Декларативное метапрограммирование:

Обработка списков на этапе компиляции

Вадим Винник

Пролегомены

Три источника и три составных части

- Аппарат шаблонов: специализации, SFINAE.
- Поиск с возвратом, сопоставление с образцом.
- Типы как данные.

- Метапрограммирование.
- Декларативное программирование.
- Вычисления на этапе компиляции.

… плюс практическая польза

Метапрограммирование

- В широком смысле - создание кода, управляющего созданием кода.
  - [Кстати] Трансляторы как “программирующие программы”.
- Различие между обобщённым (generic) программированием и метапрограммированием - определяется прагматикой.
  - Воплощая обобщённую функцию наподобие поиска в контейнере, программист обычно не воспринимает это как метапрограммирование.
- Нестрогий, но разумный критерий: при метапрограммировании от компилятора ожидают принятия нетривиальных решений.
  - Например, выбрать алгоритм сортировки или поиска в зависимости от того, поддерживает ли контейнер произвольный доступ.
- В C++ часто (но не всегда) основывается на шаблонах (template),
  - которые образуют Тьюринг-полный “язык в языке”.

Декларативное программирование

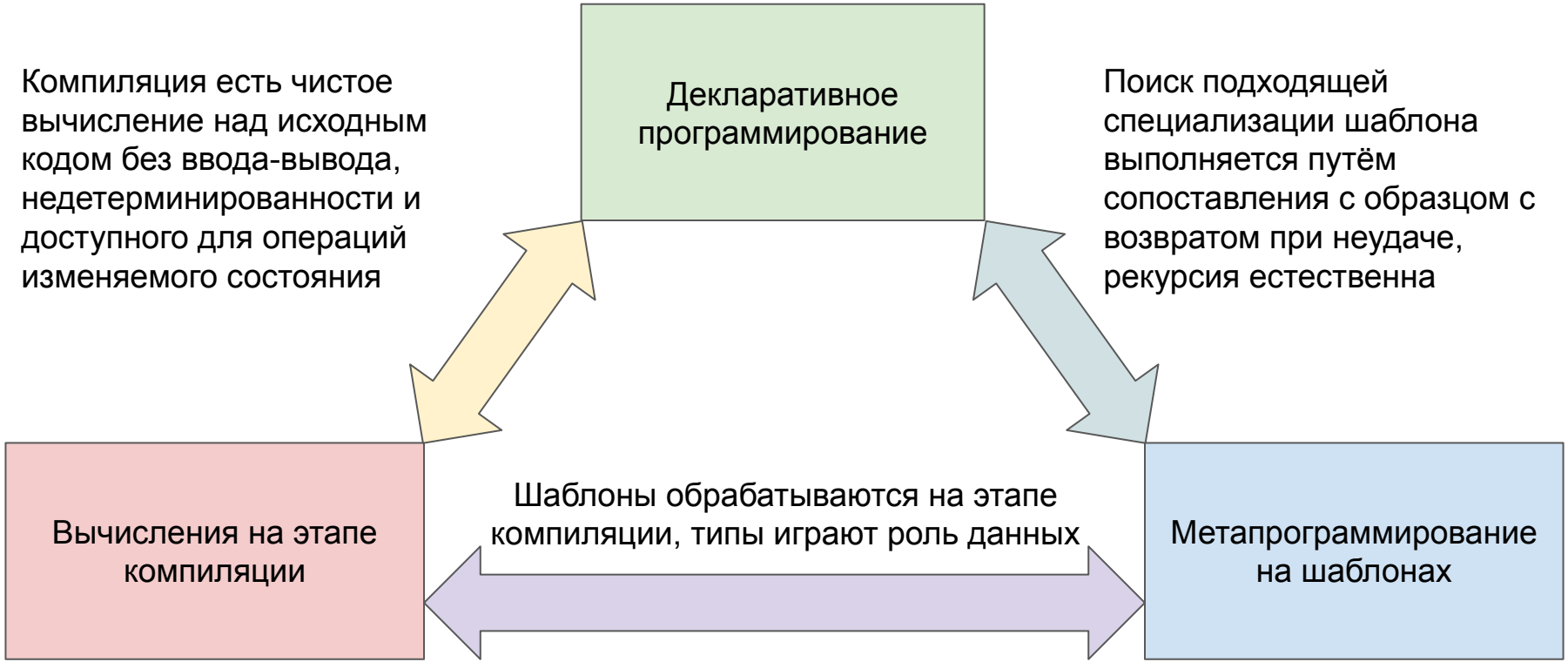
- Программа описывает требования к искомому результату…
  - … а не пошаговый способ его получения.
  - Система сама выбирает способ построения искомого результата.
- Диалектическое отрицание императивной парадигмы.
- Нет изменяемого состояния:
  - отсутствуют операторы присваивания.
  - вместо циклов используется рекурсия.
- Д.п. часто предполагает чистые вычисления,
  - детерминированность, отсутствие ввода-вывода,
  - гарантия одинаковых результатов при одинаковых входных данных.
- Типичны: сопоставление с образцом и поиск с возвратом (backtracking).

Вычисления на этапе компиляции

- В процессе компиляции кода выполняются нетривиальные вычисления.
- Их результат подставляется в исполняемый код в качестве данных.
- Исходный код служит инструкцией по вычислениям двоякого рода:
  - на этапе компиляции - для транслятора как абстрактной машины;
  - на этапе выполнения - для физического процессора или виртуальной машины.
- На этап компиляции удобно переносить вычисления констант и таблиц.
  - Контрольные суммы строк, таблицы тригонометрических функций, таблицы CRC…
- Альтернатива - генерация кода:
  - вспомогательная программа вычисляет константы и генерирует код с их объявлениями;
  - необходимость тестирования, отладки, поддержки вспомогательного кода;
  - усложнение сборки.
- Вычисление констант интегрировано в код, который их использует.

