляра Шаблона, классы памяти применяются к

5 Смотри пример в разделе **Именованные поля кортежей** о способе расширить кортеж из Phobos.

типам в момент создания экземпляра. Это также работает для аргументов шаблонов функций:

```
module storage;

void init(T)(ref T t)
{
    t = T.init;
}

unittest
{
    int i = 10;
    init(i);
    assert(i == 0);
}
```

Если необходимость возникнет, вы можете подогнать ваши классы памяти для соответствия аргументам шаблона. Для этого нет встроенного синтаксиса, так что вы должны прибегнуть к нашим хорошим друзьям `static if` и одноимённому трюку:

```
// У кого-нибудь есть пример получше?
module init;

template init(T)
{
    static if (is(T == immutable) || is(T == const))
        void init(T t) {} // ничего не делать
    else static if (is(T == class))
        void init(ref T t)
        {
            t = new T();
        }
    else
        void init(ref T t)
        {
            t = T.init;
        }
}
```

Свойства выводятся автоматически

В D, функция может иметь следующие свойства:

- Функция может быть обозначена свойством `pure`, что означает, что у неё нет побочных эффектов: значение, которое вы получите обратно — это единственное, что имеет значение.
- Они могут также быть обозначены `@safe`, `@trusted` и `@system`. `@safe` означает, что функция не может испортить память. `@trusted` функция может вызывать функции `@safe`, но даёт никаких других гарантий относительно памяти. И функция `@system` может делать всё, что ей захочется.
- Последнее свойство — это `nothrow`, оно означает, что функция не вызывает никаких исключений.

Так как компилятор получает полный доступ к коду шаблона функции, он может проанализировать его и автоматически вывести свойства для вас. Эта характеристика все еще

совсем новая на момент написания статьи, но, кажется, это работает. Таким образом, **все** ваши шаблоны функций получат некие свойства, когда они создадут экземпляры (эти свойства, конечно, меняются в зависимости от параметров шаблона).

Предусловия in и out

Предусловия `in` и `out` для функции дают полный доступ к параметрам шаблона. Как и для другого параметризованного кода, это означает, что вы можете использовать `static if`, чтобы включать или выключать код, в зависимости от аргументов шаблона.

```
module inoutclauses;
import std.complex, std.math, std.traits;

auto squareRoot(N)(N n) if (isNumeric!N || isComplex!N)
in
{
    // нет необходимости делать это для complex.
    static if (isNumeric!N)
        assert(n > 0);
}
body
{
    return sqrt(n);
}
```

Изменение функций

Этот раздел покажет вам, как использовать шаблоны-обёртки, добавляющие новую функциональность к предопределённым функциям. Показаны более мощные примеры, но они используют шаблоны, которые мы пока не видели. Не забудьте всё-таки бегло познакомиться с ними, когда сможете.

Незакончено! Я хочу поместить здесь несколько небольших шаблонов функций-оберток:

- making a function accept tuples
- making a function have named parameters (sort of)
- making a function have default values for its args
- making a function accept more args than the original
- making a function accept arguments of a different type (that's useful when mapping on tuples, like in section [Tuples As Sequences])

Приём кортежа

```
module acceptingtuple;

template tuplify(alias fun)
{
    auto tuplify(T...)(Tuple!T tup)
    {
        return fun(tup.expand);
    }
}
```

Еще один интересный (и значительно более сложный) пример `juxtapose` (см раздел

Соседские функции).

Отображение (mapping) n диапазонов параллельно

TODO Какое-нибудь объяснение. Показать интересность отображения n диапазонов параллельно.

```
module nmap;
import std.algorithm;
import std.typetuple: allSatisfy;
import acceptingtuple;

// Очень легко сделать, теперь:
auto nmap(alias fun, R...)(R ranges) if (allSatisfy!(isInputRange,R))
{
    return map!(tuplify!fun)(zip(ranges));
}
```

Более сложный вариант: std.algorithm.map принимает больше, чем одну функцию в качестве аргументов шаблона. В этом случае, функции полностью отображаются параллельно на диапазоне, непосредственно используя std.functional.adjoin. Здесь мы можем расширить `nmap`, чтобы он принимал *n* функций тоже параллельно. Есть первая трудность:

```
auto nmap(fun..., R...)(R ranges) if (allSatisfy!(isInputRange, R))
{ ... }   ^^^^^^^^^^^^ Uh?
```

Видите проблему? Нам нужны как переменный список функций, так и переменный список диапазонов. Но кортеж в параметрах шаблона должен быть последним параметром: он может быть только один. Тут приходят на помощь Двухэтапные шаблоны:

```
template nmap(fun...) if (fun.length >= 1)
{
    auto nmap(R...)(R ranges) if (allSatisfy!(isInputRange, R))
    {...}
}
```

Теперь две части хорошо разделены. Что даёт нам окончательный код:

```
module nmap2;
import std.algorithm;
import std.functional : adjoin;
import std.range;
import std.typetuple : allSatisfy;
import acceptingtuple; // tuplify

template nmap(fun...) if (fun.length >= 1)
{
    auto nmap(R...)(R ranges) if (allSatisfy!(isInputRange, R))
    {
        alias adjoin!(staticMap!(tuplify, fun)) _fun;
        return map!(_fun)(zip(ranges));
    }
}
```

И вот версия n-диапазонов для std.algorithm.filter:

```
module nfilter;
import std.algorithm;
import std.range;
import std.typetuple : allSatisfy;

import acceptingtuple; // tuplify
```

```
auto nfilter(alias fun, R...)(R ranges) if (allSatisfy!(isInputRange, R))
{
    return filter!(tuplify!fun)(zip(ranges));
}
```

TODO. Примеры, много примеров.

Шаблоны структур

Синтаксис

Как вы могли догадаться, объявление шаблона структуры делается с помощью установки списка параметров шаблона после имени структуры, подобно этому:

```
module tree1;
import std.array;

struct Tree(T)
{
    T value;
    Tree[] children;

    bool isLeaf() @property { return children.empty;}
    /* Остальные функции дерева: добавление потомков, удаление чего-то, ... */
}
```

> **`Tree[]`** или **`Tree!(T)[]`**? Помните, что в декларации шаблона, имя шаблона относится к текущему экземпляру. Так что внутри `Tree(T)`, имя `Tree` относится к `Tree!T`.

Это дает нам обыкновенное типичное дерево, которое создаётся так же, как и любой другой шаблон:

```
module using_tree;
import tree1;

void main()
{
    auto t0 = Tree!int(0);
    auto t1 = Tree!int(1, [t0,t0]);
    Tree!int[] children = t1.children;
}
```

Как и для всех предыдущих шаблонов, вы можете параметризовать ваши структуры, используя много больше, чем простые типы:

```
module heap1;

bool lessThan(T)(T a, T b) { return a<b;}

struct Heap(Type, alias predicate = lessThan, float reshuffle = 0.5f)
{
    // predicate управляет внутренним сравнением
    // reshuffle имеет дело с внутренней реорганизацией кучи
    Type[] values;
/*...*/
}
```

Шаблоны структур усиленно используются в модулях std.algorithm и std.range для ленивых вычислений, взгляните туда.

Фабричные функции

Теперь, есть одно ограничение: конструкторы структур не активизируют IFTI, как это происходит у шаблонов функций. В предыдущем подразделе, для создания экземпляра Tree(T), я должен явно указать T:

```
auto t0 = Tree!int(0); // Да.

auto t1 = Tree(0); // Ошибка, нет автоматического вычисления, что T - это int.
```

Дело в том, что существует возможность применять Шаблонные конструкторы, и они могут иметь параметры шаблона, отличающиеся от параметров самого глобального шаблона структуры. Но честно говоря, это сильно раздражает, тем более для шаблонов структур со многими аргументами шаблона. Конечно, существует решение: используйте шаблон функции, которая создаёт правильную структуру и возвращает её. Вот пример такой фабричной функции для `Tree`:

```
module tree2;
import tree1;

auto tree(T)(T value, Tree!T[] children = null)
{
    return Tree!(T)(value, children);
}

void main()
{
    auto t0 = tree(0); // Да!
    auto t1 = tree(1, [t0,t0]); // Да!

    static assert(is( typeof(t1) == Tree!int ));

    auto t2 = tree(t0); // Да! typeof(t2) == Tree!(Tree!(int))
}
```

Еще раз, взгляните на модули std.algorithm и std.range, они показывают многочисленные примеры этой идиомы.

Даём доступ ко внутренним параметрам

Как было сказано в разделе Внутренний псевдоним, аргументы шаблона становятся недоступными извне, как только для шаблона будет создан экземпляр. На примере `Tree`, вы могли бы захотеть получить легкий доступ к `T`. Как и для любых других шаблонов, вы можете вывести параметры наружу, определив их псевдоним. Давайте завершим наше определение дерева `Tree`:

```
module tree3;
import std.array;

struct Tree(T)
{
    alias T Type;
    T value;
    Tree[] children;

    bool isLeaf() @property { return children.empty;}
}

void main()
```

```
{
    Tree!string t0 =Tree!string("abc");
    alias typeof(t0) T0;

    static assert(is( T0.Type == string ));
}
```

Шаблонизированные методы

Шаблон структуры является шаблоном, подобным любому другому: вы можете объявить шаблоны внутри, даже шаблоны функций. Это означает, что вы можете получить шаблоны функций-членов.

> **Хромой D.** Окей, поскольку я не ввёл некоторые мощные возможности D, как например, шаблоны mixin и строковые mixins, часть кода мне придётся здесь дублировать. Извините за это, но это условие того, чтобы все именованные куски кода компилировались.

Отображение (mapping) в дереве

Давайте используем шаблонную функцию-член, чтобы придать нашему дереву `Tree` способность отображения. Для диапазона вы можете использовать std.algorithm.map, чтобы применить функцию в свою очередь к каждому элементу, таким образом обеспечивая преобразование диапазона. Тот же процесс можно выполнить для дерева, этим самым сохраняя общую *форму*, но модифицируя элементы. Мы легко могли бы сделать это отдельной функцией, но это раздел о методах (функциях-членах).

Давайте подумаем об этом немного до того, как кодить. `map` должна быть шаблоном функции, который принимает любое имя функции как параметр-псевдоним шаблона (подобно std.algorithm.map,). Давайте назовем этот псевдоним `fun`. Член `value` должен быть преобразован функцией `fun`, это легко сделать. Мы хотим вернуть новое дерево `Tree`, которое получает параметр типа как тип результата функции `fun`. Если `fun` преобразует `A` в `B`, тогда `Tree!A` отображается в `Tree!B`. Тем не менее, поскольку `fun` может быть шаблоном функции, она может не иметь предопределенного возвращаемого типа, который можно было бы получить с помощью std.traits.ReturnType. Мы просто применим её к значению `T` (получив `T.init` и взяв его тип. Так что `B` будет `typeof(fun(T.init))`.

Что насчет потомков? Мы отобразим функцию `fun` на них тоже, и соберём результат в новый массив потомков. У них будет тот же тип: `Tree!(B)`. Если отображаемое дерево `Tree` является листом (т.е., если у него нет детей), процесс прекратится.

Так как это рекурсивный шаблон, мы должны немного помочь компилятору с возвращаемым типом. Вот так:[6]

```
module tree4;
import std.array;

auto tree(T)(T value, Tree!T[] children = null)
{
    return Tree!(T)(value, children);
}
```

6 Отличие от функции map в Phobos в том, что наша версия не является ленивой.

```
struct Tree(T)
{
    alias T Type;
    T value;
    Tree[] children;

    bool isLeaf() @property { return children.empty;}

    Tree!(typeof(fun(T.init))) map(alias fun)()
    {
        alias typeof(fun(T.init)) MappedType;
        MappedType mappedValue = fun(value);
        Tree!(MappedType)[] mappedChildren;
        foreach(child; children) mappedChildren ~= child.map!(fun);
        return tree(mappedValue, mappedChildren);
    }
}
```

Давайте используем это:

```
module using_tree4;
import std.conv;
import tree4;

int addOne(int a) { return a+1;}

void main()
{
    auto t0 = tree(0);
    auto t1 = tree(1, [t0,t0]);
    auto t2 = tree(2, [t1, t0, tree(3)]);

/* t2 - это    2
             / | \
            1  0  3
           / \
          0   0
*/

    // t2 - это Tree!(int)
    static assert(is( t2.Type == int ));

    // Добавляем единицу ко всем значениям

    auto t3 = t2.map!(addOne);

/* t3 - это    3
             / | \
            2  1  4
           / \
          1   1
*/

    assert(t3.value == 3);

    // Преобразуем все значения в строки string
    auto ts = t2.map!(to!string); // мы преобразуем каждую величину в строку;

/* ts - это   "2"
             / | \
          "1""0""3"
           / \
        "0"  "0"
*/
```

```
    assert(is( ts.Type == string ));
    assert(ts.value == "2");
}
```

Свёртка Дерева

Вы можете почувствовать, что функция map не является настоящим методом: она не принимает никаких аргументов. Давайте сделаем другое преобразование на деревьях Tree: их свёртку (folding), которая схлопывает все величины в одну новую. Эквивалент для диапазонов — std.algorithm.reduce, который схлопывает целый диапазон (линейный) в одну величину, будь это числовая величина, другой диапазон или что там у вас есть.

Для дерева, свертка может, например, генерировать все величины в пред-порядке или после-порядке, вычислять высоту дерева, количество листов... Как и в случае диапазонов, свертка является чрезвычайно разносторонней функцией. Фактически, её можно использовать для преобразования дерева `Tree` в массив или в другое дерево. Мы сделаем это.

Взяв вдохновение из `reduce`, нам нужно начальное значение (seed) и две свёртывающие функции. Первая, `ifLeaf`, будет вызываться на бездетных узлах, чтобы возвращать `ifLeaf(value, seed)` при выполнении `fold`. Второй, `ifBranch`, будет вызываться для узлов с детьми. В этом случае, мы сначала применяем функцию `fold` на всех потомков, а затем возвращаем `ifBranch(value, foldedChildren)`. В некоторых простых случаях мы можем использовать ту же самую функцию, что и для листьев, следовательно, для `ifBranch` будет вариант по-умолчанию. Вот код:[7]

```
module tree5;
import std.array;

Tree!(T) tree(T)(T value, Tree!T[] children = null)
{
    return Tree!(T)(value, children);
}

struct Tree(T)
{
    alias T Type;
    T value;
    Tree[] children;

    bool isLeaf() @property { return children.empty;}

    typeof(ifLeaf(T.init, S.init))
    fold(alias ifLeaf, alias ifBranch = ifLeaf, S)(S seed)
    {
        if (isLeaf)
        {
            return ifLeaf(value, seed);
        }
        else
        {
            typeof(Tree.init.fold!(ifLeaf, ifBranch)(seed))[] foldedChildren;
            foreach(child; children)
```

7 Технически, std.algorithm.reduce — это левая свёртка, в то время как то, что показано здесь — правая свёртка. В нашем случае различие не существенно.

```
                foldedChildren ~= child.fold!(ifLeaf, ifBranch)(seed);
            return ifBranch(value, foldedChildren);
        }
    }
}
```

Давайте поиграем с ним немного. Сначала, мы хотим просуммировать все значения `value` дерева. Для листьев мы просто возвращаем величину узла `value` плюс начальное значение `seed`. Для ветвей нам даются величина и массив, содержащий суммы величин для всех детей. Нам нужно просуммировать значения этого массива, добавить его к значению узла и вернуть результат. В этом случае, мы не применяем `seed`.

```
module summingtree;
import std.algorithm;
import tree5;

typeof(T.init + S.init)
sumLeaf(T, S)(T value, S seed)
{
    return value + seed;
}

T sumBranch(T)(T value, T[] summedChildren)
{
    return value + reduce!"a+b"(summedChildren);
}

import std.stdio;

void main()
{
    auto t0 = tree(0);
    auto t1 = tree(1, [t0,t0]);
    auto t2 = tree(2, [t1, t0, tree(3)]);

    int sum = t2.fold!(sumLeaf, sumBranch)(0);
    assert(sum == 2 + (1 + 0 + 0) + (0) + (3));
}
```

В том же семействе, но немного интереснее, получить все величины для итераций в порядке вглубь: принимаем узел дерева, возвращаем массив, содержащий локальную величину и, затем, величины для всех узлов, рекурсивно.

```
module inordertree;
import std.algorithm;
import tree5;

T[] inOrderL(T, S)(T value, S seed)
{
    return [value] ~ seed;
}

T[] inOrderB(T)(T value, T[][] inOrderChildren)
{
    return [value] ~ reduce!"a~b"(inOrderChildren);
}

void main()
{
    auto t0 = tree(0);
    auto t1 = tree(1, [t0,t0]);
```

```
    auto t2 = tree(2, [t1, t0, tree(3)]);

    int[] seed; // пустой массив
    auto inOrder = t2.fold!(inOrderL, inOrderB)(seed);
    assert(inOrder == [2, 1, 0, 0, 0, 3]);
}
```

И, как последний пример использования, почему бы не построить дерево?

TODO. Просто написать это.

Шаблоны конструкторов

Конструкторы структур являются методами, так что их тоже можно шаблонизировать. Они не обязаны иметь такие же параметры шаблона, как у определения структуры:

```
module templatedconstructors;

struct S(T)
{
    this(U)(U u) { /*...*/ }
}

void main()
{
    auto s = S!string(1); // T  - это string, U - это int.
}
```

Как вы можете видеть, IFTI работает для конструкторов. U автоматически выводится, на всё же в этом случае вы должны указать T. Тем не менее, этот пример сильно ограничен: вы не можете иметь никаких величин типа U в структуре, поскольку U не существует за пределами конструктора. Немного более полезным было бы собрать псевдоним (функции, например) и использовать его для инициализации структуры. Если он используется только для инициализации, он может быть отвергнут впоследствии. Но зато, IFTI не активизируется псевдонимом...

Наиболее интересное использование, которое я видел — это выполнить преобразование во время создания структуры:

```
module holder;
import std.conv;

struct Holder(Type)
{
    Type value;

    this(AnotherType)(AnotherType _value)
    {
        value = to!Type(_value);
    }
}

void main()
{
    Holder!int h = Holder!int(3.14);
    assert(h.value == 3);
}
```

Таким образом, `Holder!int` может быть создан с любой величиной, но если преобразование возможно, оно всегда будет содержать `int`.

Внутренние структуры

Вы можете создавать и возвращать внутренние структуры и использовать локальные параметры шаблона в их определении. Мы могли бы использовать фабричную функцию для `Heap` (куча), вроде такой:

```
module heap2;

auto heap(alias predicate, Type)(Type[] values)
{
    struct Heap
    {
        Type[] values;
        this(Type[] _values)
        {
            /* некоторый код инициализации значений, использующий predicate */
        }
        /* больше кода, работающего с кучей */
    }

    return Heap(values); // alias predicate тут скрыт
}
```

В этом случае, структура `Heap` изолирована внутри функции `heap` и использует псевдоним предиката внутри, но сама она не является шаблоном структуры. Я не использовал дерево `Tree` в качестве примера, поскольку с рекурсивными типами это становится сложным.

Между прочим, как ни странно, хотя вы всё же не можете объявлять «чистые» шаблоны внутри функций, вы можете объявить шаблоны структур. Помните функцию `adder` в разделе Анонимных функций? Она не должна быть шаблонизирована с одним типом для каждого аргумента, а большую часть времени, когда вы добавляете числа, они имеют более или менее один и тот же тип. Но что насчет функции, которая преобразует свои аргументы в строки `string` перед их сцеплением?

```
module anonymous_error;

auto concatenate(A)(A a)
{
    /* !! Недопустимый код D !! */
    return (B)(B b) { return to!string(a) ~ to!string(b);};
}
```

Предыдущий пример не является допустимым D-кодом (помните **Мантру**). Конечно, решение существует: просто возвращайте структуру с шаблонным методом, в данном случае оператором `opCall`:

```
module innerconcatenate;
import std.conv;

auto concatenate(A)(A a)
{
    struct Concatenator
    {
        A a;

        auto opCall(B)(B b) @trusted
        {
```

```
            return to!string(a) ~ to!string(b);
        }
    }

    Concatenator c;
    c.a = a; // Инициализируем таким образом, чтобы не активировать opCall()

    return c;
}

void main()
{
    auto c = concatenate(3.14);
    auto cc = c("abc");
    assert(cc == "3.14abc");
}
```

Смотрите раздел Перегрузка операторов

Как насчет шаблонных внутренних структур в шаблонах структур? Это вполне допустимо:

```
module templatedinner;

struct Outer(O)
{
    O o;

    struct Inner(I)
    {
        O o;
        I i;
    }

    auto inner(I)(I i) { return Inner!(I)(o,i);}
}

auto outer(O)(O o) { return Outer!(O)(o);}

void main()
{
    auto o = outer(1); // o — это Outer!int;
    auto i = o.inner("abc"); // Outer.Outer!(int).Inner.Inner!(string)
}
```

Шаблон параметров This

Незакончено! Было бы неплохо привести здесь какой-нибудь пример.

Внутри шаблона структуры или класса, есть другой тип параметра шаблона: шаблон параметра, объявляемого идентификатором **this**. Идентификатор затем получает тип ссылки **this**. Это полезно главным образом для использования в двух случаях:

- шаблоны mixin, где вы не знаете заблаговременно объемлющий тип. Пожалуйста, посмотрите в этом разделе несколько примеров.
- для того, чтобы определять, как квалифицирован тип ссылки **this** (**const**, **immutable**, shared, **inout** или unqualified).

Пример: a Concat / Flatten Range

Мы используем то, что мы узнали о шаблонах структур, чтобы создавать ленивый диапазон, который выравнивает многоуровневый диапазон диапазонов в линейные диапазоны. Помните выравнивающую функцию `flatten` из раздела Сглаженные массивы и диапазоны? Она работала довольно хорошо, но была энергичной, а не ленивой: при получении бесконечного диапазона (цикла, например), она захлебнётся. Здесь мы сделаем ленивый выравниватель.

Если вы посмотрите на диапазоны, определенные в std.range, вы увидите, что большинство их (если не все) являются структурами. Это основной способ получать ленивость в D: структура содержит состояние итерации и предоставляет основные примитивы диапазона. По крайней мере для *входного диапазона* (input range) — простейшего вида диапазона — тип должен иметь следующие члены, будь то свойства, методы или объявления констант:

`front` (спереди)

возвращает первый элемент диапазона.

`popFront`

отбрасывает первый элемент и сдвигает диапазон на один шаг.

`empty` (пусто)

возвращает **`true`**, если диапазон больше не имеет элементов, в противном случае **`false`**.

С этой простой основой, можно разработать мощные алгоритмы, действующие над диапазонами. D определяет более усовершенствованные концепции диапазона, добавляя другие ограничения. *Лидирующий диапазон* (forward range) добавляет член `save`, который используется для сохранения внутреннего состояния диапазона и предоставляет алгоритм, позволяющий начать снова из сохранённой позиции. *Двунаправленный диапазон* (bidirectional range) также имеет примитивы `back` (сзади) и `popBack` для доступа к концу диапазона, и так далее.

Здесь мы начнём с создания простого входного диапазона, который принимает диапазон диапазонов и выполняет итерации по внутренним элементам. Давайте начнём с самых основ:

```
module flatten2;

import std.range;
import rank2;

struct Flatten(Range)
{
    Range range;
    /*...*/
}

auto flatten(Range)(Range range)
{
```

```
    static if (rank!Range == 0)
        static assert(0, "функции flatten требуется диапазон.");
    else static if (rank!Range == 1)
        return range;
    else
        return Flatten!(Range)(range);
}
```

Итак, мы имеем шаблон структуры и связанную с ним фабричную функцию. Она не имеет смысла для создания экземпляров `Flatten` с любым старым типом, так что происходит проверка `Range` на то, что он является диапазоном, с использованием шаблона `rank`, который мы видели в подразделе Ранг для диапазонов. Мы пока не рассматривали ограничения шаблона (они описаны в разделе Ограничения), но они хорошо подошли бы здесь.

Диапазон диапазонов можно представить как-то так:

```
[ subRange1[elem11, elem12,...]

, subRange2[elem21, elem22,...]

, ... ]
```

Мы хотим, чтобы `Flatten` возвращала элементы в таком порядке: `elem11`, `elem12`,... `elem21`, `elem22`,... Заметьте, что для диапазонов рангом выше чем 2, эти элементы `elemxy` сами являются диапазонами. В любой данный момент времени, `Flatten` работает с под-диапазоном, выполняя итерации на его элементах, и отбрасывая его, если он пуст. Итерации прекратятся, когда израсходован последний поддиапазон, т.е. когда пуст сам диапазон `range`.

```
1  module flatten3;
2  import std.range;
3
4  import rank2;
5
6  struct Flatten(Range)
7  {
8      alias ElementType!Range    SubRange;
9      alias ElementType!SubRange Element;
10
11     Range range;
12     SubRange subRange;
13
14     this(Range _range) {
15         range = _range;
16         if (!range.empty) subRange = range.front;
17     }
18
19     Element front() { return subRange.front;}
20
21     bool empty() { return range.empty;}
22
23     void popFront() {
24         if (!subRange.empty) subRange.popFront;
25         while(subRange.empty && !range.empty) {
26             range.popFront;
27             if (!range.empty) subRange = range.front;
28         }
29     }
```

```
30 }
```

- Я немного жульничаю со стандартом расстановки фигурных скобок в D, поскольку он безоглядно съедает вертикальное пространство.
- Мы начинаем в строке 8 и 9 с определения некоторых новых типов, используемых методами. Они не строго необходимы, но упрощают понимание кода и позволяют вывести их во внешний мир, если они нужны.
- Для инициализации структуры теперь необходим конструктор.
- `front` возвращает элемент поддиапазона.

Но тогда это работает только для диапазона диапазонов (ранг < 2). Мы хотим чего-то, что выравнивает диапазоны любого ранга вплоть до линейного диапазона. Это легко сделать, мы просто добавляем рекурсию в фабричную функцию:

```
module flatten4;
import rank2;
public import flatten3;

auto flatten(Range)(Range range)
{
    static if      (rank!Range == 0)
        static assert(0, "функции flatten требуется диапазон.");
    else static if (rank!Range == 1)
        return range;
    else static if (rank!Range == 2)
        return Flatten!(Range)(range);
    else         // ранг 3 или выше
        return flatten(Flatten!(Range)(range));
}
```

И, тестирование:

```
module using_flatten4;
import std.algorithm;
import std.range;
import std.string;
import rank2;
import flatten4;

void main()
{
    auto rank3 = [[[0,1,2],[3,4,5],[6] ]
                 ,[[7],[],[8,9],[10,11]]
                 ,[[],[12]             ]
                 ,[[13]                ]];

    auto rank1 = flatten(rank3);
    assert(rank!(typeof(rank1)) == 1); // Да, это линейный диапазон
    assert(equal( rank1, [0,1,2,3,4,5,6,7,8,9,10,11,12,13] ));

    auto stillRank1 = flatten(rank1);
    assert(equal( stillRank1, rank1 )); // Нет необходимости настаивать

    auto text =
   "Sing, O goddess, the anger of Achilles son of Peleus,
    that brought countless ills upon the Achaeans.
    Many a brave soul did it send hurrying down to Hades,
    and many a hero did it yield a prey to dogs and vultures,
```

```
    for so were the counsels of Jove fulfilled
    from the day on which the son of Atreus, king of men,
    and great Achilles, first fell out with one another.";

    auto lines = text.splitLines;   // массив строк string
    string[][] words;
    foreach(line; lines) words ~= array(splitter(line, ' '));
    assert( rank!(typeof(words)) == 3); // диапазон диапазонов строк
                                        // диапазон диапазонов массива символов
    auto flat = flatten(words);

    assert(equal(take(flat, 50),
            "Sing,Ogoddess,theangerofAchillessonofPeleus,thatbr"));
}
```

Вот. Это работает, и мы использовали шаблон структуры, static if, внутренний псевдоним, фабричные функции и IFTI.

Шаблоны классов

Данный раздел нуждается в вас! Я не являюсь ООП-программистом, и я не привык создавать интересные иерархии. Если у того, кто читает это, есть пример шаблонов классов, которые можно было бы использовать на протяжении всего раздела, то я в игре.

Синтаксис

Здесь нет сюрприза, просто поставьте список параметров шаблона между классом и необязательным указанием на наследование:

```
module classsyntax1;

class Base {}
interface Interface1 {}
interface Interface2 {}

class MyClass(Type, alias fun, bool b = false)
    : Base, Interface1, Interface2
{ /*...*/ }
```

Что ещё забавнее — вы можете параметризовать наследование: различные параметры шаблона определяются перед списком базовых классов, и вы можете их использовать здесь:

```
module classsyntax2;

class Base(T) {}
interface Interface1 {}
interface Interface2(alias fun, bool b) {}

class MyClass(Type, alias fun, bool b = false)
    : Base!(Type), Interface1, Interface2!(fun, b)
{ /*...*/ }
```

Шаблоны Интерфейса? Да, вы можете. Смотрите раздел Другие Шаблоны.

Это открывает интересные перспективы, где то, что класс наследует, определяется аргументами его шаблона (так, `Base` может быть множеством различных классов, или даже интерфейсов, в зависимости от `Type`). В самом деле, посмотрите на это:

```
module classsyntax3;
```

```
enum WhatBase { Object, Interface, BaseClass }

template Base(WhatBase whatBase = WhatBase.Object)
{
    static if (is(T == WhatBase.Object))
        alias Object Base; // MyClass напрямую наследуется от Object
    else static if(is(T == WhatBase.Interface))
        alias TheInterface Base;
    else
        alias TheBase Base;
}

class MyClass(Type) : Base!Type {}
```

При этом, `MyClass` может наследовать от `Object`, корня иерархии классов D, от интерфейса, или от другого класса. Очевидно, шаблон-диспетчер можно сделать поизящнее. Со вторым параметром шаблона базовый класс сам может параметризоваться, и так далее.

Что этот синтаксис , однако, *НЕ может* сделать — изменить количество интерфейсов во время компиляции.[8] Это сложно сказать: "с *этим* аргументом, `MyClass` унаследуется от `I`, `J` и `K`, а с *тем* аргументом он унаследуется только от `L`." You'd need the previous interfaces to all participate in the action, to all be templates and such. Если нужные интерфейсы — все предопределены и не шаблонизированы, вам понадобятся обёртывающие (wrapping) шаблоны. Это — боль. Тем не менее, можно использовать тип кортежей, для существенного упрощения.

Шаблоны методов

Методы объектов не являются чем-то большим, чем делегаты со ссылкой `this` на локальный контекст. Как мы уже видели, методы структур тоже могут быть шаблонами.

> **Незакончено!** Мне нужно написать что-то о перегрузке методов с шаблонами. Также, мне нужно найти некий интересный пример метода. Люди, я не пользуюсь классами!

Условие `invariant`

В том же семействе, что и предусловия **`in`**/**`out`** для функций (см. раздел Предусловия in и out), предусловие **`invariant`** шаблона класса имеет доступ к параметру шаблона. Вы не можете сделать так, чтобы он полностью исчез, но вы можете заставить его быть пустым с помощью оператора **`static if`**.

```
module classinvariant;

class MyClass(T, U, V)
{
    /*...*/

    invariant ()
    {
        static if (is(T == U))
        {
            /* код инварианта */
```

8 Кроме, может быть, случаев, когда шаблон интерфейса будет пустым для определённых параметров, таким образом, в сущности, исчезая из списка.

```
        }
        else
        { } /* пустой инвариант */
    }
}
```

Внутренние классы

Здесь тот же принцип, как и для структур. Вы можете определить внутренние классы, используя параметры шаблона. Вы можете даже дать им шаблоны метода, который использует другие аргументы шаблона. Тут действительно нет отличий от внутренних структур.

Анонимные классы

В D вы можете возвращать анонимные классы непосредственно из функции или метода. Могут они быть шаблонными? Хорошо, они не могут быть шаблонами класса, это не имеет смысла. Но вы можете возвращать анонимные классы с шаблонными методами, если вам действительно нужно.

```
module anonymousclass;

// сохраняет функцию и значение, возвращаемое по-умолчанию.
auto acceptor(alias fun, D)(D defaultValue)
{
    return new class
    {
        this() {}

        auto opCall(T)(T t)
        {
            static if (__traits(compiles, fun(T.init)))
                return fun(t);
            else
                return defaultValue;
        }
    };
}

unittest
{
    int add1(int i) { return i+1;}
    auto accept = acceptor!(add1)(-1);

    auto test1 = accept(10);
    assert(test1 == 11);

    auto test2 = accept("abc");
    assert(test2 == -1); // значение по-умолчанию
}
```

По поводу выражения `__traits(compiles,...)`, смотрите здесь онлайн, и в разделе __traits в этом документе.

