

Ограничения и концепты (C++20)

Проблема

SFINAE позволяет проверить выражение на "корректность" - допустимы ли операции над данными operandами.

```
template <class T, class = void>
struct Addable : std::false_type {};
```



```
template <class T>
struct Addable<T,
              std::void_t<decltype(std::declval<T>() + std::declval<T>())>>
: std::true_type {};
```



Проблема

Почему это выглядит так... не очень?

Использование SFINAE - это костыль, эксплуатирование механизма языка под нужды, для которых он не был задуман.

```
template <class T, class = void>
struct Addable : std::false_type {};
```



```
template <class T>
struct Addable<T,
              std::void_t<decltype(std::declval<T>() + std::declval<T>())>>
: std::true_type {};
```



requires выражение

`requires` выражение проверяет корректность вложенных выражений, а также выполнение требуемых свойств и возвращает `true / false` в зависимости от результата проверки.

`requires [(список параметров)] { <выражения/ограничения> }`

Выражения не вычисляются, а лишь анализируются!

```
template <class T>
inline constexpr bool is_addable = requires { std::declval<T>() + std::declval<T>(); };
```

```
template <class T>
inline constexpr bool is_subtractable = requires (T x) { x - x; };
```

```
std::cout << is_addable<int*> << ' ' << is_subtractable<int*> << '\n' ;
```

requires выражение

- в списке параметров можно перечислять произвольный набор параметров (в том числе пакеты аргументов) и они не обязаны быть шаблонными.

```
template <class... Args>
inline constexpr bool is_addable = requires (Args... args, int x) {
    (args + ...) + x; // результат суммы можно сложить с целым числом
};
```

- список параметров не может содержать аргументы по умолчанию (а смысл?)
- если проверяемое условие должно для любого шаблонного параметра (или вовсе не зависит от него), то программа считается некорректной (IFNDR)

```
template <class T>
inline constexpr auto is_addable = requires { "" + ""; }; // CE
```

requires выражение

requires выражение может содержать требования следующих видов:

1. Простые требования (simple requirements)
2. Типовые требования (type requirements)
3. Составные требования (compound requirements)
4. Вложенные требования (nested requirements)

requires выражение: простые требования

Простое требование - произвольное выражение, которое требуется проверить на корректность.

Все требования в примере - простые (по одному на строку):

```
template <class T, class U>
inline constexpr bool example = requires (T x, U y) {
    x + y;
    x - y;
    x * y;
    x = y;
    F(x, y, x / y);
};
```

requires выражение: типовые требования

Типовое требование - проверяет существование требуемого типа.

Начинается со слова `typename`, за которым следует имя типа:

```
template <class T, class U>
inline constexpr bool example = requires {
    typename std::vector<T>;
    typename T::value_type;
    typename A<T>::type;
};
```

requires выражение: составные требования

Составное требование - позволяет проверить дополнительные свойства выражения.

```
{ <выражение> } [noexcept] [ -> <ограничение> ]
```

```
template <class T, class U>
inline constexpr bool example = requires (T x, U y) {
    { x + y; }; // то же, что и простое требование
    { x * y; } noexcept; // верно ли, что умножение noexcept
    { x - y; } -> std::convertible_to<int>; // приводим ли результат к int
    { +x; } noexcept -> std::same_as<T>; // операция noexcept и возвращает T
};
```

requires выражение: вложенные требования

Вложенное требование - позволяет проверить выполнение некоторого условия (истина/ложь).

```
requires <условие>;
```

```
template <class T, class U>
inline constexpr bool example = requires (T x, U y) {
    std::is_base_of_v<T, U>; // проверяет можно ли инстанцировать переменную
    requires std::is_base_of_v<T, U>; // проверяет значение переменной

    sizeof(T) < 10; // проверяет корректность записи (всегда корректно)
    requires sizeof(T) < 10; // проверяет истинность условия

    requires !requires { x + y; }; // проверяет, что T и U НЕЛЬЗЯ складывать
};
```

Ограничения (C++20)

Ограничения (C++20)

Ключевое слово `requires` может быть использовано в объявлении шаблона. В этом случае следующее за ним условие определяет множество типов, для которых этот шаблон объявлен.

```
template <class T>
requires (sizeof(T) <= 4) // может идти после "шапки" шаблона
void Print(T) {
    std::cout << "Small type\n";
}
template <class T>
void Print(T x) requires (sizeof(x) > 4) { // либо после прототипа
    std::cout << "Large type\n";
}
```

```
Print(0); // Large type
```

Общий принцип похож на SFNAE: сначала отбираются кандидаты с пройденными требованиями, затем действуют правила выбора перегрузки.

Ограничения (C++20)

```
struct A {
    A();
    A(const A&);
    A(A&&);

    template <class T>
    requires (!std::is_same_v<std::remove_cvref_t<T>, A>)
    A(T&&);
};
```

```
template <class Iterator> requires IsIterator<Iterator>
void Advance(Iterator& it, int n) {
    while (n-- > 0) ++it;
    while (n++ < 0) --it;
}

template <class Iterator> requires IsRandomAccessIterator<Iterator>
void Advance(Iterator& it, int n) {
    it += n;
}
```

Ограничения (C++20)

Есть ли разница?

```
template <size_t N>
requires (Fibonacci(N) > 1000)
void F() { ... } // 1

template <size_t N>
requires requires { Fibonacci(N) > 1000; }
void F() { ... } // 2
```

Ограничения (C++20)

