О ДЕНОТАЦИИ

Разрешение имён и самое недооценённое изменение в C++23

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

Денотанты

- В естественном языке мы часто прибегаем к обозначениям вещей вместо вещей.
- Например некто спрашивает "Использует ли создатель языка C++ наследование от std::vector?"
- Поскольку создателем C++ является Бьёрн Страуструп, то этот вопрос можно понять как вопрос "Использует ли Бьёрн Страуструп наследование от std::vector?" и в точности на него ответить.
- Мы говорим, что обозначение "создатель языка C++" имеет обозначаемым или денотантом Бьёрна Страуструпа.

Проблема денотации в философии

- Допустим некто спрашивает: "является ли Скотт Майерс автором книги Effective C++?".
- Что он хочет спросить?
- Очевидно у предиката "автор книги Effective C++" денотантом является Скотт Майерс.
- Не хочет ли человек спросить является ли Скотт Майерс сам собой?

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

Проблема денотации в языке C++

```
namespace B { int x = 0; }
namespace C { int x = 0; }

namespace A {
  using namespace B;
  void f() {
    using C::x;
    A::x = 1; // ???
    x = 2;    // ???
  }
}
```

- Какой объект является денотантом для `x` и какой для `A::x`?

- Если мы спрашиваем совпадает ли `A::x` с `B::x` или с `C::x`, о чём мы спрашиваем?
- Может быть нас интересует является ли `A::x` собой или нет?
- Кажется нас интересует развитие некоего процесса в языке…

Квалифицированные имена

```
namespace B { int x = 0; }
namespace C { int x = 0; }

namespace A {
  using namespace B;
  void f() {
    using C::x;
    A::x = 1; // sets B::x
    x = 2;    // sets C::x
  }
}
```

- A::x это квалифицированное имя. Его поиск это поиск по квалифицирующему пространству имён.
- x это не квалифицированное имя. Его поиск это поиск изнутри наружу.

https://godbolt.org/z/19sbxaGWz

Такие разные имена

- `x, x1, MyArr, T, f, N, A`
- `MyArr<T, 1>, f<N::A>(N::A());`
- `~Foo, operator "" _y("f")`
- `::std::cout, Foo::~Foo`

- `this->f`
- `MyArr<T, 1>::PFoo->~A<char>();`

- **identifier**
- **template-id**
- **unqualified-id**
- **qualified-id**
- **nested-name-specifier**
- **terminal name**
- **qualified name**
- **unqualified name**

Квалификация до, а не после

```
namespace A {
  struct std { struct Cout {}; static Cout cout; };

  void operator << (std::Cout, const char *) {
    ::std::cout << "World\n";
  }
}

int main() {
  using A::std;
  ::std::cout << "Hello, "; // qualified std → namespace
  std::cout << "Hello";     // unqualified std → struct
}
```


https://godbolt.org/z/9Ghrc8PsM

Эволюция языка

- Инстанцирование шаблонов тоже требует поиска имён.

```
namespace N {
  template <typename T> struct Y {};
}

using N::Y;

template class Y<int>; // GCC OK, clang error

template class N::Y<int>; // both OK
```

- Можем ли мы сделать такого рода явное инстанцирование?

https://godbolt.org/z/nGezK4PnE

Эволюция языка

- C++11 [temp.expl.spec] [...] An explicit specialization whose declarator-id is not qualified shall be declared in the nearest enclosing namespace of the template
- C++17 [temp.expl.spec] An explicit specialization may be declared in any scope in which the corresponding primary template may be defined

```
<source>:9:16: error: explicit instantiation of 'N::Y' must occur in namespace 'N'
    9 | template class Y<int>;
      |                ^
<source>:2:33: note: explicit instantiation refers here
    2 |   template <typename  T> struct Y {
      |                                 ^
```

Более забавный пример

```
namespace N {
  template <typename T> struct Y;
}
using N::Y;
template <typename T> struct Y {};
```

- Вроде бы опять gcc ok, clang error, но есть нюанс.

```
<source>:6:23: error: declaration conflicts with target of using declaration
already in scope
    6 | template <typename T> struct Y {};
```


https://godbolt.org/z/vn6E3b3xs

Новости в C++20: модули

```
// TU1
export module A;
struct X {};
export using Y = X;
// TU2
import A;
Y y; // ???
X x; // ???
```

- Модули вводят в язык разделение на visible (видимо для поиска имён) и reachable (достижимо).
- Грубо говоря, даже если тип не **виден для поиска имён**, но при этом достижим, его можно инстанцировать.
- Модули существенно усложняют ситуацию, делая неверным умолчание о том, что каждый поиск ограничен своей единицей трансляции.

Обсуждение

- Я попросил моих подписчиков в телеграме выбрать самое существенное нововведение в C++23.

