

Яндекс

Яндекс

Эволюция метапрограммирования. Как правильно работать со списками типов

Олег Фатхиев

~~Ranges~~

Эволюция метапрограммирования: списки типов

1. ОСНОВЫ

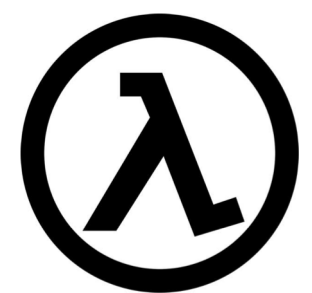


1. Основы: зачем метапрограммирование?

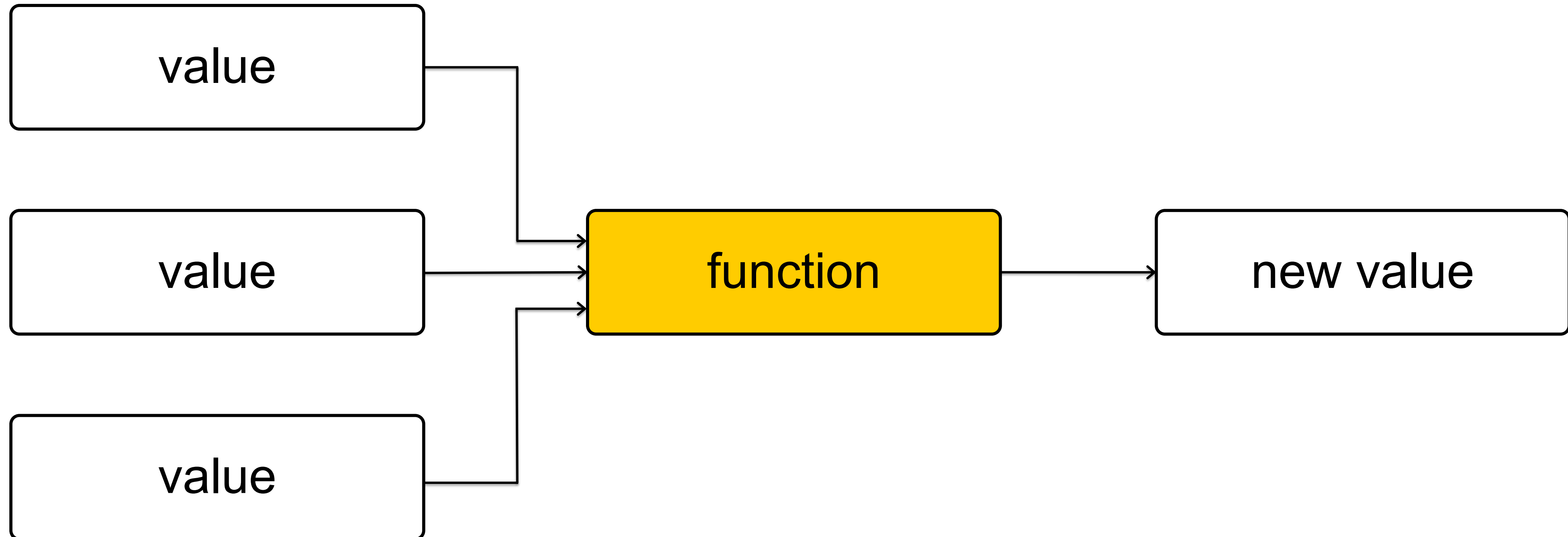
- › Ускорение runtime-а с помощью переноса вычислений в compile time
- › Кодогенерация
- › Condition checking

1. Основы: что такое метапрограммирование?

- › Compile time вычисления
- › Нестандартное использование стандартных языковых конструкций
- › В случае шаблонного метапрограммирования, в основе лежит функциональное программирование (со всеми вытекающими принципами)



1. Основы: compile time вычисления



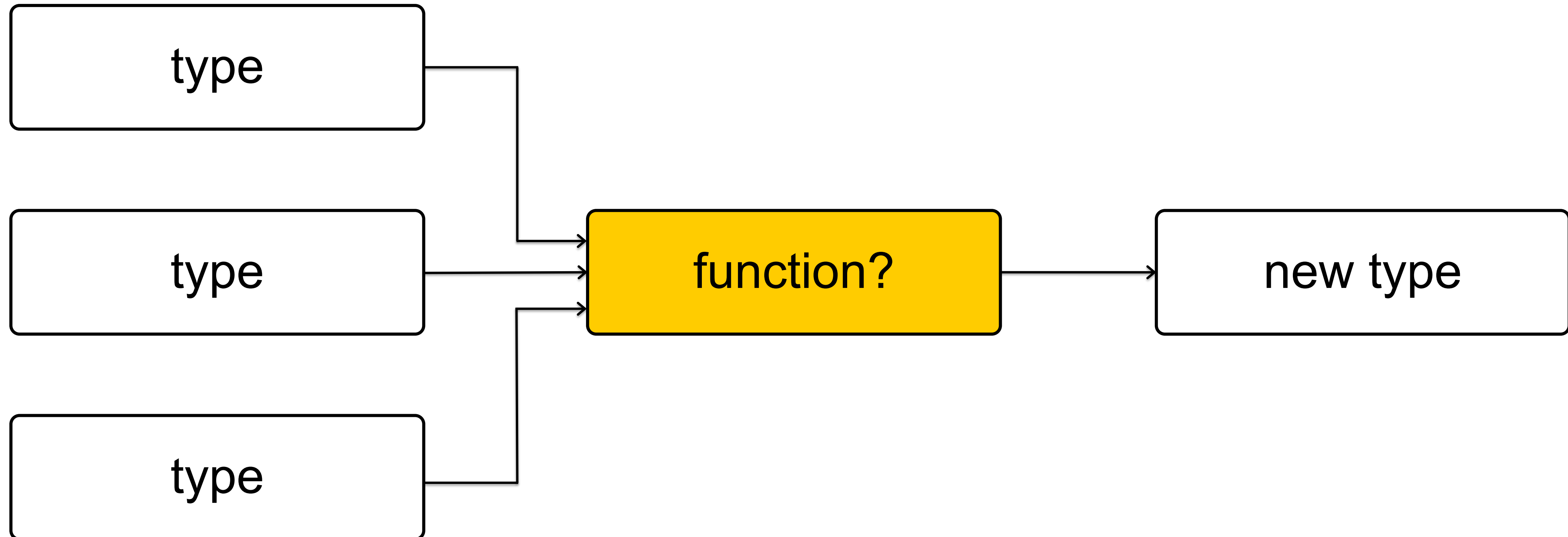
1. Основы: constexpr функции

```
constexpr int foo(int a, int b, int c) {  
    return a * b + c;  
}
```

```
constexpr int res = foo(1, 2, 3);
```

```
static_assert(res == 5);
```


1. Основы: функция, работающая с типами?



