ЛЕКЦИЯ 17
КОНЦЕПТЫ

АЛГОРИТМИЗАЦИЯ И
ПРОГРАММИРОВАНИЕ

ЛЕКТОР ФУРМАВНИН С.А.

ОБОБЩЕННЫЙ КОД

- Типичный пример достаточно общего алгоритма.

```
template<typename R, typename T>
bool contains(R const& range, T const& value) {
  for (auto const& x : range)
  if (x == value)
    return true;
  return false;
}
```

- Какие требования этот код накладывает на шаблонные параметры R и T?

ОБОБЩЕННЫЙ КОД

- Типичный пример достаточно общего алгоритма.

```
template<typename R, typename T>
bool contains(R const& range, T const& value) {
  for (auto const& x : range)
  if (x == value)
    return true;
  return false;
}
```

- Какие требования этот код накладывает на шаблонные параметры R и T?
- `R::iterator` как минимум `InputIterator`.
- `R::value_type` сравним с T на равенство

ОБОБЩЕННЫЙ КОД

- Типичный пример достаточно общего алгоритма.

```
template<typename R, typename T>
bool contains(R const& range, T const& value) {
  for (auto const& x : range)
  if (x == value)
    return true;
  return false;
}
```

- Что если нарушить эти требования?

```
vector<string> v {"0", "1", "2"};
bool is_in = contains(v, 1); // oops
```

МЕНЕЕ ОБОЩЕННЫЙ КОД

- Заметим, что для динамически полиморфных функций таких проблем нет.

```
bool contains(IRange const *range, IVal const *value) {
  IEnumerator const *beg = range->start();
  while (beg != range->end()) {
    if (beg->get()->compare(value))
      return true;
    beg->next();
  }
  return false;
}
```

- Здесь легко додумать интерфейсные классы с соответствующими виртуальными функциями.

ДИНАМИЧЕСКИЙ ПОЛИМОРФИЗМ

Собственно каждый интерфейс здесь задаёт все способы использовать себя правильно (явным перечислением методов).

```
struct IEnumerator {
  virtual IVal *get() = 0;
  virtual void next() = 0;
  virtual ~IEnumerator() {}
};
struct IRange {
  virtual IEnumerator const *start() = 0;
  virtual IEnumerator const *end() = 0;
  virtual ~IRange() {}
};
```

Тут есть свои недостатки: например нельзя передать `int` и `string`

ОБСУЖДЕНИЕ

- Говорят, что интерфейсы в статическом полиморфизме являются неявными.
- Хорошо ли, что они неявные?
- Должны ли они быть неявными?
- Что если взять пример попроще и, находясь в реалиях C++17, попробовать сформулировать явный интерфейс в терминах типов?

ПРИМЕР ПОПРОЩЕ

- В следующей функции неявный контракт состоит из одного пункта: равенство.

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

- Разумеется, это требование можно сформулировать явно.

```
template <typename T, typename U, typename = void>
struct is_equality_comparable : false_type {};

template <typename T, typename U>
struct is_equality_comparable <T, U,
void_t<decltype(declval<T>() == declval<U>())>>: true_type
{};
```

- Вопрос в том, как его лучше всего **проверить**?

ПРИМЕР ПОПРОЩЕ

- В следующей функции неявный контракт состоит из одного пункта: равенство.

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

- Опция по умолчанию в таких случаях это enable_if

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

- Теперь сообщение будет выглядеть как-то так:

error: no matching function for call to 'check_eq'

ОБСУЖДЕНИЕ

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

- Какие проблемы вы здесь видите?

ОБСУЖДЕНИЕ

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

- Какие проблемы вы здесь видите?
- Используется шаблонный параметр, которого на самом деле не существует

```
check_eq<int, std::string, void>(1, "1"); // oops, 157 err lines
```

- В случае проблемы будет выдано сообщение, что нет такой функции, но не будет ничего или почти ничего сказано о том **почему** её нет

ИНТЕРЕСНАЯ ИДЕЯ

- Заслуживает внимания идея `if constexpr` + `static assert`

```
template <typename T, typename U>
bool check_eq(T &&lhs, U &&rhs) {
  if constexpr (!is_equality_comparable<T, U>::value>) {
    static_assert(0 && "equality comparable expected");
  }
  return (lhs == rhs);
}
```

- Стало лучше?
- Но мне не нравится эта идея. Как вы думаете почему?

