Яндекс

Яндекс

Эволюция метапрограммирования. Как правильно работать со списками типов

Олег Фатхиев

~~Ranges~~

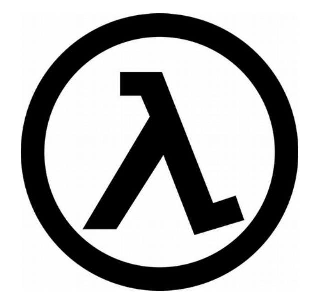
1. Основы

Зачем нам метапрограммирование?

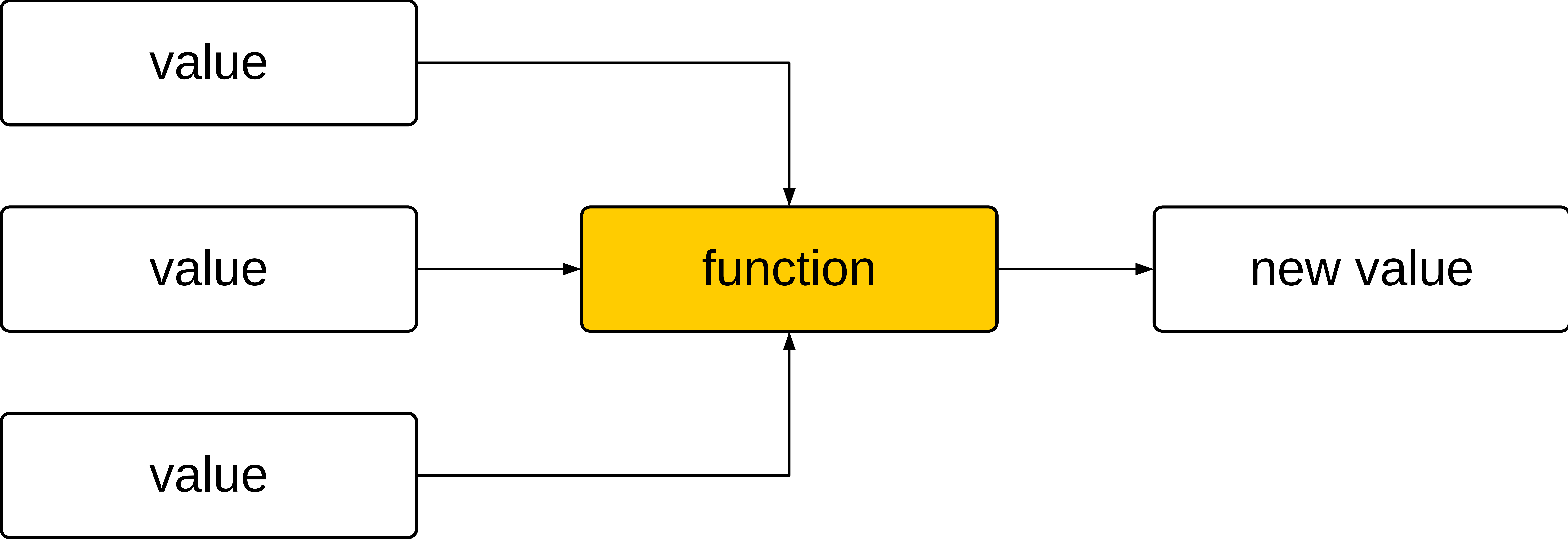
- Ускорение runtime-а с помощью переноса вычислений в compile time
- Кодогенерация
- Condition checking

Что такое метапрограммирование?

