ПОРОЖДЕНИЕ

Особенности шаблонного инстанцирования.

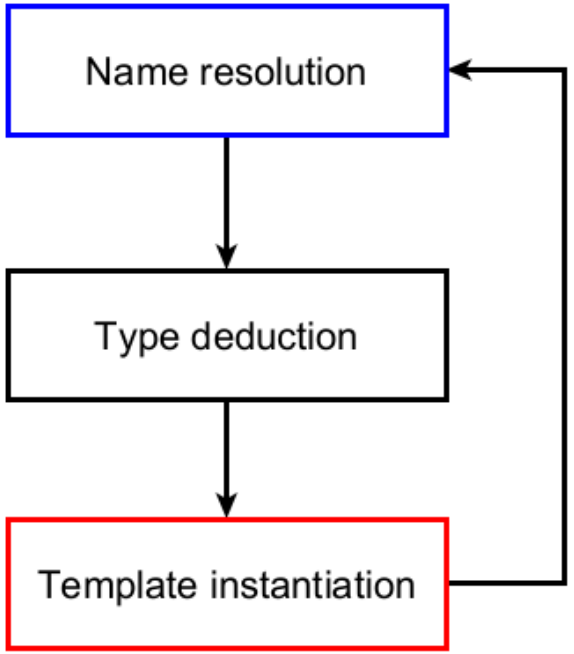
К. Владимиров, Syntacore, 2024
mail-to: konstantin.vladimirov@gmail.com

Нормальный процесс

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

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

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

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



Инстанцирование

- Инстанцирование это процесс порождения экземпляра специализации.

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

....

max<int>(2, 3); // порождает template<> int max(int, int)
```

- Мы называем этот процесс неявным (implicit) инстанцированием.
- Оно порождает код через подстановку параметра в шаблон и осуществляется по требованию (то есть лениво).

Порождение специализаций

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

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

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

Управление инстанцированием

- Инстанцирование может быть явно запрещено в этой единице трансляции.

```
extern template int max<int>(int, int);
```

- Инстанцирование может быть явно вызвано.

```
template int max<int>(int, int);
```

- Эта техника может использоваться для уменьшения размера объектных файлов при инстанцировании тяжёлых функций.

instantiations-demo

Явная специализация

- Кроме инстанцирования (явного или неявного), шаблонная функция или структура может быть явно специализирована.

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

- Специализация обязана физически следовать за основным шаблоном.

```
template <> int min<int>(int x, int y) { .... } // ошибка
template <typename T> T min(T x, T y) { .... } // primary тут
```

Пример явной специализации: OpenCL

- Общий случай:

```
template<typename T> struct ReferenceHandler { };
```

- Конкретные случаи:

```
template <> struct ReferenceHandler<cl_mem> {
  static cl_int retain(cl_mem memory)
    { return ::clRetainMemObject(memory); }
  static cl_int release(cl_mem memory)
    { return ::clReleaseMemObject(memory); }
};
```

- Теперь `ReferenceHandler<X>::release()` это либо release X либо ошибка.

CL/opencl.hpp

Правила игры

- Общее правило для функций и не только [temp.spec.general]
  - Явное инстанцирование единожды в программе.
  - Явная специализация единожды в программе.
  - Явное инстанцирование должно **следовать за** явной специализацией.

```
template <typename T> T max(T x, T y) { .... }
template <> int max<int>(int x, int y) { .... }
template int max<int>(int, int); // вынудили инстанцировать
```

- Нарушение влечет за собой IFNDR.
- Как вы думаете, как играет запрет инстанцирования со специализацией?

https://godbolt.org/z/MoafWWM9s

Инстанцирование и специализация

- Явная специализация может войти в конфликт с инстанцированием

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

// OK, указываем явную специализацию
template <> double max(double x, double y) { return 42.0; }

// никакой implicit instantiation не нужно
int foo() { return max<double>(2.0, 3.0); }

// процесс implicit instantiation нужен и он произошёл
int bar() { return max<int>(2, 3); }

// ошибка: мы уже породили эту специализацию
template <> int max(int x, int y) { return 42; }
```


https://godbolt.org/z/Y8zc7xdsd

Удаление специализаций

- Частным случаем явной специализации является её запрет.

```
// для всех указателей
template <typename T> void foo(T*);

// но не для char* и не для void*
template <> void foo<char>(char*) = delete;
template <> void foo<void>(void*) = delete;
```

- Как вы думаете, что произойдёт если мы сначала сгенерируем специализацию, а потом запретим её?

https://godbolt.org/z/o5WThaoxY

Non-type параметры

- Параметры не являющиеся типами могут быть **структурными типами**
- Структурные типы это:
  - Скалярные типы (кроме плавающей точки).
  - Левые ссылки.
  - Структуры у которых все поля и базовые классы public и не mutable. И при этом все поля и поля всех базовых классов тоже структурные типы или массивы.

