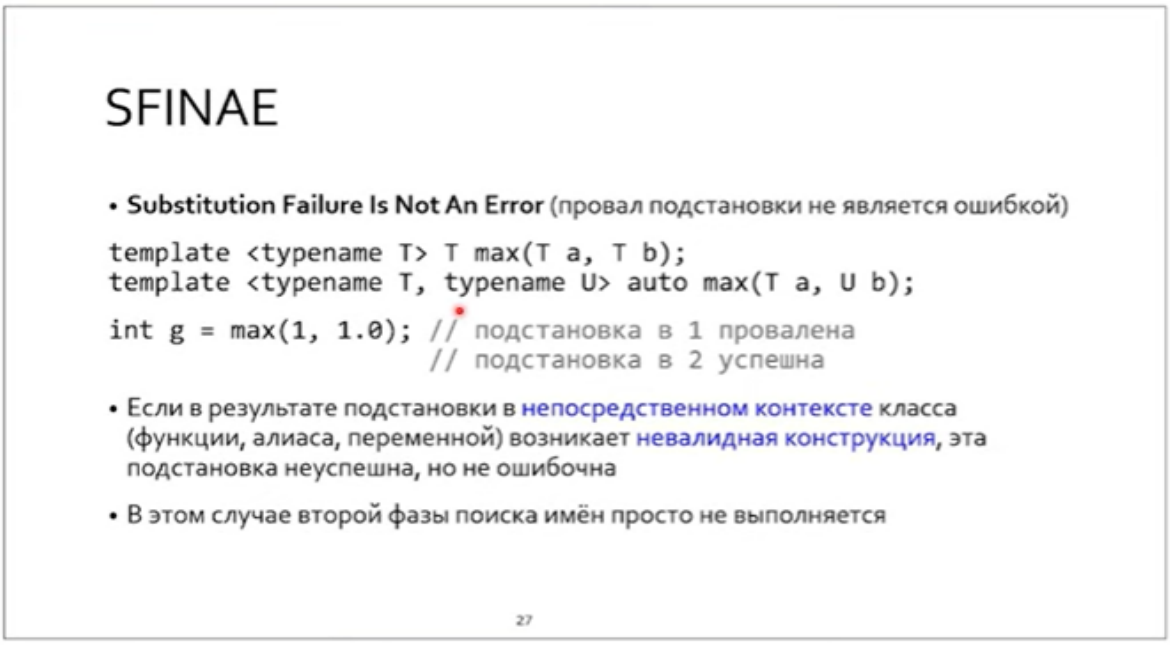
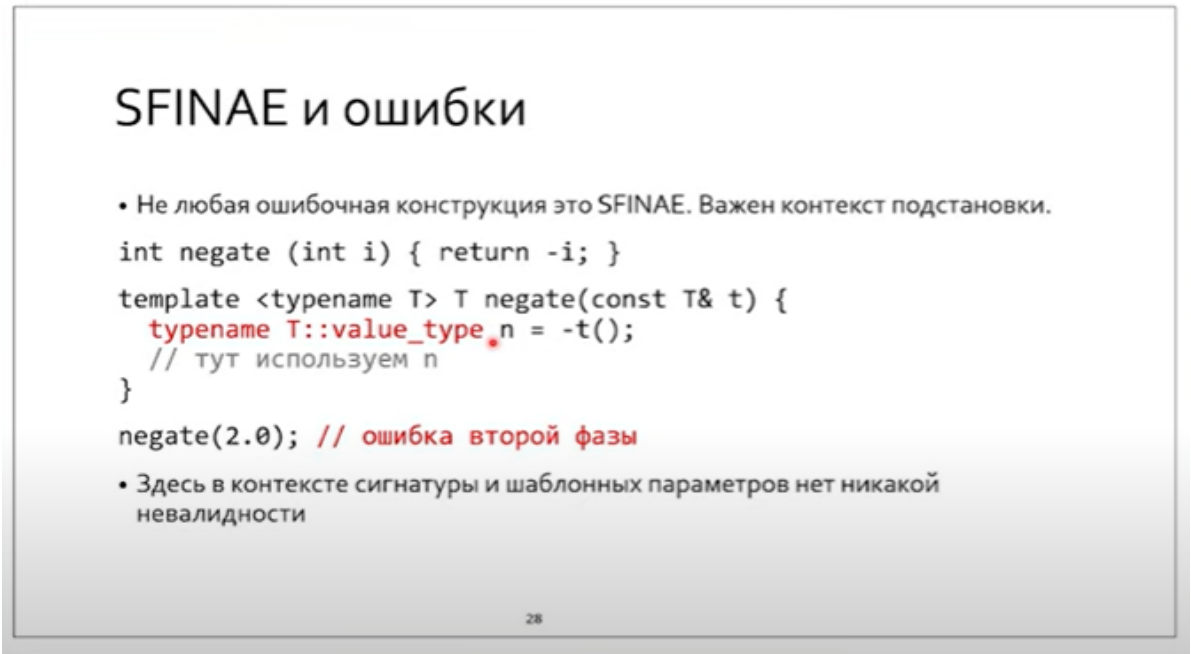
SFINAE

