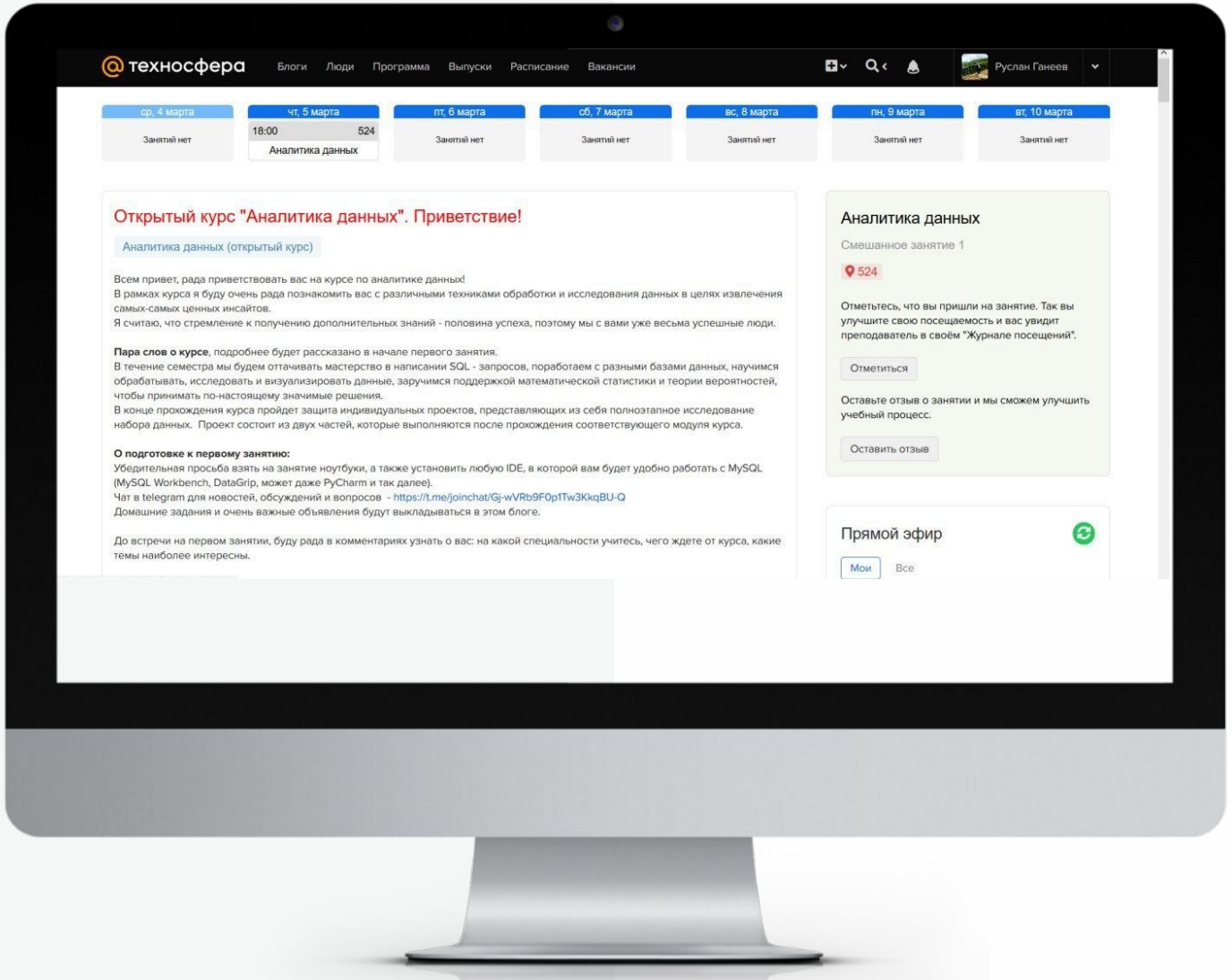
ТЕХНОСФЕРА

Шаблоны

Антон Кухтичев

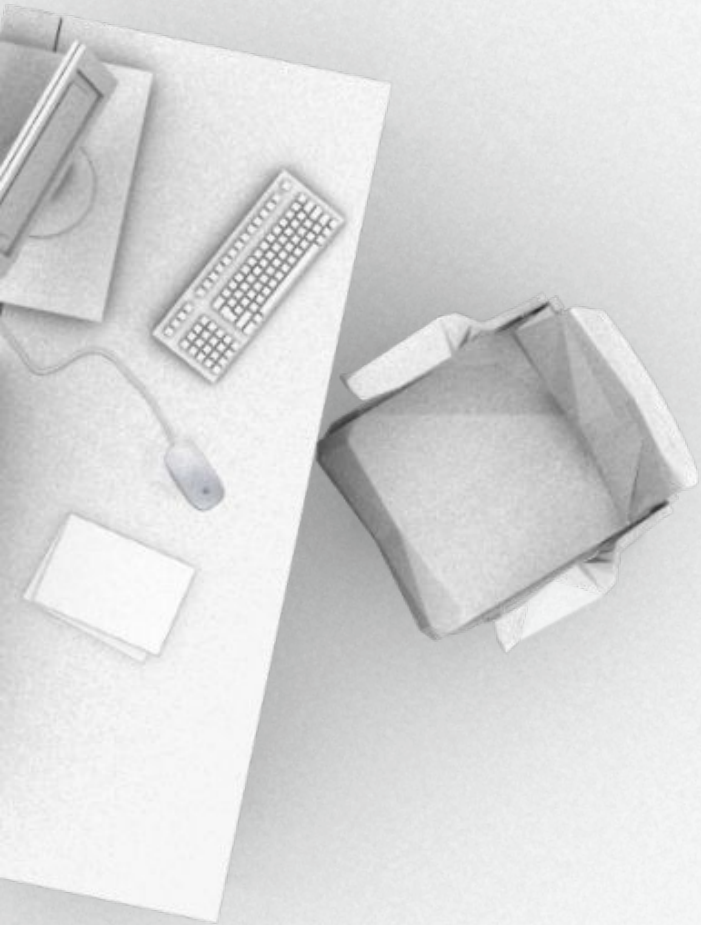


Не забудьте отметиться на портале!!!

Иначе всё плохо будет.

Содержание занятия

1. Введение в шаблоны
2. Инстанцирование шаблона
3. Специализация шаблона
4. Новые штучки в C++11/14
5. SFINAE



Введение в шаблоны

Мотивация (1)

```
class Matrix
{
private:
    double *data;
public:
    Matrix operator+(const Matrix& lhs);
    Matrix operator-(const Matrix& lhs);
    ...
};
```

Мотивация (2)

```
class MatrixDouble
{
private:
    double *data;
    ...
};

class MatrixInt
{
private:
    int *data;
    ...
};
```

Мотивация (3)

```
template <class T>
class Matrix
{
    T* data_;
};
Matrix<double> m;
Matrix<int> m;
```

