

SFINAE

![image-20221217131241788](media/image-20221217131241788.png)

SFINAE - substitution failure is not an error

SFINAE

- **Substitution Failure Is Not An Error** (провал подстановки не является ошибкой)

```
template <typename T> T max(T a, T b);  
template <typename T, typename U> auto max(T a, U b);  
  
int g = max(1, 1.0); // подстановка в 1 провалена  
                  // подстановка в 2 успешна
```

- Если в результате подстановки в **непосредственном контексте** класса (функции, алиаса, переменной) возникает **невалидная конструкция**, эта подстановка неуспешна, но не ошибочна
- В этом случае второй фазы поиска имён просто не выполняется

27

непосредственный контекст класса - это может быть декларация. Если ошибка в теле, то это не SFINAE, а ошибка второй фазы.

SFINAE и ошибки

- Не любая ошибочная конструкция это SFINAE. Важен контекст подстановки.

```
int negate (int i) { return -i; }  
  
template <typename T> T negate(const T& t) {  
    typename T::value_type n = -t();  
    // тут используем n  
}  
  
negate(2.0); // ошибка второй фазы
```

- Здесь в контексте сигнатуры и шаблонных параметров нет никакой невалидности

28

Помним: разрешение зависимых имен откладывается до подстановки шаблонного параметра

Из этого можно получить SFINAE:

Обсуждение

- Техника SFINAE кажется очень простой, но вообще-то её приложения многочисленны и часто очень нетривиальны
- Рассмотрим задачу: у вас есть два типа и вам нужно определить равны ли они.

```
template <typename T, typename U> int foo() {  
    // как вернуть 1 если T == U и 0 если нет?  
}
```

- Обратим внимание, что это задача отображения из типов на числа.
- Прежде чем её решать, решим обратную задачу.

30

???

SFINAE используется вместе со специализацией и частичной специализацией.

Пример 1: отображение int-type

Каждому числу сопоставим тип: для каждого int есть type, мы делаем implicit cast типа к ЗНАЧЕНИЮ.

Интегральные константы

- Отображение из целых чисел на типы называется интегральной константой

```
template <typename T, T v> struct integral_constant {  
    static const T value = v;  
    typedef T value_type;  
    typedef integral_constant type;  
    operator value_type() const { return value; }  
};
```

- Возможна даже арифметика

```
using ic6 = integral_constant<int, 6>;  
auto n = 7 * ic6{};
```

31

определим некоторые такие интегральные типы:

Истина и ложь для типов

- Самые полезные из интегральных констант – самые простые

```
using true_type = integral_constant<bool, true>;
```

```
using false_type = integral_constant<bool, false>;
```

- Всё это есть в стандарте: `std::integral_constant` и т.д.

- Попробуем написать простой определитель, чтобы проверить одинаковые ли два типа

```
template<typename T, typename U>  
struct is_same : std::false_type {};
```

- По умолчанию разные. Что дальше?

32

а теперь запускаем SFINAE:

primary_template:

Истина и ложь для типов

- Самые полезные из интегральных констант – самые простые

```
using true_type = integral_constant<bool, true>;
```

```
using false_type = integral_constant<bool, false>;
```

- Всё это есть в стандарте: `std::integral_constant` и т.д.

- Попробуем написать простой определитель, чтобы проверить одинаковые ли два типа

```
template<typename T, typename U>  
struct is_same : std::false_type {};
```

- По умолчанию разные. Что дальше?

32

specialization:

Равенство типов

- Теперь можно решить задачу определения равенства типов

```
template<typename T, typename U>
struct is_same : std::false_type {};

template<typename T>
struct is_same<T, T> : std::true_type {}; // для T == T

template<typename T, typename U>
using is_same_t = typename is_same<T, U>::type;
```

- Благодаря SFINAE, будет работать

```
assert(is_same<int, int>::value && !is_same<char, int>::value);
```