1. Основы: что было до constexpr?

- › Как создать функцию, работающую в compile time без constexpr?
- › Как создать функцию, работающую с типами?

1. Основы: метафункции

- › Нам помогут метафункции!

1. Основы: что такое метафункции в C++?

- › `struct` – функции в мире метапрограммирования
- › типы – значения в мире метапрограммирования

1. Основы: полезные аналогии

// Wonderful metaworld!

// Boring usual world...

1. Основы: полезные аналогии

```
// Wonderful metaworld!  
template <class A, class B>
```

```
// Boring usual world...  
int foo(int a, int b)
```

1. Основы: полезные аналогии

```
// Wonderful metaworld!  
template <class A, class B>  
struct foo {  
    using type = /*...*/  
};
```

```
// Boring usual world...  
int foo(int a, int b)  
{  
    return /*...*/  
}
```

1. Основы: полезные аналогии

```
// Wonderful metaworld!
template <class A, class B>
struct foo {
    using type = /*...*/
};

using type =
    typename foo<int, double>::type;
```

```
// Boring usual world...
int foo(int a, int b)
{
    return /*...*/
}

int value = foo(2, 8);
```


1. Основы: полезные аналогии

```
// Wonderful metaworld!
template <class A, class B>
struct foo {
    using type = /*...*/
};

using type =
    typename foo<int, double>::type;

using type2 =
    typename foo<type2, char>::type;
```

```
// Boring usual world...
int foo(int a, int b)
{
    return /*...*/
}

int value = foo(2, 8);

int value2 = foo(value, 5);
```

1. Основы: метафункции: замечания

- › Синтаксис вызова метафункции немного уродлив
- › К этому нужно привыкнуть
- › В отличие от обычных функций, метафункции могут «возвращать» множество значения

1. Основы: iterator_traits

Template parameters

Iterator - the iterator type to retrieve properties for

Member types

Member type	Definition
difference_type	Iterator::difference_type
value_type	Iterator::value_type
pointer	Iterator::pointer
reference	Iterator::reference
iterator_category	Iterator::iterator_category

If Iterator does not have all five member types difference_type, value_type, pointer, reference, and iterator_category, then this template has no members by any of those names (`std::iterator_traits` is SFINAE-friendly)

(since C++17)

1

1. Основы: где найти другие метафункции?

1. Основы: где найти другие метафункции?

Standard library header `<type_traits>`

This header is part of the [type support](#) library.

Classes

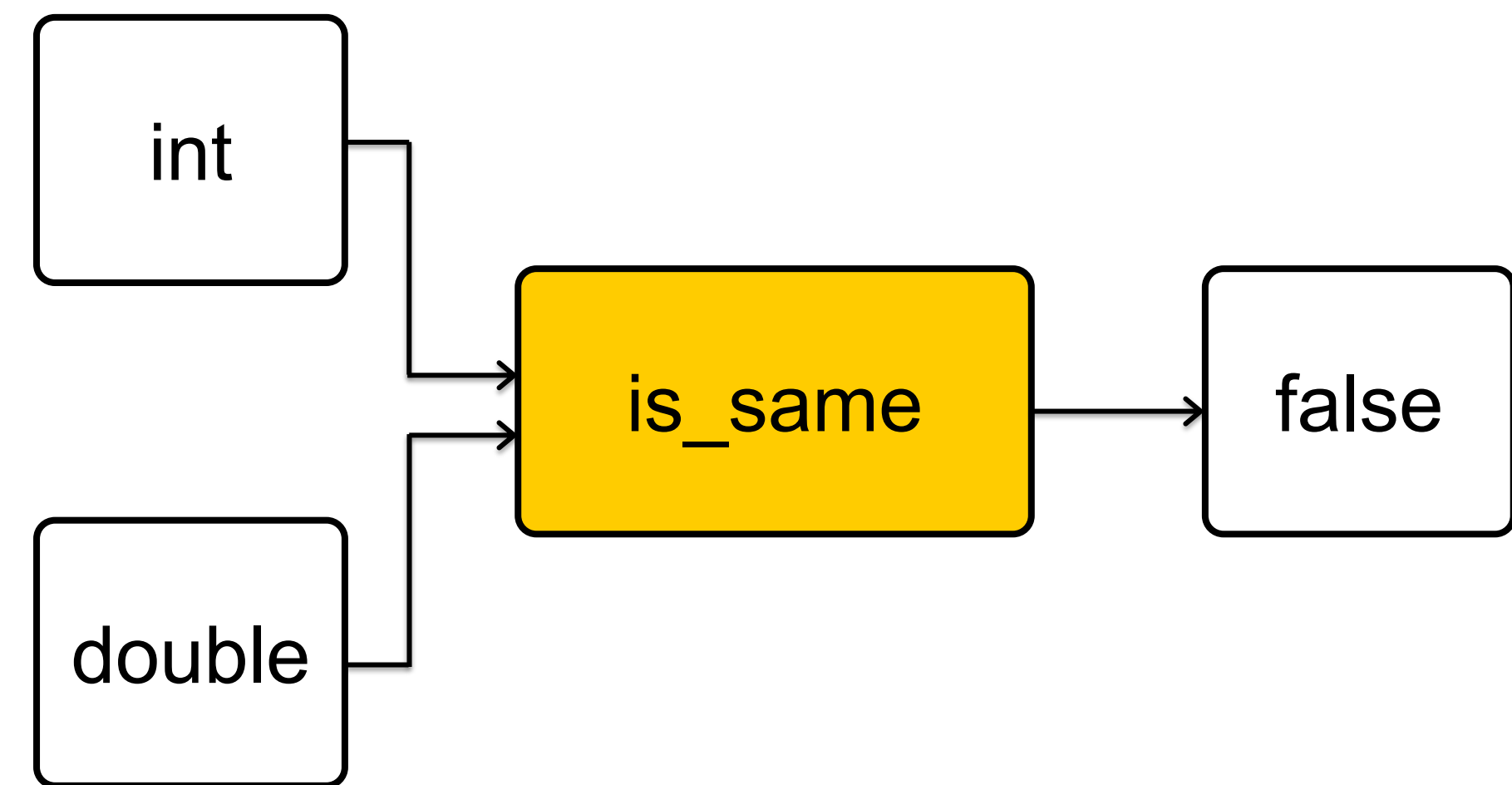
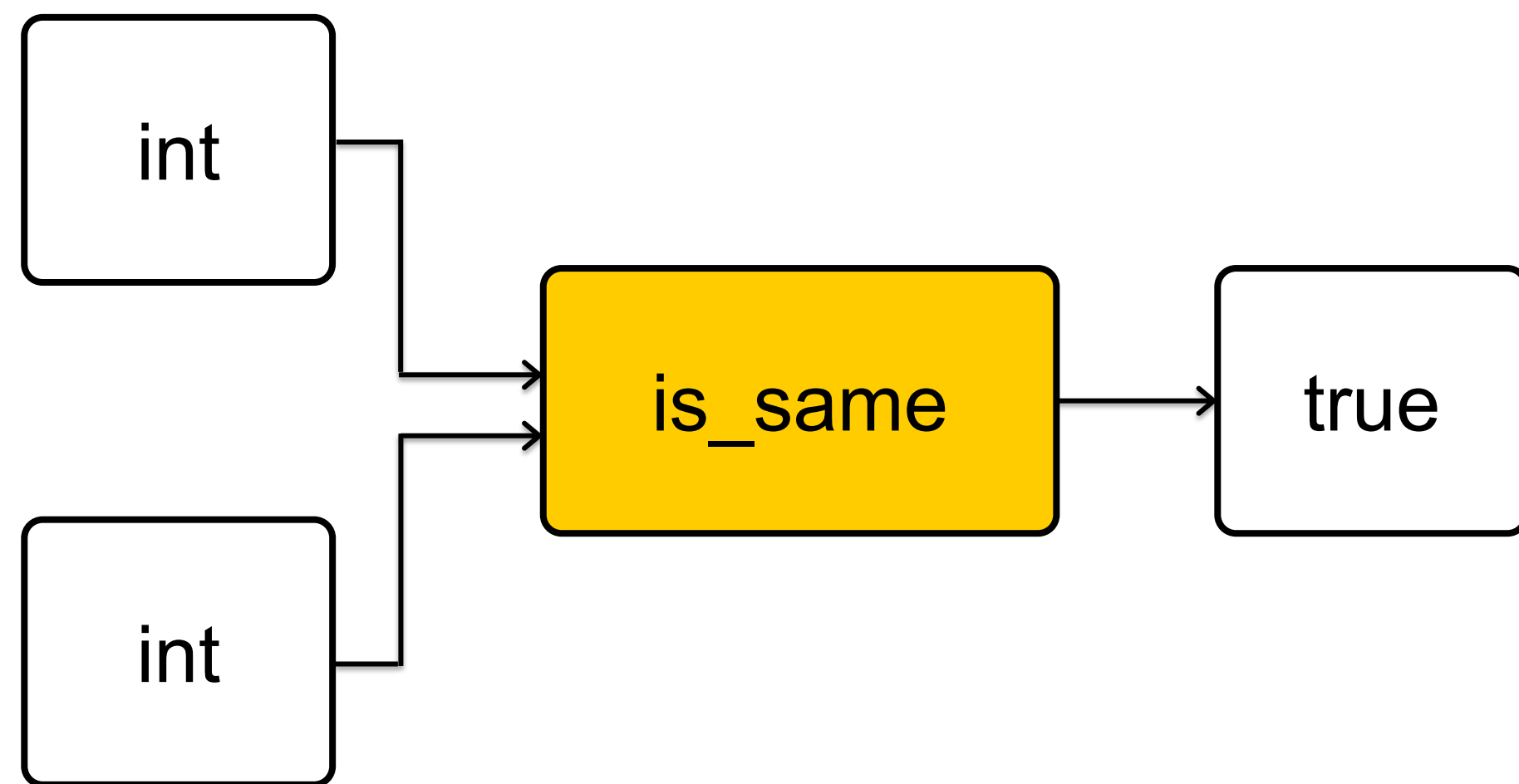
Helper Classes

