ИМЕНА И ПЕРЕГРУЗКА

Введение в обобщённое программирование и семантические процессы. Поиск имён и перегрузка функций.

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

Возводим число в степень

- "The first step is to get the algorithm right. The second step is to figure out which sorts of things (types) it works for" – Alex Stepanov
- Начнём с первого.

```
unsigned nth_power(unsigned x, unsigned n); // returns x^n
```

- Как написать тело этой функции?

Выбираем правильный алгоритм

- "The first step is to get the algorithm right. The second step is to figure out which sorts of things (types) it works for" – Alex Stepanov

```
unsigned nth_power(unsigned x, unsigned n) {
  unsigned acc = 1;
  if ((x < 2) || (n == 1)) return x;
  while (n > 0) {
    if ((n & 0x1) == 0x1) { acc *= x; n -= 1; }
    x *= x; n /= 2;
  }
  return acc;
}
```


https://godbolt.org/z/bvr7PKKvY

Ищем возможности обобщения

- "The first step is to get the algorithm right. The second step is to figure out which sorts of things (types) it works for" – Alex Stepanov

```
unsigned nth_power(unsigned x, unsigned n) {
  unsigned acc = 1;
  if ((x < 2) || (n == 1)) return x;
  while (n > 0) {
    if ((n & 0x1) == 0x1) { acc *= x; n -= 1; }
    x *= x; n /= 2;
  }
  return acc;
}
```

Наивное обобщение

- "The first step is to get the algorithm right. The second step is to figure out which sorts of things (types) it works for" – Alex Stepanov

```
template <typename T> T nth_power(T x, unsigned n) {
  T acc = 1;
  if ((x < 2) || (n == 1)) return x;
  while (n > 0) {
    if ((n & 0x1) == 0x1) { acc *= x; n -= 1; }
    x *= x; n /= 2;
  }
  return acc;
}
```

Проблемы наивного обобщения

- "The first step is to get the algorithm right. The second step is to figure out which sorts of things (types) it works for" – Alex Stepanov

```
template <typename T> T nth_power(T x, unsigned n) {
  T acc = 1;
  if ((x < 2) || (n == 1)) return x;
  while (n > 0) {
    if ((n & 0x1) == 0x1) { acc *= x; n -= 1; }
    x *= x; n /= 2;
  }
  return acc;
}
```


https://godbolt.org/z/qKz8qY7s9

Менее наивное обобщение

- "The first step is to get the algorithm right. The second step is to figure out which sorts of things (types) it works for" – Alex Stepanov

```
template <typename T> T nth_power(T x, unsigned n) {
  T acc = id<T>();
  if ((x == acc) || (n == 1)) return x;
  while (n > 0) {
    if ((n & 0x1) == 0x1) { acc *= x; n -= 1; }
    x *= x; n /= 2;
  }
  return acc;
}
```

Заводим traits

- "The first step is to get the algorithm right. The second step is to figure out which sorts of things (types) it works for" – Alex Stepanov

```
template <typename T, typename Trait = default_id_trait<T>>
T nth_power(T x, unsigned n) {
  T acc = Trait::id();
  if ((x == acc) || (n == 1)) return x;
  while (n > 0) {
    if ((n & 0x1) == 0x1) { acc *= x; n -= 1; }
    x *= x; n /= 2;
  }
  return acc;
}
```


https://godbolt.org/z/aET9j6qKo

Отделяем чистую часть

- "The first step is to get the algorithm right. The second step is to figure out which sorts of things (types) it works for" – Alex Stepanov

```
template <typename T> T nth_power(T x, T acc, unsigned n) {
  while (n > 0)
    if ((n & 0x1) == 0x1) { acc *= x; n -= 1; }
    else { x *= x; n /= 2; }
  return acc;
}

unsigned nth_power(unsigned x, unsigned n) {
  if (x < 2u || n == 1u) return x;
  return nth_power<unsigned>(x, 1u, n);
}
```


https://godbolt.org/z/dvrc3Pj8n

Перегрузка функций

```
template <typename T> T nth_power(T x, T acc, unsigned n);

unsigned nth_power(unsigned x, unsigned n);
int nth_power(int x, unsigned n);
Matrix nth_power(Matrix x, unsigned n);
```

- Перегрузка означает необходимость связи имени с сущностью.

```
auto t = nth_power(x, n); // ???
```

- Это означает, что в языке должны быть **поиск имён** и **разрешение перегрузки**.

Обсуждение

- Как вообще в языке при разрешении имён мы понимаем как имя связывается с сущностью?

```
foo(s);
```

- Например как может быть истолкована строчка выше?

https://godbolt.org/z/WT51drTsq

Связывание имён 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 обозначается как x и какой как A::x?

