



 **ТЕХНОСФЕРА**

# Исключения

Антон Кухтичев





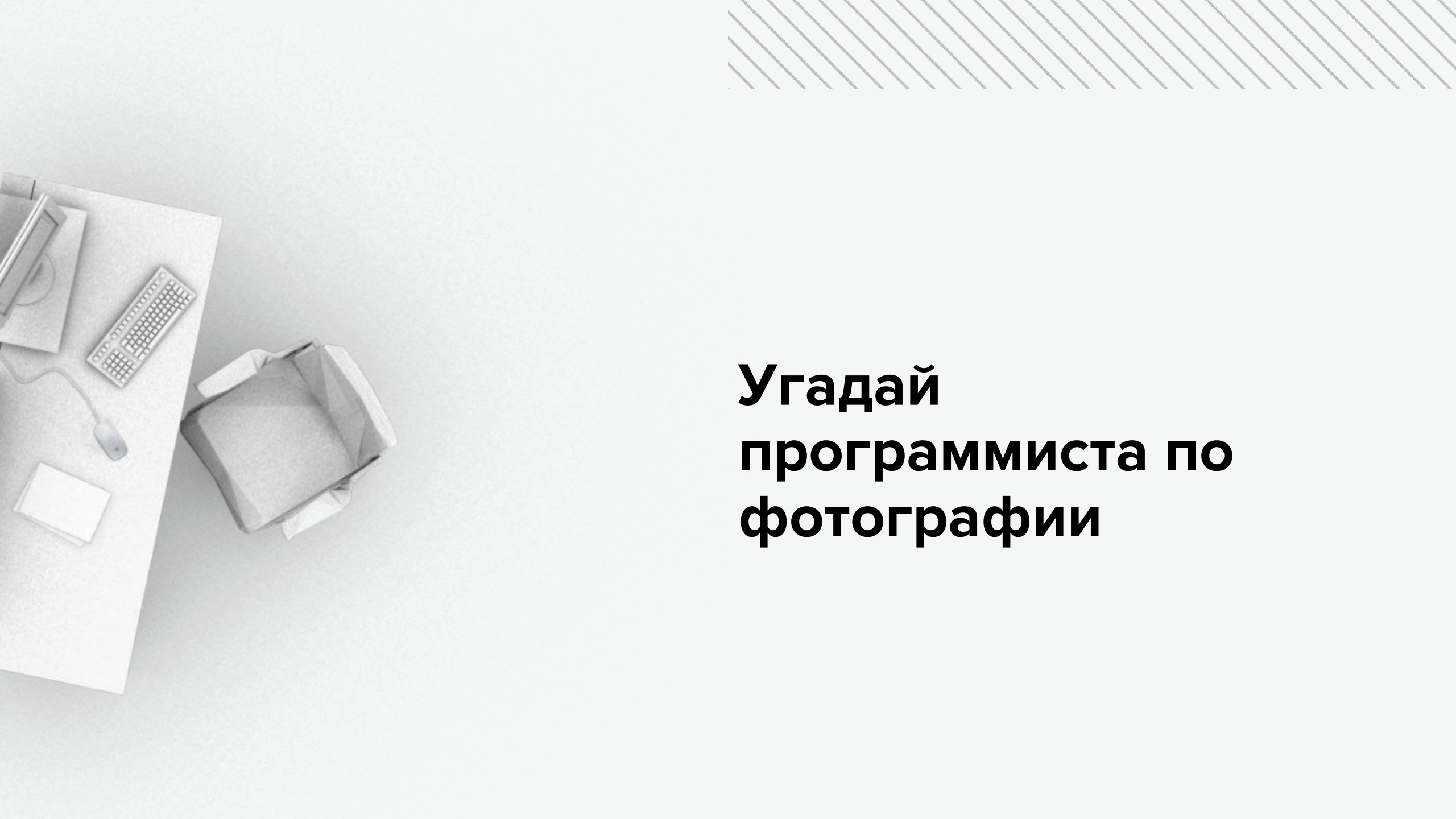
Не забудьте  
отметиться на  
портале!!!

Иначе всё плохо будет.



# Содержание занятия

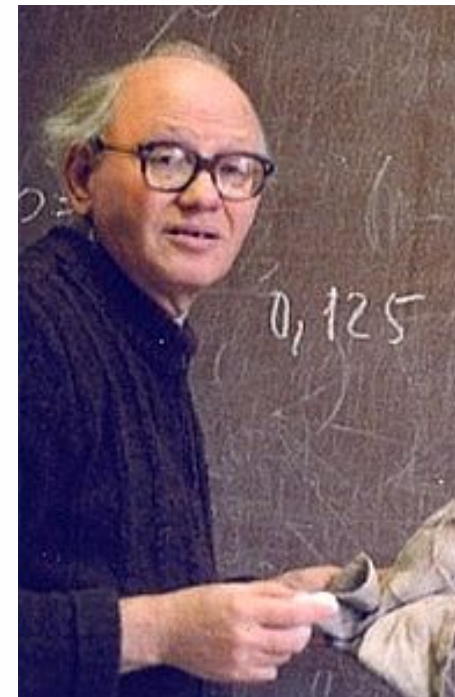
1. Обработка ошибок
2. noexcept
3. Гарантии безопасности исключений (exception safety)
4. Поиск подходящего обработчика
5. Исключения в конструкторе/деструкторе
6. Точки следования (sequence points)