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

template <int N, int *PN, int &RN, Pair P> int foo();
```

- Базовая интуиция: всё должно быть compile-time known.

https://godbolt.org/z/399f8eYbK

Специализация по nontype параметрам

- Нет никаких проблем в том, чтобы специализировать класс по нетиповым параметрам.

```
template <typename T, int N> foo(T(&Arr)[N]);

template <> foo<int, 3>(int(&Arr)[3]) {
  // тут более эффективная реализация для трёх целых
```

- Обратите внимание: при явной специализации функций вы обязаны указать все параметры.
- Как вы видите себе специализацию по указателям, ссылкам и структурным типам?
- Массив в шаблонном параметре редуцируется до указателя (как в функции).

https://godbolt.org/z/8jxPjn39f

Шаблонные шаблонные параметры

- Параметрами могут быть шаблоны классов

```
template <template<typename> typename Cont, typename Elt>
void print_size(const Cont<Elt> &a);
```

- Разумеется специализация по ним тоже возможна.
- Пока что кажется, что это переусложнение.

```
template <typename Container>
void print_size(const Container &a);
```

- Мы ещё вернёмся к этой технике.

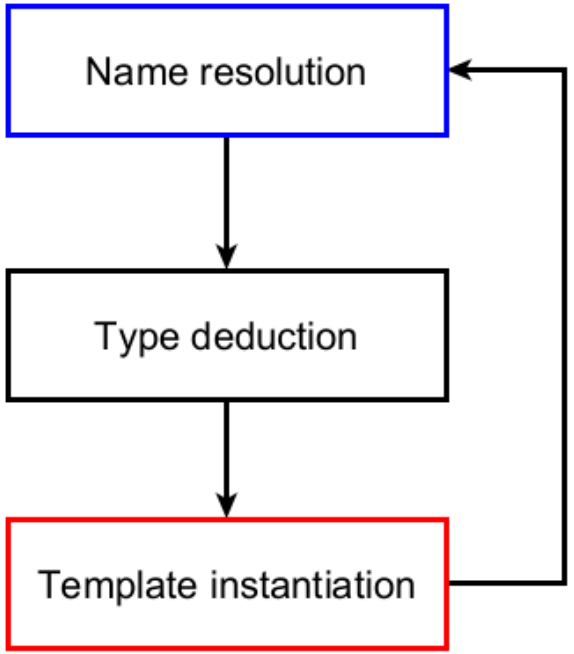
https://godbolt.org/z/f7MaYfvv1

Нормальный процесс

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

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

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

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

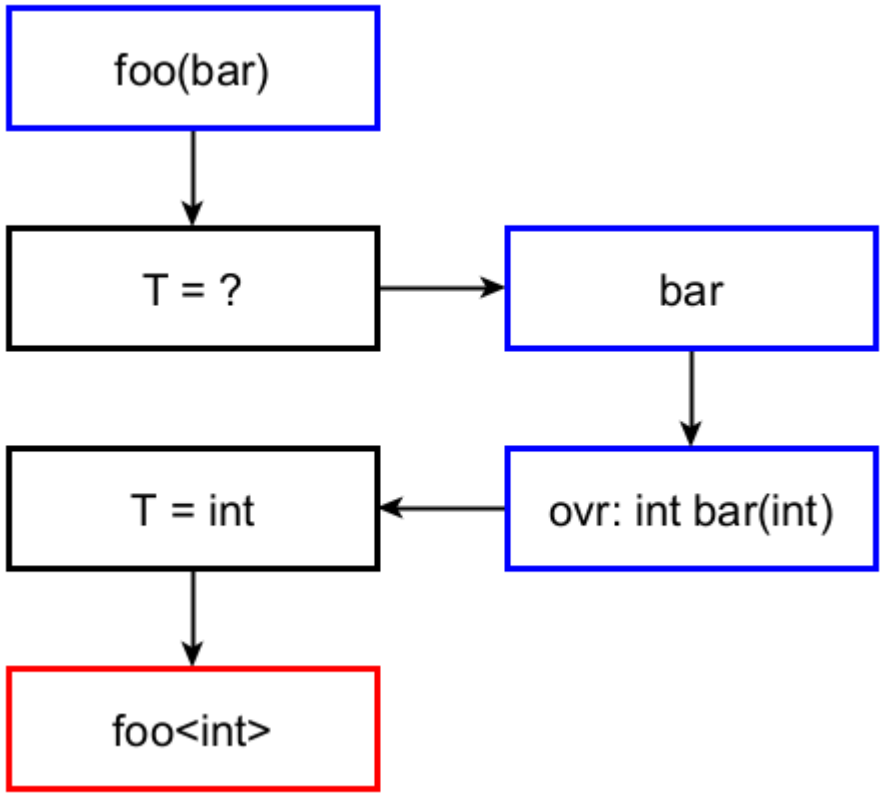


Взаимодействие процессов в языке

- Иногда вывод типов требует разрешения перегрузки.