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

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

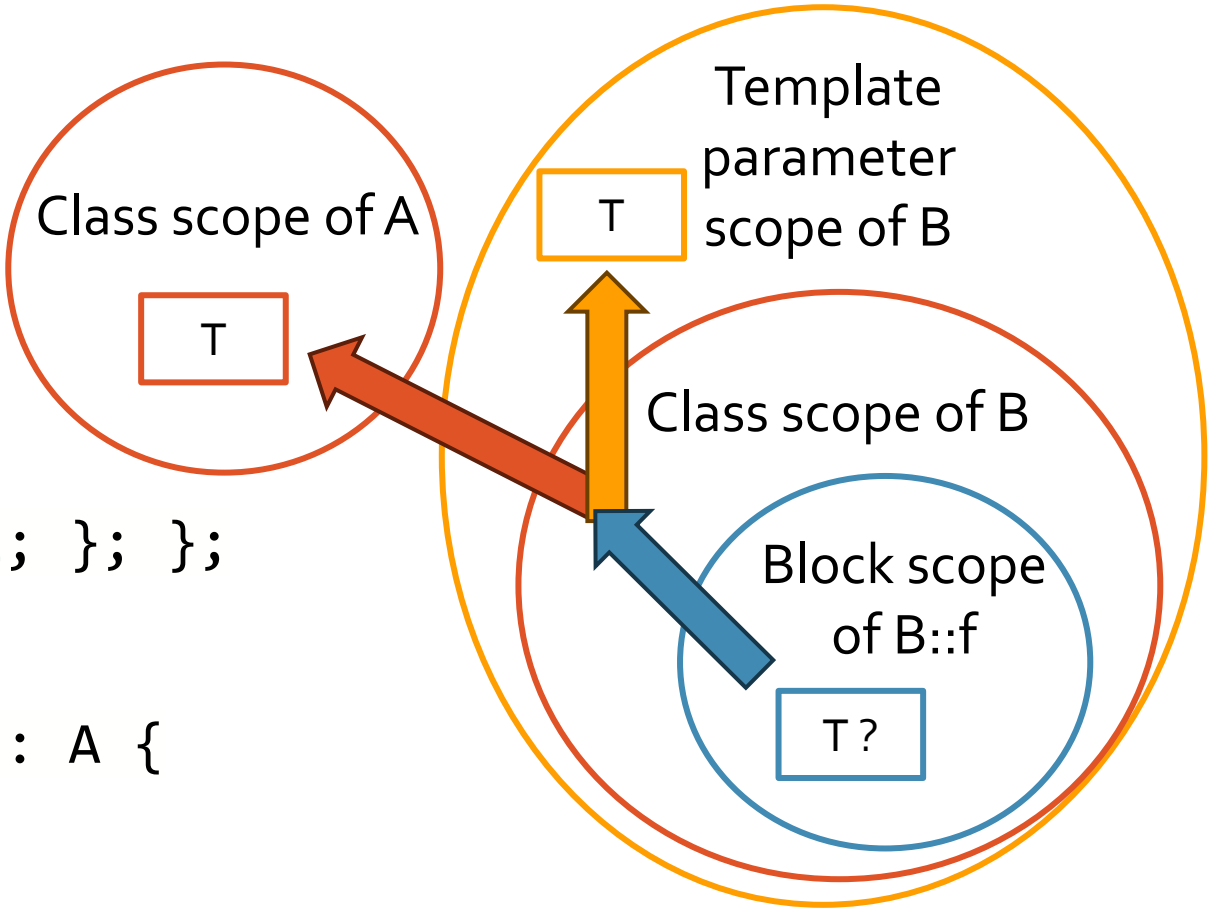
Поиск в областях видимости (scopes)

- 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?



