



Функции в C++

Антон Кухтичев





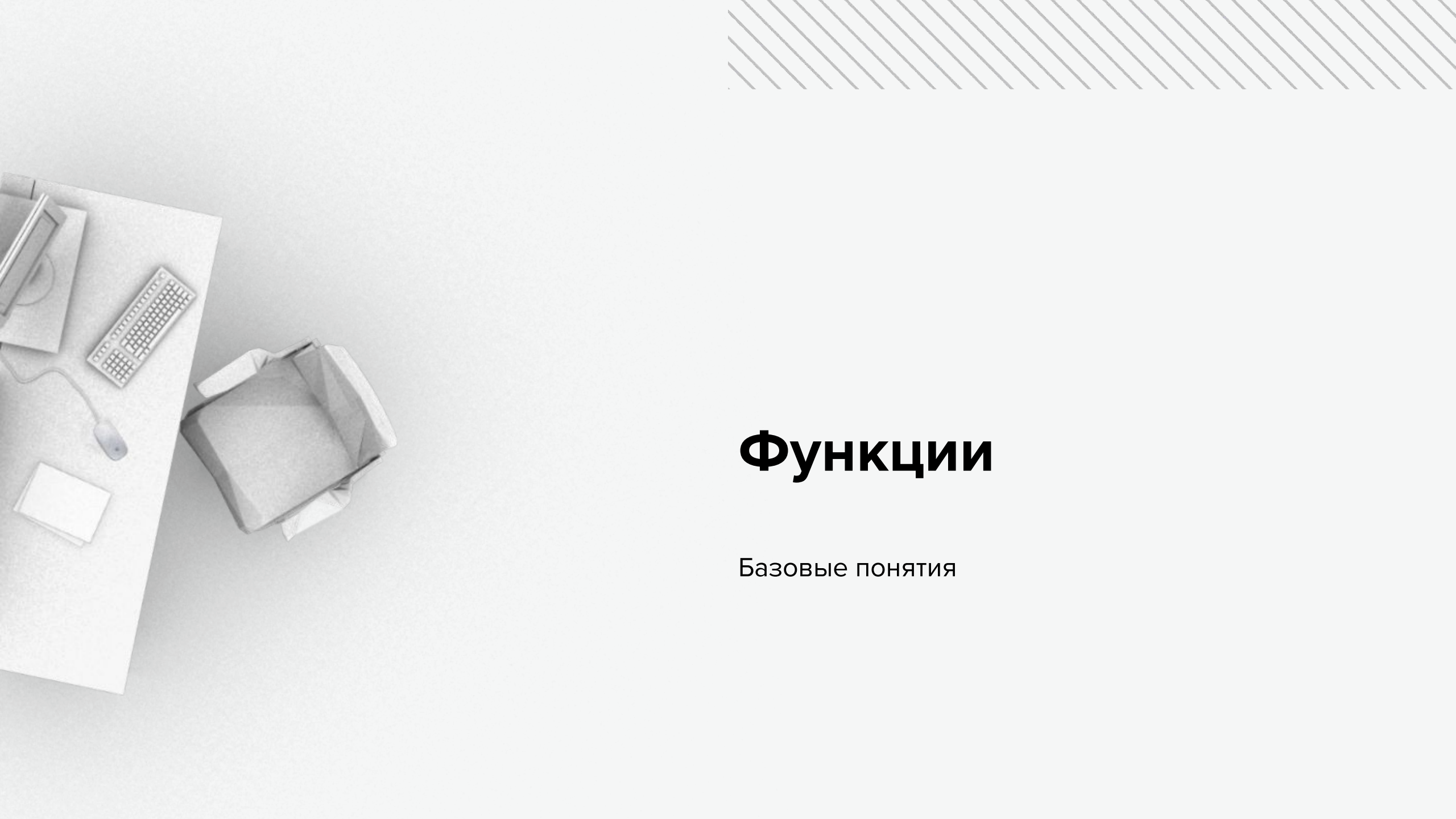
Не забудьте
отметиться на
портале!!!

Иначе всё плохо будет.



Содержание занятия

1. Функции
2. Соглашения о вызовах
3. Встраиваемые функции (inline)
4. Ссылки
5. λ-выражения
6. Функции высшего порядка
7. `std::function`

