Концепты

В C++20 появилась новая сущность, объявляемая ключевым словом **concept**

```
template <typename T>
concept NothrowDefaultConstructible = noexcept(T{});
```

concept задает compile-time булевский предикат над шаблонными параметрами

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

Концепты

Шаблонные параметры могут быть и нетиповыми

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

Можно использовать их и вперемешку

```
template<typename T, size_t MaxSize>
concept SmallerThan = sizeof(T) < MaxSize;
```

Использование концептов

Но чем вообще концепт тогда отличается от **`constexpr bool`**?

Концепты можно использовать везде, где нужен bool на этапе компиляции

```
static_assert(SmallerThan<int, 6>);
```

```
template<typename T>
void f() noexcept(NothrowDefaultConstructible<T>) {
    T t;
    // ...
}
```

Использование концептов

Также, концепты можно использовать для ограничения шаблонного параметра

```
template <class T>
concept DefaultConstructible = std::is_default_constructible_v<T>;
```

```
template <DefaultConstructible T>
struct S {
    T value{};
};
```

Использование концептов

Возможно и частичное применение концептов

```
template<SmallerThan<5> T>
void f(T a) {

}
```

```
f(5);     // OK
f(5ll);   // Compilation Error
```

Предыдущую запись можно упростить с помощью **auto**

```
void f(SmallerThan<5> auto a) {

}
```

Использование концептов

Концепт можно применить даже перед auto для возвращаемого значения

```
template<typename T>
concept SmallerThanPointer = SmallerThan<T, sizeof(void *)>;

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

Использование концептов

В случае неудовлетворения условию концепта, компилятор выдаст гораздо более читаемую ошибку

```
std::list<int> l = {3, -1, 10};
std::sort(l.begin(), l.end());
//  error: cannot call std::sort with std::_List_iterator<int>
//  note:  concept RandomAccessIterator<std::_List_iterator<int>> was not satisfied
```

Использование концептов

Концепты не могут ссылаться на себя и их нельзя ограничить

```
template<typename T>
concept V = V<T*>; // Ошибка: рекурсивный концепт

template<class T>
concept C1 = true;
template<C1 T>
concept Error1 = true; // Ошибка: C1 T ограничивает концепт
```

requires

В концептах можно задавать более сложные условия при помощи конструкции **requires**

```
template<typename T, typename... Args>
concept Constructible = requires(Args... args) { T{args...}; };

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

requires

В requires можно указывать требования и на существование типа

```
template<typename T>
concept C = requires { typename T::inner; }
```

requires

Теперь попробуем ограничить функцию sort

```
template<typename Iterator>
concept RandomAccessIterator = BidirectionalIterator<Iterator> && requires /* ... */;

template<typename It>
concept Sortable = RandomAccessIterator<It> && Comparable<ValueType<It>>;

template<Sortable Iter>
void sort(Iter first, Iter last) {/* ... */}
```

requires

requires можно использовать также и для ограничения

```
template<typename It>
concept Sortable = RandomAccessIterator<It> && Comparable<ValueType<It>>;

template<typename Iter> requires Sortable<Iter>
void sort(Iter first, Iter last) {/* ... */}
```

requires

Теперь концепт `Sortable` можно и не объявлять

```
template<typename Iter> requires
  RandomAccessIterator<It> && Comparable<ValueType<It>>
void sort(Iter first, Iter last) {/* ... */}
```

requires

```
template<typename T>
concept Addable = requires (T x) { x + x; };

template<typename T> requires Addable<T>
T add(T a, T b) { return a + b; }

template<typename T>
requires requires (T x) { x + x; }
T add(T a, T b) { return a + b; }
```