

Lecture 28

Constraints

Констрейнты

- Констрейнты были введены чтобы сделать статические интерфейсы явными

```
template <typename T, typename U> bool  
    requires is_equality_comparable<T, U>::value  
check_eq (T &&lhs, U &&rhs) { return (lhs == rhs); }
```

- Больше нет мусорного параметра шаблона. Языковые средства используются для того, для чего должны

- Сообщение об ошибке куда как лучше

'is_equality_comparable<T, U, void>::value' evaluated to false

- Внутри requires может быть что угодно, вычисляемое на этапе компиляции

68

`requires` - означает проверку выражения на этапе компиляции. Если они выполнены, то инстанцирование происходит. Это есть "входные ворота" к инстанцированию. Оно зависит от `T`, `U`.

`is_equality_comparable` - проверка на операции эквивалентности.

Засчет SFINAE, отсутствие подходящего шаблона на этапе инстанцирования не ошибка. Но в случае с `requires` это не так.

Проблема ODR

Полное покрытие

- Все помнят почему не работает очевидный SFINAE подход к разграничению?

```
template <typename T, typename = enable_if_t<(sizeof(T) > 4)>>  
void foo (T x) { сделать что-то с x }  
  
template <typename T, typename = enable_if_t<(sizeof(T) <= 4)>>  
void foo (T x) { сделать что-то ещё с x }
```

- Очевидный подход через констрейнты вполне работает

```
template <typename T> requires (sizeof(T) > 4)  
void foo (T x) { сделать что-то с x }  
  
template <typename T> requires (sizeof(T) <= 4)  
void foo (T x) { сделать что-то ещё с x }
```

61

<https://godbolt.org/z/8KeGbbabK>

В красном случае значения по умолчанию не входят в манглирование, а значит первые две функции с точки зрения перегрузки идентичны. Получаем ошибку. Ее можно исправить так:

```
1 template <typename T, enable_if_t<(sizeof(T) > 4)>* = nullptr>
2 void foo(T x) { /* ... */ }
3
4 template <typename T, enable_if_t<(sizeof(T) <= 4)>* = nullptr>
5 void foo(T x) { /* ... */ }
```

Или использовать `requires`. Он входит в манглирование.

Недостатки `requires`

Недостатки `sfniae-constraints`

- Увы, SFINAE определители не упорядочены в отношении ограниченности

```
template <typename It>
struct is_input_iterator: std::is_base_of<
    std::input_iterator_tag,
    typename std::iterator_traits<It>::iterator_category>{};

template <typename It>
struct is_random_iterator: std::is_base_of<
    std::random_access_iterator_tag,
    typename std::iterator_traits<It>::iterator_category>{};
```

- Это просто два разных шаблона. И это приводит к проблемам, когда мы пытаемся исправить `distance`

62

`is_base_of<Base, Derived>` - возвращает true, если `Base` is base of `Derived`.

Недостатки `sfniae-constraints`

- Увы, SFINAE определители не упорядочены в отношении ограниченности

```
template <typename Iter>
requires is_input_iterator<Iter>::value
int my_distance(It first, It last) {
    int n = 0; while (first != last) { ++n; ++first; } return n;
}

template <typename Iter>
requires is_random_iterator<Iter>::value
int my_distance(It first, It last) { return last - first; }
```

- При реальном использовании здесь будет неоднозначность для `std::vector`

63

<https://godbolt.org/z/8KoGbbabK>

По реализации `std::random_access_iterator_tag` наследует реализацию `std::input_iterator_tag`. Поэтому все, что удовлетворяет второй функции, также удовлетворяет первой. Но для `requires` не реализована приоритетность или отношение порядка. Поэтому код для `my_distance` выдаст ошибку.

Requires requires

Сложные ограничения

- Вернёмся к простому примеру

```
template <typename T, typename U> bool
    requires is_equality_comparable<T, U>::value
check_eq (T &&lhs, U &&rhs) { return (lhs == rhs); }
```

- То же самое можно записать через `requires-expression`

```
template <typename T, typename U> bool
    requires requires(T t, U u) { t == u; }
check_eq (T &&lhs, U &&rhs) { return (lhs == rhs); }
```

- Да, `requires-requires` может смущать. Но вспомните `noexcept-clause` и `noexcept-expression`

64

У `requires` две функции:

- Ограничение инстанцирования при использовании `statement` - ограничивает инстанцирование (превращает SFINAE в ошибку компиляции)
- `requires expression` - выражение, которое проверяет SFINAE условие, но в отличие от `constexpr` не выполняет предикат, а проверяет его семантическую корректность.

SFINAE if: выражение либо ДА, либо провал подстановки

Requires if: если `expression` семантически возможен, то он возвращает `true`, иначе - `false`

Главное отличие сложных ограничений

- Простые ограничения вычисляются на этапе компиляции

```
template <typename T> constexpr int somepred() { return 14; }
template <typename T>
    requires (somepred<T>() == 42)
bool foo (T&& lhs, U&& rhs);
```

- В сложных ограничениях проверяется синтаксическая валидность выражения

```
template <typename T>
    requires requires (T t) { somepred<T>() == 42; }
bool bar (T&& lhs, U&& rhs);
```

- В итоге вызов `foo` будет ошибкой, а вызов `bar` нет

66

<https://godbolt.org/z/WY6jsbMTx>

В первом случае проверяется `14 == 42`

Во втором случае проверяется валидность выражения `int == int`. При этом само выражение не выполняется.

