Вывод типов, часть 2

Практическое объектно-ориентированное программирование

02.10.2024

План

- Свертка ссылок, decltype
- Perfect forwarding
- Сводим знания воедино

Правила свертки ссылок

Что нам говорит об этом стандарт:
It is permitted to form references to references through type manipulations in templates or typedefs, in which case the reference collapsing rules apply: rvalue reference to rvalue reference collapses to rvalue reference, all other combinations form lvalue reference:

```
typedef int& lref;
typedef int&& rref;
int n;

lref& r1 = n; // type of r1 is int&
lref&& r2 = n; // type of r2 is int&
rref& r3 = n; // type of r3 is int&
rref&& r4 = 1; // type of r4 is int&&
```

Правила свертки ссылок

• В контексте вывода типов ссылки сворачиваются и левые ссылки выигрывают.
• При этом само наличие T&& в контексте вывода заставляет компилятор выводить T& для lvalue и T&& для rvalue.

```
template <typename T,
        typename R = std::remove_reference_t<T>&&>
R almost_move(T&& a) {
        return static_cast<R>(a);
}
int x; auto&& t = almost_move(x); // T → int&, T&& -> int&,
// R -> int&&, t -> int&&
```

Универсальность ссылок

Универсальность ссылок

• 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&

• Это в целом работает и для шаблонов

template <typename T> void foo(T&& t);

foo(x); // аналогично

• Такие ссылки называют forwarding references. Майерс предложил называть их универсальными ссылками.

• Важное требование универсальности ссылки: контекст вывода типов.

Неуниверсальные ссылки

• Контекст сворачивания требует вывода типов, а не их подстановки:

```
template <typename T> struct Buffer {
        void emplace(T&& param); // substitute T
}
template <typename T> struct Buffer {
        template <typename U> void emplace(U&& param); // deduce U
}
```

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

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

•Это так же верно для auto

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

Мотивация для decltype

• Ключевое слово auto либо режет типы либо добавляет уточнители.
• Кроме того оно использует правую часть как для вывода, так и для присваивания.

```
const int x = 5;
auto y = x; // y → int
auto &z = x; // z → const int&
```

•Чтобы вывести точный тип, у нас есть decltype.

```
decltype(x) t = 6; // t → const int
```

•Но тут есть вопросики....

Decltype: что такое точный тип?

•Приоритет для decltype это точный тип параметра.
const int &x = 42;
decltype(x) y = 42; // → const int &y = 42;

•Это прекрасно. Но есть проблема:
struct Point int x, y; ;
Point porig 1, 2;
const Point &p = porig;
decltype(p.x) x = 0; // здесь int x или const int &x?

Decltype: name and expression

struct Point { int x, y; };
Point porig 1, 2;
const Point &p = porig;
•Случай decltype(id-expr)
decltype(p.x) x = 0; // → int x = 0;
•Случай decltype(expr)
decltype((p.x)) x = 0; // → const int &x = 0;
•Точный тип это decltype(name), а вот decltype(expr) работает от категории.

Правила для decltype(id-expr)

int x; decltype(x) t1 = y; // name $\rightarrow$ int

- decltype(id-expr) различает категории значений.
- Для lvalues он добавляет левую ссылку.

decltype((x)) t2 = y; // lvalue expr $\rightarrow$ int&

- Для xvalues он добавляет правую ссылку.

decltype(std::move(x)) t3 = 1; // xvalue expr $\rightarrow$ int&&

- Для prvalues он ничего не добавляет.

decltype(x + 0) t4; // prvalue expr $\rightarrow$ int

Абстракция значения

• В некоторых случаях (например для использование внутри decltype) хочется получить значение некоего типа.
• Часто для этого используется конструктор по умолчанию

```
template <typename T> struct Tricky {
    Tricky() = delete;
    const volatile T foo();
};
decltype(Tricky<int>().foo()) t; // ошибка
```

•Но что делать, если его нет? Что такое "значение вообще"для такого типа?

Абстракция значения: memory chunk

• Человек, искушённый в C мог бы сказать, что значение как таковое это результат приведения сырой памяти.

```
template <typename T> struct Tricky {
    Tricky() = delete;
    const volatile T foo();
};
decltype(((Tricky<int>*)0)->foo()) t; // работает, но это UB
```

•В compile-time любой reinterpret_cast запрещён стандартом.

