НЕСБЫТОЧНЫЕ МЕЧТЫ

Или немного про семантические процессы, основанные на отношениях частичного порядка и про те, которые являются его строительными блоками.

К. Владимиров, 2026
mail-to: konstantin.vladimirov@gmail.com

НЕСБЫТОЧНЫЕ МЕЧТЫ

Или немного про семантические процессы, основанные на отношениях частичного порядка и про те, которые являются его строительными блоками.

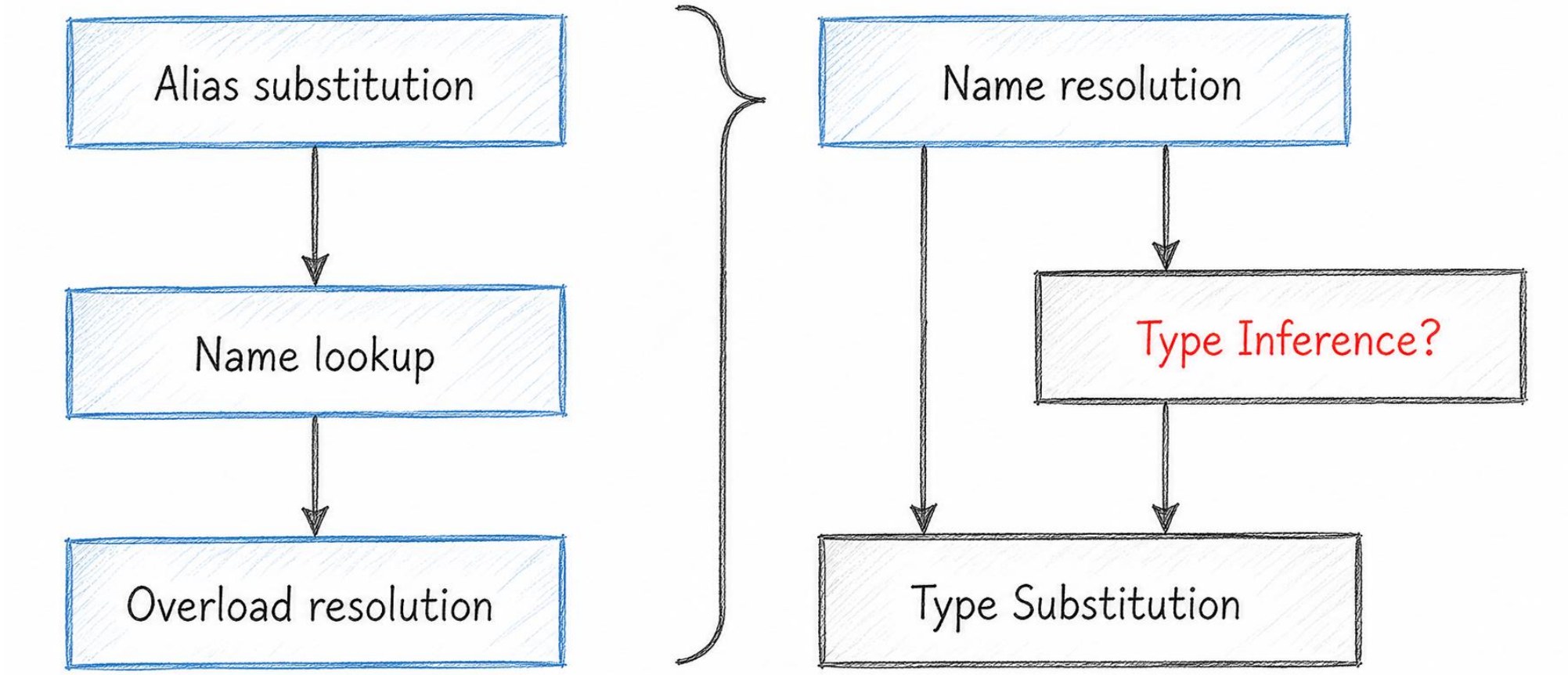
К. Владимиров, 2026
mail-to: konstantin.vladimirov@gmail.com

НЕСБЫТОЧНЫЕ МЕЧТЫ

Или немного про семантические процессы, основанные на отношениях частичного порядка и про те, которые являются его строительными блоками.

К. Владимиров, 2026
mail-to: konstantin.vladimirov@gmail.com

Семантические процессы



Семантические процессы крупный план

У меня есть мечта

```
auto transform(auto f, auto ls) {
  if (ranges::empty(ls)) return {};
  auto first = ranges::begin(ls);
  auto last = ranges::end(ls);
  auto xs = ranges::subrange{ranges::next(first), last};
  return views::single(f(*first))
       | views::concat(transform(f, xs));
}
```

- И мы отсюда могли бы вывести тип.

```
template <input_range R, typename F>
requires invocable<F, range_reference_t<R>>
transformed_view<F, R> transform(F f, R r);
```


https://godbolt.org/z/aj8YTKnWd

В языке Haskell это вполне реально

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

- Компилятор выведет для нас наибольший общий (principle) type.
- "Принципиальный тип" может быть шаблоном, тогда компилятор выведет ограничения на параметры шаблона.

```
map :: (Z -> Y) -> [Z] -> [Y]
```