19% Explicit object parameter (deducing this)

10% if consteval

24% import std

4% Метки и goto в constexpr функциях

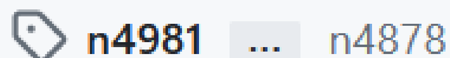
12% Многомерные индексаторы

9% Статические операторы [] и ()

3% Чисто бюрократические уточнения в стандарт относительно некоторых семантических процессов

Существенное по какому признаку?

```
Additional, larger editorial changes appear in subsequent commits.
```



https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p1787r6.html

https://github.com/cplusplus/draft/pull/4398

Немного философии

- Яркая точка в небе, которая последней исчезает утром издавна называлась "утренняя звезда".
- Аналогично "вечерней звездой" называлась яркая точка которая первой появлялась вечером.
- Чтобы установить общий денотант этих двух выражений (планету Венера) человечеству понадобилось:
  - развитие астрономии
  - мощная наблюдательная техника
- Что нужно в искуственном языке программисту знать, чтобы установить связь между обозначающим и обозначаемым?

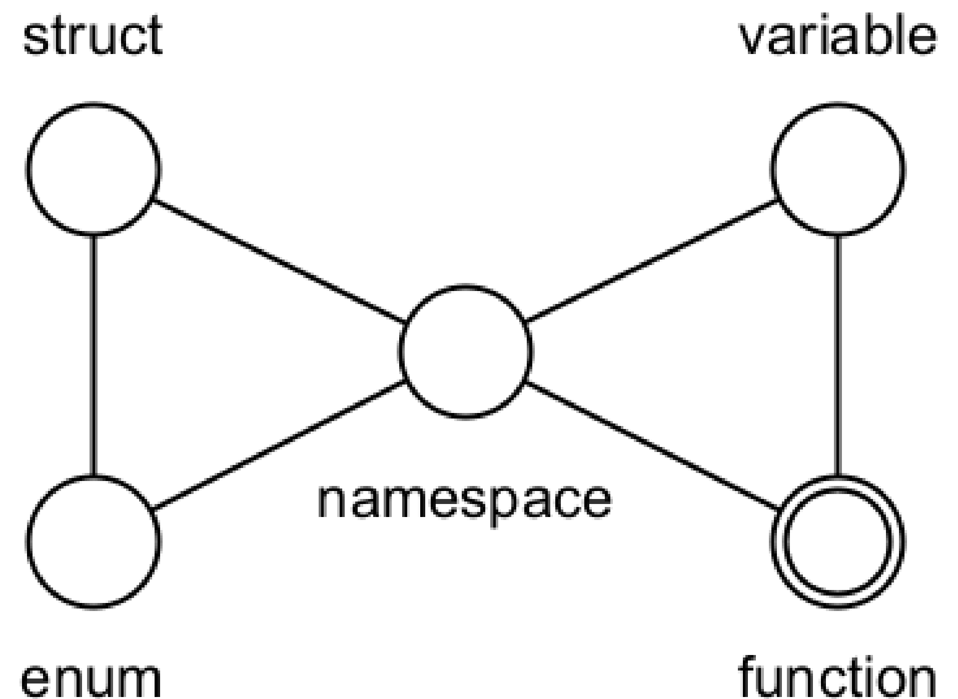
Чеклист практикующего программиста

- Какие имена и в каких областях видимости мы ищем.
- В какой точке начинается поиск имени.
- По каким правилам он происходит.
- Где поиск имени заканчивается.
- Что случается до и после поиска имён.
- И скорее всего вы интуитивно всё это и так **почти** знаете. Как я много раз повторяю, язык C++ довольно прост в интуитивном освоении.

Граф совместимости имён

```
int v0;
int v0(); // FAIL
struct v0{}; // OK
```

- Совместно могут существовать переменная или функция вместе со структурой или перечислением с тем же именем.
- Только функции и шаблоны функций могут сосуществовать с самими собой.



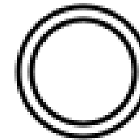
https://godbolt.org/z/TazxaWoco

Граф ошибок сокрытия имён

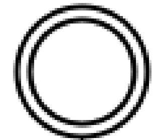
```
int v0;
void foo() {
  double v0; // OK
  int v0(); // FAIL
  struct v0{}; // OK
  namespace v0 = N; // OK
```

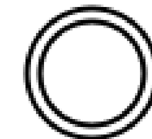
- Переопределения могут быть в куда более широких пределах.

struct

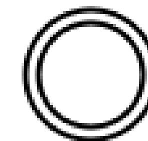


variable

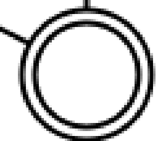




namespace



enum



function

https://godbolt.org/z/TazxaWoco

Поиск в областях видимости (scopes)

- В примере, приведённом ниже, иллюстрируется зависимость поиска имён от области видимости.
- Какой тут интуитивно ответ?

```
struct A { struct T { int x = 1; }; };

struct S { int x = 2; };

template <typename T> struct B : A {
  int f() { T t; return t.x; }
};

B<S> B; std::cout << B.f() << std::endl; // 1 or 2?
```

