SFINAE

SFINAE

SFINAE (Substitution Failure Is Not An Error) - "неудачная подстановка - не ошибка" правило языка, которое гласит, что если в результате подстановки шаблонных параметров в объявление функции или частичной специализации класса возникает некорректная конструкция, то это не приводит к ошибке компиляции. Компилятор просто отбрасывает этот шаблон/специализацию из списка кандидатов.

```
template <class T>
T& F(int);

template <class T>
void F(long);

F<char>(0);  // char& F(int) (выигрывает первый из-за более подходящего типа аргумента)
F<void>(0);  // void F(long) (первый даже не рассматривается - не существует void&)
```

SFINAE

SFINAE (Substitution Failure Is Not An Error) - "неудачная подстановка - не ошибка" правило языка, которое гласит, что если в результате подстановки шаблонных параметров в объявление функции или частичной специализации класса возникает некорректная конструкция, то это не приводит к ошибке компиляции. Компилятор просто отбрасывает этот шаблон/специализацию из списка кандидатов.

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

template <class T>
struct IsReferenceable<T&> : std::true_type {};

IsReferenceable<int>::value;   // true
IsReferenceable<void>::value;  // false (специализация даже не рассматривается)
```

SFINAE: еще примеры

```
template <size_t N, class = int[N % 2]> void A(int);  // 1
template <size_t N> void A(long);  // 2

A<5>(0);  // 1
A<2>(0);  // 2 (c-style массивы нулевой длины запрещены)
```

```
template <class T> typename T::value_type B(int);  // 1
template <class T> T B(long);  // 2

B<std::vector<int>>(0);  // 1
B<int>(0);  // 2
```

```
template <class T> auto C(T x, T y) -> decltype(x + y);  // 1
template <class T, class U> T C(T x, U y);  // 2

C("aba", "caba");  // 2 (нельзя сложить два const char*)
C(1, 2);  // 1
```

не SFINAE

Тело шаблонной функции **не** является *SFINAE*-контекстом!

```
template <size_t N> void A(int) { int arr[N]; }       // 1
template <size_t N> void A(long) { int arr[N + 1]; }  // 2

A<0>(0);  // CE (hard error)
A<5>(0);  // 1
```

```
template <class T> void B(int) { typename T::value_type x = 0; }  // 1
template <class T> void B(long) {} // 2

B<std::vector<int>>(0);  // 1
B<int>(0);  // CE (hard error)
```

```
template <class T> void C(T x, T y) { x + y; } ;  // 1
template <class T, class U> void C(T x, U y) {};  // 2

C("aba", "caba");  // CE (hard error)
C(1, 2);  // 1
```

SFINAE: применение не по назначению

SFINAE - полезный механизм языка, у которого был обнаружен интересный побочный эффект. **С помощью него можно метапрограммировать!**

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

template <class T>
struct IsReferenceable<T&> : std::true_type {};
```

SFINAE метапрограммирование

Детекторы

С помощью `SFINAE` можно выявлять допустимость тех или иных конструкций с типами:

```
template <class T>
decltype(std::declval<T>().size(), std::true_type{}) Test(int) noexcept;
template <class T>
std::false_type Test(long) noexcept;

template <class T>
struct HasSize : decltype(Test<T>(0)) {};
```

```
HasSize<int>::value;                // false
HasSize<std::vector<int>>::value;   // true
HasSize<int[5]>::value;             // false
```

Детекторы

С помощью `SFINAE` можно выявлять допустимость тех или иных конструкций с типами:

```
template <class T>
decltype(std::declval<T>().size(), std::true_type{}) Test(int) noexcept;
template <class T>
std::false_type Test(long) noexcept;

template <class T>
struct HasSize : decltype(Test<T>(0)) {};
```

```
HasSize<int>::value;                // false
HasSize<std::vector<int>>::value;   // true
HasSize<int[5]>::value;             // false
```

Детекторы: IsAssignable

Задача: проверить, можно ли элементу одного типа присвоить элемент второго.

Детекторы: IsAssignable

Задача: проверить, можно ли элементу одного типа присвоить элемент второго.

```
template <class T, class U>
decltype(std::declval<T>() = std::declval<U>(), std::true_type{})
Test(int) noexcept;

template <class, class>
std::false_type Test(long) noexcept;

template <class T, class U>
struct IsAssignable : decltype(Test<T, U>(0)) {};
```

```
IsAssignable<int, int>::value;     // false
IsAssignable<int&, int>::value;    // true
IsAssignable<int&, char>::value;   // true
IsAssignable<int&, char*>::value;  // false
```

Детекторы: IsNothrowMoveConstructible

Задача: проверить, существует ли небросающий конструктор перемещения.