**Угадай  
программиста по  
фотографии**

---

# Кто это?








# Обработка ошибок



# Обработка ошибок

- 
1. Возврат кода ошибки
  2. Исключения

# Возврат кода ошибки

```
enum class Error
{
    Success,
    Failure
};

Error doSomething()
{
    return Error::Success;
}

if (doSomething() != Error::Success)
{
    showError();
}
```





## Возврат кода ошибки

- + Простота
- Ошибку можно проигнорировать
- Делает код громоздким

---

# Code time!



1. Напишем обработку ошибок

# Исключения

```
struct Error
{
    std::string message_;
    const char* fileName_;
    int line_;
    Error(const std::string& message,
          const char* fileName, int line)
        : message_(message)
        , fileName_(fileName)
        , line_(line)
    {
    }
};
```

# Исключения

```
void doSomething()
{
    throw Error(
        "doSomething error", __FILE__, __LINE__);
}

try
{
    doSomething();
}
catch (const Error& error)
{
    showError();
}
```



# Исключения

- Вопросы производительности
- При неправильном использовании могут усложнить программу
- + Нельзя проигнорировать



## Что такое исключительная ситуация?

Ошибка которую нельзя обработать на данном уровне и игнорирование которой делает дальнейшую работу программы бессмысленной.



**noexcept**



# noexcept

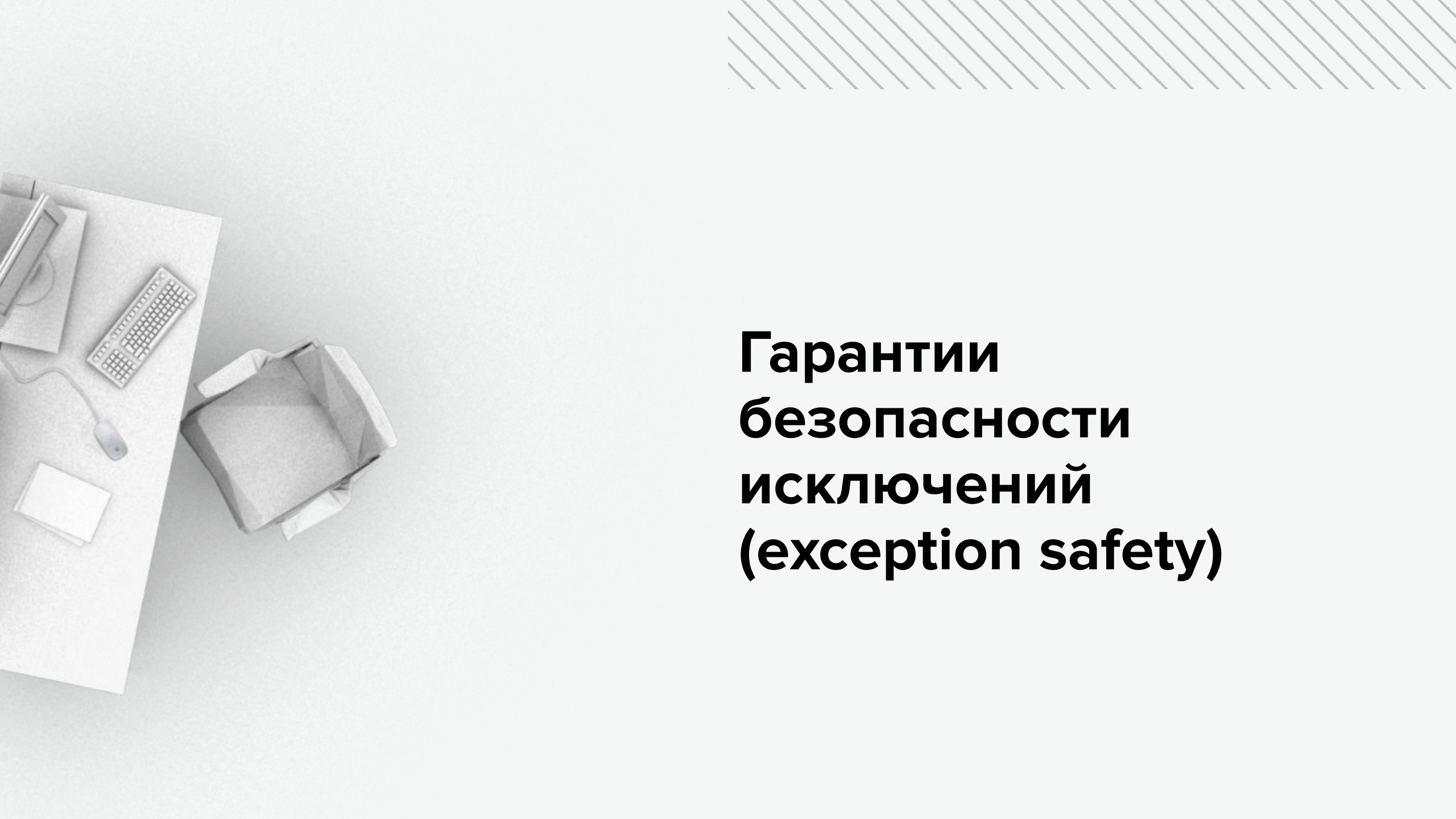
```
void foo() noexcept  
{  
}
```

noexcept говорит компилятору, что функция не выбрасывает исключений - это позволяет компилятору генерировать более компактный код, но если фактически исключение было выброшено, то будет вызвана функция terminate.