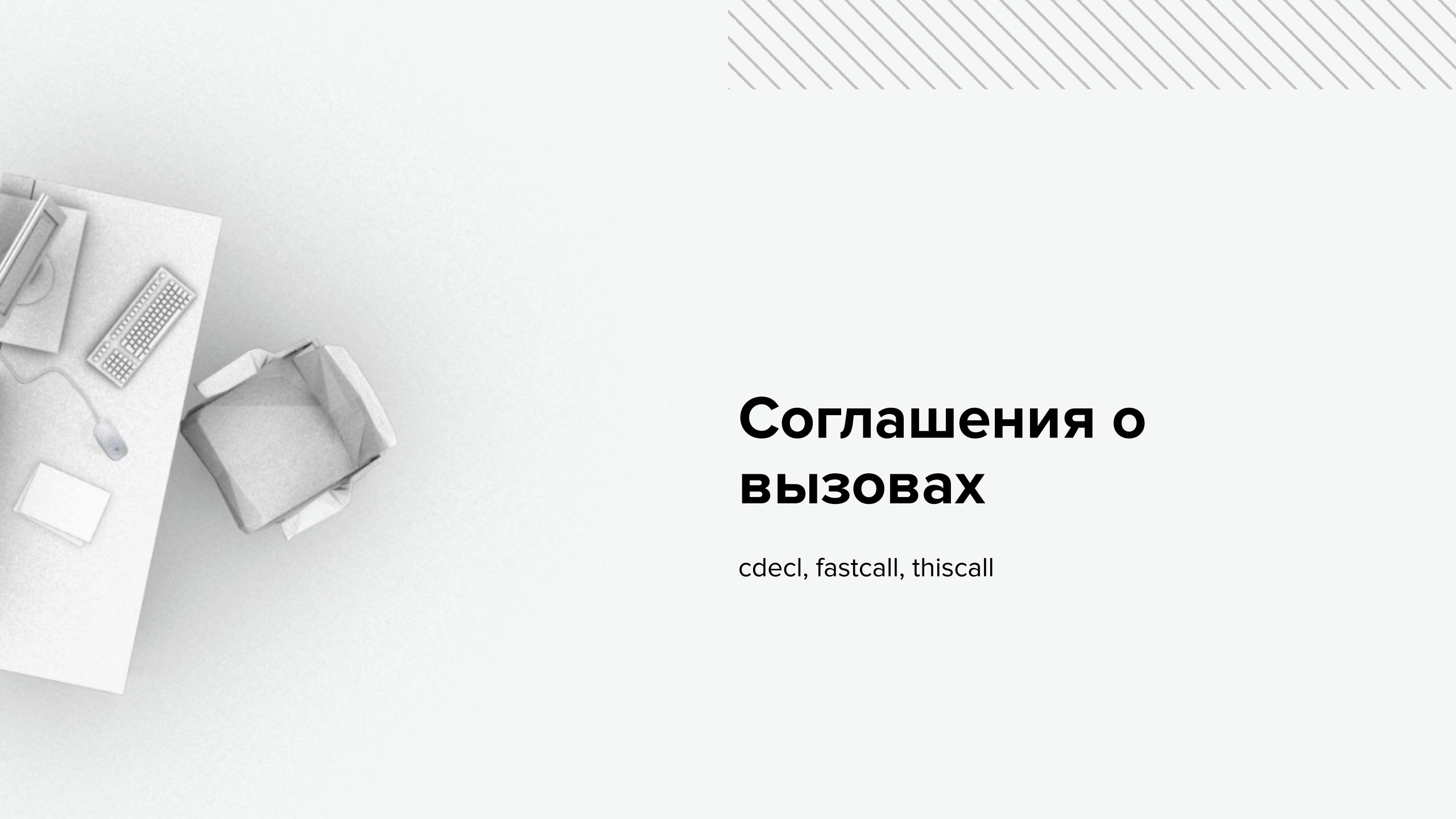


Функции

Базовые понятия

Функции

- Это кусок кода, который может выполнить процессор, который находится по определённому адресу;
- Функция может возвращать значение, тип которого в большинстве случаев аналогично типу самой функции;
- Если функция не возвращает никакого значения, то она должна иметь тип `void` (такие функции иногда называют процедурами);
- Функция может принимать параметры (а может и не принимать);
- Функции можно перегружать;



Соглашения о вызовах

cdecl, fastcall, thiscall

cdecl (c-declaration)

Перед вызовом функции вставляется код, называемый прологом (prolog) и выполняющий следующие действия:

- сохранение значений регистров, используемых внутри функции
- запись в стек аргументов функции

После вызова функции вставляется код, называемый эпилогом (epilog) и выполняющий следующие действия:

- восстановление значений регистров, сохранённых кодом пролога
- очистка стека (от локальных переменных функции)




thiscall

Соглашение о вызовах, используемое компиляторами для языка C++ при вызове методов классов.

Отличается от **cdecl**-соглашения только тем, что указатель на объект, для которого вызывается метод (указатель `this`), записывается в регистр `ecx`.