Параметризованные базовые классы

Вы можете использовать параметр шаблона непосредственно как базовый класс:

```
module parametrizedbaseclass;

interface ISerializable
```

```
{
    size_t serialize() @property;
}

class Serializable(T) : T, ISerializable
{
    size_t serialize() @property
    {
        return this.tohash;
    }
}
```

В этом примере `Serializable!SomeClass` может действовать как `SomeClass`. Это не отличается от того, что вы могли бы сделать с нормальными классами, за исключением того, что идиома теперь абстрагирована в базовом классе: вы пишете шаблон один раз, он может быть использован в любом классе.

Если у вас есть другие интерфейсы, подобные этому, вы можете делать эти свойства вложенными:

```
auto wrapped = new Serializable!(Iterable!(SomeClass))(/*...*/);
```

Конечно, базовый класс и интерфейс сами могут быть параметризованы:

```
module serializetemplate;

enum SerializationPolicy { policy1, policy2 }

interface ISerializable
(SerializationPolicy policy = SerializationPolicy.policy1)
{
    static if (is(policy == SerializationPolicy.policy1))
        void serialize() { /*...*/ }
    else
        void serialize() { /*...*/ }
}

class Serializable(T, Policy) : T, ISerializable!Policy
{
    /*...*/
}
```

В D вы можете также получить такой эффект с помощью объявления `alias X this;` в вашем классе или структуре. Вы должны также бегло познакомиться с Шаблонами mixin и Обёртывающими шаблонами для других идиом, которые строят примерно для тех же целей.

Добавление функциональности через наследование

Вот пример, взятый у Timon Gehr. Данный базовый класс `Cell` (ячейка) с приватным полем и методами `get/set`, даёт возможность обеспечить дополнительную функциональность через наследование. Определяя шаблоны для подклассов `Cell`, каждый с новой возможностью (удвоение величины, логирование величины, и т.п.), можно выбирать эти новые действия через наследование:

```
/**
 * Timon Gehr timon.gehr@gmx.ch via puremagic.com
 */
module inheritanceexample;

import std.stdio;
```

```
abstract class Cell(T)
{
    abstract void set(T value);
    abstract const(T) get();
private:
    T field;
}

class AddSetter(C: Cell!T,T): C
{
    override void set(T value){field = value;}
}

class AddGetter(C: Cell!T,T): C
{
    override const(T) get(){return field;}
}

class DoubleCell(C: Cell!T,T): C
{
    override void set(T value){super.set(2*value);}
}

class OneUpCell(C: Cell!T,T): C
{
    override void set(T value){super.set(value+1);}
}

class SetterLogger(C:Cell!T,T): C
{
    override void set(T value)
    {
        super.set(value);
        writeln("ячейке присвоено '",value,"'!");
    }
}

class GetterLogger(C:Cell!T,T): C
{
    override const(T) get()
    {
        auto value = super.get();
        writeln("'",value,"' было извлечено!");
        return value;
    }
}

// ConcreteCell имеет 'настоящий' getter и 'настоящий' setter
class ConcreteCell(T): AddGetter!(AddSetter!(Cell!T)){}

// OneUpDoubleSetter имеет setter, который добавляет 1 к сохраняемому значению
// и удваивает его
class OneUpDoubleSetter(T): OneUpCell!(DoubleCell!(AddSetter!(Cell!T))){}

// doubleOneUpSetter имеет setter, который удваивает значение и добавляет 1
class DoubleOneUpSetter(T): DoubleCell!(OneUpCell!(AddSetter!(Cell!T))){}

void main()
{
    Cell!string x;
    x = new ConcreteCell!string;
    x.set("hello");
    writeln(x.get());
```

```
    Cell!int y;
    y = new SetterLogger!(ConcreteCell!int);
    y.set(123); // напечатает: "ячейке присвоено '123'!

    y = new GetterLogger!(DoubleCell!(ConcreteCell!int));
    y.set(1234);
    y.get(); // напечатает: "'2468' было извлечено!"

    y = new AddGetter!(OneUpDoubleSetter!int);
    y.set(100);
    writeln(y.get()); // напечатает "202"
    y = new AddGetter!(DoubleOneUpSetter!int);
    y.set(100);
    writeln(y.get()); // напечатает "201"
}
```

Для тех, кто знает язык программирования Scala, это немного напомнит *traits*, которые в Scala являются своеобразными классами-заглушками, которые могут быть добавлены к другим классам, чтобы обеспечить новые функции. В D, за исключением вышеуказанной возможности, вы также можете использовать Шаблоны Mixin и Строки mixin.

Странно рекурсивный шаблон

```
module crtp;

class Base(Child) { /*...*/ }

class Derived : Base!Derived { /*...*/ }
```

Стоп, что это значит? `Base` должен легко пониматься. Но что насчет `Derived`? Он сообщает, что наследуется от другого класса, который шаблонизирован... самим `Derived`? Но `Derived` не определён на этом этапе! Или как? Да, это работает. Это называется CRTP, что означает *Странно рекурсивный шаблон* (Curiously Recurring Template Pattern, смотрите Википедию об этом. *К сожалению, русского варианта этого документа не существует, но есть украинский — прим. пер.*). Но в чём может быть интерес к такому трюку?

Как вы можете видеть в документе из Википедии, он используется либо для получения своего рода связывания времени компиляции или, чтобы вводить код в ваш производный класс. Что касается последнего, D предлагает Шаблоны mixin, на которые вам следует взглянуть. CRTP происходит из C++, где возможно множественное наследование. Я боюсь, что в D это не так интересно. Не стесняйтесь, докажите мне, что я не прав, я охотно изменю этот раздел.

Пример: Автоматическая динамическая диспетчеризация

Этот пример от Andrej Mitrovic. Вы можете найти его также на D wiki: http://wiki.dlang.org/Dispatching_an_object_based_on_its_dynamic_type.

Иногда вам может понадобиться вызывать функцию с объектом, где функция принимает только определенный тип производного объекта. В таком случае вам нужно проверить, что динамический тип объекта является определённым типом, прежде чем пытаться привести его к этому типу, и передавать в функцию. Например:

```
class A { }
class B : A { }
```

```
class C : A { }

void foo(B b) { }
void foo(C c) { }

void main()
{
    A b = new B;
    A c = new C;

    foo(b); //не принимается, объект должен быть преобразован либо в B, либо в C
    foo(c); //то же
}
```

Поскольку объекты `b` и `c` статического типа `A` могут быть любого динамического типа, производного от класса `A`, пользователь должен пытаться приводить объекты к этим типам, чтобы узнать, может ли он вызвать функцию `Foo` с такими объектами. Код, осуществляющий механизм диспетчеризации вручную, как правило, будет выглядеть следующим образом:

```
import std.stdio;

class A { }
class B : A { }
class C : A { }

void foo(B b) { writeln("called foo(B b);"); } //вызван foo(B b);
void foo(C c) { writeln("called foo(C c);"); } //вызван foo(C c);

void main()
{
    A b = new B;
    A c = new C;

    if (auto obj = cast(C)b)
        foo(obj);
    else if (auto obj = cast(B)b)
        foo(obj);

    if (auto obj = cast(C)c)
        foo(obj);
    else if (auto obj = cast(B)b)
        foo(obj);
}
```

Тем не менее это — одновременно неэффективно и трудоёмко для ввода, множество кода добавляется копированием-вставкой, и подвержено появлению ошибок, поскольку добавление нового листового класса означает, что пользователь должен проверить и отредактировать код диспетчеризации. Альтернативный метод должен использовать структуру `classinfo` (извлекаемую с помощью функции `typeid()`), связанную с каждым экземпляром объекта, и, сравнивать её со всеми существующими структурами `classinfo`. Как только у нас появится совпадение, мы можем благополучно выполнить статическое приведение типа объекта и передать его в функцию.

С помощью шаблонов мы можем автоматизировать весь процесс. Единственная информация, в которой нуждается такой шаблон — это список листовых классов, так что он может построить класс дерева, чтобы правильно диспетчерировать объект. Полная реализация представлена здесь:

```d
import std.stdio;
import std.typetuple;
import std.traits;
import std.string;

template ClassTreeImpl(Leaves...)
{
    static if (Leaves.length > 1)
    {
        alias TypeTuple!(Leaves[0], BaseClassesTuple!(Leaves[0]),
                         ClassTreeImpl!(Leaves[1..$])) ClassTreeImpl;
    }
    else
    static if (Leaves.length == 1)
    {
        alias TypeTuple!(Leaves[0], BaseClassesTuple!(Leaves[0])) ClassTreeImpl;
    }
    else
    {
        alias TypeTuple!() ClassTreeImpl;
    }
}

template ClassTree(Leaves...)
{
    alias DerivedToFront!(NoDuplicates!(ClassTreeImpl!(Leaves))) ClassTree;
}

template AutoDispatch(Leaves...)
{
    void AutoDispatch(alias func, Args...)(Args args)
        if (Args.length >= 1 && is(Args[0] == class))
    {
        auto objInfo = typeid(args[0]);
        foreach (Base; ClassTree!Leaves)
        {
            if (objInfo == Base.classinfo)
            {
                // избежать ошибок времени компиляции из-за разворачивания
                // статического foreach
                static if (__traits(compiles, {
                    return func(cast(Base)(cast(void*)args[0]), args[1..$]); }() ))
                {
                    return func(cast(Base)(cast(void*)args[0]), args[1..$]);
                }
            }
        }

        string[] arguments;
        arguments ~= objInfo.toString();
        foreach (arg; args[1..$])
            arguments ~= typeof(arg).stringof;

        assert(0, format("function '%s' is not callable with types '(%s)'",
                         __traits(identifier, func), arguments.join(", ")));
                         //функция func не вызывается с типами arguments
    }
}

class A { }
class B : A { }
class C : B { }
class D : B { }
```

```
void foo(C c, int x) { writefln("foo(C) : received %s", x); }
void foo(D d, int x, int y) { writefln("foo(D) : received %s %s", x, y); }

void main()
{
    A c = new C;
    A d = new D;
    A a = new A;

    alias AutoDispatch!(C, D) callFunc;

    callFunc!foo(c, 1);    // ok
    callFunc!foo(d, 2, 3); // ok
    callFunc!foo(a, 3);    // срабатывает assert во время выполнения
}
```

`AutoDispatch` принимает список листовых классов, и извлекает класс дерева, используя traits (признаки) из модуля std.traits. Внутреннюю функцию `AutoDispatch` можно затем использовать, чтобы посылать объекты и любые дополнительные аргументы в функцию. Реализация работает только для одно-объектных аргументов, но более общее решение, которое диспетчеризует несколько объектов, можно реализовать.

Другие шаблоны?

Ещё два составных типа в D могут быть шаблонизированы с использованием того же синтаксиса: интерфейсы и объединения.

Шаблоны интерфейсов

Синтаксис в точности такой, каким вы могли бы его себе представить:

```
module interfacesyntax;

interface Interf(T)
{
    T foo(T);
    T[] bar(T,int);
}
```

Шаблоны интерфейсов бывают иногда полезными, но так как они выглядят, как шаблоны класса, я не описываю их. Так же, как и до этого, помните Мантру: шаблоны интерфейсов **не** являются интерфейсами, это — планы для производства интерфейсов в любой момент.

Шаблоны объединений

Вот синтаксис, никаких сюрпризов:

```
module unionsyntax;

union Union(A,B,C) { A a; B b; C c;}
```

Шаблоны объединений кажутся хорошей идеей, но честно говоря, я ни разу не видел ни одного. Любой читатель этого документа, пожалуйста, дайте мне пример, если вы о нём знаете.

Шаблоны Перечислений?

Удивительно, но перечисления не имеют предыдущего упрощённого синтаксиса. Для

того, чтобы объявить шаблон перечисления, используйте шаблон с одноимённым трюком (eponymous):

```
module enumsyntax;

template Enum(T)
{
    enum Enum : T { A, B, C}
}
```

Некоторые более сложные вопросы

В предыдущей части мы увидели то, что все должны знать о шаблонах D. Но фактически, существует гораздо больше. То, что рассмотрено здесь, не обязательно сложнее, но, вероятно, используется несколько реже. По мере развития этого документа некоторые темы могут перетекать из Основ в Сложные вопросы и наоборот.

Ограничения

Ограничения Шаблона являются способом блокировать создание экземпляра шаблона, если не выполнено некоторое условие. Разрешается любое условие, которое может быть определено на этапе компиляции, что делает ограничения расширением (надмножеством) специализаций шаблона. Таким образом, их использование стало быстро расти, как только они были введены в язык, и, если рассматривать Phobos в качестве индикатора, специализации шаблонов наоборот становятся менее распространёнными.

Синтаксис

Чтобы получить ограничение, расположите условие **if** непосредственно после списка параметров шаблона и перед закрытой областью:

```
template templateName(T,U,V) if (некотороеУсловие для T, U или V)
{
   ...
}
```

Когда компилятор пытается создать экземпляр шаблона, сначала проверяется ограничение. Если его значение равно **false**, объявление шаблона не является частью рассматриваемого множества. Таким образом, с помощью ограничений, вы можете сохранять или отбрасывать шаблоны. Здесь выражения **is** — ваши друзья, они позволяют вам получить во время компиляции интроспекцию по типам. Смотрите приложение Выражение is, чтобы получить для ускоренный курс о нём.

Вы можете объявлять множество шаблонов с одним и тем же именем, и с отличающимися ограничениями (фактически, это очень часто используемый случай для ограничений). В зависимости от активного ограничения, некоторые или все из них будут рассматриваться компилятором.

```
module constrained;

template Constrained(T)
    if (is(T : int)) { /*...*/ } // #1
template Constrained(T)
    if (is(T : string)) { /*...*/ } // #2
template Constrained(T,U)
    if (is(T : int) && !is(U : float)) { /*...*/ } // #3
template Constrained(T,U)
    if (is(T : int) && is(U : float)) { /*...*/ } // #4

alias Constrained!(int) C1;       // #1
// alias Constrained!(string) C2; // Ошибка, нет подходящего объявления (string)
alias Constrained!(int,string) C3;// рассматриваются #3 и #4, но #4 отбрасывается.
                                  // Так что это #3.
```

Не понятно, почему для `string` не подходит вариант `#2` — прим.пер.

Таким же синтаксис будет и для специальных случаев шаблонов, рассмотренных в разделах Шаблоны функций, Шаблоны структур, Шаблоны классов и Другие шаблоны . Единственный не сразу очевидный момент у шаблонов классов, там может возникнуть вопрос: где разместить ограничение, до или после списка наследования? Ответ: до.

```
module constraintsyntax;

T theFunction(T)(T argument)
    if (is(T : int) || is(T : double))
{ return argument; }

struct TheStruct(T)
    if (is(T : int) || is(T : double))
{ /*...*/ }

class TheClass(T)
    if (is(T : int) || is(T : double))
    : BaseClass!T, Interface1
{ /*...*/ }
```

Только помните, что когда вы записываете ограничения, они создаются во время компиляции. Для `theFunction`, аргумент `argument` не известен во время компиляции, только его тип, `T`. Не используйте `argument` в вашем ограничении. Если вам нужна величина типа `T`, используйте `T.init`. Например:

```
module ctvalue;

auto callTwice(alias fun, T)(T arg)
    // Возможно ли выполнить fun(fun(от некоторого T))?
    if (is(typeof({
                        fun(fun(T.init));
                    }())))
{
    return fun(fun(arg));
}
```

Использование ограничений

Ограничения приходят из той же идеи, как концепции `concept` для C ++ 0x, э-э ..., концепция, которую проще определить, понять и, как показано в D, реализовать. Идея заключается в том, чтобы определить набор условий для типа, которые он должен соблюдать, чтобы быть представителем «концепции», и проверить их до создания экземпляра.

Бегло познакомимся с живым примером ограничений: Диапазонами (`ranges`). Мы их упоминали в разделе Сглаженные массивы и диапазоны.

std.range (*сейчас это модуль std.range.primitives — прим. пер.*) определяет набор шаблонов функций, которые проверяют соответствие различным видам диапазонов, с именами `isInputRange`, `isForwardRange`... Я называю эти `bool`-шаблоны Шаблонами-предикатами и говорю о них в этом разделе. Использовать их очень просто:

```
module rangewrapper;
import std.range;

struct RangeWrapper(Range)
    // Range подчиняется «концепции» входного диапазона?
```

```
    if (isInputRange!Range)
{
   /* Здесь мы знаем, что Range имеет по крайней мере три функции-члена:
     .front(), .popFront() и .empty(). Мы благополучно можем их использовать.*/
}

// В фабричной функции тоже.
auto rangeWrapper(Range)(Range range) if (isInputRange!Range)
{
    return RangeWrapper!(Range)(range);
}
```

Фактически, это немного похоже на интерфейс времени компиляции или на «утиную типизацию» времени компиляции: мы не заботимся о «типе» параметра `Range`: это может быть структура или класс, насколько нам известно. Важно то, что он подчиняется концепции входного диапазона.

Хорошие новости в том, что компилятор пожалуется, если он не сможет создать экземпляр шаблона из-за не соблюдающихся ограничений. Таким образом, это даст удобные сообщения об ошибках (хотя, вряд-ли вам нужно именно это).

Пределы ограничений

Основная проблема в том, что по сравнению со специализациями шаблонов, вы не сможете сделать так:

```
module constraints_error;

template Temp(T)
if (is(T:int))   // #1, специально для int
{ /*...*/ }

template Temp(T) // #2, общий случай
{ /*...*/ }

alias Temp!int TI;// Ошибка!
```

Почему ошибка? Потому что компилятор обнаруживает, что экземпляр может быть создан как для `int`-специализированной, так и для общей версии. Он не может решить, какой из них вы имеете в виду и, совершенно правильно, ничего не делает. Вы скажете, что это не проблема, мы просто добавим ограничение к общей версии:

```
template Temp(T) if (is(T:int)) // #1
{ ... } // специально для int

template Temp(T) if (!is(T:int))// #2
{ ... } // общий случай

Temp!int // Работает!
```

Теперь, когда вы пытаетесь создать экземпляр с `int`, шаблон #2 ему не соответствует (ограничение возвращает ложь и отбрасывается из списка шаблонов), и мы можем использовать #1. Ура? Не совсем. Ограничение #1 пролезло в общую версию, добавив код, которого изначально не было. Представьте себе, что у вас было не одно, а три различных специализированных версии:

```
template Temp(T) if (is(T:int[])) // #1a
{ ... } // специально для массивов с int
```

```
template Temp(T) if (isRange!T) // #1b
{ ... } // специально для диапазонов

template Temp(T) if (is(T:double[n], int n)) // #1c
{ ... } // специально для статических массивов с double

template Temp(T) // #2, общий
   if ( /* Какое ограничение? */
{ ... }
```

Итак, быстро: какое ограничение для #2? Дополняющее *все* остальные ограничения. Смотрим:

```
template Temp(T) // #2, общий
   if ( !(is(T:int[]))
     && !isRange!T
     && !is(T:double[n], int n))
{ ... }
```

Это становится сложным для поддержки. Если кто-нибудь еще добавит четвертую специализацию, вам понадобится добавить четвертую обратную версию его ограничения. Ещё хуже, вы всё это компилируете и вызываете `Temp: Temp!(int[])`. И там ошибка! Почему? Потому что ограничения #1a и #1b не являются взаимно исключающими: int[] также является входным диапазоном. Это означает, что для #1b вам понадобится добавить исключения для массивов `int` и, может быть, модифицировать ограничение #2.

Ох.

Так что, да, ограничения замечательны, но у них есть недостатки. В качестве отправной точки, автор использует их всё время, даже если иногда Специализации более удобны: большинство из тех, что мне требуется накладывать и проверять для своих типов, не получится сделать через Специализации.

Ограничения, Специализации и static if

Я имею в виду, пора! Три разных способа решить, существует ваш шаблон или нет?

```
template Specialized(T : U[], U)
{ ... }

template Constrained(T) if (is(T : U[], U))
{ ... }

template StaticIfied(T)
{
    static if (is(T : U[], U))
    { ... }
    else // остановка компиляции
        static assert(0, "Для StaticIfied не может быть создан экземпляр.");
}
```

О чём думали разработчики D? Хорошо, они получили специализации из кузена D, из C++. Две других подсистемы были добавлены на несколько лет позже, когда мощь метапрограммирования времени компиляции в D становилась более явной и понадобились более мощные инструменты. Так что, «современные» подсистемы — это ограничения и **`static if`** (Static if). Ограничения являются более мощными, чем специализации, таким образом то, что вы можете проверить с помощью специализации, вы можете проверить через выражение с ограничением. А **`static if`** очень полезен вне экземпляра шаблона, так что они

оба хорошо имплантированы в D и должны там остаться.

Как насчет специализаций? Во-первых, они очень хороши при перенесении некоторого кода C++. Во-вторых, у них есть хороший эффект, которого нет у ограничений: когда экземпляр может быть создан для нескольких объявлений, приоритет даётся более специализированному. Вы видели объяснение в предыдущем подразделе.

Так что, в конечном итоге, в качестве вывода немного дзена D: вам даны инструменты, мощные инструменты. Так как они мощны, иногда они могут делать тоже, что и другие варианты в вашем инструментарии. D не ограничивает (!) вас, но выбирайте благоразумно.

Шаблоны-предикаты

Когда вы начинаете набирать снова и снова одно и то же выражение `is` или сложное ограничение, наступает пора абстрагировать его в другой шаблон, `bool`-подобный. Если вы бегло ознакомитесь с этим разделом, вы увидите способ проверить, что определённая часть кода на D работает (компилируется) или нет. Другой способ выяснить это — использовать `__traits(compiles, someCode)`.

В Примерах, в разделе Превращения типов, показан другой пример шаблона предиката.

Проверка для члена

Например, если вы хотите протестировать, может ли некоторый тип быть сериализован через метод `size_t serialize()`:

```
module isserializable;

template isSerializable(Type)
{
    static if (__traits(compiles, {
                                    Type type;
                                    size_t num = type.serialize;
                                   }))
        enum bool isSerializable = true;
    else
        enum bool isSerializable = false;
}
```

Проверка для операций

Как видно в предыдущих разделах (Struct Flatten, Constraints Usage), мы пишем здесь своего рода интерфейс времени компиляции. Любой тип может пройти этот тест, до тех пор, пока он имеет элемент `.serialize`, который возвращает `size_t`. Конечно, вы не ограничены проверкой методов. Вот шаблон, который проверяет, что некоторый тип работает с арифметическими операциями:

```
module hasarithmeticoperations;

template hasArithmeticOperations(Type)
{
    static if (__traits(compiles,
                        {
                            Type t, result;
                            result = t + t; // сложение
                            result = t - t; // вычитание
```

```
                            result = t * t; // умножение
                            result = t / t; // деление
                            result = +t;    // унарный +
                            result = -t;    // унарный -
                        }))
        enum bool hasArithmeticOperations = true;
    else
        enum bool hasArithmeticOperations = false;
}

static assert(hasArithmeticOperations!int);
static assert(hasArithmeticOperations!double);

struct S {}
static assert(!hasArithmeticOperations!S);
```

Как видно, вы можете протестировать D-код для любого типа, что означает, что это мощнее, чем Специализации шаблонов или выражение `is`.

Вы можете также получить определенное ощущение... картины, возникающей из двух предыдущих примеров. All the scaffolding, the boilerplate, is the same. And we could easily template it on what operator to test, for example. Это можно сделать, но это означает выработку кода во время компиляции. Подождите, пока вы не изучите строки mixin и CTFE в главе Вокруг шаблонов.

Завершение выравнивания диапазонов (Flatten)

Давайте вернёмся к `Flatten` (выравнивание) из раздела Struct Flatten. Используя проверяющие шаблоны, мы проверим, является ли диапазоном тип обёртки, и дополним `Flatten`, чтобы он стал лидирующим диапазоном (Forward range), если `Range` сам является лидирующим диапазоном:

```
import std.range;

struct Flatten(Range) if (isInputRange!Range)
{
    /* тот же код, что и раньше */

    static if (isForwardRange!Range)
        Flatten save() @property
        {
           return this;
        }
}
```

Структура расширена двумя путями: сначала, в ней запрещается создание экземпляра для не-диапазона. Это хорошо, поскольку с кодом из раздела Struct Flatten, вы могли бы обойти фабричную функцию и вручную создать `Flatten!int`, что приведёт ни к чему хорошему. Теперь вы не сможете. Во-вторых, если обёрнутый диапазон является лидирующим диапазоном (forward range), тогда `Flatten!Range` — также им будет. Это открывает совершенно новые алгоритмы для `Flatten`, с помощью достаточно удобочитаемой небольшой части кода.

Вы могли бы расширить пример, чтобы он так же позволял `Flatten` быть двунаправленным диапазоном (bidirectional range), но вам понадобится член `backSubRange`, в котором отслеживается состояние диапазона сзади.

Кортеж параметров Шаблона

Определение и Основные Свойства

И теперь будет одна из моих любимых тем: Кортеж параметров шаблона (template tuple parameters). Как мы видели в разделе Объявления шаблонов, они объявляются установкой идентификатора ... в последнем параметре шаблона. Кортеж поглотит любой тип, псевдоним или литерал, переданный ему. По этой самой причине (что это может быть несколько типов, перемежающихся с идентификаторами), некоторые рассматривают его как нечистопородное, гибридное дополнение к шаблонам D. Это действительно так, но удобство использования и гибкость, которые он дает нам, по-моему, хорошо компенсируют стоимость недостаточной чистоты. Кортежи шаблонов D имеют член `.length` (очевидно, определяемый во время компиляции), их элементы могут быть доступны с использованием стандартного синтаксиса индексирования, и их можно даже вырезать (символ `$` — псевдоним длины кортежа):

```
module dropfront;

template DropFront(T...)
{
    static if ( T.length > 0     // .length. По крайней мере, один элемент
             && is(T[0] == int)) // Индексирование
        alias T[1..$] DropFront; // Срез
    else
        alias void DropFront;
}

alias DropFront!(int, double, string) Shortened;
static assert(is( Shortened[0] == double));
static assert(is( Shortened[1] == string));
```

Вы можете объявить значение типа `tuple` (кортеж). Эта величина (называемая кортеж выражения), также имеет длину, может быть проиндексирована и может быть разрезана. Вы можете также передать её непосредственно в функцию, если типы совпадут со списком параметров функции. Если вы бросите её в массив, она растает и инициализирует массив:

```
module tupledemonstration;

template TupleDemonstration(T...)
{
    alias T TupleDemonstration;
}

unittest
{
    TupleDemonstration!(string, int, double) t;

    assert(t.length == 3);
    t[0] = "abc";
    t[1] = 1;
    t[2] = 3.14;
    auto t2 = t[1..$];
    assert(t2[0] == 1);
    assert(t2[1] == 3.14);

    void foo(int i, double d) {}
    foo(t2); // OK.

    double[] array = [t2]; // заметьте, между [ и ]
```

```
    assert(array == [1.0, 3.14]);
}
```

Самый простой возможный кортеж уже определен в Phobos в модуле std.typetuple.TypeTuple (*сейчас переименован в std.meta.AliasSeq, как сказано в справке, «чтобы снять неоднозначность с термина* ***tuple****», в модуле std.typetuple остался только псевдоним шаблона* `AliasSeq` *— прим. пер.*):

```
template TypeTuple(T...)
{
   alias T TypeTuple; // Просто выставляем T наружу
}

alias TypeTuple!(int, string, double) ISD;
static assert(is( TypeTuple!(ISD[0], ISD[1], ISD[2]) == ISD ));
```

Чистые кортежи параметров шаблона автоматически плоские: они не являются вложенными:

```
module ttflatten;
import std.typetuple;

alias TypeTuple!(int, string, double) ISD;
alias TypeTuple!(ISD, ISD) ISDISD;
// ISDISD - это не ((int, string, double), (int, string, double))
// Это (int, string, double, int, string, double)
static assert(is(ISDISD == TypeTuple!(int,string,double,int,string,double)));
```

Это — одновременно баг и фича. На отрицательной стороне, это обрекает нас на линейные структуры: без деревьев типов кортежей. А ветвящиеся структуры были бы мощнее, чем линейные. На положительной стороне, это позволяет нам легко объединять кортежи (и, значит, легко итерировать). Если вам нужны рекурсивные/ветвящиеся структуры, вы можете получить их, используя std.typecons.Tuple или любой настоящий тип шаблона структуры/класса: там типы не выравниваются. Смотрите, например, раздел Полиморфное Дерево.

Последнее свойство кортежей — по ним можно итерировать: используйте выражение `foreach`, так же, как для массива. С `foreach` вы можете выполнять итерации как для кортежей типов, так и для кортежей выражений. Индексная версия также возможна, но вы не можете запрашивать напрямую `ref`-доступ к значениям (но посмотрите пример внизу). Эта итерация выполняется во время компиляции, и фактически это один из основных способов выполнить цикл во время компиляции в D.

```
module ttiteration;

import std.typetuple;

unittest
{
    alias TypeTuple!(string, int, double) T;
    T t;
    t[0] = "abc";
    t[1] = 1;
    t[2] = 3.14;

    string[T.length] s;

    foreach(index, Type; T) // Итерации на типах.
```

```
                                // Type является различным, гм, типом в каждой позиции
    {
        static if(is(Type == double))
            s[index] = Type.stringof;
    }
    assert(s ==  ["", "", "double"]);

    void bar(T)(ref T d) { T t; d = t;}

    foreach(index, value; t) // Итерации на значениях.
                    // value (величина) имеет различный тип в каждой позиции!
    {
        bar(t[index]); // используйте t[index],
                       // а не 'value' для получения ref-доступа
    }

    assert(t[0] == "");
    assert(t[1] == 0);

    import std.math;
    assert(std.math.isnan(t[2]));
}
```

Так как величины этого типа можно создавать и называть, они *почти* перво-классные. Тем не менее, у них есть два ограничения:

- Нет встроенного синтаксиса для объявления кортежа. В предыдущем примере, вызов `T.stringof` возвращает строку `"(string, int,double)"`. Но Вы не можете написать `(string, int, double) myTuple;` непосредственно. Парадоксально, если Вы имеете тип кортежа `(string, int, double)` с именем `T`, вы можете выполнить `T myTuple;`.
- Эти кортежи нельзя возвращать из функции. Вы должны завернуть их в структуру. Это то, что предлагает std.typecons.Tuple.

Tuple, tuple, T... и .tupleof

Общий вопрос от новичков в D — различие и толкование между различными кортежами, которые можно найти в языке и в стандартной библиотеке. Я попытаюсь объяснить:

Кортежи параметров шаблона (template tuple parameters)

> внутренние для шаблонов. Они объявляются с помощью `T...` в последней позиции в списке параметров. Они группируют вместе список параметров шаблона, будь они типами, величинами или псевдонимами. Обычно используются два «подтипа»:

Кортежи типов (Type tuples)

> кортежи параметров шаблона, которые содержат только типы.

Кортежи выражений (Expression tuples)

> кортежи, которые содержат только выражения. Они являются тем, что вы получите, когда объявляете переменную типа `type tuple'.

Кортежи параметров функции (Function parameter tuples)

Вы можете получить кортеж типов параметров функции из std.traits.ParameterTypeTuple. Это точно такой же тип кортежа, как мы видели прежде. Величину этого типа можно объявлять и передавать в функцию с теми же параметрами.

Свойство .tupleof

свойство составных типов: классов и структур. Оно возвращает кортеж выражения, содержащий значения членов.

Кортеж имён членов (Member names tuple)

кортеж строк, которые вы получите, используя `__traits(members, SomeType)`. Он содержит все имена членов `SomeType`, в виде строк (включая методы, конструкторы, псевдонимы и т.д.).

std.typetuple.TypeTuple

std.typetuple.TypeTuple — предопределенный шаблон в Phobos, который является самым простым возможным шаблоном, содержащим кортеж. Это — обычный способ D, чтобы иметь дело с типом кортежей. Имя не совсем правильное, потому что это стандартный кортеж параметров шаблона: он может содержать как типы, так и значения.

std.typecons.Tuple

std.typecons.Tuple и std.typecons.tuple — предопределенные шаблоны структур/функций в Phobos, которые дают простой синтаксис манипулирования кортежами и возвращения их из функций.

Тип кортежей

Вы можете получить тип кортежа, используя `typeof(tuple)`, подобно любому другому типу D. Ограничения есть в двух случаях:

Одноэлементные кортежи

Есть различие между кортежем одного элемента и одиночным типом. Вы не можете инициализировать стандартное значение с 1-элементным кортежем. Вы должны извлечь первый элемент (и только) перед этим. По той же идее, 1-элементный кортеж имеет длину и может быть разрезан: действия, которые не имеют смысл для стандартного типа.

Ноль-элементные кортежи.