# noexcept

- noexcept является частью интерфейса функции, а это означает, что вызывающий код может зависеть от наличия данного модификатора.
- Функции, объявленные как noexcept, предоставляют большие возможности оптимизации, чем функции без такой спецификации
- Спецификация noexcept имеет особое значение для операции перемещения, обмена, функций освобождения памяти и деструкторов





# **Гарантии безопасности исключений (exception safety)**




## Гарантировано исключений нет (No-throw guarantee)

Операции всегда завершаются успешно, если исключительная ситуация возникла она обрабатывается внутри операции.



## Строгая гарантия (Strong exception safety)



Также известна как коммит ролбек семантика (commit/rollback semantics).  
Операции могут завершиться неудачей, но неудачные операции  
гарантированно не имеют побочных эффектов, поэтому все данные сохраняют  
свои исходные значения.

## Строгая гарантия (Strong exception safety)

```
std::vector<int> source = ...;
try
{
    std::vector<int> tmp = source;
    tmp.push_back(getNumber());
    tmp.push_back(getNumber()); // <-- Исключение
    tmp.push_back(getNumber());
    source.swap(tmp);
}
catch (...)
{
    return;
}
```

## Базовая гарантия (Basic exception safety)

Выполнение неудачных операций может вызвать побочные эффекты, но все инварианты сохраняются и нет утечек ресурсов (включая утечку памяти). Любые сохраненные данные будут содержать допустимые значения, даже если они отличаются от того, что они были до исключения.

```
source.push_back(getNumber());  
source.push_back(getNumber()); // <-- Исключение  
source.push_back(getNumber());
```

