

Lecture 19

Вариабельные шаблоны

`emplace_back` в отличие от `push_back` принимает не объект, а конструктор объекта. Таким образом, не будет происходить лишнего копирования. Объект будет создан прямо на месте контейнера.

- Теперь единственным облачком на горизонте остался `emplace`

```
struct S {  
    S();  
    S(int, double, int);  
};  
  
std::vector<S> v;  
v.emplace_back(1, 1.0, 2); // создали на месте
```

- Но как это может работать для любого типа, если мы в общем случае не знаем количество аргументов конструктора?

40

Т.е. в `emplace_back` поступают аргументы конструктора, но их количество неопределено. Необходима переменность.

Значения троеточия в языке Си

1. `...` as `VA_ARGS`
2. `...` as `VA_ARGS` в макросах

Значения троеточия в C++

- `...` as catch in templates
- **Сущность после троеточия:** `...` as entry: `template<typename ... Args> void f(Args ... args)` - здесь принимается произвольное число типов и произвольное число аргументов, **но равное числу типов**.
- `sizeof...(Args)` / `sizeof...(args)` - возвращает **число** типов / аргументов.

Вариабельные шаблоны

- Пример вариабельно шаблонной функции

```
template<typename ... Args> void f(Args ... args);
```

- Способы вызова:

```
f(); // OK, пачка не содержит аргументов
```

```
f(1); // OK, пачка содержит один аргумент: int
```

```
f(2, 1.0); // OK, пачка состоит из: int, double
```

- Специальная конструкция `sizeof...(Args)` либо `sizeof...(args)` возвращает размер пачки в штуках

42

- **Сущность до троеточия:** `...` as decode - раскрывает аргументы (т.е. превратить в перечисление через запятую). Он может быть составным

При раскрытии оператор `...` жадно матчит все, что слева от него, до тех пор, пока левое нечто синтаксически корректно.

Паттерны раскрытия

- Говорят, что пачка параметров "раскрывается" в теле функции или класса

```
template<typename ... Types> void f(Types ... args);
```

```
template<typename ... Types> void g(Types ... args) {  
    f(args ...); // → f(x, y);  
    f(&args ...); // → f(&x, &y);  
    f(h(args) ...); // → f(h(x), h(y));  
    f(const_cast<const Types*>(&args)...);  
        // → f(const_cast<const int*>(&x),  
        const_cast<const double*>(&y));  
}  
g(1, 1.0); // → g(int x, double y);
```

43

При этом при раскрытии пары `Args, args` троеточие при раскрытии работает как молния (zip).

Вопрос: а может в качестве аргументов функции выступать несколько пачек variadic? Да, такое бывает.

Упражнения

Решение

- Допустим `args` это пачка параметров `x, y, z`
- Тогда следующее выражение имеет сложный паттерн раскрытия пачки

```
f(h(args...) + h(args)...); // → f(h(x, y, z) + h(x),  
                                   h(x, y, z) + h(y),  
                                   h(x, y, z) + h(z));
```

- Аналогично (если чувствовать технологию, эти задачи однообразны)

```
f(h(args, args...)...); // → f(h(x, x, y, z),  
                               h(y, x, y, z),  
                               h(z, x, y, z));
```

45

Прозрачная оболочка и переменные шаблоны

`decltype(auto)` нужно для того, чтобы сохранять `&`, `&&` в случае вывода функцией ссылок.

`Arg&&` - для reference collapsing + форварда

В случае `T&&` все зависит от типа выражения: lval / rval.

Если lval - добавляет `&` и делает reference collapsing

Если rval - добавляет `&&` и делает reference collapsing

При свертке ссылок `& = 0`, `&& = 1`, свертка - логическое И.

`std::forward<Arg>` нужен для того, чтобы не делать лишнее копирование.

Работа `std::forward`:

Если `T`, то `std::move`

Если `T&` (lval ref), то ничего

Если `T&&`, то `std::move`

Снова прозрачная оболочка

- На лекции по rvalue refs была написана почти идеальная прозрачная оболочка для одного аргумента

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

- Можно ли использовать переменный шаблон и переписать её для произвольного количества аргументов?