Синтез

Компиляция есть чистое вычисление над исходным кодом без ввода-вывода, недетерминированности и доступного для операций изменяемого состояния



Поиск подходящей специализации шаблона выполняется путём сопоставления с образцом с возвратом при неудаче, рекурсия естественна

Вычисления на этапе компиляции

Факториал двумя способами

```
constexpr unsigned long long factorial_old(int const n)
    { return n == 0 ? 1 : n * factorial_old(n - 1); }

constexpr unsigned long long factorial_new(int n) {
    unsigned long long r = 1;
    while (n != 0) {
      r *= n;
      --n;
    }
    return r;
}
```

Контрольная сумма строки

```
constexpr std::uint32_t adler32_helper(
    unsigned char const* s,
    uint32_t a,
    uint32_t b)
{
  static constexpr std::uint16_t prime = 65521;
  auto const c = static_cast<std::uint32_t>(*s);
  return 0 == c
    ? (a | (b << 16))
    : adler32_helper(s+1, (a + c) % prime, (a + b + c) % prime);
}
```

Поиск строки в списке (1)

- Пусть каждая функция в приложении пишет сообщение в лог.
- Первым аргументом в функцию log передаётся имя функции…
- … известное на этапе компиляции.
- Некоторые функции считаются “интересными”.
- Список имён интересных функций жёстко задан в коде…
- … т.е. известен во время компиляции.
- Логгер должен игнорировать сообщения от неинтересных функций.
- Поиск в контейнере “интересных” имён на этапе выполнения - долго.
- Нужно сделать распознаватель вхождения строки в контейнер, работающий на этапе компиляции.

Поиск строки в списке (2)

```
constexpr bool str_eq(
    char const* const s,
    char const* const t)
{
  return (*s == *t)
      && (*s == 0 || str_eq(s + 1, t + 1));
}

constexpr char const* s1 = "A this is a string";
constexpr char const* s2 = "B this is a string";
static_assert(str_eq(s1 + 2, s2 + 2), "Must be recognized equal");
```

Поиск строки в списке (2)

```
constexpr bool str_in(
    char const* const s,
    char const* const* const ps,
    char const* const* const ps_end)
{
  return (ps != ps_end)
      && (str_eq(s, *ps) || str_in(s, ps + 1, ps_end));
}
```

Поиск строки в списке (3)

```
template <std::size_t N>
constexpr bool str_in(
    char const* const s, char const* const (&ps)[N])
{
  return str_in(s, &(ps[0]), &(ps[N]));
}

constexpr char const* const patterns[] { "First", "Second" };

static_assert( str_in("First", patterns), "must be found");
static_assert(!str_in("Third", patterns), "must be missing");
```

Извлечение уроков

- На этапе компиляции можно вычислить много полезного.
- Код, предназначенный для вычислений на этапе выполнения и на этапе компиляции, пишется на одном языке и располагается в одном модуле.
- Значения, вычисленные на этапе компиляции, *имманентны* остальному коду, описывающему этап выполнения
  - А не *трансцендентны*, как в случае генерации кода вспомогательной программой.
- Вместо циклов с изменяемым состоянием используется рекурсия и аккумуляторы промежуточных результатов.
- Стиль написания кода, работающего на этапе компиляции, несколько специфичен.

Декларативное программирование

Ключевые понятия

- Сопоставление с образцом.
- Поиск с возвратом (backtracking).
- Рекурсия.

Основные характеристики

- Подходит для работы с объектами сложной структуры.
- Подходит для рекуррентно определённых математических понятий.
- Рекурсия хорошо моделирует математическую индукцию.
- Популярные языки: Lisp, Prolog, Haskell.
- Всё больше проникает в C++.

Аксиомы Пеано

1. Существует натуральное число **z**, называемое *нулём*.
2. Для каждого натурального числа *x* однозначно определенно натуральное число **s**(*x*), называемое *непосредственно следующим за x*.
3. Число **z** не следует непосредственно ни за каким натуральным числом.
4. Каждое натуральное число непосредственно следует не более чем за одним натуральным числом.
5. Любое подмножество M множества натуральных чисел N, содержащее **z**, и вместе с каждым числом *x* из M содержащее **s**(*x*), совпадает с множеством N.

Рекуррентные определения операций

- $x + \mathbf{z} = x$. сложение: база
- $x + \mathbf{s}(y) = \mathbf{s}(x + y)$. сложение: шаг
- $x \times \mathbf{z} = \mathbf{z}$. умножение: база
- $x \times \mathbf{s}(y) = (x \times y) + x$. умножение: шаг

	1 + 2	расшифровать числа через **z** и **s**
=	**s**(**z**) + **s**(**s**(**z**))	**s**(**s**(**z**)) есть **s**(y) при y=**s**(**z**)
=	**s**(**s**(**z**) + **s**(**z**))	второе слагаемое **s**(**z**) есть **s**(y) при y=**z**
=	**s**(**s**(**s**(**z**) + **z**))	дошли до базы рекурсии
=	**s**(**s**(**s**(**z**)))	перевод в привычную запись
=	3	окончательный результат

Реализация арифметики Пеано на Prolog

```
convert(z, 0).
convert(s(X), M) :- convert(X, N), M is N+1.

add(z, Y, Y).
add(s(X), Y, s(Z)) :- add(X, Y, Z).

mul(z, _, z).
mul(s(X), Y, Z) :- mul(X, Y, T), add(Y, T, Z).

factorial(z, s(z)).
factorial(s(X), Y) :- factorial(X, T), mul(s(X), T, Y).
```

Реализация арифметики Пеано на Prolog (2)

```
convert(z, 0).
convert(s(X), M) :- convert(X, N), M is N+1.

add(z, Y, Y).
add(s(X), Y, s(Z)) :

mul(z, _, z).
mul(s(X), Y, Z) :- mul(X, Y, T), add(Y, T, Z).

factorial(z, s(z)).
factorial(s(X), Y) :- factorial(X, T), mul(s(X), T, Y).
```

```
?- convert(X, 6).
X = s(s(s(s(s(s(z))))))
?- convert(s(s(s(s(s(s(z)))))), N).
N = 6
```

Реализация арифметики Пеано на Prolog (3)

```
convert(z, 0).
convert(s(X), M) :- convert(X, N), M is N+1.

add(z, Y, Y).
add(s(X), Y, s(Z)) :- add(X, Y, Z).

mul(z, _, z).
mul(s(X), Y, Z) :- m

factorial(z, s(z)).
factorial(s(X), Y) :- factorial(X, T), mul(s(X), T, Y).
```

```
?- add(s(s(s(s(z)))), s(s(s(z))), R).
R = s(s(s(s(s(s(s(z)))))))
```