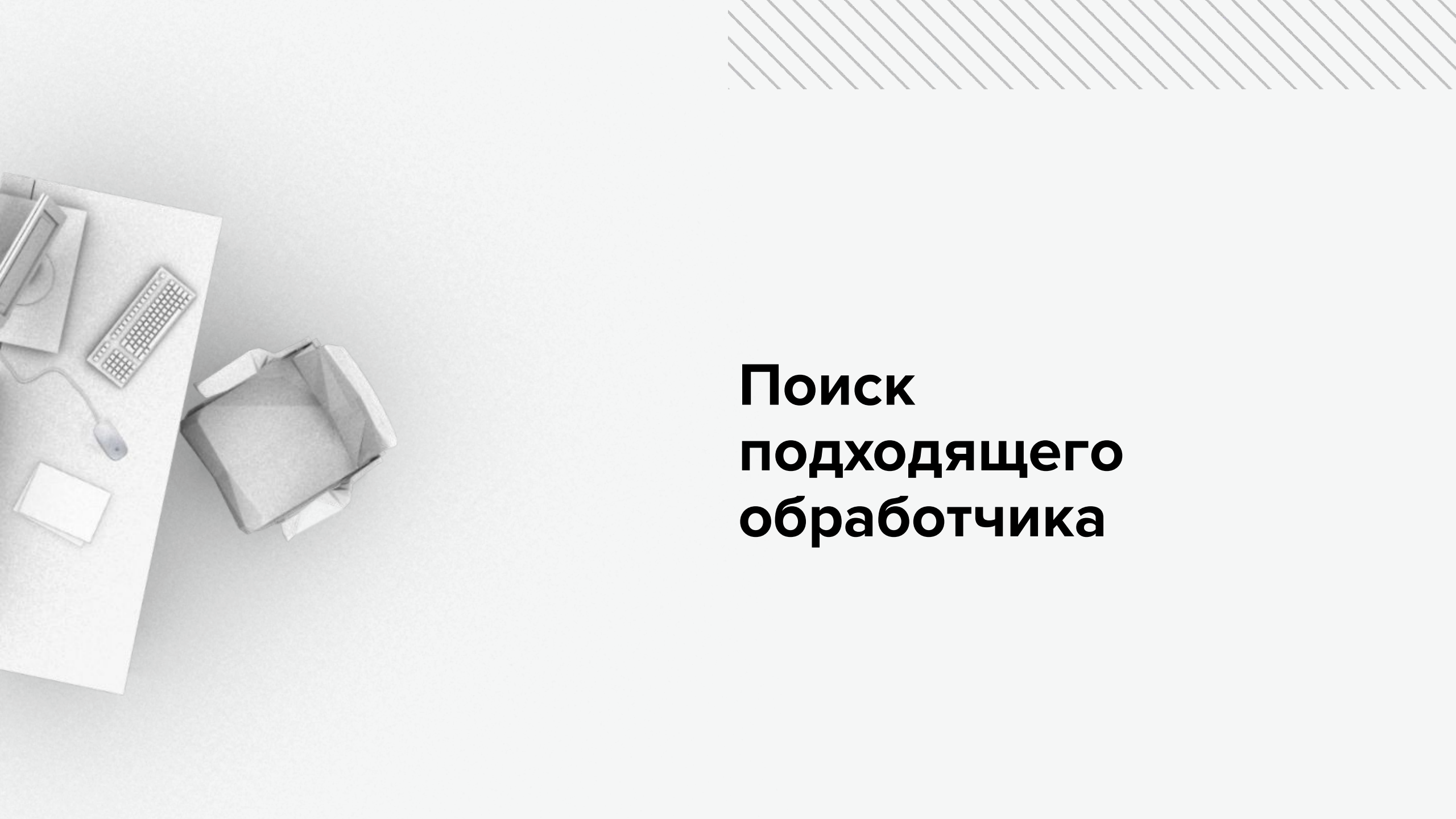




# Никаких гарантий (No exception safety)

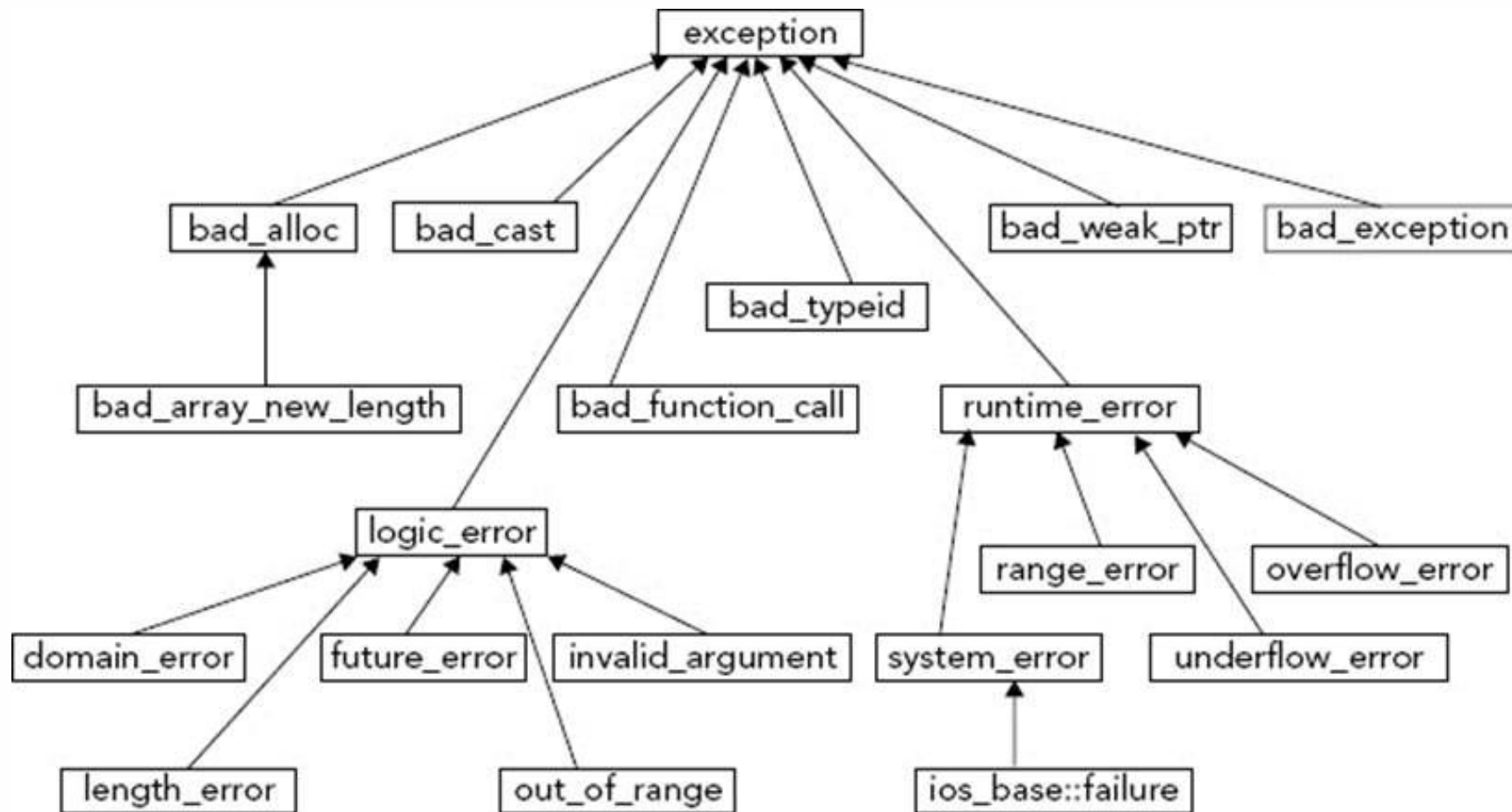


Так делать не стоит.



# **Поиск подходящего обработчика**

# Иерархия исключений



# Поиск подходящего обработчика

1. Поиск подходящего обработчика идет в порядке следования обработчиков в коде
2. Полного соответствия типа не требуется, будет выбран первый подходящий обработчик
3. Если перехватывать исключение по значению, то возможна срезка до базового класса
4. Если наиболее общий обработчик идет раньше, то более специализированный обработчик никогда не будет вызван
5. Три точки - перехват любого исключения

# Поиск подходящего обработчика

1. Поиск подходящего обработчика идет в порядке следования обработчиков в коде
2. Полного соответствия типа не требуется, будет выбран первый подходящий обработчик
3. Если перехватывать исключение по значению, то возможна срезка до базового класса
4. Если наиболее общий обработчик идет раньше, то более специализированный обработчик никогда не будет вызван
5. Три точки - перехват любого исключения

---

## Поиск подходящего обработчика идет в порядке следования обработчиков в коде

```
try {  
    ...  
} catch (...) {  
    ...  
}  
catch (std::invalid_argument &ex) { // этот блок никогда  
    ...                             // не получит управление  
}
```

## Полного соответствия типа не требуется, будет выбран первый подходящий обработчик

```
try {  
    ...  
} catch (std::logic_error & ex) {  
    ...  
}  
catch (std::invalid_argument &ex) { // этот блок никогда  
    ...                             // не получит управление  
}
```



---

## Три точки - перехват любого исключения

```
try {  
    ...  
} catch (...) {  
    ...  
}
```



# Раскрутка стека

Поиск подходящего обработчика вниз по стеку вызовов с вызовом деструкторов локальных объектов - раскрутка стека.