<code>integral_constant</code> (C++11)	compile-time constant of specified type with specified value
<code>bool_constant</code> (C++17)	(class template)
<code>true_type</code>	<code>std::integral_constant<bool, true></code>
<code>false_type</code>	<code>std::integral_constant<bool, false></code>

Primary type categories

<code>is_void</code> (C++11)	checks if a type is <code>void</code> (class template)
<code>is_null_pointer</code> (C++14)	checks if a type is <code>std::nullptr_t</code> (class template)
<code>is_integral</code> (C++11)	checks if a type is integral type (class template)
<code>is_floating_point</code> (C++11)	checks if a type is floating-point type (class template)
<code>is_array</code> (C++11)	checks if a type is an array type (class template)
<code>is_enum</code> (C++11)	checks if a type is an enumeration type (class template)
<code>is_union</code> (C++11)	checks if a type is an union type (class template)

1. Основы: is_same



1. Основы: is_same

```
template <class A, class B>
struct is_same {
    static constexpr bool value = false;
};

template <class A>
struct is_same<A, A> { // template specialization
    static constexpr bool value = true;
}

static_assert(is_same<int, int>::value);
static_assert(!is_same<int, double>::value);
```

1. Основы: is_same

```
template <class A, class B>
struct is_same : std::false_type {};

template <class A>
struct is_same<A, A> : std::true_type {};

static_assert(is_same<int, int>::value);
static_assert(!is_same<int, double>::value);
```


Эволюция метапрограммирования: списки типов

2. Списки типов



2. Списки типов

- › Гетерогенное хранилище «объектов» ...
- › ... с которыми можно работать в compile time ...
- › ... при помощи value-based подхода
- › Поэтому `std::tuple` нам не подходит

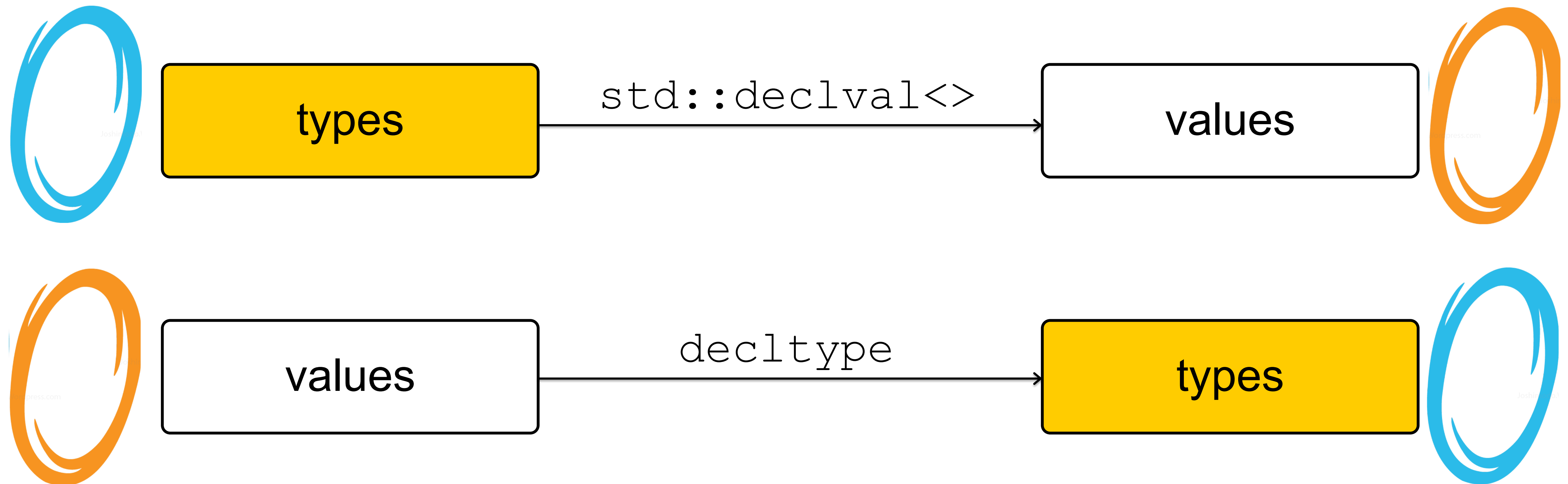
2. Списки типов: value-based?