```
template <typename T>
void foo(T (*p)(T));

int bar(char); // 1
int bar(int); // 2

foo(bar); // → 2
```

- Это создаёт не нормализованный семантический процесс.

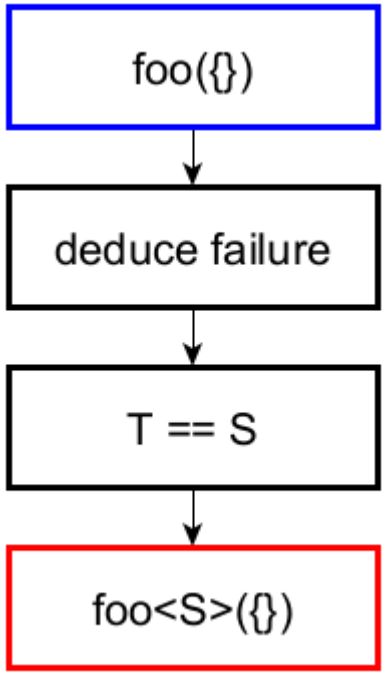


https://godbolt.org/z/fMTK1qjPc

Неудача вывода пробует подстановку

- Шаблонные параметры по умолчанию пробуются после того как вывод не удался.

```
struct S {
  S(int i = 0) {}
};

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

foo(0); // вывод T = int
foo({}); // подстановка T = S
```

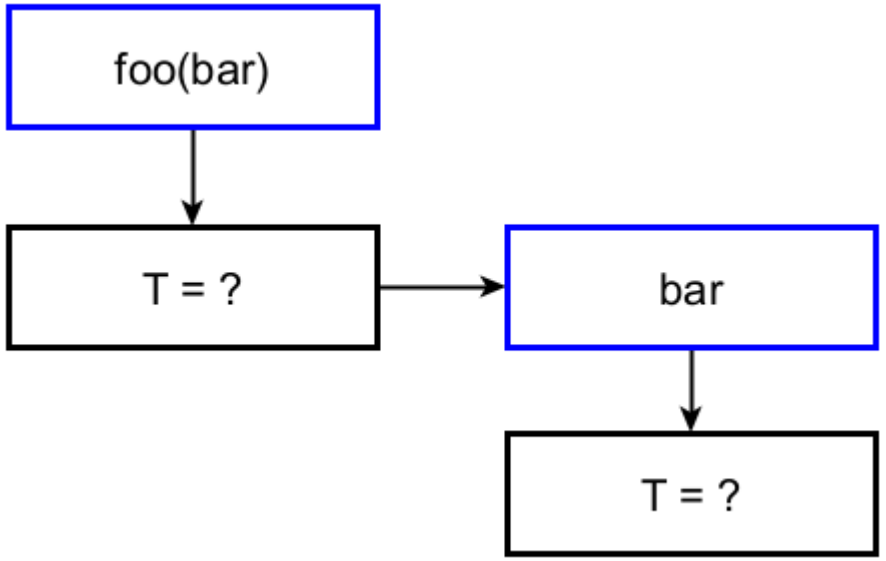


https://godbolt.org/z/Wac3K9PP9

Совсем ненормальный процесс

```
template <typename T>
void foo(T (*p)(T));