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

SFINAE - substitution failure is not an error

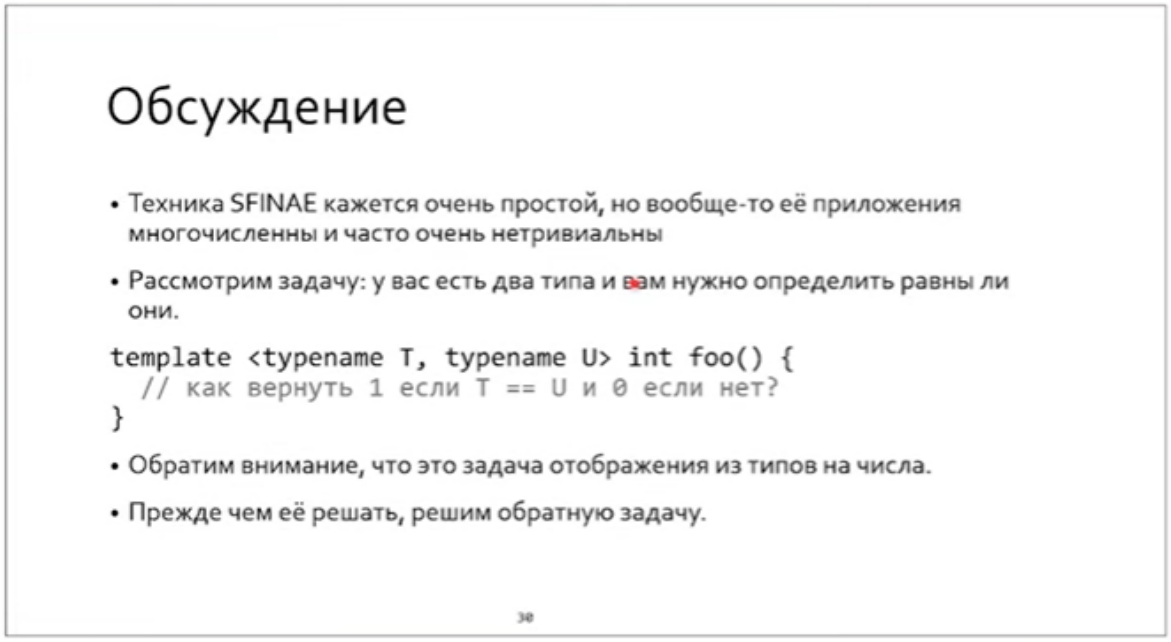


непосредственный контекст класса - это может быть декларация. Если ошибка в теле, то это не SFINAE, а ошибка второй фазы.



> Помним: разрешение зависимых имен откладывается до подстановки шаблонного параметра

Из этого можно получить SFINAE:

