

ОБОБЩЕНИЯ ТИПОВ

Специализация, инстанцирование и вывод типов

К. Владимиров, Intel, 2022
mail-to: konstantin.vladimirov@gmail.com

➤ Специализация и инстанцирование

- ❑ Разрешение имён

- ❑ Вывод типов

- ❑ Свертка и проброс ссылок

Инстанцирование

- Инстанцирование это процесс порождения 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.

Инстанцирование и специализация

- Явная специализация может войти в конфликт с инстанцированием

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

```
// ОК, 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: ODR violation
```

```
template <> int max(int x, int y) { return 42; }
```

Удаление специализаций

- Частным случаем явной специализации является запрет специализации

```
// 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;
```

- Подобным образом можно удалять и перегрузки

```
void foo(char*) = delete;
void foo(void*) = delete;
```

Специализация по nontype параметрам

- Нет никаких проблем в том, чтобы специализировать класс по любой разновидности шаблонных параметров.
- Например по целым числам.

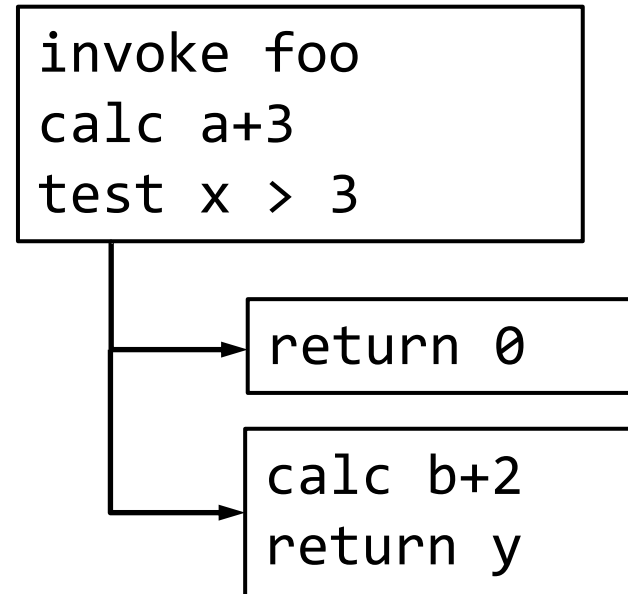
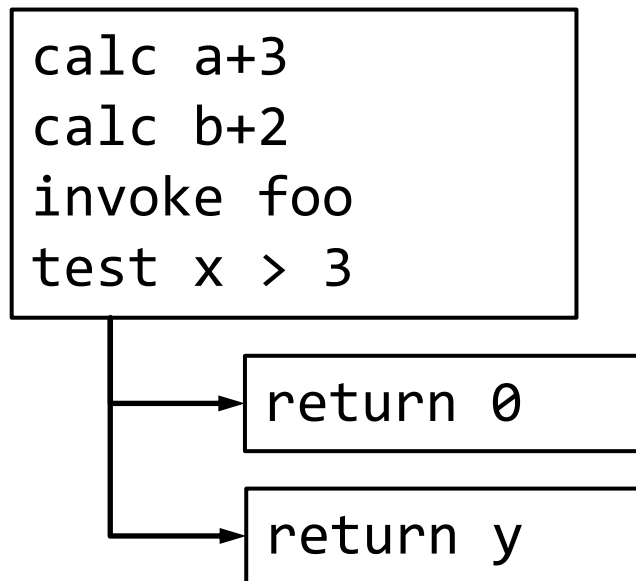
```
template <typename T, int N> class Array;
```

```
template <typename T> class Array<T, 3> {  
    // тут более эффективная реализация для трёх элементов
```

- Немного сложнее придумать разумный пример специализации по указателям и ссылкам, можете подумать дома.

Ленивость и энергичность

```
int foo (int x, int y) { return (x > 3) ? 0 : y; }  
foo (a + 3, b + 2);
```



Инстанцирование – ленивый процесс

- Ниже если бы инстанцирование было энергичным, была бы ошибка

```
template <int N> struct Danger {  
    using block = char[N]; // ошибка если N меньше нуля  
};  
  
template <typename T, int N> struct Tricky {  
    void test_lazyness() { Danger<N> no_boom_yet; }  
};  
  
int main() {  
    Tricky<int, -2> ok; // ошибка только при ok.test_lazyness()  
}
```

- Но в данном случае инстанцировалось ровно то, что мы попросили

Явное инстанцирование

- Неявное инстанцирование компилятор проводит где захочет.
- Но вы можете взять точку инстанцирования под контроль.