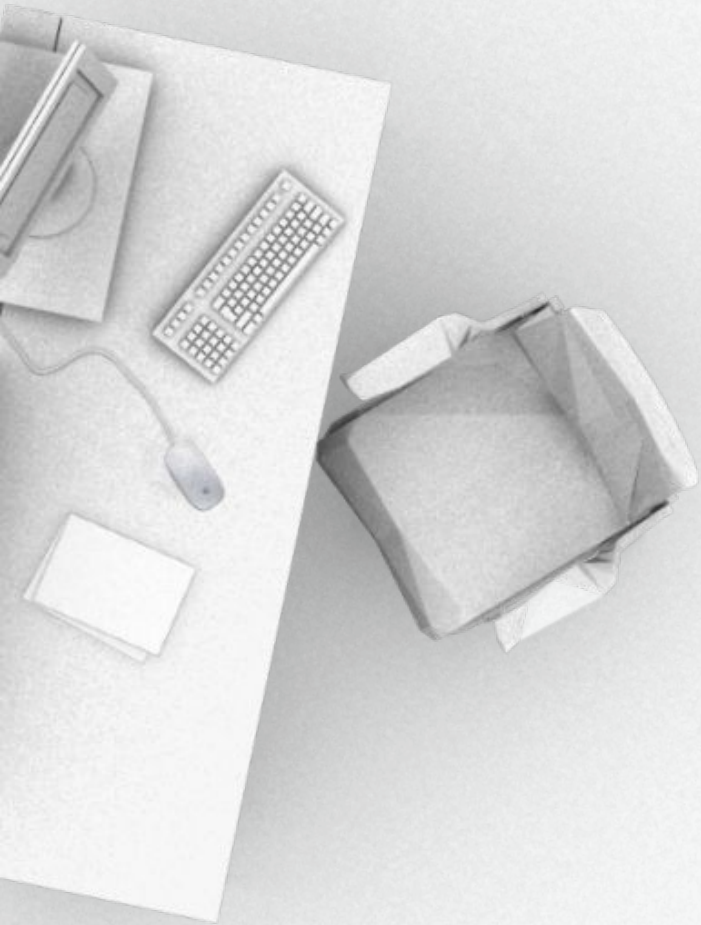
Шаблоны функций

```
template<class T>
void printLine(const T& value)
{
    std::cout << value << std::endl;
}

printLine<int>(5);
printLine(5);
```

Компилятор может самостоятельно вывести тип шаблона в зависимости от аргументов вызова. Но для классов нет!

class или typename

```
template<class T>
void printLine(const T& value)
{
    std::cout << value << std::endl;
}

template<typename T>
void printLine(const T& value)
{
    std::cout << value << std::endl;
}
```



Инстанцирование шаблона

Инстанцирование шаблона

Инстанцирование шаблона - это генерация кода функции или класса по шаблону для конкретных параметров.

```
template <class T>
bool lessThan7(T value) { return value < 7; }

lessThan7(5); //  Инстанцирование
// bool lessThan7(int value) { return value < 7; }

lessThan7(5.0); //  Инстанцирование
// bool lessThan7(double value)  { return value < 7; }
```

Явное указание типа

```
lessThan7<double>(5); //  Инстанцирование
// bool lessThan7(double value) { return value < 7; }
```

Константы как аргументы шаблона

```
template <class T, size_t Size>
class Array
{
    T data_[Size];
};

Array<int, 5> a;
```

Ограничения на параметры шаблона не являющиеся типами

Так можно:

```
template <int N>
int foo()
{
    return N * 2;
}
```

А `double` нельзя (`float` тоже нельзя):

```
template <double N> // Ошибка
void foo()
{
}
```

Параметры шаблона должны быть известны на этапе компиляции

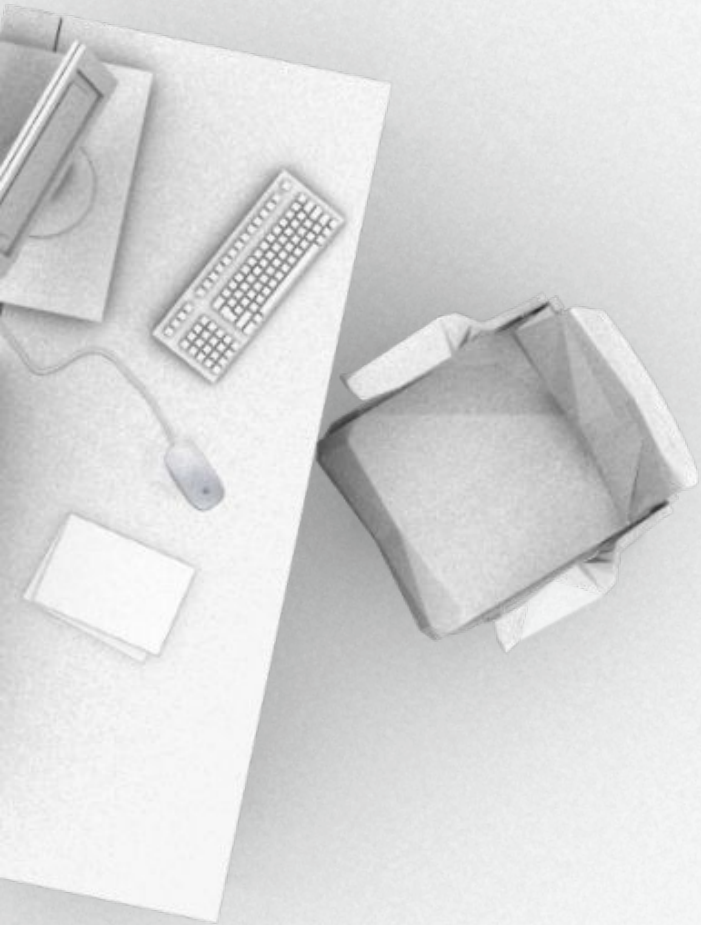
```
template <int N>
void foo() { }

int x = 3;
foo<x>(); // Ошибка
```