Если подходящий обработчик не был найден вызывается стандартная функция terminate.

# Раскрытие стека

```
struct A {};  
struct Error {};  
struct FileError : public Error {};  
  
void foo() {  
    A a1;  
    throw Error();  
}
```

```
void bar() {  
    A a2;  
    try  
    {  
        A a3;  
        foo();  
    }  
    catch (const FileError&)  
    {  
    }  
}  
  
bar();
```



# terminate

Вызывает стандартную функцию C - abort.

abort - аварийное завершение программы, деструкторы объектов вызваны не будут.

Поведение terminate можно изменить установив свой обработчик функцией set\_terminate.

# Исключения под капотом

```
struct A
{
    A() {}
    ~A() {}
};
```

```
void bar() noexcept
{
}
```

```
void foo()
{
    A a;
    bar();
}
```

```
A::A() [base object constructor]:
    ret
```

```
A::~~A() [base object destructor]:
    ret
```

```
bar():
    ret
```

```
foo():
    push    rbp
    mov     rbp, rsp
    sub     rsp, 16
    lea     rdi, [rbp - 8]
    call    A::A() [base object constructor]
    call    bar()
    lea     rdi, [rbp - 8]
    call    A::~~A() [base object destructor]
    add     rsp, 16
    pop     rbp
    ret
```

# Убираем noexcept

```
struct A
{
    A() {}
    ~A() {}
};

void bar() {}

void foo()
{
    A a;
    bar();
}
```

```
A::A() [base object constructor]:
    ret
A::~~A() [base object destructor]:
    ret
bar():
    ret
foo():
    call    A::A() [base object constructor]
    call    bar()
    jmp     .LBB1_1
.LBB1_1:
    call    A::~~A() [base object destructor]
    ret
.LBB1_2: # landing pad
    call    A::~~A() [base object destructor]
    call    _Unwind_Resume
```

## Добавляем блок catch

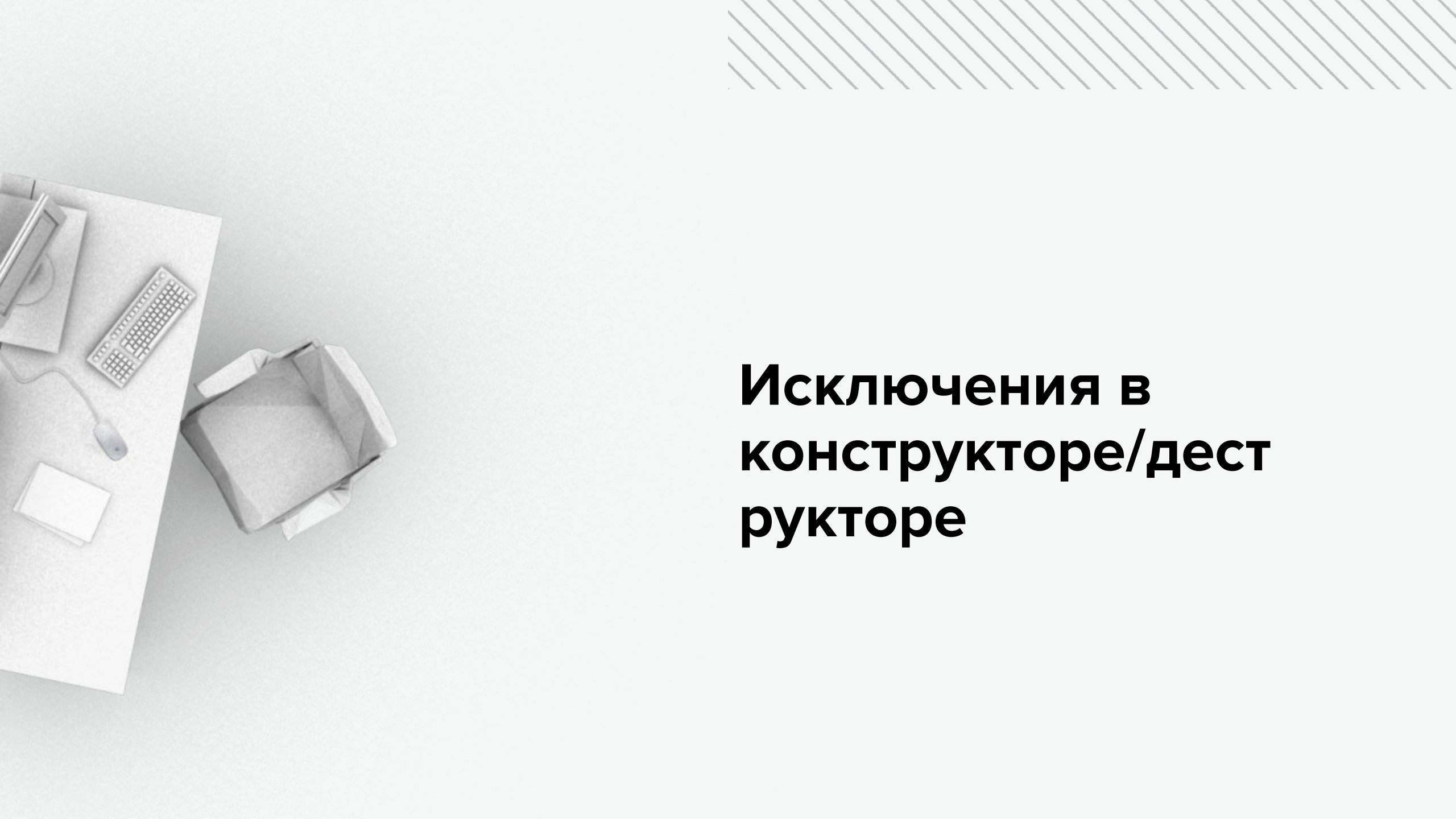
```
struct A
{
    A() {}
    ~A() {}
};

void bar() {}
void baz() noexcept {}

void foo()
{
    A a;
    try {
        bar();
    }
    catch (...) {
        baz();
    }
}
```

```
foo():
    call    A::A() [base object constructor]
    call    bar()
    jmp     .LBB2_1
.LBB2_1:
    jmp     .LBB2_5
.LBB2_2:
    call    __cxa_begin_catch
    call    baz()
    call    __cxa_end_catch
    jmp     .LBB2_4
.LBB2_4:
    jmp     .LBB2_5
.LBB2_5:
    call    A::~~A() [base object destructor]
    ret
.LBB2_6:
    call    A::~~A() [base object destructor]
    call    _Unwind_Resume
```