fastcall

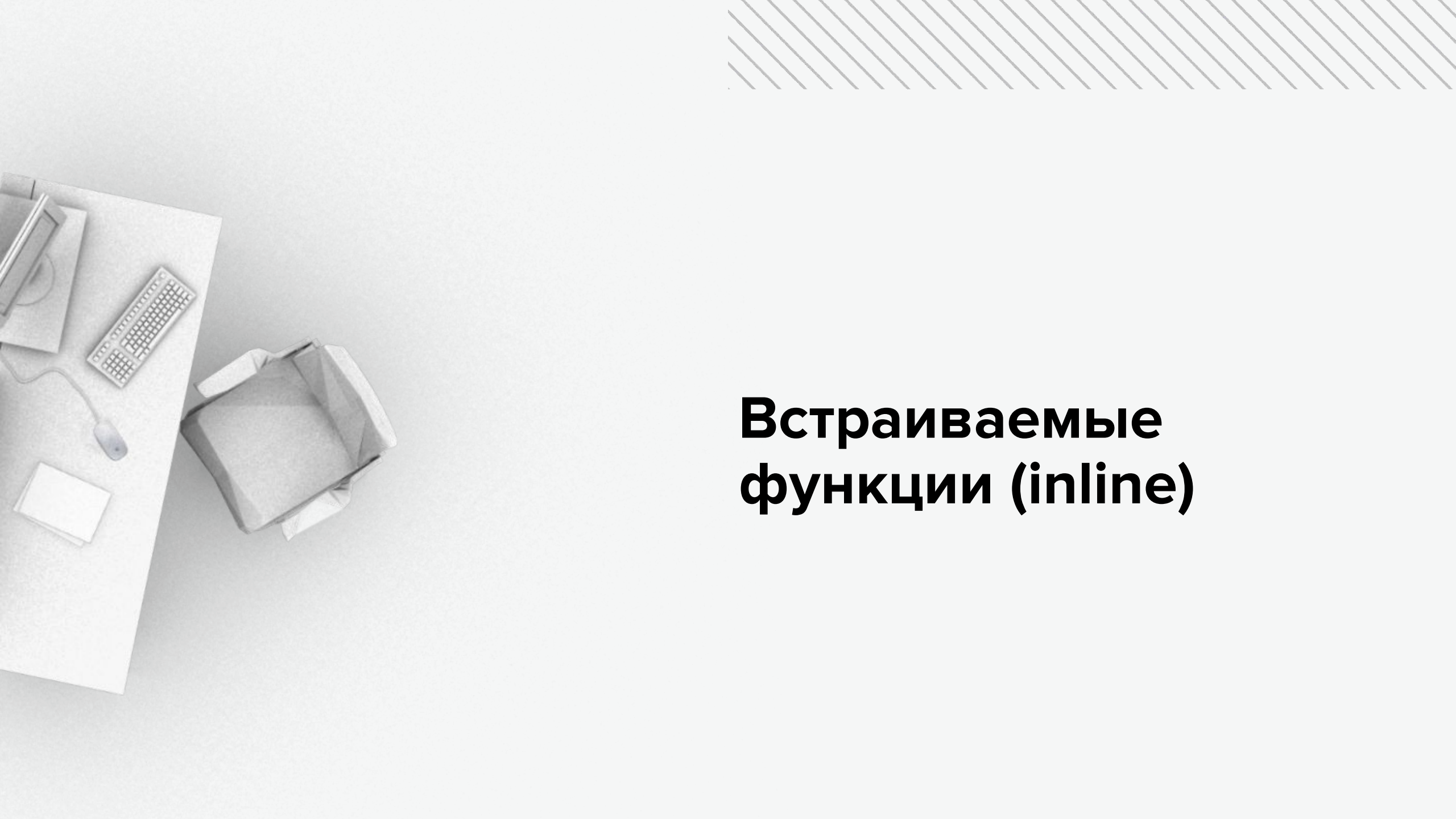


Передача параметров через регистры: если для сохранения всех параметров и промежуточных результатов регистров недостаточно, используется стек (в gcc через регистры ecx и edx передаются первые 2 параметра).



System V AMD64 ABi (Linux, MacOS, FreeBSD)

- 6 регистров (RDI, RSI, RDX, RCX, R8, R9) для передачи integer-like аргументов
- 8 регистров (XMM0-XMM7) для передачи double/float
- если аргументов больше, они передаются через стек
- для возврата integer-like значений используются RAX и RDX (64 бит + 64 бит)



Встраиваемые функции (inline)

inline

- Указывает компилятору, что он должен пытаться каждый раз генерировать в месте вызова код, соответствующий функции;
- Компилятор умный и он может не встроить код.

```
inline void foo()  
{  
}
```

Просим ещё настойчивее

- `__attribute__((always_inline))`
- `__forceinline`

Компилятор пытается встроить функцию вне зависимости от характеристик функции.

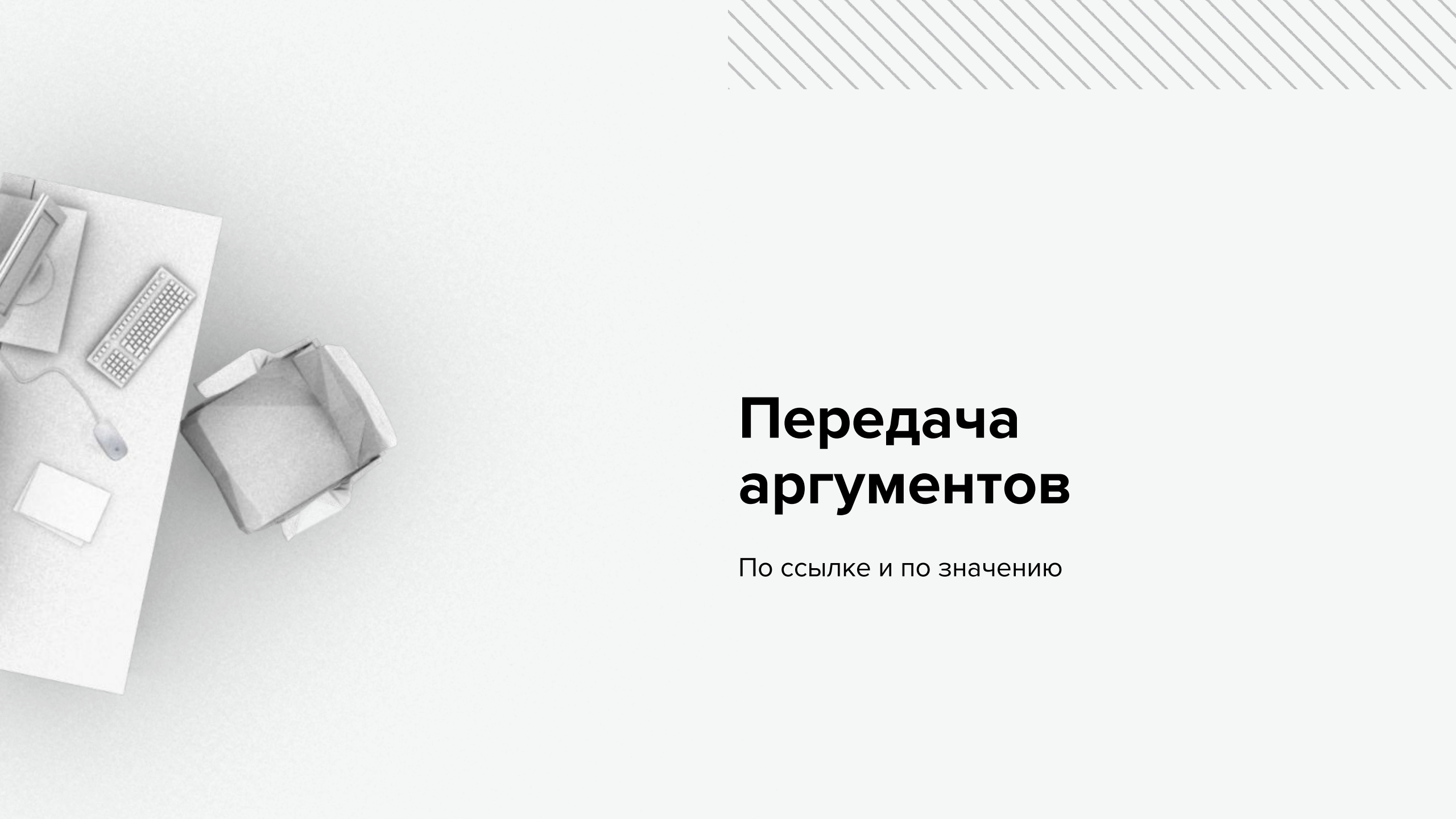
В некоторых случаях компилятор может игнорировать встраивание:

- Рекурсивная функция никогда не встраивается в себя;
- Если в функции используется функция `alloca()`;
- Всё равно гарантий нет!