- › По-минимуму создавать новые типы
- › Работать в основном с constexpr функциями
- › Комитет по стандартизации движется в сторону value-based метапрограммирования

2. Списки типов: почему value-based?

- › Созданные типы никогда не «деаллоцируются»
- › Как следствие, увеличивается размер программы
- › Создание новых типов серьезно замедляет компиляцию

2. Списки типов: $value \leftrightarrow type$ телепортации?



2. Списки типов: type holder

```
// Type holder aka std::type_identity from C++20 (but shorter)
template <class T>
struct just_type { using type = T; };
```

2. Списки типов: type holder

```
// Type holder aka std::type_identity from C++20 (but shorter)
template <class T>
struct just_type { using type = T; };

// Extracting type from type holder
constexpr auto type = just_type<int>{};
using type_t = typename decltype(type)::type;
static_assert(std::is_same_v<type_t, int>);
```

2. Списки типов: как **не** надо

2. СПИСКИ ТИПОВ: как **не** надо

```
// Never do that!  
template <class T, class... Ts>  
struct type_pack {  
    using head = T;  
    using tail = type_pack<Ts...>;  
    ...  
    // more useless code  
};
```

2. СПИСКИ ТИПОВ: как надо

```
// Just like that!  
template <class... Ts>  
struct type_pack {};
```

2. Списки типов: основные средства

2. СПИСКИ ТИПОВ: ОСНОВНЫЕ СРЕДСТВА

```
// Everything is under tp namespace  
using empty_pack = type_pack<>;
```

2. СПИСКИ ТИПОВ: ОСНОВНЫЕ СРЕДСТВА

```
// Everything is under tp namespace
using empty_pack = type_pack<>;

template <class... Ts>
constexpr size_t size(type_pack<Ts...>) {
    return sizeof...(Ts);
}
```

2. СПИСКИ ТИПОВ: ОСНОВНЫЕ СРЕДСТВА

```
// Everything is under tp namespace
using empty_pack = type_pack<>;

template <class... Ts>
constexpr size_t size(type_pack<Ts...>) {
    return sizeof...(Ts);
}

template <class... Ts>
constexpr bool empty(type_pack<Ts...> tp) {
    return size(tp) == 0;
}
```

2. СПИСКИ ТИПОВ: ОСНОВНЫЕ СРЕДСТВА

```
// Do we need it at all?
template <class T, class... Ts>
constexpr just_type<T> head(type_pack<T, Ts...>) {
    return {};
}

template <class T, class... Ts>
constexpr type_pack<Ts...> tail(type_pack<T, Ts...>) {
    return {};
}
```

2. СПИСКИ ТИПОВ: ОСНОВНЫЕ СРЕДСТВА

```
template <class... Ts, class... Us>  
constexpr bool operator==(type_pack<Ts...>, type_pack<Us...>) { return false; }
```


2. СПИСКИ ТИПОВ: ОСНОВНЫЕ СРЕДСТВА

```
template <class... Ts, class... Us>
constexpr bool operator==(type_pack<Ts...>, type_pack<Us...>) { return false; }

template <class... Ts>
constexpr bool operator==(type_pack<Ts...>, type_pack<Ts...>) { return true; }
```

2. СПИСКИ ТИПОВ: ОСНОВНЫЕ СРЕДСТВА

```
template <class... Ts, class... Us>
constexpr bool operator==(type_pack<Ts...>, type_pack<Us...>) { return false; }

template <class... Ts>
constexpr bool operator==(type_pack<Ts...>, type_pack<Ts...>) { return true; }

template <class... Ts, class... Us>
constexpr bool operator!=(type_pack<Ts...>, type_pack<Us...>) { return true; }

template <class... Ts>
constexpr bool operator!=(type_pack<Ts...>, type_pack<Ts...>) { return false; }
```

2. СПИСКИ ТИПОВ: ОСНОВНЫЕ СРЕДСТВА

```
template <class T, class U>
constexpr bool operator==(just_type<T>, just_type<U>) { return false; }
template <class T>
constexpr bool operator==(just_type<T>, just_type<T>) { return true; }
```

```
template <class T, class U>
constexpr bool operator!=(just_type<T>, just_type<U>) { return true; }
template <class T>
constexpr bool operator!=(just_type<T>, just_type<T>) { return false; }
```

2. Списки типов: push и pop

2. Списки типов: push и pop

```
// type-based  
template <class T, class... Ts>  
constexpr type_pack<T, Ts...> push_front(type_pack<Ts...>) { return {}; }
```

2. Списки типов: push и pop

```
// type-based
template <class T, class... Ts>
constexpr type_pack<T, Ts...> push_front(type_pack<Ts...>) { return {}; }

// value-based
template <class... Ts, class T>
constexpr type_pack<T, Ts...> push_front(type_pack<Ts...>, just_type<T>) {
    return {};
}
```

2. Списки типов: push и pop

```
// type-based
template <class T, class... Ts>
constexpr type_pack<T, Ts...> push_front(type_pack<Ts...>) { return {}; }

// value-based
template <class... Ts, class T>
constexpr type_pack<T, Ts...> push_front(type_pack<Ts...>, just_type<T>) {
    return {};
}

static_assert(push_front<int>(type_pack<double, char>{}) ==
               type_pack<int, double, char>{});
```

2. Списки типов: push и pop

```
template <class T, class... Ts>
constexpr type_pack<Ts...> pop_front(type_pack<T, Ts...>) {
    return {};
}

static_assert(pop_front(type_pack<int, double, char>{}) ==
               type_pack<double, char>{});
```


2. Списки типов: push и pop

```
// type-based
template <class T, class... Ts>
constexpr type_pack<Ts..., T> push_back(type_pack<Ts...>) { return {}; }

// value-based
template <class... Ts, class T>
constexpr type_pack<Ts..., T> push_back(type_pack<Ts...>, just_type<T>) {
    return {};
}

static_assert(push_back<int>(type_pack<double, char>{}) ==
               type_pack<double, char, int>{});
```