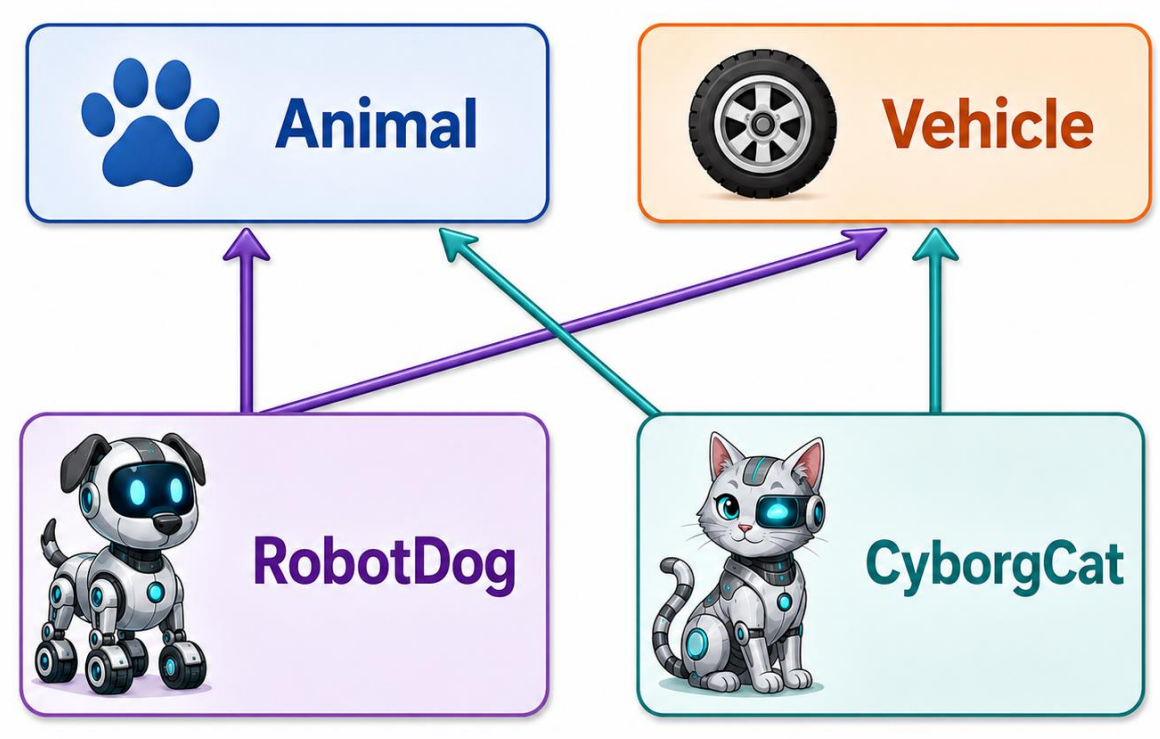
- Это волшебство работает на **алгоритме унификации Хиндли-Милнера.**
- **Почему мы не можем иметь это в C++?**

Унификация ломается наследованием

- Допустим в схеме наследования справа нам надо вывести f.

```
g :: Animal → Number
h :: Vehicle → Number

f g h x = g x + h x
```

- Что такое type(x)?



Унификация ломается наследованием

- Допустим в условиях ограничений справа нам надо вывести f.

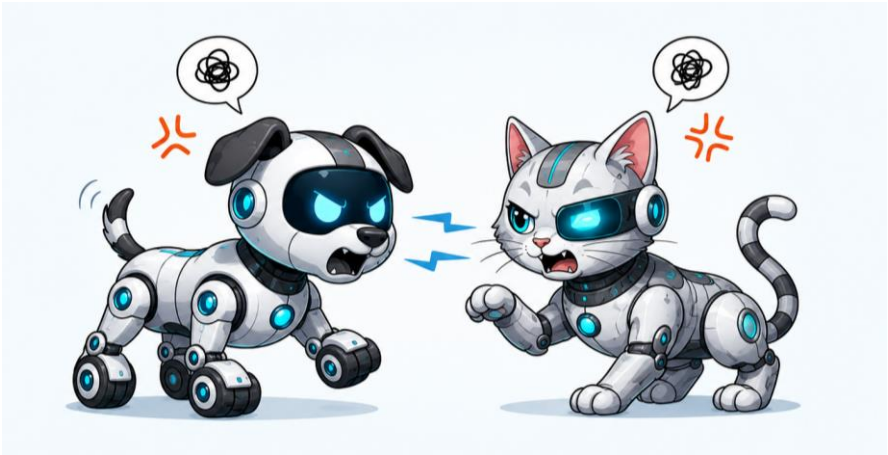
```
g :: Animal → Number
h :: Vehicle → Number

f g h x = g x + h x
```

- Вместо равенств имеем неравенства.

```
type(x) ≤ Animal
type(x) ≤ Vehicle
```

- Решением для x являются типы CyborgCat и RobotDog, но между ними нельзя выбрать.

```
CyborgCat ≤ Animal
CyborgCat ≤ Vehicle
RobotDog ≤ Animal
RobotDog ≤ Vehicle
```



Унификация ломается перегрузкой

- У перегрузок для разного числа параметров нет принципиального типа.

```
g :: Int → Int
g :: Int → Int → Int

f h = h (g 1)
```

- Классический Хиндли-Милнер тут выведет

```
f :: (Int -> a) -> a
f :: ((Int -> Int) -> a) -> a
```

- Значит principal type `f :: (b -> a) -> a`
- Но он слишком общий, допуская например `(Float -> Double) -> Double`

Почему перегрузка это удобно?

- Разные, но взаимосвязанные типы.

```
void Foo(const char* s);
void Foo(std::string s) { Foo(s.c_str()); }
```

- Разное количество параметров.

```
auto s1 = twine("Hello", name).str();
auto s2 = twine("Hello", name, " ", surname).str();
```

- Оптимизации.

```
void vector<T>::push_back(const T&);
void vector<T>::push_back(T&&);
```

Overload set: transform

- Хорошее ли это множество перегрузки?

```
template <class InputIt, class OutputIt, class UnaryOp>
OutputIt transform(InputIt first1, InputIt last1,
                   OutputIt d_first, UnaryOp unary_op);

template <class ExecutionPolicy,
          class ForwardIt1, class ForwardIt2, class UnaryOp>
ForwardIt2 transform(ExecutionPolicy&& policy,
                     ForwardIt1 first1, ForwardIt1 last1,
                     ForwardIt2 d_first, UnaryOp unary_op);

template <class InputIt1, class InputIt2, class OutputIt, class BinaryOp >
OutputIt transform(InputIt1 first1, InputIt1 last1, InputIt2 first2,
                   OutputIt d_first, BinaryOp binary_op);
```


https://en.cppreference.com/cpp/algorithm/transform

Правила Винтерса

1. Человек не должен быть обязан проводить в уме процесс перегрузки.
2. Единый комментарий может описать всё множество.
3. Каждый элемент множества перегрузки делает примерно одно и то же.

