

ЛЕКЦИЯ 19

ИСКЛЮЧЕНИЯ

ОСНОВЫ АЛГОРИТМИЗАЦИИ И
ПРОГРАММИРОВАНИЯ



ЛЕКТОР ФУРМАВНИН С.А.

ОБРАБОТКА ОШИБОК В СТИЛЕ C

Определяется область целочисленных кодов ошибок

```
enum error_t { E_OK = 0, E_NO_MEM, E_UNEXCEPTED };
```

Как функция сигнализирует, что ее результат исполнения это E_OK?

ОБРАБОТКА ОШИБОК В СТИЛЕ C

Определяется область целочисленных кодов ошибок

```
enum error_t { E_OK = 0, E_NO_MEM, E_UNEXCEPTED };
```

Вернет код ошибки

```
error_t open_file(const char *name, FILE **handle);
```

Использует thread-local facility, например errno/GetLastError

```
FILE *open_file(const char *name);
```

Вернет error_t* в списке параметров

```
FILE *open_file(const char *name, error_t *errcode);
```

И УЖЕ У НАС ПРОБЛЕМЫ

Замечательная стандартная функция

```
int atoi(const char *nptr);
```

В случае, если конвертировать невозможно, возвращает 0

- Действительно ли возвращать 0 – хорошая идея?

В случае, если число слишком большое, возвращает HUGE_VAL и устанавливает `errno = ERANGE`

- Часто ли вы проверяете на возврат ошибки и HUGE_VAL в частности?

ПРОБЛЕМЫ В C++

```
class MyVector {  
    double *arr_ = nullptr;  
    size_t size_, used_ = 0;  
public:  
    MyVector(size_t sz) : size_(sz) {  
        arr_ = static_cast<double*>(malloc(sizeof(double) * sz))  
    }  
    // тут все остальное
```

Видите проблему в коде?

ПРОБЛЕМЫ В C++

```
class MyVector {  
    double *arr_ = nullptr;  
    size_t size_, used_ = 0;  
public:  
    MyVector(size_t sz) : size_(sz) {  
        arr_ = static_cast<double*>(malloc(sizeof(double) * sz))  
        // тут должна быть обработка случая arr_ == nullptr  
    }  
    // тут все остальное
```

Не обработана ситуация, когда malloc возвращает nullptr

ЧЕМ ЭТО ЧРЕВАТО?

```
MyVector v(100);  
// тут объект v может оказаться в несогласованном состоянии  
// v.arr_ = 0 т.к. память закончилась  
// v.size_ = 100 т.к. конструктор не обработал ошибку
```

- Хуже всего то, что объект в несогласованном состоянии никак не отличается от нормального объекта
- Несогласованность может появиться через тысячи строк кода
- Это даже не UB. Несогласованное состояние вполне корректно

ПОПЫТКА РЕШЕНИЯ: IOSTREAM STYLE

```
class MyVector {
    double *arr_ = nullptr;
    size_t size_, used_ = 0;
    bool valid_ = true;
public:
    MyVector(size_t sz) : size_(sz) {
        arr_ = static_cast<double*>(malloc(sizeof(double) * sz))
        if (!arr_) valid_ = false;
    }
    bool is_valid() const { return valid_; }
    // тут все остальное
```


ОБСУЖДЕНИЕ

Покритикуйте решение в стиле потоков ввода-вывода

```
MyVector v(1000);
```

```
if (!v.is_valid())  
    return -1;
```

```
// здесь используем v
```

Кому это нравится?

КОПИРОВАНИЕ И ПРИСВАИВАНИЕ

Кажется, такой вектор тяжело использовать

```
MyVector v(1000); assert(v.is_valid());  
MyVector v2(v); assert(v2.is_valid());  
v2.push_back(3); assert(v2.is_valid());  
v = v2; assert(v.is_valid());
```

Есть идеи по лучше?

ПЕРЕГРУЗКА ОПЕРАТОРОВ

Делает вещи ещё хуже

```
Matrix operator+(Matrix a, Matrix b);
```

- Здесь неоткуда вернуть код возврата
- И поскольку это отдельная функция, здесь негде хранить goodbit
- Конечно мы всё еще можем вернуть errno. Кому нравится идея его проверять в таких случаях?

ОСНОВНАЯ ИДЕЯ РЕШЕНИЯ

Выйти из вызванной функции в вызывающий код в обход обычных механизмов возврата управления

Аннотировать этот **нелокальный** метод информацией о случившемся

Но что вообще мы знаем о нелокальных переходах?