Возможно получить пустой кортеж, содержащий ноль типов, его не следует путать с неинициализированным кортежем или с кортежем, содержащим `void` в качестве типа. Фактически, 0-элементный кортеж может только иметь одно значение: значение инициализации. По этой причине, его иногда называют Unit-тип.[9]

9 `bool` — тип с двумя величинами (**`true`** и **`false`**). `()`, пустой кортеж — тип, который имеет только одно значение. А `void` является типом, который не имеет значений.

Кортежи, содержащие void, и пустые кортежи:

Кортеж типов может содержать тип `void`, подобно любому другому типу D. Он «использует слот» в кортеже, и поэтому кортеж, содержащий только `void`, не является пустым кортежем.

```
module emptytuple;
import std.typetuple;

alias TypeTuple!(void) Void;
alias TypeTuple!() Empty;
static assert( !is(Void == Empty) );

static assert(!is( TypeTuple!(int, void, string) == TypeTuple!(int, string)));
```

Пример: Функции с переменным числом аргументов (Variadic Functions)

Кортежи очень полезны для создания шаблонов variadic-функций (то есть, принимающих различное количество параметров). Без ограничения на передаваемые типы, вам в большинстве случаев понадобится шаблон другой функции для обработки аргументов. Стандартный пример этого — превращение всех параметров в `string`:

```
module variadic;

string toStrings(string sep = ", ", Args...)(Args args)
{
    import std.conv;
    string result;
    foreach(index, argument; args)
    {
        result ~= to!string(argument);
        if (index != args.length - 1)
            result ~= sep; // не для последнего
    }
    return result;
}

unittest
{
    assert( toStrings(1, "abc", 3.14, 'a', [1,2,3])
                  == "1, abc, 3.14, a, [1, 2, 3]");
}
```

Если вы хотите ограничить количество параметров или их типов, используйте ограничения шаблона:

```
int howMany(Args...)(Args args)
    if (Args.length > 1 && Args.length < 10)
{
    return args.length; // == Args.length
}
```

Представьте, что у вас есть набор диапазонов. Поскольку они все имеют различные типы, вы не можете поместить их в массиве. А так как большинство из них являются структурами, вы не можете сбросить их к базовому типу, как для классов. Таким образом, вы содержите их в кортеже. Затем, вам нужно вызывать у них базовые методы диапазона: вызывать `popFront` для всех, и т.п. Вот возможный способ добиться этого:

```
module popallfronts;
import std.range, std.algorithm;
```

```
import areallranges;

void popAllFronts(Ranges...)(ref Ranges ranges)
    if(areAllRanges!Ranges)
{
    foreach(index, range; ranges)
        ranges[index].popFront; // чтобы получить ref-доступ
}

unittest
{
    auto arr1 = [0,1,2];
    auto arr2 = "Hello, World!";
    auto arr3 = map!"a*a"(arr1);

    popAllFronts(arr1, arr2, arr3);

    assert(arr1 == [1,2]);
    assert(arr2 == "ello, World!");
    assert(equal( arr3, [1,4]));
}
```

Это работает для любого количества диапазонов, это круто. И это проверяется во время компиляции, вы не можете передать ему отдельный `int`, надеясь, что никто не увидит: шаблон `areAllRanges` работает, следя за этим. Код является классическим примером рекурсии на кортежах типов:

```
module areallranges;
import std.range;

template areAllRanges(Ranges...)
{
    static if (Ranges.length == 0) // Базовый случай: стоп.
        enum areAllRanges = true;
    else static if (!isInputRange!(Ranges[0])) // Найден не-диапазон: стоп.
        enum areAllRanges = false;
    else // Продолжаем рекурсию
        enum areAllRanges = areAllRanges!(Ranges[1..$]);
}
```

Люди, использующие языки, подобные lisp/Scheme или Haskell, будут тут практически у себя дома. Для остальных, небольшое объяснение для порядка:

- когда вы получаете кортеж типов, он или пуст или нет.
  - Если он пуст, тогда все элементы, содержащиеся в нём, являются диапазонами, и мы возвращаем **`true`** (истину).
  - Если он не пуст, у него есть по крайней мере один элемент, который может быть доступен по индексу. Давайте проверим его: диапазон это или нет.
    - Если это — не диапазон, останавливаем итерации: не все элементы являются диапазонами, мы возвращаем **`false`** (ложь).
    - Если же это — диапазон... мы ещё ничего не доказали, и нужно продолжать.

Рекурсия интересная: определяя константу `areAllRanges`, мы активизируем одноимённый трюк, который инициализирует константу в значение, полученное вызовом шаблона с сокращённым кортежем. Отрезая, мы выбрасываем первый тип (он уже был

протестирован) и продолжаем со следующим. В конце получится, что мы или исчерпали кортеж (случай длина == 0), или мы находим не-диапазон.

Одноэлементные кортежи: Прием типов и псевдонимов

Иногда для шаблона имеет смысл, чтобы он принимал или тип параметра, или псевдоним. Например, шаблон, который возвращает строку, представляющую его аргумент. В этом случае, поскольку параметр-тип не принимает идентификаторы в качестве аргументов, и то же самое для псевдонима, вы обречены на повторения:

```
module nameof2;

template nameOf(T)
{
    enum string nameOf = T.stringof;
}

template nameOf(alias a)
{
    enum string nameOf = a.stringof;
}

unittest
{
    assert(nameOf!(double[]) == "double[]");

    int i;
    assert(nameOf!(i) == "i");
}
```

Поскольку кортежи могут принимать как типы, так и псевдонимы, вы можете использовать их, чтобы немного упростить ваш код:

```
module nameof3;

template nameOf(T...) if (T.length == 1) // ограничено для одного аргумента
{
    enum string nameOf = T[0].stringof;
}
```

TODO Лучше объяснить по порядку. Я не так уж себя убедил.

Пример: Списки наследования

Незакончено Весь этот раздел должен быть переписан. Компилятор оказался более терпимым, чем я думал.

Используя шаблоны класса, мы могли бы захотеть отрегулировать список наследования во время компиляции. Кортежи типов являются хорошим способом для этого: сначала определить шаблон, который будет псевдонимом кортежу типов, затем наследовать класс от шаблона:

```
module interfacelist;
import std.typetuple;

interface I {}
interface J {}
interface K {}
interface L {}
```

```
class BaseA {}
class BaseB {}

template Inheritance(Base) if (is(Base == class))
{
    static if (is(Base : BaseA))
        alias TypeTuple!(Base, I, J, K) Inheritance;
    else static if (is(Base : BaseB))
        alias TypeTuple!(Base, L)       Inheritance;
    else
        alias Base                      Inheritance;
}

// Наследование от Base
class MyClass : Inheritance!BaseA { /*...*/ }
class MyOtherClass : Inheritance!BaseB { /*...*/ }
```

Здесь я шаблонизировал наследование на основе базового класса, но вы легко могли сделать шаблон на основе глобального `enum`, например. В общем случае, критерий выбора абстрагируется, и выбирающий код находится в одном месте, чтобы вы могли легко его изменить.

TODO Здесь мне нужно переписать связку между двумя частями этого раздела.

Давайте начнем с что-то более простого: принимаем тип и кортеж типов, и уберём все вхождения типа из кортежа.

```
1  module eliminate;
2  import std.typetuple;
3
4  template Eliminate(Type, TargetTuple...)
5  {
6      static if (TargetTuple.length == 0) // Кортеж исчерпан,
7          alias TargetTuple
8                Eliminate;    // работа выполнена.
9      else static if (is(TargetTuple[0] : Type))
10         alias Eliminate!(Type, TargetTuple[1..$])
11               Eliminate;
12     else
13         alias TypeTuple!(TargetTuple[0], Eliminate!(Type, TargetTuple[1..$]))
14               Eliminate;
15 }
16
17 unittest
18 {
19     alias TypeTuple!(int,double,int,string) Target;
20     alias Eliminate!(int, Target) NoInts;
21     static assert(is( NoInts == TypeTuple!(double, string) ));
22 }
```

Единственная трудность находится в строке 13: если первый тип не совпадает с `Type`, мы должны сохранить его и продолжить рекурсию:

```
Eliminate!(Type,  Type0, Type1, Type2, ...)
->
Type0, Eliminate!(Type,  Type1, Type2, …)
```

Мы не можем сопоставить типы, как я только что сделал, мы должны обернуть их в шаблоне. Phobos определяет `TypeTuple` в модуле std.typetuple для такого использования.

Теперь, когда мы знаем, как убрать все вхождения типа в кортеже типов, мы запишем

шаблон, устраняющий все дубликаты. Алгоритм прост: берём первый тип, убираем все случаи вхождения этого типа в остальном кортеже типов. Затем вызываем удаление дубликатов заново из результирующего кортежа типов, в то же самое время собирая в кортеж с первым типом.

```
module noduplicates;
import std.typetuple;
import eliminate;

template NoDuplicates(Types...)
{
    static if (Types.length == 0)
        alias Types NoDuplicates; // Нет типов, нечего делать.
    else
        alias TypeTuple!(
                          Types[0]
                        , NoDuplicates!(Eliminate!(Types[0], Types[1..$]))
                        ) NoDuplicates;
}

static assert(is( NoDuplicates!(int,double,int,string,double)
                  == TypeTuple!(int,double,string)));
```

Между прочим, код, который это делает, также с именем `NoDuplicates`, — уже есть в Phobos. Его можно найти в модуле std.typetuple (*теперь в std.meta — прим. пер.*). Я нашел код, который тоже является хорошим упражнением по обработке кортежа типов. Вы можете найти несколько примеров этого типа шаблонов в разделе Превращения типов.

Последней частью задачи будет получить список наследования данного класса. Выражение `is` даёт нам его, через Специализации типа:

```
module superlist;
import std.typetuple;

template SuperList(Class) if (is(Class == class))
{
    static if (is(Class Parent == super))
        alias TypeTuple!(Parent, SuperList!Parent) SuperList;
}

// для Object, is(Object Parent == super)) даёт пустой кортеж типов
template SuperList()
{
    alias TypeTuple!() SuperList;
}

unittest
{
    class A {}
    class B : A {}
    class C : A {}
    class D : C {}

    static assert(is(SuperList!Object == TypeTuple!()));
    static assert(is(SuperList!A      == TypeTuple!(Object)));
    static assert(is(SuperList!B      == TypeTuple!(A, Object)));
    static assert(is(SuperList!D      == TypeTuple!(C, A, Object)));
}
```

TODO Переписать конец.

К сожалению, этот раздел в оригинале так и не был дописан до конца — прим. пер.

Перегрузка операторов

D позволяет переопределять некоторые операторы, чтобы повысить удобочитаемость в коде. И догадайтесь как? Перегрузка операторов основана на методах шаблонов. Они описаны в документации.

Синтаксис

Следующие подразделы рассказывают об операторах, которые вы можете перегружать, и о том, какие методы шаблона вы должны определить в вашем собственном типе. `u` — величина типа `U`, для которого вы хотите определить операторы. `v` — величина типа `V`, с которым `u` будет взаимодействовать. `V` может быть тем же типом, что и `U`, или отдельным типом.

op является одним из «рассматриваемых» операторов, вводимых в каждом разделе.

Унарные операторы

Рассматриваемые операторы: `+`, `-`, `++`, `--`, `*` и `~`.

Операция	Определяемый метод шаблона
op`u`	`opUnary(string s)() if (s == "`*op*`")`
op`u[i0]`	`opIndexUnary(string s)(size_t i0) if (s == "`*op*`")`
op`u[i0, i1]`	`opIndexUnary(string s)(size_t i0,size_t i1) if (s == "`*op*`")`
op`u[i0, i1, in]`	`opIndexUnary(string s)(size_t i0, ...) if (s == "`*op*`")`
op`u[i..j]`	`opSliceUnary(string s)(size_t i, size_t j) if (s == "`*op*`")`
op`u[]`	`opSliceUnary(string s)() if (s == "`*op*`")`

Бинарные операторы

Рассматриваемые операторы: `+`, `-`, `*`, `/`, `%`, `^^`, `~`, **`in`**, `&`, `|`, `^`, `<<`, `>>` и `>>>`.

Операция	Определяемый метод шаблона
`u` *op* `v`	`opBinary(string s, V)(V v) if (s == "`*op*`")`
`v` *op* `u`	`opBinaryRight(string s, V)(V v) if (s == "`*op*`")`

Присвоение

Рассматриваемые операторы: `+`, `-`, `*`, `/`, `%`, `^^`, `~`, `&`, `|`, `^`, `<<`, `>>` и `>>>` (отсутствует **`in`**).

Операция	Определяемый метод шаблона
`u` *op* `= v`	`opOpAssign(string s, V)(V v) if (s == "`*op*`")`

Индексированное присвоение

Рассматриваемые операторы: `+`, `-`, `*`, `/`, `%`, `^^`, `~`, `&`, `|`, `^`, `<<`, `>>` и `>>>` (отсутствует **`in`**).

Операция	Определяемый метод шаблона
`u[i0, i1, in]` *op* `= v`	`opIndexOpAssign(string s, V)(V v, size_t i0, ...)` `  if (s == "`*op*`")`

Присвоение среза

Рассматриваемые операторы: `+`, `-`, `*`, `/`, `%`, `^^`, `~`, `&`, `|`, `^`, `<<`, `>>` и `>>>` (отсутствует **`in`**).

Операция	Определяемый метод шаблона

`u[i..j] op = v`
 	
`u[] op = v`
	`opSliceOpAssign(string s, V)(V v, size_t i, size_t j)`
	`  if (s == "op")`
	`opSliceOpAssign(string s, V)(V v)`
	`  if (s == "op")`

Оператор Cast

Операция	Определяемый метод шаблона
`cast(T)u`	`T opCast(T)()`

Множество других операторов можно перегружать в D, но в них не требуются шаблоны.

Пример: Арифметические операторы

TODO Сообщить где-нибудь, что это возможно:

```
Foo opBinary(string op:"+")(...) { ... }
```

Идея, стоящая за этим странным способом перегружать операторы — позволить вам переопределять множество операторов сразу с помощью всего одного метода. Например, возьмём эту структуру, оборачивающую число:

```
struct Number(T) if (isNumeric!T)
{
    T num;
}
```

Чтобы передать ей четыре основных арифметических оператора с другой `Number` и другим `T`, вы определяете `opBinary` для `+`, `-`, `*` и `/`. Это активизирует операции, где `Number` находится слева. В случае, если она находится справа, вы должны определить `opBinaryRight`. Поскольку эти перегрузки стремятся использовать строки mixins, я использую их даже несмотря на то, что они вводятся только в разделе Строки Mixin. Основная идея: примешиваются (mixins) строки кода (передаваемые как строки времени компиляции) там, где они размещены.

```
module number;
import std.traits;

struct Number(T) if (isNumeric!T)
{
    T num;

    auto opBinary(string op, U)(U u)
         if ((op == "+" || op == "-" || op == "*" || op == "/")
         && ((isNumeric!U) || is(U u == Number!V, V)))
    {
 mixin("alias typeof(a" ~ op ~ "b) Result;
        static if (isNumeric!U)
            return Number!Result(a"~op~"b);
        else
            return Number!Result(a"~op~"b.num);");
    }
}
```

`op` является параметром шаблона, он используется, чтобы выполнять во время компиляции константную свёртку: в этом случае происходит конкатенация строк для генерации кода D. Код написан так, чтобы структуры `Number` соблюдали глобальные правила D. `Number!int` плюс `Number!double` возвращает `Number!double`.

Специальный случай: in

Незакончено

Специальный случай: cast

Незакончено

Шаблоны Mixin

Вплоть до этого момента, все шаблоны, которые мы видели, образуют экземпляры в той же области видимости, что и их объявления. Шаблоны Mixin (примесь) имеют другое поведение: код, который они содержат, устанавливается *прямо на месте вызова*. Таким образом, они используются совершенно иначе, чем другие шаблоны.

Синтаксис

Чтобы отличать стандартные шаблоны от шаблонов mixin, последние имеет несколько другой синтаксис. Вот как они объявляются и вызываются:

```
/* Объявление */
mixin template NewFunctionality(T,U) { ... }


/* Реализация */
class MyClass(T,U,V)
{
    mixin NewFunctionality!(U,V);

    ...
}
```

Как вы можете видеть, слово **`mixin`** ставится перед объявлением и перед вызовом экземпляра. Все остальные тонкости шаблонов (ограничения, значения по умолчанию, ...) — по-прежнему существуют для вашего рассмотрения. Поиск идентификаторов ведётся в локальной области видимости, и результирующий код вставляется там, где делается вызов, следовательно, впрыскивая новую функциональность.

Насколько я знаю, нет специального синтаксиса для шаблонов функций, классов и структур, чтобы они были шаблонами mixin. Вы должны завернуть их в стандартное объявление шаблона **`template`**. Точно также, как нет понятия одноимённого трюка с шаблонами mixin: нет вопроса о том, как дать доступ к содержимому шаблона, с момента, как шаблон открылся для вас, и его настоящее содержимое вложено в ваш код.

TODO проверить `mixin T foo(T)(T t) { return t;}`

Между прочим, вы *не можете* смешивать со стандартным шаблоном внутри. Раньше это было так, но больше нельзя. Теперь шаблоны mixin и не-mixin строго разделённые кузены.

Примешивание кода

Что такого хорошего в этих кузенах шаблонов, которые мы видели до сих пор? Они дают вам хороший способ устанавливать параметризованную реализацию в классе или структуре. Еще раз, шаблоны являются возможностью уменьшить повторения в коде. Если некоторая часть кода появляется в различных местах вашего кода (например, в структурах, у которых нет наследования, позволяющего избегать дублирования кода), вы должны поискать способ поместить её в шаблоне mixin.

Также, вы можете поместить небольшие функциональности в шаблоны mixin, давая клиентскому коду доступ к ним, чтобы выбрать, как лучше строить типы.

Заметьте, что код, размещённый в шаблоне mixin, не обязан иметь самостоятельного смысла (он может ссылаться на **this** или на любые другие, пока ещё не определённые, идентификаторы). Это просто должен быть синтаксически правильный код на D.

Например, помните код, перегружающий оператор, который мы видели в разделе Перегрузка Операторов? Вот mixin, содержащий функциональность конкатенации (объединения):

```
module mixinconcatenate;
import std.typecons;

mixin template Concatenate()
{
    Tuple!(typeof(this), U) opBinary(string op, U)(U u)
    if (op == "~")
    {
        return tuple(this, u);
    }
}
```

Как вы можете видеть, здесь используется **this**, даже если нет структуры или класса в поле зрения. Это можно использовать как-то так, чтобы придать возможность конкатенации структуре (в виде кортежа):

```
module usingmixinconcatenate;
import std.typecons;
import mixinconcatenate;

struct S
{
    mixin Concatenate;
}

unittest
{
    S s,t;
    auto result = s ~ t;
    assert(result == tuple(s,t));
}
```

Идея, которую следует принять: код конкатенации пишется один раз. Именно тогда предлагается функциональность для любой области видимости клиента (типа), где это нужно. Это могут быть арифметические действия, операторы преобразования типа **cast**, или новые методы, подобные log (журналирование), register (регистрация), новые члены или что-то ещё. Стройте ваш собственный набор Mixin'ов и используйте их свободно. И помните, что

они не ограничены классами и структурами: вы можете использовать их также в функциях, областях видимости модуля, других шаблонах...

Ограничения. Шаблоны Mixin вставляют код в локальной области. Их нельзя добавлять к условию `invariant` в классе, или к условиям `in`/`out` в функции. Их можно вставлять в условия `invariant`/`in`/`out`.

Пример Mixin: Подписчик и Стек

Этот пример получен от Bjorn Lietz-Spendig, который был достаточно любезен, чтобы позволить мне использовать его. Спасибо Bjorn!

Мы определим два шаблона mixin, `PublisherMixin` и `StackMixin`, первый осуществляет движок подписки/отказа от подписки, второй обеспечивает стандартные операции стека (`push`, `pop`,...). Я хотел бы, чтобы вы заметили здесь две важные вещи:

- D позволяет локальный импорт, `PublisherMixin` импортирует механизмы, необходимые для ее функционирования: `std.functional.toDelegate` и `std.stdio`. Локальный импорт действительно позволяет шаблонам mixin в D обеспечивать хорошо обёрнутую функциональность в виде единого целого.
- `alias typeof(this)` (здесь названная `Me`) — также хорошая возможность для запоминания: mixin 'осмотрится', получит тип локального `this`, а затем может предоставить общий готовый к использованию код.

```
module publisher;

public mixin template PublisherMixin()
{
    import std.functional : toDelegate;
    import std.stdio;

    alias void delegate(Object sender, string event) CallBack;
    alias void function(Object sender, string event) CallBackFun;


    bool[CallBack] callBacks;  // int[0][CallBack] может быть даже меньше

    //Регистрация подписчика
    void register(CallBack callBack)
    {
    // Удостовериться, что подписчик еще не зарегистрирован.
        if (callBack in callBacks)
            writeln("Подписчик уже зарегестрирован.");
        else
            callBacks[callBack] = true; //from;
    }

// Register Subscriber via function ptr.
    void register(CallBackFun callBackFun)
    {
        register( toDelegate(callBackFun) );
    }

    // Удаление Подписчика
    void unRegister(CallBack callBack)
    {
```

```
        if (callBack in callBacks)
            callBacks.remove(callBack);
        else
            writeln("Попытка удаления неизвестного обратного вызова.");
    }

    // (Remove Subscriber via function ptr.)
    void unRegister(CallBackFun callBackFun)
    {
        unRegister(toDelegate(callBackFun));
    }

    // Уведомить ВСЕХ Подписчиков
    void notify(Object from, string evt)
    {
        foreach ( CallBack cb, bool origin ; callBacks )
        {
            cb( from, evt );
        }
    }
}

mixin template StackMixin()
{
    // получить родительский тип
    alias typeof(this) Me;
    static Me[] stack;

protected:

    @property bool empty()  { return stack.length == 0;  }
    @property size_t count()   { return stack.length; }

    void push(Me element) // Поместить
    {
        stack ~= element;
    }

    Me pop() // Извлечь
    {
        Me el = peek();
        stack.length -= 1;
        return el;
    }


    Me peek() // Читать
    {
        if ( stack.length == 0 )
            throw new Exception("чтение пустого стека.");

        Me el = stack[stack.length-1];
        return el;
    }
}
```

Теперь, когда mixin'ы определены, мы можем впрыснуть (inject) их в любую структуру с заданным `this`. Здесь мы используем классы, чтобы показать, что mixin'ы наследуются. Давайте заведём несколько Друзей (`Friends`):

```
module friends;
import std.stdio;
import publisher;
```

```d
class FriendStack
{
    private string name;
    private int age;

    // Наши Mixin'ы
    mixin StackMixin;
    mixin PublisherMixin;

    void pushFriend(string name, int age)
    {
        // Создать новый экземпляр стека.
        auto person = new FriendStack();
        person.name = name;
        person.age = age;

        push(person);

        // Оповестить всех подписчиков
        notify(person, "Push");
    }

    //Pop
    void popFriend()
    {
        auto person = pop();
        notify( person, "Pop");
    }

    // Метод подписчика
    void inHouseInfo(Object sender, string msg)
    {
        auto p = cast(FriendStack)sender;
        writefln("Подписчик: в доме , Имя: %s, Возраст: %s, Сообщение: %s \n",
                 p.name, p.age, msg);
    }
}

class ExtFriendStack : FriendStack // наши VIP'ы
{
    private bool isDeveloper;

    // Push
    void pushFriend(string name, int age, bool D)
    {
        // Создать новый экземпляр стека.
        auto xperson = new ExtFriendStack();
        xperson.name = name;
        xperson.age = age;
        xperson.isDeveloper = D;

        push(xperson);

        // Оповестить всех подписчиков
        notify(xperson, "Push");
    }
}

/**
 *
 *  FRIENDSTACK Функции подписчиков - Используется FriendStack И ExtFriendStack.
 */
void twitter(Object sender, string msg)
```

```
{
    auto p = cast(FriendStack)sender;
    writefln("Подписчик: Twitter, Имя: %s, Возраст: %s, Сообщение: %s \n",
             p.name, p.age, msg);
}

void reddit(Object sender, string msg)
{
    auto p = cast(FriendStack)sender;
    writefln("Подписчик: Reddit, Имя: %s, Возраст: %s, Сообщение: %s \n",
             p.name, p.age, msg);
}

void mail(Object sender, string msg)
{
    auto p = cast(FriendStack)sender;
    writefln("Подписчик: eMail, Имя: %s, Возраст: %s, Сообщение: %s \n",
             p.name, p.age, msg);
}

/**
 *
 *  EXTENDED FRIENDSTACK  Функции подписчиков
 *  содержит дополнительную —isDeveloper-- (разработчик?) информацию.
 */
void blog(Object sender, string msg)
{
    if ( cast(ExtFriendStack)sender )
    {
        auto p = cast(ExtFriendStack)sender;
        writefln("Подписчик: Blog, Имя: %s, Возраст: %s, Разработчик?: %s,
Сообщение: %s \n",
                 p.name, p.age, p.isDeveloper, msg);
    }
    else
    {
        writeln("----- Блог только для VIP ------ Сообщение привязано к email");
        mail(sender, msg);

        // Нисхождение также возможно
        //auto p = cast(FriendStack)sender;
        //writefln("Подписчик: Blog, Имя: %s, Возраст: %s, Сообщение: %s \n",
        //         p.name, p.age, msg);
    }
}
```

И теперь мы можем использовать стеки для наших сердечных писем и добавления некоторых друзей:

```
module usingfriends;
import friends;

void main()
{
    auto p = new FriendStack();
    auto p2 = new ExtFriendStack();

    // Регистрация нескольких подписчиков.
    p.register(  &twitter );
    p2.register( &twitter );
    p2.register( &blog );
    p2.register( &blog );

    // Push и Pop
```

```
    p.pushFriend( "Alex", 19 );
    p.pushFriend( "Tommy", 55);
    p2.pushFriend( "Hans", 42, false);
    p2.pushFriend( "Walt", 101, true);

    p.popFriend();
    p2.popFriend();
    p2.popFriend();

    p.unRegister( &twitter );
    p2.unRegister( &twitter );
    p.register( &blog );
    p.pushFriend( "Alexej", 33 );
}
```

Если вы запустите предыдущий код, вы увидите что `DeveloperStack` наследует mixin'ы из `PersonStack`: он одновременно стек и издатель. Благодаря `alias typeof(this)`, определенному в `StackMixin`, `ExtFriendStack` может содержать дополнительную информацию.

opDispatch

Синтаксис

`opDispatch` — своего рода перегруженный оператор (это в том же самом месте), который имеет дело с вызовами членов (методов или данных-членов). Определение такое же, как и у оператора:

```
... opDispatch(string name)()
... opDispatch(string name, Arg)(Arg arg)
... opDispatch(string name, Args...)(Args args)
```

Можно использовать обычные ограничения шаблона: ограничения на имя, ограничения на аргументы.

Когда тип имеет метод `opDispatch`, и при вызова члена этот член отсутствует среди определённых, вызов пересылается в `opDispatch` с вызываемым именем в виде строки.

```
module dispatcher;
import std.typetuple;

struct Dispatcher
{
    int foo(int i) { return i*i;}
    string opDispatch(string name, T...)(T t)
    {
        return "Dispatch активирован: " ~ name ~ ":" ~ TypeTuple!(T).stringof;
    }
}

void main()
{
    Dispatcher d;

    auto i = d.foo(1); // компилятор находит foo, вызывает foo.
    auto s1 = d.register("abc"); // нет члена register -> opDispatch активирован;
    assert(s1 == "Dispatch активирован: register:(string)");

    auto s2 = d.empty; // нет члена empty, нет аргумента.
    assert(s2 == "Dispatch активирован: empty:()");
```

```
}
```

Как только `opDispatch` был вызван с именем и аргументами, вам решать что, что делать: вызвать свободные функции, вызывать другие методы, или использовать строку времени компиляции для генерации нового кода (смотрите раздел Строки Mixin).

Поскольку строки mixins действительно ходят рука об руку с `opDispatch`, я использую их даже несмотря на то, что я не ввёл их прямо сейчас. Краткое изложение: они вставляют код D (передаваемый как строки времени компиляции) там, где они вызываются. Там.

Getters и Setters

Например, предположим, что у вас есть куча данных-членов, все со спецификатором доступа **`private`**, и вы хотите, чтобы клиентский код имел к ним доступ через старые добрые `setXXX/getXXX`-методы. Только вы не хотите писать все эти методы самостоятельно. Вы счастливчик, так как `opDispatch` может вам помочь.

```
module getset;

class GetSet
{
    private int i;
    private int j;
    private double d;
    private string theString;

    auto opDispatch(string name)() // версия без аргументов -> getter
    if (name.length > 3 && name[0..3] == "get")
    {
        enum string member = name[3..$]; // "getXXX" -> "XXX"
        // Мы проверяем, присутствует ли "XXX":
        // т.е. if is(typeof(this.XXX)) is true
        static if (__traits(compiles,
                            mixin("is(typeof(this." ~ member ~ "))")))
        mixin("return " ~ member ~ ";");
        else
            static assert(0, "Ошибка GetSet: не существует вызываемого члена " ~
                          member);
    }

    auto opDispatch(string name, Arg)(Arg arg) // setter
        if (name.length > 3 && name[0..3] == "set")
    {
        enum string member = name[3..$]; // "setXXX" -> "XXX"
        // Мы проверяем, можно ли "member" на что-то назначить.
        // this.member = Arg.init
          static if (__traits(compiles, mixin("{" ~ member ~ " = Arg.init;}")))
          {
              mixin(member ~ " = arg;");
              mixin("return " ~ member ~ ";");
          }
          else
              static assert(0, "Ошибка GetSet: не существует вызываемого члена " ~
                            member);
    }
}

unittest
{
```

```
    auto gs = new GetSet();
    gs.seti(3);
    auto i = gs.geti;
    assert(i == 3);

    gs.settheString("abc");
    assert(gs.gettheString == "abc"); // "abc"
}
```

Изящно, а? Могло быть немного лучше, имей мы дело с капитализацией первой буквы: `getTheString`, но и сейчас достаточно хорошо. Еще лучше, если вы могли бы разместить этот код в шаблоне Mixin, чтобы дать эту get/set-возможность любой структуре или классу (смотрите раздел Шаблоны Mixin).

Обёртки и Подтипы: Расширение Типов

Обёртывающие Шаблоны

Незакончено Идея в том, чтобы поместить внутрь некоторую функциональность перед (и возможно, после) диспетчеризации кода.

Мы видели, как впрыскивать код с помощью Шаблонов mixin, или использовать наследование шаблона класса для модификации кода ваших классов (Добавление функциональности через наследование). Мы также видели, как вы можете определить обёртывающую структуру вокруг диапазона, чтобы вывести новую схему итераций для его элемента (Структура Flatten). Все эти идиомы являются способом модифицировать ранее существующий код.

Но что если вы хотите поместить функциональность журналирования (logging) ко встроенной структуре, чтобы регистрировался вызов любого метода? Для класса вы можете унаследоваться от него и определить подкласс с новыми, модифицированными, методами. Но вам придётся сделать это "вручную", так сказать. А в случае со структурой вам не повезло.

Но шаблоны могут прийти на помощь, с чуточкой магии opDispatch.

TODO Закончить это.

- оберните Type в структуру Logger.
- получите Type.tupleof
- вызовите typeof() на this.
- opDispatch Test if wrapped.foo() легально.

alias this: Прозрачные типы

Мы поиграем немного с конструкцией `alias this`. Давайте определим небольшой обёртывающий тип:

```
module transp;

struct Transparent(T)
{
    T value;
    alias value this;
}
```

```
// фабричная функция
Transparent!T transparent(T)(T t)
{
    return Transparent!T(t);
}
```

Мы определили `Transparent` (прозрачно), небольшую структуру, содержащую одинокую величину `value` типа `T`. Если `Transparent!T` просят сделать то, для чего он не имеет ни какого-либо метода, ни правильного типа (то есть, почти любое действие, которое вы себе можете представить!), активизируется **`alias this`** и использует `value` вместо `Transparent!T`. Это даст нам ту же прозрачную обёртку:

```
module usingtransparent;
import transp;

void main()
{
    auto i = transparent(10);
    assert(i == 10); // проверка равенства

    i = 1; // Присваивание от int

    ++i; // ++ работает на i.value
    assert(i == 2);

    // функция, принимающая int
    int foo(int ii) { return ii+1;}

    assert(foo(i) == 3); // вызов функции

    int trueInt = i; // передача своего значения value

    assert(trueInt == 2);
    assert(i == 2);
    assert(i == trueInt);

    // Последовательные вызовы всё сворачивают в один Transparent.
    i = transparent(transparent(i));
}
```

Как вы можете видеть, `Transparent` ведет себя очень грациозно: почти во всех обстоятельствах, которым может соответствовать `int`, `Tranparent!int` действует как реальный `int`. Конечно, его тип по-прежнему в порядке:

```
assert(is(typeof(i) == Transparent!int));
```

Можно добавить другую обёртку:

```
module transp2;
import transp;

struct Thin(T)
{
    T value;
    alias value this;
}

Thin!T thin(T)(T t)
{
    return Thin!T(t);
}
```

```
unittest
{
    auto i = transparent(10);
    auto t = thin(i); // Работает.
    assert(t == 10); // Да, значение value было передано правильно.

    assert(is(typeof(t) == Thin!(Transparent!int)));
}
```

Какая польза от такой невидимой обертки? Во-первых, это — отличный подтип, который — сам является целью `alias X this`. Вы можете добавить функциональность к вашему типу: новые методы, и т.п.. Выглядит как наследование класса, не правда ли? Разве что `Transparent` можно обернуть вокруг (т.е., он станет подтипом) любого типа, на который он направлен. Если вы заинтересованы в многочисленных способах создавать подтипы типа, смотрите раздел Библиотека Typedef, показывающую библиотеку typedef от Trass3r.