Реализация арифметики Пеано на Prolog (4)

```
convert(z, 0).
convert(s(X), M) :- convert(X, N), M is N+1.

add(z, Y, Y).
add(s(X), Y, s(Z)) :

mul(z, _, z).
mul(s(X), Y, Z) :- mul(X, Y, T), add(Y, T, Z).

factorial(z, s(z)).
factorial(s(X), Y) :- factorial(X, T), mul(s(X), T, Y).
```

```
?- mul(s(s(z)), Y, s(s(s(s(s(s(z)))))).
Y = s(s(s(z)))
```

Реализация арифметики Пеано на Prolog (5)

```
convert(z, 0).
convert(s(X), M) :- convert(X, N), M is N+1.

add(z, Y, Y).
add(s(X), Y, s(Z)) :

mul(z, _, z).
mul(s(X), Y, Z) :- m

factorial(z, s(z)).
factorial(s(X), Y) :- factorial(X, T), mul(s(X), T, Y).
```

```
?- projection([R]),
   convert(X, 9),
   factorial(X, T),
   convert(T, R).
R = 362880
```

Реализация арифметики Пеано на Haskell

```
data Natural = Z | S(Natural)  deriving (Show)

fromNatural Z = 0
fromNatural (S x) = 1 + (fromNatural x)
toNatural 0 = Z
toNatural x  = S (toNatural (x - 1))

add Z y = y
add (S x) y = S (add x y)

mul Z _ = Z
mul (S x) y = add y (mul x y)
```

Извлечение уроков

- Декларативная программа состоит из определений или правил.
- Правило может охватывать частный случай…
  - … т.е. требовать, чтобы входные данные имели какой-то определённый вид;
  - например, вид s(*x*), где *x* - переменная, или вид z (константа).
- Для применения правила к конкретному аргументу делается попытка сопоставить аргумент с образцом в левой части правила.
  - При успешном сопоставлении переменные из образца получают конкретные значения.
  - При неуспешном сопоставлении - делается попытка применить следующее правило.
- Чтобы реализовать алгоритм в декларативном стиле, его нужно математически описать в рекуррентной форме.
  - Тогда программа будет записью математического определения в других обозначениях.

Важно: функции высшего порядка

- Очень характерно для декларативного программирования в целом
  - и для функционального в особенности.
- Аргументом функции f может быть функция h.
- Функция f управляет применением функции h.
- Пример: функция фильтрации принимает предикат

```
filter :: (a -> Bool) -> [a] -> [a]
filter _ [] = []
filter p (x:xs) = if p x then x:ys else ys
                  where ys = filter p xs
```

- Функции имеют тот же статус, что и объекты данных.

Метапрограммирование на шаблонах

Снова арифметика Пеано

- На практике бесполезно, но уж очень занимательно :-)
- Числа моделировать **типами**.
  - Тип `zero` моделирует число 0.
  - Если `T` - тип, соответствующий числу n, то `succ<T>` соответствует n+1.
- Операции над числами - моделировать операциями **над типами**.
- Операция над типами - это шаблон.
- Шаблон принимает типы в качестве параметров и строит новый тип.
- Например, шаблон `add` из типов, моделирующих числа 3 и 7, должен построить тип, который соответствует числу 10.
- Интересно сравнить с решениями на языках Prolog и Haskell.

Арифметика на типах: числа как типы

```
using value_t = unsigned long long int;

struct zero {
  static constexpr value_t value = 0;
};

template <typename T>
struct succ {
  static constexpr value_t value =
      1 + T::value;
};
```

```
using _0 = zero;
using _1 = succ<_0>;
using _2 = succ<_1>;
using _3 = succ<_2>;
using _4 = succ<_3>;
using _5 = succ<_4>;
using _6 = succ<_5>;
using _7 = succ<_6>;
using _8 = succ<_7>;
using _9 = succ<_8>;
```

Операция сложения

```
template <typename X, typename Y> struct add; // не определён

template <typename X>
struct add<X, zero> { // частный случай: второе слагаемое есть 0
    using result = X;
};

template <typename X, typename Y>
struct add<X, succ<Y>> { // частный случай 2: сложение с n+1
    using result = succ<typename add<X, Y>::result>;
};

static_assert(add<_8, _9>::result::value == 17);
```

Операция сложения

```
template <typename X, typename Y> struct add; // не определён

template <typename X>
struct add<X, zero> { // частный с
    using result = X;
};

template <typename X, typename Y>
struct add<X, succ<Y>> { // частны
    using result = succ<typename add<X, Y>::result>;
};

static_assert(add<_8, _9>::result::value == 17);
```

```
add(z, Y, Y).
```

```
add Z y = y
```

```
add(s(X), Y, s(Z)) :- add(X, Y, Z).
```

```
add (S x) y = S (add x y)
```

Операция умножения

```
template <typename X, typename Y> struct mul;

template <typename X>
struct mul<X, zero> {
    using result = zero;
};

template <typename X, typename Y>
struct mul<X, succ<Y>> {
    using result = typename add<typename mul<X, Y>::result, X>::result;
};

static_assert(mul<_8, _9>::result::value == 72);
```

Операция умножения

```
template <typename X, typename Y> struct mul;

template <typename X>
struct mul<X, zero> {
    using result = zero;
};

template <typename X, typename Y>
struct mul<X, succ<Y>> {
    using result = typename add<typename mul<X, Y>::result, X>::result;
};

static_assert(mul<_8, _9>::result::value == 72);
```

```
mul(z, _, z).
```

```
mul Z _ = Z
```

```
mul(s(X), Y, Z) :- mul(X, Y, T), add(Y, T, Z).
```

```
mul (S x) y = add y (mul x y)
```

Извлечение уроков

- Типы используются не для того, чтобы работать с объектами этих типов на этапе выполнения, а в качестве данных для компилятора.
- Логика метакода построена на правилах выбора специализации.
- Помимо общего определения шаблона должны существовать специализации, отвечающие за частные случаи.
- Определение для общего случая “хуже” частного с точки зрения алгоритма выбора специализации.
- Компилятор пытается применять специализации, сопоставляя типы-аргументы с образцами в заголовках шаблона.
- Метапрограммирование на шаблонах совпадает с программированием на декларативным языках с точностью до обозначений.