template <typename T = char>
int bar(T); // 1

int bar(int); // 2

foo(bar); // → ???
```

- Как вы думаете что тут произойдёт?

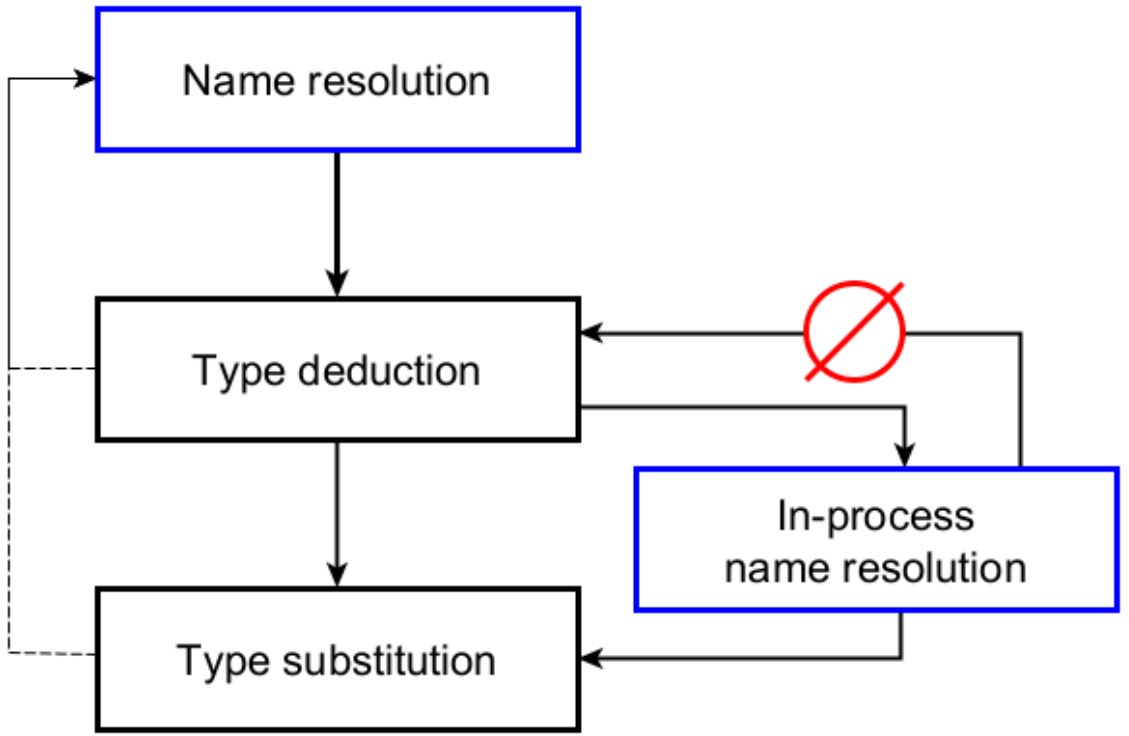


https://godbolt.org/z/Msddrz3fc

Запретный шаг вывода

```
template <typename T>
void foo(T (*p)(T));

template <typename T = char>
int bar(T); // 1

int bar(int); // 2

foo(bar); // → FAIL
```

- Если мы уже находимся в контексте вывода и нам снова требуется вывод, это всегда failure.



Что если вывод удался дважды?

```
template <typename T> int foo(T x); // 1
template <typename T> int foo(T* x); // 2
int **x = 0;
foo(&x); // ???
```


https://godbolt.org/z/sz3Pfox6q

Частичный порядок функций

```
template <typename T> int foo(T x); // 1
template <typename T> int foo(T* x); // 2
int **x = 0;
foo(&x); // → int **
```

- Подходят оба, но возьмётся **более специальный шаблон**.
- Значит нам нужен какой-то механизм установления более специального шаблона.

Как определить более специальную?

- С помощью вывода типов.
- Изобретаются типы T_1 и T_2 после чего вывод повторяется дважды.

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`*)`

`f(`T_2`*)` подходит `f(T)`, `f(`T_1`)` не подходит `f(T*)`

- Значит `f(T)` более общая чем `f(T*)`.

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/sd3zqzfjo

Что если вывод удался дважды?

- Вернемся к примеру когда вывод удался дважды.

```
template <typename T> int foo(T x);  // 1
template <typename T> int foo(T* x); // 2
```

- В точке вызова у нас нечто вроде.

```
int ***x = 0;
foo(&x); // → (1) > (2) поэтому выигрывает (2)
```

- Итак, перегрузку выиграл более специализированный шаблон.
- При этом термин "более специализированный" определяется через механизм вывода типов.

Некая двусмысленность в выводе

```
template <typename T> void foo(T);  // 1
template <typename T> void foo(T*); // 2
template <> void foo(int*);         // 3
int x;
foo(&x); // вызовет [3], и это в целом ok, но....
```

- Вопрос является (3) специализацией для (2) или для (1) не имеет смысла. Она одинаково хорошо подходит для обоих, поэтому специализирует выигравший перегрузку шаблон.
- В связи с этим могут возникать неприятные сюрпризы.

Контрпример Димова-Абрамса

```
template <typename T> void foo(T);  // 1
template <> void foo(int*);         // 2
template <typename T> void foo(T*); // 3
int x;
foo(&x); // вызовет [3], хотя [2] подходит лучше
```

- Важно помнить: специализации не участвуют в перегрузке. Сначала разрешается перегрузка, потом ищется наименее общая специализация.
- Но в данном случае (2) не специализирует (3), так как встречается раньше.
- В целом это аргумент против специализации.

Контрольный вопрос

```
template <typename T, typename U> void foo(T, U);   // 1
template <typename T, typename U> void foo(T*, U*); // 2
template <> void foo<int*, int*>(int*, int*);       // 3
int x;
foo(&x, &x); // ???
```


https://godbolt.org/z/d5v49dq5q

Обсуждение

- Итак, инстанцирование нетривиально взаимодействует с перегрузкой и выводом типов.
- Можем ли мы что-то сказать про взаимодействие инстанцирования с поиском имён?
- О да, мы можем.

Двухфазный поиск имён

- Как мы учим ему детей?
- Первая фаза: до инстанцирования. Шаблоны проходят общую синтаксическую проверку, а также разрешаются независимые имена.
- Вторая фаза: во время инстанцирования. Происходит специальная синтаксическая проверка и разрешаются зависимые имена.