ТИПЫ ПЕРЕДАЧИ УПРАВЛЕНИЯ

Локальная передача управления

- условные операторы
- циклы
- локальный goto
- прямой вызов функции и возврат из них

Нелокальная передача управления

- косвенный вызов функции (по указателю, например)
- возобновление/приостановка сопрограммы
- **исключения**
- переключение контекста потоков
- нелокальный longjmp и вычисляемый goto

ИСКЛЮЧЕНИЯ

- Исключительные ситуации уровня аппаратуры (например `undefined instruction exception`)
- Исключительные ситуации уровня операционной системы (например, `data page fault`)
- Исключения C++ (только о них и будем мы говорить)

ИСКЛЮЧИТЕЛЬНЫЕ СИТУАЦИИ

Ошибки (исключительными ситуациями не являются)

- рантайм ошибки, после которых состояние не восстановимо (например, segmentation fault)
- ошибки контракта функции (assertion failure из-за неверных аргументов, невыполненные предусловия вызова)

Исключительные ситуации

- Состояние программы должно быть восстановимо (например: исчерпание памяти или отсутствие файла на диске)
- Исключительная ситуация не может быть обработана а том уровне, на котором возникла (программа сортировки не обязана знать, что делать при нехватке памяти на временный буфер)

ПОРОЖДЕНИЕ ОШИБКИ

```
struct UnwShow {  
    UnwShow() { std::cout << "ctor\n"; }  
    ~UnwShow() { std::cout << "dtor\n"; }  
};  
int foo(int n) {  
    UnwShow s;  
    if (n == 0) abort(); // abort - это убийство  
    foo(n - 1);  
}  
foo(4); // что на экране?
```


ПОРОЖДЕНИЕ ОШИБКИ

```
struct UnwShow {  
    UnwShow() { std::cout << "ctor\n"; }  
    ~UnwShow() { std::cout << "dtor\n"; }  
};  
int foo(int n) {  
    UnwShow s;  
    if (n == 0) abort();  
    foo(n - 1);  
}  
foo(4); // что на экране?
```

```
ctor  
ctor  
ctor  
ctor  
ctor  
тут программа прерывается
```


ПОРОЖДЕНИЕ ИСКЛЮЧЕНИЯ

```
struct UnwShow {  
    UnwShow() { std::cout << "ctor\n"; }  
    ~UnwShow() { std::cout << "dtor\n"; }  
};  
  
int foo(int n) {  
    UnwShow s;  
    if (n == 0) throw 1;  
    foo(n - 1);  
}  
  
// ВЫЗОВ ВНУТРИ try-блока  
foo(4); // что на экране?
```

```
ctor  
ctor  
ctor  
ctor  
ctor  
dtor  
dtor  
dtor  
dtor  
dtor
```

тут программа входит в
try-блок


PACKPYTKA CTEKA



foo(3)
s: 0x74fd60
foo(2)
s: 0x74fd10
foo(1)
s: 0x74fcc0
foo(0)
s: 0x74fc70

```
#0 UnwShow::~UnwShow(this=0x74fc70) at exception_ex.cpp:10
#1 0x0000000000401627 in foo(n=0) at exception_ex.cpp:10
#2 0x0000000000401627 in foo(n=1) at exception_ex.cpp:21
#3 0x0000000000401627 in foo(n=2) at exception_ex.cpp:21
#4 0x0000000000401627 in foo(n=3) at exception_ex.cpp:21
```

PACKPYTKA CTEKA



foo(3)
s:0x74fd60
foo(2)
s:0x74fd10
foo(1)
s:0x74fcc0

```
#0 UnwShow::~UnwShow(this=0x74fc70) at exception_ex.cpp:10
#1 0x0000000000401627 in foo(n=0) at exception_ex.cpp:10
#2 0x0000000000401627 in foo(n=1) at exception_ex.cpp:21
#3 0x0000000000401627 in foo(n=2) at exception_ex.cpp:21
```

БОЛЬШЕ ПРО THROW

Конструкция **throw <expression>** означает следующее:

- Создать объект исключения
- Начать раскрутку стека

Примеры:

```
throw 1;  
throw new int(1);  
throw MyClass(1, 1);
```

Исключения отличаются от ошибок тем, что их нужно **ЛОВИТЬ**.

ЛОВЛЯ ИСКЛЮЧЕНИЙ

Производится внутри **try** блока

```
int divide(int x, int y) {  
    if (y == 0) throw OVF_ERROR; // это так себе идея  
    return x / y;  
}  
  
// где-то далее  
try {  
    c = divide(a, b);  
} catch (int x) {  
    if (x == OVF_ERROR) std::cout << "Overflow" << std::endl;  
}
```

НЕКОТОРЫЕ ПРАВИЛА

- Ловля по точному типу

```
try { throw 1; } catch(long l) {}; // не поймали
```

- Или по ссылке на точный тип

```
try { throw 1; } catch(const int& ci) {}; // поймали
```

- Или по указателю на точный тип

```
try { throw new int(1); } catch(int *pi) {}; // поймали
```