Вариадические шаблоны

- Вариадическим называется шаблон функции или класса, имеющий заранее неизвестное число параметров.
- Кортеж из произвольного числа (в том числе 0) разнородных параметров называется пакетом параметров.
- Вариадический шаблон обладает хотя бы одним пакетом параметров.
- Объявление вариадического шаблона класса:
  - ```
    template <typename... Args>
    class variadic_demo { /*. . . */ };
    ```
- Экземпляры вариадического шаблона:
  - ```
    using t1 = variadic_demo<std::string>;
    ```
  - ```
    using t2 = variadic_demo<void, void, void>;
    ```
    ```

Что делать с пакетом параметров?

- Разбирать по одному рекурсивно (на этапе компиляции, не выполнения).
- Общее объявление шаблона (часто оставляют без реализации):

```
template <typename... Args>
struct example;
```

- База рекурсии: специализация с одним параметром или без параметров.

```
template <>
struct example<> { /* … */ };
```

- Шаг рекурсии - отщепление первого параметра от пакета.

```
template <typename Head, typename... Tail>
struct example<Head, Tail...> { …example<Tail>… };
```

- Базу (реже - шаг) часто совмещают с общим определением.

Произведение типов: примитивное решение

```
template <typename... Args>
struct product {};

template <
    typename Head,
    typename... Tail>
struct product<Head, Tail...> {
  using first_t = Head;
  using others_t = product<Tail...>;
  first_t first;
  others_t others;
};
```

Произведение типов: примитивное решение

```
template <typename... Args>
struct product {};

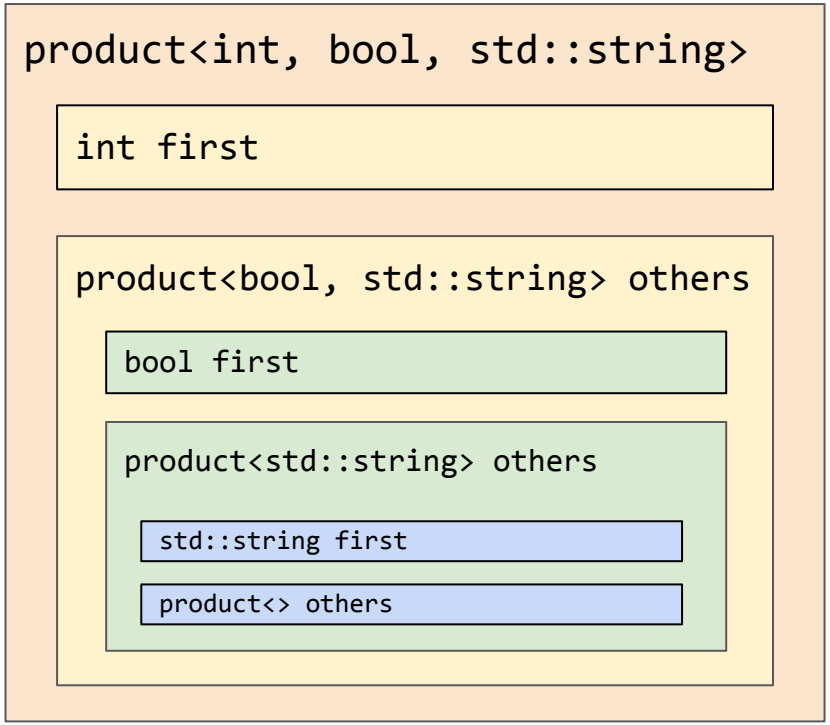
template <
    typename Head,
    typename... Tail>
struct product<Head, Tail...> {
  using first_t = Head;
  using others_t = product<Tail...>;
  first_t first;
  others_t others;
};
```

- Общее определение совмещено с базой рекурсии.
- Компилятор придёт сюда только если пакет Args пуст.
- Иначе его можно представить в виде `<Head, Tail...>`.
- Тогда бы его перехватила специализация.

Произведение типов: примитивное решение

```
template <typename... Args>
struct product {};

template <
    typename Head,
    typename... Tail>
struct product<Head, Tail...> {
  using first_t = Head;
  using others_t = product<Tail...>;
  first_t first;
  others_t others;
};
```



Произведение типов: примитивное решение

```
template <typename... Args>
struct product {};

template <
    typename Head,
    typename... Tail>
struct product<Head, Tail...> {
  using first_t = Head;
  using others_t = product<Tail...>;
  first_t first;
  others_t others;
};
```

```
product<int, bool, std::string> r;
r.first = 9;
r.others.first = true;
r.others.others.first = "a";
```

Это была пре-ам-бу-ла. А теперь начнется амбула.

А. и Б. Стругацкие. “Полдень, XXII век”

Постановка задачи

- Для приложений может понадобиться вычисление таблиц значений:
  - таблица простых чисел;
  - таблица тригонометрических функций (с фиксированной запятой);
  - таблица числа единиц в двоичном представлении 8-битных целых беззнаковых;
  - поразрядное представление длинного целого числа.
- Нужны инструменты обработки списков значений на этапе компиляции.
- Пример существующей библиотеки в близкой области - Boost.MPL.
  - Поддерживает множество структур данных (список, множество, массив).
  - Содержит множество алгоритмов (сортировка, фильтрация, свёртка и развёртка).
  - Поддерживает обработку как списков типов, так и списков значений.
  - Объёмна и сложна в освоении.
- Здесь показано пошаговое построение компактной библиотеки.

Элементарные определения

```
template <typename T, T... Xs>
struct list {
  using item_type = T;
  static constexpr auto size = sizeof...(Xs);
};

// тип U должен иметь вид list<T, T... Xs>

template <typename U> using item_t = typename U::item_type;

template <typename U> constexpr auto size_v = U::size;

template <typename U> constexpr bool is_empty_v = (size_v<U> == 0);
```

Элементарные определения