Scope dependency

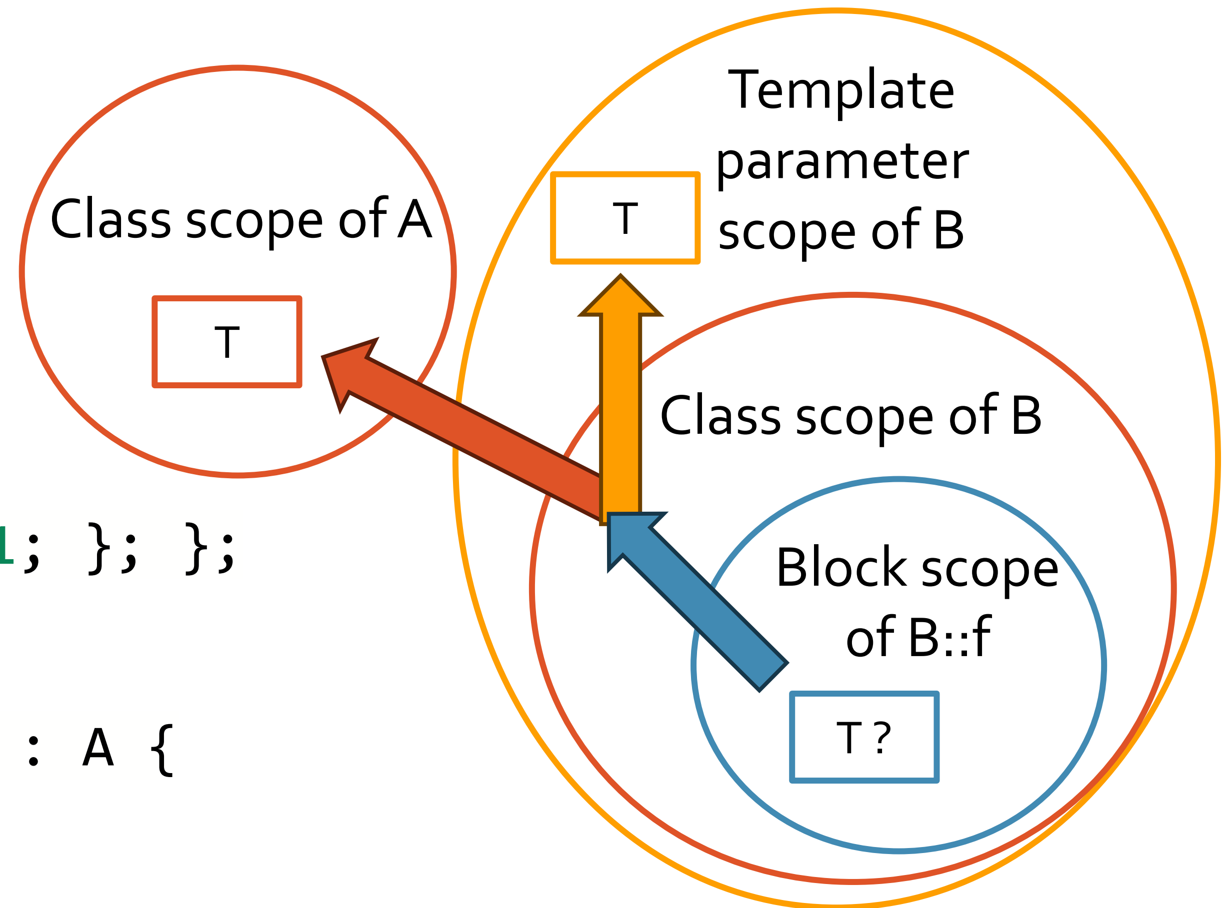
- В примере, приведённом ниже, иллюстрируется зависимость поиска имён от области видимости.
- Какой тут интуитивно ответ?

```
struct A { struct T { int x = 1; }; };

struct S { int x = 2; };

template <typename T> struct B : A {
  int f() { T t; return t.x; }
};

B<S> B; std::cout << B.f() << std::endl; // 1 or 2?
```



https://godbolt.org/z/PT1z7hGxv

Разные scope очень отличаются

```
namespace A { int i; }

namespace A1 {
  using A::i;
  using A::i; // OK: double declaration
}

void f() {
  using A::i;
  using A::i; // error: double declaration
}
```

namespace scope

template parameters scope

enum scope

class scope

function parameters scope

block scope

lambda scope

https://godbolt.org/z/s48vvE4rK

Ещё раз про семантические процессы

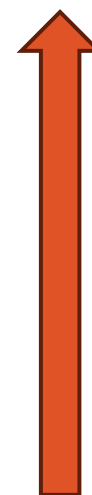
- Компилятор на самом деле вычисляет вещи сверху-вниз.
- Человек думает о процессе, запускающемся из точки снизу-вверх.

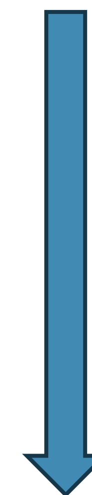
```
struct S { int x = 2; };

struct A { struct T { int x = 1; }; };

template <typename T> struct B : A {
  int f() { T t; return t.x; }
};

B<S> Y; std::cout << Y.f() << std::endl;
```





S, S::x

A, A::T, A::T::x

<T>, B

resolve T to A::T

instantiate B<S>

Где начинается поиск имён

- Независимые имена связываются в точке появления.

```
int foo() { return 1; }

template <typename T> int bar() {
  return foo();
};

// Если перенести foo сюда то будет ошибка

int main() {
  auto t = bar<int>(); // t == 1
```

lookup of foo started

instantiation of bar here

https://godbolt.org/z/o453r5see

Где начинается поиск имён

- Зависимые имена связываются в точке инстанцирования.

```
template <typename T> struct B { int foo() { return 1; } };

template <typename T> struct S : B<T> {
  int bar() { return this->foo(); };
};

template <> struct B<int> { int foo() { return 2; } };

int main() {
  auto t = S<int>{}.bar(); // t == 2
```

← instantiation of bar here

← lookup of foo started

https://godbolt.org/z/67WYaz4Wx

Три разных случая

- Поиск чисто независимого имени.
- Поиск чисто зависимого имени.
- Поиск независимого имени, похожего на вызов функции с зависимыми параметрами.

```
template <typename T> int bar(T t) {
  return foo(t);
```

- Только в этом случае поиск имён действительно **двухфазный** но об этом позже.
- И он стартует с обоих точек.