Примерчик

```
__attribute__((always_inline)) void foo()  
{  
}
```

```
#ifdef __GNUC__  
#define __forceinline __attribute__((always_inline))  
#endif
```



Передача аргументов

По ссылке и по значению

Передача аргументов по значению

- В функции окажется копия объекта, ее изменение не отразится на оригинальном объекте;
- Копировать большие объекты может оказаться накладно;

```
void foo(int a) { a += 1; }
```

```
int a = 1;
```

```
foo(a); // всё ещё 1.
```


Передача аргументов по ссылке

- Можно передавать аргументы по ссылке
 - Копирования не происходит, все изменения объекта внутри функции отражаются на объекте;
 - Следует использовать, если надо изменить объект внутри функции.

```
void foo(int &a);
```

Передача аргументов по константной ссылке

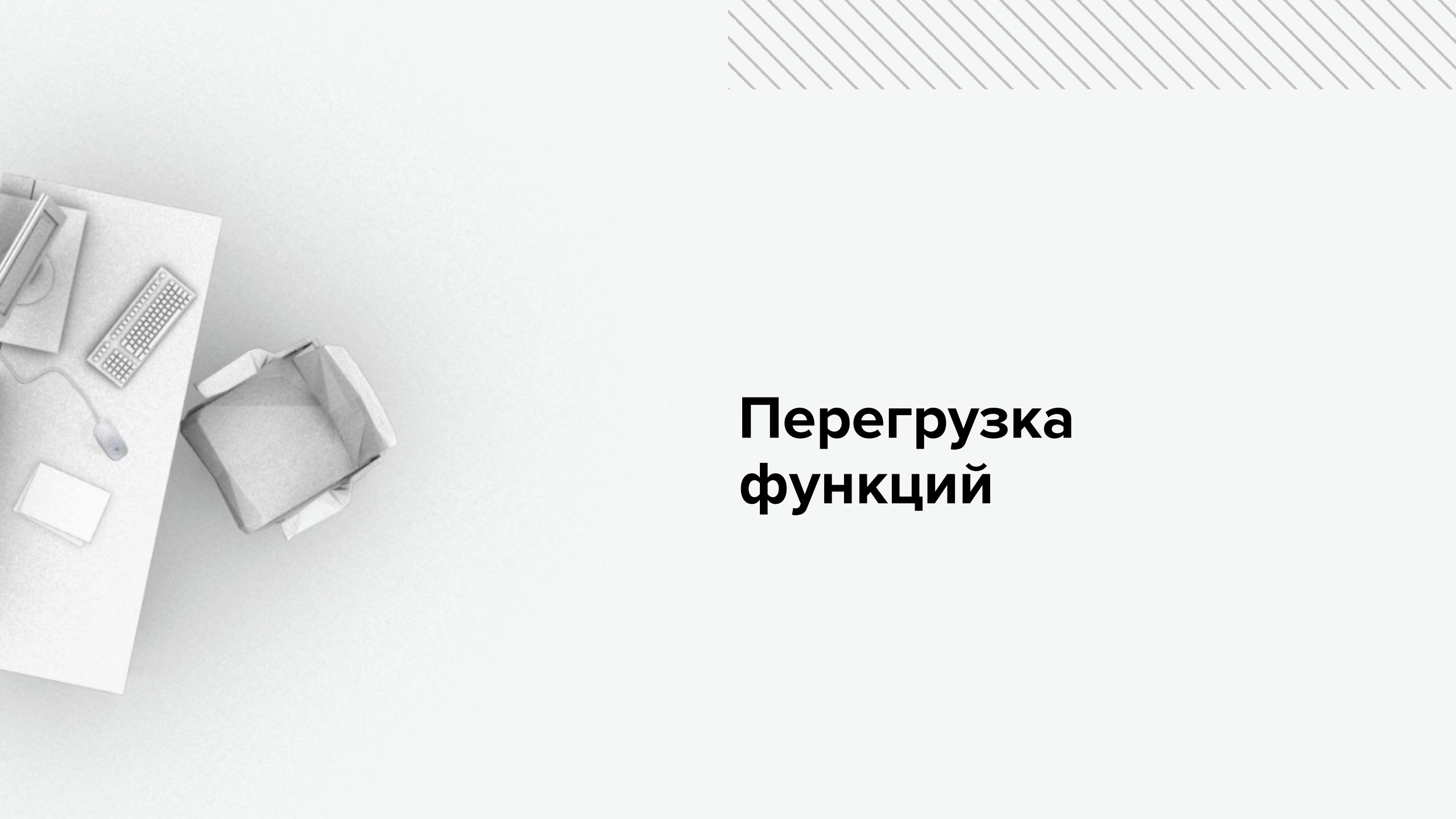
- Копирования не происходит, при попытке изменения объекта будет ошибка
- Большие объекты выгоднее передавать по ссылке, маленькие - наоборот

```
void foo(const int &a);
```

Передача аргументов по указателю

- Копирования не происходит;
- Если указатель на константный объект, то при попытке изменения объекта будет ошибка;
- Есть дополнительный уровень косвенности, возможно придется что-то подгрузить в кеш из дальнего участка памяти;
- Реализуется optional-концепция.

```
void foo(int *a);
```



