ВЫВОД ТИПОВ

Правые ссылки, перегрузка по ссылкам и вывод типов.

К. Владимиров, Syntacore, 2024
mail-to: konstantin.vladimirov@gmail.com

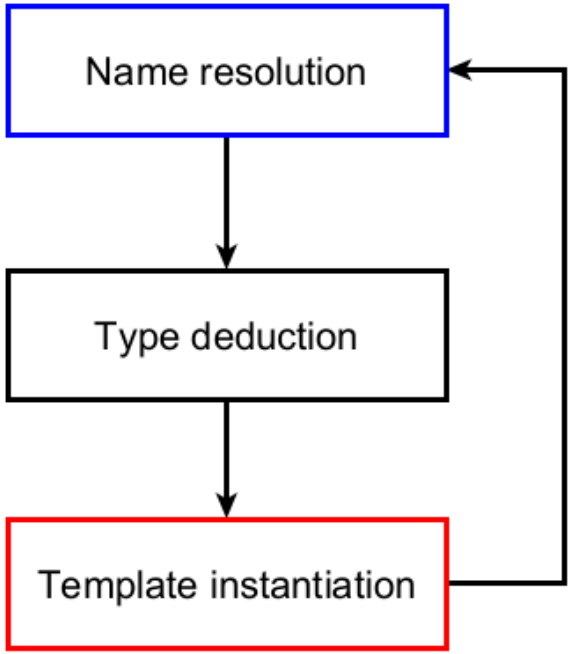
Нормальный процесс

```
template <typename T> T min(T x, T y) {
  return x < y ? x : y;
}

template <typename T> T min(T x, T y, T z) {
  auto t = min(x, y);
  return min(t, z);
}
```

- Новое инстанцирование порождает новые имена, для них тоже делается lookup и т. д.

`min(1, 2, 3) → min(x, y) → min(t, z) → ⊥`



Вывод типов функциями

- Для параметров, являющихся типами, работает вывод типов
- `int x = max(1, 2); // → int max<int>(int, int);`
- При выводе режутся ссылки и внешние cv-квалификаторы

```
const int& a = 1;
const int& b = 2;
int x = max(a, b); // → int max<int>(int, int);
```

- Вывод не работает, если он не однозначен

```
unsigned x = 5; do_nth_power(x, 2, n); // FAIL
int a = 1; float b = 1.0; max(a, b); // FAIL
```

Вывод типов после подстановки

- Вывод типов внутри шаблонной функции даёт точки вывода, где разрешить тип можно только после подстановки

```
template <typename T> T max(T x, T y) { .... }

template <typename T> T min(T x, T y) { .... }

template <typename T> bool
test_minmax(const T &x, const T &y) {
  if (x > y) return test_minmax(y, x);
  return min(x, y) == x && max(x, y) == y;
}
```

- Таким образом поиск имён, вывод и подстановка включаются попеременно.

Вывод уточнённых типов

- Иногда шаблонный тип аргумента может быть уточнён ссылкой или указателем и cv-квалификатором

```
template <typename T> T max(const T& x, const T& y);
```

- В этом случае выведенный тип тоже будет уточнён

```
int a = max(1, 3); // → int max<int>(const int&, const int&);
```

- Уточнённый вывод иначе работает с типами: он сохраняет cv-квалификаторы.

```
template <typename T> void foo (T& x);

const int &a = 3;
foo(a); // → void foo<const int>(const int& x);
```


https://godbolt.org/z/Y6csWaec6

Вывод ещё более уточнённых типов

- Вывод типов работает шире, чем люди обычно думают

```
template<typename T> int foo(T(*p)(T));
int bar(int);
foo(bar); // → int foo<int>(int(*)(int));
```

- Могут быть выведены даже параметры, являющиеся константами

```
template<typename T, int N> void buz(T const(&)[N]);
buz({1, 2, 3}); // → void buz<int, 3>(int const(&)[3]);
```

- Общее правило: вывод типов матчит сложные композитные типы.

https://godbolt.org/z/o6E1Kh5fq

Частичный вывод типов

- В некоторых случаях у нас просто нет **контекста вывода**.

```
template <typename DstT, typename SrcT>
DstT implicit_cast(SrcTconst& x) {
  return x;
}

double value = implicit_cast(-1); // fail!
```