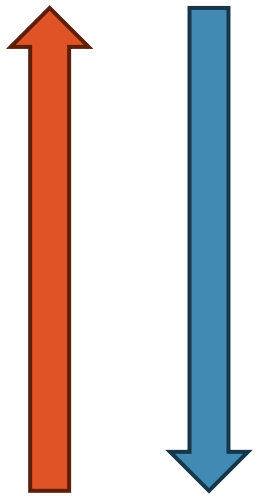

```
struct A { struct T { int x = 1; }; };

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

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>

Обсуждение: что такое семантика

- Такое чувство, что такие семантические процессы, как поиск имён должны как-то пересекаться с синтаксическим разбором и взаимодействовать с ним?

7.6.2 Unary expressions [expr.unary]

7.6.2.1 General [expr.unary.general]

1 Expressions with unary operators group right-to-left.

unary-expression:
postfix-expression
unary-operator cast-expression
`++` *cast-expression*
`--` *cast-expression*
await-expression
`sizeof` *unary-expression*
`sizeof (` *type-id* `)`
`sizeof ... (` *identifier* `)`
`alignof (` *type-id* `)`
noexcept-expression
new-expression
delete-expression

unary-operator: one of
`* & + - ! ~`

7.6.2.2 Unary operators [expr.unary.op]

1 The unary `*` operator performs *indirection*: the expression to which it is applied shall be a pointer to an object type, or a pointer to a function type and the result is an lvalue referring to the object or function to which the expression points. If the type of the expression is "pointer to `T`", the type of the result is "`T`".

Синтаксис языка

- Грамматическая **продукция** "A : B" означает "A раскрывается в B".
- Курсивом выделены **нетерминальные** символы. Обычным шрифтом выделены **терминальные** символы.
- В конечной программе на C++ вы не видите нетерминалов.
- В грамматике C++ вы слева всегда видите один нетерминал. Это означает, что она **контекстно-свободна**.

unary-expression:
postfix-expression
unary-operator cast-expression
`++` *cast-expression*
`--` *cast-expression*
await-expression
`sizeof` *unary-expression*
`sizeof (` *type-id* `)`
`sizeof ... (` *identifier* `)`
`alignof (` *type-id* `)`
noexcept-expression
new-expression
delete-expression

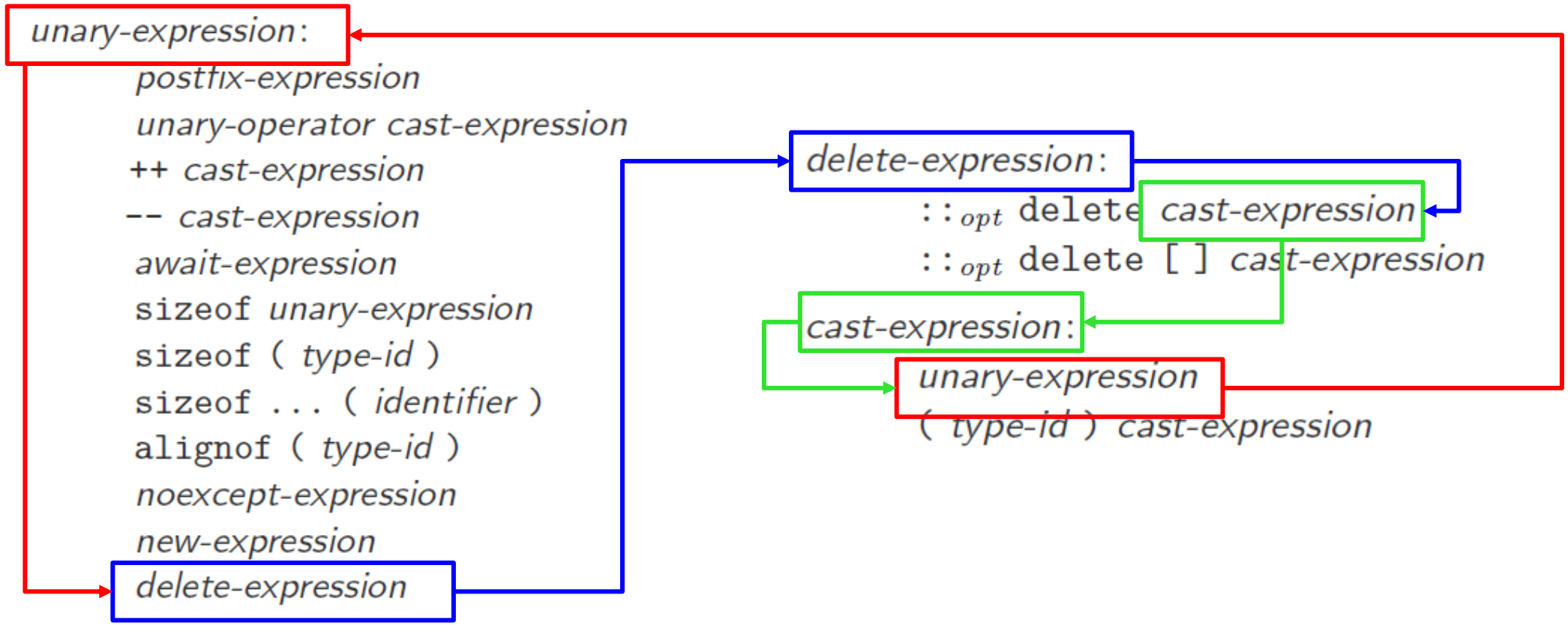
https://eel.is/c++draft/gram

Простой вопрос

- Корректно ли это выражение синтаксически?

```
auto *p = new int;
delete delete delete p;
```

Воспользуемся синтаксисом не глядя



Проблема грамматик: типы

- У этого выражения проблемы с **семантикой**.

```
auto *p = new int;
delete delete delete p;
```

```
<source>:5:17: error: type 'void' argument given to 'delete', expected pointer
    5 |   delete delete delete p;
      |                 ^~~~~~~
```

- Синтаксически оно правильное. Оно не проходит **проверку типов**.
- Проверка типов не регулируется контекстно свободной грамматикой.

Проблема грамматик: неоднозначности

- Скомпилируется ли это?

```
delete [] {
  return new int;
}();
```

- Из синтаксиса мы делаем вывод что такая продукция есть.
- И более того она указана первой.

delete-expression:
    ::$_{opt}$ `delete` *cast-expression*
    ::$_{opt}$ `delete [ ]` *cast-expression*

cast-expression → *unary-expression*

unary-expression → *postfix-expression*

postfix-expression → *primary-expression*

primary-expression → *lambda-expression*

Синтаксические неоднозначности