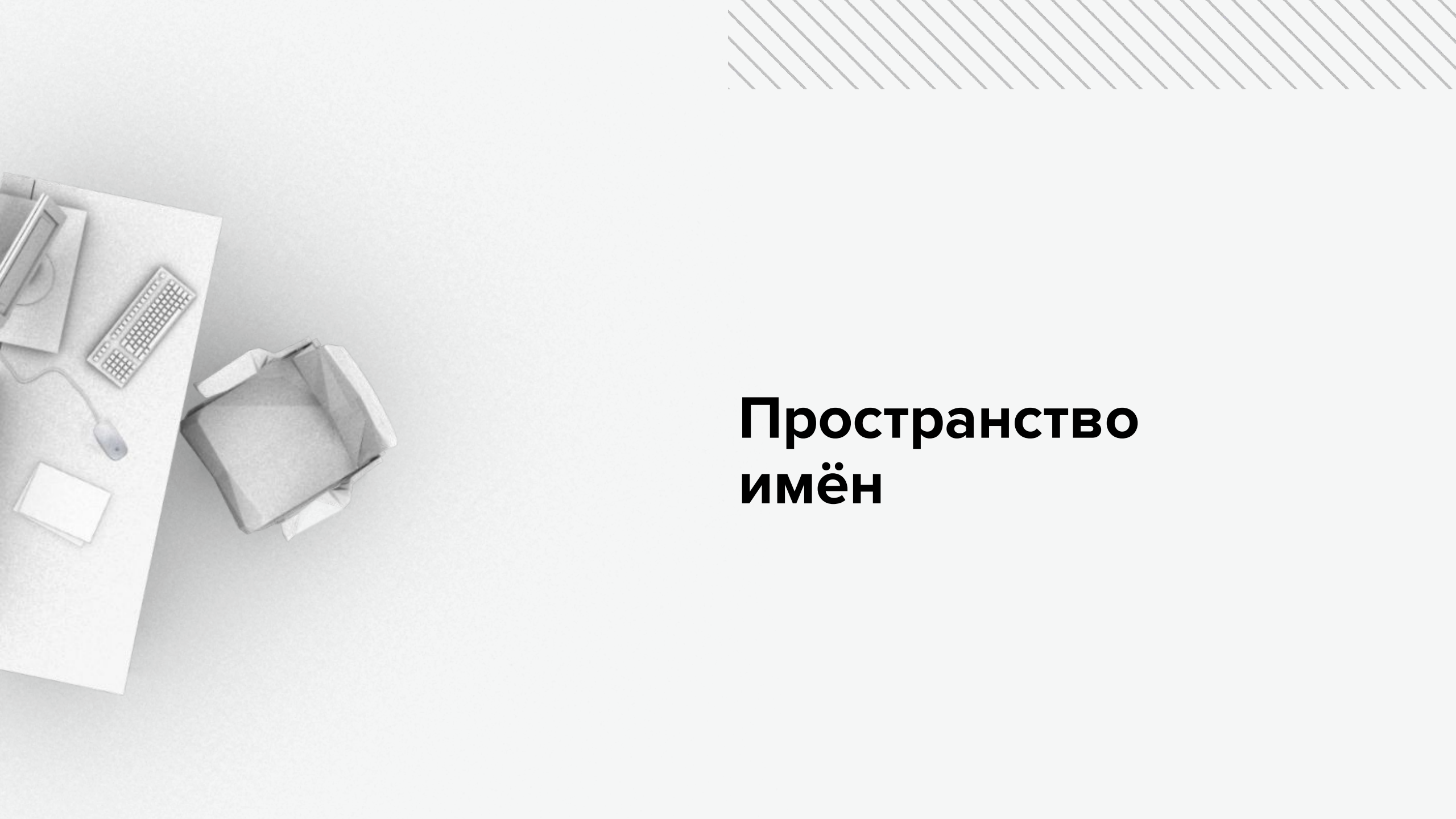
Перегрузка функций

Перегрузка функций

- Использование одного имени для операции, выполняемой с различными типами, называется **перегрузкой**;
- Процесс поиска подходящей функции из множества перегруженных заключается в нахождении наилучшего соответствия типов формальных и фактических аргументов;
- Функции, объявленные в различных областях видимости (не пространствах имён), не являются перегруженными;
- Объявление небольшого количества перегруженных вариантов функции может привести к неоднозначности;
- Перегруженная функция - декорированная функция;

Перегрузка функций

- Точное соответствие типов; то есть полное соответствие или соответствие, достигаемое тривиальными преобразованиями типов;
- Соответствие, достигаемое “продвижением” интегральных типов (например `bool` в `int`, `char` в `int`, `short` в `int`, `float` в `double`)
- Соответствие, достигаемое путём стандартных преобразований (например, `int` в `double`, `double` в `int`, `int` в `unsigned int`)
- Соответствие, достигаемое при помощи преобразований, определяемых пользователем
- Соответствие за счёт многоточий (...) в объявлении функции



Пространство имён

Пространства имён

- Проблема

```
// math.h
```

```
double cos(double x);
```

```
// ваш код
```

```
double cos(double x);
```

Пространства имён

- Решение в стиле C

// ваш код

```
double fastCos(double x);
```

Пространства имён

- Решение в стиле C++


```
namespace fast
{
    double cos(double x);
}
```

```
fast::cos(x);
```

```
cos(x); // ВЫЗОВ ИЗ math.h
```