Общие правила для поиска

- В любой области видимости предпочтителен **простой поиск**, являющийся общей подпрограммой
- Если область видимости это класс или шаблон, то там производится **базовый поиск**.
- Если имя не квалифицированное проводится **неквалифицированный поиск**.
- После неквалифицированного поиска при необходимости он повторяется как **аргументно-зависимый** (ADL).
- Если имя квалифицированное проводится **квалифицированный поиск** по вложенным областям видимости.

simple search

search

unqualified

ADL

qualified

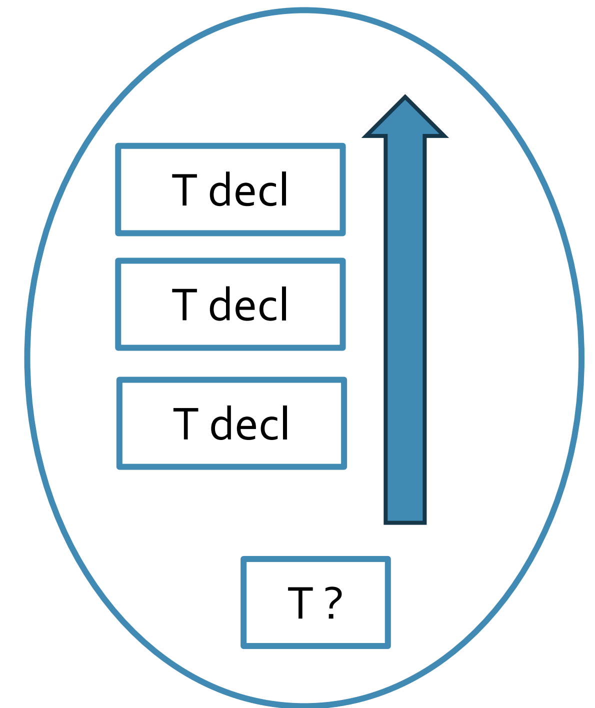
Простой поиск

- Простой поиск (single search) в области видимости S для имени N, введённое в точке P находит все определения N в этой области.

```
namespace N {
  int X = 1;
};

int main() {
  int N = 2;
  N += N::X; // OK
```

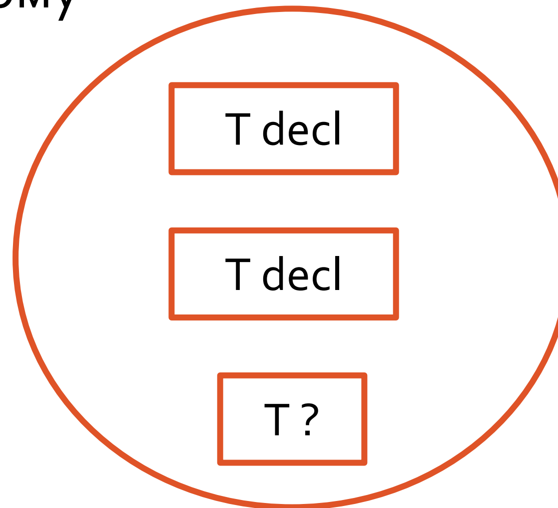
- Даже простой поиск не так прост т.к. ограничивает разные имена и сущности.

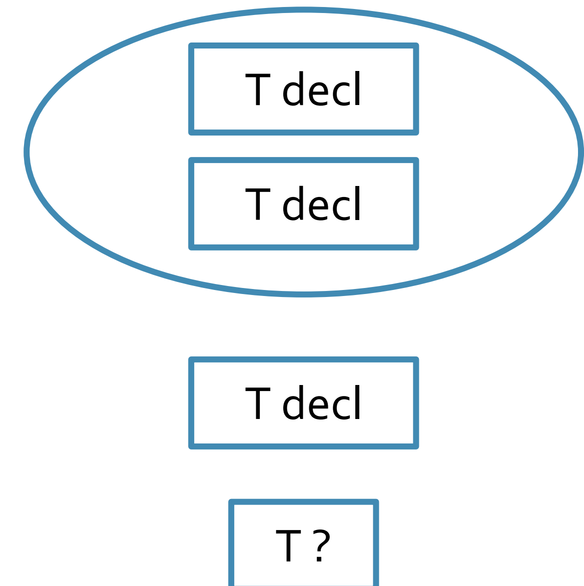


https://godbolt.org/z/9Pbsn74xq

Базовый поиск

- Базовый поиск (search) делается в классах или в шаблонах классов.
- Для имени N строится множество S, состоящее из множества определений и множества подобъектов.
- Оно строится немного по разному в зависимости от того, само имя ищется внутри класса или снаружи.
- В основном это простой поиск в области видимости класса.