Абстракция значения: declval

•Интересный способ решить эти проблемы это ввести шаблон функции (который выводит типы) без тела (чтобы его нельзя было по ошибке вызвать).

```
template <typename T>
std::add_rvalue_reference_t<T> declval(); // нет тела
```

•Теперь всё просто

```
template <typename T> struct Tricky {
    Tricky() = delete;
     const volatile T foo ();
};
decltype(declval<Tricky<int»().foo()) t = 0; // ok, cv-int
```

Вершина всего: decltype(auto)

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

Улучшаем almost_move

- Предыдущий вариант:

```
template <typename T,
        typename R = std::remove_reference_t<T>&&>
R almost_move(T&& a) {
        return static_cast<R>(a);
}
```

- Используем decltype(auto)

```
template <typename T> decltype(auto) almost_move(T&& a) {
    using R = std::remove_reference_t<T>&&;
    return static_cast<R>(a); // decltype(auto) от xvalue
}
```

Прозрачная оболочка

• Идея прозрачной оболочки: вызывать переданную ей функцию с минимальными накладными расходами.

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

• Увы, её недостаток в том, что она не слишком прозрачна.

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

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); }
```

- Теперь вроде всё хорошо

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

- Вы видите тут ещё одну проблему?

И еще раз прозрачная оболочка

- Решение для проблемы числа перегрузок: универсализовать ссылку.

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

- Увы, у нас всё ещё копируется аргумент если он rvalue.

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

Чего бы хотелось?

- Нам бы хотелось условного перемещения.

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

- И в языке оно есть.

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

Внутри forward

- forward это тоже просто static_cast

```
template<typename S>
S&& almost_forward(std::remove_reference_t<S>& a) {
    return static_cast<S&&>(a);
}
```

- Тут нет контекста вывода.
- Поэтому необходимо использование управляющего типа S, который универсально сворачивается в нужный тип ссылки.

Так используем же std::forward

• Теперь когда мы понимаем как это работает....

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

•Видно, что всё хорошо.

```
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 forward<T>.

Обсуждение

- Пожалуй есть всего три функции, для которых имеет смысл возвращать правую ссылку (то есть производить xvalue)
- std::move
- std::forward
- std::declval

Если вы хотите написать свою функцию, которая будет возвращать && это значит, что:

- Вы что-то делаете не так
- Вы хотите ещё раз написать одну из упомянутых выше функций
- Вы пишете функцию, аннотированную как &&

Пример Йосьюттиса

```
class Customer {
    MyString fst, snd;
public:
    Customer(const MyString &s1, const MyString &s2 = ) :
        fst(s1), snd(s2)
};
```

- Кажется этот пример очень неэффективен.
- Улучшим его форвардингом?

Уменьшаем копирования

```
class Customer {
    MyString fst, snd;
public:
    template <typename S1, typename S2>
    Customer(S1 &&s1, S2 &&s2 = "") :
        fst(std::forward<S1>(s1)), snd(std::forward<S2>(s2)) {} };
```

- Казалось бы тут можно и закончить. Увы...

Внезапная проблема

```
class Customer {
    MyString fst, snd;
public:
    template <typename S1, typename S2>
Customer(S1 &&s1, S2 &&s2 = "") :
fst(std::forward<S1>(s1)), snd(std::forward<S2>(s2))
};
.....
Customer N("Nico");
```

Спасаем ситуацию

```
class Customer {
    MyString fst, snd;
public:
    template <typename S1, typename S2 = const char *>
    Customer(S1 &&s1, S2 &&s2 = "") :
        fst(std::forward<S1>(s1)), snd(std::forward<S2>(s2))
};
.....
Customer N("Nico"); // ok
```

Копирующий конструктор

```
class Customer {
    MyString fst, snd;
public:
template <typename S1, typename S2 = const char *> Customer(S1
&&s1, S2 &&s2 = “”) :
fst(std::forward<S1>(s1)), snd(std::forward<S2>(s2)) {}
};
.....
Customer N("Nico");
Customer M(N); // как вы думаете?
```

Концепты спешат на помощь

```
class Customer {
MyString fst, snd;
public:
template <typename S1, typename S2 = const char *>
requires(!std::is_same_v<
std::remove_reference_t<S1>, Customer>)
Customer(S1 &&s1, S2 &&s2 = ) :
fst(std::forward<S1>(s1)), snd(std::forward<S2>(s2))
};
.....
Customer N("Nico"); Customer M(N); // ok
```