# Исключения в конструкторе/дест рукторе

# Исключения в конструкторе

В C++ удаляются только *полностью сконструированные* объекты, то есть такие, конструкторы которых уже завершили выполнение кода.



1. Скотт Мейерс. Наиболее эффективное использование C++. Правило 10: Не допускайте утечки ресурсов в конструкторе..



# Исключения в деструкторе

Исключение покинувшее деструктор во время раскрутки стека или у глобального/статического объекта приведет к вызову terminate.

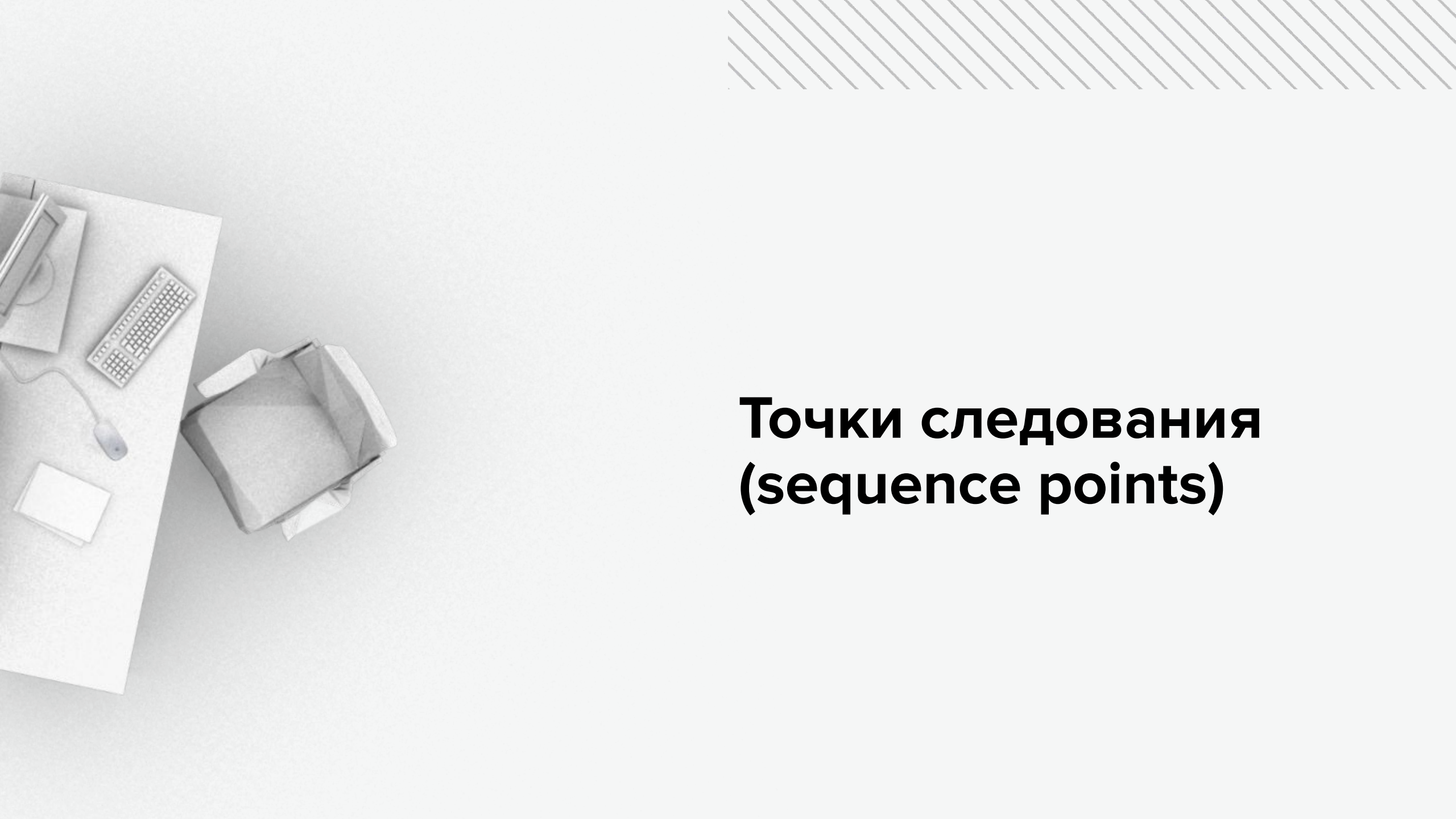
Начиная с C++11 все деструкторы компилятором воспринимаются как помеченные noexcept - теперь исключения не должны покидать деструктора никогда.

