C++

Spring 2021

compscicenter.ru

Philipp Grabovoy

t.me/phil-grab

Lecture X

SFINAE → Concepts

concept

example

```
template<typename T>
concept CopyConstructible = \
    std::is_copy_constructible_v<T>;

static_assert(CopyConstructible<int>);

template<CopyConstructible T>
void foo(T t) { /* ... */ }
```

- задает ограничения на аргументы

syntax

```
template < template-parameter-list >
concept concept-name = constraint-expression;
```

- constraint-expression:
  - `||`, `&&`, `( /* … */ )`
  - atomic: `C<A1, A2, …>`, `constexpr`, `requires`, …
- normalisation + substitution [*]

parameter list example

```
template<int I>
concept Even = I % 2 == 0;
```

example

```
template<class T>
concept SomeCompoundConcept = \
          SomeConcept<T> && \
          SomeOtherConcept<T> \
    || \
          std::is_nothrow_move_assignable_v<T> && \
          requires( /* ... */ ) { /* ... */ }  // [*]
```

constraint with cast to bool

- невозможен — ожидается true/false, без преобразований типов:

```
template <typename T>
constexpr int my_is_copy_constructible = \
    std::is_copy_constructible_v<T>;

template<typename T>
concept CopyConstructible = my_is_copy_constructible<T>;
  // Error: not bool
```

using defined concepts

- in type requirements in requires [*]
- ...

in template parameters

```
template<C T>          // на самом деле C<T>
void foo(T) { }


template <class T, class U>
concept Derived = std::is_base_of_v<U, T>;

template<Derived<Base> T> // проверяется
                          // Derived<T, Base>
void bar(T);
```

- аргумент типа `T` добавляется в начало

in auto type specifier

```
void foo() {
    C auto var = makeObject();
}


template<typename T, size_t MaxSize, size_t MaxAlign>
concept Small = \
    sizeof(T) <= MaxSize && \
    alignof(T) <= MaxAlign;

bar(Small<SIZE, ALIGNMENT> auto const & t) {}
```

as constexpr bool value + more auto

```
template<typename T>
concept LEPtr = Small<T, sizeof(void *), alignof(void *)

static_assert(LEPtr<void*>);

template<typename T>
LEPtr auto get_handle(T& object) {
  if constexpr(LEPtr<T>)
    return object;
  else
    return &object;
}
```

requires expressions

syntax

```
requires { requirement-seq }
requires ( parameter-list ) { requirement-seq }
```

- выражение с типом `bool` (prvalue)
- `parameter-list` — как в декларации функций
  - без дефолтных параметров
  - поддержаны универсальные ссылки
- `requirement-seq`: {simple | type | compound | nested}; ...

no parameter list example

```
template<typename T>
concept Hashable = requires { std::hash<T>(); };
```

simple requirement

- проверяется компилируемость выражения (без вычислений)

```
template <class T, class U = T>
concept Swappable = requires(T&& t, U&& u) {
    swap(std::forward<T>(t), std::forward<U>(u));
    swap(std::forward<U>(u), std::forward<T>(t));
};
```

type requirement

- начинается с `typename`
- проверяет корректность имени + обозначение типа

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

template<typename T> concept C = requires {
    typename T::inner; // вложенное имя, задающее тип
    typename S<T>;     // валидная специализация шаблона
};
```

compound requirement

```
{ expression } [noexcept] [-> type-constraint]
```

- проверяет компилируемость + noexcept + ограничения на тип `decltype((expression))`

```
template<typename T> concept C = requires(T x) {
    {x + x} -> std::same_as<T>;
    {x * 1} -> std::convertible_to<T>;

    {x * x} -> Hashable;
};
```

nested requirement

- начинаются на `require` — как правило проверяют параметры

```
template <class T>
concept SomeConcept = \
    SomeOtherConcept<T> && \
    requires(T a) {
        requires std::same_as<T*, decltype(&a)>;
    };
```

example

```
template<typename Container>
concept Sizable = requires(const Container& c) {
  c.size() -> size_t;
};
```

test with vector

```
template<typename Container>
concept Sizable = requires(const Container& c) {
    c.size() -> size_t;
};

static_assert(Sizable<vector<int>>);  // nope
```

- simple requirement: `c.size()→size_t;` — member from pointer access

random access iterator concept example