Мы теряем SFINAE

- Перенося проверку корректности из констекста подстановки в тело функции мы меняем SFINAE-out на ошибку.
- Но часто мы хотим именно SFINAE-out.

ЗАГАДОЧНЫЙ DISTANCE

Представим, что мы написали итератор `junk_iter_t` и пытаемся использовать его.

```
int arr[10];
junk_iter_t fst(arr), snd(arr + 3);
auto dist = std::distance(fst, snd);
```

Он выдаёт ошибку

```
error: no matching function for call to 'distance(junk_iter_t&, junk_iter_t&)'
```

Проблема в том, что вы вроде бы всё определили.

ЗАГАДОЧНЫЙ DISTANCE

В более-менее реалистичном коде ошибка выглядит куда хуже

D:\GitHub\coelacanth\src\lib\controlgraph\controlgraph.cc:220:35: error: no matching function for call to 'distance(cg::nbr_iterator_t<boost::detail::out_edge_iter<__gnu_cxx::__normal_iterator<boost::detail::stored_edge_iter<unsigned int, std::_List_iterator<boost::list_edge<unsigned int, cg::edgeprop_t> >, cg::edgeprop_t>*, std::vector<boost::detail::stored_edge_iter<unsigned int, std::_List_iterator<boost::list_edge<unsigned int, cg::edgeprop_t> >, cg::edgeprop_t>, std::allocator<boost::detail::stored_edge_iter<unsigned int, std::_List_iterator<boost::list_edge<unsigned int, cg::edgeprop_t> >, cg::edgeprop_t> > > >, unsigned int, boost::detail::edge_desc_impl<boost::bidirectional_tag, unsigned int>, int> >&, cg::nbr_iterator_t<boost::detail::out_edge_iter<__gnu_cxx::__normal_iterator<boost::detail::stored_edge_iter<unsigned int, std::_List_iterator<boost::list_edge<unsigned int, cg::edgeprop_t> >, cg::edgeprop_t>*, std::vector<boost::detail::stored_edge_iter<unsigned int, std::_List_iterator<boost::list_edge<unsigned int, cg::edgeprop_t> >, cg::edgeprop_t>, std::allocator<boost::detail::stored_edge_iter<unsigned int, std::_List_iterator<boost::list_edge<unsigned int, cg::edgeprop_t> >, cg::edgeprop_t> > > >, unsigned int, boost::detail::edge_desc_impl<boost::bidirectional_tag, unsigned int>, int> >&)'
int dist = std::distance(it, eit);

ВЕРНЕМСЯ К ПРИМЕРУ ПОПРОЩЕ

Начиная с C++20 одобрено синтаксическое расширение, которое называется ограничениями (constraints) на шаблоны.

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

Теперь записывается в `requires` syntax

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

СТАЛО ГОРАЗДО ЛУЧШЕ

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

- Во-первых больше нет мусорного параметра шаблона. Языковые средства используются для того, для чего должны
- Во-вторых сообщение об ошибке куда как лучше

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

Сразу видно что требуется и что именно пошло не так

КОМБИНИРОВАНИЕ ОГРАНИЧЕНИЙ

Ограничения легко комбинируются

```
template <typename Iter>
  requires(is_forward_iterator<Iter>::value &&
    is_totally_ordered<typename Iter::value_type>::value)
Iter my_min_element(Iter first, Iter last) {
```

- Здесь требуется и одно и другое.
- При этом ошибки показываются разные в зависимости от того что пошло не так

```
note: 'is_forward_iterator::value' evaluated to false
note: 'is_totally_ordered<typename Iter::value_type, void>::value'
evaluated to false
```

ПЕРЕГРУЗКА ПО ОГРАНИЧЕНИЯМ

По ограничениям можно перегружать.

```
struct Foo {
  template <typename Int>
    requires std::is_integral<Int>::value
  Foo (Int x) { std::cout << "Creating int-like object\n";
}
  template <typename Float>
    requires std::is_floating_point<Float>::value
  Foo (Float x) { std::cout << "Creating float-like
object\n" }
};
```

Но тут кажется есть риск столкнуться с неясными ошибками если провалятся оба варианта (как было со `std::distance`)?