Заметьте, что это отличается от идеи, показанной в разделе Обёртывающие шаблоны. В этом разделе использование `opDispatch` позволяет обёртке прерывать вызовы перед их перенаправлением. Этим способом вы можете, например, добавить журналирование к типу для экземпляра. Это то, что вы не можете сделать с `alias this`. С другой стороны, диспетчерирование у `alias this` выполняется автоматически посредством компилятора, не требуя писать прерывающий код.

Например, здесь — другая обёртка, которая дает описание удерживаемого типа:

```
module descriptor;
import std.stdio;

struct Descriptor(T)
{
    T value;
    alias value this;

    string description() @property
    {
        import std.conv;

        return "Descriptor содержит " ~ T.stringof
             ~ " со значением " ~ to!string(value) ~ ".\n"
             ~ "(его размер: " ~ to!string(T.sizeof) ~ " байт)";
    }
}

void main()
{
    auto d = Descriptor!double(1.0);
    writeln(d.description); // "Descriptor содержит double со значением 1
                            //  (его размер: 8 байт)"
}
```

Однажды, D получит возможность использовать множественные `alias this`, как это описано в TDPL. Это откроет возможность быть подтипом многих различных типов одновременно, буквально действуя как хамелеон в зависимости от обстоятельств. Мы сможем группировать несопоставимые типы и функциональные возможности в одном месте.

Я жду это с возрастающим интересом.[10]

Тем не менее, насколько я понимаю, основной интерес к таким подтипам — не столько возможность добавить «параллельную» функциональность, сколько кодирование информации в самом типе. Смотрите следующий раздел:

Библиотека Typedef

Как мы видели в предыдущем подразделе, `alias this` дает отличный подтип, способный сымитировать свой родительский тип во всех обстоятельствах. Но что, если требуются другие производные типы? В D было ключевое слово `typedef` для определения нового типа (по сравнению с повторным определением псевдонима `alias`, которое просто определяет новый идентификатор/имя для типа), но было выкинуто из языка в 2011 году. Однако, необходимость определения новых типов, связанных с ранее существующими типами, по-прежнему существует. Шаблоны могут помочь в определении решения библиотечного уровня, который описывается здесь.

Главным образом, когда новый тип определяется связанным с другим, уже существующим, целевым типом, может быть четыре различных связи:

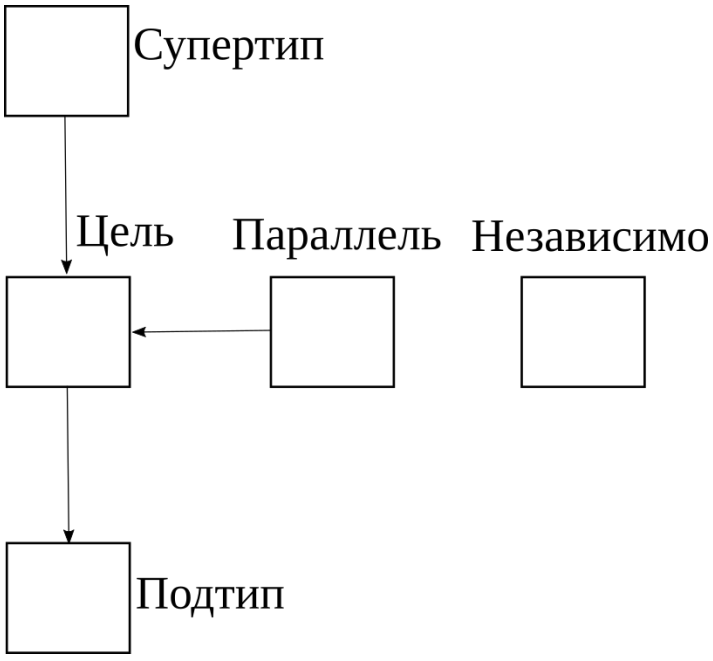

Рис.1: Отношения Typedef

подтип

> Это — стандартный подкласс или отношение `alias this`: подтип может действовать вместо своего родительского типа во всех обстоятельствах. Независимо от того, что родитель может делать, подтип действует также. Кроме того, подтип можно принудительно привести (cast) к родительскому типу (вспомните классы).

10 Это было написано в январе 2012 года, давайте посмотрим, когда мы получим множественные `alias this`.

супертип

> Обратно: супертипу можно назначить величину "целевого" типа, подтипа (поскольку подтип может действовать как супертип), и не более того. Как только супертип и функции, действующие на нем созданы, оригинальный целевой тип там также может использоваться.

параллель

> Параллельный тип не является ни подтипом, ни супертипом оригинальной цели (он не может имитировать цель), но он может быть создан с целевыми значениями. Это — более или менее похоже на поведение ныне покойного ключевого слова **typedef**.

независимо

> Независимый тип просто есть и не имеет никакого отношения к оригинальному, целевому типу. Я привожу его здесь, но интерес определять такой тип и при этом иметь в виду "целевой" тип, ограничен …

Вот чистый маленький кусочек кода от Trass3r, который объединяет все эти понятия в одном шаблоне:

```
module librarytypedef;

enum Relationship
{
    Independent,
    Super,
    Sub,
    Parallel,
}

struct Typedef( Target,
                Relationship relation = Relationship.Sub,
                Target init = Target.init,
                string _f = __FILE__,
                int _l = __LINE__ )
{
    Target payload = init;

    static if ( relation != Relationship.Independent )
        this( Target value )
        {
            payload = value;
        }

    static if ( relation == Relationship.Sub)
        // typedef int foo; foo f;
        // f.opCast!(t)() == cast(t) f
        Target opCast(Target)()
        {
            return payload;
        }

    static if ( relation == Relationship.Sub
              || relation == Relationship.Parallel )
        alias payload this;
```

```
    static if ( relation == Relationship.Super )
        typeof( this ) opAssign( Target value )
        {
            payload = value;
            return this;
        }
    else static if ( relation == Relationship.Sub )
        @disable void opAssign( Target value );
}
```

TODO Дать больше примеров.

Типы как информация

Литералы, задаваемые пользователем

В http://drdobbs.com/blogs/tools/229401068, Walter Bright приводит убедительные доводы для использования шаблонов, как определяемых пользователем литералов. Его пример стал теперь обёрткой std.conv.octal, которая позволила D выкинуть из конструкций компилятора восьмеричные литералы и вытолкнуть их в пространство библиотеки. Таким образом, вы можете использовать:

```
auto o1 = octal!"755";
// или даже:
auto o2 = octal!755;
```

как дублера для замены восьмеричных литералов 0755. Это не сложнее набирать (добавлено всего несколько символов), но для читателя вашего кода (который может пропустить начальный 0) яснее, что он имеет дело с восьмеричной записью. Реализация в виде библиотечного кода лучше: легче добраться, легче отлаживать, менее пугающе, чем погружаться в код компилятора, и легче продублировать для другого кодирования. Этим способом можно сделать:

```
auto h1 = hexa!"deadbeef";
auto b1 = binary!1011011;
// или даже:
auto number = base!(36, "4d6r7th2h7y");
```

За сценой, `octal` читает свой `string`- или `int`-аргумент и преобразует его в `int` или `long` в зависимости от размера аргумента. Это — хорошая часть работы, которая могла бы проложить путь для аналогичных хорошо-интегрированных языковых расширений.

Незакончено Также, DSL в строках на D, смотрите Статически проверяемый Writeln, Кодирование информации с типами и Комментирование типов.

Кодирование информации с типами

Это расширение идеи предыдущего раздела, которое можно найти, например, в std.range.assumeSorted и std.range.SortedRange. Эти конструкции в Phobos кодируют некоторую информацию в типе (в данном случае, то, каким образом отсортирован диапазон, с соответствующим предикатом). Таким образом, последующие операции, действующие на диапазоне, могут использовать лучший алгоритм, если они знают, как он отсортирован.

Этот тип кодирования может быть использован для множества различных схем:

- Для библиотеки матриц вы могли бы иметь, например, `assumeHermitian` или

`assumeTriangular`, которые просто обертывали бы предварительно существующую матрицу.

- Для XML/HTML библиотеки, вы могли бы представить себе `Validated` (подтверждённые) структуры, которые используются, для указания, эм, проверенного содержимого. Любой внешний вход должен пройти через проверяющую функцию (`validate`), которая формирует `Validated` объект. Последующие операции будут работать только с `Validated` данными, а не с исходными.
- Библиотека единиц измерения (таких, как kg.m/s^2), — в основном разные величины `double` (или `complex`), завернутых в типы с кучей проверок, которые допускают только некоторые действия, в зависимости от типов. `meter(5) + kilo(gram(10))` — нельзя-нельзя, а `meter(5)*kilo(gram(10))` — можно.

Кроме того, легко предоставить дополнительную информацию:

```
struct MinMax(T)
{
    T min;
    T max;

    T value;

    alias value this;
}
```

В этом случае, `MinMax` означает, что оборачиваемое значение находится в пределах между `min` и `max` (код, который обеспечит выполнение, может быть, например, в фабричной функции).

Всё становится интереснее при использовании многочисленных обёрток друг в друге. Представьте себе что у нас есть три обёртки, для численных диапазонов с операцией упорядочения:

- `minmaxed`, утверждающей (хорошо, по крайней мере передающей сообщение), что обернутая величина находится между экстремумами.
- `periodic`, которая кодирует идею того, что диапазон имеет период, доступный через свойство `.period`.
- `derivative` (производная), в которой говорится, что разница между последовательными элементами не больше, чем число `.slope` (наклон).

```
import std.range: cycle;

auto range = cycle([0,1,2,3,4,3,2,1]);
auto withMetadata = minmaxed(
                              periodic(
                                       derivative( range
                                                 , 1)
                                     , 8
                            , 0, 10);

assert(withMetadata.period == 8); // всё распространяется вверх.
```

Так, в предыдущем примере, элементы `withMetadata` одновременно ограничены в изменении, ограничены в величине и в периоде. Беда случится, когда мы захотим переместить метаданные, сравнивая типы с метаданными, упорядоченными по-разному: очевидно, что для нас, людей, диапазон с ограничениями и периодом тоже самое, что и периодический с ограничениями. Но что касается D, `minmaxed(periodic())` и `periodic(minmaxed())` — не один и тот же тип. Подробнее об этой идее смотрите раздел о Комментировании типов.

TODO Пример с единицами измерений в главе **Примеры**? Он может оказаться великоват...

Шаблоны на шаблонах

Иногда вы знаете, что пользователям вашего кода нужно направлять в ваш шаблон первый список параметров (неопределенной длины), за которым следует второй список. Таким образом, вы можете захотеть написать что-то такое:

```
template MyTemp(A..., B...)
{ ... }
```

Увы, два кортежа параметров шаблона не имеют смысла (смотрите раздел Объявления шаблона). Но это — не тупик. Во-первых, вы могли бы попытаться написать:

```
template(AB...)
{ ... }
```

И фильтровать `AB`, чтобы найти те, что вам нужны. Это можно сделать посредством `staticFilter`, представленного в разделе Статический фильтр (и связанной с ним функцией на аргументах функции будет `tupleFilter`, показанная в разделе Фильтрация кортежей). В этом случае, две серии аргументов можно полностью смешать (и не разделять в двух списках), это даже мощнее. С другой стороны, вы должны быть в состоянии разделять их только по типам, что может оказаться невозможным.

Шаблоны на пути вниз

К счастью, исходная задача легко может быть решена следующим образом:

```
template MyTemp(A...)
{
    template MyTemp(B...)
    {
    (...)
    }
}
```

Помните, шаблон может иметь шаблоны-члены. Итак, вы можете вложить шаблоны внутрь шаблонов, каждый со своими собственными ограничениями и промежуточным кодом. Вы можете использовать одинаковое имя снова и снова, активизируя одноимённый трюк, но иногда использование отличающегося имени на каждом этапе может сделать ваш код более удобочитаемым.

Например, что если вы хотите сравнить два кортежа, чтобы посмотреть, содержат ли они одни и те же аргументы?

```
module compare;

template Compare(First...)
{
    template With(Second...)
    {
        static if (First.length != Second.length)
            enum With = false;
        else static if (First.length == 0) // Конец сравнения
            enum With = true;
        else static if (!is(First[0] == Second[0]))
            enum With = false;
        else
            enum With = Compare!(First[1..$]).With!(Second[1..$]);
    }
}

//Использование:
unittest
{
    alias Compare!(int, double, string).With!(int, double, char) C;
    static assert(C == false);
}
```

В этом случае, использование `With` внутри `Compare` позволяет добиться, чтобы код был совсем легким для использования. Обратите внимание, что одноимённый трюк применяется только в `With`, потому что нам нужен результат этого внутреннего шаблона.

Возвращаясь к `MyTemp`, его использование немного сложнее:

```
1  // Да:
2  alias MyTemp!(int, double, string) MyTemp1;
3  MyTemp1!(char, void);
4
5  // Нет:
6  MyTemp!(int, double, string)!(char, void);
7  // Нет:
8  MyTemp!(int, double, string).!(char, void);
```

Грамматика D не разрешает множественные вызовы шаблонов, подобные тем, что на строках 6 и 8. Вы должны использовать псевдоним для промежуточного этапа. Это не такое уж резкое ограничение, поскольку если вы создали шаблон как многоступенчатую конструкцию, это, скорее всего, потому, что вы хотели сделать многоступенчатый вызов...

Двухступенчатые шаблоны функций

Для шаблонов функций это может дать очень мощные вещи. У вас может быть место в вашем коде, где принимаются параметры времени компиляции, которые поставляет другой, созданный-только-для-ваших-нужд, шаблон, экземпляр которого вы можете создать позже. Смотрите, например, Интерполяция Строк и функция-в-шаблоне `interpolate`.

Политики особенно хороши с этой идиомой. Раздел Мемоизация функции представляет шаблон, который превращает стандартную D-функцию в мемоизированную. Вот что можно было бы сделать, если это был двухступенчатый шаблон:

```
/*
 * memoizer будет сохранять первый миллион кортежей аргументов
 * и выкидывать половину из них при достижении максимума для
 * освобождения памяти. На этом этапе мемоизируемая функция не известна.
```

```
 * Пользователь решает, что для этого конкретного экземпляра
 * мемоизация была бы наилучшей стратегией в его коде.
 * memoizer может быть (и будет!) применён во многих различных функциях.
 */
alias memoize!( Storing.maximum
              , 1_000_000
              , Discarding.fraction
              , 0.5f
              ) memoizer;

auto longCalculation1(int i, double d) { ... }
auto longCalculation2(double d, string s) { ... }

/*
 * И longCalculation1, и longCalculation2 будут получать пользу
 * от мемоизации, даже если они имеют разные сигнатуры, разные
 * аргументы, и возвращают различный тип.
 */
alias memoizer!longCalculation1 mlc1;
alias memoizer!longCalculation2 mlc2;
```

TODO OK, теперь, может быть, я должен обеспечить скорректированную версию memoize в Мемоизации функции.

Кортежи с именованными полями

Позвольте привести другой пример, чтобы использовать IFTI. В Phobos, функция std.typecons.tuple позволяет вам создавать кортеж на лету. Это очень хороший пример IFTI в действии:

```
module usingtuple;
import std.typecons;

void main()
{
    // tuple1 - это Tuple!(int,double,string)
    auto tuple1 = tuple(1, 3.14159, "abc");
    // tuple2 - это Tuple!(char,char,char)
    auto tuple2 = tuple('a','b','c');
}
```

Но кортеж в Phobos мощнее. У него можно именовать параметры:

```
Tuple!(int, "counter", string, "name") myCounter;
myCounter.counter = -1;
myCounter.name = "The One and Only Counter Around There";
```

Согласно записанному, Phobos не обеспечивает фабричную функцию кортежа, допускающую именованные аргументы хорошим, автоматизированным способом. Вот что мне хотелось бы:

```
module usingnamedtuple;
import namedtuple;

void main()
{
    auto myCounter = tuple!("counter", "name")
                           (-1, "Who's his Daddy's counter?");

    myCounter.counter = 1;

    // Или даже:
    alias tuple!("counter", "name") makeCounter;
```

```
    auto c1 = makeCounter(0, "I count ints!");
    auto c2 = makeCounter("Hello", "I'm a strange one, using strings");

    c2.counter ~= " World!";
}
```

В предыдущем примере, `c1` — это `Tuple!(int, string)`, в то время, как `c2` — это `Tuple!(string, string)`. Это означает что `makeCounter` — фабричная функция для кортежей с двумя полями `counter` и `name`, которые увидят свои типы при определении позже. Я хочу это, так что давайте напишем код.

Сперва, очевидно, что нам нужен двухступенчатый шаблон:

```
import std.typetuple: allSatisfy;

template tuple(names...)
if (names.length && allSatisfy!(isAStringLiteral, names))
{
    auto tuple(T...)(T args)
    {
    (...)
    }
}
```

Ограничение здесь добавлено, чтобы проверить, что пользователь передает по крайней мере одно имя, а также, что все переданные имена являются на самом деле литералами строки. Я использую std.typetuple.allSatisfy, чтобы проверить условие на них всех. Я не могу использовать непосредственно std.traits.isSomeString, поскольку этот шаблон действует на типах, в то время как мне нужно что-то для проверки строкового литерала.

```
module isastringliteral;
import std.traits: isSomeString;

template isAStringLiteral(alias name)
{
    enum isAStringLiteral = isSomeString!(typeof(name));
}
```

В этом месте нам нужно создать правильный кортеж. Имена аргументов предоставляются сразу после каждого типа (с тем, чтобы позволить смешивание между именованными и анонимными полями):

```
/*
 * Первые два поля и четвертое именованы. Третье анонимное.
 */
alias Tuple!( int,    "counter"
            , string, "name"
            , double  /* Анонимное */
            , double, "total"
            ) MyTuple;
```

Для нашей функции мы считаем, что при передаче *n* имен первые *n* аргументов будут именованы, и остальные (если есть), будут анонимными. Это дает нам другое ограничение: если пользователь предоставляет меньше *n* аргументов, мы отклоняем ввод и останавливаем в этом месте компиляцию.

```
import std.typetuple: allSatisfy;

template tuple(names...)
if (names.length && allSatisfy!(isAStringLiteral, names))
{
    auto tuple(T...)(T args) if (T.length >= names.length)
    {
    (...)
    }
}
```

Теперь, нам просто нужны альтернативные имена и типы аргументов. К счастью, этот документ описывает шаблон `Interleave` в разделе Чередование типов, который делает это просто:

```
module namedtuple;
import std.typecons;
import std.typetuple;
import interleave;
import isastringliteral;

template tuple(names...)
if (names.length && allSatisfy!(isAStringLiteral, names))
{
    auto tuple(T...)(T args) if (T.length >= names.length)
    {
        return Tuple!(Interleave!(T).With!(names))(args);
    }
}
```

И, вуаля, здесь у нас есть фабричная функция именованных кортежей. Разве это не приятно? Пример замыкания в разделе Замыкания — Объекты для бедных мог бы использовать это для упрощения его возвращаемой величины.

TODO И каррированный шаблон.

__FILE__ и __LINE__

Незакончено Этот раздел нуждается в серьёзном тестировании. Моя конфигурация D была испорчена, когда я писал эту часть. Так что берите всё, что в тут есть, с большим скептицизмом. В моем списке TODO всё это проверить и переписать.

В разделе Значения по умолчанию мы видели, что параметры шаблона могут иметь значение по умолчанию. Есть также два специальных, резервных, идентификатора, которые определены в D: `__FILE__` и `__LINE__`. Они используются в стандартных шаблонах (не-`mixin`), но своим поведением напоминают mixin'ы: при создании экземпляра, они заменяются строками, содержащими имя файла и номер строки в файле *места вызова экземпляра*. Да, это — своего рода двусторонний диалог: модуль `a.d` определяет шаблон `T`. Модуль `b.d` требует экземпляр `T`. Этот экземпляр создаётся в модуле `a.d`, но номер строки и имя файла берутся из `b.d`!

По большей части их объявляют как-то так:

```
module filelinetagging;

struct Unique(T, string file, size_t line)
{
```

```
    enum size_t l = line;
    enum string f = file;
    T t;
}

auto unique(T, string file = __FILE__, size_t line = __LINE__)(T t)
{
    return Unique!(T, file, line)(t);
}
```

Как подсказывает имя `Unique`, это способ получить уникальные экземпляры. За исключением случаев, когда вы вызываете этот же шаблон дважды в той же строке вашего файла, это в значительной степени гарантирует, что ваш экземпляр будет единственным. (*Т.е. если мы в цикле создаём несколько экземпляров, их тип уже не будет уникальным — прим. пер.*) Помните, что аргументы шаблона становятся частью имени области видимости шаблона при Создании экземпляра шаблона.

```
module usingunique;
import filelinetagging;

void main()
{
    auto u = unique(1); // Unique!(int, "thefile.d", 4)

    auto v = unique(1); // Unique!(int, "thefile.d", 6)

    static assert(!is( typeof(v) == typeof(u) ));
}
```

Даже если `u` и `v` объявлены одним и тем же способом, у них будут разные типы.

За исключением единственных-в-своем-роде типов, это также полезно для отладки: вы можете использовать строки в сообщениях об ошибках:

```
auto flatten(Range, file == __FILE__, line == __LINE__)(Range range)
{
    static if (rank!Range == 0)
        static assert(0, "Файл: " ~ file ~ " в строке: " ~ line
                       ~ ", flatten вызвана с типом ранга 0: "
                       ~ Range.stringof);
    else static if (rank!Range == 1)
        return range;
    else
        return Flatten!(Range)(range);
}
```

И вот небольшой подарок:

```
module debugger;

/** Использование:
 * Debug!(templateToBeTested).With!(Argument0, Argument1, Argument2);
 */
template Debug(alias toTest, string file = __FILE__, size_t line = __LINE__)
{
    template With(Args...)
    {
        static if (is( toTest!Args ))
            alias toTest!Args With;
        else
            static assert(0, "Ошибка: " ~ to!string(toTest)
                           ~ " вызван с аргументами: "
```

```
                                ~ Args.stringof);
    }
}
```

Таким образом, нет необходимости изменять ваши красивые шаблоны.

TODO Протестировать это.

Вокруг Шаблонов: Другие инструменты времени компиляции

Существует еще несколько средств метапрограммирования времени компиляции в D, *не только* шаблоны. Эта глава описывает наиболее распространенные инструменты: Строки mixin, Вычисление функций времени компиляции (CTFE) и __traits, рассматривая их в связи с шаблонами. Хорошие новости: их все можно совмещать. Строки mixin замечательно внедряют код в ваши шаблоны, вычисляемые-во-время-компиляции функции могут выступать в качестве параметров шаблона и сами могут быть шаблонными. И, лучшее из лучших, шаблонные функции времени компиляции могут возвращать строки, которые в свою очередь можно примешивать в ваши шаблоны. Подходите и смотрите, это забавно!

К сожалению, эта глава кажется недописанной. Количество вставок, озаглавленных как ***TODO*** *и* ***Незакончено*** *слишком много. С другой стороны, она, вроде как и не о шаблонах — замечание пер.*

Строки Mixin

Строки mixin размещают код D там, где их вызывают, просто перед компиляцией. После внедрения, код является *настоящим кодом* D, подобно любому другому. Кодом манипулируют как строками, отсюда и название.

Синтаксис

Синтаксис немного отличается от Шаблонов mixin:

```
mixin("немного кода в виде string");
```

Постарайтесь не забыть о скобках. Строки mixin являются средством исключительно времени компиляции, поэтому строка также должна определяться на этапе компиляции.

Примешивание кода, с использованием шаблонов

Конечно, просто внедрение наперёд заданного кода кажется немного скучным:

```
mixin("int i = 3;"); // Не забывайте про две точки с запятой:
                     // Одна внутри примешанного кода,
                     // другая после вызова mixin().
i++;
assert(i == 4);
```

В этом нет никакого интереса по сравнению с непосредственным написанием стандартного кода D. Веселье начинается с мощной способностью свёртки констант D: в D, строки можно конкатенировать (складывать) во время компиляции. Вот где строки mixin встречаются с шаблонами: шаблоны могут производить строки во время компиляции и могут получать эти строки в виде параметров. Вы уже видели это в разделах Перегрузка операторов и opDispatch, так как я не мог удержаться, и говорил об этом заранее.

Теперь, представьте себе для примера, что нам нужен шаблон, который генерирует структуры. Вы хотите иметь возможность называть структуры, как вы пожелаете. Скажем, мы хотели бы использовать его как-то так:

```
module usingmystruct1;
import mystruct1;

mixin(MyStruct!"First");  // создаём новый тип, структуру, с именем First
mixin(MyStruct!"Second"); // и другой, с именем Second

void main()
{
    // "First" code was injected right there, in module 'mine'.
    First f1, f2;
    Second s1;

    assert(is( typeof(f1) == First));
}
```

А вот и генерирующий код:

```
module mystruct1;

template MyStruct(string name)
{
    enum string MyStruct = "struct " ~ name
                         ~ " { "
                         ~ "/+ некоторый код +/"
                         ~ " }";
}

// Например, если name == "First", он вернёт
//   "struct First { /+ некоторый код +/ }"
//
```

В этом случае, строка собирается внутри шаблона во время создания экземпляра, выводится наружу через одноимённый трюк, и затем примешивается там, где вам это нужно. Заметьте, что строка генерируется в модуле, содержащем `MyStruct`, но, при этом имена `First` и `Second` определяются в месте вызова **`mixin`**`()`. Если вы используете mixin в различных модулях, это приведет к определению множества различных структур, и все будут называться одинаково. Это может быть именно тем, что вам нужно, а может, и нет.

Чтобы получить один и тот же тип структуры в различных модулях, код нужно организовать немного иначе: структура должна быть сгенерирована в модуле с шаблоном.

```
module mystruct2;

template MyStruct(string name)
{
    alias MyStructImpl!(name).result MyStruct;
}

template MyStructImpl(string name)
{
    enum string code = "struct " ~ name
                     ~ " { "
                     ~ "/+ some code +/"
                     ~ " }";
    mixin(code);
    mixin("alias " ~ name ~ " result;");
}

module usingmystruct2;
import mystruct2;
```

```
MyStruct!"First" f1, f2;
MyStruct!"Second" s1;
```

Использование отличается, как вы можете видеть. В этом случае, `First` генерируется внутри `MyStructImpl` и выводится наружу через псевдоним (данное выражение с псевдонимом — само порождается строкой Mixin). На самом деле, весь код можно поместить в Mixin:

```
module mystruct3;

template MyStruct(string name)
{
    alias MyStructImpl!(name).result MyStruct;
}

template MyStructImpl(string name)
{
mixin("struct " ~ name
    ~ " {"
    ~ "/* some code */"
    ~ " }\n"
    ~ "alias " ~ name ~ " result;");
}
```

Вот пример, использующий тернарный оператор `?:`, выполняющий некий выбор кода во время компиляции, подобно тому, что можно сделать с помощью **`static if`** (**Static If**):

```
module getset2;

enum GetSet { no, yes}

struct S(GetSet getset = GetSet.no, T)
{
    enum priv = "private T value;\n"
              ~ "T get() @property { return value;}\n"
              ~ "void set(T _value) { value = _value;}";

    enum pub = "T value;";

    mixin( (getset == GetSet.yes) ? priv : pub);
}
```

Код:

```
import getset2;

void main()
{
    S!(GetSet.yes, int) gs;
    gs.set(1);
    assert( gs.get == 1);
}
```

сгенерирует:

```
struct S!(GetSet.yes, int)
{
    private int value;
    int get() @property { return value;}
    void set(int _value) { value = _value;}
}
```

Пределы

Создавать код всё ещё немного неудобно, поскольку я пока не ввел CTFE (смотрите CTFE). Итак, сейчас мы ограничены простой конкатенацией: выполнение цикла, например, возможно с шаблонами, но значительно проще с CTFE. Даже в этом случае, оно уже удивительно мощное: вы можете формировать код D с некоторыми «отверстиями» (вместо типов, имён, чего угодно), которые будут заполнены при создании экземпляра шаблона, и затем примешивать куда-нибудь ещё. Вы можете создать любой другой тип кода D с помощью этого.

Вы можете размещать выражения `mixin`() почти в любое место, куда вы хотите, но …

TODO Проверить ограничения: внутри выражений static if, например

Избегание строк. Одна обычная проблема с манипуляцией кодом D, как строкой — как иметь дело со строками в коде? Вы должны избегать их. Либо используйте \", чтобы создавать кавычки в строках, нечто подобное было сделано в разделе Синтаксис Шаблона функций, чтобы сгенерировать сообщение об ошибке. Или вы можете размещать строки между q{ и }, или между обратными кавычками.

Вычисление функции времени компиляции

Вычисление во время компиляции

Вычисление функции времени компиляции (Compile-Time Function Evaluation, с этого места CTFE) — это расширение свёртки констант, которое выполняется во время компиляции в коде D: если вы можете вычислить `1+2+3*4` во время компиляции, почему не распространить это на вычисление целой функции? С этого места я буду называть вычисляемые во время компиляции функции CTE-функциями.

Это очень горячая тема в D прямо сейчас и референсный компилятор продвинулся очень быстро в 2011 и 2012 годах. Пределы того, что можно делать с помощью CTE-функций отодвигаются дальше с каждой новой версией. Все утверждения `foreach`, `while`, `if`/`else`, манипуляции с массивами, структурами, функциями — всё есть. Даже арифметика указателей возможна! Когда я начинал этот документ (DMD 2.055), ограничения были по большей части: нет классов и нет исключений. Это было изменено в версии DMD 2.057, допускающей обработку классов во время компиляции. Затем в DMD 2.058-2.061 уничтожили огромное количество ошибок, связанных с CTFE.

Фактически опасность лежит в другом месте: легко забыть, что CTE-функции должны также быть стандартными, времени выполнения, функциями. Помните, что некоторые действия имеют смысл только во время компиляции, или с константами, инициализированными в время компиляции: индексирование в кортежах, например:

__ctfe

Незакончено Написать что-то об этой новой функциональности, которая позволяет проводить тестирование внутри функции, находимся ли мы во время компиляции, или во время выполнения.

Шаблоны и CTFE

Незакончено Надо добавить несколько сочных примеров.

Это означает: Вы можете скармливать константы времени компиляции вашей классической D-функции и код будет вычисляться во время компиляции. Что касается шаблонов, то это означает, что возвращаемые значения функции можно использовать в качестве параметров шаблона или как инициализацию для `enum`:

```
/* */
```

Шаблоны функций прекрасно могут вызывать функции, вычисляемые во время компиляции:

Шаблоны с CTFE и Строками Mixin, ох!

И фейерверк — когда вы смешиваете (!) это со строками mixin: код можно сгенерировать функциями, дающими доступ к почти полному языку D для его изготовления. Этот код можно смешивать в шаблонах, чтобы производить то, что вам нужно. И, для того, чтобы закрыть цикл: функция, возвращающая код-как-строку, сама может быть шаблоном, используя параметры другого шаблона как собственные параметры.