```
template <typename T, T... Xs>
struct list {
  using item_type = T;
  static constexpr auto size = sizeof...(Xs);
};

// тип U должен иметь вид list<T, T... Xs>

template <typename U> using item_t = typename U::item_type;

template <typename U> constexpr auto size_v = U::size;

template <typename U> constexpr bool is_empty_v = (size_v<U> == 0);
```

```
using s = list<int, 10, 20, 30>;

static_assert(size_v<s> == 3);

static_assert(!is_empty_v<s>);
```

Элементарные определения

```
template <typename T, T... Xs>
struct list {
  using item_type = T;
  static constexpr auto size = sizeof...(Xs);
};

// U should be list<T, T...

template <typename U> us

template <typename U> co

template <typename U> co
```

- Рекурсия для подсчёта длины списка не нужна.
- В отличие от реализаций на языках Lisp, Haskell, Prolog и др.
- ```
  length :: [a] -> Integer
  length []     = 0
  length (x:xs) = 1 + (length xs)
  ```
- В C++ длина пакета параметров дана непосредственно.
  ```

Простейшие конструирующие операции

```
template <typename U, item_t<U> X>
struct append; // компилятор никогда не достигает общего случая

template <typename T, T... Xs, T X>
struct append<list<T, Xs...>, X>
  { using type = list<T, Xs..., X>; }; // prepend - аналогично

template <typename U, typename V>
struct concat; // компилятор никогда не достигает общего случая

template <typename T, T... Xs, T... Ys>
struct concat<list<T, Xs...>, list<T, Ys...>>
  { using type = list<T, Xs..., Ys...>; };
```

Простейшие конструирующие операции

```
template <typename U, item_t<U> X>
struct append; // компилятор никогда

template <typename T, T... Xs, T X>
struct append<list<T, Xs...>, X>
  { using type = list<T, Xs..., X>;

template <typename U, typename V>
struct concat; // компилятор никогда не достигает общего случая

template <typename T, T... Xs, T... Ys>
struct concat<list<T, Xs...>, list<T, Ys...>>
  { using type = list<T, Xs..., Ys...>; };
```

```
using s = list<int, 10, 20, 30>;

using t = append<s, 40>;
using u = list<int, 10, 20, 30, 40>;
static_assert(std::is_same<t, u>::value);

using v = concat<s, t>;
using w = list<10, 20, 30, 10, 20, 30, 40>;
static_assert(std::is_same<v, w>::value);
```

Простейшие конструирующие операции

```
template <typename U, item_t<U> X>
struct append; // компилятор никогда не достигает общего случая

template <typename T, T... Xs, T X>
struct append<list<T, Xs...>, X>
  { using type = list<T, Xs..., X>; }; // prepend - аналогично

template <typename U, ty
struct concat; // компил

template <typename T, T.
struct concat<list<T, Xs
  { using type = list<T,
```

- Добавление элемента в конец списка не требует рекурсии.
- Конкатенация списков не требует рекурсии.
- Для сравнения - реализация на Haskell

```
concat :: [a] -> [a] -> [a]
concat [] ys = ys
concat (x:xs) ys = x : (concat xs ys)
```

Простейшие операции-анализаторы

```
template <typename U>
struct uncons; // общий случай оставлен без определения

template <typename T, T X, T... Xs>
struct uncons<list<T, X, Xs...>>
{
  static constexpr T head = X;
  using tail = list<T, Xs...>;
};

template <typename U> using tail_t = typename uncons<U>::tail;
template <typename U> constexpr item_t<U> head_v = uncons<U>::head;
```

Простейшие операции-анализаторы

```
template <typename U>
struct uncons; // общий случай

template <typename T, T X, T...
struct uncons<list<T, X, Xs...
{
  static constexpr T head = X;
  using tail = list<T, Xs...>;
};

template <typename U> using tail_t = typename uncons<U>::tail;
template <typename U> constexpr item_t<U> head_v = uncons<U>::head;
```

```
using s = list<int, 10, 20, 30>;

static_assert(head_v<s> == 10);

using t = list<int, 20, 30>;
static_assert(std::is_same<tail_t<s>, t>::value);
```

Простейшие операции-анализаторы

```
template <typename U>
struct uncons; // общий случай оставлен без определения

template <typename T, T X, T... Xs>
struct uncons<list<T, X, Xs...>>
{
  static constexpr T head = X;
  using tail = list<T, Xs...>;
};

template <typename U> using tail
template <typename U> constexpr
```

- Требуется непустой список:
  - в противном случае - ошибка компиляции.
- Операция uncons вычисляет сразу два результата:
  - голову и хвост.

Первая сложность: деконструкция справа

- Вместо головы и хвоста - последний элемент и начало.
- Встроенные механизмы языка не позволяют разобрать список с конца.
  - Пакет параметров должен стоять последним среди параметров шаблона.
- Остаётся рекурсивное решение.

```
template <typename U>
struct unconsr; // общий случай оставлен без определения

template <typename U>
using init_t = typename unconsr<U>::init;

template <typename U>
constexpr item_t<U> last_v = unconsr<U>::last;
```

Первая сложность: деконструкция справа

```
template <typename T, T X>
struct unconsr<list<T, X>> {
  static constexpr T last = X;
  using init = list<T>;
};

template <typename T, T X, T... Xs>
struct unconsr<list<T, X, Xs...>> {
private:
  using remainder = list<T, Xs...>;
public:
  static constexpr T last = last_v<remainder>;
  using init = prepend_t<init_t<list<T, Xs...>>, X>;
};
```

Первая сложность: деконструкция справа

```
template <typename T, T X>
struct unconsr<list<T, X>> {
  static constexpr T last = X;
  using init = list<T>;
};

template <typename T, T X, T... Xs>
struct unconsr<list<T, X, Xs...>> {
private:
  using remainder = list<T, Xs...>;
public:
  static constexpr T last = last_v<remainder>;
  using init = prepend_t<init_t<list<T, Xs...>>, X>;
};
```

- База рекурсии - список из одного элемента.
- Единственный элемент и есть последний.
- Начало есть пустой список.