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



Compile time вычисления



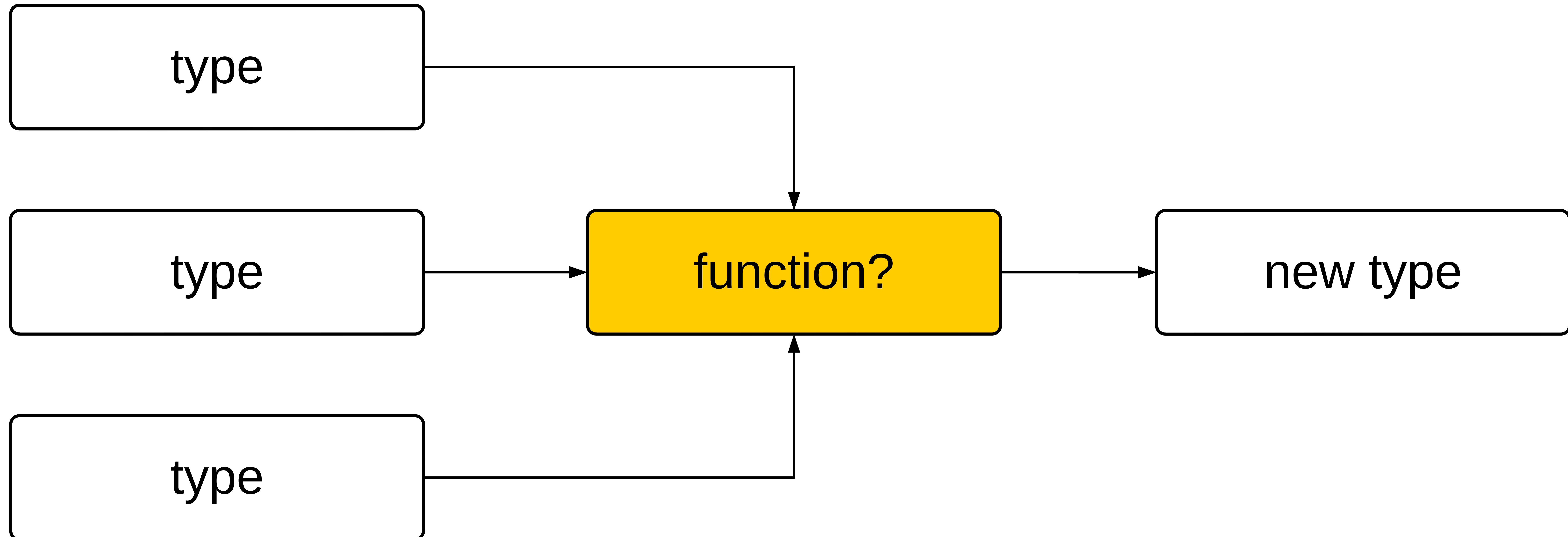
Constexpr функции

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

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

static_assert(res == 5);
```

Функция, работающая с типами?



Что было до constexpr?

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

Что такое метафункции в C++?

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

Полезные аналогии в мета-мире

```
// Wonderful metaworld!
```

```
// Boring usual world...
```

Полезные аналогии в мета-мире

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

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

Полезные аналогии в мета-мире

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

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

Полезные аналогии в мета-мире

```
// 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);
```

Полезные аналогии в мета-мире