Детекторы: IsNothrowMoveConstructible

Задача: проверить, существует ли небросающий конструктор перемещения.

```
template <class T>
std::bool_constant<noexcept(new(std::declval<T*>()) T(std::declval<T&&>()))>
Test(int) noexcept;

template <class, class...>
std::false_type Test(long) noexcept;

template <class T, class... Args>
struct IsNothrowMoveConstructible : decltype(Test<T>(0)) {};
```

```
IsNothrowMoveConstructible<int>::value;           // true
IsNothrowMoveConstructible<std::string>::value;   // true
IsNothrowMoveConstructible<std::istream>::value;  // false
```

Детекторы: IsPolymorphic

Задача: проверить, является ли класс полиморфным.

Детекторы: IsPolymorphic

Задача: проверить, является ли класс полиморфным.

dynamic_cast к `void*` *возвращает указатель на самого глубокого потомка полиморфного класса, на который указывает аргумент*

```
template <class T>
decltype(dynamic_cast<const volatile void*>(std::declval<T*>()), std::true_type{})
Test(int) noexcept;

template <class T>
std::false_type Test(long) noexcept;

template <class T>
struct IsPolymorphic : decltype(Test<T>(0)) {};
```

Детекторы: IsPublicBaseOf

Задача: проверить, является ли класс B публичным наследником A.

Детекторы: IsPublicBaseOf

Задача: проверить, является ли класс B публичным наследником A.

```
template <class T>
std::true_type ImaginaryFunction(T);

template <class A, class B>
decltype(ImaginaryFunction<A>(std::declval<B>())) Test(int) noexcept;

template <class A, class B>
std::false_type Test(long) noexcept;

template <class A, class B>
struct IsConvertible : decltype(Test<A, B>(0)) {};

template <class A, class B>
struct IsPublicBaseOf
  : std::bool_constant<std::is_class_v<A> && std::is_class_v<B> &&
                       IsConvertible<const volatile* A, B*>::value> {};
```

Детекторы: IsBaseOf*

Задача: проверить, является ли класс B наследником A.

Детекторы: IsBaseOf*

Задача: проверить, является ли класс B наследником A.

```
template <class T>
std::true_type ImaginaryFunction(const volatile* T);  // есть наследование

template <class>
std::false_type ImaginaryFunction(const volatile* void);  // нет наследования

template <class A, class B>
decltype(ImaginaryFunction<A>(std::declval<B*>())) Test(int) noexcept;

template <class A, class B>
std::true_type Test(long) noexcept;  // сюда попадаем, если наследование приватное

template <class A, class B>
struct IsBaseOf
  : std::bool_constant<std::is_class_v<A> && std::is_class_v<B> &&
                       decltype(Test<A, B>(0))::value> {};
```

`std::enable_if`

Проблема

Часто хочется сделать так, чтобы в зависимости от каких-то условий компилятор игнорировал данный шаблон:

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

  template <class T>
  A(T&&);
};
```

```
A a;
auto copy = a;
```

Для кого жиза? В чем проблема?

Проблема

Часто хочется сделать так, чтобы в зависимости от каких-то условий компилятор игнорировал данный шаблон:

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

  template <class T>
  A(T&&);
};
```

```
A a;
auto copy = a;  // выбирается шаблонный конструктор с T=A&
```

Можно прописать конструкторы на все ситуации (A&, const A&, A&&, const A&&).

Но может проще просто "отключить" шаблон на эти ситуации?

`std::enable_if`

`std::enable_if` - шаблонный класс, который параметризован булевским значением и типом `T` (по умолчанию - `void`), содержит внутренний тип `type`, если значение `true` и не содержит иначе.

```
template <bool B, class T = void>
struct enable_if {
  using type = T;
};

template <class T>
struct enable_if<false, T> {};

template <bool B, class T = void>
using enable_if_t = typename enable_if<B, T>::type;
```

std::enable_if

```cpp
template <class T>
std::enable_if_t<std::is_integral_v<T>, T> F(T x) { ... };  // 1

template <class T>
std::enable_if_t<std::is_floating_point_v<T>, T> F(T x) { ... };  // 2
```

```cpp
F(0);  // 1
F(0.0);  // 2
```

```cpp
template <class... Args, class = std::enable_if_t<(std::is_arithmetic_v<Args> && ...)>>
auto Sum(Args... args) {
  return (args + ...);
}
```

```cpp
Sum(0, 1, 4.5, 3);  // Ok
Sum(std::string("a"), std::string("ba"));  // CE
```

std::enable_if

Решение заявленной проблемы

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

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

```
A a;
auto copy = a;  // A(const A&), шаблон не рассматривается
```