46

Снова прозрачная оболочка

- На лекции по rvalue refs была написана почти идеальная прозрачная оболочка для одного аргумента

```
template<typename Fun, typename... Args>
decltype(auto) transparent(Fun fun, Args&&... args) {
    return fun(forward<Args>(args)...);
}
```

- Это очень простое и чисто техническое изменение
- Следует обратить особое внимание на паттерн совместного раскрытия при пробросе

47

Еще одна поправка:

`Fun` - вызываемый объект (foo-подобный объект): указатель на функцию, лямбда, класс с перегруженными `()`. А что если в последнем случае оператор `&&` аннотирован (чтобы нельзя было вызвать для lval)? Тогда `fun(args)` не подставится в `Fun fun` - это lvalue, а у нас rval. Поэтому применим то же самое, что с args.

Обсуждение: пробросим функцию?

- В функции-подобном объекте оператор вызова может быть && аннотирован

```
template<typename Fun, typename... Args>
decltype(auto) transparent(Fun&& fun, Args&&... args) {
    return std::forward<Fun>(fun)(std::forward<Args>(args)...);
}
```

- Теперь функции тоже не требуется быть обязательно копируемой
- Выглядит это чуть страшнее, зато теперь тут не к чему особо придраться

48

Мы написали свой `emplace`

Такая техника позволит избежать нам лишних копирований (см семинар 13).

Контейнеры тяжёлых классов

- Мы уже говорили о хранении тяжелых классов в контейнерах

```
template <typename T> class Stack {
    struct StackNode {
        T elem; StackNode *next;
        StackNode(T e, StackNode *nxt) : elem (e), next (nxt) {}
    };
public:
    void push(const T& elem) { top_ = new StackNode (elem, top_); }
    // .... и так далее ....
}
```

- Подумаем о следующем коде:

```
s.push(Heavy(100, 200, 300)); // всё очень плохо
```

49

empi-bench-ineff.cc

Тут будет два копирования: одно при передаче по значению в Stacknode, а второе - при инициализации поля elem.

```
1  template<typename T> class Stack {
2      /* ... */
3      struct StackElem {
4          T elem;
5          StackElem *next;
6          StackElem(StackElem *nxt, T e) : elem(e), next(next) {}
7
8          template <typename... Args>
9              StackElem(StackElem *nxt, Args &&... args)
10                 : elem(std::forward<Args>(args)...), next(nxt) {}
11     };
12     /* ... */
```

```

13     template <typename... Args> void emplace_back(Args &&... args);
14 };
15
16 template<typename T> // class template args
17 template<typename... Args> // method template args
18 void Stack<T>::emplace_back(Args &&... args) {
19     top_ = new StackElem(top_, std::forward<Args>(args)...);
20 }
21
22 int main() {
23     Stack<Heavy> s; // эта строка не поменялась
24     /* ... */
25     s.emplace_back(100);
26 }
27
28

```

Теперь не будет никаких вызовов copy constructor.

Специализация шаблонных методов

Специализация шаблонных методов

- При специализации шаблонных методов, важно понимать: вы должны специализировать их по всем аргументам

```

template <typename T> struct Foo {
    template <typename U> void foo() { .... }
};

```

```

template <>
template <>
void Foo<int>::foo<int>() { .... }

```

- Иначе это будет частичная специализация

Как мы помним, частичная специализация функций (в том числе методов) запрещена. Это значит, что мы не можем специализировать только U. Только одновременно U и V.

При это должна присутствовать пара `template<>` - синтаксис специализации.

Без ключиков: либо deduction hint, либо `extern template` (явная специализация).