- Ниже пример **очень плохого** дизайна.

```
// returns smallest co-prime for n
int least_coprime(int n);

// process coma-separated list of co-primes
// to deduce least possible n
int least_coprime(const std::string& x);
```

Overload set: конструкторы строки

- Хорошее ли это множество перегрузки?

```
basic_string();
basic_string(size_type count, CharT ch);
basic_string(const basic_string& other, size_type pos);
basic_string(const basic_string& other, size_type pos, size_type count);
basic_string(basic_string&& other, size_type pos, size_type count);
basic_string(const CharT* s, size_type count);
basic_string(const CharT* s);
template <class InputIt> basic_string(InputIt first, InputIt last);
basic_string(const basic_string& other);
basic_string(basic_string&& other);
basic_string(std::initializer_list<CharT> ilist);
template <class StringViewLike> explicit basic_string(const StringViewLike& t);
template <class StringViewLike> explicit basic_string(const StringViewLike& t,
                                                      size_type pos, size_type count);
```


basic_string ctors

Проверим правило Винтерса.

```
std::string s('a', 'b');
std::println("s.size() = {}", s.size()); // ?
```

- Вам пришлось провести в уме разрешение перегрузки для ответа на этот вопрос?

https://godbolt.org/z/M7sqxP9oe

Почему перегрузка это сложно?

```
void foo(int); // a
void foo(float); // b

template <typename T>
void bar(T t) { foo(t); };

void foo(double); // c

int main() {
  short s = 1; bar(s); // 1
  double d = 1.0; bar(d); // 2
}
```

Почему перегрузка это сложно?

```
void foo(int);
void foo(float);

template <typename T>
void bar(T t) { foo(t); };

void foo(double);

int main() {
  short s = 1; bar(s); // ok, foo(int) called
  double d = 1.0; bar(d); // compilation error, ambiguity
}
```


https://godbolt.org/z/v6dsYW7bn

Частичный порядок

- Представьте что у вас есть два кандидата.

$F_1(p_1, p_2, \dots, p_n)$

$F_2(q_1, q_2, \dots, q_n)$

- И вызов функции.

$F(t_1, t_2, \dots, t_n)$

- Тогда есть четыре варианта.

$F_1 \leq F_2$ и при этом $F_2 \not\leq F_1$

Тогда мы вызовем F_1.

$F_1 \not\leq F_2$ и при этом $F_2 \leq F_1$

Тогда мы вызовем F_2.

$F_1 \leq F_2$ и при этом $F_2 \leq F_1$

$F_1 \not\leq F_2$ и при этом $F_2 \not\leq F_1$

Это два ошибочных случая.

Частичный порядок для перегрузки

- Для двух **жизнеспособных** кандидатов $F_1(p_1, p_2, \dots, p_n)$ и $F_2(q_1, q_2, \dots, q_n)$ и для вызова $F(t_1, t_2, \dots, t_n)$.
- Не худшим является тот, у которого $\forall i: ICS(t_i \to p_i) \leq ICS(t_i \to q_i)$.

```
void foo(int, int); // viable для foo(x, y)
void foo(int, ...); // viable для foo(x, y)
void foo(int, float, int = 0); // viable для foo(x, y)
struct S { S(int); };
void foo(S, float); // viable для foo(x, y)
```

https://eel.is/c++draft/over.match.best.general

Цепочки неявных преобразований (ICS)

- Для двух жизнеспособных кандидатов $F_1(p_1, p_2, \dots, p_n)$ и $F_2(q_1, q_2, \dots, q_n)$ и для вызова $F(t_1, t_2, \dots, t_n)$
- Если $t_i \to p_i$ содержит только **стандартные преобразования**, а $t_i \to q_i$ содержит не только их, то $ICS(t_i \to p_i) \leq ICS(t_i \to q_i)$
- Если все преобразования стандартные, то $ICS(t_i \to p_i) \leq ICS(t_i \to q_i)$ если
  - $t_i \to p_i$ является подмножеством $t_i \to q_i$.
  - $t_i \to p_i$ имеет **лучший ранг**, чем $t_i \to q_i$

https://eel.is/c++draft/over.ics.rank

Частичный порядок преобразований

- Три ранга стандартных преобразований: exact match, promotion, conversion.
- При этом $E < P < C$

```
void foo(int); // promotion short -> int (wins)
void foo(float); // conversion short -> float

int main() {
  short s = 1; foo(s); // foo(int)
```

- Контрольный вопрос: а что для long?

https://godbolt.org/z/aThjGfGxz

Пользовательское преобразование

```
struct S { S(long); };

void foo(S);

int x = 42;

foo(x); // int -> long -> S
```

```
struct T { T(int); };

struct U { U(double); };

struct S { S(T); S(U); };

void foo(S);
int x = 42;
foo(x); // int -> T -> S

double d = 42.0;
foo(d); // double -> U -> S
```


https://godbolt.org/z/zc47se7zj

До-определяем частичный порядок

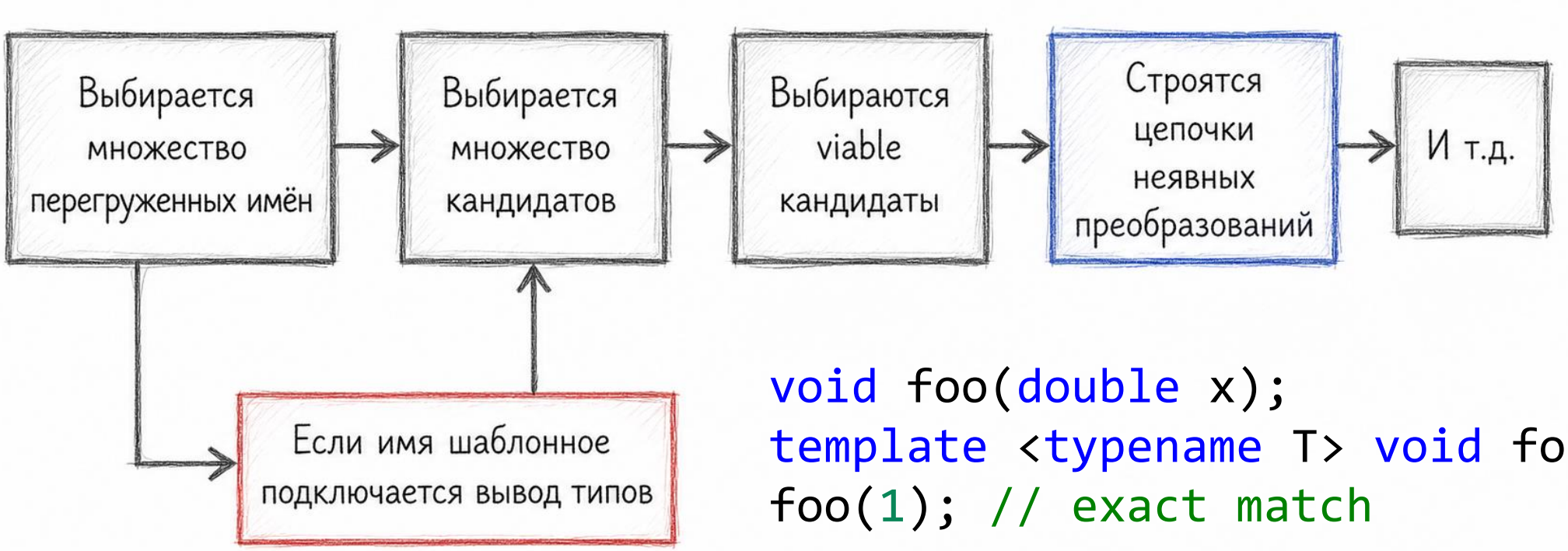
- Две цепочки, в каждом единственное пользовательское преобразование.
- $t_i \to U \to u_i \to p_i$ и $t_i \to V \to v_i \to q_i$
- Сравниваются по хвосту $ICS(u_i \to p_i) \leq ICS(v_i \to q_i)$

```
struct A {
  operator int(); // 1
  operator double(); // 2
};

int i = A{}; // calls (1)
```