```
// 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);
```

Метафункции: замечания

- Синтаксис вызова метафукнции немного уродлив
- К этому нужно привыкнуть
- В отличие от обычных функций, метафункции могут “возвращать” множество значения

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)

Где найти другие метафункции?

Где найти другие метафункции?

Standard library header <type_traits>

This header is part of the type support library.

Classes

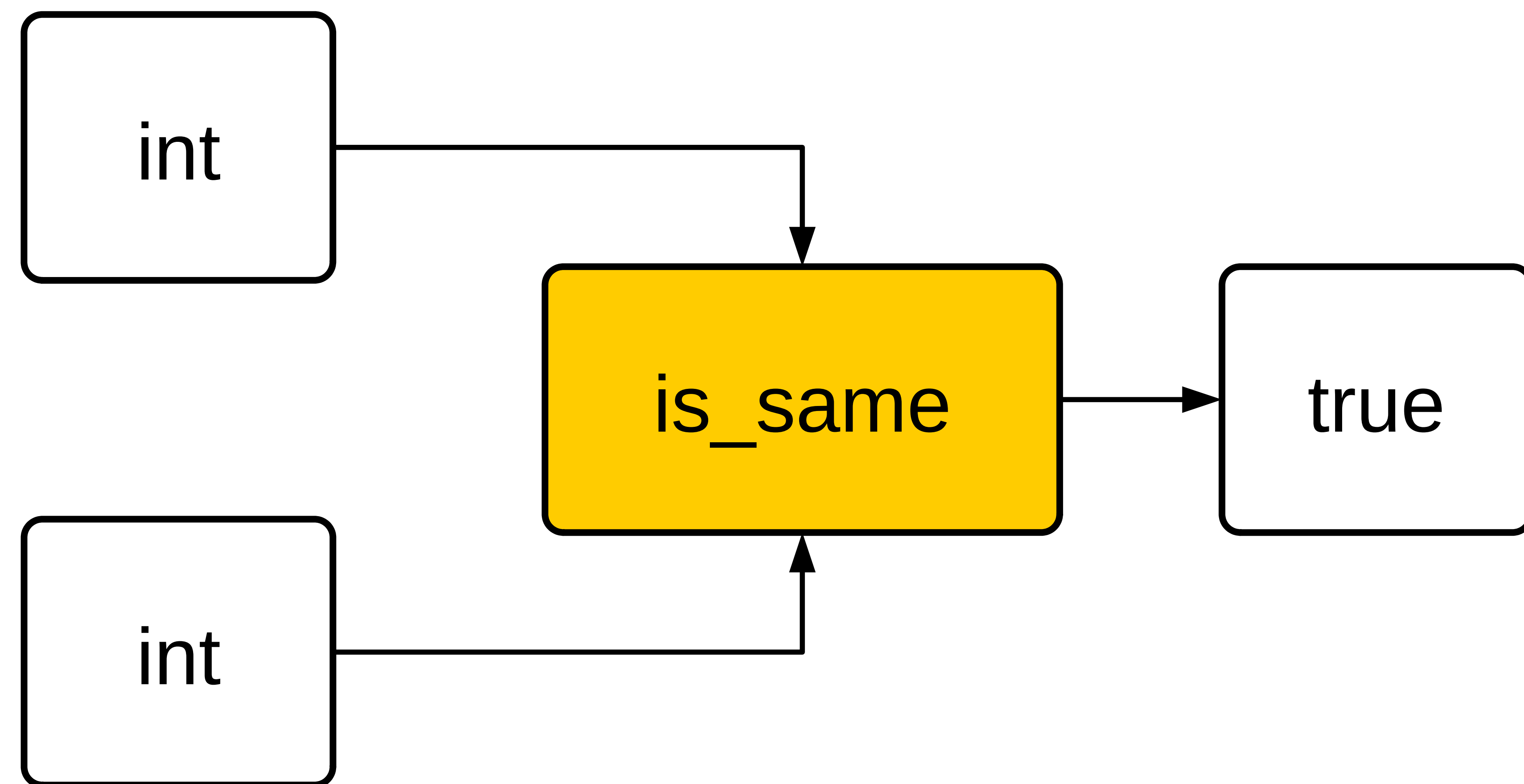
Helper Classes

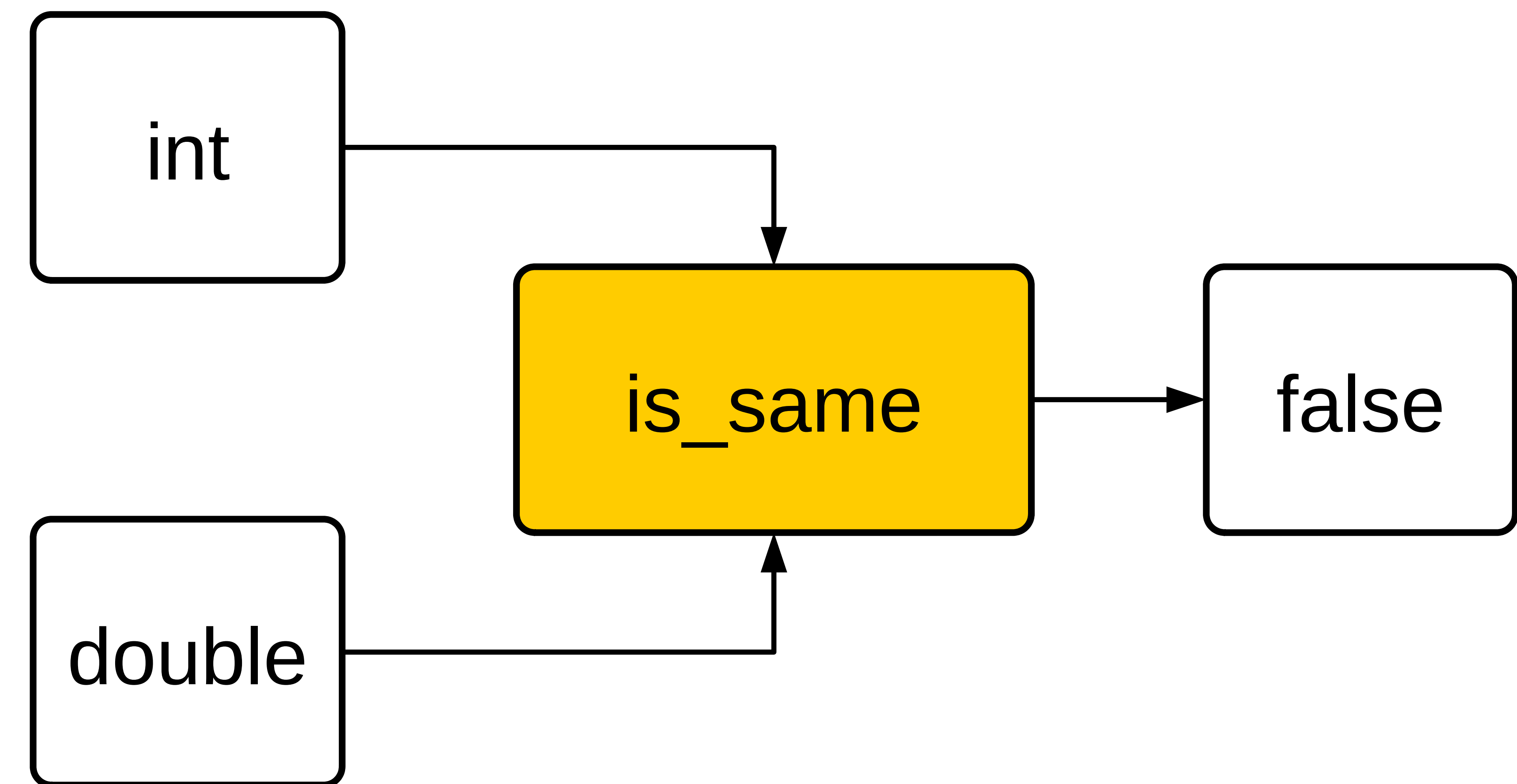
integral_constant (C++11) **bool_constant** (C++17)	compile-time constant of specified type with specified value (class template)
true_type	`std::integral_constant<bool, true>`
false_type	`std::integral_constant<bool, false>`

Primary type categories

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

Метафункция is_same





Метафункция 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);
```