Константы на литералы можно:

```
const int x = 3;
foo<x>(); // Ok
```

Параметры шаблона по умолчанию

```
template <class X, class Y = int>
void foo()
{
}
foo<char>();

template <class T, class ContainerT = std::vector<T>>
class Queue
{
    ContainerT data_;
};
Queue<int> queue
```



Специализация шаблона

Специализация шаблона

```
template <class T>
class Vector
{
    ...
}

template <>
class Vector<bool>
{
    ...
};
```

Псевдонимы типов

Старый способ:

```
typedef int Seconds;
typedef Queue<int> IntegerQueue;

Seconds i = 5;
IntegerQueue j;
```

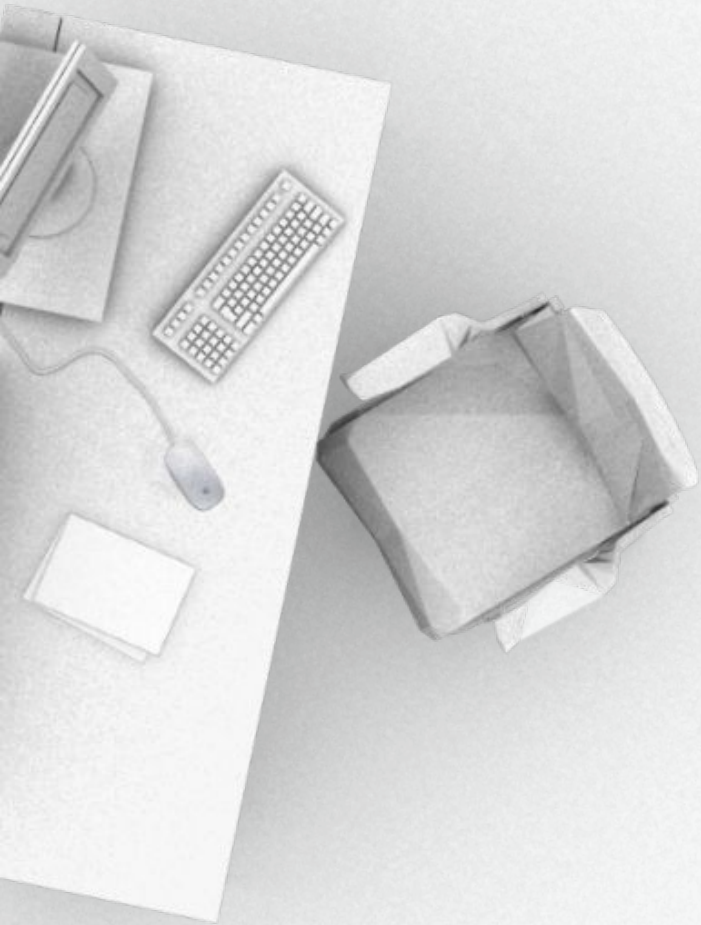
Новый (рекомендуемый) способ:

```
using Seconds = int;
using IntegerQueue = Queue<int>;

Seconds i = 5;
IntegerQueue j;
```

Псевдонимы типов для шаблонов (alias template)

```
template <class T>
using MyQueue = Queue<T, std::deque<T>>;

MyQueue<int> y;

template<class T>
using MyVector = std::vector<T, MyCustomAllocator<T>>;
```

С typedef такого добиться нельзя.