- Шаг рекурсии - список произвольной длины…
- … отличной от 1 (перехватывается выше).
- Последний элемент хвоста есть последний элемент всего списка.
- Начало списка есть начало его хвоста с приписанным слева первым элементом.

Первая сложность: деконструкция справа

```
template <typename T, T X>
struct unconsr<list<T, X>> {
  static constexpr T last = X;
  using init = list<T>;
};

template <typename T, T X, T... Xs>
struct unconsr<list<T, X, Xs...>> {
private:
  using remainder = list<T, Xs...>;
public:
  static constexpr T last = last_v<remainder>;
  using init = prepend_t<init_t<list<T, Xs...>>, X>;
};
```

```
using s = list<int, 10, 20, 30>;

static_assert(last_v<s> == 30);

using t = list<int, 10, 20>;
static_assert(std::is_same<init_t<s>, t>::value);
```

Переворачивание списка

```
template <typename U>
struct revert
{
  using type = append_t<
       revert<tail_t<U>>::type,
       head_v<U>>;
};

template <typename T>
struct revert<list<T>>
{
  using type = list<T>;
};
```

Переворачивание списка

```
template <typename U>
struct revert
{
  using type = append_t<
      revert<tail_t<U>>::type,
      head_v<U>>;
};

template <typename T>
struct revert<list<T>>
{
  using type = list<T>;
};
```

- Шаг рекурсии - непустой список.
- Чтобы перевернуть непустой список, нужно:
  - отделить первый элемент;
  - перевернуть остаток;
  - поставить изъятый элемент в конец.
- Шаг рекурсии совмещён с общим определением.
- Сопоставления с образцом не нужно.
  - Вместо него применены функции-анализаторы.

- База рекурсии - пустой список.
- Инверсия пустого списка есть пустой список.

Переворачивание списка

```
template <typename U>
struct revert
{
  using type = append_t<
      revert<tail_t<U>>::type,
      head_v<U>>;
};

template <typename T>
struct revert<list<T>>
{
  using type = list<T>;
};
```

```
using s = list<int, 10, 20, 30>;

using t = list<int, 30, 20, 10>;
static_assert(std::is_same<revert_t<s>, t>::value);
```

Разделение по предикату

- На вход поступают список U и предикат P.
- С выхода ожидаются два списка: V и W.
  - V - элементы списка U, удовлетворяющие предикату P; W - не удовлетворяющие.
- Предикат - это шаблон:
  - в параметр шаблона подставляется значение аргумента;
  - в инстанцированном шаблоне должен содержаться член-constexpr value типа bool.
- Логика реализации:
  - `partition` есть фасад над движком `partition_loop`;
  - `partition_loop` распознаёт пустой список U и завершает рекурсию;
  - иначе берёт из U первый элемент X, вычисляет P(X) и вызывает `partition_loop_if`;
  - `partition_loop_if` кладёт элемент X либо в V, либо в W и вызывает `partition_loop`;
  - промежуточное состояние передаётся через аргументы-накопители.

Разделение по предикату

```
template <
  typename U, // остаток списка
  typename V, // удовлетворяют P
  typename W, // нарушают P
  template <auto> typename P,
  auto X,     // текущий элемент
  bool B>     // значение P(X)
struct partition_loop_if
{ // только если B есть true
  using step = partition_loop<
      U, append_t<V, X>, W, P>;
};
```

```
template <
  typename U,
  typename V,
  typename W,
  template <auto> typename P,
  auto X>
struct partition_loop_if<
    U, V, W, P, X, false>
{
  using step = partition_loop<
      U, V, append_t<W, X>, P>;
};
```

Разделение по предикату

```
template <
  typename U, // ещё не обработанный остаток списка
  typename V, // накопитель элементов, удовлетворяющих P
  typename W, // накопитель элементов, не удовлетворяющих P
  template <auto> typename P>
struct partition_loop { // только если список не пуст
private:
  using step = typename partition_loop_if<
    tail_t<U>, V, W, P, head_v<U>, P<head_v<U>>::value>::step;
public:
  using left = typename step::left;
  using right = typename step::right;
```

Разделение по предикату

```
template <
  typename T,
  typename V,
  typename W,
  template <auto> typename P>
struct partition_loop<list<T>, V, W, P> { // список-аргумент исчерпан
  using left = V;   // окончательный результат находится в накопителях
  using right = W;
};

template <typename U, template <auto> typename P, typename T = item_t<U>>
using partition = partition_loop<U, list<T>, list<T>, P>;
```

Разделение по предикату: пример использования

```
template <int X>
struct is_odd
{
    static constexpr bool value = (X % 2) != 0;
}; // or using is_odd = std::bool_constant<(X % 2) != 0>;

using u = list<int, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9>;
using p = partition<u, is_odd>;
static_assert(
    std::is_same<typename p::left, list<int, 1, 3, 5, 7, 9>>::value);
static_assert(
    std::is_same<typename p::right, list<int, 0, 2, 4, 6, 8>>::value);
```

Ещё одна похожая метафункция: span

- Принимает список U и предикат P.
- Вычисляет два списка: V и W.
- V есть префикс списка U наибольшей длины, в котором все элементы удовлетворяют P.
- W - остаток списка U, начиная с первого элемента, нарушающего P.

```
using u = list<int, 0, 1, 2, 3, 4, 5, 0, 0, 0, 0>;

using p = span<u, is_less_than_5>;

using v = typename p::left;    // 0, 1, 2, 3, 4
using w = typename p::right;   // 5, 0, 0, 0, 0
```

Поэлементное преобразование списка

- Широко распространено в функциональных языках
  - Например, map и fmap в Haskell.
- Наличие такой функции делает контейнерный тип *функтором*.
- Метафункция fmap принимает список U и метафункцию F.
- Строит список результатов применения F к элементам U.
- Реализация без рекурсии - язык C++ предоставляет всё необходимое.

Как же выглядит текст реализации?

