

академия  
больших  
данных

mail.ru  
group

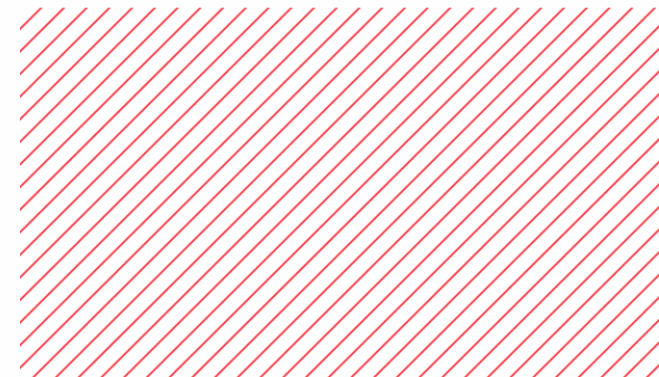
made

SFINAE

# Лекция 9

Гуров Роман

Advanced C++





# SFINAE

---

Substitution **F**ailure Is **N**ot **A**n **E**rror

Неудачная шаблонная подстановка – не ошибка компиляции



# Пример

---

Что вызовется?

```
template <typename T>  
typename T::type f(T x) {  
  
}
```

```
template <typename...>  
void f(...) {  
  
}
```

`f(2);`

# Пример

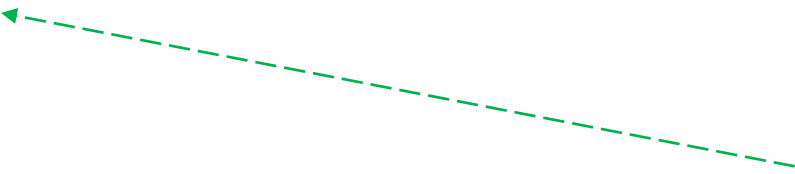
---

Точнее подходит первая функция

```
template <typename T>  
typename T::type f(T x) {  
  
}
```

```
template <typename...>  
void f(...) {  
  
}
```

f(2);



# Пример

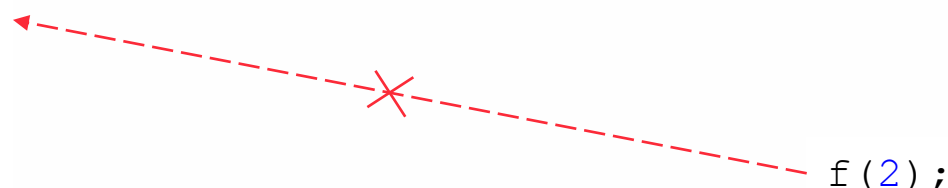
---

Но подстановка `int::type`  
невозможна. Ошибка компиляции?

```
template <typename T>  
typename T::type f(T x) {  
  
}
```

```
template <typename...>  
void f(...) {  
  
}
```

`f(2);`



# Пример

Ошибки не будет

```
template <typename T>  
typename T::type f(T x) {  
  
}
```

```
template <typename...>  
void f(...) {  
  
}
```

f(2);

Компилятор просто не будет  
рассматривать такие перегрузки

# Ремарка о шаблонных функциях

---

У шаблонных функций не бывает частичных специализаций.  
Бывают только полные (т.е. с **template** <>)

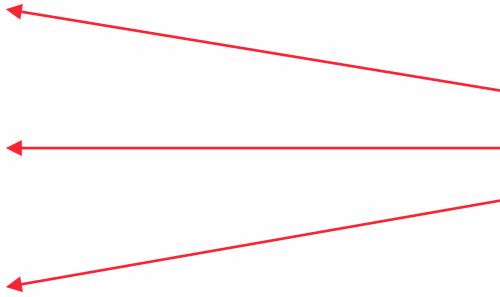
Вспомним пример с одной из прошлых лекций

```
template <class T1, class T2>  
void g(T1 x, T2 y);
```

```
template <class T>  
void g(int x, T y);
```

```
template <>  
void g(int x, int y);
```

```
int main() {  
    g(2., 2.);  
    g(2, 2.);  
    g(2, 2);  
}
```