https://godbolt.orgz//dz1T1zrMK

Перегрузка требует вывода типов?

- Если в перегрузке участвуют шаблоны функций, то да.




```
void foo(double x);
template <typename T> void foo(T x);
foo(1); // exact match
```

Глубока ли кроличья нора?

- Насколько сложными могут быть отношения частичного порядка?

```
template <typename T> void f(T);  // 1

template <typename T> void f(T*); // 2

int ***a;
f(a); // -> ?
```

- При выводе тип может быть более общим или же более специальным.

Установим частичный порядок

- Сначала трансформируются параметры.

```
template <typename T> void f(T);  // (1) → f(T1)
template <typename T> void f(T*); // (2) → f(T2*)
```

- Далее вывод типов запускается крест накрест.

```
template <typename T> void f(T*);
f(T1); // → FAIL, (1) ≰ (2)

template <typename T> void f(T);
f(T2*); // -> OK, (2) ≤ (1)
```


https://godbolt.org/z/s6TjrM6TG

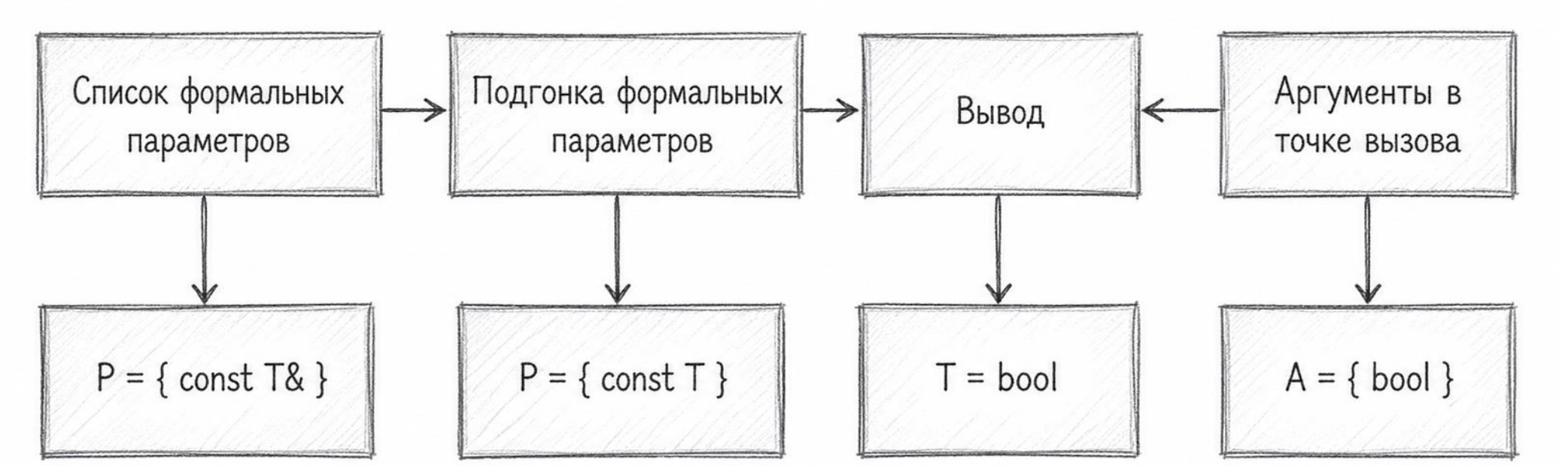
Обсуждение

- Вывод типов запускаемый в процессе перегрузки попарно должен быть довольно простым.
- Его можно сформулировать как своего рода игру.

Игра в вывод типов

- Возможные преобразования включают [temp.deduct.call] снятие или добавление константности, преобразование типа к базовому и прочее

```
template <typename T> void f(const T& t)          f(false)
```



https://eel.is/c++draft/temp.deduct#call

Одношаговая унификация (дедукция)

- Если задан прототип функции с параметрами и конкретный вызов с аргументами.

```
template <typename T1, ..., Tn>
F(T1 P1, ..., Tn Pn);

F(A1, ..., An)
```

- Строится система $P_i \preccurlyeq A_i$ где $\preccurlyeq$ это направленная одношаговая дедукция, подчиняющаяся простым и очевидным правилам.
- Особую непередаваемую **простоту** ей придаёт наличие в языке const, volatile, указателей и двух видов ссылок.

Пример

```
template<typename T> void f(std::vector<T> const&);

std::vector<int> v; f(v);
```

- Вывод происходит примерно так:

```
std::vector<T> const& ≼ std::vector<int>&
std::vector<T> ≼ std::vector<int>
T ≼ int
T = int
```

CTAD и помощь в дедукции

- Конструкторы классов выводят типы.