Поэлементное преобразование списка

- Широко распространено в функциональных языках
  - Например, map и fmap в Haskell.
- Наличие такой функции делает контейнерный тип *функтором*.
- Метафункция fmap принимает список U и метафункцию F.
- Строит список результатов применения F к элементам U.
- Реализация без рекурсии - язык C++ предоставляет всё необходимое.

```
template <typename R, typename T, T... Xs, template <auto> typename F>
struct fmap<R, list<T, Xs...>, F> {
  using result = list<R, F<Xs>::value...>;
};
```

Сшивание двух списков

- Метафункция zip_with принимает два списка, X и Y, и метафункцию F двух аргументов.
- Строит список вида $F(X_1, Y_1)$, …, $F(X_n, Y_n)$.
- Длина результата равна наименьшей из длин списков-аргументов.

```
using xs = list<int, 0, 1, 2, 3, 4>;
using ys = list<int, 5, 6, 7, 8>;

template <int A, int B>
struct plus { static constexpr int value = A + B; };

using zs = zip_with<xs, ys, int, plus>; // 5, 7, 9, 11
```

Сшивание двух списков (фрагмент реализации)

```
template <
  typename S, S X, S... Xs,
  typename T, T Y, T... Ys,
  typename R, template <auto, auto> typename F>
struct zip_with<list<S, X, Xs...>, list<T, Y, Ys...>, R, F> {
private:
  static constexpr auto head = F<X, Y>::value;
  using tail = zip_with_t<list<S, Xs...>, list<T, Ys...>, R, F>;
public:
  using result = prepend_t<tail, head>;
};
```

Сортировка списка

- Метод вставок. Сложность $O(n^2)$.
- Сортировка пустого списка тривиальна.
- Отсортировать список, кроме первого элемента X (рекурсия).
- Найти в отсортированном списке первый элемент, превышающий X.
- Вставить X перед найденным элементом.
- Компаратор передавать в метафункцию сортировки как аргумент.

```
using s = list<int, 0, 9, 1, 8, 2, 7, 3, 6, 4, 5>;

using t = typename sort<s, less>; // 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
```

Сортировка списка (фрагмент реализации)

```
template <typename U, template <auto, auto> typename C>
struct sort { // только если список не пуст
private:
  template <auto Z> // превратить двухместный предикат в одноместный
  struct cmp_curr : C<Z, head_v<U>> {};
  using sorted_tail = sort_t<tail_t<U>, C>; // рекурсия
  using parts = partition<sorted_tail, cmp_curr>; // найти позицию вставки
  using old_right = typename parts::right;
  using new_right = prepend_t<old_right, head_v<U>>; // вставить
public:
  using result = concat_t<typename parts::left, new_right>;
};
```

Свёртка списка

- Рекуррентное определение суммы списка:
  - $s_0 = 0$;
  - $s_{k+1} = s_k + a_{k+1}$.
- Пример: $s_3 = s_2 + a_3 = (s_1 + a_2) + a_3 = ((s_0 + a_1) + a_2) + a_3 = ((0 + a_1) + a_2) + a_3$.
- Это - *левая свёртка* по операции “+” с начальным значением 0.
- *Правая* свёртка - симметрично: $a_1 + (a_2 + (a_3 + 0))$.

```
foldl :: (b -> a -> b) -> b -> [a] -> b

foldl f y [] = y
foldl f y (x:xs) = foldl f (f y x) xs
```

Свёртка списка (фрагмент реализации)

```
template <typename U, auto S, template <auto, auto> typename F>
struct foldl {
  static constexpr auto value =
    foldl_v<tail_t<U>, F<S, head_v<U>>::value, F>;
};

template <typename T, auto S, template <auto, auto> typename F>
struct foldl<list<T>, S, F> {
  static constexpr auto value = S;
};
```

Свёртка списка (фрагмент реализации)

```
template <typename U, auto S, template <auto, auto> typename F>
struct foldl {
  static constexpr auto value =
    foldl_v<tail_t<U>, F<S, head_v<U>>::value, F>;
};

template <typename T, auto S,
struct foldl<list<T>, S, F> {
  static constexpr auto value
};
```

```
using arg = list<int, 0, 1, 2, 4, 8>;
constexpr int result = foldl_v<arg, 0, plus>;

constexpr int expected = 0 + 1 + 2 + 4 + 8;

static_assert(result == expected);
```

Развёртка

- Операция, *двойственная* к свёртке.
- Развёртка из исходного значения строит список.
- Входные параметры:
  - исходное состояние S
  - функция F (распознаватель завершения);
  - функция G (генератор элементов-результатов);
  - функция H (преобразователь состояния).
- Алгоритм:
  - если F(S) даёт значение false, завершить;
  - добавить в список-результат значение G(S);
  - продолжить с новым состоянием H(S).

Развёртка (фрагмент)

```
template <
  typename T,
  typename S,
  template <typename> typename F,
  template <typename> typename G,
  template <typename> typename H,
  bool B> // только если B есть true
struct unfoldr_if { // вспомогательная метафункция
  using result = prepend_t<
    unfoldr_t<T, typename H<S>::result, F, G, H>,
    G<S>::value>;
};
```

Развёртка (фрагмент)

```
template <
  typename T,
  typename S,
  template <typename> typename F,
  template <typename> typename G,
  template <typename> typename H>
// база рекурсии: условие продолжения ложно - вернуть пустой список
struct unfoldr_if<T, S, F, G, H, false>
{
  using result = list<T>;
};
```

Фасад развёртки

Все параметры метафункции развёртки удобно запаковать в один класс свойств под понятными для человека именами:

```
template <typename T>
using generate_t = unfoldr_t<
  typename T::item_type,
  typename T::init,
  T::template can_proceed,
  T::template get_item,
  T::template step>;
```

Генератор арифметической прогрессии

```
template <typename T, T A, T B, T D>
struct iota {
  using item_type = T;
  using init = std::integral_constant<T, A>;
  template <typename X>
  using can_proceed = std::bool_constant<(X::value < B)>;
  template <typename X>
  using get_item = X;
  template <typename X>
  struct step
  { using result = std::integral_constant<T, X::value + D>; };
};
```

Генератор арифметической прогрессии

```
template <typename T, T A, T B, T D>
struct iota {
  using item_type = T;
  using init = std::integral_constant<T, A>;
  template <typename X>
  using can_proceed = std::bool_constant<(X::value < B)>;
  template <typename X>
  using get_item = X;
  template <typename X>
  struct step
  { using result = std::integral_constant<T, X::value + D>; };
};
```