Новые штучки в C++11/14

auto, decltype, ranged-base for, constexpr

Новый синтаксис функций

```
auto foo() -> void
{
}
```

auto

Позволяет статически определить тип по типу выражения.

```
auto i = 5;
auto j = foo();
```


1. Скотт Мейерс. Эффективный и современный C++. Пункт 1.2. Вывод типа auto.

range-based for и auto

```
for (auto i : { 1, 2, 3 })
    std::cout << i;


for (auto& i : data)
    i.foo();
```

decltype (1)

Позволяет статически определить тип по типу выражения.

```
int foo() { return 0; }

decltype(foo()) x = 5;
// decltype(foo()) -> int
// int x = 5;

void foo( decltype(bar()) i )
{
}
```


1. Скотт Мейерс. Эффективный и современный C++. Пункт 1.3. Знакомство с decltype.

decltype (2)

```
template<typename Container, typename Index>
auto authAndAccess(Container &c, Index i) -> decltype(c[i])
{
    authenticateUser();
    return c[i];
}
```

constexpr

```
constexpr int sum(int a, int b)
{
    return a + b;
}

void func()
{
    // значение переменной будет посчитано на этапе компиляции.
    constexpr int c = sum(5, 12);
}
```


1. Скотт Мейерс. Эффективный и современный C++. Пункт 3.9. Используйте, где это возможно constexpr.

Определение типа аргументов шаблона функций (1)

```
template <typename T>
T min(T x, T y)
{
    return x < y ? x : y;
}

min(1, 2); // ok
min(0.5, 2); // error
min<double>(0.5, 2); // ok
```

Определение типа аргументов шаблона функций (2)

```
template <typename X, typename Y>
X min(X x, Y y)
{
    return x < y ? x : y;
}

min(1.5, 2); // ok
min(1, 0.5); // ok?
```

Определение типа аргументов шаблона функций (3)

```
template <typename X, typename Y>
auto min(X x, Y y) -> decltype(x + y)
{
  return x < y ? x : y;
}

min(1.5, 2); // ok
min(1, 0.5); // ok
```

typename (1)

```
template <class Container>
class Parser
{
    Container::const_iterator *x; // Ошибка
};
```

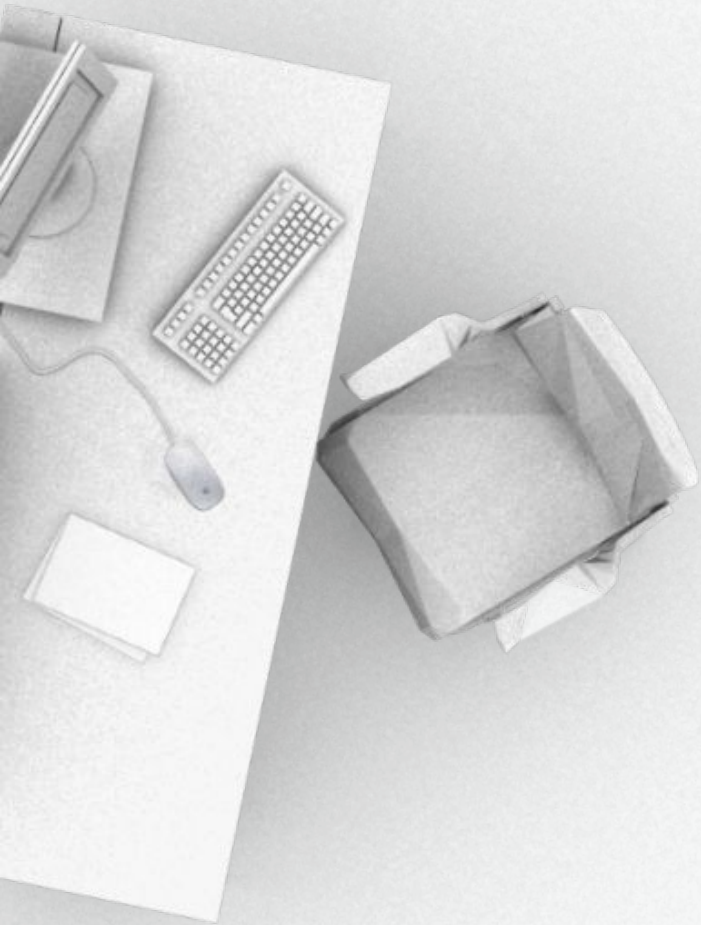
Если компилятор встречая идентификатор в шаблоне, может его трактовать как тип или что-то иное (например, как статическую переменную), то он выбирает иное.