- Тогда мы можем указать необходимое и положиться на вывод остального.

```
double value = implicit_cast<double, int>(-1); // ok

double value = implicit_cast<double>(-1); // ok
```


https://godbolt.org/z/7h9G6MPTh

Параметры по умолчанию

- Допустим у вас есть функция, берущая по умолчанию плавающее число.

```
template <typename T> void foo(T x = 1.0);
```

- Увы, если его не указать, вывод типов работать не будет.

```
foo(1);             // ok, foo<int>(1);
foo<int>();         // ok, foo<int>(1.0 narrowed to int);
foo();              // fail
```

- Тем не менее, ситуацию можно исправить. Трюк не так уж и сложен. Догадки?

Шаблонные параметры по умолчанию

- Допустим у вас есть функция, берущая по умолчанию плавающее число.

```
template <typename T = double> void foo(T x = 1.0);
```

- Заметьте: во втором случае ниже вывода типов всё ещё нет.

```
foo(1);             // ok, foo<int>(1);
foo<int>();         // ok, foo<int>(1.0 narrowed to int);
foo();              // fail
```

- Параметр по умолчанию шаблона в данном случае подсказывает компилятору что делать.

https://godbolt.org/z/j4PW7j66s

Вывод типов non-type параметров

- Что делать компилятору если в шаблоне вывод типов требуется в списке параметров?

```
template <auto n> int foo() { .... }

foo<1>();
foo<1.5>();
```

- В этом случае компилятор выводит тип построением invented expression.

```
auto n = 1;
auto n = 1.5;
```

- Только clang 18 поддержал FP NTTP из C++20

https://godbolt.org/z/3MWxd3n7c

Вывод конструкторами классов (C++17)

- Начиная с C++17 конструкторы классов могут использоваться для вывода типов

```
template<typename T> struct container {
  container(T t);
  // и так далее
};

container c(7); // → container<int> c(7);

auto c = container(7); // → аналогично

auto c = new container(7); // → аналогично
```

Проблема:

- Конструктор класса сам может быть шаблонным

```
template<class T> struct container {
 template<class Iter> container(Iter beg, Iter end);
 // и так далее
};

vector<double> v;
auto d = container(v.begin(), v.end()); // → container<double>?
```

```
<source>:19:12: error: no viable constructor or deduction guide for deduction of
template arguments of 'container'
   19 |    auto d = container(v.begin(), v.end());
      |             ^
```

https://godbolt.org/z/zs7bcGa8j

Хинты для вывода (C++17)

- Также пользователь может помочь выводу в сложных случаях

```
template<class T> struct container {
 template<class Iter> container(Iter beg, Iter end);
 // и так далее
};

// пользовательский хинт для вывода

template<class Iter> container(Iter b, Iter e) ->
   container<typename iterator_traits<Iter>::value_type>;

vector<double> v;
auto d = container(v.begin(), v.end()); // → container<double>
```

https://godbolt.org/z/sY8nc7bnG

Неявные хинты

- В случае если конструктора нет, мы должны писать явный хинт.

```
template <typename T> struct Value { T value; };

template <typename T> Value(T) -> Value<T>; // explicit
```

- Но если конструктор есть, то хинт будет неявным (т.е. сгенерированным).

```
template <typename T> struct Value {
  T value_;
  Value(T val) : value_(val) {}
};

template <typename T> Value(T) -> Value<T>; // implicit
template <typename T> Value(Value<T>) -> Value<T>; // implicit
```

Обсуждение

- Помните ли вы что такое правые ссылки?

Перегрузка по ссылке

- Перегрузка функций

```
int foo(int &p);  // 1
int foo(int &&p); // 2

int x = 1;

foo(x); // → 1
foo(1); // → 2
```

- Можно сделать пример перегрузку интересней, добавив:

```
int foo(const int &p);  // 3
int foo(const int &&p); // 4
```


https://godbolt.org/z/vPKocGjGf

Перемещающие конструкторы

- Конструктор берущий rvalue ref не обязан сохранять значение (т.к. это rvalue).

```
class Buffer {
  size_t sz; int *buf;

public:

  Buffer(Buffer&& rhs) noexcept : sz(rhs.sz), buf(rhs.buf) {
    rhs.sz = 0; rhs.buf = nullptr;
  }

  Buffer(const Buffer& rhs) : sz(rhs.sz), buf(new int[sz]) {
    std::copy(rhs.buf, rhs.buf + sz, buf); // довольно дорого
  }
};
```