Пространства имён

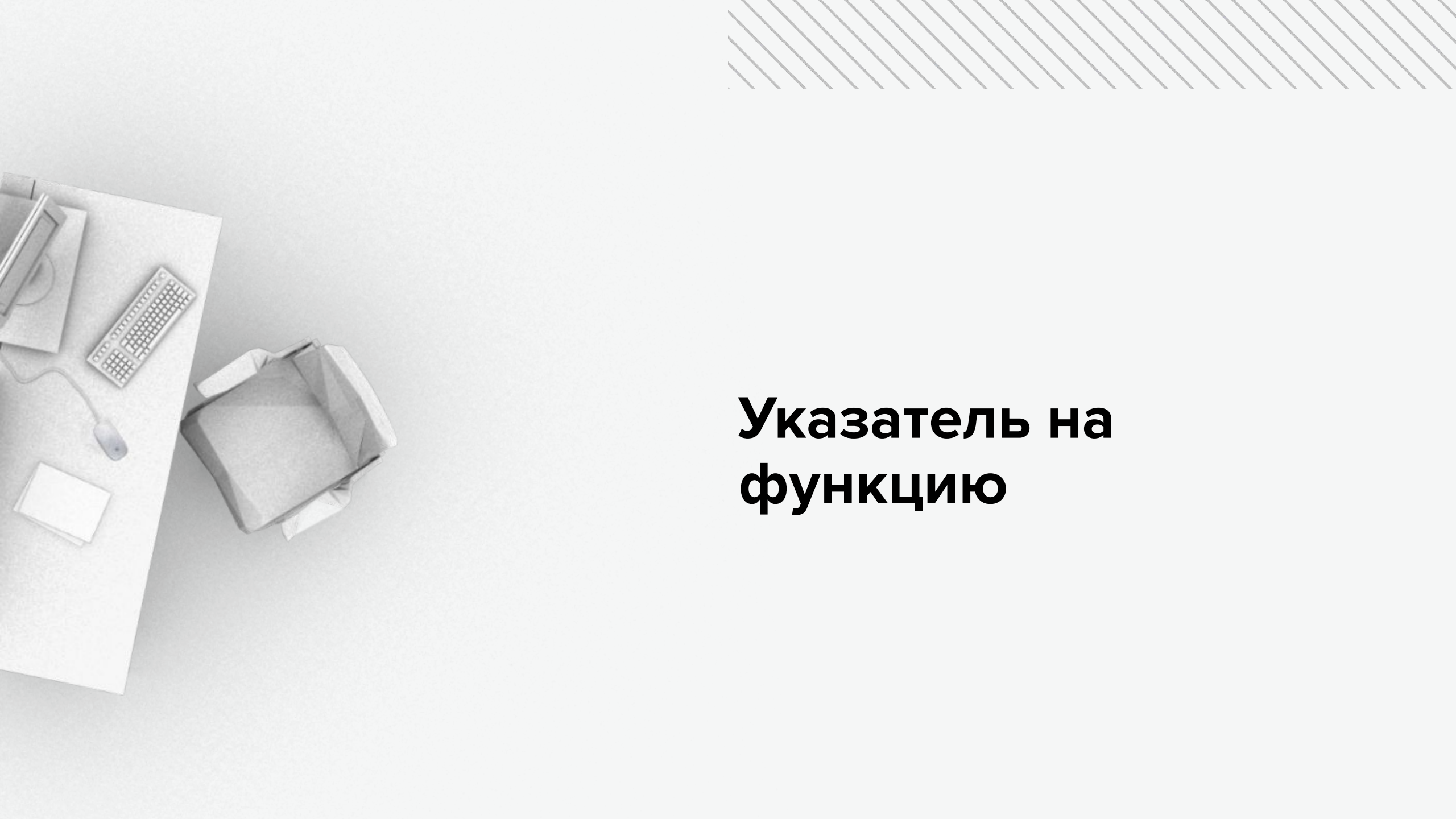
- 
1. Проверка в текущем namespace;
 2. Если имени нет и текущий namespace глобальный - ошибка;
 3. Рекурсивно поиск в родительском namespace.

Пространства имён

```
void foo() {} // ::foo

namespace A
{
    void foo() {} // A::foo

    namespace B
    {
        void bar() // A::B::foo
        {
            foo(); // A::foo
            ::foo(); // foo()
        }
    }
}
```



**Указатель на
функцию**

Указатель на функцию

```
void foo(int x) { }
```

```
typedef void (*FooPtr)(int);
```

```
FooPtr ptr = foo;  
ptr(5);
```

```
// или используя using
```

```
using FooPtr = void (*)(int);
```

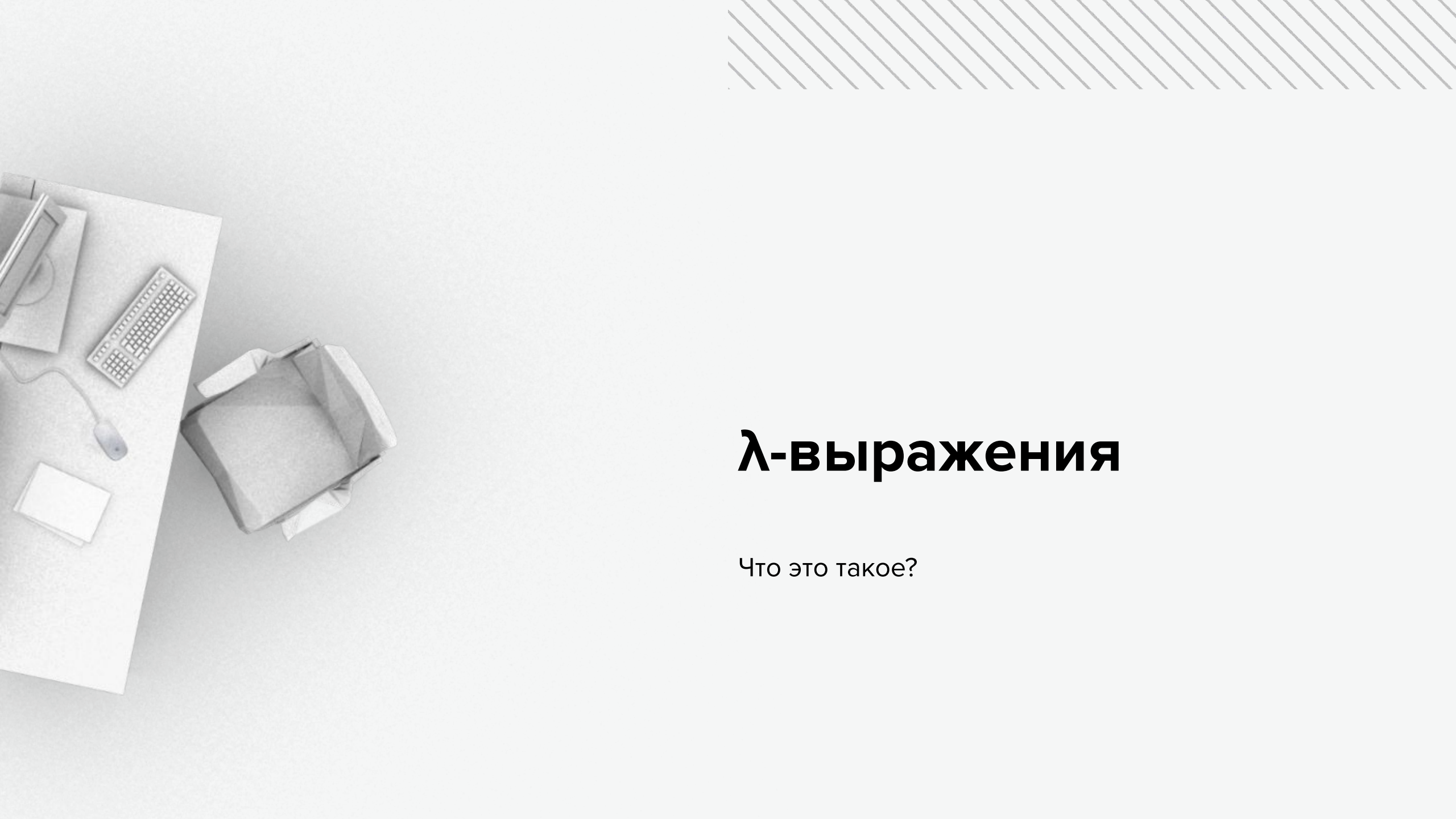
Функции высшего порядка

- Функция высшего порядка — функция, принимающая в качестве аргументов другие функции или возвращает другую функцию в качестве результата.

```
void sort(int* data, size_t size, bool (*compare)(int x, int y));
```

```
bool less(int x, int y)
{
    return x < y;
}
```

```
sort(data, 100, less);
```



λ-выражения

Что это такое?