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

container c(7); // → container<int>

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

Немного безумия

```
template <class T> struct A {
  using value_type = T;
  A(value_type) {}
};

template <class T> A(A<T>) -> A<A<T>>;

A a(42); // -> A<int>

A b = a; // -> A<A<int>>

A c = b; // -> A<A<A<int>>>
```


https://godbolt.org/z/s5P91MrYq

Лямбда это тоже класс

```
auto inc = [](std::integral auto x) -> decltype(x) {
  return x++;
};

decltype(inc) inc_again; // ok
```

- Unevaluated контекст её вычисления не составляет проблем.

```
template <typename T>
decltype([](auto x) -> T { return (T)x; }) c_cast; // still ok

double d = 1.5;

auto x = c_cast<int>(d); // -> 1
```

Вывод где угодно?

```
template <auto f = []{}> struct S {};

S x, y;

static_assert(std::is_same_v<decltype(x), decltype(y)>);
```

- **GCC**:

```
<source>:6:20: error: static assertion failed
    6 | static_assert(std::is_same_v<decltype(x), decltype(y)>);
      |               ~~~~~^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

- **Clang**:

```
<source>:4:1: error: template arguments deduced as 'S<(lambda at
<source>:3:20){}>' in declaration of 'x' and deduced as 'S<(lambda at
<source>:3:20){}>' in declaration of 'y'
    4 | S x, y;
      | ^ ~  ~
```


https://godbolt.org/z/4e3hx31Gs

Важное наблюдение

- Пример можно починить, просто убрав вывод типов.

```
template <auto f = []{}> struct S {};

S<> x, y;

static_assert(std::is_same_v<decltype(x), decltype(y)>); // OK
```

- Также можно починить, оставив его раздельным.

```
template <auto f = []{}> struct S {};
S x; S y;
static_assert(!std::is_same_v<decltype(x), decltype(y)>); // OK
```

Анализ с точки зрения стандарта

- Для S будет построен deduction guide

```
template <auto f = []{}> S() -> S<f>;
```

- Далее auto выведется из invented declaration.

```
T tmp = []{};
```

- Таким образом получаем:

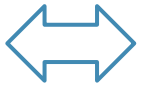
```
typeof([]{} use #1) = L1
decltype(x) = S<L1{}>
```

```
typeof([]{} use #2) = L2
decltype(y) = S<L2{}>
```

Так что я на стороне clang

```
template <auto f = []{}> struct S {};
S x, y; // ill-formed
static_assert(std::is_same_v<decltype(x), decltype(y)>);
```

```
typeof([]{} use #1) = L1
decltype(x) = S<L1{}>
```



```
typeof([]{} use #2) = L2
decltype(y) = S<L2{}>
```

The placeholder is replaced by the return type of the function selected by overload resolution for class template deduction ([over.match.class.deduct]). If the *decl-specifier-seq* is followed by an *init-declarator-list* or *member-declarator-list* containing more than one *declarator*, the type that replaces the placeholder shall be the same in each deduction.

https://eel.is/c%2B%2Bdraft/dcl.type.class.deduct#1

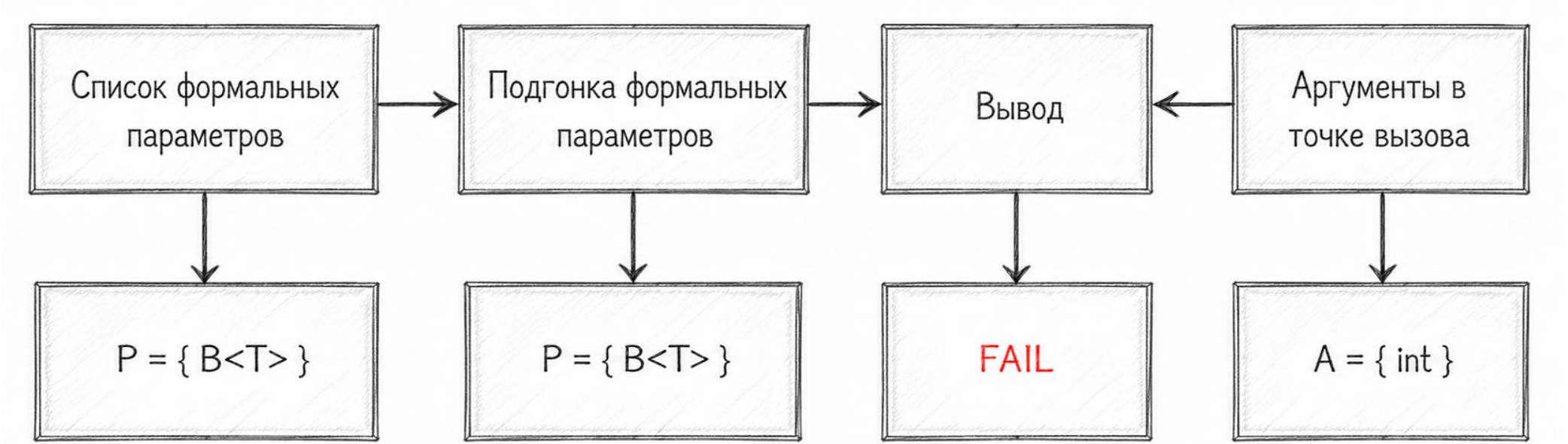
Одношаговая, кому говорят

```
template <typename T> struct B { B(T) {} };
template <typename T> int buz(B<T>) { return 0; }
// ...
buz(1);
```

Игра в вывод типов: мы проиграли

- Вывод типов может сделать прямую инициализацию списком инициализации или инициализацию простого агрегата, но не более того.

```
template <typename T> int f(B<T>)                    f(1)
```



https://godbolt.org/z/sMfjPb

У меня есть мечта...

```
template <int> struct A {};

struct X { operator A<0>(); operator A<1>(); };

template <int...> struct good : std::false_type {};

template <>
struct good<0, 1, 1, 0, 1, 0, 1, 1> : std::true_type {};

template <int... Ts> requires good<Ts...>::value
int hard(A<Ts>...);

hard(X{}, X{}, X{}, X{}, X{}, X{}, X{}, X{});
```


https://godbolt.org/z/KPqq9nKcM

Ограничение: non-deduced context

- В некоторых случаях мы попадаем в non-deduced context

```
template<typename T> void f(typename T::value_type);
f(x);
```

- Здесь итоговое уравнение не считается уравнением для T.

```
typename T::value_type ≼ type(x)
```

- Точно также non-deduced создаётся через decltype или sizeof.

```
template<typename T> void foo(decltype(std::declval<T>()));
int x; foo(x); // FAIL
```

Увы, NTTP это non-deduced контекст

- Подстановка ок.