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

```
template int max<int>(int x, int y); // инстанцировать тут
```

- Вы можете (и часто должны) также заблокировать инстанцирование в остальных модулях, указав, что оно уже проведено где-то ещё.

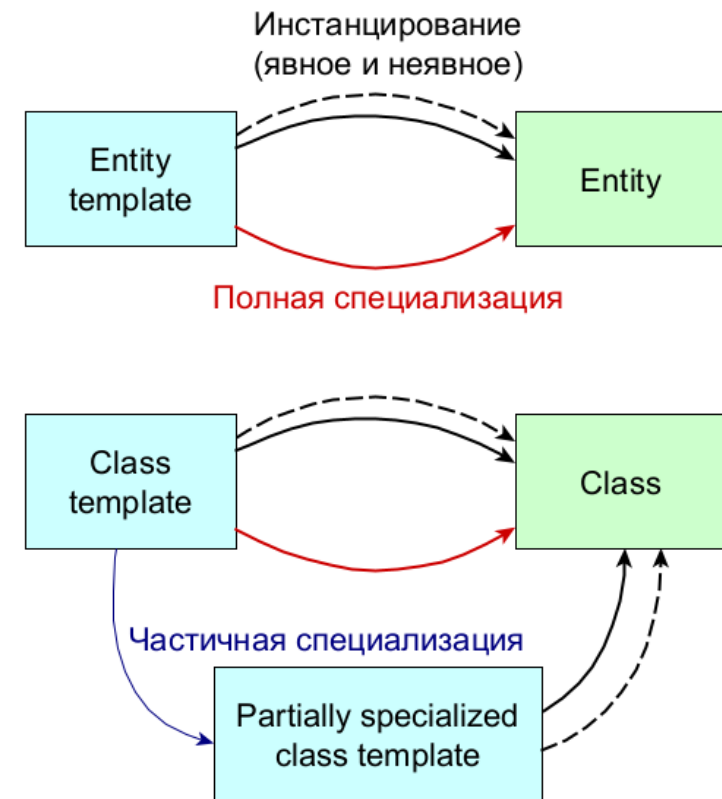
```
extern template double max<double>(double x, double y);
```

- При явном инстанцировании вы лишаетесь ленивого поведения.

Частичная специализация

- Для классов доступна также возможность специализировать шаблон частично.

```
template <typename T, typename U>  
class Foo {}; // primary template  
  
template <typename T>  
class Foo<T, T> {}; // case T == U  
  
template <typename T>  
class Foo<T, int> {}; // case U == int  
  
template <typename T, typename U>  
class Foo<T*, U*> {}; // case pointers
```



Специализация для похожих типов

- Частичная специализация возможна по семейству похожих типов.

```
template <typename T> struct X;
```

```
template <typename T> struct X<std::vector<T>>;
```

```
X<int> a; // → primary template X<T>
```

```
X<std::vector<int>> b; // → X<std::vector<T>>
```

- Примерно так же можно специализировать для всех функций

```
template <typename T> struct Y;
```

```
template <typename R, typename T> struct Y<R(T)>;
```

Упрощение имён в специализациях

- Внутри основного шаблона класса мы всегда можем сокращать имя.

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

- Это отлично работает также внутри частичной специализации.

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

- Разумеется указывать полные имена вполне легально (и часто лучше читается).

Case study: unique_ptr

- Рассмотрим следующее использование unique_ptr

```
std::unique_ptr<int> ui{new int[1000]()}; // грубая ошибка
```

- В чём по вашему состоит грубая ошибка?
- Можем ли мы добавить к чему-то частичную специализацию, чтобы как-то предложить законный метод делать такие вещи?

```
std::unique_ptr<int[]> ui{new int[1000]()}; // хотелось бы так
```

- Хорошая ли идея добавлять частичную специализацию к самому классу unique_ptr?