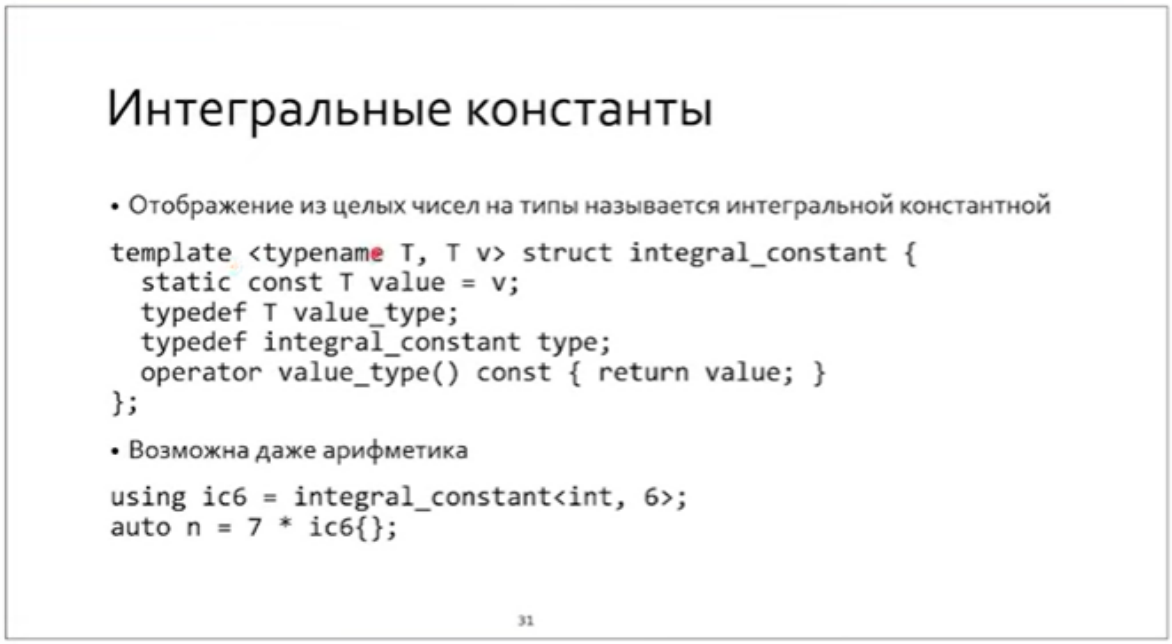


???

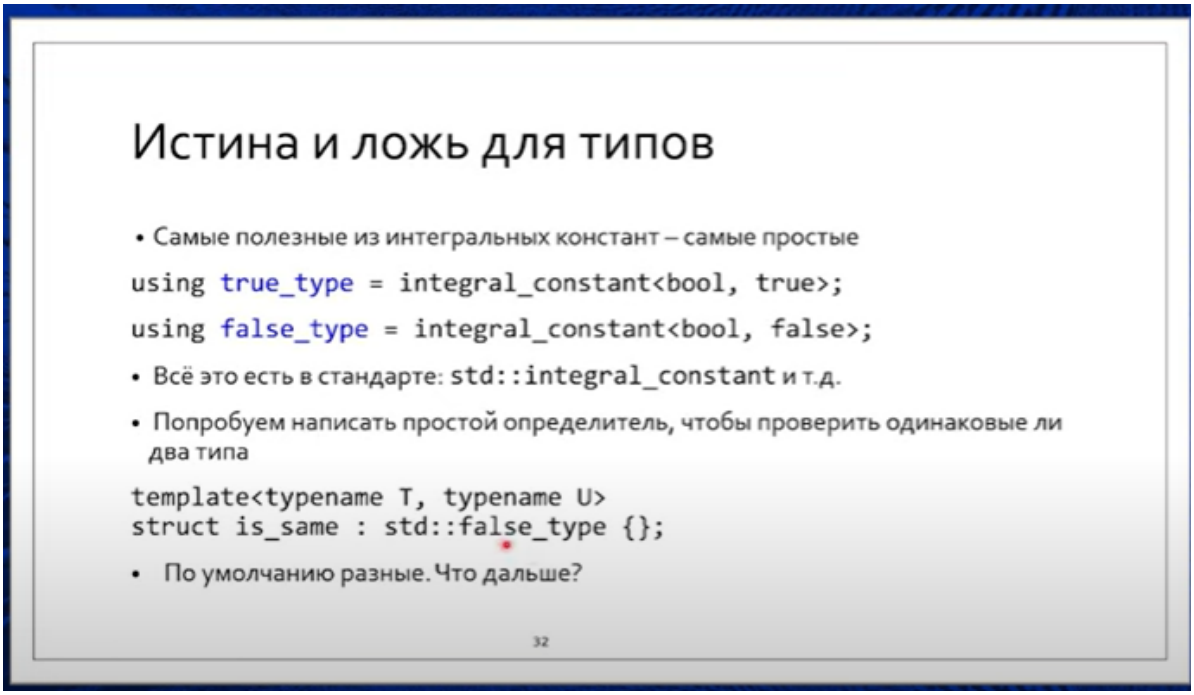
SFINAE используется вместе со специализацией и частичной специализацией.