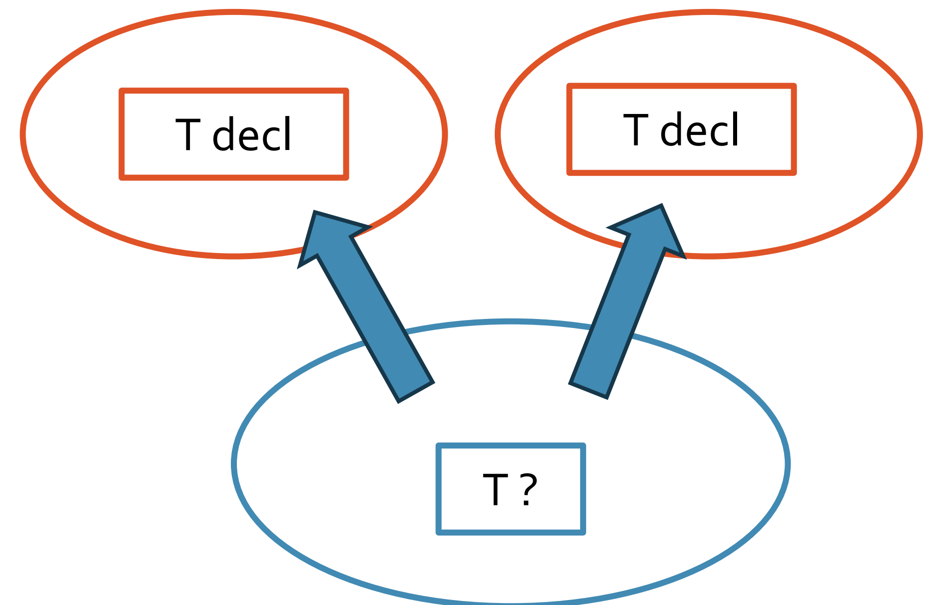
Поиск в базовых классах

- Базовый поиск (search) делается в классах или в шаблонах классов.
- Множество подобъектов объединяет (возможно виртуальные) базы класса.

```
struct A { int x; }; struct D : A {};
struct B { double x; };
struct C: public A, public B { };
// S = { invalid, A in C, B in C }
```

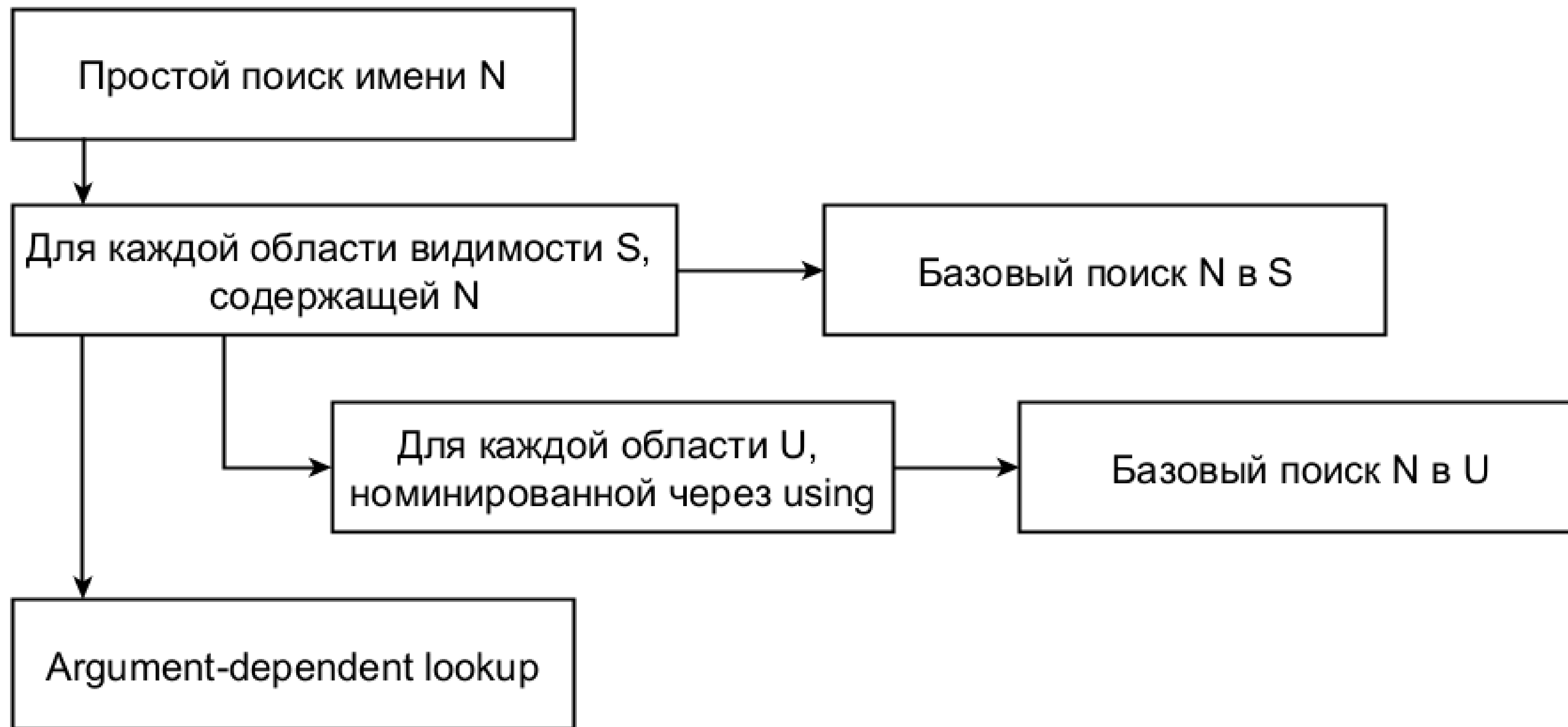
- Даже если результат определён, использование может быть ошибочным:

```
struct E: public A, public D {};
```