Вспоминаем структуру unique_ptr

- Удаление отделено в параметр шаблона.

```
template <typename T, typename Deleter = default_delete<T>>
class unique_ptr {
    T *ptr_;
    Deleter del_;
public:
    unique_ptr(T *ptr = nullptr, Deleter del = Deleter()) :
        ptr_(ptr), del_(del) {}

    ~unique_ptr() { del_(ptr_); }
    // и так далее
```

- Вспоминаем как мог бы выглядеть default_delete?

Частичная специализация

- На помощь приходит **частичная специализация для массивов**

```
template <typename T> struct default_delete {  
    void operator()(T *ptr) { delete ptr; }  
};
```

```
template <typename T> struct default_delete<T[]> {  
    void operator()(T *ptr) { delete [] ptr; }  
};
```

- Теперь при массиво-подобном T у нас будет вызван правильный deleter

Обсуждение

- Можно ли шаблонную специализацию назвать разновидностью наследования?
- В наследовании тоже более специализированный класс наследует более общему.

Нарушение LSP для шаблонов

- Увы, но (частично) специализированный шаблон может не иметь ничего общего с его полной версией (вплоть до разных имен методов).
- С точки зрения наследования это нарушение LSP.

```
template <typename T> struct S { void foo(); };
```

```
template <> struct S<int> { void bar(); };
```

```
S<double> sd; sd.foo(); // → primary template S<T>
```

```
S<int> si; si.bar();    // → specialization S<int>
```

- И, разумеется, шаблоны инвариантны к шаблонной генерализации. Каждая специализация считается новым, не связанным с прочими, типом.

Обсуждение

- Рассмотрим вызов `si.bar()` внутри шаблонной функции

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

- Учитывая ленивость подстановки и возможность специализаций, в какой момент компилятор должен принять решение валиден ли этот вызов?

❑ Специализация и инстанцирование

➤ Разрешение имён

❑ Вывод типов

❑ Свертка и проброс ссылок

Постановка проблемы

- Должно ли разрешение имён в шаблонах (в том числе классов) происходить до инстанцирования или после?

```
template <typename T> struct Foo {  
    int use() { return illegal_name; }  
};
```

- Здесь `illegal_name` выглядит нелегальным именем, но может быть оно будет как-то легализовано после того как будет подставлен конкретный `T`?
- Нужно ли выдавать ошибку сразу или подождать подстановки параметра?

Двухфазное разрешение имён

- 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
- Зависимое имя — это имя, которое семантически зависит от шаблонного параметра. Шаблонный параметр может быть его типом, он может участвовать в формировании типа и так далее

```
template <typename T> struct Foo {  
    int use() { return illegal_name; } // независимое имя, ошибка  
};
```

Двухфазное разрешение имён

- 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
- Зависимое имя — это имя, которое семантически зависит от шаблонного параметра. Шаблонный параметр может быть его типом, он может участвовать в формировании типа и так далее

```
template <typename T> struct Foo {  
    int use() { return T::illegal_name; } // зависимое имя, ок  
};
```

Двухфазное разрешение имён

- 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
- Зависимое имя — это имя, которое семантически зависит от шаблонного параметра. Шаблонный параметр может быть его типом, он может участвовать в формировании типа и так далее
- Следует запомнить золотое правило: **разрешение зависимых имён откладывается до подстановки шаблонного параметра**

Пример Вандерворда

- Можем ли мы как-то исправить ситуацию?

```
template <typename T> struct Base {  
    void exit();  
};
```

```
template <typename T> struct Derived : Base<T> {  
    void foo() {  
        exit(); // можно подумать, что это Base::exit()  
                 но exit — не зависимое имя, так что нет.  
    }  
};
```


Пример Вандерворда

- Есть несколько способов сделать имя `exit` зависимым.

```
this->exit();
```

```
Base::exit(); // читается как Base<T>::exit();
```

- Это одно из немногих рациональных использований явного `this`.

```
template <typename T> struct Derived : Base<T> {  
    void foo() {  
        this->exit(); // ага, мы стреляем в двухфазное разрешение
```

- Хочется ещё раз призвать не использовать явный `this` нерационально.