Пример 1: отображение int-type

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

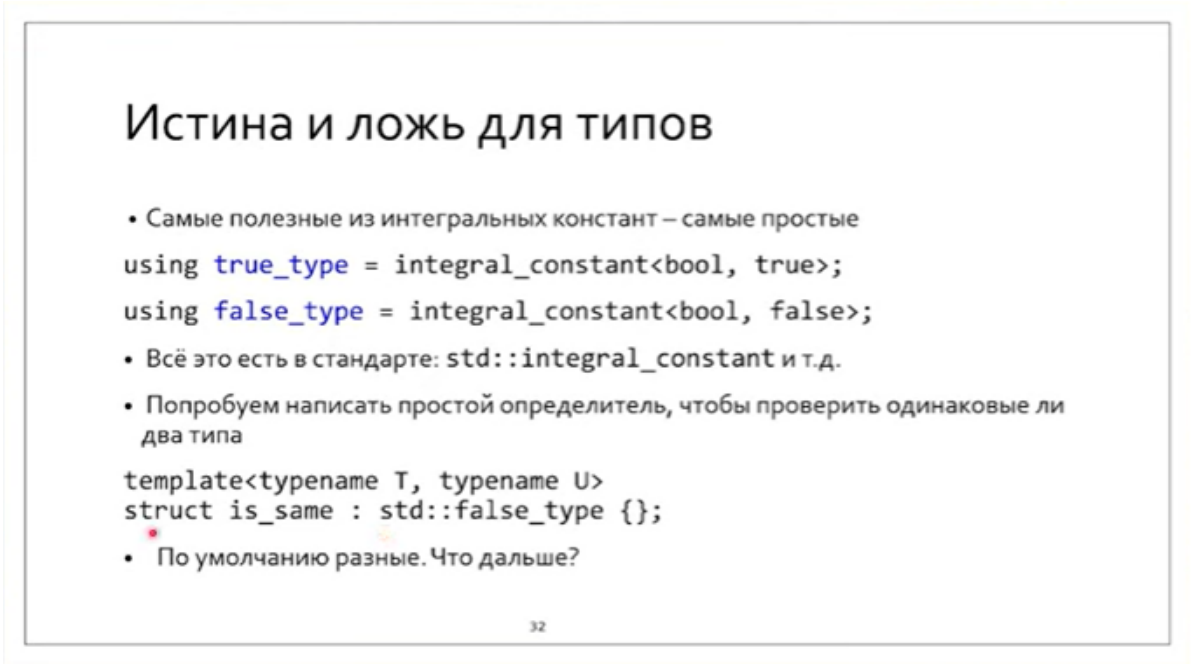


определим некоторые такие интегральные типы:



а теперь запускаем SFINAE:

primary_template:



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.