```
<source>:4:13: error: expected primary-expression before '{' token
    4 |   delete [] { return new int; } ();
      |             ^
```

7.6.2.9 Delete **[expr.delete]**

1 The *delete-expression* operator destroys a most derived object (6.7.2) or array created by a *new-expression*.

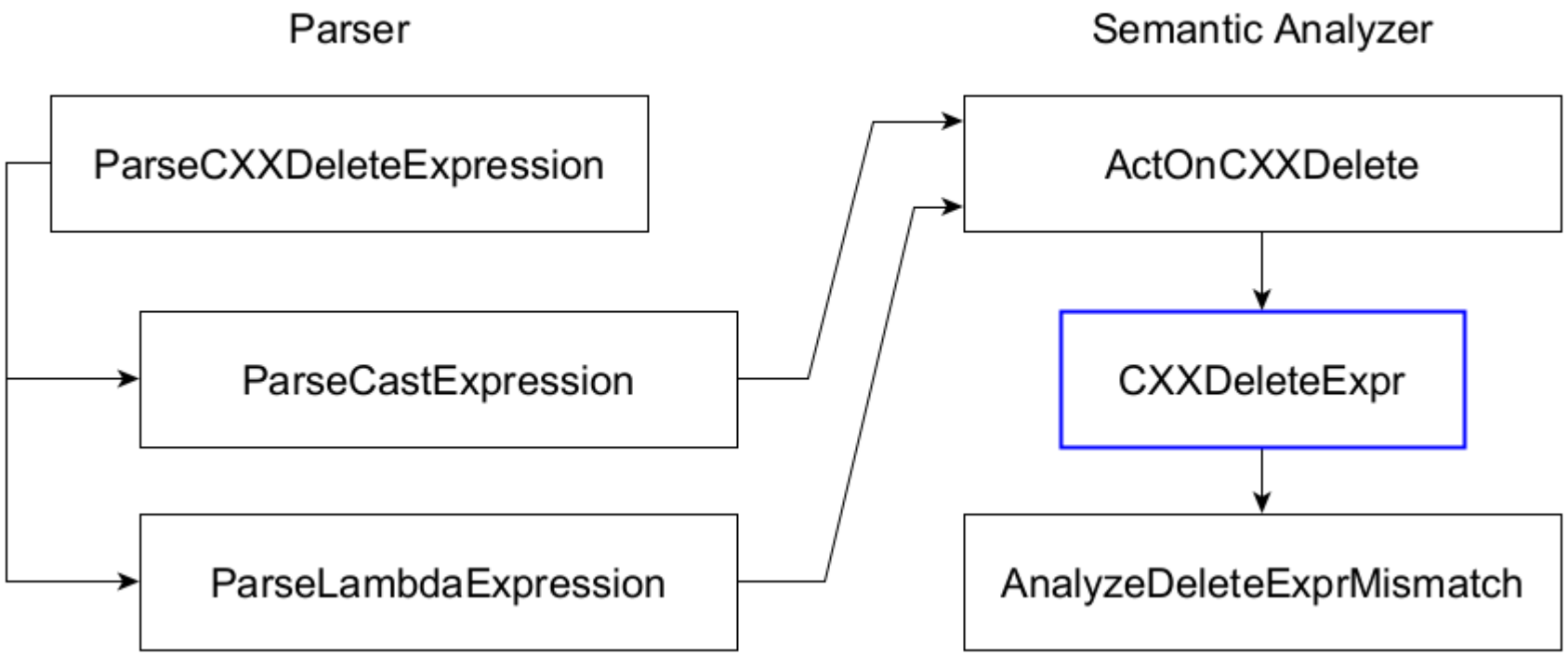
delete-expression:
`::`$_{opt}$ `delete` *cast-expression*
`::`$_{opt}$ `delete [ ]` *cast-expression*

The first alternative is a *single-object delete expression*, and the second is an *array delete expression*. Whenever the `delete` keyword is immediately followed by empty square brackets, it shall be interpreted as the second alternative.[76] The operand shall be of pointer to object type or of class type. If of class type, the operand is contextually implicitly converted (7.3) to a pointer to object type.[77] The *delete-expression*'s result has type `void`.

Обсуждение

- В языке есть **контекстно-зависимые** конструкции: например смысл квадратных скобок определяется тем что до и после них.
- Почему же у нас для языка сделана **контекстно-свободная** грамматика?

Локальная контекстная зависимость



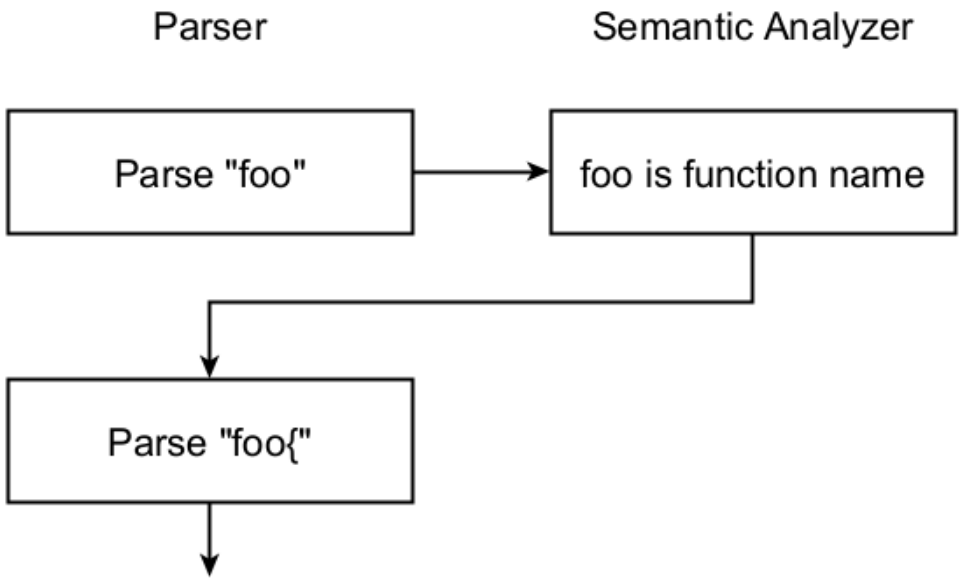
Особенности разрешения имён

```
void foo(int);              // 1
struct foo { foo(int); }; // 2
foo(0); // ???
foo{0}; // ???
```

Взаимодействие с парсером

```
void foo(int);                  // 1
struct foo { foo(int); };       // 2
foo(0); // → 1
foo{0}; // FAIL
```

- Здесь "foo" трактуется как discarded function pointer.




```
<source>:11:6: error: expected ';' before '{' token
   11 |   foo{0};
      |      ^
```


https://godbolt.org/z/YGzf6Kbnf

Семантические процессы

- Я предлагаю все контекстно-зависимые сложности языка свести к трём* основным **семантическим процессам**.
  - Поиску и разрешению имён, включая разрешение перегрузки.
  - Выводу и подстановке типов.
  - Инстанцированию шаблонов.
- В **нормальных случаях** они следуют один за другим.