УЛУЧШЕНИЯ ДИАГНОСТИКИ

Если ни один вариант не подошёл, выводятся провалившиеся ограничения с каждого.

```
struct S{};
Foo fs(S{});
```

Ошибки будут выглядеть доходчиво.

```
note: constraints not satisfied
note: 'std::is_integral::value' evaluated to false
note: constraints not satisfied
note: 'std::is_floating_point::value' evaluated to false
```

ПОЛНОЕ ПОКРЫТИЕ

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 }
```

Это связано с особым статусом констрейнтов.

[over.dcl] two function declarations of the same name refer to the same function if they are **in the same scope** and have **equivalent parameter declarations** and **equivalent trailing requires-clauses**, if any.

ИНОГДА ЭТО ЗАХОДИТ СЛИШКОМ ДАЛЕКО

Выражения внутри requires не требуют даже constexpr evaluation.

```
consteval bool C() { return true; }

  template<typename T> struct A {
  int f() requires (C()) { return 1; }

  // this is not a redeclaration
  int f() requires true { return 2; }
};
```

То есть сравнение `requires clauses` идёт на этапе ODR до начала других семантических процессов.

НЕДОСТАТКИ SFINAE-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`

НЕДОСТАТКИ SFINAE-CONSTRAINTS

Увы, SFINAE определители не упорядочены в отношении ограниченности

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

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

НЕДОСТАТКИ SFINAE-CONSTRAINTS

- Итак, sfinae-constraints не упорядочены по ограниченности, они все одноранговые, это просто значение времени компиляции.
- Кроме того их бывает банально сложно и неприятно писать.

```
template <typename T, typename = void>
struct is_totally_ordered : std::false_type {};

template <typename T>
struct is_totally_ordered <T, std::void_t<
  decltype(std::declval<T>() == std::declval<T>()),
  decltype(std::declval<T>() <= std::declval<T>()),
  decltype(std::declval<T>() < std::declval<T>())
 >
> : std::true_type {};
```

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`.

ЕЩЕ ЛУЧШЕ ДИАГНОСТИКА

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

Выражение

```
check_eq(std::string{"1"}, 1);
```

Даёт

```
note: the required expression '(t == u)' would be ill-formed
```

Здесь сказано не только название констрейнта, но ещё и конкретный `illformed expression` в нём

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 валидность выражения.

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

СИНТАКСИС СЛОЖНОГО ОГРАНИЧЕНИЯ

- Думайте о сложном ограничении как о `constexpr` функции, возвращающей `bool`.

```
requires(T t, U u) {
  u + v; // true если u + v синтаксически возможно [simple]
  typename T::inner; // true если T::inner есть [type]
}
```

- О каждом ограничении внутри него думайте как о SFINAE-конъюнкте.
- Простые ограничения [simple] и ограничения типов [type] это простые варианты сложных ограничений.
- Есть ещё два: составные [compound] и вложенные [nested].

КОНЦЕПТЫ: CONVERTIBLE_TO

Не всегда удобно писать `requires-requires`.
Чтобы выделять системы ограничений, в C++20 введено специальное ключевое слово concept.

```
template<class From, class To>
concept convertible_to =
  std::is_convertible_v<From, To> &&
  requires(From (&f)()) { static_cast<To>(f()); };
```

Думайте о концепте как об аббревиатуре для `requires-expression`.

```
template <typename T> int foo(T x)
requires convertible_to<T, int> // requires concept
```

СОСТАВНЫЕ ОГРАНИЧЕНИЯ

- Составные ограничения проверяют совместимость типов с выражениями.

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

- Составное ограничение может использовать концепты.

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

- Также есть спецсинтаксис `noexcept`.

```
requires requires(T t) {
  { ++t } noexcept;
}
```

ВЛОЖЕННЫЕ ОГРАНИЧЕНИЯ

Внутри requires-expression оно может быть повторено. Это nested requirement.