Зависимые имена типов

- Зависимые имена типов могут вызывать неожиданные проблемы

```
struct S {  
    struct subtype {};  
};  
  
template <typename T> int foo(const T& x) {  
    T::subtype *y;  
    // и так далее  
}  
  
foo<S>(S{}); // казалось бы всё хорошо?
```

Зависимые имена типов

- Зависимые имена типов могут вызывать неожиданные проблемы

```
struct S {  
    struct subtype {};  
};
```

```
template <typename T> int foo(const T& x) {  
    typename T::subtype *y;  
    // и так далее  
}
```

```
foo<S>(S{}); // теперь всё хорошо
```

- Эта техника называется устранением неоднозначности (disambiguation)

Зависимые имена шаблонов

- Зависимые имена шаблонов также могут вызывать неожиданные проблемы

```
template<typename T> struct S {  
    template<typename U> void foo(){}  
};
```

```
template<typename T> void bar() {  
    S<T> s; s.foo<T>();  
}
```

- Тут, как вы думаете, что-то не так или всё ок?

Зависимые имена шаблонов

- Зависимые имена шаблонов также могут вызывать неожиданные проблемы

```
template<typename T> struct S {  
    template<typename U> void foo(){}  
};
```

```
template<typename T> void bar() {  
    S<T> s; s.template foo<T>();  
}
```

- Без разрешения неоднозначности первая треугольная скобка означала бы оператор меньше
- Вместе: `typename T::template iterator<int>::value_type v;`

Обсуждение

- Итак, для разрешения имён нужно иметь информацию о типах.
- Нельзя ли использовать эту информацию для вывод типов?

- ❑ Специализация и инстанцирование

- ❑ Разрешение имён

- Вывод типов

- ❑ Свертка и проброс ссылок

Обсуждение

- Вернемся к примеру с функцией `max`

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

....

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

- Компилятор видит тип `int` для литералов, поэтому его явное указание не нужно

```
a = max(2, 3); // тоже ок
```

```
a = max(2, 3.0); // неоднозначность, вывод типов не сработает
```

```
a = max<int>(2, 3.0); // тоже ок, мы помогли компилятору
```


Неуточнённые типы

- По исторической традиции вывод неуточнённого типа режет ссылки, константность и прочее

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

```
const int &b = 1, &c = 2;
```

```
a = max(b, c); // → template<> int max<int>(int, int)
```

- Это сделано чтобы уменьшить число неоднозначностей

```
int e = 2; int &d = e; // вроде разные типы, но вывод работает
```

```
a = max(d, e); // → template<> int max<int>(int, int)
```

Уточнённые типы

- Всё меняется когда мы уточняем тип левой ссылкой или указателем.

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

- Теперь компилятор считает, что программисту виднее.

```
const int x = 42;  
foo(x); // → template<> void foo<const int>(const int& x)
```

- Интересно, что иногда вы вроде уточнили, а компилятор... срезал уточнение.

```
template <typename T> void bar(const T x);  
bar(x); // → template<> void bar<int>(int x)
```

- Особая статья это уточнение правой ссылкой, это мы пока отложим.

Вывод конструкторами классов (C++17)

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

```
template<typename T> struct container {  
    container(T t);  
    // и так далее  
};  
  
container c(7); // → container<int> c(7);
```

- Внезапно будет работать также списочная инициализация но пока неясно как.

```
std::vector v {1, 2, 3}; // → std::vector<int>
```

Проблема: вывод через косвенность

- Конструктор класса сам может быть шаблонным

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

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

- Компилятор умён, но не **настолько** умён чтобы сходить в `std::iterator_traits`
- Тут надо как-то ему подсказать где искать `value_type`.

Хинты для вывода (C++17)

- Пользователь может помочь выводу в сложных случаях

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

```
// пользовательский хинт для вывода
```

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

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

Вывод без конструктора

- Агрегатное значение может и не иметь конструктора

```
template <typename T> struct NamedValue {  
    T value;  
    std::string name;  
};
```

- Также можно немного помочь компилятору.

```
NamedValue(const char*, const char*) -> NamedValue<std::string>;
```

- Теперь конструируем агрегат из двух строк.

```
NamedValue n{"hello", "world"}; // → NamedValue<std::string>
```

Обсуждение

- Мы хотим такой же гибкости для локальных переменных?