```
template <typename T> T min(T x, T y);
...
int a = min(10, 15); // looked up min, deduced min<int>,
                     // specialization min<int> created
```

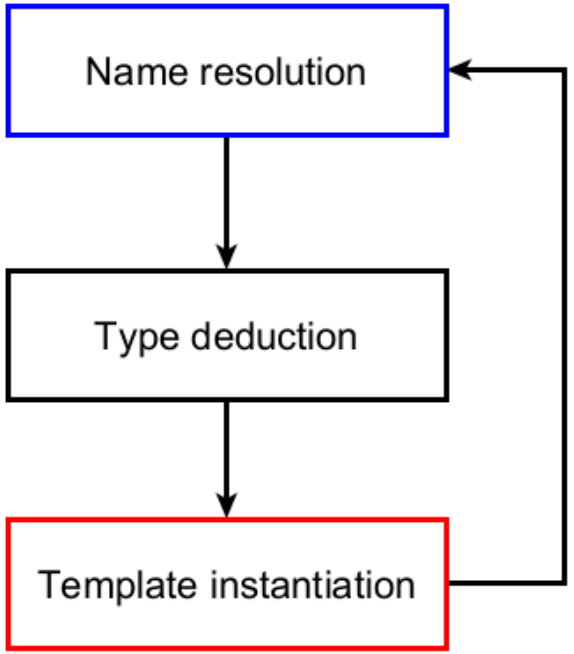

*трём плюс одному

Инстанцирование это цикл

```
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) → ⊥
```



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

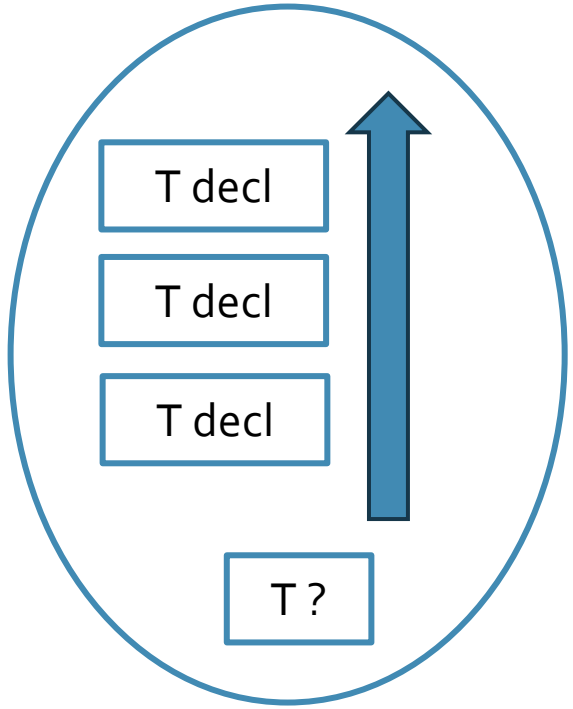
qualified

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
```

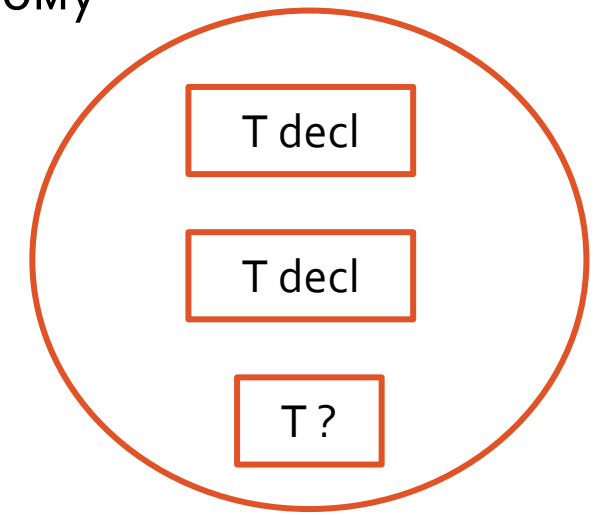
- Даже простой поиск не так прост т.к. ограничивает разные имена и сущности.

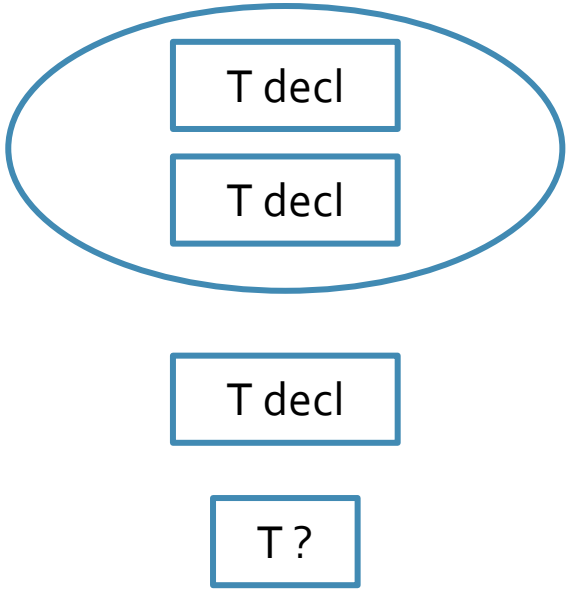


https://godbolt.org/z/9Pbsn74xq

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.