Метафункция 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);
```

2. Списки типов

Что такое списки типов?

- Гетерогенное хранилище “объектов” …
- … с которыми можно работать в compile time …
- … при помощи value-based подхода
- Поэтому std::tuple нам не подходит

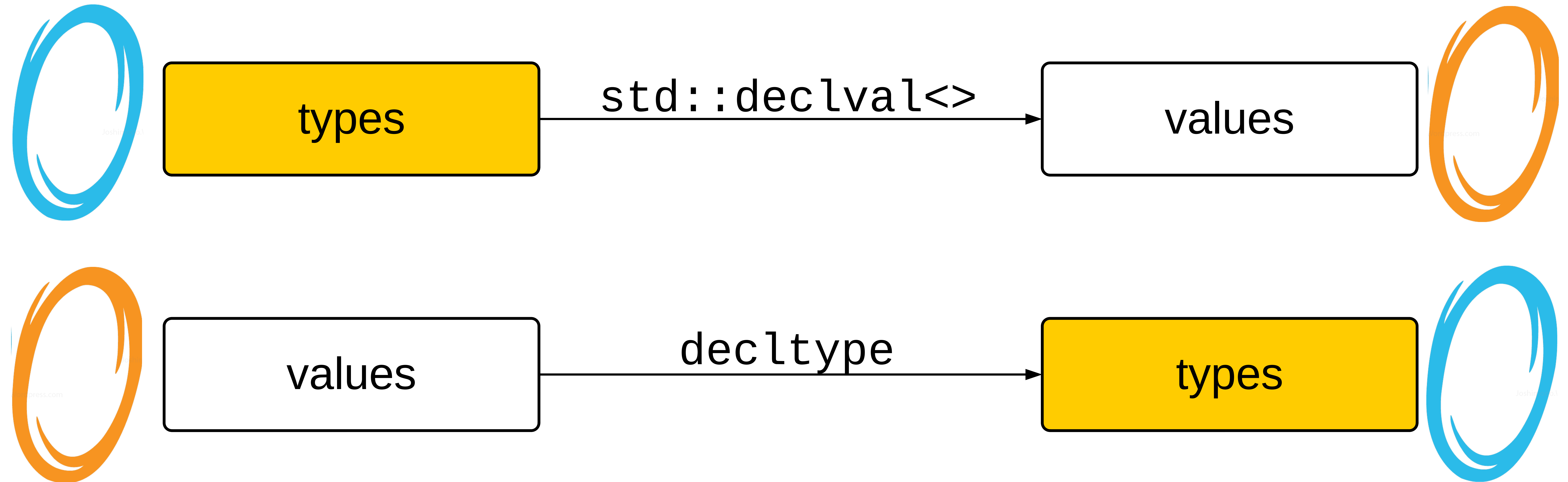
Value-based?

- По-минимому создавать новые типы
- Работать в основном с constexpr функциями
- Комитет по стандартизации двигается в сторону value-based метапрограммирования

Почему value-based?

- Созданные типы никогда не “деаллоцируются” и могут оставить свой след
- Создание новых типов серьезно замедляет компиляцию

Value <-> type телепортации



Type holder

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

Type holder

