extra12. Концепты в C++20. Синтаксис определения концептов, ключевое слово requires. Использование концептов в качестве параметров шаблонов. Определение концептов ForwardIterator, BidirectionalIterator, RandomAccessIterator. Решение проблемы с реализацией функции std::advance с помощью концептов.

Проблемы обобщённого программирования на C++:

1. *Ошибки при использовании шаблонов вылезают непонятно где, они трудны для восприятия.*

2. *Трудно писать разные реализации одной шаблонной функции для разных категорий типов.* Пусть, я хочу написать функцию, которая проверяет, что два числа достаточно близки друг к другу. Для целых чисел достаточно проверить, что числа равны между собой, для чисел с плавающей точкой — то, что разность меньше некоторого ε.

Задачу можно решить хаком SFINAE, написав две функции. Хак использует `std::enable_if`. Это специальный шаблон в стандартной библиотеке, который содержит ошибку в случае если условие не выполнено. При инстанцировании шаблона компилятор отбрасывает декларации с ошибкой:

```
#include

template
T Abs(T x) {
    return x >= 0 ? x : -x;
}

// вариант для чисел с плавающей точкой
template
std::enable_if_t, bool>
AreClose(T a, T b) {
    return Abs(a - b) < static_cast(0.000001);
}

// вариант для других объектов
template
std::enable_if_t, bool>
AreClose(T a, T b) {
    return a == b;
}
```

Обе проблемы легко решить, если добавить в язык всего одну возможность — накладывать **ограничения на шаблонные параметры**. Например, требовать, чтобы шаблонный параметр был контейнером или объектом, поддерживающим сравнение. Это и есть концепт.

Перепишем нашу плавающую точку при помощи концептов:

```
#include

template
T Abs(T x) {
    return x >= 0 ? x : -x;
}

// вариант для чисел с плавающей точкой
template
```

```
requires(std::is_floating_point_v)
bool AreClose(T a, T b) {
    return Abs(a - b) < static_cast(0.000001);
}

// вариант для других объектов
template
bool AreClose(T a, T b) {
    return a == b;
}
```

Аналогично можно расписать функцию печати контейнера:

```
#include
#include

template
concept HasBeginEnd =
    requires(T a) {
        a.begin();
        a.end();
    };

template
void Print(std::ostream& out, const T& v) {
    for (const auto& elem : v) {
        out << elem << std::endl;
    }
}

template
void Print(std::ostream& out, const T& v) {
    out << v;
}
```

Концепт — это имя для ограничения.

Мы свели его к другому понятию, определение которого уже содержательно, но может показаться странным:

Ограничение — это шаблонное булево выражение.

Грубо говоря, приведённые выше условия «быть итератором» или «являться числом с плавающей точкой» — это и есть ограничения. Вся суть нововведения заключается именно в ограничениях, а концепт — лишь способ на них ссылаться.

Для ограничений доступны булевы операции и комбинации других ограничений; в ограничениях можно использовать выражения и даже вызывать функции. Но функции должны быть constexpr — они вычисляются на этапе компиляции:

```
template
concept Integral = std::is_integral::value;

template
concept SignedIntegral = Integral &&
                         std::is_signed::value;
template
concept UnsignedIntegral = Integral &&
                           !SignedIntegral;

template
constexpr bool get_value() { return T::value; }

template
    requires (sizeof(T) > 1 && get_value())
```

```
void f(T); // 1

void f(int); // 2

void g() {
    f('A'); // вызывает 2.
```

Для ограничений есть отличная возможность: проверка корректности выражения — того, что оно компилируется без ошибок. Посмотрите на ограничение Addable. В скобках написано a + b. Условия ограничения выполняются тогда, когда значения a и b типа T допускают такую запись, то есть T имеет определённую операцию сложения:

```
template
concept Addable =
requires (T a, T b) {
    a + b;
};
```

Ограничение может требовать не только корректность выражения, но и чтобы тип его значения чему-то соответствовал. Здесь мы записываем:

выражение в фигурных скобках,

->,

другое ограничение.

```
template concept C1 =
requires(T x) {
    {x + 1} -> std::same_as;
};
```

Фулл

Концепты для итераторов:

```
template<class I>
  concept forward_iterator =
    std::input_iterator<I> &&
    std::derived_from</*ITER_CONCEPT*/<I>, std::forward_iterator_tag> &&
    std::incrementable<I> &&
    std::sentinel_for<I, I>;
```

фулл с пояснениями - 1

```
template<class I>
  concept bidirectional_iterator =
    std::forward_iterator<I> &&
    std::derived_from</*ITER_CONCEPT*/<I>, std::bidirectional_iterator_tag>
   &&
    requires(I i) {
      { --i } -> std::same_as<I&>;
      { i-- } -> std::same_as<I>;
    };
```

фулл с пояснениями - 2

```
template<class I>
  concept random_access_iterator =
    std::bidirectional_iterator<I> &&
    std::derived_from</*ITER_CONCEPT*/<I>, std::random_access_iterator_tag>
   &&
    std::totally_ordered<I> &&
    std::sized_sentinel_for<I, I> &&
    requires(I i, const I j, const std::iter_difference_t<I> n) {
      { i += n } -> std::same_as<I&>;
      { j +  n } -> std::same_as<I>;
```

```
      { n +  j } -> std::same_as<I>;
      { i -= n } -> std::same_as<I&>;
      { j -  n } -> std::same_as<I>;
      {  j[n]  } -> std::same_as<std::iter_reference_t<I>>;
    };
```

фулл с пояснениями - 3

Беды с std::advance

std::advance(iter, n) puts its iterator iter n position further. Depending on the iterator, the implementation can use pointer arithmetic or just go n times further. In the first case, the execution time is constant; in the second case, the execution time depends on the stepsize n. Thanks to concepts, you can overload std::advance on the iterator category.

```
template<InputIterator I>
void advance(I& iter, int n){...}

template<BidirectionalIterator I>
void advance(I& iter, int n){...}

template<RandomAccessIterator I>
void advance(I& iter, int n){...}

// usage

std::vector<int> vec{1, 2, 3, 4, 5, 6, 7, 8, 9};
auto vecIt = vec.begin();
std::advance(vecIt, 5);       // RandomAccessIterator

std::list<int> lst{1, 2, 3, 4, 5, 6, 7, 8, 9};
auto lstIt = lst.begin();
std::advance(lstIt, 5);       // BidirectionalIterator

std::forward_list<int> forw{1, 2, 3, 4, 5, 6, 7, 8, 9};
auto forwIt = forw.begin();
std::advance(forwIt, 5);      // InputIterator
```

фулл