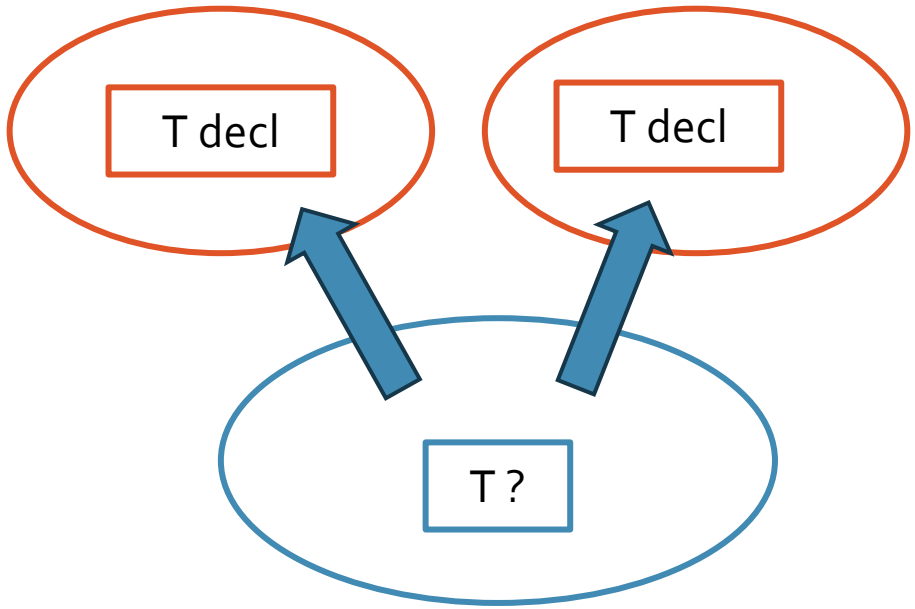
Поиск в базовых классах

- 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 }
```

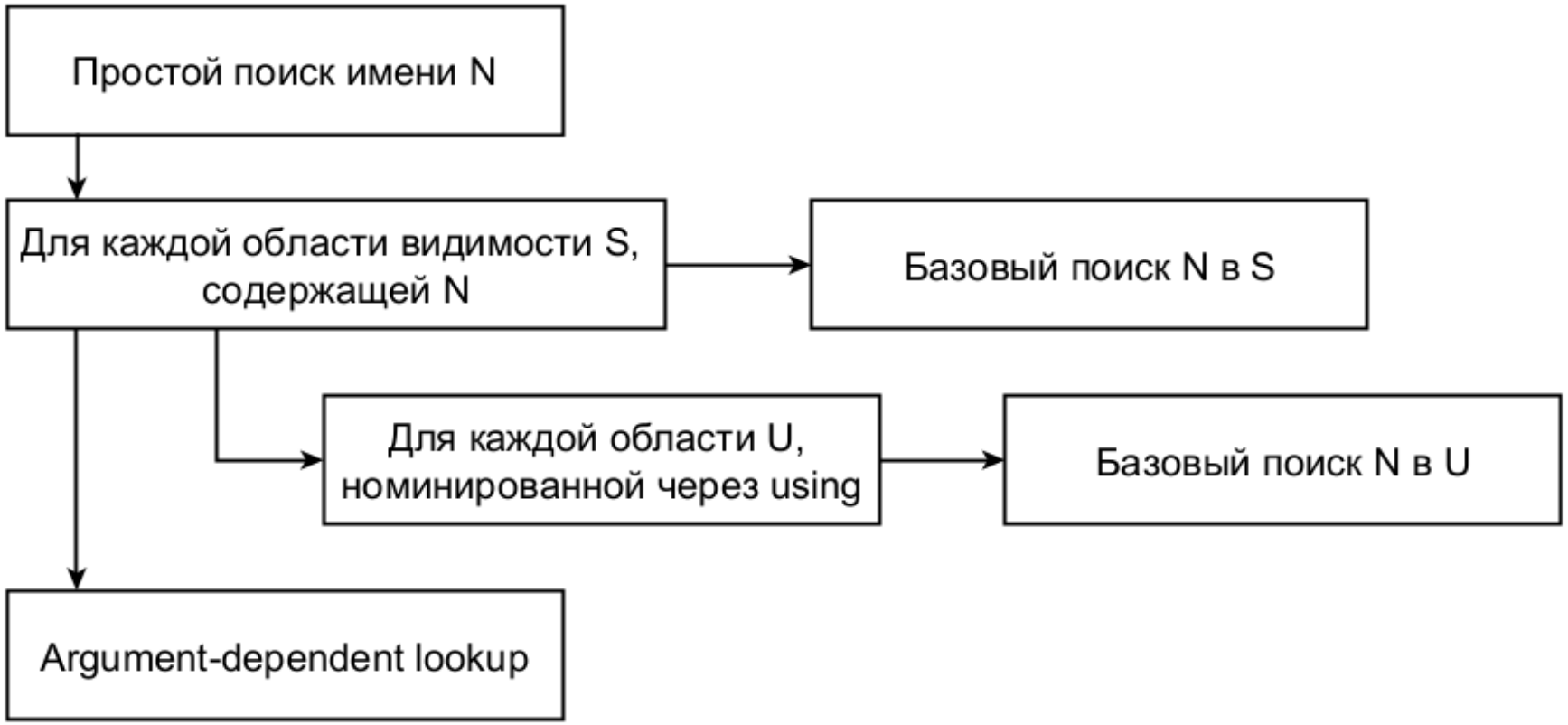
- Даже если результат определён, использование может быть ошибочным:

```
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

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]
- 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

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

Обсуждение

- Что является атомом проектирования в языке C++?