```
template <typename T> struct Foo {
  int use1() { return illegal_name; } // независимое имя
  int use2() { return T::illegal_name; } // зависимое имя
};
```

- Разрешение зависимых имён откладывается до подстановки.

TS-проблема

```
template<typename T> void foo (T) { cout << "T"; }

struct S { };

template<typename T> void call_foo (T t, S x) {
  foo (x);
  foo (t);
}

void foo (S) { cout << "S"; }

void bar (S x) {
  call_foo (x, x); // что на экране?
}
```


https://godbolt.org/z/Mdos1Weeq

TS-проблема, всё ясно, на экране "TS"

```
template<typename T> void foo (T) { cout << "T"; }

struct S { };

template<typename T> void call_foo (T t, S x) {
  foo (x); // x независимое имя, разрешается в foo<S>(x)
  foo (t); // t зависимое имя, разрешение откладывается
}

void foo (S) { cout << "S"; }

void bar (S x) {
  call_foo (x, x); // здесь t разрешается в foo(S)
}
```


https://godbolt.org/z/Mdos1Weeq

TS-проблема, внезапно на экране "TT"

```
template<typename T> void foo (T) { cout << "T"; }

using S = int; // ой, всё

template<typename T> void call_foo (T t, S x) {
  foo (x); // x независимое имя, разрешается в foo<S>(x)
  foo (t); // t зависимое имя, разрешение откладывается
}

void foo (S) { cout << "S"; }

void bar (S x) {
  call_foo (x, x); // здесь t разрешается в foo(S)
}
```


https://godbolt.org/z/rsh9x33rv

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

- Мы легко выкручиваемся (type_identity доступно с C++20).

```
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

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
```

- Как вы думаете, в чём разница?

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
```

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

Перегрузка или специализация?

- Специализация уникальна в том, что позволяет избегать dummy аргументов.

```
template<class T> T zero();

template<> int zero() { return 0; }

template<> long zero() { return 0L; }
```

- Специализации не участвуют в выводе типов.

```
template<class T> T add(T a, T b);
template<> inline long add(long a, long b) { return a + b; }

int main() { return add(0, 1L); } // FAIL
```

Обсуждение

- Главное отличие функций от типов: у нас нет перегрузки типов

У нас нет перегрузки типов

```
template <typename T, typename... Ts>
struct chain : T, chain<Ts...> {
  chain(T t, Ts... ts) : T(t), chain<Ts...>(ts...) {}
};

template <typename T> struct chain : T {
  chain(T t) : T(t) {}
};

template <typename... T>
auto make_chain(T... t) { return chain<T...>(t...); }
```


https://godbolt.org/z/c8WvGh67s

У нас есть специализация

```
template <typename T, typename... Ts>
struct chain : T, chain<Ts...> {
  chain(T t, Ts... ts) : T(t), chain<Ts...>(ts...) {}
};

template <typename T> struct chain<T> : T {
  chain(T t) : T(t) {}
};

template <typename... T>
auto make_chain(T... t) { return chain<T...>(t...); }
```

Точки объявления (PoD)

- Ниже приведены два фрагмента кода.

```
int y = 2; { int y = y; }
```

```
int x = 2; { int x[x]; }
```

- Оба сомнительны, но в одном случае будет использовано значение переменной из внешней области видимости, а в другом случае нет.

Точки объявления (PoD)

- Ниже приведены два фрагмента кода.

```
int y = 2; { int y /* PoD */ = y; }
```

```
int x = 2; { int x[x] /* PoD */; }
```

- Оба сомнительны, но в одном случае будет использовано значение переменной из внешней области видимости, а в другом случае нет.
- Точка объявления [basic.scope.pdecl] это точка в которой объявление завершено (после полного объявления но перед инициализатором).
- До точки объявления имя не считается введённым в область видимости (соответственно используется имя из прошлой области видимости).

https://godbolt.org/z/5Tx4vExs1

Шаблоны классов

- Обобщённый тип, задаваемый шаблоном считается объявленным в PoD.

```
template <typename T> struct fwnode /* POD */ {
  T data_;
  fwnode<T> *next_;
};
```

- В теле таким образом может быть использован указатель или ссылка на себя (как на неполный тип).

Шаблоны классов: упрощение имён

- Обобщённый тип, задаваемый шаблоном считается объявленным в PoD.

```
template <typename T> struct fwnode /* POD */ {
  T data_;
  fwnode *next_;
};
```

- В теле таким образом может быть использован указатель или ссылка на себя (как на неполный тип).
- Для удобства, шаблонные параметры рядом с именем можно не указывать (только внутри тела класса).