typename (2)

```
template <class Container>
class Parser
{
    typename Container::const_iterator *x; // ок
};
```

Общее правило просто: всякий раз, когда вы обращаетесь к вложенному зависимому имени в шаблоне, вы должны предварить его словом typename.

1. Скотт Мейерс. Эффективное использование C++. Правило 42: Усвойте оба значения ключевого слова typename.



SFINAE

Substitution Failure Is Not An Error
Неудавшаяся подстановка — не ошибка

SFINAE (1)

При определении перегрузок функции ошибочные инстанциации шаблонов не вызывают ошибку компиляции, а отбрасываются из списка кандидатов на наиболее подходящую перегрузку.

Неудачное инстанцирование шаблона - это не ошибка.

Ошибка будет в трёх случаях:

1. Не нашлось ни одной подходящей перегрузки.
2. Нашлось несколько таких перегрузок, и компилятор не может решить, какую взять.
3. Перегрузка нашлась, она оказалась шаблонной, и при инстанцировании шаблона случилась ошибка.

SFINAE (1)

```
(1) void f(int, std::vector<int>);
(2) void f(int, int);
(3) void f(double, double);
(4) void f(int, int, char, std::string, std::vector<int>);
(5) void f(std::string);
(6) void f(...);
(7) template<typename T>
    void f(T, T);
(8) template<typename T>
    void f(T, typename T::iterator);

f(1, 2);
```

SFINAE (2)

Например, позволяет на этапе компиляции выбрать нужную функцию:

```
template<typename T>
void clear(T& t,
    typename std::enable_if<std::is_pod<T>::value>::type* = nullptr)
{
    std::memset(&t, 0, sizeof(t));
}

// Для не-POD типов
template<typename T>
void clear(T& t,
    typename std::enable_if<!std::is_pod<T>::value>::type* = nullptr)
{
    t = T{};
}
```

is_pod

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

template <>
struct is_pod<int>
{
    static constexpr bool value = true;
};
```

enable_if

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

// Частичная специализация для true
template<typename T>
struct enable_if<true, T>
{
    using type = T;
};

enable_if<false, int>::type // Ошибка, нет type
enable_if<true, int>::type // Ок, type == int
```

Code time!

Хотим получить на этапе компиляции информацию о типе, например, проверим есть ли у класса некий метод.

type_traits

В стандартной библиотеки функции определения свойств типов is_* находятся в заголовочном файле type_traits

Примеры:

```
is_integral         // Является ли тип целочисленным
is_floating_point // Является ли тип типом с плавающей точкой
is_array            // Является ли тип типом массива
is_const            // Содержит ли тип в себе квалификатор const
is_pod              // Является ли тип POD-типом
has_virtual_destructor // Имеет ли виртуальный деструктор

// И так далее
```

Шаблоны свойств (traits)