Обобщённые функции

- Функция это скорее протон или электрон.

```
template <typename T> T nth_power(T x, T acc, unsigned n);
unsigned nth_power(unsigned x, unsigned n);
double nth_power(double x, unsigned n);
```

- **Атомом обобщённого программирования является множество перегрузки.**
- Мы будем сталкиваться с этим неоднократно: у классов проектируются множества перегруженных конструкторов, операторов, методов и т.п.

Примеры хорошего проектирования

- 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&&);
```

Правила Винтерса

- Человек не должен быть обязан проводить в уме процесс перегрузки.
- Единый комментарий может описать всё множество.
- Каждый элемент множества перегрузки делает примерно одно и то же.
- Ниже пример **очень плохого** дизайна.

```
// if calling first version returns smallest co-prime for n
// otherwise process coma-separated list of co-primes
// to deduce least possible n
int least_coprime(int n);
int least_coprime(const std::string& x);
```

Разбор примера с прошлой лекции

- В конце прошлой лекции была поставлена задача написать `operator==` для `basic_string`
- Один из простых вариантов решения

```
template<typename CharT, typename Traits, typename Alloc>
bool operator==(const basic_string<CharT, Traits, Alloc>& lhs,
                const basic_string<CharT, Traits, Alloc>& rhs) {
  return lhs.compare(rhs) == 0;
}
```

- Чем он плох?

Разбор примера с прошлой лекции

- В конце прошлой лекции была поставлена задача написать `operator==` для `basic_string`
- Один из простых вариантов решения

```
template<typename CharT, typename Traits, typename Alloc>
bool operator==(const basic_string<CharT, Traits, Alloc>& lhs,
                const basic_string<CharT, Traits, Alloc>& rhs) {
  return lhs.compare(rhs) == 0;
}
```

- Он неэффективен. Подумайте про (`"hello" == str`), тут явно создаётся лишняя копия. Мы бы хотели его перегрузить, как обычную функцию.

Лучший вариант сравнения

- Принятый (в т.ч. в libstdc++) вариант решения использует перегрузки.

```
template<typename CharT, typename Traits, typename Alloc>
bool operator==(const basic_string<CharT, Traits, Alloc>& lhs,
                const basic_string<CharT, Traits, Alloc>& rhs) {
  return lhs.compare(rhs) == 0;
}

template<typename CharT, typename Traits, typename Alloc>
bool operator==(const CharT* lhs, const basic_string<CharT, Traits, Alloc>& rhs) {
  return rhs.compare(lhs) == 0;
}

template<typename CharT, typename Traits, typename Alloc>
bool operator==(const basic_string<CharT, Traits, Alloc>& lhs, const CharT* rhs) {
  return lhs.compare(rhs) == 0;
}
```

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);
```

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

Обсуждение: правила перегрузки

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

int foo(int x);                          // 1
int foo(const int &x);                   // 2
template <typename T> int foo(T x);      // 3
int foo(char x);                         // 4
int foo(short x);                        // 5
int foo(double x);                       // 6
int foo(S x);                            // 7
int foo(S x);                            // 8

foo(1); // ???
```

Общий обзор правил перегрузки

- Выбирается множество **перегруженных имён**.
- Выбирается множество **кандидатов**.
- Из множества кандидатов выбираются **жизнеспособные** (viable) кандидаты для данной перегрузки.
- Лучший из жизнеспособных кандидатов выбирается на основании **цепочек неявных преобразований** для каждого параметра.
- Если лучший кандидат **существует и является единственным**, то перегрузка разрешена успешно, иначе программа ill-formed.

Видимые и невидимые имена

- Вопрос на поиск имён: что на экране?

```
struct B {
  void f(int) { std::cout << "B" << std::endl; }
};

struct D : B {
  void f(const char*) { std::cout << "D" << std::endl; }
};

int main() {
  D d; d.f(0);
}
```


https://godbolt.org/z/jf66q65jM

Введение имени в scope

- Прошлый пример может показаться странным, но давайте его упростим

```
void f(int) { std::cout << "B" << std::endl; }
void f(const char*) { std::cout << "D" << std::endl; }

int main() {
  extern void f(const char *);
  f(0); // тут всё очевидно
}
```

- Определения ищутся внутри общего scope, что упрощает реализацию компилятора.
- Но тут может возникнуть вопрос насчёт пространств имён.

Проблема: операторы

- Обычно оператор может находиться в любом пространстве имён.

```
std::cout << "Hello\n"!;
```

- Это вполне может быть эквивалентно следующему.

```
operator<< (std::cout, "Hello\n"!);
```

- Чтобы это работало, это должен быть оператор из пространства имён std.

```
std::operator<< (std::cout, "Hello\n"!);
```

- Но компилятор не может об этом догадаться из записи `std::a << b`

Решение: ADL или поиск Кёнига

- Эндрю Кёниг предложил решение в начале 90-х

1. Компилятор ищет имя функции из текущего и всех охватывающих пространств имён.
2. Если оно не найдено, компилятор ищет имя функции в пространствах имён её аргументов.

```
namespace N { struct A; int f(A*); }

int g(N::A *a) { int i = f(a); return i; }
```

Решение: поиск Кёнига

- Эндрю Кёниг предложил решение в начале 90-х

1. Компилятор ищет имя функции из текущего и всех охватывающих пространств имён.
2. Если оно не найдено, компилятор ищет имя функции в пространствах имён её аргументов.