Спецсемантика

- Компилятор знает, что копирующий конструктор копирует, а перемещающий перемещает.
- Это даёт ему возможность игнорировать побочные эффекты в конструкторах.
- И это же ставит вопрос о том, как компилятор узнаёт специальный конструктор?

https://godbolt.org/z/oqYvW8WEf

Что может быть copy и move ctors

- Не-шаблонный `X([cv] X&)` это конструктор копирования [class.copy.ctor].
- Не-шаблонный `X([cv] X&&)` это конструктор перемещения.
- Допустимы последующие аргументы, если они дефолтные.

```
struct Buffer {
  Buffer(volatile Buffer&& rhs, int x = 10); // ok, move ctor
```

- Интересно что для copy и move operator= правила те же но дополнительные аргументы запрещены [class.copy.assign].
- Вы не имеете права (в C++20 как минимум) иметь конструктор `X([cv] X)`.
- И если это не определили вы, это определит компилятор.

Правила для implicit методов

- Деструктор по умолчанию
  - defaulted если нет ни одного пользовательского деструктора
- Конструктор копирования (аналогично ведёт себя копирующее присваивание)
  - deleted если его нет, но есть пользовательское перемещение или move ctor.
  - Иначе defaulted если его нет (но если при этом есть деструктор или парный копирующий метод, то это deprecated).
- Перемещающий конструктор (аналогично перемещающее присваивание)
  - deleted если его нет, но при этом есть парный перемещающий метод или copy ctor или копирующее присваивание или деструктор.
  - Иначе defaulted если его нет.
- Вы видите какие-нибудь интересные следствия из этих правил?

Интересное следствие

- Наличие вирутального деструктора убивает перемещение членов класса.

```
class Base {
  std::string s_;
public:
  Base(std::string s) : s_(s) {}
  virtual int foo();
  virtual ~Base();
};

Base b1;
Base b2 = std::move(b1); // копирование строки
```

Правило пяти

- Классическая идиома проектирования rule of five утверждает, что:

"Если ваш класс требует нетривиального определения хотя бы одного из пяти методов:

1. *копирующего конструктора*
2. *копирующего присваивания,*
3. *перемещающего конструктора*
4. *перемещающего присваивания*
5. *деструктора*

то вам лучше бы нетривиально определить все пять"

Moved-from state

```
int x = 1;
int a = move(x);
assert(x == a); // ???

Buffer y{10};
Buffer b = move(y);
assert(y == b); // ???
```

- Что можно сказать о приведённых assertions?

Moved-from state

```
int x = 1;
int a = move(x);
assert(x == a); // ok (unspecified for library type)

Buffer y{10};
Buffer b = move(y);
assert(y == b); // fail due to Buffer move-ctor
```

- В принципе стандарт не определяет moved-from как особое состояние.
- Хорошо спроектированная программа либо не изменит rhs, либо приведет его в "консистентное непредсказуемое состояние".

Аннотация методов

- Объект класса в зависимости от того lvalue он или rvalue может вызывать разные методы.

```
struct S {
  int foo() &; // 1
  int foo() &&; // 2
};

extern S bar ();
S x {};

x.foo(); // 1
bar().foo(); // 2
```

Аннотация методов

```
class X {
  vector<char> data_;
public:
  X() = default;
  vector<char> const & data() const & { return data_; }
  vector<char> && data() && { return move(data_); }
};

X obj;
vector<char> a = obj.data(); // copy
vector<char> b = X().data(); // move
```

Вывод типов для переменных

- Ключевое слово auto выводит типы так же как шаблоны функций с теми же правилами для уточнённых типов

```
const int &x = 42;
auto y = x; // → int y = x;
auto& z = x; // → const int& z = x;
```

- Интересный вопрос тут что auto выведет для rvalue ref?

```
auto&& v = x; // → ???
```

- Или что то же самое, что выведет шаблон в тех же обстоятельствах?

```
template <typename T> foo(T&& v); foo(x); // → ???
```