Конкретно, здесь — код из раздела Getters и Setters, переделанный:

```
module getset3;
import std.conv;

enum GetSet { no, yes}

string priv(string type, string index)
{
    return
    "private "~type~" value"~index~";\n"
  ~ type~" get"~index~"() @property { return value"~index~";}\n"
  ~ "void set"~index~"("~type~" _value) { value"~index~" = _value;}";
}

string pub(string type, string index)
{
    return type ~ "value" ~ index ~ ";";
}

string GenerateS(GetSet getset = GetSet.no, T...)()
{
    string result;
    foreach(index, Type; T)
        static if (getset == GetSet.yes)
             result ~= priv(Type.stringof, to!string(index));
        else
            result ~= pub(Type.stringof, to!string(index));
    return result;
}

struct S(GetSet getset = GetSet.no, T...)
{
    mixin(GenerateS!(getset,T));
}

void main()
{
```

```
    S!(GetSet.yes, int, string, int) gs;

/* Генерирует:
struct S!(GetSet.yes, int, string, int)
{
    private int value0;
    int get0() @property { return value0;}
    void set0(int _value) { value0 = _value;}

    private string value1;
    string get1() @property { return value1;}
    void set1(string _value) { value1 = _value;}

    private int value2;
    int get2() @property { return value2;}
    void set2(int _value) { value2 = _value;}
}
*/

    gs.set1("abc");
    assert(gs.get1 == "abc");
}
```

Этот код мощнее, чем тот, что мы видели прежде: количество типов варьируется, и весь набор механизмов getters/setters генерируется по запросу. Всё это сделано посредством простого подключения вместе функций, возвращающих `string`, и небольшого цикла посредством **`foreach`** времени компиляции.

Простое заполнение строк

Вся эта игра с оператором конкатенации (~) становится обузой. Мы должны написать функцию заполнения (interpolation) строки, конечно, рассчитываемую во время компиляции, чтобы помочь нам в нашей задаче. Вот как бы мне хотелось это использовать:

```
import stringinterpolation;

alias interpolate!"struct #0 { #1 value; #0[#2] children;}" makeTree;

enum string intTree = makeTree("IntTree", "int", 2);
enum string doubleTree = makeTree("DoubleTree", "double", "");

static assert(intTree
          == "struct IntTree { int value; IntTree[2] children;}");
static assert(doubleTree
          == "struct DoubleTree { double value; DoubleTree[] children;}");
```

Как вы можете видеть, строка, которую нужно заполнить, передаётся как параметр шаблона. Меткой-заполнителем выберем символ, обычно не используемый в D-коде: #. Энный параметр — #n, начиная с 0. В качестве уступки практичности, одинокий # считается эквивалентным #0. Аргументы, которые будут помещаться в строку, передаются как стандартные параметры (не-шаблонные) и могут быть любого типа.

```
module stringinterpolation;
import std.conv;

template interpolate(string code)
{
    string interpolate(Args...)(Args args) {
        string[] stringified;
        foreach(index, arg; args) stringified ~= to!string(arg);
```

```
        string result;
        int i;
        int zero = to!int('0');

        while (i < code.length) {
            if (code[i] == '#') {
                int j = 1;
                int index;
                while (i+j < code.length
                    && to!int(code[i+j])-zero >= 0
                    && to!int(code[i+j])-zero <= 9)
                {
                    index = index*10 + to!int(code[i+j])-zero;
                    ++j;
                }

                result ~= stringified[index];
                i += j;
            }
            else {
                result ~= code[i];
                ++i;
            }
        }

        return result;
    }
}
```

TODO Синтаксис может быть немного расширен: вставить множественные строки, вставить диапазон строк, все аргументы к концу.

Пример: расширение std.functional.binaryFun

Незакончено Это дорого моему сердцу. Отображение n диапазонов в параллель — одна из первых вещей, которую я захотел сделать с диапазонами, для примера создавать диапазоны структур с конструктором, принимающим более одного параметра.

Phobos имеет два действительно интересных шаблона: std.functional.unaryFun и std.functional.binaryFun.

Так как автор не озаботился объяснить, что делают эти шаблоны

TODO Объяснить, что целью является расширение их на случай функций n-аргументов.

```
bool isaz(char c) {
    return c >= 'a' && c <= 'z';
}

bool isAZ(char c) {
    return c >= 'A' && c <= 'Z';
}

bool isNotLetter(char c) {
    return !isaz(c) && !isAZ(c);
}

int letternum(char c) {
    return to!int(c) - to!int('a') + 1;
}
```

```
//  arity — количество аргументов — прим. пер.
int arity(string s) {
    if (s.length == 0) return 0;

    int arity;
    string padded = " " ~ s ~ " ";
    foreach(i, c; padded[0..$-2])
        if (isaz(padded[i+1])
         && isNotLetter(padded[i])
         && isNotLetter(padded[i+2]))
            arity = letternum(padded[i+1]) > arity ?
                    letternum(padded[i+1])
                  : arity;
    return arity;
}

// возвращает строку вида "A, B, C" - прим. пер.
string templateTypes(int arit) {
    if (arit == 0) return "";
    if (arit == 1) return "A";

    string result;
    foreach(i; 0..arit)
        result ~= "ABCDEFGHIJKLMNOPQRSTUVWXYZ"[i] ~ ", ";

    return result[0..$-2];
}

// возвращает строку вида "A a, B b, C c" - прим. пер.
string params(int arit) {
    if (arit == 0) return "";
    if (arit == 1) return "A a";

    string result;
    foreach(i; 0..arit)
        result ~= "ABCDEFGHIJKLMNOPQRSTUVWXYZ"[i]
               ~ " " ~ "abcdefghijklmnopqrstuvwxyz"[i]
               ~ ", ";

    return result[0..$-2];
}

string naryFunBody(string code, int arit) {
    return interpolate!"auto ref naryFun(#0)(#1) { return #2;}"
                       (templateTypes(arit), params(arit), code);
}

// Срабатывает одноимённый трюк, для этого идентификатор
// naryFun присутствует в строке,
// возвращаемой функцией naryFunBody — прим. пер.
template naryFun(string code, int arit = arity(code))
{
    mixin(naryFunBody(code, arit));
}
```

Автору не мешало бы немного объяснить, как всё это работает. Я, в конце-концов, вроде разобрался, но времени на это ушло прилично. Комментарии перед функциями и шаблоном мои — прим. пер.

Сортирующие сети

Сортирующие сети является хорошим примером того, что генерация кода во время

компиляции может подкупить вас. Сортировка является очень обширной темой, и получение близкого к оптимальному вида во многих случаях довольно сложно. Но в некоторых случаях, если вы знаете вашу входную длину, вы можете сформировать предопределенный список сравнений между элементами, и поменять их местами если они не находятся в правильном порядке. Здесь, я представлю сортирующие сети как пары индексов во входном диапазоне: принимаем индексы `i` и `j`, сравниваем `input[i]` и `input[j]`, и так далее.

Например, дан массив или диапазон с произвольным доступом (random-access range) длины `n`, в следующей таблице представлены возможные (оптимальные, в данном случае) сортирующие сети.

Таблица 7. Первые сортирующие сети

n	Сортирующие сети
2	[[0, 1]]
3	[[0, 2], [0, 1], [1, 2]]
4	[[0, 2], [1, 3], [0, 1], [2, 3], [1, 2]]
5	[[0, 4], [0, 2], [1, 3], [2, 4], [0, 1], [2, 3], [1, 4], [1, 2], [3, 4]]

Код генерации списка индексов можно найти в книге Кнута Искусство Компьютерного Программирования или в сети. Код, данный внизу, я перевёл из Лиспа (!) на D, взяв из Doug Hoyte's Let Over Lambda, очень занудной книги о макросах в Common Lisp.

```
module sortingnetwork;

int ceilLog2(int n)
{
    int i;
    if ((n & (n-1)) == 0) i = -1;
    while (n > 0) { ++i; n/= 2;}
    return i;
}

/**
 * Принимает длину n, возвращает массив пар индексов,
 * соответствующий сортирующей сети для длины n.
 * Выглядит как код на C, не правда ли?
 */
int[2][] sortingNetwork(int n)
{
    int[2][] network;
    auto t = ceilLog2(n);
    auto p = 1 << (t-1);
    while (p > 0)
    {
        auto q = 1 << (t-1);
        auto r = 0;
        auto d = p;
        while (d > 0)
        {
            for(int i = 0; i<=(n-d-1); ++i)
            {
                if (r == (i & p)) network ~= [i, i+d];
            }
            d = q-p;
            q /= 2;
```

```
                r = p;
            }
            p /= 2;
        }
        return network;
}
```

`sortingNetwork` возвращает массив пар индексов. Из этого должно быть просто сгенерировать код. Основной строительный блок сделан через шаблон `interpolate` (смотрите **Простое заполнение строк**):

```
module buildsortingcode;
import stringinterpolation;
import sortingnetwork;

string buildSortingCode(size_t l)()
{
    enum network = sortingNetwork!(l);
    string result;
    foreach(elem; network) result ~=
        interpolate!(
        "t1 = input[#0];
         t2 = input[#1];
         if (!binaryFun!pred(t1, t2))
         {
             auto temp = t2;
             input[#1] = t1;
             input[#0] = temp;
         }\n")(elem[0], elem[1]);
    return result;
}
```

И отсюда я хочу получить шаблон, который заранее генерирует шаблонную функцию сортирующей сети. Как и для std.algorithm.sorting.sort, предикат `pred` определяет способ сравнения индивидуальных элементов диапазона.

```
module networksort;
import std.range;
import std.functional;
import std.exception;
import buildsortingcode;
import stringinterpolation;

template networkSort(size_t l)
{
    mixin(
interpolate!(
    "void networkSort(alias pred = \"a < b\", R)(ref R input)
     if (isRandomAccessRange!R)
     {
         enforce(input.length >= #,
                 \"Calling networkSort!# with a range of less than # elements\");
          ElementType!R t1, t2;")(l)
      ~ buildSortingCode!(l)
 ~ "}");
}
```

Странная конструкция mixin-в-шаблоне должна резать шаблон на два, чтобы позволить пользовательскому коду нечто подобное:

```
// где-то в вашем коде
alias networkSort!32 sort32;
```

```
// где-то ещё
sort32(myArr);
sort32!"a > b"(myArr);
```

sort32 предназначен работать с диапазонами с произвольным доступом длиной 32-элемента, не зная заранее ни природу диапазона, ни используемый для сортировки предикат. Если вы вызываете его на диапазоне с длиной больше, чем 32, он отсортирует только первые 32 элемента. В случае меньше 32 элементов, выполнение потерпит неудачу.

Итак, чем же эта маленькая сортирующая процедура нас подкупает? Она оказывается очень эффективной для массивов небольшого размера, по сравнению с std.algorithm.sorting.sort, но заметно, что производительность падает с определённого момента. Следующая таблица сравнивает один миллион сортировок n-элементых произвольно-перемешанных массивов, выполненных с помощью `networkSort` и `std.algorithm.sort`, и дает коэффициент прироста скорости приведенного пред-вычисленного сортирующего кода.

Таблица 8. Сравнение `networkSort` и `std.algorithm.sort`

n	Sorting network (ms)	Standard sort (ms)	коэффициент
5	324	532	1.642
10	555	1096	1.975
10	803	1679	2.091
20	1154	2314	2.005
25	1538	3244	2.109
30	2173	3508	1.614
35	4075	4120	1.011
40	5918	5269	0.890
45	7479	5959	0.797
50	9179	6435	0.701

Так, по крайней мере в этом сравнении, `networkSort` превосходит Phobos по скорости вплоть до 100% для диапазонов между 5 и 40 элементами.[11] Конечно, сортировка `sort` в Phobos более универсальна, поскольку она работает на диапазонах с длиной, известной только во время выполнения, но если вы знаете вашу входную длину заранее,

11 Если кому-то интересно, в моем компьютере граница оказалась в районе 38-39 элементов.

`networkSort` может быть хороша для использования.

В любом случае, стоит вооружиться идеей о том, что если вы хорошо представляете себе, на что будут похожи ваши данные во время выполнения, вы, вероятно, можете заранее для них сгенерировать оптимальный код во время компиляции, а затем выполнить сортировку во время выполнения.

`__traits` (Признаки)

Общий синтаксис `__traits` можно найти здесь. Признаки являются по сути другим инструментом интроспекции (самоанализа) времени компиляции, сопряженным с выражением `is` (смотри Приложение). Большую часть времени, `__traits` будет возвращать ответы `true` или `false` на простые вопросы интроспекции о типе (это тип, или идентификатор абстрактного класса, или финализированная функция?). Как в D это часто бывает, иногда эти вопросы вы могли бы задать, используя `is` или ограничения шаблона, но иногда нет. Интересно то, что вы можете делать некоторую интроспекцию не только на типах, но также на идентификаторах или выражениях.

Да/Нет вопросы с __traits

Учитывая, что этот документ о шаблонах, и что мы уже видели множество инструментов интроспекции, здесь — краткий список вопросов да/нет, которые вы можете задать, о том, что можно или нельзя выяснить с помощью `is`:[12]

Таблица 9. Сравнение между __traits и другими инструментами интроспекции

Вопрос	Выполнимо с другими инструментами?
isArithmetic	Да
isAssociativeArray	Да
isFloating	Да
isIntegral	Да
isScalar	Да
isStaticArray	Да
isUnsigned	Да
isAbstractClass	Нет
isFinalClass	Нет
isVirtualFunction	Нет
isAbstractFunction	Нет
isFinalFunction	Нет
isStaticFunction	Нет
isRef	Нет
isOut	Нет

12 Как и для любого другого утверждения в этом документе, читатели могут доказать мне, что я не прав. Это должно быть не слишком трудно.

isLazy	Нет
hasMember	Нет (Да?)
isSame	Нет
compiles	Да (в некотором роде)

Все они могут быть весьма полезны в вашем коде, но я проигнорирую их, так как они не сильно связаны с шаблонами. Более интересным, по-моему, будет использование `__traits` для получения новой информации о типе. Это действительно отличается от других инструментов интроспекции, и я буду иметь дело с ним более подробно прямо сейчас.

identifier

`identifier` дает вам его имя идентификатора в виде строки. Это довольно интересно, поскольку некоторые идентификаторы я бы назвал активными: например, если `foo` — это функция, то `foo.stringof` попытается сначала выполнить `foo` и тогда `.stringof` потерпит неудачу. Кроме того, `.stringof`, хотя и в высшей степени полезна, иногда возвращает странно отформатированые строки. `identifier` лучше себя ведёт.

Давайте вернёмся к одному из самых первых шаблонов в этом документе, `nameOf` здесь. Первоначально это было написано так:

```
template nameOf(alias a)
{
    enum string name = a.stringof; // enum: объявление константы,
                                   // определяемой во время компиляции
}
```

Но это не сработает для функций:

```
int foo(int i, int j) { return i+j;}

auto name = nameOf!foo; // Error, 'foo' must be called with 2 arguments
                        // Ошибка, 'foo' должна вызываться с 2 аргументами
```

Так что значительно лучше использовать `__traits` (также тут Одноимённый трюк):

```
module nameof;

template nameOf(alias a)
{
    enum string nameOf = __traits(identifier, a);
}

unittest
{
    int foo(int i, int j) { return i+j;}
    enum name = nameOf!foo; // name доступна во время компиляции

    assert(name == "foo");
}
```

Заметьте, что это работает для множества (всех?) типов идентификаторов: имён шаблонов, имён классов, даже модулей:

```
import std.typecons;
import nameof;
```

```
enum name2 = nameOf!(nameOf); // "nameOf(alias a)"
enum name3 = nameOf!(std.typecons); //"typecons"
```

getMember

В двух словах, `__traits(getMember, name, "member")` даст вам прямой доступ к `name.member`. Это реальный член: вы можете получить его величину (если имеется), задать её заново, и т.п.. Любая конструкция D, содержащая внутренние члены, подойдёт для `name`. Если вы удивляетесь почему агрегат вызывается непосредственно своим собственным именем, в то время как для имени его члена нужна строка, это потому, что агрегат является действительным идентификатором (он существует самостоятельно), а имя его члена не существует за пределами агрегата (или даже хуже, оно может ссылаться на другую, не связанную, конструкцию).

> **Агрегаты.** Я использую слово «агрегат» в качестве всеохватного термина для любой конструкции D, которая содержит члены. Структуры и классы являются очевидными агрегатами, как и интерфейсы, но интересно иметь в виду, что шаблоны тоже могут иметь члены (помните раздел Создание экземпляра Шаблона? Шаблон является именованной, параметризованной областью видимости). Итак, все вызовы `__traits`, показанные в этом разделе можно использовать на шаблонах. Это нужно иметь в виду. Ещё интереснее, на мой взгляд, то, что модули тоже агрегаты, даже если они не являются гражданами первого класса в стране D. Вы увидите примеры этого в следующих разделах.

allMembers

Это круто. Получая имя агрегата, `__traits(allMembers, aggregate)` возвращает кортеж строковых литералов, каждый из которых — это имя члена. Для класса члены родительских классов также включаются. Встроенные свойства (такие, как `.sizeof`) не включены в этот список. Заметьте, что я сказал 'кортеж': он несколько мощнее массива, поскольку над ним можно выполнять итерации (обрабатывать в цикле) во время компиляции.

Имена не повторяются для перегруженных членов, но способ получить перегрузки можно увидеть в следующем разделе.

```
module usingallmembers1;

class MyClass
{
    int i;

    this() { i = 0;}
    this(int j) { i = j;}
    ~this() { }

    void foo() { ++i;}
    int foo(int j) { return i+j;}
}

void main()
{
    // Вкладываем в массив для более удобочитаемой для человека печати
    enum myMembers = [__traits(allMembers, MyClass)];

    // Обратите внимание, "i" и "foo" между стандартными членами класса
    // "foo" появляется только один раз, несмотря на перегрузку.
```

```
    assert(myMembers == ["i", "__ctor", "__dtor", "foo", "toString",
                         "toHash", "opCmp", "opEquals", "Monitor", "factory"]);
}
```

Так что этот код является хорошим способом получения членов, как полей (подобно `i`), так и методов (подобно `foo`). Если вас удивляют имена `"__ctor"` и `"__dtor"`, то это внутренние названия конструкторов и деструкторов в D. Но они вполне пригодны в вашем коде! Для структур список гораздо менее загромождённый, поскольку они получают только имена конструктора и деструктора, а также `opAssign`, оператор присваивания (=).

Поскольку эти признаки возвращают строки, их можно непосредственно направить в `getMember`. Немного дальше вы увидите способ получить хороший список всех участников и их типов.

Теперь давайте нарастим всё это немного: как насчет шаблонов классов и структур? Давайте попробуем это:

```
module allmemberstemplate;

class MyClass(T)
{
    T field;
}

static assert([__traits(allMembers, MyClass)] == ["MyClass"]);
```

Ох, что случилось? Помните из Основ, что шаблоны структур, классов и функций являются просто синтаксическим сахаром для «реального» синтаксиса:

```
template MyClass(T)
{
    class MyClass
    {
        T field;
    }
}
```

Так что, `MyClass` — на самом деле имя внешнего шаблона, единственным членом которого является класс `MyClass`. Итак, всё хорошо. Если вы создадите экземпляр шаблона, он будет функционировать так, как вы могли ожидать:

```
module allmemberstemplate2;
import allmemberstemplate;

static assert([__traits(allMembers, MyClass!int)]
           == ["field", "toString", "toHash", "opCmp",
               "opEquals", "Monitor", "factory"]);
```

Если вы помните из раздела Создание экземпляра Шаблона в начале книги, что шаблон является именованной, параметризованной областью видимости, это может вызвать интересную интроспекцию:

```
module templateintrospection;

template Temp(A, B)
{
    A a;
    B foo(A a, B b) { return b;}
    int i;
```

```
    alias A    AType;
    alias A[B] AAType;
}

static assert([__traits(allMembers, Temp)]
           == ["a", "foo", "i", "AType", "AAType"]);
static assert([__traits(allMembers, Temp!(double,string))]
           == ["a","foo", "i", "AType", "AAType"]);
```

Как вы можете видеть, вы получаете также имена псевдонимов. Между прочим, это верно также и для структур с шаблонами.

Другой забавный факт о том, что модули D поддаются вызовам `__traits`, все-же является истиной только для пакетов модулей (то есть, `import pack.mod;` импортирует идентификаторы `pack` и `mod`, но `import mod;` не импортирует ничего).

```
module allmembersmodule;
import std.algorithm;
import std.compiler;

// Огрооомный список имён
enum algos = [__traits(allMembers, std.algorithm)];
// Немного короче
enum compiler = [__traits(allMembers, std.compiler)];

void main()
{
    assert(compiler == ["object", "name",
                        "Vendor", "vendor",
                        "version_major", "version_minor",
                        "D_major", "D_minor"]);
}
```

В предыдущем коде, вы видите, что среди членов есть `"object"` (подразумевается модуль `object.d`, импортируемый во время выполнения), и `"std"`, глобальный пакет, который фигурирует здесь всякий раз, когда вы импортируете `std.*` модуль. Кажется легко представить себе шаблон, который рекурсивно изучает члены, находит модули и пытается рекурсивно входить в них, чтобы получить полное дерево импорта с шаблоном. Увы, `"std"` блокирует это, так как пакет сам по себе не имеет членов.[13]

Вообще-то я и самого `"std"` в списке не вижу — прим. пер.

Интроспекция. Я вполне уверен, авто-интроспекция (модуль, вызывающий `AllMembers` на своём собственном имени) имела обыкновение работать осенью 2011 года. Наступил 2012 год, и это больше не работает. М-да.

На всякий случай, я проверил у себя (компилятор dmd версии 2.069) — у меня работает:

```
module proba4;
import proba4;

void main() {
      enum proba4_str = [__traits(allMembers, proba4)];
      assert(proba4_str ==["object", "proba4", "main"]);
```

13 Если кто-то найдет способ сделать это, я был бы рад увидеть, как это делается!

```
}
```

Замечание про интроспекцию при импорте отдельных модулей (без пакетов) тоже теперь неверно, у меня получилось посмотреть содержимое соседнего модуля без пакета — прим. пер.

Как и для других агрегатов, вы также получаете имена псевдонимов и юниттестов, определённых в модуле.[14] [15]

> **Что такое точка проверки модуля?** Хорошо, сначала это было просто для удовольствия и, чтобы посмотреть, можно ли продублировать модуль или создать структуру с эквивалентным списком членов (всё переадресуется собственным членам модуля). Но реально важным я это посчитал при использовании строк mixin для генерации некоторого типа. Если пользователь использует mixin в своем собственном модуле, он может создать конфликты с уже существующими именами. Поэтому я искал способ, как шаблон mixin может проверить модуль, в котором он в настоящий момент создаёт экземпляр. Затем, я захотел написать шаблон, который, получая имя класса, должен дать мне всю его иерархию (как это видит локальная область видимости модуля, что было достаточно для меня). Этот шаблон `Hierarchy` должен быть показан в этом документе. Затем, при тестировании std.traits.ParameterTypeTuple, я увидел, что он даёт кортеж параметров *одной* функции, даже если она перегружена. Таким образом, проверка модуля — это также способ получить полный список функций с определённым именем и получением кортежа параметров для каждого из них.

TODO Вставить здесь Hierarchy. **TODO** Написать более мощную версию ParameterTypeTuple.

derivedMembers

На самом деле это тоже самое, что и выше, за исключением того, что вы не получите члены родительских классов, а только собственные члены класса.

getOverloads

Дано:

- имя агрегата или экземпляр этого агрегата
- имя члена в виде строки

Тогда, `__traits(getOverloads, name, "member")` даст вам кортеж всех локальных перегрузок `name.member`. Под словом 'локальный', я подразумеваю, что для классов вы не получите перегрузки классов родителей. Есть различие между использованием `getOverloads` на типе и на экземпляре: в первом случае вы получите кортеж всех перегрузок. Хорошо, что используя `"__ctor"`, вы также получите прямой доступ к перегрузкам конструктора типа. Это может быть очень удобно в некоторых случаях.

14 Для любопытных, они называются `__unittestXXX` где `XXX` - число. Их тип является более или менее `void delegate()`.

15 Я не пробовал статические конструкторы в модулях. Не стесняйтесь играть с ними и скажите, что получилось.

Получение всех членов, даже перегруженных

Теперь, если вы в чём-то похожи на меня, у вас появится сильное желание смешать `allMembers` (который возвращает имена членов без перегрузок) и `getOverloads` (который возвращает перегрузки одного члена). Так давайте сделаем это.

Сначала, немного техники: я хочу, чтобы члены описывались по имени и типу. Давайте создадим содержащую их структуру, шаблонную, конечно:

TODO Получение полных имен было бы лучше.

```
module member;

struct Member(string n, T)
{
    enum name = n; // для внешнего доступа
    alias T Type;
}
```

Нам нужен шаблон, который, получая имя члена, предоставит связанную с ним структуру `Member`. Также нам нужен способ вводить имя, это пригодится в дальнейшем:

```
module makemember;
public import member;
import nameof;

template MakeMember(alias member)
{
    mixin( "alias Member!(\""
         ~ nameOf!member
         ~ "\", typeof(member)) MakeMember;");
}

template MakeNamedMember(string name)
{
    template MakeNamedMember(alias member)
    {
        mixin( "alias Member!(\""
             ~ name
             ~ "\", typeof(member)) MakeNamedMember;");
    }
}
```

Теперь, получив имя агрегата и имя члена (в виде строки, поскольку самостоятельно они не существуют), мы хотим вывести список структур `Member`, содержащий всю информацию:

```
module overloads1;
import std.typetuple;
import makemember;

template Overloads(alias a, string member)
{
    alias staticMap!(MakeMember, __traits(getOverloads,a, member))
          Overloads;
}
```

`staticMap` объясняется в разделе Отображение на кортежах типов.

Теперь, это уже работает:

```
module myclass;

class MyClass
{
    int i; // поле
    alias i j; // псевдоним идентификатора

    alias int Int; // псевдоним типа

    struct Inner {} // внутренний тип

    template Temp(T) { alias T Temp;} // шаблон

    this() { i = 0;} // конструктор #1
    this(int j) { i = j;} // конструктор #2
    ~this() { }

    void foo(int j) { ++i;} // перегрузка foo #1
    int foo(int j, int k = 0) { return i+j;} // перегрузка foo #2

    alias foo bar; // псевдоним идентификатора

    unittest
    {
        int i;
    }
}

module usingoverloads1;
import std.stdio;
import overloads1;
import myclass;

void main()
{
    alias Overloads!(MyClass, "foo") o;

    /*
    напечатает:
    foo, с типом: void(int j)
    foo, с типом: int(int j, int k = 0)
    */
    foreach(elem; o)
        writeln(elem.name, ", с типом: ", elem.Type.stringof);
}
```

Мы на самом деле получили два экземпляра `Member`, для каждой перегрузки. Каждая структура `Member` содержит имя `"foo"` и тип перегрузки. Исключение, здесь есть ловушка: для поля, не будет никаких перегрузок. С псевдонимами тоже проблемы. Нам нужно обратить внимание на это в `Overloads`:

```
module overloads2;
import std.typetuple;
import makemember;

/**
 * Получение перегрузок данного члена, в виде кортежа из Member.
 */
template Overloads(alias a, string member)
{
    // a.member — это метод
    static if (__traits(compiles, __traits(getOverloads, a, member))
```

```
        && __traits(getOverloads, a, member).length > 0)
        alias staticMap!(MakeNamedMember!(member), __traits(getOverloads, a,
member))
                Overloads;
    else // поле или псевдоним
    // a.member — это поле или псевдоним идентификатора
        static if (is(typeof(__traits(getMember, a, member))))
            mixin( "alias Member!(\""
                 ~ member
                 ~ "\", typeof(__traits(getMember, a, member))) Overloads;");
    // a.member — это псевдоним типа
    else static if (mixin( "is(Member!(\""
                         ~ member
                         ~ "\", __traits(getMember, a, member)))"))
        mixin( "alias Member!(\""
             ~ member
             ~ "\", __traits(getMember, a, member)) Overloads;");
    // a.member — это шаблон
    else
        mixin( "alias Member!(\""
             ~ member
             ~ "\", void) Overloads;");
}
```

Весь шаблон может казаться немного пугающим, но он должен дать (по большей части) корректный способ обрабатывать большинство видов членов, которые может содержать агрегат: поля, методы, псевдонимы типов, псевдонимы идентификаторов, имена шаблонов. На момент написания статьи, он не верно имеет дело с внутренними типами: я думаю, он должен давать им тип `Outer.Inner`, в то время как здесь формируется только `Inner`.[16] Кроме того, если появятся unittest-блоки, то они выдадут тип `void`, случай по-умолчанию в этом шаблоне. Я думаю, что для них должен получаться тип `void()`.

Последний шаг — это получить все члены данного агрегата. Это совсем просто:

```
1   module allmembers;
2   import overloads2;
3   import std.typetuple;
4
5   template GetOverloads(alias a)
6   {
7       template GetOverloads(string member)
8       {
9           alias Overloads!(a, member) GetOverloads;
10      }
11  }
12
13  template AllMembers(alias a)
14  {
15      alias staticMap!(GetOverloads!(a), __traits(allMembers, a)) AllMembers;
16  }
```

Странная двухступенчатая конструкция `GetOverloads` — это просто способ более простого отображения в строке 15. Так, это довольно долго объяснять, но работает хорошо:

Хм, «долго объяснять»! Ради объяснений подобных моментов люди и читают учебники — недовольство пер.

```
module usingallmembers2;
```

16 Может быть, используя шаблон `qualifiedName`, показанный в разделе родитель.

```
import std.typetuple;
import std.stdio;
import myclass;
import allmembers;

void main()
{
    alias AllMembers!(MyClass) O;
    writeln(O.stringof);

    foreach(o; O)
        writeln(o.name, ", с типом: " ~ o.Type.stringof);

    /*
    напечатает:
    i, с типом: int
    j, с типом: int
    Int, с типом: int
    (...)
    __ctor, с типом: MyClass()
    __ctor, с типом: MyClass(int j)
    (...)
    foo, с типом: void(int j)
    foo, с типом: int(int j, int k = 0)
    (...)
    */
}
```

Это круто, каждый член и его перегрузки принимаются во внимание: тут два конструктора, также деструктор, и, конечно, `i`, с типом `int`.

TODO Пример: сохранить все члены в хэш-таблице или полиморфном ассоциативном списке. Как mixin, поместить внутрь типов для отражений времени выполнения? `(a.send("someMethod", args), a.setInstanceVariable("i",5))`

Тестирование Реализации интерфейса

Предыдущий раздел дал мне способ получить список членов агрегата. Отсюда легкий шаг получить список членов интерфейса и посмотреть, содержат ли члены данного идентификатора весь список членов интерфейса:

```
module implements;
import std.typetuple;
import allmembers;

/**
 * Статическая проверка, реализует ли идентификатор 'a' интерфейс I
 * (то есть, что все члены I присутствуют также в a).
 */
template implements(alias a, I) if (is(I == interface))
{
    alias implementsImpl!(a, AllMembers!I) implements;
}

template implementsImpl(alias a, Items...)
{
    static if (Items.length == 0)
        enum implementsImpl = true;
    else static if (staticIndexOf!(Items[0], AllMembers!a) == -1)
        enum implementsImpl = false;
    else
        enum implementsImpl = implementsImpl!(a, Items[1..$]);
```

```
}

interface I
{
    int foo(int i);
    void foo();

    string toString();
}

class Bad
{
    void foo(int i) {}
}

struct Good
{
    int field;

    int foo(int i) { return i;}
    void foo() { field = 1;}

    string toString() { return "Я хорошая структура!";}
}

unittest
{
    assert( implements!(Good, I));
    assert(!implements!(Bad, I));
}
```

getVirtualFunctions

Это из того же семейства, что и getOverloads и так далее. Даёт вам список виртуальных перегрузок для метода класса. Принимает имя класса, обнаруживает все перегрузки всех полей, даже переопределённых (`override`), оставлю читателю в качестве упражнения.

parent (родитель)

`__traits(parent, symbol)` возвращает идентификатор, являющийся его родителем. Это — родитель *не* в смысле «иерархии классов», здесь речь о полных именах и раздевании на один уровень. Как только вы достигнете верхнего уровня области, будет возвращено имя модуля (это может быть опасно, поскольку сами модули не имеют родителей). Смотрите:

```
module cclass;
import nameof;

class C
{
    int i;
    int foo(int j)
    {
        int k; // k - это "cclass.C.foo.k"
        assert(nameOf!(__traits(parent, k)) == "foo");
        return i+k;
    }
}

module parent;
import nameof;
```

```
import cclass;

// более точно C  - это cclass.C)
static assert(nameOf!(__traits(parent, C)) == "cclass");

void main()
{
    auto c = new C(); // c - это "parent.main.c"
    assert(nameOf!(__traits(parent, c)) == "main");
    assert(nameOf!(__traits(parent, c.i)) == "C");
    c.foo(1);
}
```

Даже если в `__traits` отсутствует `qualifiedIdentifier` (полный идентификатор), мы можем создать шаблон для его получения:

```
module qualifiedname;
import nameof;

template qualifiedName(alias a)
{
    // у него есть родитель?
    static if (__traits(compiles, __traits(parent, a)))
    // Если да, получаем имя и выполняем рекурсию
        enum qualifiedName = qualifiedName!(__traits(parent, a))
                           ~  "." ~ nameOf!(a);
    // если нет, это - имя модуля. Остановка.
    else
        enum qualifiedName = nameOf!a;
}

module usingqualifiedname;
import qualifiedname;
import cclass;

void main()
{
    auto c = new C();
    assert(qualifiedName!c == "usingqualifiedname.main.c");

    assert(qualifiedName!(c.foo) == "cclass.C.foo"); // в обоих случаях
    assert(qualifiedName!(C.foo) == "cclass.C.foo"); // одинаково
}
```

Заметьте, что `c.foo` не квалифицируется как `usingqualifiedname.main.c.foo`.