Встречаем auto и decltype

- Для локальных переменных ключевое слово auto работает по правилам вывода типов шаблонами.

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

```
const int &t;
```

```
foo(t); // → foo<int>(int x)
```

```
auto s = t; // → int s
```

- Для точного вывода существует decltype

```
decltype(t) u = 1; // → const int& u
```


Категории выражений

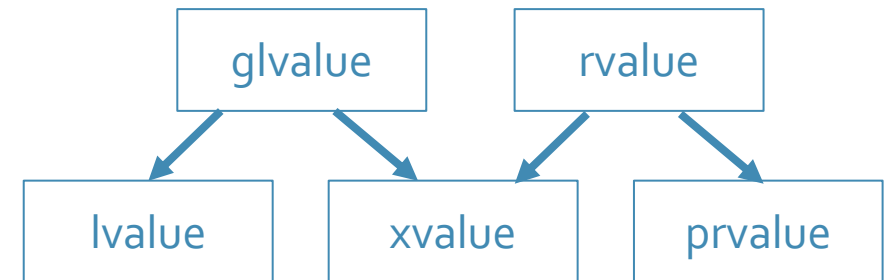
- Любое выражение в языке относится к одной из категорий

`int x, y;`

`x = x + 1;` `x = x;`
lvalue prvalue lvalue lvalue to prvalue

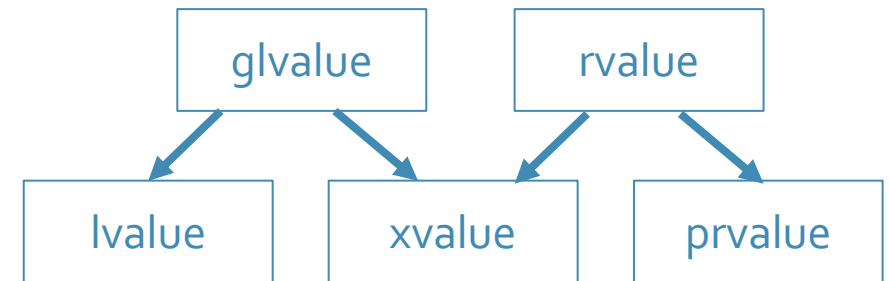
`y = std::move(x);`
lvalue xvalue

- Есть две обобщающие категории: glvalue и xvalue



Четыре формы decltype

- decltype существует в двух основных видах: для имени и для выражения
- decltype(name) выводит тип с которым было объявлено имя
- decltype(expression) работает чуточку сложнее
 - decltype(lvalue) это тип выражения + левая ссылка
 - decltype(xvalue) это тип выражения + правая ссылка
 - decltype(prvalue) это тип выражения



- В итоге левые или правые ссылки встречаются в неожиданных местах.

```
int a[10]; decltype(a[0]) b = a[0]; // → int& b
```

- Это может выглядеть странно, но это логично – ссылка определяет lvalueness

Проблема в C++11

- Итак, мы в 2012 году и у нас нет auto для возвращаемого типа функций

```
template <typename T> auto // C++11 Error!  
makeAndProcessObject (const T& builder) {  
    auto val = builder.makeObject();  
    // что-то делаем с val  
    return val;  
}
```

- Как написать эту функцию в реалиях 2012 года?

Попытка решения

- На самом деле эта проблема сохраняется в свежих версиях стандарта, но её стало сложнее демонстрировать
- Итак, мы в 2012 году и у нас нет `auto` для возвращаемого типа функций

```
template <typename T> decltype(builder.makeObject()) // Fail
makeAndProcessObject (const T& builder) {
    auto val = builder.makeObject();
    // что-то делаем с val
    return val;
}
```

- Это не работает, так как имя `builder` ещё не введено в область видимости.