```
typedef int f;

namespace N { struct A; int f(A*); }

int g(N::A *a) { int i = f(a); return i; }
```

Поиск Кёнига и шаблоны

- Следующий пример не работает

```
namespace N {
  struct A;
  template <typename T> int f(A*);
}

int g(N::A *a){
  int i = f<int>(a); // FAIL
  return i;
}
```

- Кто-нибудь может угадать причину?

Поиск Кёнига и шаблоны

- Можно заставить это работать, введя f как имя шаблонной функции

```
namespace N {
  struct A;
  template <typename T> int f(A*);
}

template <typename T> void f(int); // неважно какой параметр

int g(N::A *a){
  int i = f<int>(a); // теперь всё ok
  return i;
}
```

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

Общий обзор правил перегрузки

- Выбирается множество **перегруженных имён**.
- Выбирается множество **кандидатов**.
- Из множества кандидатов выбираются **жизнеспособные** (viable) кандидаты для данной перегрузки.
- Лучший из жизнеспособных кандидатов выбирается на основании **цепочек неявных преобразований** для каждого параметра.
- Если лучший кандидат **существует и является единственным**, то перегрузка разрешена успешно, иначе программа ill-formed.

Идея построения цепочки

- С наивной точки зрения цепочку преобразований входят:
  - С высшим приоритетом: стандартные преобразования.
  - Немного ниже: пользовательские преобразования.
  - С низшим приоритетом: троеточия.
- Сложности начинаются когда их комбинируется много разных.

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

void foo(int); // 1
void foo(S);   // 2
void foo(...); // 3

foo(1); // → 1
```


https://godbolt.org/z/1eafvcYW6

Стандартные преобразования

Conversion	Category	Rank	Subclause
No conversions required	Identity	Exact Match	
Lvalue-to-rvalue conversion	Lvalue Transformation		7.3.1
Array-to-pointer conversion			7.3.2
Function-to-pointer conversion			7.3.3
Qualification conversions	Qualification Adjustment		7.3.5
Function pointer conversion			7.3.13
Integral promotions	Promotion	Promotion	7.3.6
Floating-point promotion			7.3.7
Integral conversions	Conversion	Conversion	7.3.8
Floating-point conversions			7.3.9
Floating-integral conversions			7.3.10
Pointer conversions			7.3.11
Pointer-to-member conversions			7.3.12
Boolean conversions			7.3.14

Стандартные преобразования

- Трансформации объектов (ранг точного совпадения).

```
int arr[10]; int *p = arr; // [conv.array]
```

- Коррекции квалификаторов (ранг точного совпадения).

```
int x; const int *px = &x; // [conv.qual]
```

- Продвижения (ранг продвижения).

```
int res = true; // [conv.prom]
```

- Конверсии (ранг конверсии).

```
float f = 1; // [conv.fpint]
```

Пользовательские преобразования

- Задаются implicit конструктором либо оператором преобразования.

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

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

- При этом (1) лучше чем (2) потому что для него нужно меньше стандартных преобразований.
- Интуитивно: у цепочки короче хвост значит она лучше.

https://godbolt.org/z/hP1c7n

Тонкости построения цепочек

```
struct S { S(long){} };
void foo(S) {}
int x = 42;
foo(x); // int -> long -> S
```

```
struct T { T(int){} };
struct S { S(T){} };
void foo(S) {}
int x = 42;
foo(x); // int -> T -> S
```

Задачи, часть 1

- Охарактеризуйте следующий пример с точки зрения C++23

```
template <class T1, class T2> struct Pair {
  template<class U1 = T1, class U2 = T2> Pair(U1&&, U2&&) {}
};

struct S { S() = default; };
struct E { explicit E() = default; };

int f(Pair<E, E>) { return 1; }
int f(Pair<S, S>) { return 2; }

static_assert(f({{}, {}}) == 2, ""); // Error or Assert or OK?
```

Задачи, часть 2

- Охарактеризуйте следующий пример с точки зрения C++23

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

struct X {
  operator Bar() { std::cout << "Bar\n"; return Bar{}; }
  operator Baz() const { std::cout << "Baz\n"; return Baz{}; }
};

void foo(const Foo &f) {}

int main() { foo(X{}); } // Bar or Baz?
```

Задачи, часть 3

- Язык `[RAM` работает с обращениями в заранее заданную одномерную память по следующим правилам.
- Форма `[addr]` это обращение в память по адресу `addr`. Форма `base[off]` это обращение в память по адресу `base + off`, `[` это синоним переменной `c`, и `][` это синоним переменной `x`.
- При этом первый приоритет имеют массивы, т.е. `a[][]` это не a[x], а ошибка.
- Второй приоритет имеют синонимы и третий приоритет имеет адрес без базы т.е. просто `][+[+1]` это обращение по `[x + c + 1]`.

```
input a; [ = a[5]; ][ = [[ + a - 2]; a = [[+][]; print a;
```

- Ваша задача: написать простой интерпретатор этого языка.

Литература

- ISO/IEC, "Information technology – Programming languages – C++", ISO/IEC 14882: 2023
- Bjarne Stroustrup, The C++ Programming Language (4th Edition), 2013
- Alexander A. Stepanov, Paul McJones, Elements of programming, Addison-Wesley, 2009
- Alexander A. Stepanov, Daniel E. Rose, From mathematics to generic programming, Addison-Wesley, 2014
- Titus Winters, Modern C++ Design (2 parts), CppCon, 2018
- Simon Brand, Overloading: The Bane of All Higher-Order Functions, CppCon, 2018
- Ansel Sermersheim, Back To Basics: Overload Resolution, CppCon, 2021
- Mateusz Pusz, Back to Basics — Name Lookup and Overload Resolution in C++, CppCon, 2022