Шаблонные методы - зло, за исключением `emplace_back`

Шаблонные классы нарушают инкапсуляцию.

```

class Foo {
    int donottouch_ = 42;

public:
    template <typename T> void foo() {
        std::cout << donottouch_ << std::endl;
    }
};

struct MyTag {};

template <>
void Foo::foo<MyTag>() {
    donottouch_ = 14;
}

int main() {
    Foo f;
    f.foo<MyTag>(); // change private data
    f.foo<int>();
}

```

Ву, мы изменили `private` часть снаружи. На экране 14, а не 42.

Лучше избегать шаблонных методов. Кроме `emplace_back`.

void_t

template <typename...> using void_t = void;

```

1 void_t <T, U, V> // легален тогда и только тогда, когда T, U, V - легальные
2   типы. В противном случае будет substitution failure.
3 // Это логическое И для типов во время работы механизма SFINAE.

```

```

#include <iostream>
#include <type_traits>

template <typename, typename = void>
struct has_typedef_foobar: std::false_type { };

template <typename T>
struct has_typedef_foobar<T,
    std::void_t<typename T::foobar>>: std::true_type{};

struct Foo { typedef float foobar; };
struct Bar {};

int main() {
    std::cout << std::boolalpha;
    std::cout << has_typedef_foobar<Foo>{} << std::endl;
    std::cout << has_typedef_foobar<Bar>{} << std::endl;
}

```

Тут `Foo` содержит тип `foobar`, тогда как `Bar` его не содержит.

`has_typedef_foobar` по умолчанию наследуется от класса-трейта `std::false_type`, у которого перегружен булев оператор - он возвращает `false`.

В частичной специализации идет наследование от `std::true_type`, но компилятор провалится в нее только в случае валидной подстановки.

Подстановка валидна только тогда, когда валидна `std::void_t<typename T::foobar>` - тогда она вернет `void`. Однако если типа `T::foobar` не существует - будет substitution failure и вызовется специализация по умолчанию (`has_typedef_foobar : std::false_type`).

Аналогично мы можем проверять наличие функциональности, используя `decltype` / `decltype`.

Конструирование из итераторов

- Можно попытаться решить задачу с итераторами вот так

```
MyVector(size_t nelts, T value);  
template <typename Iter,  
          typename = void_t<decltype(*Iter{}),  
                        decltype(++Iter{})>  
          >  
MyVector(Iter fst, Iter lst);
```

- Увы это не слишком изящно. Дело в том, что инкремент требует lvalue.
- Но его-то мы как раз пока и не можем создать. Хотя иногда везет.

58

<https://godbolt.org/z/cx9vHz1b3>

Но тут есть проблемы. `++Iter{}` - преинкремент временного объекта. Вообще для преинкремента нужно lvalue. А еще нужен конструктор по умолчанию, иначе операции не сработают.

Абстракция значения

Как решить проблему с отсутствием конструктора по умолчанию?

Абстракция значения

- В некоторых случаях (например для использования 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?

59

1. UB: cast `nullptr` to `Tricky<int>`
2. Lval ref не требует конструктора по умолчанию, но нету каста для нее. А есть каст к rval ref? `std::move` требует вызова тела, не подойдет. А вот `declval` - функция, у которой по стандарту нету тела. Только декларация, которая возвращает rval ref:

```
1 | template <typename T> add_rvalue_reference_t<T> declval();
```

Засчет отсутствия вызова аргумента все и работает.

Почему declval возвращает правую ссылку, а не левую? Потому что && аннотированная foo не пройдет. А с rval работают & и && аннотированные методы.

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 &&

61

Таким образом,

Конструирование из итераторов

- Теперь мы видим совсем изящное решение

```
MyVector(size_t nelts, T value);  
template <typename Iter,  
          typename = void_t<decltype(*std::declval<Iter&>()),  
          decltype(++std::declval<Iter&>())>  
>  
MyVector(Iter fst, Iter lst);  
....  
MyVector v1(10, 3); // 1, поскольку 2 провалилось  
MyVector v2(v1.begin(), v1.end()); // 2
```

62

<https://godbolt.org/z/35YneP3za>

Заметим, что в шаблонах записан шаблонный интерфейс.

```
1 | template <typename Var, typename Checks>
```