Есть ли разница?

```
template <size_t N>
requires (Fibonacci(N) > 1000)
void F() { ... } // 1

template <size_t N>
requires requires { Fibonacci(N) > 1000; }
void F() { ... } // 2
```

Ограничения (C++20)

Есть ли разница?

```
template <size_t N>
requires (Fibonacci(N) > 1000) // проверяет условие
void F() { ... } // 1

template <size_t N>
requires requires { Fibonacci(N) > 1000; } // проверяет допустимо ли выражение
void F() { ... } // 2
```

Ограничения (C++20)

К сожалению, ограничения просто так не упорядочиваются компилятором по отношению "более частный-более общий":

```
template <class T> requires (sizeof(T) <= 4)
void F(T);

template <class T> requires (sizeof(T) <= 4 && std::is_same_v<T, int>)
void F(T);
```

```
F(0); // СЕ: неоднозначный вызов, оба шаблона подходят
```

Но вот, на чем можно ввести порядок, так это на концептах и их композициях

Концепты (C++20)

Концепты (C++20)

Концепт - именованное требование.

```
template <class T>
concept SmallType = sizeof(T) <= 4;
```

```
template <class T>
requires SmallType<T>
void F(T x);
```

Последнее можно переписать так:

```
template <SmallType T>
void F(T x);
```

и даже так:

```
void F(SmallType auto x);
```

Концепты (C++20)

В качестве первого параметра концепта может выступать "подразумеваемый" параметр (очень полезно и удобно):

```
template <class T, class U>
concept ConvertibleTo = std::is_convertible_v<T, U>;
```

```
template <ConvertibleTo<int> T> // ConvertibleTo<T, int>
void F(T x);
```

```
template <ConvertibleTo<int>... Args>
int Sum(Args... args) {
    return (static_cast<int>(args) + ...);
}
```

```
requires {
    { x + y; } -> ConvertibleTo<int>; // requires ConvertibleTo<decltype(x + y), int>
}
```

Порядок на концептах (наконец-то матлог)

Концепты можно объединять с помощью конъюнкций и дизъюнкций (`&&` и `||`):

```
template <class T>
requires SmallType<T> && ConvertibleTo<T, int>
void F(T x);
```

Это может происходить и неявно:

```
template <SmallType T> requires ConvertibleTo<T, int>
void F(T x, IsClass auto y) requires requires { x + y; };

// эквивалентно преобразуется в
template <class T, class U>
void F(T x, U y)
requires SmallType<T> && ConvertibleTo<T, int> && IsClass<U> && requires { x + y; };
```

Порядок на концептах (наконец-то матлог)

- Каждое ограничение является либо атомарным ограничением (отдельным концептом либо булевским выражением), либо конъюнкцией ограничений, либо дизъюнкцией ограничений.
- Два атомарных ограничений являются эквивалентными если они представляют собой одинаковые концепты (с одним набором фактических параметров), либо **абсолютно одинаковые булевские выражения**

```
// это разные шаблоны, так как ограничения не эквивалентны!
template <class T> requires (sizeof(T) <= 4)
void F(T);

template <class T> requires (sizeof(T) < 5)
void F(T);
```

Порядок на концептах (наконец-то матлог)

Ограничение A включает в себя (является более частным чем) ограничение B , если из A следует B (в смысле языка C++).

Формально:

- A приводится к ДНФ, B - к КНФ
- Атомарное ограничение α включает атомарное ограничение $\beta \iff \alpha$ и β эквивалентны.
- Дизъюнкт α включает в себя конъюнкт $\beta \iff$ в α есть атомарное ограничение, которое включает в себя некоторое атомарное ограничение из β
- A включает $B \iff$ каждый дизъюнкт из A включает в себя каждый конъюнкт из B .

Порядок на концептах: примеры

```
// атомарное ограничение
template <class T>
concept Iterator = requires (T x) { ++x; *x; };

// ограничение-дизъюнкт
template <class T>
concept BidirectionalIterator = Iterator<T> && requires (T x) { --x; };

template <Iterator It>
void F(It); // 1

template <BidirectionalIterator It>
void F(It); // 2
```

```
std::vector<int> v{1, 2, 3, 4, 5};
std::forward_list<int> l{1, 2, 3, 4, 5};
F(v.begin()); // 2
F(l.begin()); // 1
```

Порядок на концептах: примеры

```
// атомарное ограничение
template <class T>
concept Iterator = requires (T x) { ++x; *x; };

// ограничение-дизъюнкт
template <class T>
concept BidirectionalIterator = Iterator<T> && requires (T x) { --x; };

template <Iterator It>
void F(It); // 1

template <BidirectionalIterator It>
void F(It); // 2
```

```
std::vector<int> v{1, 2, 3, 4, 5};
std::forward_list<int> l{1, 2, 3, 4, 5};
F(v.begin()); // 2
F(l.begin()); // 1
```

Порядок на концептах: примеры

```
// атомарное ограничение
template <class T>
concept Iterator = requires (T x) { ++x; *x; };

// атомарное ограничение
template <class T>
concept BidirectionalIterator = requires (T x) { ++x; *x; --x; };

template <Iterator It>
void F(It); // 1

template <BidirectionalIterator It>
void F(It); // 2
```

```
std::vector<int> v{1, 2, 3, 4, 5};
std::forward_list<int> l{1, 2, 3, 4, 5};
F(v.begin()); // СЕ: оба подходят, нет порядка на концептах
F(l.begin()); // 1
```