2. Списки типов: push и pop

- › pop_back?...
- › ...так просто не получится

2. Списки типов: `contains`

- › Узнать, присутствует ли тип `T` в данном списке

2. СПИСКИ ТИПОВ: contains (bad)

```
template <class U, class... Ts>  
struct contains_impl;
```

2. СПИСКИ ТИПОВ: contains (bad)

```
template <class U, class... Ts>
struct contains_impl;

template <class U, class... Ts> // found case
struct contains_impl<U, U, Ts...> : std::true_type {};
```

2. СПИСКИ ТИПОВ: contains (bad)

```
template <class U, class... Ts>
struct contains_impl;

template <class U, class... Ts> // found case
struct contains_impl<U, U, Ts...> : std::true_type {};

template <class U, class T, class... Ts> // continue case
struct contains_impl<U, T, Ts...> : contains_impl<U, Ts...> {};
```

2. СПИСКИ ТИПОВ: contains (bad)

```
template <class U, class... Ts>
struct contains_impl;

template <class U, class... Ts> // found case
struct contains_impl<U, U, Ts...> : std::true_type {};

template <class U, class T, class... Ts> // continue case
struct contains_impl<U, T, Ts...> : contains_impl<U, Ts...> {};

template <class U> // not found case
struct contains_impl<U> : std::false_type {};
```

2. СПИСКИ ТИПОВ: contains (bad)

```
// type-based
template <class T, class... Ts>
constexpr bool contains(type_pack<Ts...>) {
    return contains_impl<T, Ts...>::value;
}

// value-based
template <class... Ts, class T>
constexpr bool contains(type_pack<Ts...>, just_type<T>) {
    return contains_impl<T, Ts...>::value;
}

static_assert(contains<int>(type_pack<int, double, char>{}));
```


2. Списки типов: contains (bad)

- › Рекурсивная инстанциация типов
- › Много бойлерплейтного кода

2. СПИСКИ ТИПОВ: contains (good)

```
template <class T, class... Ts>  
constexpr bool contains(type_pack<Ts...>) {
```

2. СПИСКИ ТИПОВ: contains (good)

```
template <class T, class... Ts>
constexpr bool contains(type_pack<Ts...>) {
    bool bs[] = {std::is_same<T, Ts>::value...};
```

2. СПИСКИ ТИПОВ: contains (good)

```
template <class T, class... Ts>
constexpr bool contains(type_pack<Ts...>) {
    bool bs[] = {std::is_same<T, Ts>::value...};
    bool res = false;
    for (bool b : bs) {
        res |= b;
    }
}
```

2. СПИСКИ ТИПОВ: contains (good)

```
template <class T, class... Ts>
constexpr bool contains(type_pack<Ts...>) {
    bool bs[] = {std::is_same<T, Ts>::value...};
    bool res = false;
    for (bool b : bs) {
        res |= b;
    }
    return res;
}

static_assert(contains<int>(type_pack<int, double, char>{}));
```

2. Списки типов: contains (good)

- › Никакой рекурсии
- › Код уместается на один слайд
- › Кода всё ещё много...

2. СПИСКИ ТИПОВ: contains (best)

```
template <class T, class... Ts>
constexpr bool contains(type_pack<Ts...>) {
    // C++17 fold expressions
    return (... || std::is_same_v<T, Ts>);
}
```

```
static_assert(contains<int>(type_pack<int, double, char>{}));
static_assert(!contains<int*>(type_pack<int, double, char>{}));
static_assert(!contains<int>(empty_pack{}));
```

2. Списки типов: find

- › Взять список и тип T
- › Результат – позиция первого вхождения T в список
- › Если T в списке отсутствует, результат – размер списка

2. СПИСКИ ТИПОВ: find (bad)

```
template <class U, class... Ts>  
struct find_impl;
```

2. СПИСКИ ТИПОВ: find (bad)

```
template <class U, class... Ts>
struct find_impl;

template <class U, class... Ts> // found case
struct find_impl<U, U, Ts...> : std::integral_constant<size_t, 0> {};
```

2. СПИСКИ ТИПОВ: find (bad)

```
template <class U, class... Ts>
struct find_impl;

template <class U, class... Ts> // found case
struct find_impl<U, U, Ts...> : std::integral_constant<size_t, 0> {};

template <class U, class T, class... Ts> // continue case
struct find_impl<U, T, Ts...>
    : std::integral_constant<size_t, find_impl<U, Ts...>::value + 1> {};
```

2. СПИСКИ ТИПОВ: find (bad)

```
template <class U, class... Ts>
struct find_impl;

template <class U, class... Ts> // found case
struct find_impl<U, U, Ts...> : std::integral_constant<size_t, 0> {};

template <class U, class T, class... Ts> // continue case
struct find_impl<U, T, Ts...>
    : std::integral_constant<size_t, find_impl<U, Ts...>::value + 1> {};

template <class U> // not found case (value = size(tp))
struct find_impl<U> : std::integral_constant<size_t, 0> {};
```

2. СПИСКИ ТИПОВ: find (bad)

```
// type-based
template <class U, class... Ts>
constexpr size_t find(type_pack<Ts...>) {
    return find_impl<U, Ts...>::value;
}

// value-based
template <class... Ts, class U>
constexpr size_t find(type_pack<Ts...> tp, just_type<U>) {
    return find<U>(tp);
}

static_assert(find<int>(type_pack<double, int, char>{}) == 1);
```

2. Списки типов: find (bad)

- › Рекурсия...
- › Много кода...

2. СПИСКИ ТИПОВ: find (good)

```
template <class T, class... Ts>  
constexpr size_t find(type_pack<Ts...> tp) {
```

2. СПИСКИ ТИПОВ: find (good)

```
template <class T, class... Ts>
constexpr size_t find(type_pack<Ts...> tp) {
    bool bs[] = {std::is_same_v<T, Ts>...};
    for (size_t i = 0; i < size(tp); ++i) {
        if (bs[i]) {
            return i;
        }
    }
}
```


2. СПИСКИ ТИПОВ: find (good)

```
template <class T, class... Ts>
constexpr size_t find(type_pack<Ts...> tp) {
    bool bs[] = {std::is_same_v<T, Ts>...};
    for (size_t i = 0; i < size(tp); ++i) {
        if (bs[i]) {
            return i;
        }
    }
    return size(tp);
}

static_assert(find<int>(type_pack<double, int, char>{}) == 1);
```

2. Списки типов: find (good)

- › Рекурсии нет
- › Кода мало...
- › ...но недостаточно мало...