Решение для C++11

- Для решения используется так называемый расширенный синтаксис.

```
int foo(); // обычный синтаксис  
auto foo() -> int; // расширенный синтаксис
```

- Использование очевидно

```
template <typename T>  
auto makeAndProcessObject(const T& builder) ->  
    decltype (builder.makeObject()) {  
    auto val = builder.makeObject();  
    // что-то делаем с val  
    return val;  
}
```

Решение для C++14 и позднее

- Для статического решения можно использовать нефиксированную сигнатуру.

```
int foo (); // функция с фиксированной сигнатурой  
auto foo(); // функция для которой возвращаемый тип выводится
```

- Использование также несложно

```
template <typename T>  
auto makeAndProcessObject (const T& builder) {  
    auto val = builder.makeObject();  
    // что-то делаем с val  
    return val;  
}
```

Use before deduction

- Бывают случаи когда такой вывод сбивается

```
auto bad_sum_to(int i) {  
    // use before deduction  
    return (i > 2) ? bad_sum_to(i-1) + i : i;  
}
```

- Для этой ошибки вовсе не обязательна рекурсия

```
auto func();  
  
int main() { func(); } // use before deduction  
  
auto func() { return 0; } // deduction
```

Обсуждение

- Кажется ли вам хорошей идеей нефиксированная сигнатура для внешних API, например для методов классов в общих хедерах?
- Именно поэтому даже сейчас форма со стрелочкой используется когда мы знаем как именно формируется тип.

// фиксированная сигнатура если всё внутри decltype известно
`auto foo() -> decltype(some information);`

- Бывает также абсурдное использование этой формы просто для красоты.

`auto main() -> int { return 42; } // ошибки тут нет, но....`

Идиома for-auto

- Обход итератором начиная с C++11 скрыт за for-auto идиомой
- Допустимый вариант

```
for (auto it = v.begin(), ite = v.end(); it != ite; ++it)  
    use(*it);
```

- Эквивалентный (почти эквивалентный) вариант

```
for (auto elt : v)  
    use(elt);
```

- Что если use берёт ссылку? Первый вариант отдаст ссылку перевязав её. Второй вариант, увы, срежет тип и, значит, скопирует значение

Обсуждение: AAA initializers

- Предложенный Гербом Саттером принцип AAA состоит в том, чтобы делать любую инициализацию через `auto`

```
auto x = 1;  
auto y = 1u;  
auto c = Customer{"Jim", 42};  
auto p = v.cbegin();
```

- Начиная с C++17 он действительно работает (вспоминаем prvalue elision)

```
auto a = std::atomic<int>{9}; // ок только в C++17  
auto arr = std::array<int, 100>{}; // быстро с C++17
```

- Некоторая критика этого принципа основана на сложности чтения кода.

Проблемы с ААА

- Первое: не следует тянуть ААА в нестатические функции. Эта идиома **только** для инициализации **локальных переменных**

```
auto foo(int x); // non-fixed ABI (from C++14)
int foo(auto x); // non-fixed ABI (from C++20)
```

- Второе: есть случаи когда это всё ещё не работает

```
auto x = long long {42}; // FAIL
auto x = static_cast<long long>(42); // ok, but...
```

```
const int & foo();
```

```
auto x = foo(); // decays
auto x = static_cast<const int&>(foo()); // still decays
```

- ❑ Специализация и инстанцирование
- ❑ Разрешение имён
- ❑ Вывод типов
- Свертка и проброс ссылок

Вывод типов из ссылочных типов

- Рассмотрим вывод типов с помощью auto

```
int x;  
int &y = x;  
auto && d = move(y); // → ???
```

- Уточнённое с помощью rvalue reference, auto не может игнорировать ссылку
- Формально вывод выглядит так:

```
auto &&c = y;           // → int & && c = y;  
auto &&d = move(y);     // → int && && d = move(y);
```

Чтобы получился корректный тип, ссылки должны быть свёрнуты (collapsed).