Зависимые имена внутри шаблонов

- Нет никаких проблем (как с функциями) в использовании вторичной типизации.

```
template <typename T> class Stack {
  fwnode<T> *top_;
  // всё остальное в стеке для объектов типа T общего вида
};
```

- Такой стек (построенный на односвязном списке, что вполне реалистично) ипользует fwnode. Параметр обязателен.

Порождение классов из шаблонов

```
template <typename T> class Stack {
  fwnode<T> *top_;
public:
  push(T x);
};

class Stack<int> {
  fwnode<int> *top_;
public:
  push(int x);
};

Stack<int> s;
```

Проблема: слишком общие шаблоны

- Кажется этот стек не слишком эффективен для хранения целых чисел.

```
template <typename T> class Stack {
  fwnode<T> *top_;
  // всё остальное в стеке для объектов типа T общего вида
};
```

- Для целых чисел было бы проще хранить массив.

Специализация

- Кажется этот стек не слишком эффективен для хранения целых чисел.

```
template <typename T> class Stack {
  fwnode<T> *top_;
  // всё остальное в стеке для объектов типа T общего вида
};
```

- Для целых чисел было бы проще хранить массив.

```
template <> class Stack<int> {
  int *content_;
  // всё остальное в стеке для целых чисел
};
```

Обсуждение

- Как и для функций ручная специализация уже инстанцированного типа не имеет смысла.

```
template <typename T> class Invalid {};

// тут уже произошло инстанцирование
Invalid <int> i;

// эта специализация не имеет смысла и запрещена
template <> class Invalid<int> {};
```

- Опасайтесь специализаций, происходящих внутри единицы трансляции.

Тут демонстрация danger

Частичная специализация

- Кажется этот стек не слишком эффективен для хранения всех указателей

```
template <typename T> class Stack {
  fwnode<T> *top_;
  // всё остальное в стеке для объектов типа T общего вида
};
```

- Для указателей было бы проще хранить массив указателей

```
template <typename T> class Stack<T*> {
  T **content_;
  // всё остальное в стеке для указателей
};
```

Упрощение имён в специализациях

- Рассмотренное ранее упрощение имён отлично работает в (частичных) специализациях.

```
template <typename T> class A {
  A* a1; // A здесь означает A<T>
};

template <typename T> class A<T*> {
 A* a2; // A здесь означает A<T*>
};
```

- Разумеется это опционально. Указывать полные имена – вполне легально (и часто это отличная идея).

Примеры частичной специализации

```
template <typename T, typename U> class Foo {};          // 1
template <typename T> class Foo<T, T> {};                // 2
template <typename T> class Foo<T, int> {};              // 3
template <typename T, typename U> class Foo<T*, U*> {}; // 4
Foo<int, float> mif;   // 1
Foo<float, float> mff; // 2
Foo<float, int> mfi;   // 3
Foo<int*, float*> mp;  // 4
```


https://godbolt.org/z/rxedevMv6

Немного расширения сознания

```
template <typename T> struct X {
  void print() { std::cout << "forall" << std::endl; }
};

template <typename T> struct X<vector<T>> {
  void print() { std::cout << "forvec" << std::endl; }
};

X<int> a;
X<vector<int>> b;

a.print();
b.print();
```


https://godbolt.org/z/fGsceo6gd

Ограничения специализации

- Специализация (как и для функций) всегда должна в коде следовать за объявлением шаблона общего вида.
- Частично специализированный шаблон должен быть действительно менее общим, чем тот, версией которого он является.

```
template <typename T> class X { /* .... */ };

template <typename U> class X<U> { /* .... */ }; // ошибка
```

Менее специальна, менее обобщена?

- Частичная специализация съедает какое-то количество параметров primary шаблона.
- До тех пор пока она менее специальна в некоем смысле, у неё может быть даже **больше** шаблонных параметров.

```
template <typename T> class X;

template <typename R, typename Arg> class X<R(Arg)> {}
```

- Как вы думаете что означает менее специальная для специализации?

Частичный порядок специализаций

- Частичный порядок может быть введён для специализаций шаблона.

```
1. template <typename T> class Foo { /* .... */ };
2. template <typename T> class Foo<T*> { /* .... */ };
3. template <typename T> class Foo<T**> { /* .... */ };
```

- Он разрешается через сведение к перегрузке функций и потом выводу типов.

```
template<typename T> void f_foo(Foo<T>);    f_foo(Foo<T_1>)
template<typename T> void f_foo(Foo<T*>);   f_foo(Foo<T_2*>)
template<typename T> void f_foo(Foo<T**>);  f_foo(Foo<T_3**>)
```

Ограничения специализации

- Специализация всегда должна в коде следовать за объявлением шаблона общего вида.
- Специализированный шаблон должен быть действительно менее общим, чем тот, версией которого он является.
- Полная специализация возможна и для классов и для функций, наряду с перегрузкой.
- Частичная специализация для функций невозможна.

```
template <typename T> void foo (T x);
template <> void foo<int> (int x); // ok
template <typename T> void foo<T*> (T* x); // fail
```

Обсуждение

- Частичная специализация функций запрещена.
- Можно ли сымитировать частичную специализацию функций через частичную специализацию классов?