Локальная область видимости имени

Иногда, при работе с Шаблонами mixin или со Строками mixin, вы внедряете код в неизвестную область. Чтобы выяснить, что это за место, может быть полезным получить локальное имя области видимости. Интуитивно понятно, что предыдущий пример может помочь с этим: просто создайте локальную переменную, получите полное имя его родителя, чтобы определить, в какой области происходит примешивание. Затем, выведите наружу имя области. Давайте назовем это `scopeName`, а связанный с ним любопытный шаблон `getScopeName`.

```
module scopename;
public import qualifiedname;

mixin template getScopeName()
{
```

```
    enum scopeName = qualifiedName!(__traits(parent, scopeName));
}
```

Идея в том, чтобы объявить локальный `enum` с именем `scopeName` и записать туда полное имя его собственного родителя в том же выражении (да, это работает! *Уже нет, см. ниже — прим. пер.*).

Для того, чтобы использовать getScopeName, просто примешайте его туда, где вам нужно имя локальной области видимости:

Баг: Что-то изменилось с момента, когда я последний раз тестировал этот код (2012 год). Сейчас (DMD 2.066, Август 2014), это не работает больше. Я добавлю это к своему списку TODO.

```
module usingscopename;
import std.stdio;
import scopename;

class C
{
    mixin getScopeName; // 1

    int i;
    int foo(int j)
    {
        int k;
        mixin getScopeName; // 2
        writeln(scopeName);
        return i+k;
    }
}

void main()
{
    auto c = new C();
    writeln(c.scopeName); // "usingscopename.C" (1)
    c.foo(1);             // "usingscopename.C.foo" (2)

    mixin getScopeName; // 3
    writeln(scopeName); // "usingscopename.main" (3)
}
```

У меня вся эта система модулей выдала такие ошибки при компиляции:

```
scopename.d(6): Error: circular reference to 'usingscopename.C.getScopeName!
().scopeName'
usingscopename.d(7): Error: mixin usingscopename.C.getScopeName!() error
instantiating
```

т. е. ругается на циклические ссылки в `getScopeName` — нельзя объявлять `enum scopeName`, используя для этого `scopeName`. Я не стал заморачиваться, и попробовал объявить внутри шаблона ещё один `enum`, и ссылаться на него. Вот моя версия модуля `scopename`:

```
module scopename;
public import qualifiedname;

mixin template getScopeName()
{
    enum scopeName1 = true;
    enum scopeName = qualifiedName!(__traits(parent, scopeName1));
```

```
}
```

При таком варианте ошибки не появляются, и всё работает, но есть неприятный побочный эффект — в пространстве видимости вызывающего модуля появляется дополнительный (и, скорее всего, нежелательный) идентификатор `scopeName1` *— прим. пер.*

Примеры

В этой главе представлены различные примеры, показывающие, что можно проделывать с шаблонами D, будь это обработка типов, генерация кода или языковое расширение. Большинство примеров являются кусками кода, которые были полезными мне в тот или иной момент. Я надеюсь, вам они тоже будут полезными.

> **Соавторы, добро пожаловать!** Даже более, чем в случае других частей, я приглашаю размещать любой короткий и синтетический пример того, что можно сделать с помощью шаблонов. Не стесняйтесь ввернуть словечко и вставить свой код в этот документ!

Превращения типов

Одно из наиболее фундаментальных применений шаблонов — превращения типов: создание типа, обработка типа, и т.п.. Язык D статически типизирован, и всё, что вы создаёте, будет иметь определённый тип. Иногда, они могут быть громоздкими для написания или обработки. Шаблоны могут помочь вам с этим.

Отображение, фильтрация и свёртка Типов

Как мы видели в разделе Кортеж параметров шаблона, кортежи параметров могут содержать кортежи типов (это даже их первоначальная роль). Поскольку с ними можно выполнять итерацию (обработку в цикле), индексацию или срез, они являются идеальными кандидатами для нескольких стандартных итерационных алгоритмов. Как для диапазонов, вы можете отображать другой шаблон на кортежи типов, фильтровать типы, которые вы хотите извлечь, или сворачивать (сводить) их к другому типу.

> **И не-типы?** Как насчет действий на кортежах выражений? Вы тоже можете их делать. Даже если этот раздел называется Превращения *типов*, все шаблоны здесь могут работать также на кортежах выражений. Не ограничивайте себя.

Отображение на кортежах типов

Отображение кортежа типа просто применяет другой (унарный, т.е. с одним параметром) шаблон ко всем типам в кортеже. В Phobos уже определён шаблон `staticMap` в модуле std.typetuple, но это — хорошее упражнение, чтобы запрограммировать его снова. Нам нужен шаблон, который принимает имя другого шаблона (как параметр `alias`), скажем, `Wrapper` (обёртка), и кортеж типов `(T0, T1, T2,..., Tn)`, а возвращает `Wrapper!T0, Wrapper!T1, Wrapper!T2, ..., Wrapper!Tn`.

```
module staticmap;
import std.typetuple;

template staticMap(alias M, T...)
{
    static if (T.length == 0) // Конец последовательности
        alias TypeTuple!() staticMap; // останавливаемся здесь
    else
        alias TypeTuple!(M!(T[0]), staticMap!(M, T[1..$])) staticMap;
}
```

Мы используем авто-выравнивание кортежей типов, чтобы собирать результаты в уникальный кортеж. Заметьте, как с помощью индексирования и срезов уменьшается сложность кода. Так как это уже определено в std.typetuple, я буду использовать шаблон из Phobos'a с этого момента.

Даже простой шаблон вроде этого может иметь большое применение:

- Избавление от всех квалификаторов в списке типов, путем отображения std.traits.Unqual на типы.
- Генерация огромного количества типов, используя map-функцию, возвращающую typetuple (смотрите ниже).
- Дана связка функциональных типов, получаем их возвращаемые типы или кортежи параметров.

Пример: тестирование функции

Второй пункт в списке предыдущего подраздела может быть полезным для юнит-теста части вашего кода. Предположим, что у вас есть шаблонная функция, которая предполагает работать с любым встроенным типом. Вам не нужно генерировать все возможные комбинации типов. Просто используйте `staticMap`, чтобы сгенерировать их за вас:

```
module qualified;
import std.typetuple;

/**
* Принимает тип T, генерирует все версии T с квалификаторами,
* которые вы посчитаете интересными (в общем одиннадцать версий).
*/
template Qualified(T)
{
    alias TypeTuple!(
                      T, const(T), immutable(T), shared(T),
                      T[],const(T)[],immutable(T)[], shared(T)[],
                      const(T[]), immutable(T[]), shared(T[])
                     ) Qualified;
}

// Все 16 встроенных типов, которые вас интересуют.
alias TypeTuple!(
                  bool,
                  ubyte,byte,
                  ushort,short,
                  uint,int,
                  ulong,long,
                  float,double,real,
                  char,wchar,dchar
                 ) ValueTypes;

// Взрыв, генерируется 11 * 16 типов.
alias staticMap!(Qualified,ValueTypes) QualifiedTypes;

// Если вы настоящий извращенец (заметьте, что появятся дубликаты)
alias staticMap!(Qualified, QualifiedTypes) DoublyQualifiedTypes;
```

Теперь, если ваша функция предполагает работать со всеми сгенерированными типами с квалификаторами, просто протестируйте её:

```
module tester;
import qualified;

void myFun(T)(T t) {}

template test(alias fun)
{
    void on(T...)()
    {
        foreach(Type; T)
            static if (!__traits(compiles, fun(Type.init)))
                pragma(msg, "Комбинация, не прошедшая тест: "
                          ~ fun.stringof ~ " и " ~ Type.stringof);
    }
}

unittest
{
    test!(myFun).on!(QualifiedTypes);
}
```

Фильтрация кортежей типов

Вы можете искать и извлекать некоторые типы из кортежа, используя предикат, чтобы выбрать, какой тип (или более понятно, какой элемент кортежа) вы хотите оставить. Слово предикат в этом конкретном случае означает «шаблон, который принимает тип как аргумент, и возвращает `true` или `false`». Тест, выполняемый с элементом кортежа, может быть настолько сложным, насколько вам нужно, в частности, с помощью выражения `is()` (смотрите здесь).

Это дает нам следующий код:

```
module staticfilter;
import std.typetuple;

template staticFilter(alias Pred, T...)
{
    static if (T.length == 0) // Конец последовательности
        alias TypeTuple!() staticFilter;
    else static if (Pred!(T[0]))
        alias TypeTuple!(T[0], staticFilter!(Pred, T[1..$])) staticFilter;
    else
        alias TypeTuple!(      staticFilter!(Pred, T[1..$])) staticFilter;
}
```

Использование `staticFilter` очень просто. Давайте получим целые (integral) типы из кортежа, с помощью std.traits.isIntegral:

```
module usingstaticfilter1;
import std.typetuple;
import std.traits;
import staticfilter;

alias TypeTuple!(int, double, float, string, byte, bool, float, void) Types;

alias staticFilter!(isIntegral, Types) OnlyIntegrals;

static assert(is(OnlyIntegrals == TypeTuple!(int, byte)));
```

Как насчет отделения типов от не-типов? Давайте сначала создадим шаблон, который возвращает `true` для чистых типов и `false` для не-типов:

```
module istype;

template isType(T)
{
    enum isType = true;
}

template isType(alias a)
{
    static if (is(a))
        enum isType = true;
    else
        enum isType = false;
}
```

Эй, стоп! Хорошо, мы создали два специфических шаблона, один для параметров-типов и другой для псевдонимов. Но зачем `static if`? Это потому, что объявленные пользователями типы (`MyClass` и ему подобные) — *одновременно* являются как типами, так и идентификаторами (мы видели это в разделе Объявление шаблона). Для этого конкретного применения, я хочу, чтобы они считались типами, в отличие от других идентификаторов (имён функций, имён модулей, переменных, ...). Следовательно, используется `is()`. Мы можем всё немного изменить, чтобы получить детектор встроенных типов, что также может быть интересным. Встроенные типы *нельзя* использовать как параметры-псевдонимы, на основе этого давайте создадим для них тест:

```
module isbuiltintype;

template isSymbol(alias a)
{
    enum isSymbol = true;
}

template isBuiltinType(T)
{
    static if (__traits(compiles, isSymbol!(T)))
        enum isBuiltinType = false;
    else
        enum isBuiltinType = true;
}

template isBuiltinType(alias a)
{
    enum isBuiltinType = false;
}
```

А теперь протестируем:

```
module usingstaticfilter2;
import std.typetuple;
import staticfilter;
import istype;
import isbuiltintype;

class MyClass {}
int foo(int i) { return i;}

alias staticFilter!(isType, 1, int, 3.14, "abc", foo, MyClass) Types;
alias staticFilter!(isBuiltinType, 1, int, 3.14, "abc", foo, MyClass) Builtins;

static assert(is(Types == TypeTuple!(int, MyClass)));
static assert(is(Builtins == TypeTuple!(int)));
```

Но это, по общему признанию, довольно посредственный материал. Он всё же время от времени полезен, но очень редко кто-нибудь получает кортеж чистых типов, подобный этому. Гораздо чаще подобный `staticFilter` применяется при создании сложных типов.

Пример: Создание графа

Как первый пример, представьте себе что у вас есть структура `Graph(Node, Edge)`, шаблонная по типам узлов (вершин) и рёбер (они сами шаблонные). Когда вы создаете граф с помощью фабричной функции, было бы неплохо иметь возможность смешивать узлы и рёбра естественным образом. То есть, данные функции `graph`, `node` и `edge`, действие которых очевидно, вы хотите вызывать как-то так:

```
/**
* Автоматически создаёт граф
*   - с 4 узлами, обозначенными "A", "B", "C", "D", содержащими double,
*   - с соединением узлов "A" с "B", "D" с "A" и "D" с "C".
*/
auto g = graph(node("A", 3.14159), node("B", 1.0),
               edge("A","B"),
               node("C", 2.71828), node("D", 0.0),
               edge("D","A"), edge("D", "C"));
```

Это даст пользователю, строящему свой граф, возможность создавать узлы и рёбра между этими узлами естественным способом (по сравнению с, так называемым, пакетным-построением (batch-building) всех узлов, и после этого добавлением рёбер между ними). Но, при написании библиотеки это означает, что ваша фабричная функция `graph` имеет следующую сигнатуру:

```
auto graph(NodesOrEdges...)(NodesOrEdges args)
if (/* проверка на вменяемость Nodes или Edges*/)
```

Как выполняющее проверку ограничение шаблона (Ограничения), так и построение кода внутри графа могут быть довольно сложными. `staticFilter` помогает разделить аргументы на узлы и рёбра. Чтобы не расширять этот пример слишком сильно, считаем, что в нашем распоряжении уже есть следующие шаблоны-предикаты:

```
template isNode(N) {/* true если N — это Node!(LabelType, ValueType)*/}
template isEdge(E) {/* true если E — это Edge!(LabelType)*/}

template isNodeOrEdge(T)
{
    static if (isNode!T || isEdge!T)
        enum isNodeOrEdge = true;
    else
        enum isNodeOrEdge = false;
}
```

И давайте будем также полагать, что все добросовестные объекты `Node` и `Edge` имеют свойства `.LabelType` (тип названия) и `.ValueType` (тип значения), выводящие наружу их внутренние типы (как показано в разделе Внутренний псевдоним).

Тогда, получить все узлы и рёбра легко:

```
alias staticFilter!(isNode, NodesOrEdges) Nodes;
alias staticFilter!(isEdge, NodesOrEdges) Edges;
```

Это становится все интереснее: получение типов ребер и узлов является лишь первым

строительным блоком. Теперь `graph` должен проверить, как минимум, следующие элементы:

- Все аргументы должны быть узлами или рёбрами.
- Есть, по крайней мере, *один* узел в списке?
- Если да, все ли узлы имеют одинаковый `LabelType` и одинаковый `ValueType`, или, по крайней мере, существует общий тип между всеми типами названий и тоже самое для типов значений, сохраненных в узлах?
- Имеют ли рёбра `LabelTypes` общий тип? (Заметьте, что может быть ноль рёбер).
- Названия рёбер имеют корректный тип, чтобы ссылаться на предоставленные узлы?

Заметьте, что *все* эти проверки выполняются только на типах и, таким образом, могут быть выполнены в время компиляции, тем самым обеспечивая довольно серьёзную статическую проверку строящегося графа. Что не получится сделать таким образом — это проверить во время компиляции, что рёбра на самом деле ссылаются на существующие узлы.

Давайте используем всё, что мы видели к этому моменту для создания шаблона `GraphCheck` (проверка графа), прежде, чем посмотрим другой пример использования `staticFilter`:

```
import std.traits: CommonType;

template GraphCheck(NodesOrEdges...)
{
    enum GraphCheck = GraphCheckImpl!(NodesOrEdges).result;
}

template GraphCheckImpl(NodesOrEdges...)
{
    alias staticFilter!(isNode, NodesOrEdges) Nodes;
    alias staticFilter!(isEdge, NodesOrEdges) Edges;

    // 1. Все аргументы должны быть узлами или рёбрами
    static assert (Nodes.length + Edges.length != NodesOrEdges.length,
                   "Некоторые аргументы не являются узлами или рёбрами.");

    // 2. Должен быть по крайней мере один узел
    static assert (Nodes.length == 0,
                   "Вы должны предоставить по крайней мере один узел.");

    // 3. Есть общий тип для названий узлов и значений?
    // Первый шаг: извлечение названий и значений
    template GetLabel(T) if (isNode!T || isEdge!T)
    {
        alias T.LabelType GetLabel;
    }

    template GetValue(T) if (isNode!T)
    {
        alias T.ValueType GetValue;
    }

    alias staticMap!(GetLabel, Nodes) NodesLabels;
    alias staticMap!(GetValue, Nodes) NodesValues;
```

```
    static assert (is(CommonType!(NodesLabels) == void), // нет общего типа
                   "Узлы не имеют общего типа для названия.");

    static assert (is(CommonType!(NodesValues) == void),
                   "Узлы не имеют общего типа для значения.");

    // 4. Тоже для рёбер
    alias staticMap!(GetLabel, Edges) EdgesLabels;

    static assert (is(CommonType!(EdgesLabels) == void),
                   "Ребра не имеют общего типа для названий.");

    // 5. Совместимость узлы - рёбра
    static assert(!is(CommonType!NodesLabels == CommonType!EdgesLabels),
                  "Узлы и рёбра не имеют одинакового типа для названий.");

    enum result = true;
}
```

Это — один огромный шаблон, но `staticFilter` сидит в середине и существенно упрощает код. Обратите внимание на использование **`static assert`**, сигнатура функции `graph` теперь проще:

```
auto graph(NodesOrEdges...)(NodesOrEdges args) if (GraphCheck!NodesOrEdges)
{ ... }
```

Свёртка кортежей типов

Наряду с отображением и фильтрацией, свёртка (также известная под именем *reducing*, уменьшение) — третье стандартное действие на последовательностях.[17] Идея та же, что и в std.algorithm.reduce: даётся семя `S` и двухаргументная функция `bin`, вычисляется `bin(bin(bin(S, T[0]),T[1],T[2],...))`: применяется `bin` к семени и первому типу в кортеже, затем принимает результирующий тип в качестве нового семени и снова применяет `bin` к нему и второму типу в кортеже, и так далее, пока весь кортеж не будет использован.

Так как использовать такую функцию? Она используется, чтобы *свернуть* кортеж типов в один тип. Этот тип может быть простым типом (например, "наибольший" тип в кортеже, если определено, что значит «больше»), или сложная структура, которая строится итеративно шаг за шагом вдоль кортежа: двоичное дерево, содержащее все типы, например, или обратный кортеж, или даже все типы, но отсортированные согласно предикату.

Здесь мы увидим два примера: получение максимального типа и сортировка кортежа типов.

Но сначала, вот `staticReduce`:

```
module staticreduce;

template staticReduce(alias bin, Types...) // Types[0] — это семя
{
    static if (Types.length < 2)
        static assert(0, "staticReduce: в кортеже "
            ~ Types.stringof ~ " недостаточно элементов (минимум: 2 элемента)");
    else static if (Types.length == 2) // конец рекурсии
```

17 Фактически, это — мать всех операций над последовательностями, поскольку `map` и `filter` можно определить, используя `reduce`.

```
        alias bin!(Types[0], Types[1]) staticReduce;
    else // рекурсия
        alias staticReduce!(bin, bin!(Types[0], Types[1])
                                , Types[2..$])
              staticReduce;
}
```

Итак, как нам получить наибольший тип? Просто применим двухаргументный шаблон `Max` к вашему списку типов:

```
module maxtemplate;

template Max(T1, T2)
{
    static if (T1.sizeof >= T2.sizeof)
        alias T1 Max;
    else
        alias T2 Max;
}

module usingmax;
import std.typetuple;
import maxtemplate;
import staticreduce;

alias TypeTuple!(int, bool, double, float delegate(float), string[]) Types;

alias staticReduce!(Max, Types) MaxType;
static assert(is(MaxType == float delegate(float)));
```

Вы можете поменять определение `Max` по вашему вкусу. Здесь я использовал встроенное свойство `.sizeof`, чтобы сравнить два неизвестных типа. Чтобы сравнивать по именам, нужно использовать `.stringof`, и так далее.

Сортировка типов

Ivory Tower Wankery! (*Вроде, это восклицание про «башню из слоновой кости» об оторванности от жизни, но я не уверен — прим. пер.*) Я имею в виду, когда сортировка типов может быть полезной? Это может быть полезно, читайте дальше.

Когда сортировка кортежа может быть полезной? Главным образом по тем же причинам, по каким вы могли бы захотеть отсортировать любую последовательность:

- легко найти тип (за время log(n)),
- легко получить впоследствии первые p типов или последние p типов,
- сравнить два кортежа типов на эквивалентность,
- легко избавиться от дубликатов (превратить кортеж в множество (set) типов).

Я сфокусируюсь на третьем варианте, сравнении двух кортежей типов. Рассмотрим, например, шаблонную структуру std.variant.Algebraic. `Algebraic!(Type0, Type1, ... TypeN)` может содержать величину одного из типов `Type0`, `TypeN`. Но конечно, в определенном смысле, предыдущий тип такой же, как и `Algebraic!(Type1, TypeN, ... Type0)`, или с любым другим порядком типов. Но это не тот случай:

```
module variant_error;
import std.variant;

void main()
{
    Algebraic!(int, double, string) algOne;
    algOne = 1;
    Algebraic!(double, int, string) algTwo = algOne; // провал!
}
```

Использование отсортированных внутренне типов (или даже с помощью фабричной функции для создания алгебраических величин, которая всегда возвращает отсортированный `Algebraic`) позволит бесшовно использовать алгебраические величины.

Вот один способ сортировки типов, использующий `staticReduce`. Стандартная ситуация следующая: у вас есть список уже отсортированных типов (первые *n* типов начального кортежа), который является вашим текущим состоянием, значение строится. `staticReduce` принимает этот список и помещает первый оставшийся несортированный тип в нём, а затем повторяет для следующего несортированного типа. Так что основным шагом будет добавление нового типа к отсортированному списку.

Возникает небольшая проблема: состояние — просто один тип, но он должен хранить все отсортированные типы. Он не может быть голым кортежем типов, с его авто-выравниванием (смотрите Свойства кортежа). Мы используем std.typecons.Tuple, чтобы завернуть его. Внутренние типы `Tuple` доступны через псевдоним `.Types`. Я боюсь, это немного обезобразит код.

Наконец, какой предикат использовать? `.sizeof` не годится: множество разных типов имеют одинаковый размер и, нехорошим следствием этого будет влияние начального порядка на результирующий отсортированный порядок. Я просто использую строковую версию типов, получаемую встроенным свойством `.stringof`:

```
module sorttypes;
import std.typecons;
import staticreduce;

template Max(T1, T2)
{
    static if (T1.stringof >= T2.stringof)
        alias T1 Max;
    else
        alias T2 Max;
}

template AddToSorted(Sorted, Type)
{
// Длина 0: уже отсортировано
    static if (Sorted.Types.length == 0)
        alias Tuple!(Type) AddToSorted;
// Меньше чем первый: поместить Type на первое место
    else static if (is(Max!(Sorted.Types[0], Type) == Sorted.Types[0]))
        alias Tuple!(Type, Sorted.Types) AddToSorted;
// Больше, чем последний: поместить Type в конец
    else static if (is(Max!(Sorted.Types[$-1], Type) == Type))
        alias Tuple!(Sorted.Types, Type) AddToSorted;
// Иначе, сравнение со средним типом и выполнение рекурсии
//  слева или справа от него
```

```
    else static if (is(Max!(Sorted.Types[$/2], Type) == Type))
        alias Tuple!(Sorted.Types[0..$/2],
                     AddToSorted!(Tuple!(Sorted.Types[$/2..$]),Type).Types)
              AddToSorted;
    else
        alias Tuple!(AddToSorted!(Tuple!(Sorted.Types[0..$/2]),Type).Types,
                     Sorted.Types[$/2..$])
              AddToSorted;
}

template Sort(Types...)
{
    alias staticReduce!(AddToSorted, Tuple!(), Types) Sort;
}
```

Как я сказал, в конце `Sort` просто применяет `AddToSorted` к целевому кортежу. Начальное состояние (семя) для `staticReduce` — пустой кортеж типов, `Tuple!()`.

Теперь это работает? Еще бы:

```
module sortingtypes;
import std.typetuple;
import sorttypes;

alias TypeTuple!(int, bool, string function(int), float[]) Types1;
alias TypeTuple!(int, float[], string function(int), bool) Types2;

static assert(is(Sort!Types1 == Sort!Types2));
```

Если вам не нравится сортировка типов в алфавитном порядке по имени, вы можете по-другому объявить `Max` или, даже лучше, сделать версию `Sort`, которая принимает другой, двойной, шаблон в качестве аргумента и использует этот шаблон как способ упорядочения.

> **Как насчет не-типов?** Как было сказано в самом начале раздела, шаблоны, представленные здесь, в основном работают и для других параметров шаблона: чистые числа, строки, псевдонимы. Здесь вам придётся изменить второй параметр `AddToSorted`, чтобы он принимал не-типы. Или, другим способом сделать это было бы сначала выполнить отображение элементов кортежа в строковую форму, а затем сортировать результирующие строки.

Сканирование типов, Чередование типов

TODO Определить, действительно ли полезен этот подраздел. Есть ли короткий и показательный пример?

Статическое сканирование

Сканирование — своего рода свёртывающее действие (смотрите Свёртка кортежей типов), но даёт промежуточные результаты. Оно используется в шаблоне `Juxtapose` в подразделе Соседские функции.

```
module staticscan;
import std.typetuple;

/**
Даёт кортеж типов, получающийся из последовательных применений F
для уменьшения списка T.
*/
template StaticScan(alias F, T...)
```

```
{
    static if (T.length == 0)
        alias TypeTuple!() StaticScan; // Этот вариант никогда не должен
                                       // случиться при нормальном использовании
    static if (T.length == 1)
        alias TypeTuple!(T[0]) StaticScan;
    else
        alias TypeTuple!(T[0], StaticScan!(F, F!(T[0], T[1]), T[2..$])) StaticScan;
}
```

Чередование типов

```
module interleave;
import std.typetuple;

/**
 * Принимает кортежи (T0, T1, T2, ..., Tn) и (U0, U1, ..., Um),
 * возвращает поочерёдно из первого и из второго:
 *
 * (T0, U0, T1, U1, ...
 *
 * Если один из входов короче другого,
 * более длинная часть помещается в конце чередования.
 */
template Interleave(First...)
{
    template With(Second...)
    {
        static if (First.length == 0)
            alias Second With;
        else static if (Second.length == 0)
            alias First With;
        else
            alias TypeTuple!( First[0], Second[0]
                            , Interleave!(First[1..$]).With!(Second[1..$]))
                  With;
    }
}
```

Комментирование типов

Незакончено Идея в том, чтобы завернуть величины в шаблоне, добавляющем некие метаданные. В идеале, я хотел бы получить нечто, работающее примерно так:

```
auto arr = [0,1,2,3,4]; // Массив целых типа int
auto arr2 = sorted(arr); // Теперь, мы знаем, что он отсортирован
auto arr3 = positive(arr2); // Отсортирован *и* все элементы положительные
```

Или, в более общем плане:

```
auto arr = [0,1,2,3,4]; // Массив целых типа int
auto arr2 = annotate!("sorted", (a,b) => a<b)(arr);
auto arr3 = annotate!("positive")(arr2);

assert("positive" in arr3.properties);
assert(arr3.Properties == TypeTuple!( Property!("sorted", (a,b) => a < b)
                                    , Property!("positive")));

// завёрнутое значение всё ещё там:
auto arr4 = array(filter!((a) => a%2==0))(arr3);
// избавиться от некоторых свойств
auto arr5 = arr3.discardProperty!"positive";
assert(arr5.Properties == TypeTuple!(Property!("sorted", (a,b) => a < b)));
```

```
auto arr6 = annotate!("negative")([-4, -3, -2, -1]);
auto arr7 = annotate!("sorted", (a,b) => a<b)(arr6);

assert(arr3.property!"sorted" == arr7.property!"sorted"); // тот же предикат
```

Вот первый *очень грубый и незаконченный* проект:

```
module annotation;
import std.typetuple;

struct Meta(string Name, alias Data)
{
    enum name = Name;
    alias Data data;
}

template isMeta(T)
{
    static if (__traits(hasMember, T, "name")
            && __traits(hasMember, T, "data"))
        enum isMeta = true;
    else
        enum isMeta = false;
}

template GetName(alias a)
{
    enum string GetName = a.name;
}

template isAnnotated(T)
{
    static if (__traits(compiles, T.Annotations))
        enum bool isAnnotated = true;
    else
        enum bool isAnnotated = false;
}

string getNames(Metadata...)() @property
{
    alias staticMap!(GetName, Metadata) names;
    string result;
    foreach(name; names)
        result ~= "\""~name~"\",";
    if (names.length) result = result[0..$-1];
    return "alias TypeTuple!(" ~ result ~ ") names;";
}

struct Annotated(T, Metadata...)
if (allSatisfy!(isMeta, Metadata))
{
    T value;
    alias value this;
    mixin(getNames!(Metadata));
    Metadata metadata;

    auto property(string s)() @property
    {
        static if (staticIndexOf!(s, names) != -1)
            return metadata[staticIndexOf!(s, names)];
        else
            static assert(false, "Unknown property: " ~ s);
    }
```

```
    bool hasAnnotation(string name) @property
    {
        foreach(n; names)
            if (name == n) return true;
        return false;
    }
}

// auto annotated(T)(T value)
// {
//     return Annotated!(T)(value);
// }

template annotated(Metadata...) if (Metadata.length)
{
    auto annotated(T)(T value)
    {
    //     alias TypeTuple!(Metadata) MetaTypes;
        static if (isAnnotated!(T))
            return Annotated!(T.AnnotatedType, T.Annotations, Metadata)
(value.value);
        else
        {
            Annotated!(T, Metadata) a;
            a.value = value;
            return a;
        }
    }
}
```

Кортежи как последовательности

Незакончено Идея заключается в том, чтобы рассматривать кортежи значений (например, `tuple(1, "abc", 3.14, 3, 0)`) как последовательности: отображать (полиморфную, или шаблонную) функцию на них, фильтровать их, и т.п.. Зачем это делать: представьте себе, например, связку диапазонов, все разных типов. Если вы хотите сгруппировать их в диапазоне и обрабатывать, вы не сможете, потому что диапазон должен быть однородным по типу. Функции, представленные здесь, снимут это ограничение.

Отображение на кортежах

```
module maptuple;
import std.typecons;
import std.typetuple;
import std.functional;

/**
 * Вспомогательный шаблон для получения возвращаемого типа шаблонной функции.
 */
template RT(alias fun)
{
    template RT(T)
    {
        alias typeof(fun(T.init)) RT;
    }
}

/// Отображение на кортеже, с использованием полиморфной функции.
/// Производит другой кортеж.
Tuple!(staticMap!(RT!fun, T)) mapTuple(alias fun, T...)(Tuple!T tup)
{
```

```
    StaticMap!(RT!fun, T) res;
    foreach(i, Type; T) res[i] = unaryFun!fun(tup.field[i]);
    return tuple(res);
}
```

Почему-то в отличие от автора, для меня оказалось не совсем очевидным, как должна выглядеть «полиморфная» функция, и как вызывать этот отображающий шаблон. Так что приведу здесь свой вариант использования этого шаблона, вдруг кому-то пригодится:

```
module usingmaptuple;
import maptuple;
import std.stdio, std.typecons, std.conv;

auto fun1(T)(T arg)
{
      static if (__traits(compiles, T.init + 1))
            return (arg + 1);
      else
            return to!string(arg) ~ "+1";
}

enum tt1=tuple(1, "abc", 3.14, 3, 0);
enum tt2=mapTuple!(fun1)(tt1);

void main()
{
      writeln(tt2);
}
```

выведет:

```
Tuple!(int, string, double, int, int)(2, "abc+1", 4.14, 4, 1)
```

— прим. пер.

Фильтрация кортежей

Незакончено Идеей здесь будет фильтрация кортежа в соответствии с предикатом, действующим на типах. Это очень полезно, когда вы можете получить связку параметров в функцию и хотите использовать только некоторые из них. Смотрите раздел Фильтрация кортежей типов, пример с функцией `graph`.

```
module filtertuple;
import std.typecons;
import std.typetuple;

template FilterTupleTypes(alias pred, alias tup)
{
    static if (tup.field.length)
    {
        static if (pred(tup.field[0]))
            alias TypeTuple!(tup.Types[0],
                  FilterTupleTypes!(pred, tuple(tup.expand[1..$])))
                  FilterTupleTypes;
        else
            alias FilterTupleTypes!(pred, tuple(tup.expand[1..$]))
                  FilterTupleTypes;
    }
    else
    {
        alias TypeTuple!() FilterTupleTypes;
    }
```

```
}

template FilterTupleIndices(alias pred, alias tup, size_t ind)
{
    static if (tup.field.length)
    {
        static if (pred(tup.field[0]))
            alias TypeTuple!( ind
                            , FilterTupleIndices!( pred
                                                 , tuple(tup.expand[1..$])
                                                 , ind+1
                                                 )
                            ) FilterTupleIndices;
        else
            alias FilterTupleIndices!( pred
                                     , tuple(tup.expand[1..$])
                                     , ind+1
                                     ) FilterTupleIndices;
    }
    else
    {
        alias TypeTuple!() FilterTupleIndices;
    }

}

/// Фильтрует кортеж по его величинам.
Tuple!(FilterTupleTypes!(pred, tup)) filterTuple(alias pred, alias tup)()
{
    FilterTupleTypes!(pred, tup) result;
    alias FilterTupleIndices!(pred, tup, 0) indices;
    foreach(i, ind; indices)
    {
        result[i] = tup.field[ind];
    }
    return tuple(result);
}

(1, "abc", 2, "def", 3.14)
->
((1,2),("abc","def"),(3,14))
```

Веселье с функциями

Этот раздел представит несколько шаблонов, действующих на функциях или создающих шаблонные обёртки вокруг функций для их расширения. Он не является частью раздела Шаблоны функций, поскольку тут используются шаблоны структур, и не является частью раздела Шаблоны структур, так как там я не фокусировался на обертывающих структурах.