2. СПИСКИ ТИПОВ: find (best) (C++20)

```
template <class T, class... Ts>
constexpr size_t find(type_pack<Ts...> tp) {
    bool bs[] = {std::is_same_v<T, Ts>...};
    // std::find will be constexpr in C++20
    return std::find(bs, bs + size(tp), true) - bs;
}

static_assert(find<int>(type_pack<double, int, char>{}) == 1);
static_assert(find<int*>(type_pack<double, int, char>{}) == 3);
```

2. СПИСКИ ТИПОВ: find (even better) (C++20)

```
template <class T, class... Ts>
constexpr size_t find(type_pack<Ts...> tp) {
    bool bs[] = {std::is_same_v<T, Ts>...};
    // std::find will be constexpr in C++20
    return std::find(bs, true) - bs;
}

static_assert(find<int>(type_pack<double, int, char>{}) == 1);
static_assert(find<int*>(type_pack<double, int, char>{}) == 3);
```

2. Списки типов: `find_if`

- › Взять список и предикат (метафункцию)
- › Результат – позиция первого вхождения в список элемента, удовлетворяющего предикату
- › Если элемент, удовлетворяющий предикату, в списке отсутствует, результат – размер списка

2. Списки типов: как принимать метафункции?

- › Как использовать метафункцию в качестве аргумента?

2. Списки типов: шаблонные шаблоны

```
// Haskell's partial call analogue
template <template <class...> class F, class... Ts>
class part_caller {
    template <class... Us>
    using type = typename F<Ts..., Us...>::type;
};
```

2. Списки типов: шаблонные шаблоны

```
// Haskell's partial call analogue
template <template <class...> class F, class... Ts>
class part_caller {
    template <class... Us>
    using type = typename F<Ts..., Us...>::type;
};

// convenient alias
template <template <class...> class F, class... Ts>
using part_caller_t = typename part_caller<F, Ts...>::type;
```


2. СПИСКИ ТИПОВ: find_if

```
template <template <class...> class F, class... Ts>
constexpr size_t find_if(type_pack<Ts...> tp) {
    bool bs[] = {F<Ts>::value...};
    return std::find(bs, true) - bs;
}

static_assert(find_if<std::is_pointer>(type_pack<double, int*, char>{}) == 1);
```

2. СПИСКИ ТИПОВ: find_if

```
template <template <class...> class F, class... Ts>
constexpr size_t find_if(type_pack<Ts...> tp) {
    bool bs[] = {F<Ts>::value...};
    return std::find(bs, true) - bs;
}

static_assert(find_if<std::is_pointer>(type_pack<double, int*, char>{}) == 1);

static_assert(find_if<part_caller_t<std::is_base_of, std::exception>>(
    type_pack<std::runtime_error, char, std::string>{}) ==
0);
```

2. Списки типов: `any_of`, `all_of`, `none_of`

2. Списки типов: any_of, all_of, none_of

```
template <template <class...> class F, class... Ts>
constexpr bool all_of(type_pack<Ts...>) {
    return (... && F<Ts>::value);
}
```

2. Списки типов: any_of, all_of, none_of

```
template <template <class...> class F, class... Ts>
constexpr bool all_of(type_pack<Ts...>) {
    return (... && F<Ts>::value);
}

template <template <class...> class F, class... Ts>
constexpr bool any_of(type_pack<Ts...>) {
    return (... || F<Ts>::value);
}
```

2. Списки типов: any_of, all_of, none_of

```
template <template <class...> class F, class... Ts>
constexpr bool all_of(type_pack<Ts...>) {
    return (... && F<Ts>::value);
}

template <template <class...> class F, class... Ts>
constexpr bool any_of(type_pack<Ts...>) {
    return (... || F<Ts>::value);
}

template <template <class...> class F, class... Ts>
constexpr bool none_of(type_pack<Ts...> tp) {
    return !any_of<F>(tp);
}
```

2. Списки типов: any_of, all_of, none_of

```
static_assert(all_of<std::is_pointer>(type_pack<int*, double*, char*>{}));
```

```
static_assert(all_of<std::is_pointer>(empty_pack{}));
```

```
static_assert(any_of<std::is_reference>(type_pack<int&, double, char**>{}));
```

```
static_assert(!any_of<std::is_reference>(empty_pack{}));
```

```
static_assert(none_of<std::is_void>(type_pack<int, double, char>{}));
```

```
static_assert(none_of<std::is_void>(empty_pack{}));
```

2. Списки типов: transform

- › Взять список
- › Применить метафункцию к каждому элементу списка
- › Результаты сложить в новый список

2. СПИСКИ ТИПОВ: transform

```
template <template <class...> class F, class... Ts>
constexpr auto transform(type_pack<Ts...>) {
    return type_pack<typename F<Ts>::type...>{};
}

static_assert(transform<std::add_pointer>(type_pack<int, double, char>{}) ==
    type_pack<int*, double*, char*>{});
```

2. Списки типов: `reverse`

2. СПИСКИ ТИПОВ: reverse

```
template <class... Ts> // finish case  
constexpr auto reverse_impl(empty_pack, type_pack<Ts...> res) { return res };
```

2. СПИСКИ ТИПОВ: reverse

```
template <class... Ts> // finish case
constexpr auto reverse_impl(empty_pack, type_pack<Ts...> res) { return res };

template <class T, class... Ts, class... Us> // continue case
constexpr auto reverse_impl(type_pack<T, Ts...>, type_pack<Us...>) {
    return reverse_impl(type_pack<Ts...>{}, type_pack<T, Us...>{});
}
```

2. СПИСКИ ТИПОВ: reverse

```
template <class... Ts> // finish case
constexpr auto reverse_impl(empty_pack, type_pack<Ts...> res) { return res };

template <class T, class... Ts, class... Us> // continue case
constexpr auto reverse_impl(type_pack<T, Ts...>, type_pack<Us...>) {
    return reverse_impl(type_pack<Ts...>{}, type_pack<T, Us...>{});
}

template <class... Ts>
constexpr auto reverse(type_pack<Ts...> tp) { return reverse_impl(tp, {}); }

static_assert(reverse(type_pack<int, double, char>{}) ==
               type_pack<char, double, int>{});
```

2. Списки типов: get

- › Взять список и некоторый индекс
- › Вернуть тип, находящийся в списке по данному индексу

2. Списки типов: get (bad)

```
template <size_t I, class... Ts>  
struct get_impl;
```

2. Списки типов: get (bad)

```
template <size_t I, class... Ts>
```

```
struct get_impl;
```

```
template <class T, class... Ts> // found case
```

```
struct get_impl<0, T, Ts...> { using type = T; };
```


2. Списки типов: get (bad)

```
template <size_t I, class... Ts>
struct get_impl;

template <class T, class... Ts> // found case
struct get_impl<0, T, Ts...> { using type = T; };

template <size_t I, class T, class... Ts> // continue case
struct get_impl<I, T, Ts...> {
    using type = typename get_impl<I - 1, Ts...>::type;
};
```