```cpp
// 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>);
```

Списки типов: как не надо

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
};
```

Списки типов: как надо

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

Списки типов: основные средства

Списки типов: основные средства

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

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);
}
```

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;
}
```

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 {};
}
```

Списки типов: основные средства

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

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; }
```

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; }
```

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; }
```

Функции push и pop

Функции push и pop

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

Функции 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 {};
}
```

Функции 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>{});
```

Функции 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>{});
```

Функции 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>{});
```

Функции push и pop

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

Функция contains

- Узнать, присутствует ли тип T в данном списке

Функция contains: bad

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

Функция 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 {};
```

Функция 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...> {};
```

Функция 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 {};
```

Функция 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>{}));
```

Функция contains: bad

- Рекурсивная инстанциация типов
- Много бойлерплейтного кода

Функция contains: good

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

Функция contains: good

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

Функция 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;
    }
```

Функция 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>{}));
```

Функция contains: good

- Никакой рекурсии
- Код умещается на один слайд

- Кода всё ещё много...

Функция 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{}));
```

Функция find

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

Функция find: bad

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

Функция 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> {};
```

Функция 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> {};
```

Функция 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> {};
```

Функция 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);
```

Функция find: bad

- Рекурсия…
- Много кода...

Функция find: good

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

Функция 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;
        }
    }
```

Функция 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);
```

Функция find: good

- Рекурсии нет
- Кода мало...

- …но недостаточно мало...

Функция 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);
```

Функция 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);
```

Функция 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

Как принимать метафункции?

- Как использовать метафункцию в качестве аргумента?

Шаблонные шаблоны

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

Функция 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);
```

Функция 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<std::is_base_of, std::exception>::type>(
                    type_pack<std::runtime_error, char, std::string>{}) == 0);
```

Больше value-based метапрограммирования!

```
// Metafunction wrapper for value-returning metafunctions
template <template <class...> class F>
struct value_fn {
    template <class... Ts>
    constexpr auto operator()(just_type<Ts>...) {
        return F<Ts...>::value;
    }
};
template <template <class...> class F>
constexpr value_fn<F> value_fn_v;
```

Больше value-based метапрограммирования!