```
template<typename T, int N> void buz(T const(&)[N]);

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

- А вот вывод будет провален

```
template<int N> void foo(int (&)[N + 1]);

int main() {
  int arr[10];
  foo(arr); // FAIL
```


https://godbolt.org/z/GTb59ba1T

Иногда вывод типов требует перегрузки?

```
template <class T> void foo(T (*p)(T)) {
  std::cout << p(0) << std::endl;
}

int bar(char) { return 0; } // 1
int bar(int) { return 1; } // 2

foo(bar); // -> ?
```

Самый обидный из запретов

- Перегрузка для вывода типов не может сама требовать вывода типов

```
template <class T> void foo(T (*p)(T)) {
  std::cout << p(0) << std::endl;
}

template <typename T = char> int bar(T) { return 0; } // 1

int bar(int) { return 1; } // 2

foo(bar); // -> FAIL
```

- Это не позволяет интересную рекурсию

У меня есть мечта…

- **Язык C++** (русское С, читается как Эс-Плюс-Плюс).
- Семантически неограниченный C++.
  - Рекурсия на концептах конечно же.
  - Сколько угодно user-conversions в ICS.
  - Неявные цепочки преобразований в выводе типов.
  - Никакого non-deduced вообще.
  - Мы сегодня ещё в него добавим фич…
- Для простых программ он будет вести себя в точности как C++. Для сложных… ну он будет ничем не ограничен.

Добавим огня

```
int foo(int, char) { return 0; }
int foo(double, long) { return 1; }

template <typename T, typename U>
int bar(T t, int (*)(T, U)) { return foo(t, 0); }

bar(42, &foo); // gcc OK, clang FAIL
```

```
<source>:12:3: error: no matching function for call to 'bar'
   12 |   bar(42, &foo); // gcc OK, clang FAIL
      |   ^~~
<source>:5:5: note: candidate template ignored: couldn't infer template argument 'U'
    5 | int bar(T t, int (*)(T, U)) { return foo(t, 0); }
      |     ^
```

Немного о схоластических аргументах

```
int foo(int, char) { return 0; }
int foo(double, long) { return 1; }

template <typename T, typename U>
int bar(T t, int (*)(T, U)) { return foo(t, 0); }

template <typename T, typename U>
int buz(int (*)(T, U), T t) { return foo(0, t); }

bar(42, &foo); // gcc OK, clang FAIL
buz(&foo, 42); // gcc FAIL, clang FAIL
```


https://gcc.gnu.org/bugzilla/show_bug.cgi?id=124357

Снова non-deduced context

```
template <typename T>
int f(T (*p)(T));
int g(int);
int g(char);
int i = f(g); // OK
```

```
template <typename T>
int f(T (*p)(T));
int g(int);
char g(char);
int i = f(g); // FAIL
```

```
template <typename T>
int f(T, T (*p)(T));
int g(int);
char g(char);
int i = f(1, g); // OK
```

6 When P is a function type, function pointer type, or pointer-to-member-function type:

(6.1) — If the argument is an overload set containing one or more function templates, the parameter is treated as a non-deduced context.

(6.2) — If the argument is an overload set (not containing function templates), trial argument deduction is attempted using each of the members of the set whose associated constraints ([temp.constr.constr]) are satisfied. If all successful deductions yield the same deduced A, that deduced A is the result of deduction; otherwise, the parameter is treated as a non-deduced context.

https://eel.is/c++draft/temp.deduct.call

Немного о компиляторной демократии

Drea Pinski 2026-03-07 05:25:18 UTC

clang and EDG rejects both examples.
But MSVC follows GCC here and accepts the first and rejects the second.

so it is 2 vs 2. Maybe there is a defect report here.

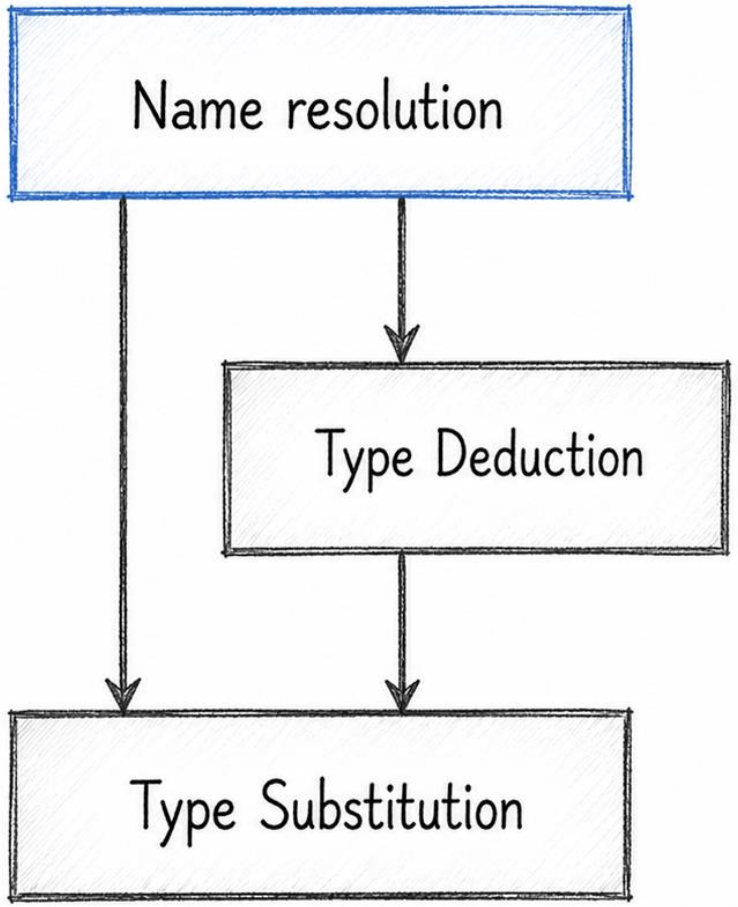
https://gcc.gnu.org/bugzilla/show_bug.cgi?id=124357

Текущее состояние C++

- C++ является подмножеством некоего куда более консистентного языка.
- Из которого убрали всё, что делало компиляцию бесконечной.
- После чего перечнем частных случаев добавили всё без чего было бы слишком сложно жить.
- И сделали это не столько люди, сколько ветер.

Общий вывод

- Самые сложные семантические процессы это те, которые используют в той или иной форме отношения частичного порядка.
  - Выбор более подходящего кандидата.
  - Определение более специального типа.
  - Определение более строгого ограничения.
- Во всех этих случаях язык C++ между решением отнять что-то у программиста и решением убрать у чего-то математическую красоту, выбирает второе.
- За это мы его собственно и.