---

# Code time!



1. Рассмотрим чем грозит исключения в конструкторе;
2. Реализуем потокоНЕбезопасный shared\_ptr;



# **Точки следования (sequence points)**



## Точки следования (sequence points)

Точки следования - это точки в программе, где состояние реальной программы полностью соответствует состоянию следуемого из исходного кода.

Точки следования необходимы для того, чтобы компилятор мог делать оптимизацию кода.

# Местонахождение точек

1. В конце каждого полного выражения - ;
2. В точке вызова функции после вычисления всех аргументов
3. Сразу после возврата функции, перед тем как любой другой код из вызываемой функции начал выполняться
4. После первого выражения (a) в следующих конструкциях:
  - a || b
  - a && b
  - a, b
  - a ? b : c

# Примеры

```
foo(  
    std::shared_ptr<MyClass>(new MyClass()),  
    bar());
```

Компилятор может заменить это выражение на следующее:

```
auto tmp1 = new MyClass();  
auto tmp2 = bar();  
auto tmp3 = std::shared_ptr<MyClass>(tmp1);  
foo(tmp1, tmp3);
```



# Примеры

`i = ++i; // undefined behavior, переменная модифицируется дважды`

`i = i + 1; // все в порядке`

`i ? i=1 : i=5; // все в порядке (там, где знак ? есть точка следования, а потом выполнится лишь одно из выражений)`

`i=1; i++; // все в порядке (после каждого выражения находится точка следования)`

`i=1, i++; // все в порядке (на операторе запятая находится точка следования)`

---

# Примеры

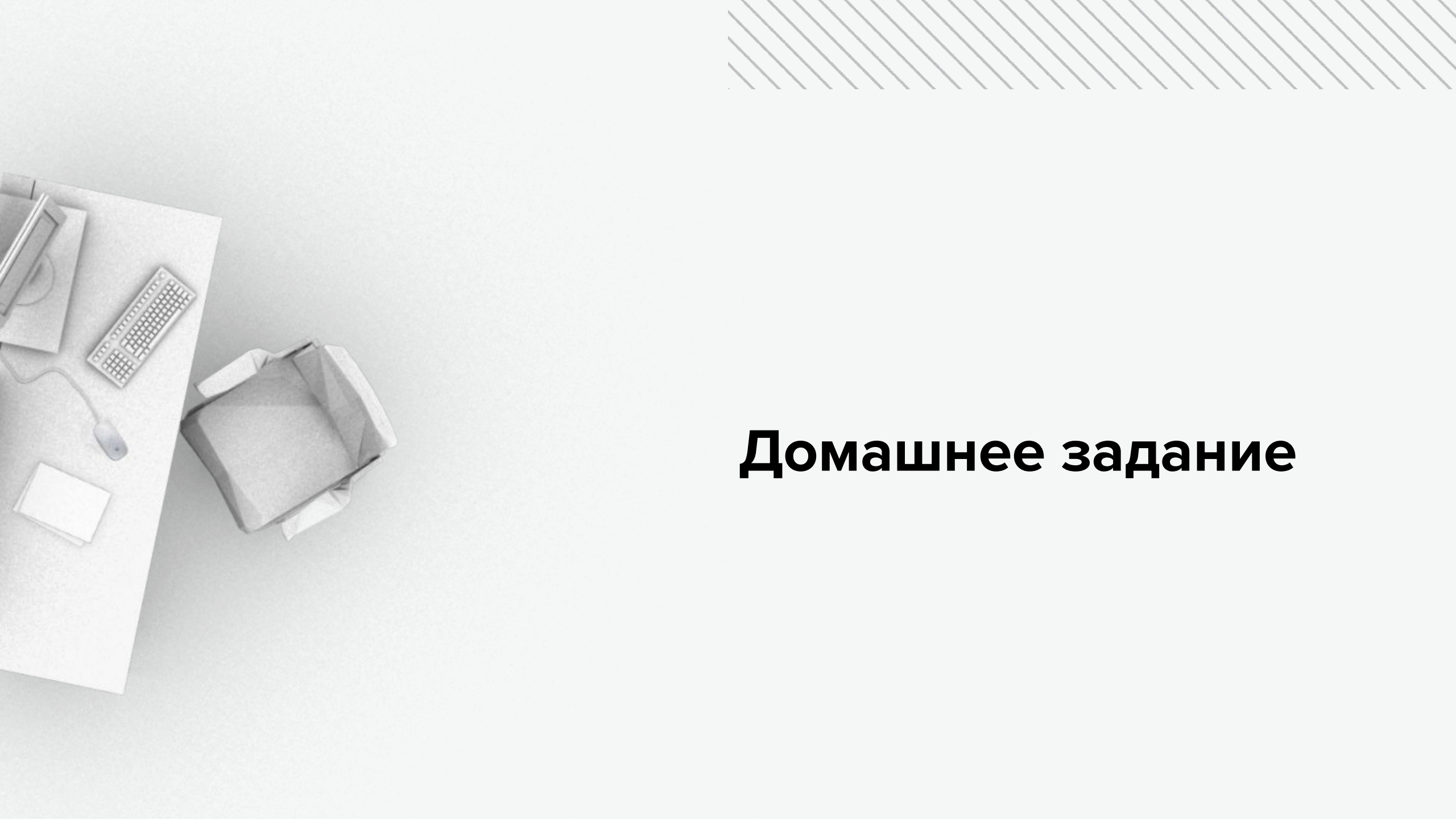
```
void f(int, int);
```

```
int g();
```

```
int h();
```

```
f(g(), h());
```

По Стандарту неизвестно, какая из функций `g` или `h` будет вызвана первой, но известно, что `f()` будет вызвана последней.