Идентичность и перемещаемость

- Результат выражения можно охарактеризовать идентичностью либо перемещаемостью.

```
extern int foo();

int x = foo(); // rvalue expression

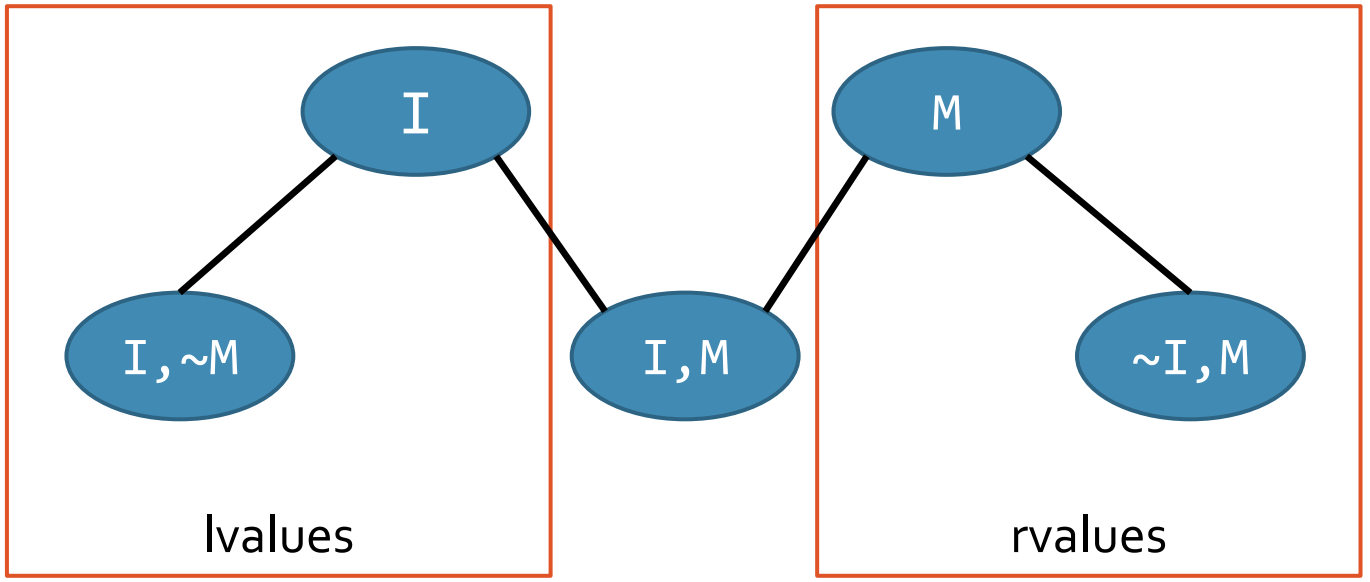
int y = x; // lvalue expression
```

- Здесь выражение x обладает идентичностью, выражение foo() перемещаемо.
- Начиная с 2011 года выражение может быть и тем и другим.

```
Buffer d; Buffer b = std::move(d); // xvalue expression
```

Идентичность и перемещаемость

IM-код	Название
I	glvalue
M	rvalue
I, ~M	lvalue
I, M	xvalue
M, ~I	prvalue



Правила свёртки ссылок

- В контексте вывода типов ссылки сворачиваются и левые ссылки выигрывают. Но как это работает для вызова функций?

```
template <typename T> int foo(T&&);

int x;
int& y = x;

foo(x); // → ?
foo(y); // → ?
```

- Должна ли тут быть разница?

Главный хак в выводе типов

- При сворачивании типов шаблонами мы должны также вывести тип шаблонного параметра.

```
template <typename T> int foo(T&&);

int x;
int& y = x;

foo(x); // → int foo<int&>(int&)
foo(y); // → int foo<int&>(int&)
foo(5); // → int foo<int>(int&&)
```

- Для консистентности он выводится в ссылку для lvalue но не для rvalue!

Заглядываем внутрь move

- Именно благодаря этому хаку работает move

```
template <typename T,
          typename R = std::remove_reference_t<T>&&>
R almost_move(T&& a) {
  return static_cast<R>(a);
}

int x; auto&& t = almost_move(x); // T → int&, T&& -> int&,
                                  R -> int&&, t -> int&&
```

Универсальность ссылок

- Правила вывода дают интересную картину: auto& это всегда lvalue ref, но auto&& это либо lvalue ref, либо rvalue ref (зависит от контекста).

```
auto &&y = x; // x это some& → y это some&
```

- Это в целом работает и для шаблонов

```
template <typename T> void foo(T&& t);
foo(x); // аналогично
```

- Такие ссылки называют forwarding references. Майерс предложил называть их универсальными ссылками.
- Важное требование универсальности ссылки: контекст вывода типов.