33

<https://godbolt.org/z/6GdndGKzM>

провал подстановки

```
1 #include <type_traits>
2 #include <iostream>
3
4 template<typename T, typename U>
5 struct is_same : std::false_type {};
6
7 template<typename T>
8 struct is_same<T, T> : std::true_type {}; // for T == T
9
10 template<typename T, typename U>
11 using is_same_t = typename is_same<T, U>::type;
12
13 int main() {
14     std::cout << std::boolalpha;
15     std::cout << is_same_t<int, int>{} << std::endl;
16     std::cout << is_same_t<int, char>{} << std::endl;
17 }
```

прослеживается SFINAE триада: *primary, specialization, alias*

Определители и модификаторы

Определитель: является ли тип ссылкой

```
template <typename T> struct is_reference : false_type {};
template <typename T> struct is_reference<T&> : true_type {};
template <typename T> struct is_reference<T&&> : true_type {};
```

Модификатор: убираем ссылку с типа, если ссылки не было, то оставляем тип

```
template <typename T> struct remove_reference { using type = T; };
template <typename T> struct remove_reference<T&> { using type = T; };
template <typename T> struct remove_reference<T&&> { using type = T; };
```

Для модификатора полезен алиас

```
template <typename T>
using remove_reference_t = typename remove_reference<T>::type;
```

34

Четырнадцать категорий

- Любой тип в языке C++ попадает хотя бы под одну из перечисленных ниже категорий

```
is_void  
is_null_pointer  
is_integral, is_floating_point // для T и для cv T& транзитивно  
is_array; // только встроенные, не std::array  
is_pointer; // включая указатели на обычные функции  
is_lvalue_reference, is_rvalue_reference  
is_member_object_pointer, is_member_function_pointer  
is_enum, is_union, is_class  
is_function // обычные функции
```

- Использование довольно тривиально

```
std::cout << std::boolalpha << std::is_void<T>::value << '\n';
```

35

Обсуждение

- Техника SFINAE кажется очень простой, но вообще-то её приложения многочисленны и часто очень нетривиальны
- Рассмотрим задачу: у вас есть два типа и вам нужно определить равны ли они.

```
template <typename T, typename U> int foo() {  
    // как вернуть 1 если T == U и 0 если нет?  
}
```

- Обратим внимание, что это задача отображения из типов на числа.
- Прежде чем её решать, решим обратную задачу.

38

Свойства типов

- Также очень полезны определители свойств типов

```
is_trivially_copyable // побайтово копируемый, метасру  
is_standard_layout // можно адресовать поля указателем  
is_aggregate // доступна агрегатная инициализация как в C  
is_default_constructible // есть default ctor  
is_copy_constructible, is_copy_assignable  
is_move_constructible, is_nothrow_move_constructible  
is_move_assignable  
is_base_of // B является базой (транзитивно, включая сам тип)  
is_convertible // есть преобразование из A к B
```

- И многие другие (их реально десятки)

36

Магистерская часть

Разрешение частичного порядка/ перегрузки

В процессе разрешения имен - вывод типов

Двухфазное разрешение имен

Линкеры исключают неиспользованные функции

Инстанцирование - порождение специализации

Как может провалиться инстанцирование?

Через вывод типов

Через синтаксически некорректный контекст

SFINAE

- Substitution Failure Is Not An Error (провал подстановки не является ошибкой).

```
template <typename T> T max(T a, T b); // 1
template <typename T, typename U> auto max(T a, U b); // 2

int g = max(1, 1.0); // подстановка в 1 провалена
                     // подстановка в 2 успешна
```

- Если в результате подстановки в **непосредственном контексте** класса (функции, алиаса, переменной) возникает **невалидная конструкция**,
- То эта подстановка неуспешна, но не ошибочна.

7

SFINAE и ошибки

- Не любая ошибочная конструкция это SFINAE. Важен контекст подстановки.

```
int negate (int i) { return -i; }

template <typename T> typename T::value_type negate(const T& t) {
    typename T::value_type n = -t();
    // тут используем n
}

negate(2.0); // substitution failure
```

- Выводится T → double и, разумеется, T::value_type невалидно.
- Здесь нет ошибки, это провал подстановки и будет вызвана менее подходящая нешаблонная функция.

9

<https://godbolt.org/z/rzzogboWx>

typename для дизамбигуации

Это означает условный переход на этапе компиляции. Т.е. метопрограммирование на уровне инстанцирования.

Пример 2

Несистемное SFINAE. HasFooBar.