Рассмотрим код

```
bool isEven(int i) { return i % 2 == 0; }
```

```
void foo()  
{  
    std::vector<int> arr;  
    ...  
    std::find_if(std::begin(arr), std::end(arr), isEven);  
}
```

λ-выражения

- Замыкание (closure) представляет собой объект времени выполнения, создаваемый лямбды-выражением.

`[список_захвата](список_параметров) { тело_функции }`

или

`[список_захвата](список_параметров) -> тип_возвращаемого_значения
{ тело_функции }`

λ-выражения

```
void foo()  
{  
    std::vector<int> arr;  
    auto comp = [](int i) { return i % 2 == 0 };  
    ...  
    std::find_if(std::begin(arr), std::end(arr), isEven);  
}
```

Список захвата

- Если не указать &, то будет захват по значению, то есть копирование объекта; если указать, то по ссылке (нет копирования, модификации внутри функции отразятся на оригинальном объекте).

// Захват всех переменных в области видимости по значению

```
auto foo = [=]() {};
```

// Захват всех переменных в области видимости по ссылке

```
auto foo = [&]() {};
```

Список захвата

- [] // без захвата переменных из внешней области видимости
- [=] // все переменные захватываются по значению
- [&] // все переменные захватываются по ссылке
- [x, y] // захват x и y по значению
- [&x, &y] // захват x и y по ссылке
- [in, &out] // захват in по значению, а out — по ссылке
- [=, &out1, &out2] // захват всех переменных по значению,
- // кроме out1 и out2, которые захватываются по ссылке
- [&, x, &y] // захват всех переменных по ссылке, кроме x



std::function<>



std::function

- Шаблон стандартной библиотеки C++11, который обобщает идею указателя на функцию;
- Может ссылаться на любой вызываемый объект, т.е. на всё, что может быть вызвано как функция;
- Могут хранить, копировать и вызывать произвольные вызываемые объекты — функции, λ -выражения, выражения связывания и другие функциональные объекты;

std::function

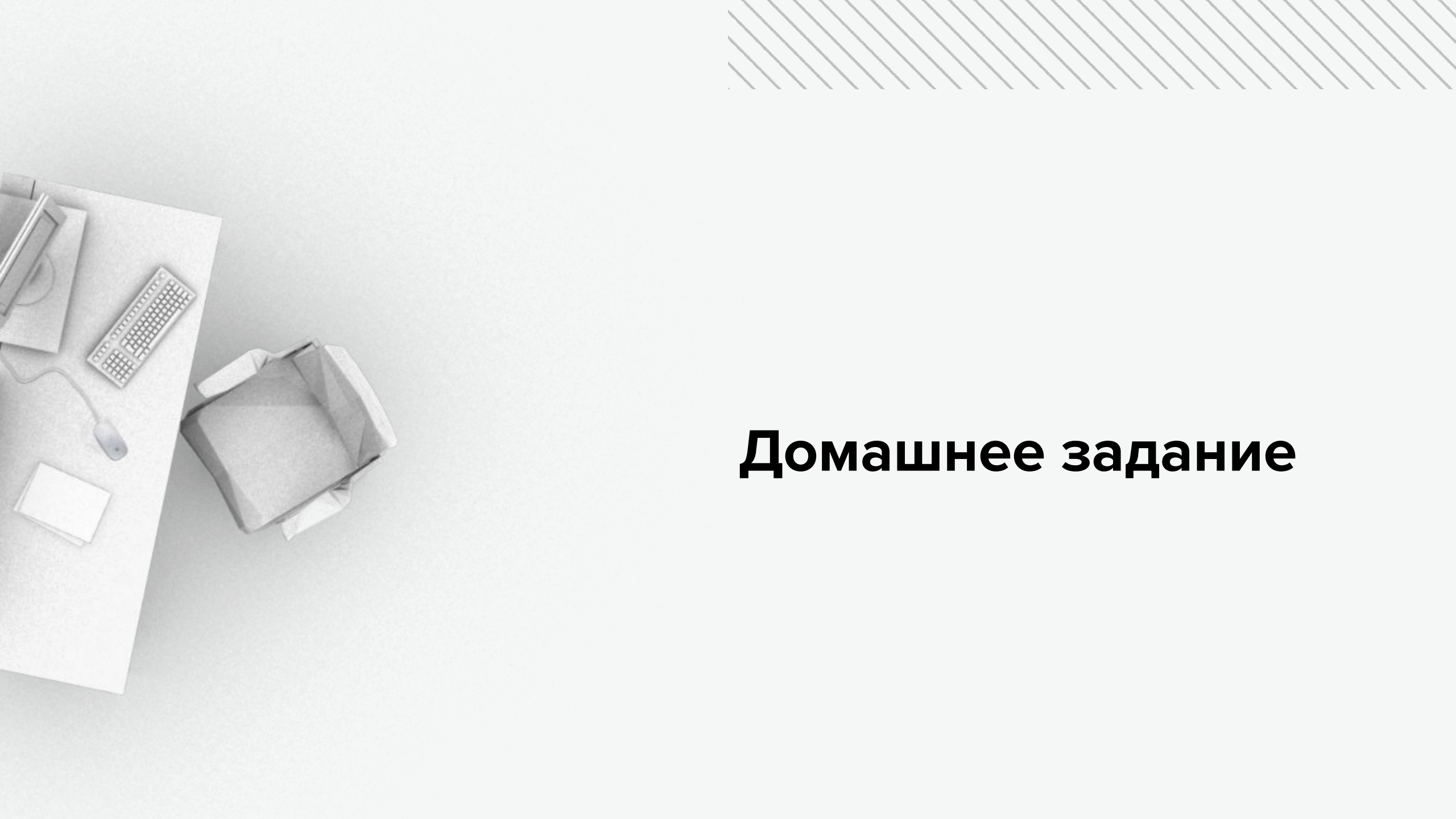
```
#include <functional>
```

```
using MoveFunction =  
    std::function<void (int& x, int& y)>;
```

```
void moveLeft(int &x, int &y) {}
```

```
MoveFunction getRandomDirection() { ... }
```

```
std::vector<MoveFunction> trajectory =  
{  
    [](int& x, int& y) { ... },  
    moveLeft,  
    getRandomDirection()  
};
```