30

Свойства типов

- Также очень полезны определители свойств типов

```
is_trivially_copyable // побайтово копируемый, memcpy
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

Реализуем is_same

Магистерская часть

Разрешение частичного порядка/ перегрузки

В процессе разрешения имен - вывод типов

Двухфазное разрешение имен

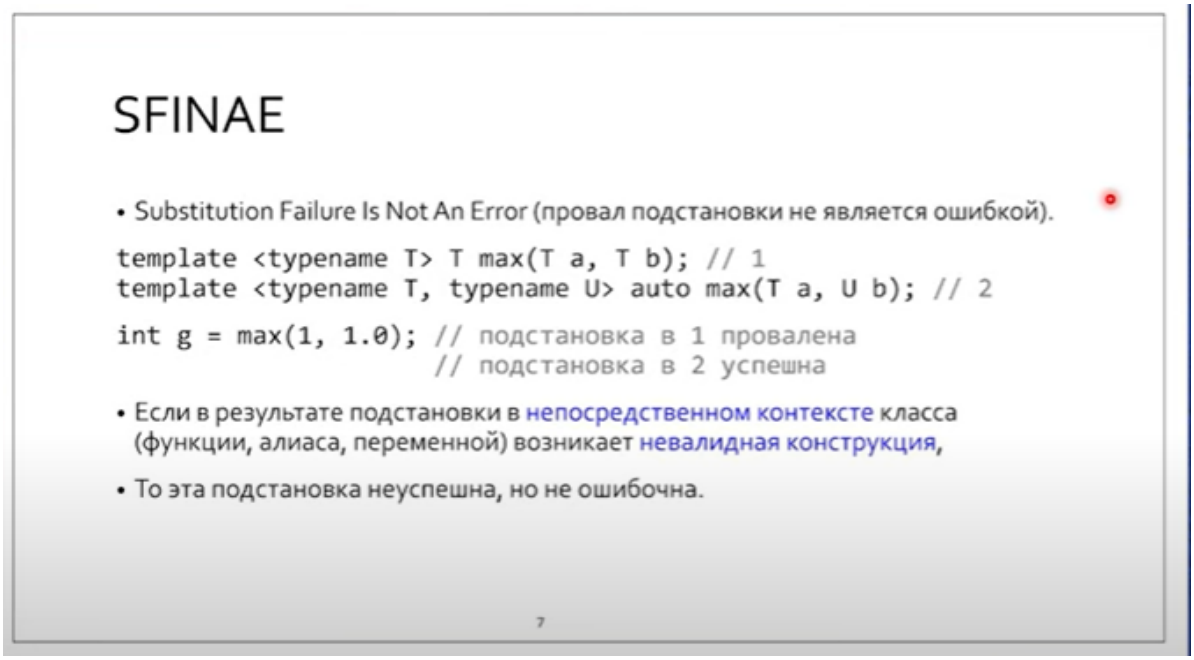