Генерируемый список состоит из целых чисел

Генератор арифметической прогрессии

```
template <typename T, T A, T B, T D>
struct iota {
  using item_type = T;
  using init = std::integral_constant<T, A>;
  template <typename X>
  using can_proceed = std::bool_constant<(X::value < B)>;
  template <typename X>
  using get_item = X;
  template <typename X>
  struct step
  { using result = std::integral_constant<T, X::value + D>; };
};
```

Генерируемый список состоит из целых чисел

Начальное значение - параметр A

Генератор арифметической прогрессии

```
template <typename T, T A, T B, T D>
struct iota {
  using item_type = T;
  using init = std::integral_constant<T,
  template <typename X>
  using can_proceed = std::bool_constant<(X::value < B)>;
  template <typename X>
  using get_item = X;
  template <typename X>
  struct step
  { using result = std::integral_constant<T, X::value + D>; };
};
```

Генерируемый список состоит из целых чисел

Начальное значение - параметр A

Продолжать, пока текущее значение не превысило параметр B

Генератор арифметической прогрессии

```
template <typename T, T A, T B, T D>
struct iota {
  using item_type = T;
  using init = std::integral_constant<T,
  template <typename X>
  using can_proceed = std::bool_constant<(X::value < B)>;
  template <typename X>
  using get_item = X;
  template <typename X>
  struct step
  { using result = std::integral_constant<T, X::value + D>; };
};
```

Генерируемый список состоит из целых чисел

Начальное значение - параметр A

Продолжать, пока текущее значение не превысило параметр B

Очередной элемент списка и есть состояние

Генератор арифметической прогрессии

```
template <typename T, T A, T B, T D>
struct iota {
  using item_type = T;
  using init = std::integral_constant<T,
  template <typename X>
  using can_proceed = std::bool_constant<(X::value < B)>;
  template <typename X>
  using get_item = X;
  template <typename X>
  struct step
  { using result = std::integral_constant<T, X::value + D>; };
};
```

Генерируемый список состоит из целых чисел

Начальное значение - параметр A

Продолжать, пока текущее значение не превысило параметр B

Очередной элемент списка и есть состояние

Новое состояние: прибавить D к текущему

Генератор простых чисел

```
template <typename T, unsigned N>
struct primes
{
  using item_type = T;
  using init = prepend_t<generate_t<iota<T, 3, N, 2>>, 2>;

  template <typename S>
  using can_proceed = std::bool_constant<!is_empty_v<S>>;

  template <typename S>
  using get_item = std::integral_constant<T, head_v<S>>;

// продолжение на следующем слайде
```

Генератор простых чисел

```
template <typename T, unsigned N>
struct primes
{
  using item_type = T;
  using init = prepend_t<generate_t<iota<T, 3, N, 2>>, 2>;

  template <typename S>
  using can_proceed = std::bool_constant<!is_empty_v<S>>;

  template <typename S>
  using get_item = std::integral_constant<T, head_v<S>>;

// продолжение на следующем слайде
```

В роли состояния - список чисел (кандидатов)

Начальное состояние: 2 и далее нечётные

Продолжать, пока есть кандидаты

Очередное простое число - первый кандидат

Генератор простых чисел

```
// продолжение

  template <typename S>
  struct step {
  private:
    template <unsigned X>
    struct predicate : std::bool_constant<(X % head_v<S> == 0)> {};
  public:
    using result = typename partition<S, predicate>::right;
  };
};
```

Генератор простых чисел

```
// продолжение

  template <typename S>
  struct step {
  private:
    template <unsigned X>
    struct predicate : std::bool_constant<(X % head_v<S> == 0)> {};
  public:
    using result = typename partition<S, predicate>::right;
  };
};
```

Новое состояние: из списка кандидатов удалить все числа, которые делятся на первое число-кандидат

Генератор простых чисел: пример использования

```
using result = generate_t<primes<unsigned, 500>>;
using expected = list<unsigned,
    2,    3,    5,    7,   11,   13,   17,   19,   23,  29,
   31,   37,   41,   43,   47,   53,   59,   61,   67,  71,
   73,   79,   83,   89,   97,  101,  103,  107,  109,  113,
  127,  131,  137,  139,  149,  151,  157,  163,  167,  173,
  179,  181,  191,  193,  197,  199,  211,  223,  227,  229,
  233,  239,  241,  251,  257,  263,  269,  271,  277,  281,
  283,  293,  307,  311,  313,  317,  331,  337,  347,  349,
  353,  359,  367,  373,  379,  383,  389,  397,  401,  409,
  419,  421,  431,  433,  439,  443,  449,  457,  461,  463,
  467,  479,  487,  491,  499>;
static_assert(std::is_same<result, expected>::value);
```

Интерфейс к этапу выполнения

```
template <typename T, T... Xs>
struct for_each<list<T, Xs...>>
{
  template <typename F>
  static void apply(F f)
  {
    int dummy[] = { (f(Xs), 0)... };
  }
};
```

Интерфейс к этапу выполнения

```
template <typename T, T... Xs>
struct for_each<list<T, Xs...>>
{
  template <typename F>
  static void apply(F f)
  {
    int dummy[] = { (f(Xs), 0)... };
  }
};
```

```
unsigned long s = 0l;
using arg = list<unsigned, 1, 2, 3, 4, 5, 6, 7, 8>;
for_each<arg>::apply([&s](unsigned x) { s = (s << 4) | x; });
REQUIRE(s == 0x1234'5678);
```

Обзор библиотеки

- span
- partition
- sort
- fmap
- zip_with
- for_each
- foldl
- foldr
- unfoldr
- generate

Генераторы последовательностей:

- Целые числа из диапазона.
- Числа Фибоначчи.
- Простые числа (решето Эратосфена)

Тесты

- На этапе компиляции.
- Через static_assert.

`https://github.com/vadimvinnik/compile_time_list`

Не бойтесь метапрограммирования -

наслаждайтесь им!

```
} // the talk
```