Определение количества аргументов функции

Незакончено Это — старый код, посмотрите, если что-нибудь изменилось здесь за два прошлых года.

```
module functionarity;
import std.traits;

template arity(alias fun)
if (isFunction!fun)
```

```
{
    enum size_t arity = ParameterTypeTuple!(fun).length;
}
```

Мемоизация функции

Когда функция выполняет длинные вычисления, вероятно, она станет эффективнее, сохраняя вычисленные результаты во внешней структуре, и на запрос эта структура выдаст результат вместо повторного вызова функции. Это называется *memoizing* (не memorizing), и этот раздел покажет, как использовать шаблон, чтобы сделать мемоизацию удобнее.

Ранее вычисленные результаты сохраняются в ассоциативном массиве, индексируемом на кортежах аргументов. Чтобы получить тип возвращаемого значения или кортеж типов параметров функции, просто используйте std.traits.ReturnType и std.traits.ParameterTypeTuple из Phobos, которые являются шаблонами, принимающими *имена* функций или типы.

```
module memoize1;
import std.traits;
import std.typecons;

struct Memoize(alias fun)
{
    alias ReturnType!fun RT;
    alias ParameterTypeTuple!fun PTT;
    RT[Tuple!(PTT)] memo; // сохраняет результат, индексация на аргументах.

    RT opCall(PTT args)
    {
        if (tuple(args) in memo)      // Мы уже видели ли эти аргументы?
        {
            return memo[tuple(args)]; // если да, используем сохранённый результат
        }
        else // если нет, вычисляем результат и сохраняем его.
        {
            RT result = fun(args);
            memo[tuple(args)] = result;
            return result;
        }
    }
}

Memoize!fun memoize(alias fun)()
{
    Memoize!fun memo;
    return memo;
}
```

Использование очень простое:

```
module usingmemoize1;
import memoize1;

int veryLongCalc(int i, double d, string s)
{
    /* ... cheating here ... */
    return i;
}

void main()
{
    auto vlcMemo = memoize!(veryLongCalc);
```

```
    // вычисление veryLongCalc(1, 3.14, "abc")
    // требует минуты!
    int res1 = vlcMemo(1, 3.14, "abc");
    int res2 = vlcMemo(2, 2.718, "def");// минуты снова!
    int res3 = vlcMemo(1, 3.14, "abc"); // несколько миллисекунд на получение res3
}
```

Вышеуказанный код тривиален и мог бы быть оптимизирован многими способами. По большей части, реальный шаблон мемоизации должен также модифицировать свое поведение в соответствии с политикой хранения. Например:

- Ограниченный или нет размер места хранения?
- В случае ограниченного размера: как определить предел, и какова должна быть политика выселения?
  - Первый пришёл — первый вышел из памяти?
  - Наименее используемая *в последнее время* память?
  - Наименее используемая?
  - Время жизни?
  - Выбросить всё и очистить память?
  - Выбросить лишь часть?
- Остановка мемоизации?

Последние X результатов можно сохранять в очереди: всякий раз, когда результат помещается в ассоциативный массив, кортеж аргументов помещается в очередь. Как только вы достигнете максимального предела места хранения, выбросите самый старый, или (например) половину сохраненных величин.

Вот возможная небольшая реализация. Это сделано для хорошего примера включения/отключения кода со **`static if`** и политиках, основанных на **`enum`**. Заметьте, что я использую динамический массив D в качестве простой очереди. Реальная очередь, вероятно, может быть более эффективной, но в стандартной библиотеке нет ни одной реализации на момент написания статьи.

```
module memoize2;
import std.traits;
import std.typecons;

enum Storing {
    always,  // сохранять всегда
    maximum  // сохранять до максимального количества
}

enum Discarding {
    oldest,   // выбросить только старший результат
    fraction, // выбросить часть (0.5 == 50%)
    all       // гори, гори!
}

struct Memoize(alias fun,
               Storing storing,
               Discarding discarding)
```

```
{
    alias ReturnType!fun RT;
    alias ParameterTypeTuple!fun PTT;

    static if (storing == Storing.maximum)
    {
        Tuple!(PTT)[] argsQueue;
        size_t maxNumStored;
    }

    static if (discarding == Discarding.fraction)
        float fraction;

    RT[Tuple!(PTT)] memo; // хранит результат, проиндексированный аргументами.

    RT opCall(PTT args)
    {
        if (tuple(args) in memo)       // Мы уже видели ли эти аргументы?
        {
            return memo[tuple(args)]; // если да, используем сохранённый результат
        }
        else                           // если нет,
        {
            static if (storing == Storing.always)
            {
                RT result = fun(args);// вычисляем результат и сохраняем его.
                memo[tuple(args)] = result;
                return result;
            }
            else // Storing.maximum
            {
                if (argsQueue.length >= maxNumStored)
                {
                    static if (discarding == Discarding.oldest)
                    {
                        memo.remove(argsQueue[0]);
                        argsQueue = argsQueue[1..$];
                    }
                    else static if (discarding == Discarding.fraction)
                    {
                        auto num = to!size_t(argsQueue.length * fraction);
                        foreach(elem; argsQueue[0..num])
                            memo.remove(elem);
                        argsQueue = argsQueue[num..$];
                    }
                    else static if (discarding == Discarding.all)
                    {
                        memo = null;
                        argsQueue.length = 0;
                    }
                }

                RT result = fun(args);// вычисляем результат и сохраняем его.
                memo[tuple(args)] = result;
                argsQueue ~= tuple(args);
                return result;
            }
        }
    }
}
```

И несколько фабричных функций, помогающих созданию этих структур `Memoize`:

```
module memoize3;
```

```
import memoize2;

// Нет аргументов времени выполнения -> всегда сохранять
Memoize!(fun, Storing.always, Discarding.all)
memoize(alias fun)()
{
    Memoize!(fun,
             Storing.always,
             Discarding.all) result;
    return result;
}

// Один аргумент времени выполнения size_t arg ->
// сохраняем максимум / выбрасываем всё
Memoize!(fun, Storing.maximum, Discarding.all)
memoize(alias fun)(size_t max)
{
    Memoize!(fun,
             Storing.maximum,
             Discarding.all) result;
    result.maxNumStored = max;
    return result;
}

// Два аргумента времени выполнения (size_t, double) ->
// сохраняем максимум / выбрасываем часть
Memoize!(fun, Storing.maximum, Discarding.fraction)
memoize(alias fun)(size_t max, double fraction)
{
    Memoize!(fun,
             Storing.maximum,
             Discarding.fraction) result;
    result.maxNumStored = max;
    result.fraction = fraction;
    return result;
}

// Один аргумент времени компиляции  (discarding oldest),
// Один аргумент времени выполнения  (max)
Memoize!(fun, Storing.maximum, discarding)
memoize(alias fun, Discarding discarding = Discarding.oldest)
(size_t max)
{
    Memoize!(fun,
             Storing.maximum,
             Discarding.oldest) result;
    result.maxNumStored = max;
    return result;
}
```

Следует отметить, что в связи с введением оператора `opCall`, не представляется возможным использовать литерал **`struct`**. Мы должны сначала создать структуру, а затем инициализировать её поля.

Большую часть времени, типа аргументов времени выполнения достаточно, чтобы определить, какого поведения вы хотите при мемоизации/сохранении. Только для (редкой?) политики выкидывания одного самого старого сохраненного результата работы пользователя необходимо указать его с помощью аргумента шаблона:

```
module usingmemoize3;
import memoize3;
```

```
int veryLongCalc(int i, double d, string s)
{
   /* ... cheating here ... */
   return i,;
}

void main()
{
    // Сохранять первый миллион результатов, очищать память полностью
    auto vlcMemo1 = memoize!(veryLongCalc)(1_000_000);

    // Сохранять первый миллион результатов, очищать половину максимума памяти
    auto vlcMemo2 = memoize!(veryLongCalc)(1_000_000, 0.5f);

    // Сохранять первые двадцать результатов, выкидывать только старший
    auto vlcMemo3 = memoize!(veryLongCalc, Discarding.oldest)(20);
}
```

Каррирование функции

Незакончено Некоторые пояснения очень помогли бы.

Другое полезное превращение функций — их каррирование[18], превращающее функцию с n-аргументами в n функций с одним аргументом внутри друг друга.

TODO Показать некоторый пример: отображение диапазона, например.

```
module checkcompatibility;

template CheckCompatibility(T...)
{
    template With(U...)
    {
        static if (U.length != T.length)
            enum With = false;
        else static if  (T.length == 0) // U.length == 0 также
            enum With = true;
        else static if (!is(U[0] : T[0]))
            enum With = false;
        else
            enum With = CheckCompatibility!(T[1..$]).With!(U[1..$]);
    }
}

module curry;
import std.traits;
import checkcompatibility;

struct Curry(alias fun, int index = 0)
{
    alias ReturnType!fun RT;
    alias ParameterTypeTuple!fun PTT;
    PTT args;

    auto opCall(V...)(V values)
        if (V.length > 0
         && V.length + index <= PTT.length)
    {
        // fun напрямую вызывается с предоставленными аргументами?
        static if (__traits(compiles, fun(args[0..index], values)))
            return fun(args[0..index], values);
        // Если нет, новые аргументы будут сохранены. Мы проверяем их типы.
```

18 по имени Haskell Curry, который формализовал идею.

```
        else static if (!CheckCompatibility!(PTT[index..index + V.length]).With!
(V))
            static assert(0, "curry: плохие аргументы. Ожидали "
                             ~ PTT[index..index + V.length].stringof
                             ~ " но получили " ~ V.stringof);
        // еще не хватает аргументов. Мы храним их.
        else
        {
            Curry!(fun, index+V.length) c;
            foreach(i,a; args[0..index]) c.args[i] = a;
            foreach(i,v; values) c.args[index+i] = v;
            return c;
        }
    }
}

auto curry(alias fun)()
{
    Curry!(fun,0) c;
    return c;
}
```

Не очень сложно понять, как работают эти шаблоны, гораздо сложнее — зачем это нужно. Я, прочитав статью в [википедии](), так этого и не понял. Запрос в Яндексе «зачем нужно каррирование» выдал несколько ссылок, в некоторых из них были примеры: [здесь](), [здесь]() и т. д. Но лично меня эти примеры не убедили. В этих примерах для общей функции существуют некие важные (или часто используемые) частные случаи. И с помощью каррирования (или частичного применения, иногда эти понятия разделяются) эти частные случаи выделяются в отдельную функцию. Как я понимаю ситуацию, если такой частный случай один, то можно использовать параметры по-умолчанию (они есть почти во всех языках программирования), а если их несколько, то введение нескольких новых функций только добавляет путаницы — прим. пер.

Соседские функции

Незакончено Идея в том, чтобы склеивать функции вместе для отображения на многих диапазонах параллельно и создания другого диапазона со множеством значений впоследствии.

Функции соседствуют (juxtaposed), то есть, пусть дано:

```
int foo(int i, int j) { return i+j;}
string bar() { return "Hello, World";}
double baz(double d) { return d*d;}
```

Тогда `juxtapose!(foo,bar,baz)` — функция, принимающая два `int`'а и `double` в качестве аргументов, и возвращающая кортеж, содержащий `int`, `string` и `double`.

```
module juxtapose;
import juxtaposehelper;

template juxtapose(Funs...)
{
    Tuple!(staticFilter!(isNotVoid, ReturnTypes!Funs))
    juxtapose(ParameterTypeTuples!Funs params)
    {
        typeof(return) result;
        alias SumOfArities!Funs arities;
        alias SumOfReturns!Funs returns;
```

```
        foreach(i, Fun; Funs)
        {
            enum firstParam = arities[i];
            enum lastParam = firstParam + arity!(Fun);
            static if (returns[i] != returns[i+1])
                result.field[returns[i]] = Fun(params[firstParam..lastParam]);
        }
        return result;
    }
}
```

Необходимая основа:

```
module juxtaposehelper;
import std.traits;
import functionarity;
import staticscan;
import maponalias;

template isNotVoid(T)
{
    enum bool isNotVoid = !is(T == void);
}

/**
 * Принимает связку имён функций, дает кортеж типов их возвращаемых значений.
 * Используется в juxtapose.
 */
template ReturnTypes(Funs...)
{
    alias MapOnAlias!(ReturnType, Funs) ReturnTypes;
}

/**
 * Принимает связку имён функций, дает (выровненный) кортеж
 * их параметров. Используется в juxtapose.
 */
template ParameterTypeTuples(alias fun, Rest...)
{
    alias MapOnAlias!(ParameterTypeTuple, fun, Rest) ParameterTypeTuples;
}

template SumOfArity(size_t zero, alias fun)
{
    enum size_t SumOfArity = zero + arity!fun;
}

template SumOfArities(F...)
{
    alias StaticScan!(SumOfArity, 0, F) SumOfArities;
}

template SumOfReturn(size_t zero, alias fun)
{
    static if (is(ReturnType!fun == void))
        enum size_t SumOfReturn = zero;
    else
        enum size_t SumOfReturn = zero + 1;
}

template SumOfReturns(Funs...)
{
    alias StaticScan!(SumOfReturn, 0, Funs) SumOfReturns;
}
```

Должен ли я также добавить это?

```
module maponalias;
import std.typetuple;

/**
 * Отображает шаблон Mapper на список псевдонимов.
 */
template MapOnAlias(alias Mapper, alias current, Rest...)
{
    static if (Rest.length)
        alias TypeTuple!(Mapper!current, MapOnAlias!(Mapper, Rest)) MapOnAlias;
    else
        alias Mapper!current MapOnAlias;
}

template MapOnAlias(alias Mapper)
{
    alias TypeTuple!() MapOnAlias;
}
```

На всякий случай добавлю, что здесь используется модуль `functionarity`, предоставляющий шаблон `arity`, считающий количество аргументов функции, а также модуль `staticscan`, предоставляющий шаблон `StaticScan` — прим. пер.

Реляционная алгебра

Вдохновение для этого примера пришло из Статьи в этом блоге.

TODO Я должен добавить код где-нибудь. Что он должен делать: извлечение из кортежа: проекция, выборка. Также, непосредственное/внутреннее/внешнее присоединение, декартово произведение. И пересечение/объединение/разница. `rename!("oldField", "newField")`. Базы данных просто динамические массивы кортежей.

```
module relational;
import std.traits;
import std.typetuple;
import std.typecons;
import std.range;
import std.conv;
import isastringliteral;
import istype;
import half;
import interleave;

struct Entry(Data...)
if ( (Data.length % 2 == 0)
  && (allSatisfy!(isAStringLiteral, Half!Data))
  && (allSatisfy!(isType, Half!(Data[1..$], void))))
{
    alias Half!Data Headers;
    alias Half!(Data[1..$], void) Values;
    alias data this;

    Tuple!(Interleave!(Values).With!(Headers)) data;

    this(Values values)
    {
        foreach(i, Unused; Values) data[i] = values[i];
    }
```

```
    string toString() @property
    {
        string s = "[";
        foreach(i, Unused; Values)
            s ~= Headers[i] ~ ":" ~ to!string(data[i]) ~ ", ";
        return s[0..$-2] ~ "]";
    }
}

template entry(Headers...)
if (allSatisfy!(isAStringLiteral, Headers))
{
    Entry!(Interleave!(Headers).With!(Values)) entry(Values...)(Values values)
    if (Values.length == Headers.length)
    {
        return typeof(return)(values);
    }
}

template isEntry(E)
{
    enum isEntry = __traits(compiles,
                      {
                           void fun(T...)(Entry!T e) {}
                           fun(E.init);
                      });
}

auto rename(string from, string to, E)(E e)
if (isEntry!E)
{
    enum index = staticIndexOf!(from, E.Headers);
    static if (index == -1)
        static assert(false, "Bad index in rename: no header called "
                             ~ from ~ " in " ~ E.Headers.stringof);
    else
        return entry!(E.Headers[0..index],
                      to,
                      E.Headers[index+1..$])(e.data.expand);
}

auto rename(string from, string to, D)(D db)
if (isDatabase!D)
{
    ReturnType!(rename!(from, to, ElementType!D))[] result;
    foreach(i, elem; db)
        result ~= rename!(from, to)(elem);
    return result;
}


template isDatabase(D)
{
    static if (isDynamicArray!D && isEntry!(ElementType!D))
        enum bool isDatabase = true;
    else
        enum bool isDatabase = false;
}

template isSomeHeader(E)
if (isEntry!E)
{
    template isSomeHeader(string s)
    {
```

```
        static if (staticIndexOf!(s, E.Headers) != -1)
            enum bool isSomeHeader = true;
        else
            enum bool isSomeHeader = false;
    }
}

template HeaderValue(E)
if (isEntry!E)
{
    template HeaderValue(string header)
    if (staticIndexOf!(header, E.Headers) != -1)
    {
        alias TypeTuple!(header, E.Values[staticIndexOf!(header, E.Headers)])
              HeaderValue;
    }
}

template project(Headers...)
if (Headers.length > 0)
{
    auto project(E)(E e)
    {
        static if (isEntry!E && allSatisfy!(isSomeHeader!E, Headers))
        {
            alias staticMap!(HeaderValue!E, Headers)
                  HeadersAndValues;
            Entry!(HeadersAndValues) result;
            foreach(i, Unused; Headers)
                mixin("result." ~ Headers[i] ~ " = e." ~ Headers[i] ~ ";");
            return result;
        }
        else static if (isDatabase!E
                     && allSatisfy!(isSomeHeader!(ElementType!E), Headers))
        {
            alias staticMap!(HeaderValue!(ElementType!E), Headers)
                  HeadersAndValues;

            Entry!(HeadersAndValues)[] result;
            Entry!(HeadersAndValues)   elem;

            foreach(i,Unused; e)
            {
                foreach(j, Unused2; Headers)
                    mixin("elem." ~ Headers[j] ~ " = e[i]." ~ Headers[j] ~ ";");
                result ~= elem;
            }
            return result;
        }
        else
            static assert(0, "Cannot project on " ~ Headers.stringof[5..$]
                           ~ " for type " ~ E.stringof);
    }
}
```

`Half` (половина) — что-то вроде обратного шаблона к `Interleave` (чередование, смотри раздел Чередование типов): принимает кортеж, и берёт из него каждый второй тип:

```
module half;
import std.typetuple;

template Half(T...)
{
    static if (T.length == 0)
```

```
        alias TypeTuple!() Half;
    else static if (T.length == 1)
        alias TypeTuple!(T[0]) Half;
    else
        alias TypeTuple!(T[0], Half!(T[2..$])) Half;
}

unittest
{
    alias TypeTuple!() Test0;
    alias TypeTuple!(int) Test1;
    alias TypeTuple!(int, float) Test2;
    alias TypeTuple!(int, float, string) Test3;
    alias TypeTuple!(int, float, string, double) Test4;

    static assert(is(Half!Test0 == TypeTuple!()));
    static assert(is(Half!Test1 == TypeTuple!(int)));
    static assert(is(Half!Test2 == TypeTuple!(int)));
    static assert(is(Half!Test3 == TypeTuple!(int, string)));
    static assert(is(Half!Test4 == TypeTuple!(int, string)));
}
```

Это дает нам:

```
module usingrelational;
import std.stdio;
import relational;

alias Entry!("Name",     string
            ,"EmployId", int
            ,"Dept", string) Employee;

alias Entry!("DeptName", string
            ,"Manager",  string) Dept;

void main()
{
    auto e = entry!("Name", "EmployId", "DeptName")("John", 1, "Tech");
    auto e2 = Employee("Susan", 2, "Financial");
    auto e3 = Employee("Bob", 3, "Financial");
    auto e4 = Employee("Sri", 4, "Tech");

    auto d1 = Dept("Tech", "Sri");
    auto d2 = Dept("Financial", "Bob");

    auto employees = [e2,e3,e4];
    auto depts = [d1, d2];

    writeln(employees);

    writeln(rename!("Dept", "DN")(employees));
}
```

*Тут ещё был раздел «**Веселье с классами и структурами**», но, к сожалению, автор его так и не начал толком писать — прим. пер.*

Порождение событий

Этот пример шаблона, включающий `Fields` и `Notify`, получен от Andrej Mitrovic. Он был достаточно любезен, чтобы позволить мне разместить его здесь и дать мне некоторые пояснения о логике, стоящей за этим шаблоном. Он использует его в своей экспериментальной библиотеке графического интерфейса пользователя (GUI) для сигналов о

событиях.

```
module fields;
import std.typetuple;
import isastringliteral;

mixin template Fields(T, fields...)
    if (allSatisfy!(isAStringLiteral, fields))
{
    alias typeof(this) This;

    static string __makeFields(T, fields...)()
    {
        string res;
        foreach(field; fields) res ~= T.stringof~ " " ~ field ~ ";\n";
        return res;
    }

    static string __makeOpBinaryFields(string op, fields...)()
    {
        string res;
        foreach(field; fields)
            res ~= "res." ~ field
                 ~ " = this." ~ field ~ op ~ " rhs." ~ field ~ ";\n";
        return res;
    }

    mixin(__makeFields!(T, fields)());

    This opBinary(string op)(This rhs)
    {
        This res;
        mixin(__makeOpBinaryFields!(op, fields)());
        return res;
    }

    void opOpAssign(string op)(This rhs)
    {
        mixin("this = this " ~ op ~ " rhs;");
    }
}
```

Пользователь примешивает его в своих собственных типах:

```
module usingfields;
import fields;

struct Point {
    mixin Fields!(int, "x", "y", "z", "w");
}

struct Size {
    mixin Fields!(int, "width", "height");
}
```

Это идет рука об руку с шаблоном структуры `Notify`:

```
module notify;
import std.conv;

struct Notify(T)
{
    alias void delegate(ref T) OnEventFunc;
    OnEventFunc onEvent;
```

```
    void init(OnEventFunc func) {
        onEvent = func;
    }

    string toString()
    {
        return to!string(raw);
    }

    auto opEquals(T)(T rhs)
    {
        return rhs == rhs;
    }

    void opAssign(T)(T rhs)
    {
        if (rhs == raw)
            return;  // избежать бесконечных циклов

        // temp используется для ref
        auto temp = rhs;
        onEvent(temp);
    }

    auto opBinary(string op, T)(T rhs)
    {
        mixin("return raw " ~ op ~ " rhs;");
    }

    void opOpAssign(string op, T)(T rhs)
    {
        // temp используется для ref
        mixin("auto temp = raw " ~ op ~ " rhs;");

        if (temp == raw)
            return;  // избежать бесконечных циклов

        onEvent(temp);
    }

    public T raw;  // raw доступен, когда мы не хотим вызывать event()
    alias raw this;
}
```

Что вы смешиваете в классах, которые вы хотите уведомлять о каких-либо изменениях их внутреннего состояния:

```
module usingnotify;
import std.stdio;
import usingfields;
import notify;

class Widget
{
    this()
    {
        point.init((ref Point pt) { writefln("changed point to %s", pt); });
        size.init((ref Size sz)   { writefln("changed size to %s", sz); });
    }

    Notify!Point point;
    Notify!Size size;
}
```

Например, пользователь может изменить поле `point` (положение) виджета через:

```
Point moveBy = Point(10, 0); widget.point += moveBy;}.
```

Это не модифицирует поле непосредственно, а только вызывает `onEvent(moveBy)`, которая, в свою очередь, выдает сигнал, содержащий ссылку `Widget` и запрашиваемую позицию `doMove.emit(this, moveBy)`. Это обрабатывается цепочкой произвольного количества слушателей. Эти слушатели принимают ссылку `moveBy`, и это оказывается полезным, когда, например, `Widget` является частью `Layout` (раскладка). Layout просто добавляет себя к цепочке слушателей и редактирует ref-аргумент если он хочет, или даже возвращает **`false`**, чтобы полностью отвергнуть какие-либо изменения в поле.

Это позволяет несколько увеличить гибкость. Например, предположим, пользователь создаёт подкласс от `Widget` (давайте назовем его `Child`) и перекрывает метод `onMove()`, чтобы ограничить его позицию:

```
min: Point(10, 10) (top-left)
max: Point(100, 100) (bottom-right)
```

Этот `Widget` имеет родительский `Widget` (называемый `Parent`), в котором есть набор раскладки. Раскладка может позволить дочерним `Widget`'ам родителя размещаться только между следующими позициями:

```
min: Point(20, 20)
max: Point(80, 80)
```

Дополнительно раскладка может принять во внимание размер ребенка, так что `Widget` никогда не перейдёт за точки раскладки `min` и `max`. Если этот `Widget` был расположен в `Point(20, 20)` и имел Размер `Size(50, 50)`, это означает, что раскладка ограничит минимальные и максимальные точки расположения `Widget`'а в:

```
min: Point(20, 20)
max: Point(30, 30)
```

Так, если пользователь пытается вызвать:

```
child.point = Point(120, 120);
```

Сначала, метод `onMove` объекта `Child` является первым слушателем некоторого сигнала `doMove`. Он модифицирует ref аргумент и устанавливает его на `Point(100, 100)`, так как пользователь запрограммировал его таким образом. Затем, раскладка модифицирует его к точке `Point(30, 30)`, поскольку она принимает размер `Widget`'а в расчет, так чтобы не случилось выхода за пределы.

Может быть любое количество слушателей, и вы могли бы добавить слушателя в любую позицию в цепочке событий (может быть, вы хотите прервать и модифицировать аргумент до того, как он достигнет родительской раскладки, если вам нужно установить временные ограничения для позиция `Widget`'а).

Наконец, всегда есть один последний слушатель. Это — внутренняя функция, которая на самом деле модифицирует поле `.raw` и вызывается во внутренних рисующей и blitting функциях, чтобы заставить `Widget` появиться в его новой позиции. Также, любой слушатель может вернуть **`false`**, чтобы прервать дальнейшую обработку событий и запретить

изменение поля.

Обратите внимание на то, что назначения никогда не произойдет, внешний код должен использовать `.raw` поле, чтобы фактически изменить внутреннюю полезную информацию, что позволяет избежать вызова `OnEvent()`. *Я не понял этот абзац, он как-то противоречит написанному в предыдущем — прим. пер.*

Поля

Из библиотеки сериализации Orange от Jacob Carlborg's:

```
module fieldsof;

/**
 * Вычисляет имена полей в данном типе
 * в массив строк
 */
template fieldsOf (T)
{
    const fieldsOf = fieldsOfImpl!(T, 0);
}

/**
 * Реализация fieldsOf
 *
 * Возвращает: массив строк, содержащий имена полей в данном типе
 */
template fieldsOfImpl (T, size_t i)
{
    static if (T.tupleof.length == 0)
        enum fieldsOfImpl = [""];

    else static if (T.tupleof.length - 1 == i)
        enum fieldsOfImpl = [T.tupleof[i].stringof[1 + T.stringof.length + 2 ..
$]];

    else
        enum fieldsOfImpl = T.tupleof[i].stringof[1 + T.stringof.length + 2 ..
$] ~ fieldsOfImpl!(T, i + 1);
}
```

Расширение enum

Этот код получен от Simen Kjaeras. Он генерирует определение enum в виде строки, получая все члены старого `enum`, и добавляя к ним переданные параметры в виде строки, и примешивает их туда.

```
module enumdefasstring;

string EnumDefAsString(T)()
if (is(T == enum))
{
   string result = "";
   foreach (e; __traits(allMembers, T))
       result ~= e ~ " = T." ~ e ~ ",";
   return result;
}
```

Эта часть кода проходит циклом по всем членам переданной `enum T`, генерируя строку, содержащую все члены и их значения. Для этого `enum`:

```
enum bar
{
    a, b, c
}
```

Сгенерированная строка выглядит похожей на это (если вы хотите проверить это, можете просто вызвать `EnumDefAsString` во время выполнения и напечатайте её результат):

```
"a = bar.a,b = bar.b,c = bar.c"
```

Как мы можем видеть, это — корректное тело для **`enum`**. Это означает, что мы можем использовать **`mixin`**`()` для генерации точно такого же **`enum`**. Но подождите — вот ещё больше:

```
module extendenum;
import enumdefasstring;

template ExtendEnum(T, string s)
   if (is(T == enum) &&
   is(typeof({mixin("enum a{"~s~"}");})))
{
   mixin(
   "enum ExtendEnum {"
   ~ EnumDefAsString!T()
   ~ s
   ~ "}");
}
```

Этот код складывает строку, сгенерированную из предыдущей функции, с переданной в функцию в виде параметра `s`. Так что с ранее определенным `bar`, и созданием экземпляра таким образом:

```
ExtendEnum!(bar, "d=25")
```

Тело функции будет выглядеть так (после расширения строкой):

```
mixin(
   "enum ExtendEnum {"
   ~ "a = bar.a,b = bar.b,c = bar.c"
   ~ "d=25"
   ~ "}");
```

Складывая эти строки, мы видим, что у нас получается корректное определение **`enum`**:

```
enum ExtendEnum {a = bar.a,b = bar.b,c = bar.c,d=25}
```

Затем это вставляется в **`mixin`**, и компилируется как обычный D-код.

Статический switch

TODO Что, отсутствует switch времени компиляции? Давайте его создадим. Пример: кортежи, фильтрация типа (в ограничениях), рекурсия, и т.п..

```
module staticswitch;

template staticSwitch(List...) // List[0] - величина, командующая переключением
                               // Она может быть типом или идентификатором.
{
    static if (List.length == 1) // Не из чего выбирать: ошибка
        static assert(0, "StaticSwitch: no match for " ~ List[0].stringof);
    else static if (List.length == 2) // Единственная позиция: значение по-умолчанию
```

```
        enum staticSwitch = List[1];
    else static if (is(List[0] == List[1]) // Сравнение по типам
                || (  !is(List[0])         // Сравнение по значениям
                   && !is(List[1])
                   && is(typeof(List[0] == List[1]))
                   && (List[0] == List[1])))
        enum staticSwitch = List[2];
    else
        enum staticSwitch = staticSwitch!(List[0], List[3..$]);
}
```

Мне кажется, нужно пояснить не сразу очевидную вещь, что сравнение идёт с элементами на нечётных позициях (`List[1]`, `List[3]`, и т. д.), а в качестве результата при совпадении возвращается следующий за ним элемент (если совпадение `List[0] == List[3]`, то возвращается `List[4]`) — прим. пер.

Общие структуры

Незакончено Этот раздел представляет несколько структур растущей сложности.

Поглощение

Давайте начнём с `Gobbler` (поглотитель), небольшого упражнения с обработкой кортежа и перегрузкой оператора. `Gobbler` — это структура, завёртывающая кортеж и определяющая только один оператор: конкатенацию справа (\~).

```
module gobbler;

struct Gobbler(T...)
{
    alias T Types;
    T store;
    Gobbler!(T, U) opBinary(string op, U)(U u) if (op == "~")
    {
        return Gobbler!(T, U)(store, u);
    }
}

Gobbler!() gobble() { return Gobbler!()();}
```

`gobble` создает поглотитель, и там активируется засасывание:

```
module usinggobbler;
import std.typetuple;
import gobbler;

void main()
{
    auto list = gobble ~ 1 ~ "abc" ~ 3.14 ~ "another string!";
    assert(is(list.Types == TypeTuple!(int, string, double, string)));
    assert(list.store[2] == 3.14);
}
```

TODO Индексирование `Gobbler`.

Я, как любитель динамического языка Python, не могу не посетовать, что всё эти радости с кортежами работают только до компиляции... Нельзя к этому объекту добавить ещё что-то после создания:

```
list = list ~ "ещё строка!"; // Error: cannot implicitly convert expression
```

Но можно сделать так:

```
auto list2 = list ~ "ещё строка!"; // OK
```

— don. пер.