https://godbolt.org/z/Enbajfb6Y

Неквалифицированный поиск



Особое правило для using namespace

```
namespace C { int x = 0; }
namespace B { int x = 0; }

namespace A {
  using namespace B;
  using C::x;
  void f() {
    x = 2;    // ok, C::x
  }
}
```

- During unqualified name lookup, the names appear as if they were declared in the nearest enclosing namespace which contains **both** the using-directive and the nominated namespace **[namespace.udir]**

- Что будет если вынести using namespace выше или внести пространство имён B внутрь A?

https://godbolt.org/z/19sbxaGWz

Интересное следствие этого правила

```
namespace C { int x = 0; }

namespace A {
  namespace B { int x = 0; }
  using namespace B;
  using C::x;
  void f() {
    x = 2;    // error, found both
  }
}
```

- В обоих случаях ошибка.

- During unqualified name lookup, the names appear as if they were declared in the nearest enclosing namespace which contains **both** the using-directive and the nominated namespace **[namespace.udir]**

https://godbolt.org/z/e717vvh3v

ADL и ассоциированное множество

- Для каждого типа T внутри вызова функции действует поиск по ассоциированному множеству сущностей.

Fundmanetal type	Class	Template of anything	Function
Empty set	Class itself, bases, wrapping classes	All associated with template parameters	All associated with arguments and retval

- Пространства имён охватывающие все сущности ассоциированного множества являются ассоциированными пространствами имён.
- ADL это базовый поиск в ассоциированных пространствах имён (я опять таки срезаю ряд деталей).

Ассоциированное множество

- Ассоциированным множеством для double является пустое множество, поэтому здесь выигрывает int.

```
int foo(int) { return 1; }
template <typename T> int bar(T t) { return foo(t); };
int foo(double) { return 2; }
int main() {
  auto t = bar(1.0); // t == 1, S(double) = {}
```


https://godbolt.org/z/needj5bxc

Ассоциированное множество

- Ассоциированным множеством для X является он сам, поэтому здесь выигрывает X.

```
struct X {};

int foo(int) { return 1; }

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

int foo(X) { return 2; }

int main() {
  auto t = bar(X{}); // t == 2, S(X) = {X}
```

- Иногда хотелось бы такого для встроенных типов.

https://godbolt.org/z/43aGWrG91

Ассоциированное множество

- С другой стороны стороны unqualified поиск может начинаться в контексте инстанцирования.

```
int foo(int, std::type_identity<int>) { return 1; }

template <typename T> int bar(T t) {
  return foo(t, std::type_identity<T>{});
};

int foo(double, std::type_identity<double>) { return 2; }

int main() {
  auto t = bar(1.0); // t == 2, S = {tid<int>, tid<double>}
```


https://godbolt.org/z/fz05EsMao

Hidden friend

- Друзья определённые внутри класса ищутся только через ADL.

```
struct X {
  friend bool operator==(X lhs, X rhs) {
    return lhs.data == rhs.data;
  }
};

struct Y {
  operator X() const { return X{}; }
};

X a, b; Y c, d;

(a == b); // OK, but (c == d); // FAIL
```


https://www.youtube.com/watch?v=POa_V15je8Y

Квалифицированный поиск

- Квалифицированный поиск в классе, перечислении или пространстве имён проводит базовый поиск по ним.
- If nothing is found by qualified lookup for a member-qualified name that is the terminal name of a nested-name-specifier and is not dependent, it undergoes unqualified lookup [basic.lookup.qual.general]
- И есть масса исключений относительно того как искать имя деструктора, но в целом они не так интересны.
- Что интересно, квалифицированный поиск это всего лишь базовый поиск, то есть он игнорирует using namespace.

Устойчивость такого рода поиска

```
namespace C { int x = 0; }
namespace B { int x = 0; }

namespace A {
  using namespace B;
  using C::x;
  void f() {
    A::x = 1; // ok, C::x
  }
}
```

```
namespace C { int x = 0; }

namespace A {
  namespace B { int x = 0; }
  using namespace B;
  using C::x;
  void f() {
    A::x = 1; // ok, C::x
  }
}
```

https://godbolt.org/z/19sbxaGWz

Небольшая загадка

```
struct S; // Неполный тип

template <typename T> struct Wrapper { T val; };
void foo(const Wrapper<S>&) {}

Wrapper<S>& get();

int main() {

  foo(get()); // FAIL

  ::foo(get()); // OK

  (foo)(get()); // OK
```


https://godbolt.org/z/EoGszE5PT

Разгадка очень проста

- По правилам языка, компилятор обязан сделать инстанцирование если от инстанцированого типа или контекста инстанцирования что-то может зависеть.
- В данном случае для неквалифицированного поиска это действительно так.
- ADL может искать в ассоциированных множествах.

```
foo(get()); // FAIL
```

- Тогда как для квалифицированного поиска это не применимо.

```
::foo(get()); // OK
```

- Разгадку почему работает `(foo)(get())` оставим на секцию вопросов.