Рекомендуемые вопросы

- Как работает унификация Хиндли-Милнера? Я слышал, что унификация Хиндли-Милнера также не очень дружит с мутабельными переменными. Интересны детали.
- Расскажите про странные аспекты рекурсивного вывода типов и есть ли у компиляторов глаза?
- Я слышал CTAD немного критикуют. Расскажите за что?
- Расскажите про взаимодействие с концептами?
- Расскажите подробнее про стандартные преобразования при перегрузке.
- Какие правила у процесса одношаговой дедукции и как там ссылки добавляют огня?
- Расскажите больше про частичный порядок шаблонов функций.

УНИФИКАЦИЯ

Как работает унификация Хиндли-Милнера?

Процедура унификации

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

- Вводим переменные.
  - Пусть A это `type(map)`
  - B это `type(f)`
  - C это `type(x)` и так далее
- Для операторов важно где именно они появляются.

```
type(rhs :) это Y -> [Y] -> [Y]
type(lhs :) это Z -> [Z] -> [Z]
```

Процедура унификации

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

- Вводим переменные.

```
type(map)    = A
type(f)      = B
type(x)      = C
type(xs)     = D
type(lhs :)  = Z -> [Z] -> [Z]
type(rhs :)  = Y -> [Y] -> [Y]
type(lhs []) = [X]
type(rhs []) = [R]
```

- Составляем систему уравнений

```
D = [Z]
C = Z
B = C -> Y
A = B -> D -> E
E = [Y]
A = B -> [X] -> [R]
X = Z
R = Y
```

Процедура унификации

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

- Решаем систему уравнений

```
D = [Z]
C = Z
B = C -> Y
A = B -> D -> E
E = [Y]
A = B -> [X] -> [R]
X = Z
R = Y
```

- Получаем решение относительно Y и Z.

```
D = [Z]
C = Z
B = Z -> Y
X = Z
R = Y
E = [Y]
A = (Z -> Y) -> [Z] -> [Y]
```


https://en.wikipedia.org/wiki/Type_inference

Проблемы с мутабельностью

- Если мы рассмотрим нечто вроде:

```
let r = ref [] in
  r := [1];
  cons(true, r)
```

- То для классического Хиндли-Милнера principal type получится очень общий.

```
r = Ref (List a)
```

- Тут должна быть ошибка если `Bool` $\nsim$ `Int`
- Либо `Ref (List Int)` если `Bool` $\leq$ `Int`

HAS BEEN SEEN

Есть ли у компиляторов глаза?

Strange Aspects of Recursive Deduction

```
auto sum(int i) {
  if (i == 1)
    return i;
  else
    return sum(i - 1) + i;
}
```

https://godbolt.org/z/TEs5oeoPT

```
auto sum(int i) {
  if (i == 1)
    return sum(i - 1) + i;
  else
    return i;
}
```

https://godbolt.org/z/q6rnW39qs

Strange Formulations in the Standard

```
auto sum(int i) {
  if (i == 1)
    return i;
  else
    return sum(i - 1) + i;
}
```

https://godbolt.org/z/TEs5oeoPT

Once a non-discarded return statement **has been seen** in a function, however, the return type deduced from that statement **can be used** in the rest of the function, including in other return statements. [dcl.spec.auto.general]

Do Compilers Have Eyes?

- In the C++23 standard, the expression "has been seen" appears only three times.

1. Once a non-discarded return statement **has been seen** in a function, however, the return type deduced from that statement **can be used** in the rest of the function, including in other return statements. [dcl.spec.auto.general]
2. A class is considered defined after the closing brace of its class-specifier **has been seen** even though its member functions are in general not yet defined. [class.pre]
3. Similarly, the use of A::B as a base-specifier is well-formed because D is derived from A, so checking of base-specifiers must be deferred until the entire base-specifier-list **has been seen**. [class.access.general]

- Its meaning is vague in all these cases.

CTAD

И его критика

CTAD

- Конструкторы могут быть использованы для вывода типов.

```
template<typename T> struct container {
  container(T t);
  // and so on
};

container c(7); // → container<int>

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

auto c = new container(7); // → container<int>
```

Implicit Guides

- Если конструктора нет, нужно писать явный хинт.

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

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

- Если конструктор есть, компилятор генерирует неявный хинт.

```
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
```

Critique of Deduction Guides

```
template <typename It> bool is_palindrome(It first, It last) {
  return std::equal(first,
                    last,
                    std::reverse_iterator(last),
                    std::reverse_iterator(first));
}
```

- Available constructors:

```
reverse_iterator(iterator_type x);

template <typename U>
reverse_iterator(const reverse_iterator<U>& other);
```


wctad-maybe-unsupported

КОНЦЕПТЫ

Взаимодействие с ограничениями.

Ограничения влияют на перегрузку

```
template <typename T>
requires (sizeof(T) > 4)
void foo(T x) { /* do something with x */ }

template <typename T>
requires (sizeof(T) <= 4)
void foo(T x) { /* do something else with x */ }

foo('c'); // OK for sure
```

Вывод типов не смотрит на ограничения

- В принципе вывод типов мог бы использовать информацию от ограничений.

```
double foo();
int foo(int = 0);
std::integral auto t = foo(); // ambiguity
```

- Но они не делают этого. Чтобы понять почему, представим себе:

```
C auto x = f(a(), b(), c(), d());
```

- Допустим тут у каждой функции по пять перегрузок…

СТАНДАРТНЫЕ ПРЕОБРАЗОВАНИЯ

Детали стандартных и пользовательских преобразований.