Линкеры исключают неиспользованные функции

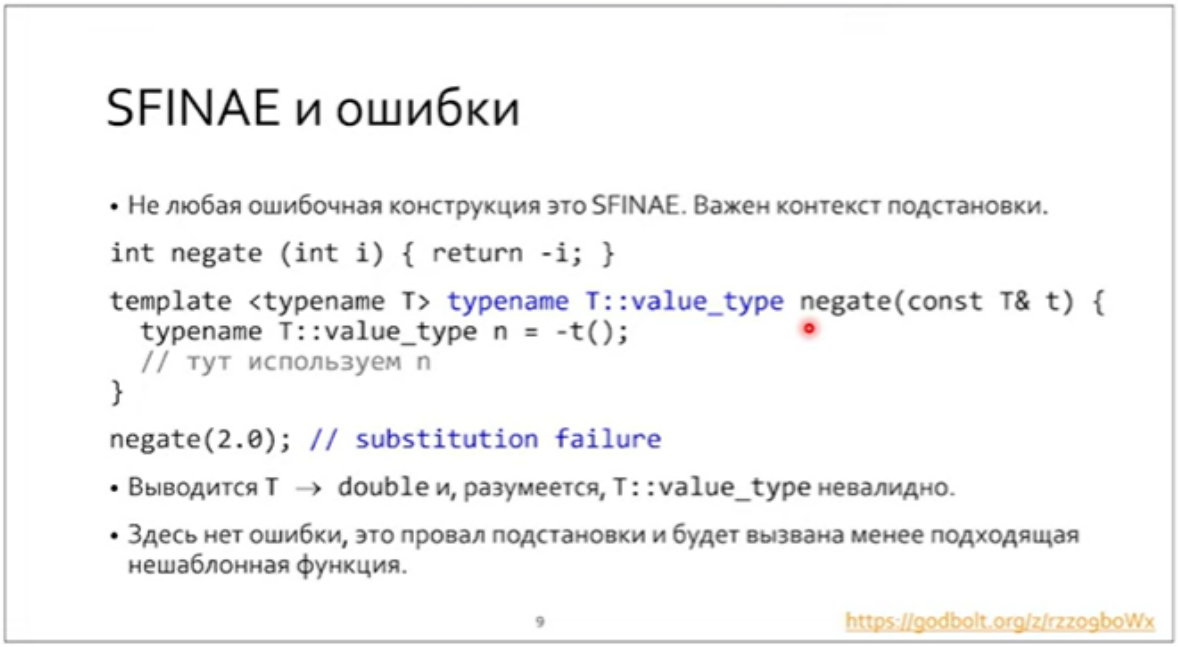
Инстанцирование - порождение специализации

Как может провалиться инстанцирование?

Через вывод типов

Через синтакисчески неккоректный контекст

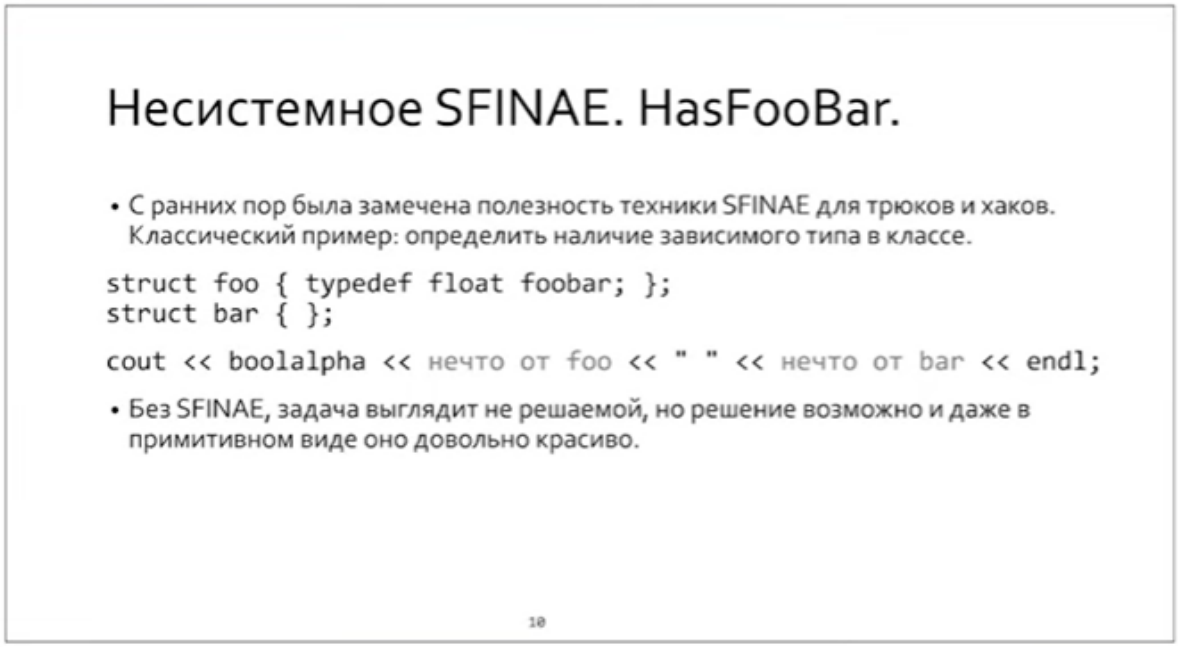




typename для дизамбигуации

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

Пример 2

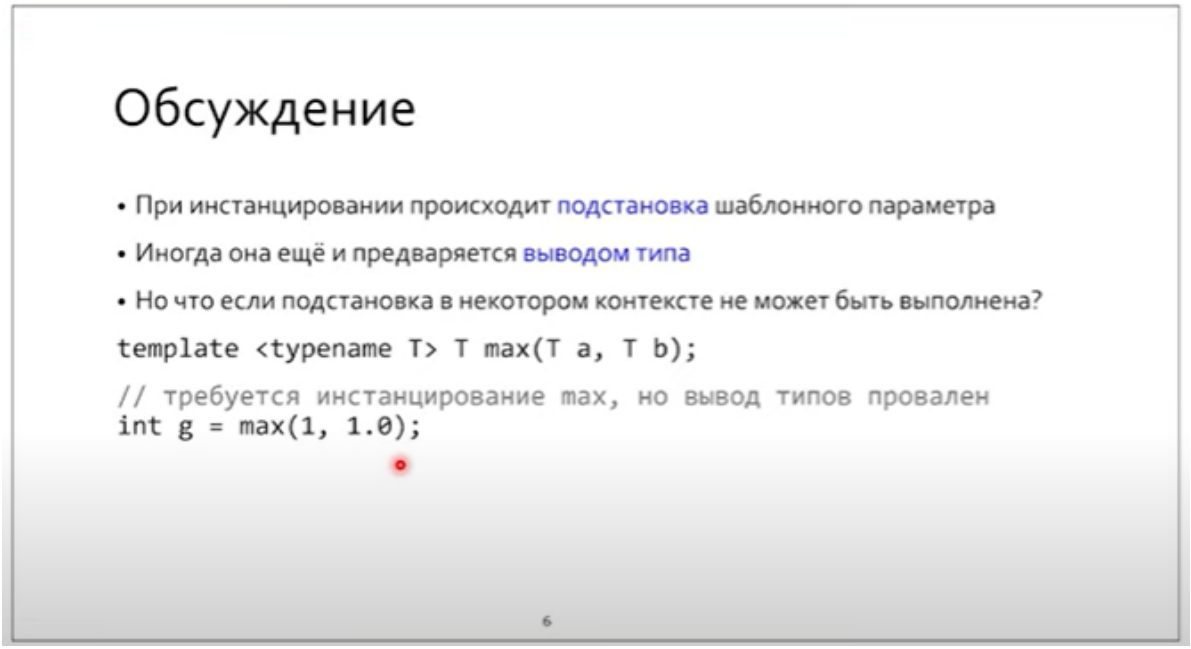