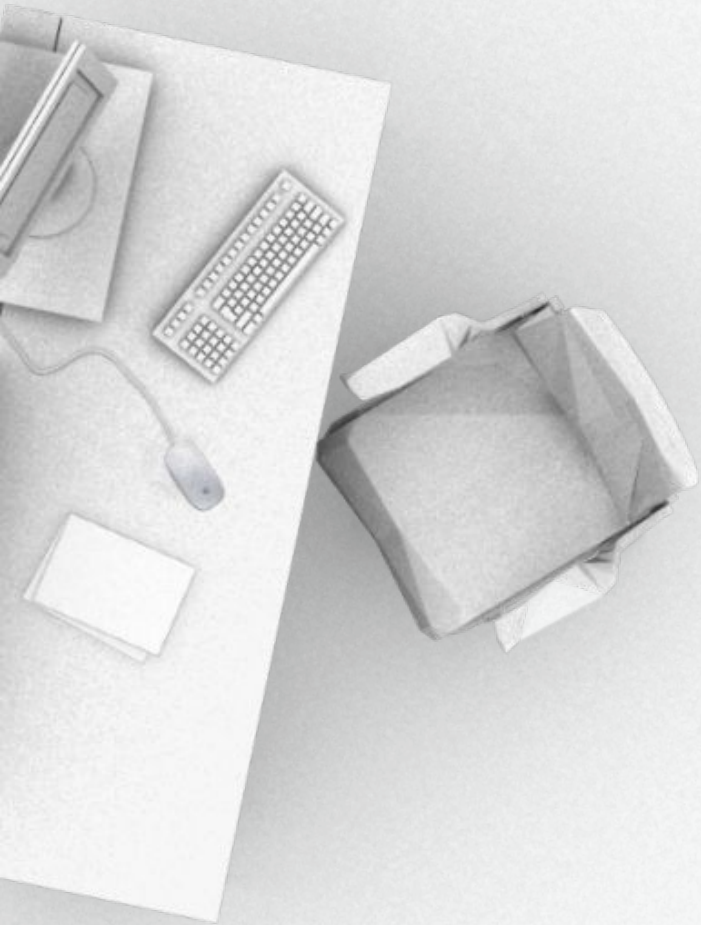
```
template <typename T>
struct NumericTraits
{
};

template <> // Специализация
struct NumericTraits<char>
{
    static constexpr int64_t min = -128;
    static constexpr int64_t max = 127;
};
```

Шаблоны свойств (traits)

```
template <typename T>
class Calculator
{
    T getNumber(const std::string& text)
    {
        const int64_t value = toNumber(text);
        if (value < NumericTraits<T>::min
            || value > NumericTraits<T>::max)
        {
            // range error
        }
    }
};
```

Шаблоны с произвольным количеством аргументов (variadic templates)

```
print(1, "abc", 2.5);

template <class T>
void print(T&& val)
{
    std::cout << val << '\n';
}

template <class T, class... Args>
void print(T&& val, Args&&... args)
{
    std::cout << val << '\n';
    print(std::forward<Args>(args)...);
}
```



Домашнее задание

Домашнее задание (1)

Простой сериализатор/десериализатор поддерживающий два типа: `uint64_t` и `bool`.

Сериализовать в текстовый вид с разделением пробелом, `bool` сериализуется как `true` и `false`.

```
struct Data
{
    uint64_t a;
    bool b;
    uint64_t c;
};

Data x { 1, true, 2 };

std::stringstream stream;
```

```
Serializer serializer(stream);
serializer.save(x);

Data y { 0, false, 0 };

Deserializer deserializer(stream);
const Error err = deserializer.load(y);

assert(err == Error::NoError);
assert(x.a == y.a);
assert(x.b == y.b);
assert(x.c == y.c);
```

Домашнее задание (2)

```
struct Data
{
    uint64_t a;
    bool b;
    uint64_t c;

    template <class Serializer>
    Error serialize(Serializer& serializer)
    {
        return serializer(a, b, c);
    }
};
```

Домашнее задание по уроку #6

Домашнее задание №5

?