# Ремарка о шаблонных функциях

---

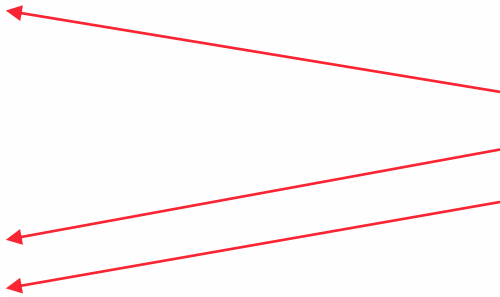
Если просто переставить определения местами,  
то специализация будет уже для другой перегрузки

```
template <class T1, class T2>  
void g(T1 x, T2 y);
```

```
template <>  
void g(int x, int y);
```

```
template <class T>  
void g(int x, T y);
```

```
int main() {  
    g(2., 2.);  
    g(2, 2.);  
    g(2, 2);  
}
```



Компилятор сначала выбирает наиболее подходящую перегрузку,  
и только потом смотрит на её шаблонные специализации.





# SFINAE

---

При выборе наиболее подходящей перегрузки функции, компилятор сначала отбрасывает все варианты, которые вообще невозможно вызвать от данных аргументов

Именно тут и определено правило SFINAE – компилятор пытается рассмотреть все сигнатуры перегрузок и, если какая-то не вывелась при подстановке шаблона, просто отбрасывает её

На ошибки подстановки в теле функции SFINAE не распространяется.

# std::enable\_if

---

Разрешает использование шаблонной перегрузки только при выполнении данного условия

```
template <typename T>
typename std::enable_if<std::is_class<T>::value, int>::type f(T x) {}
```

```
template <bool Cond, typename T = void>
struct enable_if {};
```

```
template <typename T>
struct enable_if<true, T> {
    typedef T type;
};
```

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



# std::is\_class

---

Позволяет узнать, является ли переданный тип классом

Что можно сделать с классом такого, что вызовет ошибку для не класса?

Объявить указатель на член класса:

```
int C::* p;
```



# std::is\_class

---

Позволяет узнать, является ли переданный тип классом

```
template <typename T>
std::true_type class_test(int T::*);

template <typename T>
std::false_type class_test(...);

template <typename T>
struct is_class : decltype(class_test<T>(nullptr)) {};
```

```
is_class<std::vector<int>>::value;    // true
is_class<int>::value;                // false
```



# std::is\_class

---

Но такая реализация пропустит и **union**, чего бы не хотелось

Для этого можно использовать класс `std::is_union`

```
template <typename T>
std::negation<std::is_union<T>> class_test(int T::*);

template <typename T>
std::false_type class_test(...);

template <typename T>
struct is_class : decltype(class_test<T>(nullptr)) {};
```



# std::allocator\_traits::construct

---

`allocator_traits::construct` умеет определять, есть ли метод `construct` у данного аллокатора, и вызывать `placement new`, если нет

Для этого существует внутренняя функция `has_construct`,  
проверяющая наличие метода

Но как проверить вызов метода прямо в сигнатуре функции?

```
decltype(/* expr */, std::true_type())
```

В таком случае вернется тип `std::true_type`,  
но выражение все равно будет обработано



# std::allocator\_traits::construct

---

Попробуем реализовать

```
template <class T, class... Args>
decltype(T().construct(Args()...), std::true_type()) construct_test(int);

template <class...>
std::false_type construct_test(...);

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

Осталась проблема: у `T` и `Args` может не быть  
конструкторов по умолчанию



# std::declval

---

`std::declval` используется в конструкциях **decltype** и возвращает объект данного типа