```
template <typename T> struct has_typedef_foobar {
  using yes = char[1];
  using no = char[2];
  template <typename C> static yes& test(typename C::foobar*);
  template <typename> static no& test(...);
  enum { value = (sizeof(test<T>(0)) == sizeof(yes)) };
};

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);
```

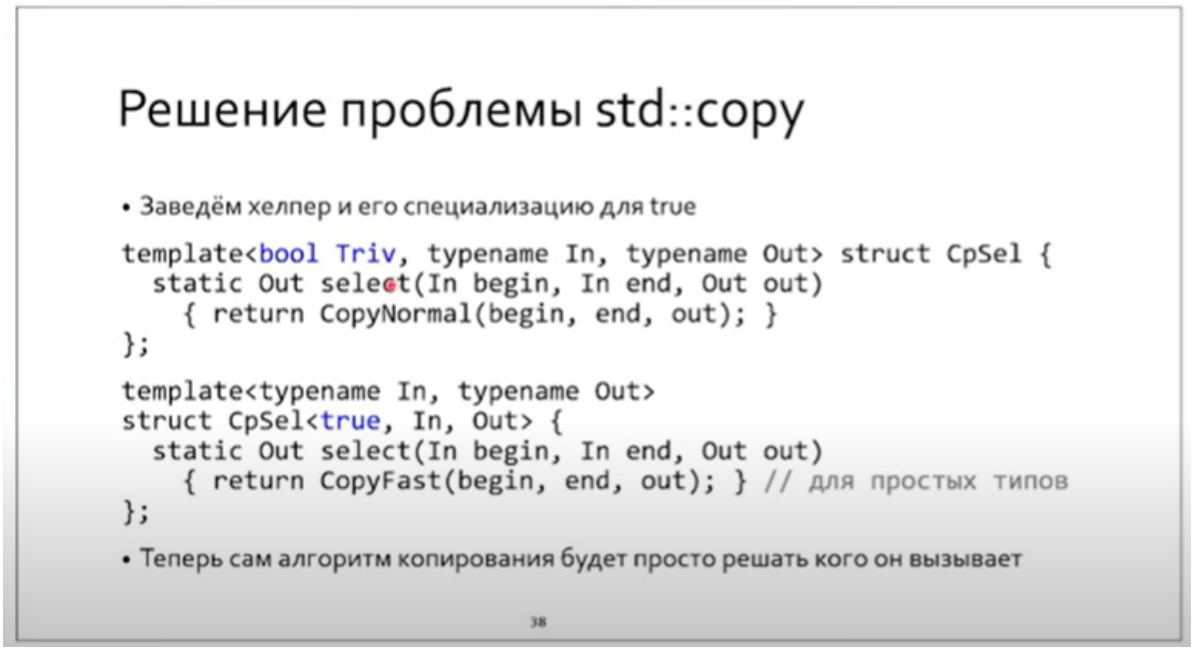
```
#include <concepts>
#include "gtest/gtest.h"

template <typename T> struct has_typedef_foobar {
  using yes = char[1];// для гарантии отработки sizeof во время компиляции
  using no = char[2];
    // две перегрузки (тип C для запуска SFINAE в теле структуры, T просто подставляется тут уже)
  template <typename C> static yes& test(typename C::foobar*);
    // здесь шаблон, потому что мы в енуме задан шаблонный параметр
    // 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 (...)
  template <typename> static no& test(...); // va_args
    // do not require def outside class
  enum { value = (sizeof(test<T>(0)) == sizeof(yes)) };
};

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



Draft



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

```
using in_type = typename std::iterator_traits<In>::value_type;
using out_type = typename std::iterator_traits<Out>::value_type;
enum {
```

```
memcpy(&*out, &*begin, sz);
```

Перегруженный опреатор *