Полиморфные ассоциативные списки

Ассоциативный список — своего рода "плоский" ассоциативный массив: он содержит пары ключ-значение в линейном списке. Полиморфный (или, шаблонный) — это кортеж, содержащий связку ключей ключ-значение, но с большей гибкостью в типах для ключей и значений. Тут можно добиться различных компромиссов между временем выполнения и временем компиляции для природы ключей и значений.

Незакончено Этот раздел представит одно решение.

Использование: немного похоже на таблицы Lua: структуры, классы (вы можете поместить в них анонимные функции?), пространства имён. Также, может быть, добавить метаданные к типу?

```
module alist;
import std.typetuple;
import std.stdio;
import half;

struct AList(T...)
{
    static if (T.length >= 2 && T.length % 2 == 0)
        alias Half!T Keys;
    else static if (T.length >= 2 && T.length % 2 == 1)
        alias Half!(T[0..$-1]) Keys;
    else
        alias TypeTuple!() Keys;

    static if (T.length >= 2)
        alias Half!(T[1..$]) Values;
    else
        alias TypeTuple!() Values;

    template at(alias a)
    {
        // ключ не найден, но присутствует значение по-умолчанию
        static if ((staticIndexOf!(a, Keys) == -1) && (T.length % 2 == 1))
            enum at = T[$-1]; // значение по-умолчанию
        else static if ((staticIndexOf!(a, Keys) == -1) && (T.length % 2 == 0))
            static assert(0, "AList: no key equal to " ~ a.stringof);
        else //static if (Keys[staticIndexOf!(a, Keys)] == a)
            enum at = Values[staticIndexOf!(a, Keys)];
    }
}

void main()
{
    alias AList!( 1,     "abc"
                , 2,     'd'
                , 3,     "def"
                , "foo", 3.14
                ,        "Default") al;

    writeln("Keys: ", al.Keys.stringof);
    writeln("Values: ", al.Values.stringof);
```

```
    writeln("at!1: ", al.at!(1));
    writeln("at!2: ", al.at!(2));
    writeln("at!\"foo\": ", al.at!("foo"));
    writeln("Default: ", al.at!4);
}
```

Полиморфное дерево

Так, что такое полиморфное дерево? Это просто кортеж, содержащий другие кортежи как элементы, как стандартный контейнер дерева, только все величины могут быть разных типов. Очевидно это означает, что деревья, содержащие величины разных типов, также будут разными типами, так как тип всего содержимого является частью сигнатуры дерева. Может быть немного непонятно, как посмотреть один элемент, но с несколькими вспомогательными функциями, трансформирующими дерево или извлекающими некоторые значения, это может быть весьма интересно использовать.

Просто, чтобы немного прочувствовать и, чтобы использовать не заезженный пример для деревьев, представьте себе, что нужно манипулировать размеченным текстом в D. Вы могли бы создать ваш документ как D-структуру, затем ввести некоторые функции, превращающие её в текст DDoc, или в документ LaTeX, в Markdown, или даже в HTML:

```
auto doc =
document(
    title("Ranges: A Tutorial"),
    author("John Doe"),
    tableOfContents,

    /* Глава 1-го уровня */
    section!1(
        title("Some Range Definitions"),
        "Ranges are a nice D abstraction...",
        definition(
            "Input Range",
            "The most basic kind of range, it must define the following
methods:",
            list(definition("front", "..."),
                 definition("popFront", "..."),
                 definition("empty", "..."))
        )
    )
    section!1(
        title("Some ranges examples"),
        "...",
        code(
            "auto m = map!((a) => a*a)([0,1,2,3]);
             assert(m.length == 4);"
        ),
        link("http://dlang.org/", "Website")
    )
    section!1(
        title("Beyond ranges"),
        "..."
    )
);

auto latex = doc.to!"LaTeX";
auto html = doc.to!"HTML";
auto ddoc = doc.to!"Ddoc";
auto simple = doc.to!"text";
```

В предыдущем коде (воображаемом, но заманчивом для меня), `doc` — это кортеж, созданный фабричной функцией `document`, и содержащий небольшие специально помеченные части текста: `title` (название), `section` (глава или раздел) или `link` (ссылка). Каждая из них — фабричная функция, производящая заданную пользователем структуру, выполняющую несколько простых соглашений. Если все типы имеют член `to!"HTML"`, который преобразует их содержимое в код HTML, то весь документ может быть выгружен в виде HTML-файла. Различные типы не должны быть классами, наследующимися от общей базы и, они должны быть запиханы (*в оригинале «shoe-horned», у меня моск взорвался искать, что может означать «обувь-рогатым» — офигевание пер.*) в заданную иерархию: Ограничения шаблона делают проверку для вас. Подумайте о диапазонах.

Это — пример полиморфного дерева.

Э-э, алло, начальник! Где реализация? — от пер.

Шаблоны выражений

Шаблоны выражений являются сортом полиморфного дерева, но ограниченного до нескольких известных операций (в большинстве случаев одинарные/двоичные/троичные операторы) и их операндов. Это позволяет хранить, например, арифметическую операцию следующим образом:

```
// Из "x + 1 * y":
Binary!("+",
             Variable!"x",
             Binary!("*",
                         Constant(1),
                         Variable!"y"))
```

Преимущество в том, что затем вы можете манипулировать результирующим деревом для упрощения выражения, или, чтобы избежать временных вычислений. Предыдущее выражение можно было бы упростить и сохранить эквивалент `x + y` (избавившись от умножения на единицу).

В более общем плане, вы можете кодировать выражение языка программирования в таком дереве:

```
AST!"
if (x == 0)
then
{
    writeln(x);
}
else
{
    ++x;
    foo(x,y);
}"
=>
If(Comparison!("==", Symbol!"x", value(0)), // Условие
// ветка то
   Block!( FunctionCall!("writeln", Symbol!"x") ),
// (опционально) ветка иначе
   Block!( Unary!("++", Symbol!"x"),
           FunctionCall!("foo", Symbol!"x", Symbol!"y")))
```

Этот путь лежит к безумию и мощи макросов, поскольку вы можете затем манипулировать результирующим Абстрактным Синтаксическим Деревом (Abstract Syntax Tree, AST) любым нужным вам способом, переписывать код, который он представляет, преобразовывать его снова в строку и записывать в файл, который будет передан компилятору.

Итак,

- Определите синтаксический анализатор времени компиляции,
- скормите его (разумную часть) грамматике D,
- определите несколько новых разрешённых конструкций, соответствующие им AST, и способ, как эти новые конструкции могут быть созданы за-сценой из существующей части языка D (т.е., макрос),
- записать код в вашем новом расширении D, вашем *драгоценном*,
- скормить его с макросом в программу, которая создаст результирующий AST, модифицирует его, как вы хотите, и пересоберёт в *настоящий* код D,
- и затем направьте его в стандартный компилятор D.

И вуаля, готово ваше собственное игрушечное расширение D. Или, вы знаете, вы могли бы просто написать в багтрекер Walter'y, и подождать, пока он не добавит нужный вам синтаксис в язык.

Статически-проверенный Writeln

Это пример проверки предметно-ориентированного языка (DSL) во время компиляции в D. Цель состоит в том, чтобы взять строку формата для std.stdio.writef или std.stdio.writefln и проверить типы аргументов перед передачей их в `writef(ln)`.

Например, при написании:

```
writefln("For sample #%d, the results are (%s, %f)", num, name, f);
```

Мы знаем, что `%d` будет означать, что аргумент должен быть целым типом, `%f` требует тип с плавающей точкой и так далее. В приведённом примере это означает, что нам известно следующее:

- Должно быть точно 3 аргумента, ни больше, ни меньше.
- Первый должен иметь целый тип.
- Второй может быть любого типа (более точно, любого типа, который может быть преобразован в строку).
- Третий должен быть некоторым типом с плавающей точкой.

Эти четыре условия можно проверить во время компиляции, что мы будем здесь делать. Я не буду программировать всю спецификацию POSIX для `printf`, следующая таблица показывает, что мы будем проверять.

Таблица 10. Стандартные спецификаторы формата, определяемые `cwritef`

Спецификатор формата	**Просит**	**Эквивалентное ограничение**
`%d, %i`	Целый тип	`isIntegral`
`%u, %x, %X, %o`	Беззнаковый целый тип	`isUnsigned`
`%f, %F, %e, %E, %g, %G`	Тип с плавающей точкой	`isFloatingPoint`
`%c`	Символьный тип	`isSomeChar`
`%s`	Любой тип	`isAnyType`
`%%`	Не спецификатор формата	Не проверяется

Для тех, кто заинтересован в деталях, эта статья в Википедии будет хорошей ссылкой. Заметьте, что не все спецификаторы формата реализованы в std.stdio, например `%p` для указателей `void*`, кажется, не работает.

Большинство упомянутых шаблонов проверки типов можно найти в std.traits: `isIntegral`, `isFloatingPoint`, `isUnsigned` и `isSomeChar` уже реализованы в Phobos. Только одно осталось: `isAnyType`, довольно благодушный шаблон:

```
module isanytype;

template isAnyType(T)
{
    enum isAnyType = true;
}
```

Единственный способ потерпеть с ним неудачу — дать ему не-тип.

Продолжая пример разработки на основе предыдущих разделов, вот что я хочу получить (`cwritefln` означает проверенный `writefln`):

```
module usingcheckedwrite;
import checkedwrite;

void main()
{
    cwritefln!"Для примера #%d, результаты (%s, %f)"( 0, "foo", 3.14); // OK

 // NOK: плохое количество аргументов: ожидалось 3 аргумента, получено 2.
 // cwritefln!" Для примера #%d, результаты (%s, %f)"( 0, "foo");

 // NOK: аргумент #3 типа double не прошел проверку isFloatingPoint
 // cwritefln!" Для примера #%d, результаты (%s, %f)"( 0, 3.14, "foo");
}
```

Теперь, получая строку форматирования, в первую очередь нужно извлечь спецификаторы формата и создать список ограничений. Здесь я использую строку mixin и нужно просто построить строку, представляющую желаемый конечный код:

```
module getformatters;
```

```d
import std.conv;
import std.traits;

string getFormatters(S)(S s) if (isSomeString!S)
{
    dstring ds = to!dstring(s);
    bool afterPercent = false;
    bool error;
    string result = "alias TypeTuple!(";
    foreach(elem; ds)
    {
        if (error) break;
        if (afterPercent)
        {
            switch (elem)
            {
                case '%':
                    afterPercent = false;
                    break;
                case 'd':
                case 'i':
                    result ~= "isIntegral,"; // целые
                    afterPercent = false;
                    break;
                case 'u':
                case 'x':
                case 'X':
                case 'o':
                    result ~= "isUnsigned,"; // беззнаковые целые
                    afterPercent = false;
                    break;
                case 'f':
                case 'F':
                case 'e':
                case 'E':
                case 'g':
                case 'G':
                    result ~= "isFloatingPoint,"; // с плавающей точкой
                    afterPercent = false;
                    break;
                case 'c':
                    result ~= "isSomeChar,"; // символ
                    afterPercent = false;
                    break;
                case 's':
                    result ~= "isAnyType,"; // любой тип, преобразуемый в строку
                    afterPercent = false;
                    break;
                /* флаги, ширина, */
                case '+':
                case '-':
                case '#':
                case '.':
                case ' ':
                case '0':
                    ..
                case '9':
                    break;
                default:
                    error = true; // Ошибка!
                    break;
            }
        }
        else
```

```
        {
            if (elem == '%') afterPercent = true;
        }
    }

    // Избавиться от последней запятой:
    if (result.length > 17) result = result[0..$-1];
    // окончание кода alias
    result ~= ") ArgsChecks;";

    if (afterPercent // finished the string but still in "afterPercent" mode
     || error)
        result = "static assert(0, \"Bad format string: \" ~ a);";

    return result;
}
```

Это — очень длинный образец, но логика за ним простая: он перебирает все символы и ищет шаблоны `%x`. Я включил здесь основную обработку для флагов и тому подобное, но как я сказал раньше, этот пример не имеет дело со всей спецификацией POSIX: целью здесь является *не* подтверждение строки форматирования, а извлечение спецификаторов формата. Когда он определяет, что строка имеет неверный формат, будет сгенерирован код **`static assert`**.

Так, в конце концов, мы получаем строки, готовые-к-примешиванию, подобные этим:

```
// нет форматирования
"alias TypeTuple!() ArgsChecks;"

// предыдущий пример
"alias TypeTuple!(isIntegral,isAnyType,isFloatingType) ArgsChecks;"

// Плохая строка
"static assert(0, \"Bad format string: %s and %z\");"
```

Как только кортеж проверок сделан, нам нужен шаблон, который в свою очередь проверяет каждый аргумент с соответствующим шаблоном. Для получения более удобного сообщения об ошибке, я использую параметр шаблона типа `int`, чтобы считать количество проверенных аргументов.

```
1   module verifychecks;
2   import std.conv;
3   import std.traits;
4   import std.typetuple;
5   import isanytype;
6   import getformatters;
7
8   template ArgsChecks(alias a) if (isSomeString!(typeof(a)))
9   {
10      mixin(getFormatters(a));
11  }
12
13  template VerifyChecks(int which, Checks...)
14  {
15      template on(Args...)
16      {
17          static if (Checks.length != Args.length)
18              static assert( 0
19                           , "ctwrite неверное количество аргументов: ожидалось "
20                           ~ to!string(Checks.length)
```

```
21  |                           ~ " аргументов, получено "
22  |                           ~ to!string(Args.length)
23  |                           ~ ".");
24  |         else static if (Checks.length == 0) // конец процесса
25  |             enum on = true;
26  |         else static if ({ alias Checks[0] C; return C!(Args[0]);}()) // рекурсия
27  |             enum on = VerifyChecks!(which+1, Checks[1..$]).on!(Args[1..$]);
28  |         else
29  |             static assert( 0
30  |                           , "cwrite неверный аргумент: аргумент #"
31  |                           ~ to!string(which)
32  |                           ~ " типа "
33  |                           ~ Args[0].stringof
34  |                           ~ " не прошёл проверку "
35  |                           ~ __traits(identifier, Checks[0]));
36  |     }
37  | }
```

Шаблон `Verify` является другим примером двухэтажного шаблона, как мы видели в разделе Шаблоны на шаблонах, чтобы получить мило вызываемый синтаксис:

```
Verify!(0, isIntegral, isFloatingPoint).on!(int, double)
```

Наиболее тонкая часть находится на строке 26:

```
else static if ({ alias Checks[0] C; return C!(Args[0]);}()) // рекурсия
```

Стандартным способом сделать это было бы:

```
alias Checks[0] Check;
// и затем
else static if (Check!(Args[0]))
```

Но это поместило бы идентификатор `Check` в локальную область шаблона (в свое время это сломало бы одноимённый трюк). Было бы возможно определить шаблон `VerifyImpl`, но использование локального делегата также работает:

```
{ alias Checks[0] C; return C!(Args[0]);}()
```

Часть `{...}` определяет делегата, а часть `()` вызывает его, возвращая **`true`** или **`false`**. Затем это вставляется в **`static if`**.

Во всяком случае, как только проверка кода выполнена, остальное просто:

```
module checkedwrite;
import std.stdio;
import std.traits;
import verifychecks;

void cwritef(alias a, Args...)(Args args)
if (isSomeString!(typeof(a))
 && VerifyChecks!(1, ArgsChecks!(a)).on!(Args))
{
    writef(a, args);
}

void cwritefln(alias a, Args...)(Args args)
if (isSomeString!(typeof(a))
 && VerifyChecks!(1, ArgsChecks!(a)).on!(Args))
{
    writefln(a, args);
}
```

Использование смотрите здесь.

Расширение класса

TODO UFCS теперь реализован. Это случилось в 2012 году!

В сообществе D постоянно высказывается желание добавить нечто, называемое Универсальный Синтаксис Вызова Функции (UFCS) — автоматическое преобразование `a.foo(b)` в `foo(a,b)` при отсутствии члена с именем `foo`, и при наличии свободной функции с именем `foo` в локальной области. Это уже работает для массивов (следовательно, и для строк), но не для других типов.

Не существует способа получить это в D для встроенных типов, кроме как взломать компилятор, но для определённых пользователем типов можно вызвать шаблоны для спасения.

Можно использовать `opDispatch` для перенаправления внешней свободной функции. Вызов `this.method(a,b)` превращается в `method(this,a,b)`.

```
module forwarder;

mixin template Forwarder()
{
    auto opDispatch(string name, Args...)(Args args)
    {
        mixin("return " ~ name ~ "(args);");
    }
}
```

В D, пустой `return` является законным:

```
return;
// или return void;
```

Так что, если `name(this,a,b)` — функция, возвращающая `void`, всё хорошо.

Основное ограничение этого трюка — то, что он не работает через границы модулей. Слишком плохо.

Приложения

Выражение is

Общий синтаксис

Выражение `is(...)` позволяет выполнять интроспекцию времени компиляции на типах (и, как побочный эффект, в выражениях D). Это описано здесь на сайте D. Это выражение имеет причудливый синтаксис, но в основном использование очень простое, и оно очень полезно в связке со `static if` (смотрите раздел Static If) и с ограничениями шаблона (смотрите раздел Ограничения). Общий синтаксис:

```
is( Type (optional identifier) )
is( Type (optional identifier) :  OtherType,
         (optional template parameters list) )
is( Type (optional identifier) == OtherType,
         (optional template parameters list) )
```

Если то, что в скобках — верно (смотрите ниже), `is()` возвращает `true` во время компиляции, иначе оно возвращает `false`.

is(Type)

Давайте начнем с самого первого синтаксиса: если `Type` — допустимый D-тип в области видимости выражения, `is()` возвращает `true`. Как бонус, внутри `static if`, дополнительный `идентификатор` становится псевдонимом для типа. Например:

```
module canbeinstantiated;

template CanBeInstantiatedWith(alias templateName, Types...)
{
    // templateName!(Types) — это допустимый тип?
    static if (is( templateName!(Types) ResultType ))
    // здесь вы можете использовать ResultType (== templateName!(Types))
        alias ResultType CanBeInstantiatedWith;
    else
        alias void       CanBeInstantiatedWith;
}
```

Заметьте, что в предыдущем коде подразумеваются шаблоны, но выражение `is()` весьма надёжно: если вы передадите в качестве `templateName` что-то, что не является именем шаблона (имя функции, например), `is` увидит, что `templateName!(Types)` станет недопустимым типом и вернёт `false`. `CanBeInstantiatedWith` будет корректно установлен в `void` и ваша программа не разрушится.

> **Тестирование для псевдонима** Иногда вы не знаете, какой аргумент шаблона вы получили — тип или псевдоним (например, при работе с элементами кортежей). В этом случае, вы можете использовать `!is(symbol)` в качестве теста. Если он на самом деле псевдоним, а не тип, это выражение вернёт `true`.

Интересный вариант использования для этой формы выражения `is` — тестирование, является ли некоторый код D допустимым. Судите сами: блоки D рассматриваются компилятором как делегаты (Их тип — это `void delegate()`). Использование их в

сочетании с `typeof` позволяет проверить правильность выражений в блоке: если `некоторый код` допустим, `typeof({ некоторый код }())` (заметьте скобки `()` в конце делегата для его активации), является реальным D-типом и `is` вернёт истину.

Давайте это как-то используем. Представьте, что у вас есть шаблон функции `fun` и несколько аргументов, но вы не знаете, можно ли вызывать `fun` с этой конкретной связкой аргументов. Если это частый случай в вашем коде, вам нужно абстрагировать это в шаблон. Давайте назовем его `validCall`, и сделаем его также функциональным шаблоном, чтобы легко использовать с аргументами:

```
module validcall;
import std.conv;

bool validCall(alias fun, Args...)(Args args)
{
    return is( typeof({ /* код для тестирования */
                        fun(args);
                        /* конец кода для тестирования */
                      }()));
}

// Использование:
T add(T)(T a, T b) { return a+b;}
string conc(A,B)(A a, B b) { return to!string(a) ~ to!string(b);}

void main()
{
    assert( validCall!add(1, 2));     // генерирует add!(int)
    assert(!validCall!add(1, "abc")); // невозможно создать экземпляр шаблона

    assert( validCall!conc(1, "abc")); // conc!(int, string) всё хорошо.
    assert(!validCall!conc(1)       ); // нет 1-аргументной версии для conc

    struct S {}

    assert(!validCall!S(1, 2.3)); // S не вызывается
}
```

Заметьте, что протестированный код является просто '`fun(args);`'. То есть, нет условия на тип `fun`: это может быть функцией, делегатом или даже структурой или классом с определенным методом `opCall`. Есть в основном два варианта, когда `fun(args);` может оказаться недействительным кодом: или `fun` не может вызываться как функция, или её можно вызывать, но аргументы окажутся недействительными.

Между прочим, может быть весело использовать этот трюк, но D предоставляет вам более чистый способ тестирования на правильность компиляции:

```
__traits(compiles, { /* some code */ })
```

`__traits` — это ещё одна из множества конструкций — швейцарских ножей в D. Вы можете найти документацию по `compiles` здесь. Ему посвящен раздел __traits.

`is(Type : AnotherType)` и `is(Type == AnotherType)`

Две других формы `is` возвращают `true`, если `Type` может быть неявно преобразован в (происходит от) `AnotherType` и если `Type` в точности совпадает с `AnotherType`,

соответственно. Я нахожу их более всего интересными в их более сложной форме, со списком параметров шаблона в конце. В этом случае, параметры шаблона действуют чем-то вроде переменных типов в уравнении. Позвольте мне объяснить:

```
is(Type identifier == SomeComplexTypeDependingOnUAndV, U, V)
```

Идентификатор SomeComplexTypeDependingOnUAndV переводится как «Некий сложный тип, зависящий от U и V» — прим. пер.

Предыдущий код на самом деле означает: "извините, Мистер Компилятор D, но — `Type` случайно не является неким сложным типом, зависящем от U и V для некоторых U и V? Если да, пожалуйста, дайте мне их." Например:

```
template ArrayElement(T)
{
    // T — это массив из U, для некоторого типа U?
    static if (is(T t : U[], U))
        alias U ArrayElement; // U можно использовать, выведем его наружу
    else
        alias void ArrayElement;
}

template isAssociativeArray(AA)
{
    static if (is( AA aa == Value[Key], Value, Key))
        /* код здесь может использовать Value и Key,
           они выведены компилятором. */
    else
        /* AA не является ассоциативным массивом,
          Value и Key не определены. */
}
```

Как ни странно, вы можете только использовать это с `is(Type identifier, ...)` синтаксис: вы *должны* иметь идентификатор. Хорошие новости в том, что сложные типы, которые вы проверяете, могут быть шаблонными типами и список параметров может быть любым параметром шаблона: не только типы, но целые величины, и т.д. Например, предположим, что вы делаете то, что делает каждый, когда сталкивается с шаблонами D: вы создаёте шаблонный тип n-мерного вектора.

```
struct Vector(Type, int dim) { ... }
```

Если вы не вывели наружу `Type` и `dim` (например, как псевдонимы, смотрите раздел Внутренний псевдоним и Даём доступ ко внутренним параметрам), вы можете использовать `is`, чтобы извлечь их:

```
Vector!( ?, ?) myVec;
// myVec — это вектор из int, любой размерности?
static if (is(typeof(myVec) mv == Vector!(int, dim), dim))

// это 1-мерный вектор?
static if (is(typeof(myVec) mv == Vector!(T, 1), T))
```

> **`is(A != B)`?** Нет, извините, этого нет. Используйте `!is(A == B)`. Но будьте осторожны, это также сработает, если A или B не являются легальными типами (что имеет смысл: если A не определена, то она по определению не может быть равна B). Если необходимо, вы можете использовать `is(A) && is(B) && !is(A == B)`.

is A - супертип B? Эй, `is(MyType : SuperType)` хорошо работает, чтобы узнать является ли `MyType` подтипом у `SuperType`. Как мне спросить, является ли `MyType` супертипом для `SubType`? Легко, просто используйте `is(SubType : MyType)`.

Для меня, основным ограничением является то, что не принимается кортеж параметров шаблона. Слишком плохо. Смотрите, представьте, что вы много используете std.typecons.Tuple. В какой-то момент вам понадобится шаблон, который тестирует, является ли что-то `Tuple!(T...)` для некоторого `T`, в котором может быть 0 или больше типов. Тут is немного разочаровывает, так как вы не можете написать:

```
template isTuple(T)
{
    static if (is(T tup == Tuple!(InnerTypes), InnerTypes...)
(...)                                          ^^^^^^^^^^^^^
```

TODO Да, вы можете! Это было изменено в DMD 2.060. Этот раздел требует обновления!

But sometimes D channels its inner perl and, lo! Есть несколько способов сделать это! Вы можете использовать IFTI и нашего хорошего друга — выражение `is(typeof({...}()))`. Вы можете также использовать `__traits`, в зависимости от вашего настроения, но так как это приложение именно об `is`:

```
1   module istuple;
2   import std.typecons;
3
4   template isTuple(T)
5   {
6       enum bool isTuple =
7           is(typeof({
8               void tupleTester(InnerTypes...)(Tuple!(InnerTypes) tup) {}
9               T.init possibleTuple;
10              tupleTester(possibleTuple);
11          }()));
12  }
```

Строка 8 определяет шаблон функции `tupleTester`, которая в качестве аргументов принимает только `Tuple` (даже если она ничего с ними не делает). Мы создаём некий экземпляр типа `T` на строке 9, используя свойство `.init`, существующее всех D-типах, и пытаемся вызвать `tupleTester` с ним. Если `T` — на самом деле `Tuple`, весь этот блок является допустимым, вызов результирующего делегата на самом деле имеет тип, и `is` возвращает `true`.

Стоит отметить здесь две вещи: во первых, `isTuple` сработает для любого шаблонного типа с именем `Tuple`, не только для std.typecons.Tuple. Если вы хотите сделать его более строгим, измените определение `tupleTester`. Во-вторых, мы таким образом не получаем доступ ко внутренним типам. Для std.typecons.Tuple это на самом деле не проблема, так как до них можно добраться через псевдоним `someTuple.Types` , но всё же...

Между прочим, к самим элементам списка параметров шаблона можно использовать синтаксис `A : B` или `A == B`:

```
static if (is( T t == A!(U,V), U : SomeClass!W, V == int[n], W, int n))
```

Это сработает, если `T` на самом деле экземпляр `A` с типами `U` и `V`, которые тоже проверяются, что `U` — это производный от `SomeClass!W` для некоторого типа `W`, и, что `V` является статическим массивом целых длины `n` (и их можно использовать впоследствии). В ветви `if` внутри **`static if`** все `U`, `V`, `W` и `n` определены.

Специализации типов

Есть последняя вещь, которую надо знать о **`is`**: с версией

```
is(Type (identifier) == Something)
```

`Something` (что-то) может также быть специализацией типа, одним из следующих ключевых слов D: **`function`**, **`delegate`**, **`return`**, **`struct`**, **`enum`**, **`union`**, **`class`**, **`interface`**, **`super`**, **`const`**, **`immutable`** или **`shared`**. Условие выполняется, если `Type` — один из них (за исключением **`super`** и **`return`**, смотрите ниже). Идентификатор затем становится псевдонимом для некоторого свойства `Type`, как описано в следующей таблице.

Таблица 11. Эффект специализации типа в `is`

Специализация	Удовлетворяет условию	`identifier` становится
`function`	Type - это функция	Кортежем типов параметров функции
`delegate`	Type - это делегат	The delegate function type
`return`	Type - это функция	Возвращаемый тип
`return`	Type - это делегат	Возвращаемый тип
`struct`	Type - это структура	Типом структуры
`enum`	Type - это enum	Базовым типом enum
`union`	Type - это union	Типом union
`class`	Type - это класс	Типом класса
`interface`	Type - это интерфейс	Типом интерфейса
`super`	Type - это класс	Кортежем типов (Базовый класс, Интерфейсы)
`const`	Type - const	Типом
`immutable`	Type - immutable	Типом
`shared`	Type - shared	Типом

Давайте как-то это применим: мы хотим фабричный шаблон, который создаёт новую структуру или новый класс, получая имя в виде параметра шаблона:

```
module maker;
import std.algorithm;
import std.conv;

template make(A)
    if (is(A a == class )
```

```
     || is(A a == struct))
{
    auto make(Args...)(Args args)
    {
        static if (is(A a == class))
            return new A(args);
        else
            return A(args);
    }
}

struct S {int i;}
class C
{
    int i;
    this(int ii) { i = ii;}
    override string toString() @property
    {
        return "C("~to!string(i)~")";
    }
}

void main()
{
    auto array = [0,1,2,3];

    auto structRange = map!( make!S )(array);
    auto classRange  = map!( make!C )(array);

     assert(equal(structRange, [S(0), S(1), S(2), S(3)]));
    assert(classRange.front.toString == "C(0)");
}
```

Вы можете найти другой подобный пример в разделе Шаблоны классов, с шаблоном `duplicator`. (*К сожалению, я не нашёл этого примера — прим. пер.*)

Ресурсы и дополнительная литература

Незакончено Добро пожаловать любому новому ресурсу!

Будучи относительно молодым языком, D не имеет десятков книг или интернет-сайтов, посвященных его изучению. Тем не менее, имеется несколько ресурсов в Интернете, которые имеют отношение к шаблонам D и связанным с ними темам.

Шаблоны D

Вот некоторые ресурсы о шаблонах D.

Ссылки

Вашей первой остановкой должны быть страницы о шаблонах на dlang.org. Они очень легкие для чтения, не стесняйтесь просматривать их и беспокоить авторов, если что-то неясно или явно неверно. Веб-сайт dlang.org — это также проект на github, здесь. В следующей таблице перечислены страницы, имеющие дело с шаблонами.

Таблица 12. Страницы, имеющие дело с шаблонами

Тема	URL

Общая страница о шаблонах	template.html
Сравнение шаблонов D и C++	templates-revisited.html
Статья о кортежах (*сейчас там не статья, а только маленький абзац с двумя ссылками — прим. пер.*)	tuple.html
Кортеж параметров шаблона	variadic-function-templates.html
Ограничения шаблонов	concepts.html
Шаблоны Mixin	template-mixin.html
Маленькая статья о строках mixin	mixin.html
`static if`	version.html
`is`	expression.html
Псевдонимы типов	declaration.html
Перегрузка операторов	operatoroverloading.html

Язык программирования D

Конечно, главная книга по D — `Язык программирования D` (The D Programming Language, также известная как TDPL), автор Andrei Alexandrescu (*переведена на русский язык, смотрите, например здесь — прим. пер.*). Это очень хороший справочник, и её также интересно читать. TDPL имеет очень интересный подход к шаблонам: Andrei сначала вводит их как естественное расширение синтаксиса функций, классов, и структур, чтобы параметризовать код. И только потом вводятся шаблоны-как-области-видимости. Следующая таблица перечисляет главы, которые имеют дело с шаблонами.

Таблица 13. Главы TDPL, имеющие дело с шаблонами

Тема	Глава
Шаблоны функций	5.3, 5.4, 5.10, и 5.12
Шаблоны классов	6.14
Шаблоны структур	7.1.10
`alias`	7.4
Обычные шаблоны	7.5
Шаблоны Mixin	7.6
Перегрузка операторов	12

Programming in D

Другая очень интересная книга является переводом с турецкого книги Ali Cehreli, `Programming in D`, которую вы можете найти здесь в свободном доступе: http://ddili.org/ders/d.en/index.html (*На Хабрахабре есть русский перевод нескольких начальных разделов — прим. пер.*). Конкретнее о том, что касается этого документа, главу о шаблонах можно найти по адресу http://ddili.org/ders/d.en/templates.html. Книга от Ali ставит целью введение в программирование на D, так что её ритм менее восторженный, у чем TDPL. Если вам неохота читать настоящий документ, глава по ранее указанной ссылке является наилучшим выбором в качестве учебника по шаблонам.

D Wiki

За исключением книг, на D wiki имеется урок о шаблонах, который можно найти по адресу http://wiki.dlang.org/Introduction_to_D_templates, он очень тщательно объясняет std.functional.unaryFun, попутно имея дело с различными темами, как, например, шаблонами, строками mixin, и областью видимости экземпляра. Я нашел его очень приятным для чтения и хорошо объясняющим трудную часть Phobos'a.

Метапрограммирование D

Я обнаружил статью Nick Sabalausky's по адресу http://www.semitwist.com/articles/EfficientAndFlexible/SinglePage/, которая была очень хорошим чтением, и полным изящных идей. В частности, Nick показывает, как использовать информацию времени компиляции для обработки кода таким образом, чтобы аргументы времени выполнения использовались для "влияния" на значения времени компиляции. Да, вы правда читаете это, и мое описание не воздаёт ему должное: прочитайте и убедитесь в этом сами. Я определенно должен разместить раздел где-то в этом документе на эту тему.

Код, показанный в статье, можно найти в проекте на github, по адресу https://github.com/Abscissa/efficientAndFlexible.