`std::move_if_noexcept`

std::move_if_noexcept

Напомним, для чего используется данная функция (упрощенный пример):

```
template <class T>
void Realloc(T*& buffer, size_t size, size_t new_capacity) {
  const auto new_size = std::min(size, new_capacity);
  auto new_buffer = new T[new_capacity];
  try {
    for (size_t i = 0; i < new_size; ++i) {
      new_buffer[i] = std::move_if_noexcept(buffer[i]);
    }
  } catch (...) {
    delete[] new_buffer;
    throw;
  }
  delete[] std::exchange(buffer_, new_buffer);
}
```

Если перемещение безопасно, то перемещаем, если нет, то для строгой гарантии безопасности нужно копировать.

`std::move_if_noexcept`

Наконец-то можем разобрать реализацию функции `std::move_if_noexcept`

```
template <class T>
enable_if_t<is_nothrow_move_constructible_v<T> || !is_copy_constructible_v<T>, T&&>
move_if_noexcept(T& value) noexcept {
  return std::move(value);
}

template <class T>
enable_if_t<!is_nothrow_move_constructible_v<T> && is_copy_constructible_v<T>, const T&>
move_if_noexcept(T& value) noexcept {
  return value;
}
```

Но, кажется, что здесь многовато дублирования кода (в возвращаемом значении)

`std::conditional`

`std::conditional<B, T, F>` - шаблонный класс, который параметризован булевским значением `B` и параметрами-типами `T` и `F`, определяет внутренний тип `type` = `T` или `F` в зависимости от истинности `B`.

```
template <bool B, class T, class F>
struct conditional {
  using type = F;
};

template <class T, class F>
struct conditional<true, T, F> {
  using type = T;
};

template <class B, class T, class F>
using conditional_t = typename conditional<B, T, F>::type;
```

`std::move_if_noexcept`

Наконец-то можем разобрать реализацию функции `std::move_if_noexcept`

```
template <class T>
conditional_t<is_nothrow_move_constructible_v<T> || !is_copy_constructible_v<T>,
T&&,
const T&>
move_if_noexcept(T& value) noexcept {
  return std::move(value);
}
```

Бонус: `std::void_t` (deus ex machina)

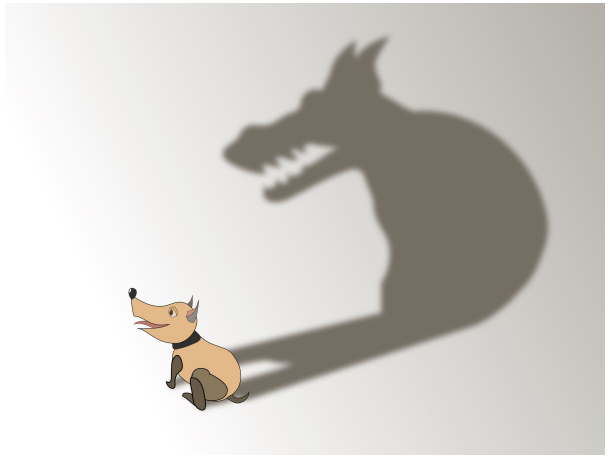
`std::void_t`

Это класс, который превращает все (или почти все) в `void`.

```
template <class...>
using void_t = void;
```

```
void_t<>;  // void
void_t<int>;  // void
void_t<void, int, char&>;  // void
```

И зачем? Причем тут SFINAE?



`std::void_t`

`std::void_t` можно использовать как тип, который существует, только при наличии каждого из его шаблонных параметров:

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

template <class T>
struct IsReferenceable<T, std::void_t<T&>> : std::true_type {};
```

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

template <class T>
struct HasSize<T, std::void_t<decltype(std::declval<T>().size())>>
  : std::true_type {};
```

std::void_t

```
template <class T, class = void>
struct IsAssignable : std::false_type {};
template <class T>
struct IsAssignable<T, std::void_t<decltype(std::declval<T>() = std::declval<T>())>>
  : std::true_type {};
```

```
template <class T, class = void>
struct HasValueType : std::false_type {};
template <class T>
struct HasValueType<T, std::void_t<typename T::value_type>> : std::true_type {};
```

```
template <class T, class = void>
struct HasFirstSecond : std::false_type {};
template <class T>
struct HasFirstSecond<T,
                      std::void_t<decltype(std::declval<T>().first),
                                  decltype(std::declval<T>().second)>> {};
```

Короче, `std::void_t` - универсальный детектор методов/полей/вложенных типов