```
requires(T t) {
  requires sizeof(T) == 4; // вычисляется [nested]
  requires somepred<T>() == 42; // consteval предикат [nested]
  requires noexcept(++t); // noexcept выражение [nested]
}
```

Упростите `requires-clause` на функции

```
template <typename T> int foo(T)
requires requires(T t) { requires noexcept(++t); } {
  return 42;
}
```

ПРИМЕР: EQUALITY COMPARABLE

- Мы можем пользоваться полной записью в явном виде.

```
template <typename T, typename U>
requires requires(T t, U u) {
    { t == u } -> std::convertible_to<bool>;
    { t != u } -> std::convertible_to<bool>;
    { u == t } -> std::convertible_to<bool>;
    { u != t } -> std::convertible_to<bool>;
  }
bool foo(T x, U y);
```

- Но хотелось бы где возможно выделять концепции и переиспользовать их.

КОНЦЕПТЫ ИЗ КОНЦЕПТОВ

- На основе простых концептов (convertible_to) можно строить более сложные.

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

- Разумеется мы можем добавить ещё массу условий.

ОГРАНИЧЕНИЯ ФУНКЦИЙ КОНЦЕПТАМИ

- 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>;
```

ПРОИЗВОДНЫЕ КОНЦЕПТЫ

- Можно переиспользовать созданные концепты напрямую

```
template <typename T>
concept StrictTotallyOrdered =
  EqualityComparable<T> &&
  requires(const std::remove_reference_t<T>& a,
    const std::remove_reference_t<T>& b) {
  { a < b } -> convertible_to<bool>;
  { a > b } -> convertible_to<bool>;
};
```

- К сожалению, концепты нельзя ограничивать другими предикатами. Также нельзя использовать рекурсивные концепты.

СИНТАКСИС ЗАПИСИ С КОНЦЕПТАМИ

- Базовый синтаксис.

```
template <typename T> requires Sortable<T> void sort(T&);
```

- Шаблонный параметр.

```
template <Sortable T> void sort(T&);
```

- Довольно интересно сделан синтаксис ограничения при нескольких аргументах.

```
template <SomeConcept<int> T> struct S; // SomeConcept<int, T>
```

- Тут вы указываете все аргументы и оставляете один специализировать

ЧАСТИЧНАЯ СПЕЦИАЛИЗАЦИЯ

У концептов как и у многого другого есть частичный порядок.

```
template <typename T>
  requires Ord<T> || Void<T>
struct less;

template <Ord T> struct less<T> {
  bool operator()(T a, T b) const { return a < b; }
};

template <> struct less<void> {
  template <Ord T>
 bool operator()(T a, T b) const { return a < b; }
};
```

ОБСУЖДЕНИЕ

- За счёт чего может работать частичная специализация?
- Как компилятор понимает, что `Ord` более специален, чем `(Ord || Void)`?

КОНЬЮНКТЫ И ДИЗЬЮНКТЫ

Любой концепт состоит из атомарных ограничений, соединённых логическими операциями (с обычными ленивыми правилами для них).

```
template<typename T>
concept Strange = (sizeof(T) == 4) ||
  (requires() {{T::value} -> convertible_to<bool>} &&
  T::value == true);

template<typename T> requires Strange<T>
void f(T);

f(1); // ok (lazy rules)
```

ОТНОШЕНИЕ SUBSUMES

Сложные концепты можно написать так, чтобы они участвовали в отношениях большей или меньшей ограниченности.

"*A constraint **P** subsumes a constraint **Q** if and only if: for every disjunctive clause P_i in the disjunctive normal form of **P** P_i subsumes every conjunctive clause Q_j in the conjunctive normal form of **Q**"* [temp.constr.order]

```
template <typename T>
concept P = Q<T> || sizeof(T) == 4; // что вы думаете?

template <typename T>
concept P = Q<T> && R<T>; // P subsumes Q and R
```