- С ранних пор была замечена полезность техники SFINAE для трюков и хаков. Классический пример: определить наличие зависимого типа в классе.

```
struct foo { typedef float foobar; };  
struct bar { };
```

```
cout << boolalpha << нечто от foo << " " << нечто от bar << endl;
```

- Без SFINAE, задача выглядит не решаемой, но решение возможно и даже в примитивном виде оно довольно красиво.

10

```
template <typename T> struct has_typedef_foobar {  
    using yes = char[1];  
    using no = char[2];  
    template <typename C> static yes& test(typename C::foobar*);  
    template <typename> static no& test(...);  
    enum { value = (sizeof(test<T>(0)) == sizeof(yes)) };  
};  
  
struct foo { using foobar = float; };  
struct bar { };  
  
TEST(sfinae, hasfoobar) {  
    EXPECT_EQ(has_typedef_foobar<foo>::value, true);  
    EXPECT_EQ(has_typedef_foobar<bar>::value, false);  
}
```

```
1  #include <concepts>  
2  #include "gtest/gtest.h"  
3  
4  template <typename T> struct has_typedef_foobar {  
5      using yes = char[1]; // для гарантии отработки sizeof во время компиляции  
6      using no = char[2];  
7      // две перегрузки (тип C для запуска SFINAE в теле структуры, T просто  
      // подставляется тут уже)  
8      template <typename C> static yes& test(typename C::foobar*);  
9      // здесь шаблон, потому что мы в енуме задан шаблонный параметр  
10     // если в енуме его убрать, то будет другая проблема - не сработает  
    вывод типов: из нуля вывод C не сработает. И в обоих случаях будет выбрана  
    (...)  
11     template <typename> static no& test(...); // va_args  
12     // do not require def outside class  
13     enum { value = (sizeof(test<T>(0)) == sizeof(yes)) };  
14 };  
15  
16 struct foo { using foobar = float; };
```



```

17 struct bar { };
18
19 TEST(sfinae, hasfoobar) {
20     EXPECT_EQ(has_typedef_foobar<foo>::value, true);
21     EXPECT_EQ(has_typedef_foobar<bar>::value, false);
22 }

```

<https://godbolt.org/z/KPdbjoodM>

Обсуждение

- При инстанцировании происходит **подстановка** шаблонного параметра
- Иногда она ещё и предваряется **выводом типа**
- Но что если подстановка в некотором контексте не может быть выполнена?

```

template <typename T> T max(T a, T b);
// требуется инстанцирование max, но вывод типов провален
int g = max(1, 1.0);

```

6

Draft

Решение проблемы std::copy

- Заведём хелпер и его специализацию для true
- ```

template<bool Triv, typename In, typename Out> struct CpSel {
 static Out select(In begin, In end, Out out)
 { return CopyNormal(begin, end, out); }
};

template<typename In, typename Out>
struct CpSel<true, In, Out> {
 static Out select(In begin, In end, Out out)
 { return CopyFast(begin, end, out); } // для простых типов
};

```
- Теперь сам алгоритм копирования будет просто решать кого он вызывает

38

Чтобы определить метод - мы реализуем переключатель



## Решение проблемы std::copy

- Также тривиально мы решаем проблему с копированием

```
template<typename In, typename Out>
Out realistic_copy(In begin, In end, Out out) {
 using in_type = pointee type (In); // как это написать?
 using out_type = pointee type (Out);

 enum { Sel = std::is_trivially_copyable<in_type>::value &&
 std::is_trivially_copyable<out_type>::value &&
 std::is_same<in_type, out_type>::value };

 return CpSel<Sel, In, Out>::select(begin, end, out);
}
```

39

[benchcopy-2.cc](#)

```
7 using in_type = typename std::iterator_traits<In>::value_type;
8 using out_type = typename std::iterator_traits<Out>::value_type;
9 enum {
10 Sel = std::is_trivially_copyable<in_type>::value &&
11 std::is_trivially_copyable<out_type>::value &&
12 std::is_same<in_type, out_type>::value };
13
14 return CpSel<Sel, In, Out>::select(begin, end, out);
15 }
```

Перегруженный оператор \*