Частичный порядок преобразований

- Ранг цепочки стандартных преобразований $t_i \to p_i$ равен наихудшему рангу входящего туда преобразования.
- Три ранга стандартных преобразований: exact match, promotion, conversion.
- При этом $\boldsymbol{E} < P < C$

```
int x = 5, y; y = x + 2; // lvalue to rvalue
int a[10], *b; b = a; // array to pointer
int f(int); int (*pf)(int); pf = f; // function to pointer

int v; const int *pv = &v; // qualification conversion
int (*pfn)(int) noexcept; pf = pfn; // function pointer
```

Частичный порядок преобразований

- Ранг цепочки стандартных преобразований $t_i \rightarrow p_i$ равен наихудшему рангу входящего туда преобразования.
- Три ранга стандартных преобразований: exact match, promotion, conversion.
- При этом $E < \boldsymbol{P} < C$

```
int i = true + 1; // integral promotion bool -> int
double d = 2.0 + 1.0f; // fp promotion float -> double
```

Частичный порядок преобразований

- Ранг цепочки стандартных преобразований $t_i \rightarrow p_i$ равен наихудшему рангу входящего туда преобразования.
- Три ранга стандартных преобразований: exact match, promotion, conversion.
- При этом $E < P < \boldsymbol{C}$

```
float fp = d; // fp conversion
short s = i; // integral conversion
float g = 1; // int to float conversion
int *pi = nullptr; // pointer conversion
```

Пользовательские преобразования

```
struct Foo {};
struct Bar : Foo {};
struct Baz : Foo {};

struct X {
  operator Bar();
  operator Baz() const;
};

void foo(const Foo &f);

void bar() {
  foo(X{});
}
```

- У нас две цепочки ICS.

`X{} → Bar → const Foo&`

`X{} → Baz → const Foo&`

- Они выглядят одинаковыми и мы видим здесь неоднозначность.
- Но оба мейнстримных компилятора почему-то предпочитают Bar.

overload-resolution-with-const-qualification

ДЕДУКЦИЯ

Правила одношаговой унификации в C++ и при чём тут ссылки.

Правила унификации

Правила (неформально дедуцированы мной из стандарта).

$t \preccurlyeq T \vdash t = T$

$T\& \preccurlyeq U\& \vdash T \preccurlyeq U$

$T\&\& \preccurlyeq U\&\& \vdash T \preccurlyeq U$ (с оговоркой на forwarding)

$cv\ T \preccurlyeq cv\ U \vdash T \preccurlyeq U$

$C\langle P_1, \dots, P_n\rangle \preccurlyeq C\langle A_1, \dots, A_n\rangle \vdash P_1 \preccurlyeq A_1, \dots, P_n \preccurlyeq A_n$

https://eel.is/c++draft/temp.deduct

Вывод чего угодно?

- Те же соображения работают для вывода довольно сложных типов.

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

Главный хак вывода типов

- В случае специальной формы T&&, вывод зависит от value-category.

```
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&&)
```

A *forwarding reference* is an rvalue reference to a cv-unqualified template parameter that does not represent a template parameter of a class template (during class template argument deduction ([over.match.class.deduct])). If P is a forwarding reference and the argument is an lvalue, the type "lvalue reference to A" is used in place of A for type deduction.

https://eel.is/c++draft/temp.deduct.call#3

Ограничение: вообще не вывод типов

- Свёртка ссылок происходит только на выводе, не на подстановке типов.

```
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
```

- Не стоит это путать с non-deduced контекстом.

[1] Template argument deduction is done by comparing each function template parameter type (call it P) that contains template parameters that participate in template argument deduction with the type of the corresponding argument of

https://eel.is/c++draft/temp.deduct.call

КРЕСТ-НАКРЕСТ

Частичный порядок шаблонов при перегрузке

Трансформация параметров

- [temp.func.order/3] for each type, non-type, or template template parameter **synthesize** a unique type, value, or class template respectively and **substitute** it for each occurrence of that parameter in the function type of the template

```
template <typename T> void f(T); // → f(T1)
template <typename T> void f(T*); // → f(T2*)
```

- Далее на эту пару трансформированных типов запускается вывод.
- [temp.deduct.partial/2] The deduction process uses the **transformed type** as the argument template and the **original type** of the other template as the parameter template.

Частичный порядок шаблонов функций

- В случае шаблонов функций, они частично упорядочены на более специализированные и менее специализированные.
- [temp.func.order] Partial ordering selects which of two function templates is **more specialized than the other** by **transforming** each template [...] and performing **template argument deduction** using the function type
- Более специализированная выиграет перегрузку, но для определения этого мы должны:
  - Трансформировать параметры.
  - **Запустить вывод типов.**

Двойной вывод

- Он происходит несколько крест-накрест.

#template	Parameter type	Argument type
$template_1$	`template <typename T> void f(T)`	`f(`T_1`)`
$template_2$	`template <typename T> void f(T*)`	`f(`T_2`*)`

[temp.deduct.partial/2] This process is done twice for each type involved in the partial ordering comparison:

- once using the transformed $template_1$ as the argument template and $template_2$ as the parameter template
- and again using the transformed $template_2$ as the argument template and $template_1$ as the parameter template

Пример с конфликтом

- Иногда (2) < (1) но при этом и (1) < (2)

```
template <typename T> void f (T, T);                    // 1
template <typename T1 typename T2> void f (T1*, T2*); // 2
```

- Явного ордеринга между ними нет

```
double t, s;
f (t, &s); // → подходит к обоим и значит конфликт
```

- Это очень важно. Легко найти что-то, что конфликта не даст

```
f (t, t); // → тривиально вызывает (1)
```

- Но если явного ордеринга нет, значит **есть что-то** для чего есть конфликт

https://godbolt.org/z/v54bzx

Результирующее отношение

- Мы видим, что нам нужно рассмотреть два случая

```
// 1. compare (1) not less specialized then (2)
template <typename T> void f(T);
T2 *a; f(a); // OK, (1) ≥ (2)
// 2. compare (2) not less specialized then (1)
template <typename T> void f(T*);
T1 b; f(b); // FAIL, (2) < (1)
```

- Откуда очевидно (1) > (2)

Ещё более просветляющий пример

- Рассмотрим перегрузку

```
template <typename T> void foo(int n); // 1
template <typename T> void foo(T t); // 2
```

- Очевидно для синтезированного типа T_1:

```
foo(T1); // не подходит для вызова <T>foo(int)
foo(int); // подходит для вызова <T>foo(T)
```

Далее смотрим для

```
foo<int>(42); // → 1
```

Оно подходит к обоим, выигрывает более специальная

https://godbolt.org/z/ce3hjP

Ещё более просветляющий пример

- Рассмотрим перегрузку

```
template <typename T> void foo(int n); // 1
template <typename T> void foo(T t); // 2
```

- Очевидно для синтезированного типа T_1:

```
foo(T1); // не подходит для вызова <T>foo(int)
foo(int); // подходит для вызова <T>foo(T)
```

Далее смотрим для

```
foo<int>(42); // → 1
```

Оно подходит к обоим, выигрывает более специальная

https://godbolt.org/z/ce3hjP