Где заканчивается поиск?

- Поиск неквалифицированного имени заканчивается как только найдено имя.

```
int foo(int) { return 1; }

namespace A {
  using foo = int;
  int bar() {
    return foo(2);
  }
}

int main() { A::bar(); // → 2
```

- Здесь довольно легко добиться вывода единицы.

https://godbolt.org/z/n4oME3fr3

Где заканчивается поиск?

- Поиск неквалифицированного имени заканчивается как только найдено имя.

```
int foo(int) { return 1; }

namespace A {
  using foo = int;
  int bar() {
    using ::foo
    return foo(2);
  }
}

int main() { A::bar(); // → 1
```


https://godbolt.org/z/f9dEW9Pzx

Место поиска имён

- Поиск имён начинается после замены алиасов но до перегрузки.

```
namespace S {
  using vector = std::vector<int>;
  void foo(vector) {}
}

int main() {
  foo(S::vector{}); // ошибка!
```

- `A typedef-name can also be introduced by an alias-declaration. The identifier following the using keyword is not looked up [dcl.typedef, 2]`

https://godbolt.org/z/WbaYz1Mf9

Дурная репутация алиасов

- Поиск имён начинается **после** замены алиасов но **до** разрешения перегрузки.

```
namespace S {
  struct vector { std::vector<int> v; };
  void foo(vector) {}
}

int main() {
  foo(S::vector{}); // ok!
```

- Это даёт алиасам их дурную репутацию в языке: они исчезают слишком рано.

https://godbolt.org/z/WbaYz1Mf9

Черные лебеди

- Древние греки приводили утверждение "все лебеди белые" как пример общезначимого утверждения в языке.
- При открытии Австралии это выражение стало ложным.
- Может ли язык изменяться в обратную сторону, то есть могут ли конструкции бывшие истинными в C++N становиться в нём ложными сильно после N-го года?

Ретроэволюция: static_assert

- Конструкция из C++11. Здесь условие независимое и ошибка первой фазы.

```
template <typename T> void buz(T t) {
  static_assert(false && "Please don't call buz");
}
```

- Довольно легко выкрутится сделав имя зависимым.

```
template <typename T> struct false_t : std::false_type {};

template <typename T> void buz(T t) {
  static_assert(false_t<T>::value && "Please don't call buz");
}
```

- Она была **ретроактивно** (CWG feb.2023) признана корректной в **C++11**.

https://godbolt.org/z/1PaEzr6Wq

Сцилла и Харибда

- Во всех компиляторах до gcc12 включительно static_assert ведёт себя (ретроактивно) некорректно.
- В gcc13 и далее сломана обратная совместимость по libstdc++.

For C++, construction of the global iostream objects `std::cout`, `std::cin`, etc. is now done inside the standard library, instead of in every source file that includes the `<iostream>` header.

This change improves the start-up performance of C++ programs, but it means that code compiled with GCC 13.1 **will crash** if the correct version of libstdc++.so is not used at runtime. See the documentation about using the right libstdc++.so at runtime.

- Знакомый выбор из двух стульев.

https://cplusplus.github.io/CWG/issues/2518.html

Общие выводы

- Процесс поиска имён сложен, но в своей основе не слишком сложен.
- Вам всего лишь надо выучить как работают пять видов поиска в семи видах областей видимости.
- Описание языка должно включать в себя точное и недвусмысленное описание процесса денотации и в 23-м стандарте к этому был сделан критически важный шаг.
- Поиск имён не единственный семантический процесс, но это уже совсем другая история.

Рекомендованные вопросы

- Вы много говорили про философию языка. А что вы думаете про парадокс лжеца?
- Может ли эволюция языка быть произвольной или она всегда жёстко определена рациональными соображениями?
- Как поиск имён взаимодействует с парсером?
- Расскажите больше про reachability в модулях.
- Покажите интересный пример введения имён.
- Как ADL работает в случае если сущность не до конца ясна?
- Покажите интересный пример с перегрузкой.

ВСЕМ СПАСИБО

А теперь ваши вопросы

Вернёмся к философии языка

- Парадокс лжеца: "Это утверждение ложно".
- Есть соблазн приписать таким утверждениям специальное значение "не определено".
- Месть лжеца: "Это утверждение или ложно или не определено".
- А тут как быть?
- Рассел формулировал это так: "рассмотрим множество всех множеств, не включащих сами себя. Включает ли оно само себя?".
- Ответ Цермело-Френкеля: запретить множество всех множеств.
- Какой ответ даст язык C++?