АТОМАРНЫЕ КОНСТРЕЙНТЫ

- *an atomic constraint A subsumes another atomic constraint B if and only if A and B are identical*.
- Учтите, что идентичность констрейнтов это идентчиность выражений и у нас могут быть функционально эквивалентные, но не идентичные констрейнты.

```
template <typename T> constexpr bool Atomic = true;
template <typename T> concept C = Atomic<T>;
template <typename T> concept D = Atomic<T*> && true;
```

- Перегрузка по D и C это IFNDR. Это реальность. Нам конечно хотелось бы:

*A constraint **P** subsumes a constraint **Q** if and only if **Q** implies **P***

ОТНОШЕНИЕ SUBSUMES

- Сложные концепты можно написать так, чтобы они участвовали в отношениях большей или меньшей ограниченности.

```
template <typename I>
concept InputIterator = Iterator<I> &&
  requires { typename iterator_category_t<I>; } &&
  DerivedFrom<iterator_category_t<I>, input_iterator_tag>;

template <typename I>
concept ForwardIterator = InputIterator<I> &&
  Incrementable<I> && Sentinel<I, I> &&
  DerivedFrom<iterator_category_t<I>,
forward_iterator_tag>;
```

- И так далее. Очевидно, что `random access iterator` будет самым ограниченным.

ТЕПЕРЬ ПЕРЕГРУЗКА РАБОТАЕТ

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

template <RandomAccessIterator 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.

КОНТРПРИМЕР САТТОНА

- Рассмотрим следующую перегрузку алгоритма копирования.

```
template <InputIterator In, OutputIterator<value_type_t<In>> Out>
Out copy(In first, In last, Out out) {
  // реализация явным циклом
}

template <ContIterator In, ContIterator Out>
  requires MemCopyable<In, Out>
Out copy(In first, In last, Out out) {
  // реализация через memcpy
}
```

- Здесь таится неоднозначность. При этом довольно сложно проследить где именно.

КОНТРПРИМЕР САТТОНА

- Саттон предлагает не особо полагаться на subsumptions.

```
template <InputIterator In, OutputIterator<value_type_t<In>> Out>
Out copy(In first, In last, Out out) {
  if constexpr(MemCopyable<In, Out>) {
    // реализация через memcpy
   } else {
    // реализация явным циклом
   }
}
```

- В конце концов, часто ли мы открываем новые категории итераторов?

КОНЦЕПТЫ, О КОТОРЫХ МЫ МЕЧТАЛИ

В первых статьях о концептах, они были гораздо интереснее

```
concept EqualityComparable<typename T> {
  requires constraint Equal<T>; // syntactic
  requires axiom Equivalence_relation<Equal<T>, T>; // semantic
  // if x == y then for any Predicate p, p(x) == p(y)
  template <Predicate P> axiom Equality(T x, T y, P p) {
    x == y => p(x) == p(y);
  }
  // inequality is the negation of equality
  axiom Inequality(T x, T y) { (x != y) == !(x == y); }
};
```

ВОПРОС

- К чему приведет решение ограничивать нешаблонные функции?

```
long foo(long l) requires (sizeof(long) == 4) { /* .... */ }
long foo(long l) requires (sizeof(long) == 8) { /* .... */ }
```

- Попробуйте поэкспериментировать самостоятельно и ответить на этот вопрос

РЕКОМЕНДУЕМАЯ ЛИТЕРАТУРА

1. Бьерн Страуструп, Язык программирования C++/ ред. А. Боборыкин. – 4-е изд. - Москва: Издательство БИНОМ, 2023. – 1213 с.
2. Дональд Кнут Искусство программирования, том 1. Основные алгоритмы = The Art of Computer Programming, vol.1. Fundamental Algorithms. — 3-е изд. — М .: «Вильямс» , 2006. – 720 с.
3. Bjarne Stroustrup, Gabriel Dos Reis, Andrew Sutton – Concepts Lite, 2013
4. Roger Orr C++ Concepts Lite in Practice, ACCU, 2016
5. Andrew Sutton – Concepts in 60: everything you need to know about concepts, CppCon, 2018
6. Arthur O'Dwyer – Concepts as she is spoke, CppCon, 2018
7. Matias Pusz – C++ concepts and ranges, C++ meeting, 2018
8. Andreas Fertig – C++20 Templates: The next level: Concepts and more, CppCon, 2021