```
// Metafunction wrapper for type-returning metafunctions
template <template <class...> class F>
struct type_fn {
    template <class... Ts>
    constexpr auto operator()(just_type<Ts>...) {
        return just_type<typename F<Ts...>::type>{};
    }
};
template <template <class...> class F>
constexpr type_fn<F> type_fn_v;
```

Функция find_if

```
template <class F, class... Ts>
constexpr std::size_t find_if(F f, type_pack<Ts...> tp) {
    bool bs[] = {f(just_type<Ts>{})...};
    for (std::size_t I = 0; I < size(tp); ++i) {
        if (bs[i]) {
            return I;
        }
    }
    return size(tp);
}
static_assert(find_if(value_fn_v<std::is_pointer>, type_pack<double, int*>{}) ==
              1);
```

Функции any_of, all_of, none_of

Функции any_of, all_of, none_of

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

Функции any_of, all_of, none_of

```
template <class F, class... Ts>
constexpr bool all_of(F f, type_pack<Ts...>) {
    return (... && f(just_type<Ts>{}));
}
template <class F, class... Ts>
constexpr bool any_of(F f, type_pack<Ts...>) {
    return (... || f(just_type<Ts>{}));
}
```

Функции any_of, all_of, none_of

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

Функции any_of, all_of, none_of

```
// Type-based versions are shorter to call in some cases...
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{}));
```

Функция transform

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

Функция transform

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

// Value-based version is too simple...

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

Функция reverse

Функция reverse

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

Функция 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...>{});
}
```

Функция 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>{});
```

Функция get

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

Функция get: bad

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

Функция 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; };
```

Функция 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;
};
```

Функция 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-frendly
struct get_impl<I> {};
```

Функция 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>{});
```

Функция 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>{});
```

Функция get: bad

- Всем понятно, почему это плохо...

Функция get: good

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

std::index_sequence

```cpp
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`:

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

```cpp
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

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

Функция get: good

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

Функция 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>... {};
};
```

Функция get: good

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

Функция 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 {}; }
```

Функция 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>{});
```

Функция get: good

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

Comma operator

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

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

Функция get: best

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

Функция get: best

```
template <class IS>
struct get_impl;

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

Функция 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*, ...);
};
```

Функция 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*, ...);
};  //  result type  ^ deduced from 3rd parameter ^ !
```

Функция 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>{});
```

Функция get: best

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

Кстати про reverse...

```
template <std::size_t... is, class TP>
constexpr auto reverse_impl(std::index_sequence<is...>, TP tp) {
    return type_pack<typename decltype(get<size(tp) - is – 1>(tp))::type...>{};
}
template <class... Ts>
constexpr auto reverse(type_pack<Ts...> tp) {
    return reverse_impl(std::index_sequence_for<Ts...>{}, tp);
}
static_assert(reverse(type_pack<int, double, char>{}) ==
                      type_pack<char, double, int>{});
```

Функция generate

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

Функция 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;
};
```

Функция 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...>;
};
```

Функция 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>{});
```

Функция generate: bad

- Рекурсия, которая не даст нам создать большой список

Функция generate: best

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

Функция 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)...);
}
```

Функция 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>{});
```

Функция 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
```

Функция filter

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

Функция filter: bad

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

Функция 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 {
```

Функция 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...>{});
    }
}
```

Функция 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{});
```

Функция filter: bad

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

Функция 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{};
    }
}
```

Функция 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...>{};
}
```

Функция 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*>{});
```

Списки типов: всё вместе

- Policy-based контейнеры и алгоритмы, без привязки к позиции в списке аргументов
- Фабрики объектов, работающие в compile time
- Основа для еще более хардкорного метапрограммирования (но об этом в следующий раз)

3. Итог

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

Итог: метапрограммирование

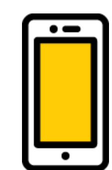
- Метапрограммирование значительно расширяет границы возможного
- Метапрограммирование не является чем-то дружелюбным, пока к нему не привыкнешь...
- … как и весь C++
- Все вместе ждём reflection

Спасибо за внимание!

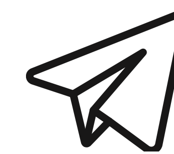
Олег Фатхиев

Младший разработчик

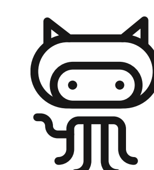
tender-bum@yandex-team.ru



+7 919 155-83-40



tender-bum



github.com/ofats/meta_evo