Что проверяют сложные ограничения

- Сложные ограничения могут проверять валидность выражений
`requires requires(T a, T b) { a + b; }`
- Либо они могут проверять существование типов
`requires requires() { typename T::inner; }`
- Есть специальный синтаксис для noexcept
`requires requires(T t) {
 { ++t } noexcept;
}`
- Они могут комбинироваться друг с другом и с простыми ограничениями

67

Концепты

Концепт - булев предикат на этапе компиляции.

Чем он лучше той же `constexpr bool foo(cond);?`

Преимущество в том, что концепт можно складывать конъюнктивными или дизъюнктивными условиями, представленными либо классическими SFINAE-конструкциями, либо `requires`-подобными.

Пример: convertible_to

- Чтобы выделять системы ограничений, в C++20 введено специальное ключевое слово `concept`
 - Простейший концепт который определён в хедере `concepts` и часто используется как вспомогательный
- ```
template<class From, class To>
concept convertible_to =
 std::is_convertible_v<From, To> &&
 requires(From (&f)()) { static_cast<To>(f()); };
```
- Он состоит из старых SFINAE определителей и из новых концептов

68

`convertible_to` не ошибочен, если определен `static_cast` (первое условие), и конвертация работает корректно. Принятие ссылки на функцию позволит избежать негативных `lval` эффектов.

# Синтаксический сахар

- Чтобы немного проще записывать одновременное требование к выражению и типу:

```
requires requires(T x) {
 *x;
 requires convertible_to<decltype(*x), typename T::inner>;
}
```

- Существует более приятная форма записи со стрелочкой.

```
requires requires(T x) {
 {*x} -> convertible_to<typename T::inner>;
}
```

69

После того, как концепт определен, он может быть использован внутри `requires`.

Requires-requires-requires не совсем эквивалентен вызову одного `requires`. В обоих случаях идет проверка на `bool`, но в первом мы имеем комплексный предикат вида `(valid && true)?`, т.е. выражение семантически валидно и оно удовлетворяет концепту. Это утверждение эквивалентно более лаконичной записи со стрелочкой (второй пункт).

Концепты могут быть *составными*. Сами концепты применимы под `requires`.

## Концепты

- На основе простых концептов можно строить более сложные.

```
template <typename T, typename U>
concept WeaklyEqualityComparableWith =
 requires(const std::remove_reference_t<T>& t,
 const std::remove_reference_t<U>& u) {
 { t == u } -> convertible_to<bool>;
 { t != u } -> convertible_to<bool>;
 { u == t } -> convertible_to<bool>;
 { u != t } -> convertible_to<bool>;
 };
};
```

- Концепт это предикат, выполняющийся на этапе компиляции.

70

<https://godbolt.org/z/Ez981d41P>

`WeaklyEqualityComparable` проверяет, что существуют семантические конструкции с операторами, и их результат конвертируем в `bool`. Заметим, что каждый `statement` со стрелочкой является конъюнктом.

Далее мы этот концепт используем в `requires`.

```
template <typename T, typename U>
requires WeaklyEqualityComparableWith<T, U>
bool foo(T x, U y) {
 if ((x == y) && (y != x)) {
 std::cout << "comparison is weak" << std::endl;
 return false;
 }
 std::cout << "comparison is strong" << std::endl;
 return true;
}
```

Структура, удовлетворяющая концепту:

```
struct W {};

bool operator==(W, int) { return true; }
bool operator==(int, W) { return true; }
bool operator!=(W, int) { return true; }
bool operator!=(int, W) { return true; }
```

Структура, не удовлетворяющая концепту (struct S):

```
int
main() {
#ifdef BAD
 struct S{};
 auto w = foo(S{}, 1); // wrong
#endif
 auto c1 = foo(1, W{}); // corr
 auto c2 = foo(1, 2); // corr
}
```

**Важно.** Для концептов не существует специализации! То, что изображено ниже, не является специализацией. Это совершенно новый концепт.

## Концепты

- Теперь при наличии концепта, довольно легко ограничить функцию

```
template <typename T, typename U>
requires WeaklyEqualityComparableWith<T, U>
bool foo(T x, U y);
```

- Это также просто как использовать обычный предикат времени компиляции
- Можно определять одни концепты в терминах других

```
template <typename T>
concept EqualityComparable = WeakEqualityComparableWith<T, T>;
```



Сокращенные записи концептов:



## Теперь перегрузка работает

```
template <std::input_iterator Iter>
int my_distance(Iter first, Iter last) {
 int n = 0;
 while (first != last) { ++first; ++n; }
 return n;
}

template <std::random_access_iterator Iter>
int my_distance(Iter first, Iter last) {
 return last - first;
}
```

- Благодаря тому, что InputIterator является менее общим (он входит как подусловие в RandomAccessIterator) тут нет неоднозначности

73

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

```
1 | template <std::input_iterator Iter>
2 | <==>
3 | template <typename Iter>
4 | requires std::input_iterator<Iter>
```

А еще можно вот так:

```
1 | std::input_iterator auto x; // все что угодно, являющееся input_iterator
```

35:28