Трюк Саттера

- Частичная специализация функций запрещена.
- Можно ли сымитировать частичную специализацию функций через частичную специализацию классов?

```
template <typename T> struct FImpl;

template <typename T> void f(T t) { FImpl<T>::f(t); }

template <typename T> struct FImpl {
  static void f(T t);
};
```

- Здесь используется то, что статический метод stateless класса мало отличается от свободной функции.

https://godbolt.org/z/q89PTjn1c

Шаблоны членов: простая задача

```
struct Data {
  template <typename T> T read() const;
};

template <typename T>  class DataReader {
  const T& source_;
public:
  DataReader(const T& s) : source_ (s) {}
  template <typename R> R read(); // вызывает source_.read()
};

// тут необходимо написать определение DataReader::read
```

Первый вариант решения

```
template <typename T> class DataReader {
  const T& source_;
public:
  template <typename R> R read();
};

template <typename T>
template <typename R>
R DataReader<T>::read() {
  R res = source_.read<R>(); // синтаксическая неоднозначность
  return res;
}
```

Разрешение неоднозначности

```
template <typename T> class DataReader {
  const T& source_;
public:
  template <typename R> R read();
};

template <typename T>
template <typename R>
R DataReader<T>::read() {
  R res = source_.template read<R>();
  return res;
}
```


https://godbolt.org/z/Mvz9GMxM9

Более сложная задача

```
template <typename T>  class DataReader {
  const T& source_;
public:
  template <typename R> R read();
};

// тут необходимо написать специализацию
// для DataReader<T>::read<string>
```

Попытка решения

```
template <typename T>  class DataReader {
  const T& source_;
public:
  template <typename R> R read();
};

// код ниже не работает, он иллюстрирует возможный ход мыслей

template <typename T> template <>
string DataReader<T>::read<string>() const {
  string foo = source_.template read<string>();
  return foo;
}
```

Правильный ответ

```
template <typename T> class DataReader {
  const T& source_;
public:
  template <typename R> R read();
};
```

- Требуемая специализация невозможна, так как она означала бы частичную специализацию метода.
- Если у нас есть конкретная структура Data, то можно написать полную специализацию для `DataReader<Data>::read<string>`.

Пример полной специализации

```
struct Data { template <typename T> T read() const; };

template <typename T> class DataReader {
  const T& source_;
public:
  template <typename R> R read();
};

template <>
template <> // это не опечатка!
string DataReader<Data>::read<string>() {
  return source_.template read<string>();
}
```


https://godbolt.org/z/W1rz05M53

Задача: параметризация методов

```
template <typename T1, typename T2> struct A {
  void func();
};
```

- Необходимо добиться следующего эффекта:

```
A <int, double> a;
A <float, double> b;

a.func(); // for int
b.func(); // for all
```

- Т.е. параметризовать метод первым аргументом шаблона.
- Задачу усложняет то, что частичная специализация для методов невозможна.

Параметризация: первая попытка

```
template <typename T1, typename T2> struct A {
  void func() {
    T1 dummy; internal_func(dummy);
  }
private:
  template <typename V> void internal_func(V) { cout << "all\n"; }
  void internal_func(int) { cout << "int\n"; }
};

A <int, double> a;
A <float, double> b;

a.func(); // for int, благодаря разрешению перегрузки
b.func(); // for all
```

Переходники типов

```
template <typename T> struct Type2Type {
  typedef T OriginalType;
};
```

- Минимальный размер.
- Прозрачность.
- Номинативная типизация.

Параметризация: переходники типов

```
template <typename T1, typename T2> struct A {
  void func() {
    internal_func(Type2Type<T1>());
  }
private:
  template <typename V> void internal_func(V) { cout << "all\n"; }
  void internal_func(Type2Type<int>) { cout << "int\n"; }
};

A <int, double> a;
A <float, double> b;

a.func(); // for int, благодаря разрешению перегрузки
b.func(); // for all
```


https://godbolt.org/z/YaEnnrzd8

Обсуждение

- Переходники Type2Type изначально были придуманы Александреску для ещё одной имитации частичной специализации функций.
- Он рассматривал шаблонную функцию-"конструктор"

```
template <typename T, typename U> T* Create(const U& arg);
```

- И решал задачу специализации его по первому аргументу. Как-то вот так:

```
template <typename U>
 Widget* Create<Widget, U>(const U& arg); // not C++
```

- Разумеется такое тупое решение не сработает
- Кто уже понял при чём здесь переходники типов?

https://accu.org/index.php/journals/385

Обсуждение

- Переходники Type2Type изначально были придуманы Александреску для ещё одной имитации частичной специализации функций.
- Решение просто и впечатляюще.

```
template <typename T, typename U>
T* Create(const U& arg, Type2Type<T>);

template <typename U>
Widget* Create(const U& arg, Type2Type<Widget>);
```

- Накладывает некоторые обязательства и есть куда покритиковать.