2. СПИСКИ ТИПОВ: get (bad)

```
template <size_t I, class... Ts>
struct get_impl;

template <class T, class... Ts> // found case
struct get_impl<0, T, Ts...> { using type = T; };

template <size_t I, class T, class... Ts> // continue case
struct get_impl<I, T, Ts...> {
    using type = typename get_impl<I - 1, Ts...>::type;
};

template <size_t I> // not found case, SFINAE-friendly
struct get_impl<I> {};
```

2. Списки типов: get (bad)

```
template <size_t I, class... Ts>
constexpr auto get(type_pack<Ts...>) {
    return just_type<typename get_impl<I, Ts...>::type>{};
}

static_assert(get<1>(type_pack<double, int, char>{}) == just_type<int>{});
```

2. Списки типов: get (bad)

```
template <size_t I, class... Ts>
constexpr auto get(type_pack<Ts...>) {
    return just_type<typename get_impl<I, Ts...>::type>{};
}

static_assert(get<1>(type_pack<double, int, char>{}) == just_type<int>{});

// compilation error: no type named 'type' in 'tp::get_impl<2>'
static_assert(get<5>(type_pack<double, int, char>{}) == just_type<int>{});
```

2. Списки типов: get (bad)

- › Всем понятно, почему это плохо...

2. Списки типов: get (good)

- › Нам нужно как-то быстро создавать списки индексов
- › В стандарте есть готовые примитивы!

2. СПИСКИ ТИПОВ: `std::index_sequence`

```
template<std::size_t... Ints>
using index_sequence = std::integer_sequence<std::size_t, Ints...>;
```

A helper alias template **`std::make_integer_sequence`** is defined to simplify creation of `std::integer_sequence` and `std::index_sequence` types with 0, 1, 2, ..., N-1 as Ints:

```
template<class T, T N>
using make_integer_sequence = std::integer_sequence<T, /* a sequence 0, 1, 2, ..., N-1 */ >;

template<std::size_t N>
using make_index_sequence = make_integer_sequence<std::size_t, N>;
```

The program is ill-formed if N is negative. If N is zero, the indicated type is `integer_sequence<T>`.

A helper alias template **`std::index_sequence_for`** is defined to convert any type parameter pack into an index sequence of the same length

```
template<class... T>
using index_sequence_for = std::make_index_sequence<sizeof...(T)>;
```

2. Списки типов: get (good)

```
template <size_t I, class T>
struct indexed_type {
    static constexpr size_t value = I;
    using type = T;
};
```


2. СПИСКИ ТИПОВ: get (good)

```
template <size_t I, class T>
struct indexed_type {
    static constexpr size_t value = I;
    using type = T;
};

template <class IS, class... Ts>
struct indexed_types;

template <size_t... Is, class... Ts>
struct indexed_types<std::index_sequence<Is...>, Ts...> {
    struct type : indexed_type<Is, Ts>... {};
};
```

2. Списки типов: get (good)

```
template <class... Ts>
using indexed_types_for =
    typename indexed_types<std::index_sequence_for<Ts...>, Ts...>::type;
```

2. Списки типов: get (good)

```
template <class... Ts>
using indexed_types_for =
    typename indexed_types<std::index_sequence_for<Ts...>, Ts...>::type;

template <size_t I, class T>
constexpr just_type<T> get_indexed_type(indexed_type<I, T>) { return {}; }
```

2. СПИСКИ ТИПОВ: get (good)

```
template <class... Ts>
using indexed_types_for =
    typename indexed_types<std::index_sequence_for<Ts...>, Ts...>::type;

template <size_t I, class T>
constexpr just_type<T> get_indexed_type(indexed_type<I, T>) { return {}; }

template <size_t I, class... Ts>
constexpr auto get(type_pack<Ts...>) {
    return get_indexed_type<I>(indexed_types_for<Ts...>{});
}

static_assert(get<1>(type_pack<double, int, char>{}) == just_type<int>{});
```

2. Списки типов: get (good)

- › Рекурсии нет
- › Количество дополнительных создаваемых типов по прежнему линейно по отношению к размеру списка

2. Списки типов: comma operator

- › Последовательно выполняет операции через запятую
- › Результат такой последовательности – последняя выполненная операция (операция, находящаяся за самой последней запятой)

2. СПИСКИ ТИПОВ: comma operator

Run this code

```
#include <iostream>
int main()
{
    int n = 1;
    int m = (++n, std::cout << "n = " << n << '\n', ++n, 2*n);
    std::cout << "m = " << (++m, m) << '\n';
}
```

Output:

```
n = 2
m = 7
```

cppreference.com
example

2. Списки типов: get (best)

```
template <class IS>  
struct get_impl;
```


2. Списки типов: get (best)

```
template <class IS>
```

```
struct get_impl;
```

```
template <size_t... Is>
```

```
struct get_impl<std::index_sequence<Is...>> {
```

2. СПИСКИ ТИПОВ: get (best)

```
template <class IS>
struct get_impl;

template <size_t... Is>
struct get_impl<std::index_sequence<Is...>> {
    template <class T>
    static constexpr T dummy(decltype(Is, (void*)0)..., T*, ...);
};
```

2. СПИСКИ ТИПОВ: get (best)

```
// get_impl<std::make_index_sequence<3>>:
struct get_impl<std::index_sequence<0, 1, 2>> {
    template <class T>
    static constexpr T dummy(decltype(0, (void*)0),
                              decltype(1, (void*)0),
                              decltype(2, (void*)0), T*, ...);
};

// =>
struct get_impl<std::index_sequence<0, 1, 2>> {
    template <class T>
    static constexpr T dummy(void*, void*, void*, T*, ...);
};
```

2. Списки типов: get (best)

```
template <size_t I, class... Ts>
constexpr auto get(type_pack<Ts...>) {
    return just_type<decltype(
        get_impl<std::make_index_sequence<I>>::dummy((Ts*)(0)...))>{};
}

static_assert(get<1>(type_pack<double, int, char>{}) == just_type<int>{});
```

2. Списки типов: get (best)

- › Количество создаваемых типов константно
- › Жуткий код
- › Плохо работает с MSVC...