Ответ в стиле C++

- Ничего неопределённого в языке нет. Высказывание "это утверждение ложно" может быть истолковано как **любое определённое высказывание**.

```
bool foo() { return !foo(); }
```

- В том числе как предложение ударить сказавшего по лицу (UB).
- На этапе компиляции любая неопределённость это ошибка (ill-formed).

```
auto foo() {
  if constexpr (std::is_same_v<decltype(foo()), std::true_type>)
    return std::false_type{};
  else
    return std::true_type{};
}
```

Месть лжеца?

- В рантайм-месть лжеца нас не пускает теорема Райса.

```
bool foo() { return loops_forever(foo) || !foo(); }
```

- Есть ли что-то, что нас не пускает в compile-time месть лжеца?

```
auto foo() {
  if constexpr (true_v<foo>::value || __ct_error(foo))
    return std::false_type{};
  else
    return std::true_type{};
}
```

- Амбициозная программа это придумать такой билтин.

Произвольная эволюция

```
namespace A { struct S {}; }
namespace B {
  using A::S; // если это закомментировать
  typedef A::S S;
  using S = A::S;


  struct S s; // тогда тут ошибка
}
```

Начало произвольной эволюции

```
namespace A { struct S {}; }
namespace B {


  typedef A::S S;
  using S = A::S;

  using A::S; // а если это переставить?

  struct S s; // тогда тут всё хорошо?
}
```


cwg_defects.html#1894

И её итог

```
namespace A { struct S {}; }
namespace B {


  typedef A::S S;
  using S = A::S;

  using A::S;
  struct S s;
}
```

Notes from the June, 2014 meeting:

CWG felt that these examples should be well-formed.

Взаимодействие с парсером

- У парсера ограничены возможности догадываться о сущностях, скрывающихся под именами.

```
template <typename T> int foo(T *p) {
  p->bar<int>(); // FAIL, < is "operator less"
  p->template bar<int>(); // OK
```

- Иначе у нас были бы очень неприятные проблемы с разбором.

```
template<typename T> bool Foo(T it) {
  return it->end < it->end;
}
```

- С этим связана также известная дизамбигуация с typename.

Достижимость

- Единица трансляции U достижима из точки P если единица трансляции, содержащая P импортирует U в любой точке до P.
- Объявление D достижимо из точки P если:
  - D находится ранее чем P в той же единице трансляции, либо
  - D находится в единице трансляции, достижимой из P, и при этом не в приватном фрагменте модуля.
- Тип считается полным когда его определение достижимо.

```
Y y; // OK, definition of X is reachable, Y is visible
X x; // error X is not visible
```

Что делать с reachable redeclarations?

```
// TU1
export module A;
struct X {};
export using Y = X;
// TU2
import A;
using X = int; // OK?
Y y = 1; // OK?
```

- Modules significantly complicate the situation, principally by invalidating the tacit assumption that **every** lookup is restricted to preceding declarations in the same translation unit.

S. Davis Herring

```
main.cpp:6:5: error: no viable conversion from
'int' to 'Y' (aka 'X')
    6 |   Y y = 1;
      |     ^   ~
```


https://godbolt.org/z/K6M8zh84E

Особенности введения имён

```
void foo(int); // 1
template <typename T> void foo(int); // OK

struct foo { foo(int); }; // 2
template <typename T> struct foo {}; // FAIL

foo(0); // → 1

foo{0}; // FAIL

struct foo{0}; // FAIL

foo x{0}; // FAIL

struct foo y{0}; // → 2
```


https://godbolt.org/z/1z3dGseen

Определяем категорию имён

- ADL вообще не начнётся если сущность которую он пытается искать никак не введена.

```
struct foo {};

namespace N {
  struct X {};
  template <typename T> void foo(X x) {}
}

// template <typename T> void foo();

int main() {
  foo<int>(N::X{}); // error (uncomment to make correct)
}
```


https://godbolt.org/z/TGqqPdGbe

Интересный пример с перегрузкой

```
struct S;

template <typename T> struct Wrapper { T val; };
using P = Wrapper<S>;

void foo(const int&) {}
void foo(const P&) {}

int& get();

int main() {
  foo(get()); // error
  ::foo(get()); // also error
```


https://godbolt.org/z/aoMnjhx4s

Что в нём интересно?

- Тут компилятор имеет право и скомпилировать и не скомпилировать.

```
If the function selected by overload resolution can be
determined without instantiating a class template definition,
it is unspecified whether that instantiation actually takes
place [temp.inst, 9]
```

```
template <typename T> struct Wrapper { T val; };
using P = Wrapper<S>;

void foo(const int&) {}
void foo(const P&) {} // можем обойтись, но обойдёмся ли?
```

- Гайдлайн понятен: просто не надейтесь на неполные типы.

Ещё одна задачка

```
namespace K {
  template <typename T, typename U = char> struct A {};
  A<short> *a;
}

template <typename T> using A = K::A<short, T>;

int main() {
  K::a->~A<char>();
}
```

- Считайте это домашней работой. Этот случай на самом деле не так прост.

Отступление к unqual lookup

```
struct A {
  int B;
  void f() {}
};

using B = A;

template<class T> void g(T *p) {
  p->B::f(); // OK, non-type A::B ignored
}

int main() {
  A a; g(&a);
}
```