```
template <class T>  
T declval() noexcept;
```

Для использования в **decltype** функции даже не нужно тело





# std::allocator\_traits::construct

---

Теперь исправим `allocator_traits::construct`

```
template <class T, class... Args>
decltype(declval<T>().construct(declval<Args>()...), std::true_type())
construct_test(int);
```

```
template <class...>
std::false_type construct_test(...);
```

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



# std::is\_constructible

---

Проверяет, можно ли сконструировать тип от данного набора типов аргументов

```
std::is_constructible_v<MyClass, size_t, double*>
```

```
template <class T, class... Args>  
decltype(T(declval<Args>()...), std::true_type()) constructible_test(int);
```

```
template <class...>  
std::false_type constructible_test(...);
```

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

# Определение типа итератора

---

`std::vector` должен уметь различать переданные ему итераторы, чтобы иметь возможность применить различные оптимизации

```
template <class _InputIterator>
vector(_InputIterator __first,
       enable_if_t<__is_cpp17_input_iterator <_InputIterator>::value &&
                   !__is_cpp17_forward_iterator<_InputIterator>::value,
                   _InputIterator> __last);
```

```
template <class _ForwardIterator>
vector(_ForwardIterator __first,
       enable_if_t<__is_cpp17_forward_iterator<_ForwardIterator>::value,
                   _ForwardIterator> __last);
```



# Определение типа итератора

---

```
template <class _Tp, class _Up, bool = __has_iterator_category_v<iterator_traits<_Tp>>>
struct __has_iterator_category_convertible_to
: is_convertible<typename iterator_traits<_Tp>::iterator_category, _Up>
{};
```

```
template <class _Tp, class _Up>
struct __has_iterator_category_convertible_to<_Tp, _Up, false> : false_type {};
```

```
template <class _Tp>
struct __is_cpp17_input_iterator
: __has_iterator_category_convertible_to<_Tp, input_iterator_tag> {};
```

```
template <class _Tp>
struct __is_cpp17_forward_iterator
: __has_iterator_category_convertible_to<_Tp, forward_iterator_tag> {};
```



# is\_nothrow\_move\_constructible

---

Позволяет узнать, является ли move-конструктор класса **noexcept**

```
template <class T>
std::conditional_t<noexcept(T(std::move(declval<T>()))), true_type, false_type>
nothrow_test(int);
```

```
template <class...>
false_type nothrow_test(...);
```

```
template <class T>
struct is_nothrow_move_constructible : decltype(nothrow_test<T>(0)) {};
```



# Реализация std::move\_if\_noexcept

---

```
template <class T>
std::conditional_t<is_nothrow_move_constructible<T>::value, T&&, const T&>
move_if_noexcept(T& x) {
    return std::move(x);
}
```



# Неполные типы в declval

---

Текущая имплементация declval не работает с неполным типом

```
class C;  
int f(C&&);  
  
decltype(f(declval<C>())) x;
```

Поэтому по стандарту он возвращает T&&

```
template <class T>  
T&& declval() noexcept;
```

Ссылка на неполный тип не является неполным типом — всё хорошо



# std::is\_base\_of

---

Позволяет проверить, является ли один класс наследником другого

```
template <class B>
std::true_type test_pre_ptr_convertible(const B*);
template <class>
std::false_type test_pre_ptr_convertible(const void*);

template <class...>
std::true_type test_pre_is_base_of(...);
template <class B, class D>
decltype(test_pre_ptr_convertible<B>(static_cast<D*>(nullptr))) test_pre_is_base_of(int);

template <class Base, class Derived>
struct is_base_of : decltype(test_pre_is_base_of<Base, Derived>(0)) {};
```





# std::common\_type

---

Возвращает общий тип, к которому могут быть неявно преобразованы все переданные типы

```
template <class U, class V>
struct common_type {
    typedef decltype(false ? declval<U>() : declval<V>()) type;
};
```