2. Списки типов: generate

- › Взять тип T и размер N
- › Создать список, содержащий N элементов T

2. Списки типов: generate (bad)

```
template <size_t I, class T, class... Ts> // continue case
struct generate_impl {
    using type = typename generate_impl<I - 1, T, Ts..., T>::type;
};
```

2. Списки типов: generate (bad)

```
template <size_t I, class T, class... Ts> // continue case
struct generate_impl {
    using type = typename generate_impl<I - 1, T, Ts..., T>::type;
};

template <class T, class... Ts> // finish case
struct generate_impl<0, T, Ts...> {
    using type = type_pack<Ts...>;
};
```


2. СПИСКИ ТИПОВ: generate (bad)

```
template <size_t I, class T, class... Ts> // continue case
struct generate_impl {
    using type = typename generate_impl<I - 1, T, Ts..., T>::type;
};

template <class T, class... Ts> // finish case
struct generate_impl<0, T, Ts...> {
    using type = type_pack<Ts...>;
};

template <size_t I, class T>
constexpr auto generate() { return typename generate_impl<I, T>::type{}; }

static_assert(generate<3, int>() == type_pack<int, int, int>{});
```

2. Списки типов: generate (bad)

- › Рекурсия, которая не даст нам создать большой список

2. Списки типов: generate (best)

```
template <class... Ts>  
constexpr type_pack<Ts...> generate_helper(Ts*...) { return {};} }
```

2. Списки типов: generate (best)

```
template <class... Ts>
constexpr type_pack<Ts...> generate_helper(Ts*...) { return {}; }

template <class T, size_t... Is>
constexpr auto generate_impl(std::index_sequence<Is...>) {
    return generate_helper(((void)Is, (T*)0)...);
}
```

2. Списки типов: generate (best)

```
template <class... Ts>
constexpr type_pack<Ts...> generate_helper(Ts*...) { return {}; }

template <class T, size_t... Is>
constexpr auto generate_impl(std::index_sequence<Is...>) {
    return generate_helper(((void)Is, (T*)0)...);
}

template <size_t I, class T>
constexpr auto generate() {
    return generate_impl<T>(std::make_index_sequence<I>{});
}

static_assert(generate<3, int>() == type_pack<int, int, int>{});
```

2. Списки типов: generate (best)

```
static_assert(generate<32768, int>() != type_pack<int, int, int>{});
```

```
clang++ ...
```

```
...
```

```
#63 0x00007f932f6c1223 __libc_start_main (/usr/bin/../lib/libc.so.6+0x24223)
```

```
#64 0x00005557b38a45fe _start (/usr/bin/clang-7+0xf5fe)
```

```
clang-7: error: unable to execute command: Segmentation fault (core dumped)
```

```
clang-7: error: clang frontend command failed due to signal (use -v to see invocation)
```

```
clang version 7.0.1 (tags/RELEASE_701/final)
```

```
Target: x86_64-pc-linux-gnu
```

```
Thread model: posix
```

2. Списки типов: filter

- › Взять список и предикат (метафункцию)
- › Вернуть список только с теми элементами, которые удовлетворили предикату

2. Списки типов: filter (bad)

```
template <template <class...> class F, class... Us>  
constexpr auto filter_impl(empty_pack, type_pack<Us...> res) { return res; }
```


2. Списки типов: filter (bad)

```
template <template <class...> class F, class... Us>
constexpr auto filter_impl(empty_pack, type_pack<Us...> res) { return res; }

template <template <class...> class F, class T, class... Ts, class... Us>
constexpr auto filter_impl(type_pack<T, Ts...>, type_pack<Us...>) {
    if constexpr (F<T>::value) {
        return filter_impl<F>(type_pack<Ts...>{}, type_pack<Us..., T>{});
    }
}
```

2. СПИСКИ ТИПОВ: filter (bad)

```
template <template <class...> class F, class... Us>
constexpr auto filter_impl(empty_pack, type_pack<Us...> res) { return res; }

template <template <class...> class F, class T, class... Ts, class... Us>
constexpr auto filter_impl(type_pack<T, Ts...>, type_pack<Us...>) {
    if constexpr (F<T>::value) {
        return filter_impl<F>(type_pack<Ts...>{}, type_pack<Us..., T>{});
    } else {
        return filter_impl<F>(type_pack<Ts...>{}, type_pack<Us...>{});
    }
}
```

2. Списки типов: filter (bad)

```
template <template <class...> class F, class... Ts>
constexpr auto filter(type_pack<Ts...> tp) {
    return filter_impl<F>(tp, {});
}
```

```
static_assert(filter<std::is_pointer>(type_pack<char, double*, int*>{}) ==
              type_pack<double*, int*>{});
```

```
static_assert(filter<std::is_pointer>(empty_pack{}) == empty_pack{});
```

2. Списки типов: filter (**bad**)

- › Рекурсия
- › В остальном, способ скорее **good**, чем **bad**

2. СПИСКИ ТИПОВ: filter (best)

```
template <template <class...> class F, class T>
constexpr auto filter_one() {
    if constexpr (F<T>::value) {
        return type_pack<T>{};
    } else {
        return empty_pack{};
    }
}
```

2. СПИСКИ ТИПОВ: filter (best)

```
template <template <class...> class F, class T>
constexpr auto filter_one() {
    if constexpr (F<T>::value) {
        return type_pack<T>{};
    } else {
        return empty_pack{};
    }
}

template <class... Ts, class... Us>
constexpr auto operator+(type_pack<Ts...>, type_pack<Us...>) {
    return type_pack<Ts..., Us...>{};
}
```

2. СПИСКИ ТИПОВ: filter (best)

```
template <template <class...> class F, class... Ts>
constexpr auto filter(type_pack<Ts...>) {
    return (empty_pack{} + ... + filter_one<F, Ts>());
}
```

```
static_assert(filter<std::is_pointer>(type_pack<char, double*, int*>{}) ==
              type_pack<double*, int*>{});
```

2. СПИСКИ ТИПОВ: ВСЁ ВМЕСТЕ

```
constexpr type_pack<int, double> tp1;
constexpr auto tp2 = tp::push_back<char*>(tp1);
constexpr auto tp3 = tp::push_front(tp2, just_type<std::string>{});
static_assert(tp3 == type_pack<std::string, int, double, char*>{});

constexpr auto tp4 = tp::filter<std::is_scalar>(tp3);
static_assert(tp4 == type_pack<int, double, char*>{});
static_assert(tp::find_if<std::is_pointer>(tp4) == 2);

constexpr auto tp5 = tp::transform<std::add_lvalue_reference>(tp4) +
                    tp::generate<2, short**>();
static_assert(tp5 == type_pack<int&, double&, char*&, short**, short**>{});
```


Эволюция метапрограммирования: списки типов

3. Итог



3. Итог: списки типов

- › Со списками типов можно работать (если постараться)
- › Готовые средства для работы с ними удобны...
- › ... но скорее всего никогда не появятся в стандарте
- › Перемешивание type- и value-based подходов является проблемой

3. Итог: метапрограммирование

- Метапрограммирование значительно расширяет границы возможного
- Метапрограммирование не является чем-то дружелюбным, пока к нему не привыкнешь...
- ... как и весь C++
- Будущее метапрограммирования туманно...

Спасибо за внимание!

Олег Фатхиев

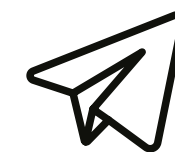
Младший разработчик



tender-bum@yandex-team.ru



+7 919 155-83-40



tender-bum



github.com/ofats/meta_evo