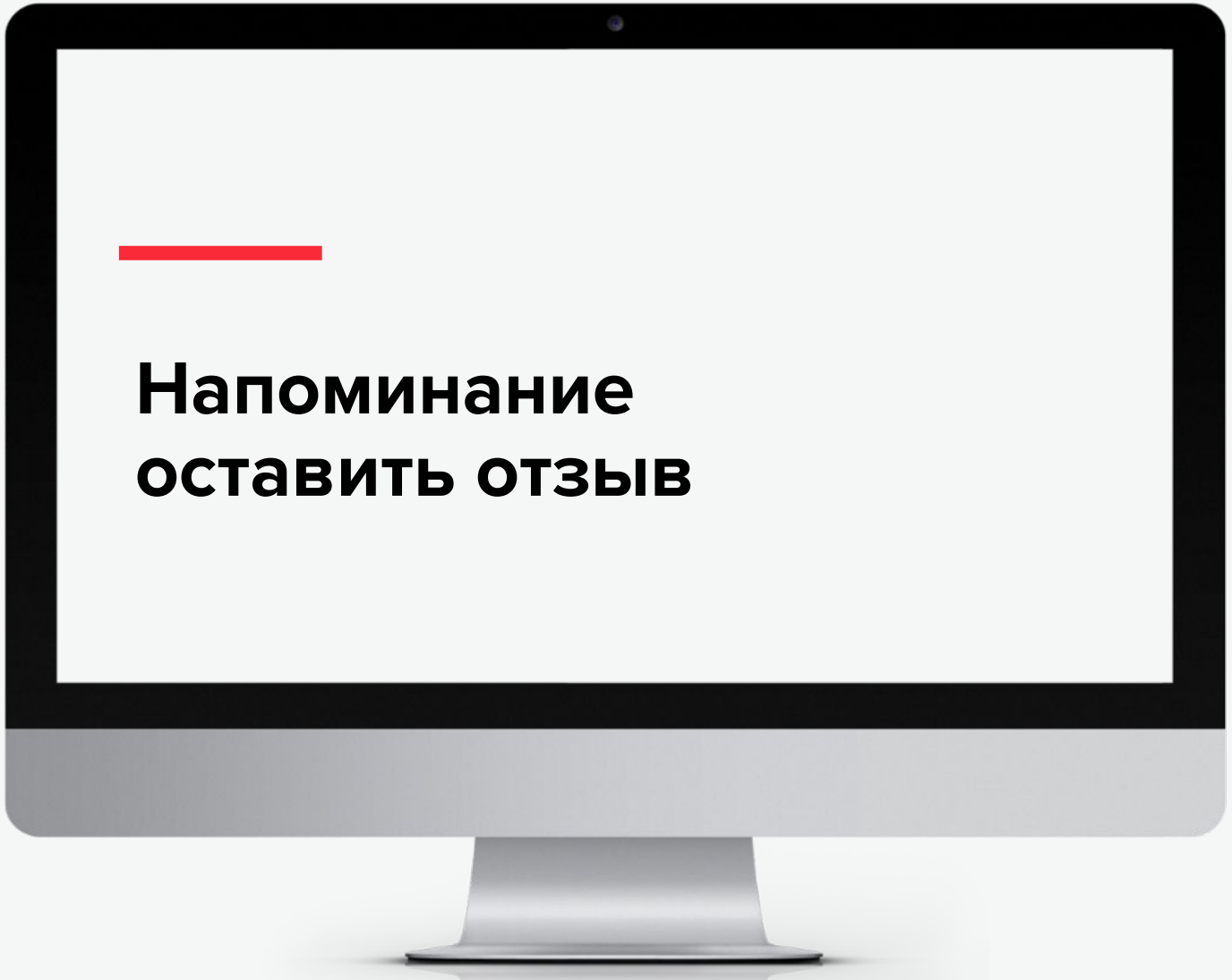
Баллов за задание

12.11.20

Срок сдачи

Полезная литература в помощь

- Скотт Мейерс “Эффективный и современный C++”
- Скотт Мейерс “Эффективное использование C++”
- Бьерн Страуструп “Языка программирования C++”


Напоминание
оставить отзыв

СПАСИБО
ЗА ВНИМАНИЕ