Неуниверсальные ссылки

- Контекст сворачивания требует вывода типов, а не их подстановки:

```
template <typename T> struct Buffer {
  void emplace(T&& param); // substitute T

template <typename T> struct Buffer {
  template <typename U> void emplace(U&& param); // deduce U
```

- Контекст для сворачивания не будет создан, если тип уточнён более, чем &&

```
template <typename T> void buz(const T&& param);
```

- Это так же верно для auto

```
const auto &&x = y; // никакого сворачивания ссылок
```

Мотивация для decltype

- Ключевое слово auto либо режет типы либо добавляет уточнители.
- Кроме того оно использует правую часть как для вывода, так и для присваивания.

```
const int x = 5;

auto y = x; // y → int
auto &z = x; // z → const int&
```

- Чтобы вывести точный тип, у нас есть decltype.

```
decltype(x) t = 6; // t → const int
```

- Но тут появляется один вопрос….

Decltype: что такое точный тип?

- Приоритет для decltype это точный тип параметра.

```
const int &x = 42;

decltype(x) y = 42; // → const int &y = 42;
```

- Это прекрасно. Но есть проблема:

```
struct Point { int x, y; };

Point porig {1, 2};
const Point &p = porig;

decltype(p.x) x = 0; // здесь int x или const int &x?
```

Decltype: name and expression

```
struct Point { int x, y; };

Point porig {1, 2};
const Point &p = porig;
```

- Случай decltype(id-expr)

```
decltype(p.x) x = 0; // → int x = 0;
```

- Случай decltype(expr)

```
decltype((p.x)) x = 0; // → const int &x = 0;
```

- Точный тип это decltype(name), а вот decltype(expr) работает от категории.

Правила для decltype(id-expr)

```
int x; decltype(x) t1 = y; // name → int
```

- `decltype(id-expr)` различает категории значений.
- Для lvalues он добавляет левую ссылку.

```
decltype((x)) t2 = y; // lvalue expr → int&
```

- Для xvalues он добавляет правую ссылку.

```
decltype(std::move(x)) t3 = 1; // xvalue expr → int&&
```

- Для prvalues он ничего не добавляет.

```
decltype(x + 0) t4; // prvalue expr → int
```

Абстракция значения

- В некоторых случаях (например для использование внутри `decltype`) хочется получить значение некоего типа.
- Часто для этого используется конструктор по умолчанию

```
template <typename T> struct Tricky {
  Tricky() = delete;
  const volatile T foo();
};

decltype(Tricky<int>().foo()) t; // ошибка
```

- Но что делать, если его нет? Что такое "значение вообще" для такого типа?

Абстракция значения: memory chunk

- Человек, искушённый в C мог бы сказать, что значение как таковое это результат приведения сырой памяти.

```
template <typename T> struct Tricky {
  Tricky() = delete;
  const volatile T foo();
};

decltype(((Tricky<int>*)0)->foo()) t; // работает, но это UB
```

- В compile-time любой `reinterpret_cast` запрещён стандартом.

Абстракция значения: declval

- Интересный способ решить эти проблемы это ввести шаблон функции (который выводит типы) без тела (чтобы его нельзя было по ошибке вызвать).

```
template <typename T>
std::add_rvalue_reference_t<T> declval(); // нет тела
```

- Теперь всё просто

```
template <typename T> struct Tricky {
  Tricky() = delete;
  const volatile T foo ();
};

decltype(declval<Tricky<int>>().foo()) t = 0; // ok, cv-int
```

Вершина всего: decltype(auto)

- Совмещает ~~худшие~~ лучшие стороны двух механизмов вывода
- Вывод типов является точным, но при этом выводится из всей правой части

```
double x = 1.0;
decltype(x) tmp = x; // два раза x не нужен
decltype(auto) tmp = x; // это именно то, что нужно
```

- Однако что стоит справа expr или id-expr? Зависит от выражения…

```
decltype(auto) tmp = x; // → double tmp = x;
decltype(auto) tmp = (x); // → double& tmp = x;
```

Улучшаем almost_move

- Предыдущий вариант:

```
template <typename T,
          typename R = std::remove_reference_t<T>&&>
R almost_move(T&& a) {
  return static_cast<R>(a);
}
```