Задачи, часть 1

- gcc 14.2 и clang 19.1 расходятся в трактовке следующего кода. Кто прав?

```
template <typename T> struct A {
  using M = T; // what if using M = int?
  struct B { using M = void; struct C; };
};

template <typename T> struct A<T>::B::C : A<T> {
   M m; // void or int?
};

int main() {
  A<int>::B::C x;
}
```


https://godbolt.org/z/zqbsbodT4

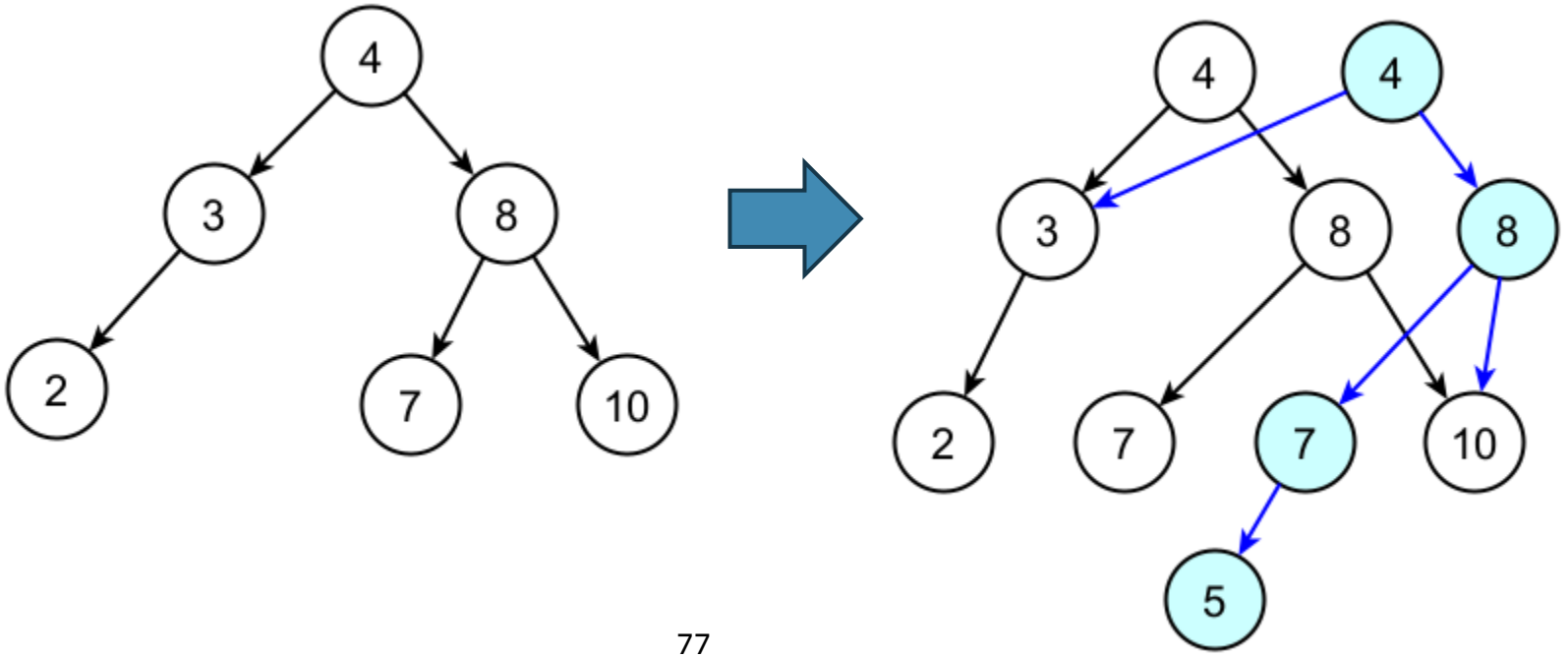
Задачи, часть 2

```
struct S; // Неполный тип

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

Wrapper<S>& get();

int main() {
  (foo)(get()); // OK?
```

- Обоснуйте со ссылками на стандарт всё ли тут хорошо?

Задачи, часть 3

- Реализуйте шаблонный класс для перманентного множества (можете не думать о балансировке дерева). Можно ли выиграть частичной специализацией для какого-то простого типа?



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