# Compile-time вычисления

Попробуем реализовать compile-time **if**

```
template <bool b>
void If() {}

template <>
void If<true>() {f();};
```

```
If<x == 5>();
```

# Пример

Можно посчитать числа Фибоначчи на этапе компиляции

```
template <int N>
struct Fibonacci {
    static const int value = Fibonacci<N - 2>::value + Fibonacci<N - 1>::value;
};

template <>
struct Fibonacci<0> {
    static const int value = 0;
};

template <>
struct Fibonacci<1> {
    static const int value = 1;
};
```

# Ключевое слово constexpr

Ключевое слово **`constexpr`** требует того, чтобы выражение было посчитано на этапе компиляции

```
constexpr int x = 5;
If<x == 5>();
```

# Ключевое слово constexpr

**constexpr** применим и к функциям

```
constexpr int fibonacci(int n) {
    return n <= 1 ? n : fibonacci(n - 1) + fibonacci(n - 2);
}
```

**constexpr** не запрещает вызывать функцию
от не-**constexpr** параметров

```
fibonacci(10);   // compile-time
fibonacci(x);    // run-time
```

# if constexpr

В C++17 добавили полноценный compile-time **`if`**

```
constexpr int x = 5;
if constexpr (x == 5) {
    f();
} else {
    g();
}
```

# type_traits

Ранее уже упоминались некоторые функции из заголовочного файла `type_traits`:

`std::remove_reference`
`std::remove_const`

# std::is_const

Позволяет узнать, является ли данный тип const

```
template <class T>
struct is_const {
    static constexpr bool value = false;
};

template <class T>
struct is_const<const T> {
    static constexpr bool value = true;
};
```

```
const int a = 1;
if (is_const<decltype(a)>::value) {}
```

# std::true_type, std::false_type

Значения **true** и **false**, оформленные в виде типов

```
struct true_type {
    static constexpr bool value = true;
};

struct false_type {
    static constexpr bool value = false;
};
```

```
template <class T>
struct is_const : std::false_type {};

template <class T>
struct is_const<const T> : std::true_type {};
```

# std::is_same

Как проверить на этапе компиляции равенство двух типов?

```
template <class U, class V>
struct is_same : std::false_type {};

template <class U>
struct is_same<U, U> : std::true_type {};
```

```
if (is_same<decltype(a), decltype(b)>::value) {}
```

# std::conjunction

Логическое И для типов

```
template <class U, class V>
struct conjunction : false_type {};
template <>
struct conjunction<true_type, true_type> : true_type {};
```

Подобная реализация не сработает в таком случае:

```
if (conjunction<true_type, is_const<decltype(a)>>::value) {}
```

Можно исправить так:

```
template <class U, class V>
struct conjunction {
    static constexpr bool value = U::value && V::value;
};
```

# std::conditional

Тернарный оператор для типов

```
template <bool Cond, class U, class V>
struct conditional {
    typedef U type;
};

template <class U, class V>
struct conditional<false, U, V> {
    typedef V type;
};
```

```
conditional<(q > 20), long long, int>::type x;
```

# std::conjunction

`conjunction` в `std` поддерживает переменное число аргументов

```
template <class U1, class... Args>
struct conjunction<U1, Args...>
: conditional<U1::value, conjunction<Args...>, U1>::type {};

template <class U1>
struct conjunction<U1> : U1 {};
```

# std::rank

Позволяет узнать размерность массива

```
template <class T>
struct rank {
    static constexpr size_t value = 0;
};
template <class T, size_t Size>
struct rank<T[Size]> {
    static constexpr size_t value = rank<T>::value + 1;
};
template <class T>
struct rank<T[]> {
    static constexpr size_t value = rank<T>::value + 1;
};
```

```
rank<int[4][6]>::value == 2
```