- Используем decltype(auto)

```
template <typename T> decltype(auto) almost_move(T&& a) {
  using R = std::remove_reference_t<T>&&;
  return static_cast<R>(a); // decltype(auto) от xvalue
}
```

Xvalues

- `xvalue` (от expiring value) это, glvalue, которое ведёт себя как функция, возвращающая правую ссылку.
- Функторы-определители
  - `is_rvalue_reference<T>::value` связывает тип с xvalue
  - `is_lvalue_reference<T>::value` связывает тип с lvalue
- Например

```
int x = 42;

assert(is_lvalue_reference<decltype(x)>::value);

assert(is_rvalue_reference<decltype(std::move(x))>::value);
```

Прозрачная оболочка

- Идея прозрачной оболочки: вызывать переданную ей функцию с минимальными накладными расходами.

```
template <typename Fun, typename Arg> decltype(auto)
transparent(Fun fun, Arg arg) {
  return fun(arg);
}
```

- Увы, её недостаток в том, что она не слишком прозрачна.

```
extern Buffer foo(Buffer x);

Buffer b;
Buffer t = transparent(&foo, b); // тут явное копирование b
```

Снова прозрачная оболочка

- Возможный выход: сделать аргумент ссылкой.

```
template <typename Fun, typename Arg> decltype(auto)
transparent(Fun fun, Arg& arg) {
  return fun(arg);
}
```

- Но появляется новая беда: теперь rvalues не проходят в функцию.

```
extern Buffer foo(Buffer x);

Buffer b;
Buffer t = transparent(&foo, b); // ok
Buffer u = transparent(&foo, foo(b)); // ошибка компиляции
```

Снова прозрачная оболочка

- Возможный выход: перегрузить по константной ссылке

```
template <typename Fun, typename Arg> decltype(auto)
transparent(Fun fun, Arg& arg) { return fun(arg); }

template <typename Fun, typename Arg> decltype(auto)
transparent(Fun fun, const Arg& arg) { return fun(arg); }
```

- Теперь вроде всё хорошо

```
Buffer t = transparent(&foo, b); // ok
Buffer u = transparent(&foo, foo(b)); // ok, но...
```

- Вы видите тут ещё одну проблему?

Снова прозрачная оболочка

- Решение для проблемы числа перегрузок: универсализовать ссылку.

```
template <typename Fun, typename Arg> decltype(auto)
transparent(Fun fun, Arg&& arg) {
  return fun(arg);
}
```

- Увы, у нас всё ещё копируется аргумент если он rvalue.

```
Buffer t = transparent(&foo, b); // ok
Buffer u = transparent(&foo, foo(b)); // ok, но копируется
```

Чего бы нам хотелось

- Нам бы хотелось условного перемещения.

```
template <typename Fun, typename Arg>
decltype(auto) transparent(Fun fun, Arg&& arg) {
  if (arg это rvalue)
    return fun(move(arg));
  else
    return fun(arg);
}
```

- И в языке оно есть.

```
template <typename Fun, typename Arg>
decltype(auto) transparent(Fun fun, Arg&& arg) {
  return foo(std::forward<Arg>(arg));
}
```

Внутри forward

- `forward` это **тоже** просто `static_cast`

```
template<typename S>
S&& almost_forward(std::remove_reference_t<S>& a) {
  return static_cast<S&&>(a);
}
```

- Тут нет контекста вывода.
- Поэтому необходимо использование управляющего типа S, который универсально сворачивается в нужный тип ссылки.

Решение: использовать std::forward

- Теперь когда мы понимаем как это работает….
- ```
  template <typename Fun, typename Arg>
  decltype(auto) transparent(Fun fun, Arg&& arg) {
    return foo(std::forward<Arg>(arg));
  }
  ```
- Видно, что всё хорошо.

```
Buffer t = transparent(&foo, b); // ok
Buffer u = transparent(&foo, foo(b)); // ok
```

- Это называется perfect forwarding и бывает удивительно полезной идиомой.
- Три главных составляющих: контекст вывода T, тип T&& и forward<T>.
```

Пример: простые случаи проброса

- При форвардинге аргументов сохраняется их тип и категория значения