# Домашнее задание

## Домашнее задание (1)

Написать функцию для форматирования строки, поддерживаться должен любой тип, который может быть выведен в поток вывода. Формат строки форматирования:

```
"{0} any text {1} {0}"
```

Номер в фигурных скобках - номер аргумента. Если аргументов меньше, чем число в скобках, и в случае прочих ошибок выбрасывать исключение `std::runtime_error`

## Домашнее задание (2)

Пример:

```
auto text = format("{1}+{1} = {0}", 2, "one");  
assert(text == "one+one = 2");
```

Фигурные скобки - зарезервированный символ, если встречаются вне контекста {n} выбрасывать исключение `std::runtime_error`.



---

## Домашнее задание по уроку #7

Домашнее задание №6

#050

?

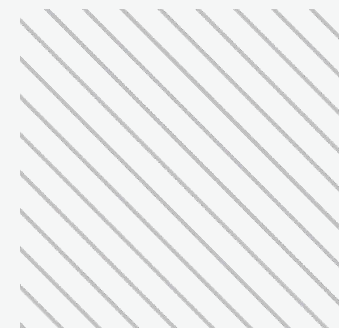
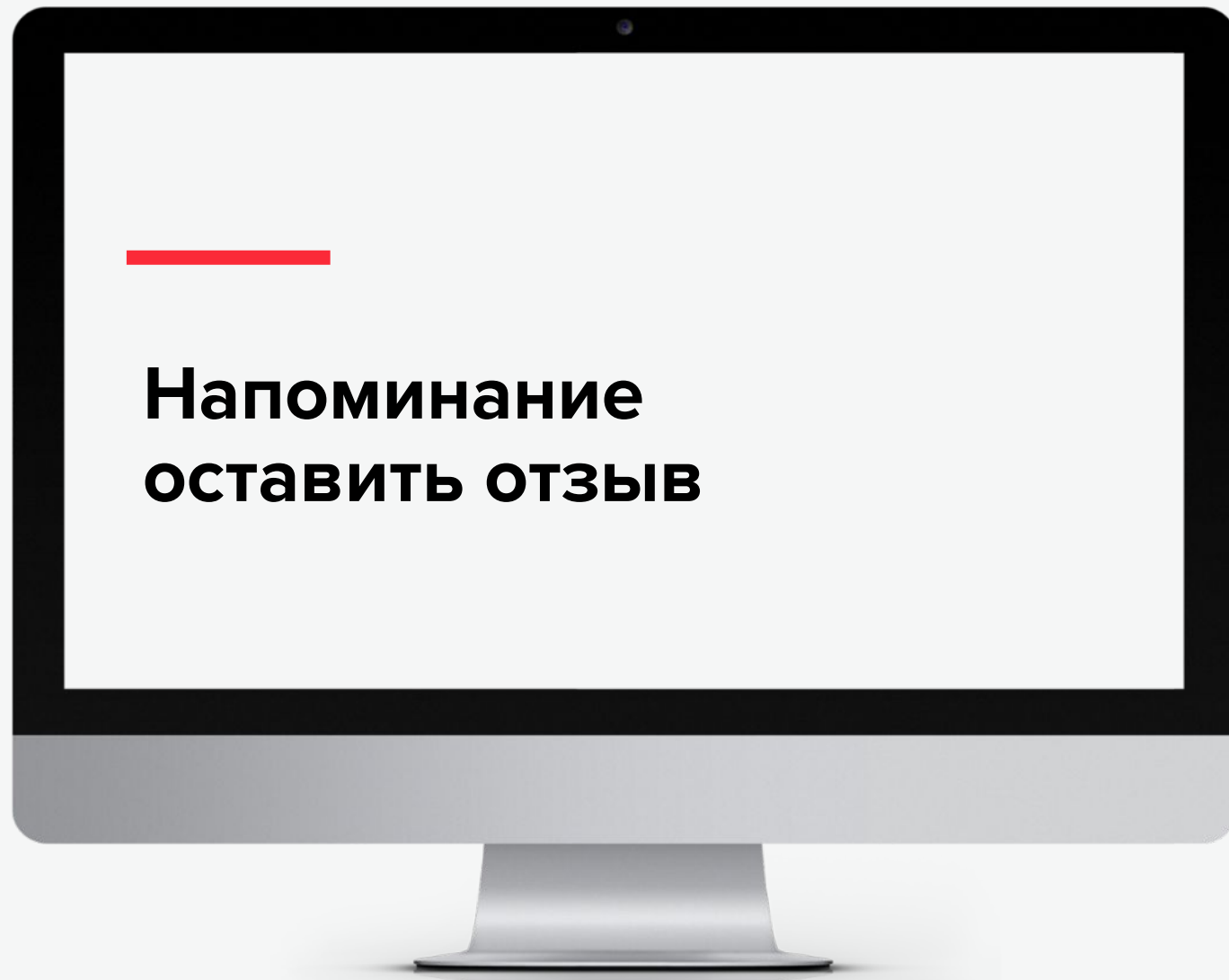
Баллов  
за задание

19.11.20

Срок  
сдачи

### Полезная литература в помощь

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



**СПАСИБО  
ЗА ВНИМАНИЕ**