Домашнее задание



Домашнее задание (1)


Необходимо написать библиотеку-парсер строк состоящих из следующих токенов

- строка
- число

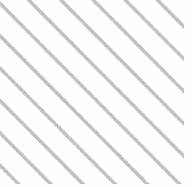
Число состоит из символов от 0 до 9, строка - все остальные символы. Токены разделены пробелами, символами табуляции и перевода строки.



Домашнее задание (2)



Пользователь библиотеки должен иметь возможность зарегистрировать callback-функцию вызываемую каждый раз, когда найден токен, функция получает токен. Должно быть возможно зарегистрировать свой обработчик под каждый тип токена. Также должны быть колбеки вызываемые перед началом парсинга и по его окончанию.



Домашнее задание по уроку #3

Домашнее задание №2

#044

?

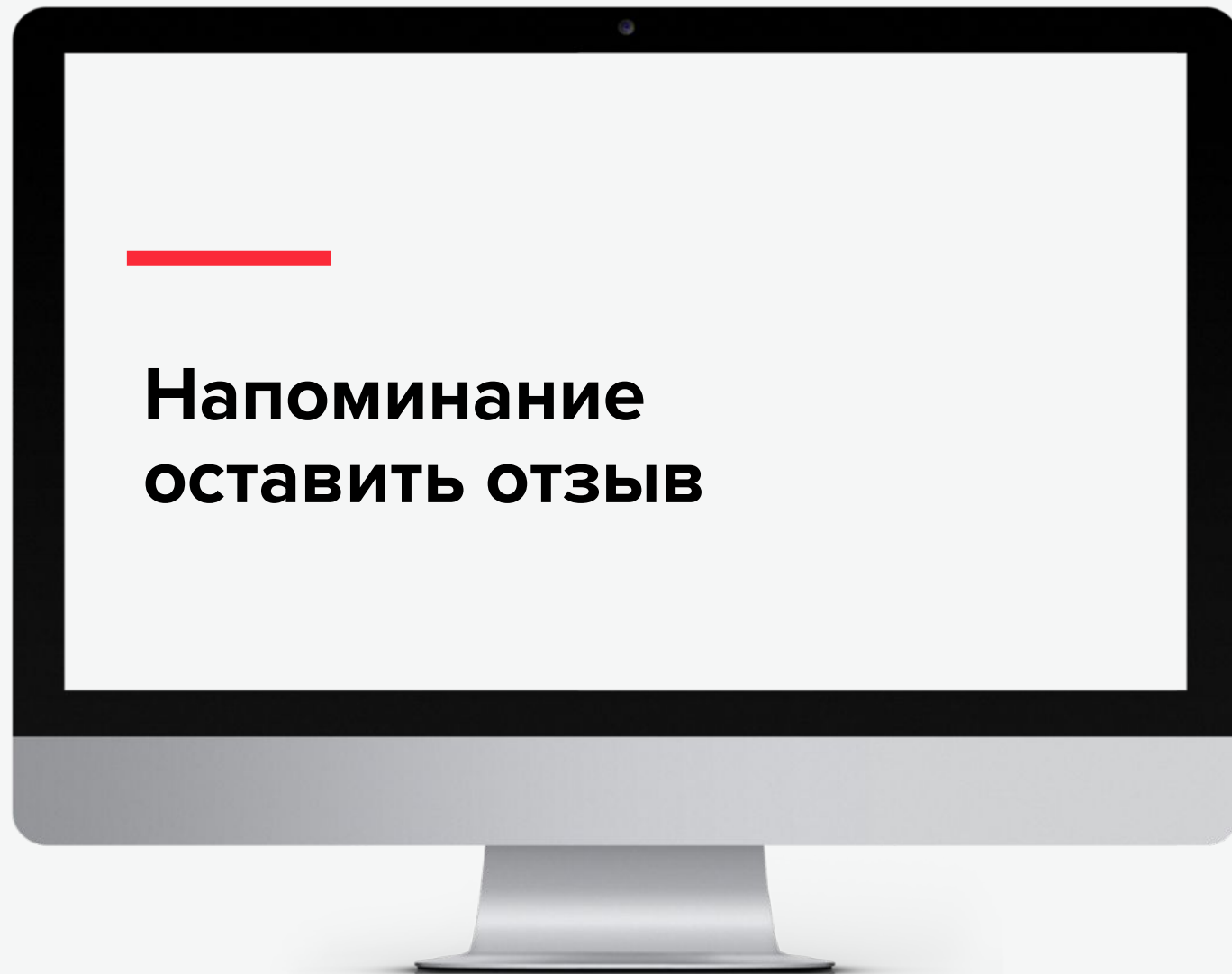
Баллов
за задание

22.10.20

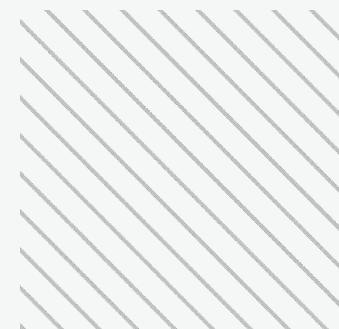
Срок
сдачи

Полезная литература в помощь

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



**Напоминание
ОСТАВИТЬ ОТЗЫВ**



**СПАСИБО
ЗА ВНИМАНИЕ**