```
template<class Iterator>
concept RandomAccessIterator = \
    BidirectionalIterator<Iterator> &&
    // ... && \
    requires(Iterator i,
             const Iterator j,
             const std::iter_difference_t<Iterator> n) {
      { i += n } -> std::same_as<Iterator&>;
      // ...
      // ...
      {  j[n]  } -> \
           std::same_as<std::iter_reference_t<Iterator>
    };
```

requires clause

usage

- используется в определении шаблона
- задает фильтр на типы (по принципу SFINAE)
  - ожидает constexpr bool значение
  - внутри могут быть &&, ||

```
template<typename T>
void foo(T) requires SomeConcept<T> { }

template<typename T> requires SomeConcept<T>
void foo(T) { }

template<typename T> \
    requires std::is_copy_constructible_v<T>
struct Baz { };
```

compound statements

```
// template<typename R>
// concept Sortable = RandomAccessIterator<Iterator<R>>
//    Comparable<ValueType<R>>;

template<typename Range> requires \
    RandomAccessIterator<Iterator<Range>> && \
    Comparable<ValueType<Range>>
void sort(Range &) {  }
```

requires requires

```
template<typename T>
void foo(T) requires requires(T t) {   { ++t; }   }
{ }
```

changing pair’s constructor

default constructor with init

```
template<typename F, typename S> struct pair {
  F f; S s;

  pair() : f(), s() {}  // пусть делает инициализацию
};

struct A {
  A(int);
};

pair<int, A> a0;          // пусть падает
pair<int, A> a1{1, 2};    // пусть будет ок
```

проблемы

`default ctor`, для `int`, `A`:

- не работает: godbolt_1
  - ломает явную инстанциацию: godbolt_2
- доступен для компилятора: godbolt_3
- это плохо: для `<int, A>` конструктор и доступен, и не работает

⇒ сделаем опциональным: не будет виден компилятору в случае проблем

step 1: adding enable_if

```
template<typename F, typename S> struct pair {
  F f; S s;

  template<typename = enable_if_t<conjunction_v<
      is_default_constructible<F>,
      is_default_constructible<S>
    >>>
  pair() : f(), s() {}
};
```

- ⇒ не дефолтный конструктор
- `F`, `S` — not dependant for template
  - ⇒ full instantiation on `pair<int, A> a1{1, 2};` (поломка: godbolt)

step 2: adding dependency + default params

```
template<typename F, typename S> struct pair {
  F f; S s;

  template<
    typename T = F,
    typename U = S,
    typename = enable_if_t<conjunction_v<
      is_default_constructible<T>,
      is_default_constructible<U>
    >>>
  pair() : f(), s() {}
};
```

- работает: godbolt

усложнение: поддержка explicit

```
struct B { explicit B(); };

pair<int, B> p = {};  // пусть на этом падает
                      // (как стандартная пара)
```

two constructors

```
template<typename F, typename S> struct pair {
  F f; S s;

  template<
    ...
    typename = enable_if_t<conjunction_v<
      is_default_constructible_v<T>,
      is_default_constructible_v<U>,
      // (!)
      is_implicitly_default_constructible<T>,
      is_implicitly_default_constructible<U>,
    >>>
  pair() : f(), s() {}

  // + repeat for explicit pair
};
```

is_implicity_default_constructible tech

```
template<typename T> true_type test(T, int);
template<typename T> false_type test(int, ...);

template<typename T>
using is_implicitly_default_constructible = \
    decltype(test<T>({}, 0));
```

using c++20 changes

- templates + enable_if → concepts + requires
- conditional explicit

ImplicitlyDefaultConstructible concept

```
template<typename T>
concept ImplicitlyDefaultConstructible = \
    requires {
        [] (T) { /* empty body*/ }   ({});
    };
```

c++20 ctor

```
template<typename F, typename S> struct pair {
  F f; S s;

  explicit(
    !ImplicityDefaultConstructible<F> || \
    !ImplicityDefaultConstructible<S>
  )
  pair() requires DefaultConstructible<F> \
                  && DefaultConstructible<S>
    : f(), s() {}
};
```

more techs

- Андрей Давыдов — Концепты: упрощаем реализацию классов std utility