Правила свёртки ссылок

- Левая ссылка выигрывает, если она есть
- Для предыдущего примера это даёт

```
auto && c = y;           // → int & && c = y;  
                        // → int &c = y;  
auto && d = move(y);     // → int && && d = move(y);  
                        // → int &&d = move(y);
```

Inner	Outer	Result
T&	T&	T&
T&	T&&	T&
T&&	T&	T&
T&&	T&&	T&&

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

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

Универсальность ссылок

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

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

- Это в целом работает и для `decltype` и для шаблонов (но для шаблонов есть одна техническая трудность)

```
decltype(x) && z = x; // int &z = x;
```

```
template <typename T> void foo(T&& t);  
foo(x); // foo<??>(int& t) как вы думаете, чему равен T?
```

- Такие ссылки называют **forwarding references** или **универсальными ссылками**

Небольшое уточнение

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

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

```
int x;
```

```
const int y = 5;
```

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

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

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

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

Неуниверсальные ссылки

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

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

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

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

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

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

Идиома for-auto&&

- Теперь мы знаем ответ на поставленный ранее вопрос

- Допустимый вариант

```
for (auto elt : v)  
    use(elt);
```

- Куда лучший вариант

```
for (auto && elt : v) // elt это T& или T&&  
    use(elt);
```

- Он лишён недостатков, которые мы замечали ранее

Обсуждение: AAARR

- Almost Always Auto Ref Ref это расширение идиомы AAA, отлично справляющееся с большинством случаев

```
auto&& y = 1u;  
auto&& c = Customer{"Jim", 42};  
auto&& p = v.cbegin();  
  
const int& foo();  
auto&& f = foo(); // ok, const int& inferred
```

- **Что вы думаете про AAARR?**

Прозрачная оболочка

- Представим теоретическую задачу сделать функцию максимально "прозрачной" то есть пробрасывающей свои аргументы без расходов

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

- Начнём с простейшего вопроса: что она возвращает?
- Функция может возвращать как правую, так и левую ссылку.

Знакомимся: decltype(auto)

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

```
double x = 1.0;
```

```
decltype(x) tmp = x; // два раза x не нужен
```

```
decltype(auto) tmp = x; // это именно то, что нужно
```

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

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

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

Обсуждение

- Пожалуйста не пользуйтесь этой штукой если абсолютно не уверены.

Прозрачная оболочка

- Кажется для прозрачной оболочки это идеально подойдёт

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

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

```
extern Buffer foo(Buffer x);
```

```
Buffer b;
```

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

Снова прозрачная оболочка

- Возможный выход: сделать аргумент ссылкой

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

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

```
extern Buffer foo(Buffer x);
```

```
Buffer b;
```

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

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


Снова прозрачная оболочка

- Возможный выход: перегрузить по константной ссылке

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

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

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

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

- Но есть проблемы:
 - Всего 10 аргументов потребуют 1024 перегрузки
 - Вызов для rvalue всё ещё требует копирования

Снова прозрачная оболочка

- Решение для первой проблемы: универсализовать ссылку

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

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

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

- Но есть проблемы:
 - ~~Всего 10 аргументов потребуют 1024 перегрузки~~
 - Вызов для rvalue всё ещё требует копирования

Чего бы нам хотелось

- Решение для второй проблемы: условное перемещение

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

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

- Это решило бы часть проблем. Но это не легальный C++. Хотя, постойте....

Решение: использовать std::forward

- Решение для второй проблемы: условное перемещение

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

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

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

Обсуждение: emplace

- Что если мы пробросим аргументы **для конструктора**?

```
MyVector<Heavy> vh;
```

```
vh.push(Heavy{100}); // создаёт, потом перемещает
```

```
vh.emplace(100); // пробрасывает, создаст на месте
```

- Это может очень существенно сократить количество операций
- Внезапно настоящий `std::vector` это умеет и более того, умеет принимать произвольное количество аргументов конструктора.
- Но об этом и многом другом в следующий раз.

Литература

- [CC11] ISO/IEC 14882 – "Information technology – Programming languages – C++", 2011
- [BS] Bjarne Stroustrup – The C++ Programming Language (4th Edition) , 2013
- [EM] Scott Meyers, "Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14"
- [SM] Scott Meyers "Type Deduction and Why You Care", CppCon, 2014
- [VJ] Davide Vandevoorde, Nicolai M. Josuttis – C++ Templates. The Complete Guide, 2nd edition, Addison-Wesley Professional, 2017