```
int g(int &&t) { return 1; }
int g(int &t) { return 2; }

template <typename T> int h(T &&t) {
  return g(std::forward<T>(t));
}

int x = 0;
int y = h(x + 0); // y == 1
int z = h(x);     // z == 2
```

Обсуждение

- Пожалуй есть всего три функции, для которых имеет смысл возвращать правую ссылку (то есть производить xvalue)
  - `std::move`
  - `std::forward`
  - `std::declval`
- Если вы хотите написать свою функцию, которая будет возвращать && это значит, что
  - Вы что-то делаете не так
  - Вы хотите ещё раз написать одну из упомянутых выше функций
  - Вы пишете функцию, аннотированную как &&

Пример Йосьюттиса

```
class Customer {
  MyString fst, snd;

public:
  Customer(const MyString &s1, const MyString &s2) :
    fst(s1), snd(s2) {}
};
```

- Кажется этот пример очень неэффективен.
- Улучшим его форвардингом?

https://godbolt.org/z/roTTbPfGa

Уменьшаем копирования

```
class Customer {
  MyString fst, snd;

public:
  template <typename S1, typename S2>
  Customer(S1 &&s1, S2 &&s2 = "") :
    fst(std::forward<S1>(s1)), snd(std::forward<S2>(s2)) {}
};
```

- Казалось бы тут можно и закончить. Увы...

https://godbolt.org/z/oco69zjd4

Внезапная проблема

```
class Customer {
  MyString fst, snd;

public:
  template <typename S1, typename S2>
  Customer(S1 &&s1, S2 &&s2 = "") :
    fst(std::forward<S1>(s1)), snd(std::forward<S2>(s2)) {}
};

.......

Customer N("Nico");
```


https://godbolt.org/z/rxMj7MPnc

Спасаем ситуацию

```
class Customer {
  MyString fst, snd;

public:
  template <typename S1, typename S2 = const char *>
  Customer(S1 &&s1, S2 &&s2 = "") :
    fst(std::forward<S1>(s1)), snd(std::forward<S2>(s2)) {}
};

.......

Customer N("Nico"); // ok
```


https://godbolt.org/z/rcngrrrje

Копирующий конструктор

```
class Customer {
  MyString fst, snd;

public:
  template <typename S1, typename S2 = const char *>
  Customer(S1 &&s1, S2 &&s2 = "") :
    fst(std::forward<S1>(s1)), snd(std::forward<S2>(s2)) {}
};

.......

Customer N("Nico");

Customer M(N); // как вы думаете?
```


https://godbolt.org/z/W5ox6TMsq

Задачи, часть 1

- Вам предлагается сделать вывод для типа void следующим образом.

```
template <typename T> int foo(T x) {
  return 42;
}

int x = foo(); // → int foo<void>()
```

- Представьте, что вы в комитете стандартизации.
- Какие будут ваши аргументы и какое решение вы примете?
- Постарайтесь при согласии или отказе серьёзно обосновать что конкретно ещё придётся исправить в механизмах языка.

Задачи, часть 2

- Обоснуйте что будет в этом случае.

```
int g(int &&t) { return 1; }
int g(int &t) { return 2; }

template <typename T>
void h(T &&t) { g(std::forward<T> (t)); }

const int x = 1; int z = h(x); // ???
```

- Предоставьте полный ручной вывод std forward.

Задачи, часть 3

- Напишите вывод типа функции из паттерн матчинга в стиле Haskell

```
f g [] = []
f g (x:xs) = (g x : f g xs)
```

- Результат вывода это сигнатура функции или провал вывода.

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

- Допустимые синтаксические элементы: вызов функции, пустой список, раскрытие списка, идентификаторы.

```
f g x = g x
```

- Тут ошибка: нельзя вывести `x`.

Литература

- ISO/IEC, Information technology – Programming languages – C++, 14882:2020
- Bjarne Stroustrup, The C++ Programming Language (4th Edition)
- Scott Meyers, "Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14", 2013
- Scott Meyers "Type Deduction and Why You Care", CppCon 2014
- Nicolai Josuttis "The Nightmare of Move Semantics for Trivial Classes", CppCon 2017
- Davide Vandevoorde, Nicolai M. Josuttis, C++ Templates. The Complete Guide, 2nd edition, Addison-Wesley Professional, 2017
- Back to Basics: C++ Move Semantics - Andreas Fertig - CppCon 2022