- Или по ссылке или указателю на базовый класс

```
try { Derived d; } catch(Base &b) {}; // поймали
```

НЕКОТОРЫЕ ПРАВИЛА

- Catch-блоки пробуются в порядке перечисления

```
try { throw 1; }  
catch(long l) {} // не поймали  
catch(const int &ci) {} // поймали
```

- Пойманную переменную можно менять или удалять

```
try {throw new Derived();} catch(Base *b) {delete b;} // ОК
```

- Пойманное исключение можно перевыбросить

```
try {throw Derived();} catch(Base &b) {throw;} // ОК
```


ОБСУЖДЕНИЕ

Чуть раньше был приведен следующий код для обработки ошибки переполнения:

```
enum class errs_t { OVF_ERROR, UDF_ERROR, и так далее };  
int divide(int x, int y) {  
    if (y == 0) throw OVF_ERROR; // это так себе идея  
    return x / y;  
}
```

- Покритикуйте, что тут плохо?
- Как можно улучшить этот код?

ОБСУЖДЕНИЕ

Очевидное улучшение – переход к классам исключений

```
class MathErr { информация об ошибке };  
class DivByZero : public MathErr { расширение };  
int divide(int x, int y) {  
    if (y == 0) throw DivByZero("Division by zero occurred!");  
    return x / y;  
}  
// где-то дальше  
catch(MathErr &e) {std::cout << e.what() << std::endl; }
```

НЕКОТОРЫЕ НЕПРИЯТНОСТИ

Какие проблемы вы видите в этом коде?

```
class MathErr { информация об ошибке };  
class Overflow : public MathErr { расширение };  
// где-то дальше  
try {  
    тут много опасного кода  
}  
catch(MathErr e) { обработка всех ошибок }  
catch(Overflow o) { обработка переполнения }
```

НЕКОТОРЫЕ НЕПРИЯТНОСТИ

Какие **еще** проблемы вы видите в этом коде?

```
class MathErr { информация об ошибке };
class Overflow : public MathErr { расширение };
// где-то дальше
try {
    тут много опасного кода
}
// 1. Правильный порядок: от частных к общим
// 2. Ловим строго по косвенности
catch(Overflow &o) { обработка переполнения }
catch(MathErr &e) { обработка всех ошибок }
```

КАК ИЗБЕЖАТЬ САМОБИТНОСТИ

Тут все неплохо, но... Неужели я первый, кто наткнулся на такие ошибки?

```
class MathErr { информация об ошибке };
class Overflow : public MathErr { расширение };
// где-то дальше
try {
    тут много опасного кода
}
// 1. Правильный порядок: от частных к общим
// 2. Ловим строго по косвенности
catch(Overflow &o) { обработка переполнения }
catch(MathErr &e) { обработка всех ошибок }
```

СТАНДАРТНЫЕ КЛАССЫ ИСКЛЮЧЕНИЙ

`std::exception`

`std::bad_alloc`

`std::bad_function_call`

`std::runtime_error`

`std::bad_cast`

`std::bad_typeid`

`std::logic_error`

`std::bad_exception`

`std::bad_weak_ptr`

СТАНДАРТНЫЕ КЛАССЫ ИСКЛЮЧЕНИЙ

`std::runtime_error`

`std::logic_error`

`std::range_error`

`std::overflow_error`

`std::domain_error`

`std::length_error`

`std::regex_error`

`std::underflow_error`

`std::invalid_argument`

`std::out_of_range`

`std::system_error`

`std::future_error`

ОБСУЖДЕНИЕ

Какой бы интерфейс вы сделали бы у `std::exception`?

ОБСУЖДЕНИЕ

Какой бы интерфейс вы сделали бы у `std::exception`?

```
struct exception {  
    exception() noexcept;  
    exception(const exception&) noexcept;  
    exception& operator=(const exception&);  
    virtual ~exception();  
    virtual const char* what() const noexcept;  
};
```

Аннотация `noexcept` означает обещание, что эта функция не выбросит исключений. Она распространяется на переопределения виртуальных функций

ИСПОЛЬЗУЕМ СТАНДАРТНЫЕ КЛАССЫ

Наследование от стандартного класса вводит расширение в иерархию

```
class MathErr : public std::runtime_error { информация };
class Overflow : public MathErr { расширение };
// где-то дальше
try {
    тут много опасного кода
}
catch(Overflow &o) { обработка переполнения }
catch(MathErr &e) { обработка всех ошибок }
```

Впрочем, у наследования есть и темные стороны...

МНОЖЕСТВЕННОЕ НАСЛЕДОВАНИЕ

```
struct my_exc1 : std::exception {  
    char const* what() const noexcept override;  
};  
struct my_exc2 : std::exception {  
    char const* what() const noexcept override;  
};  
struct your_exc3 : my_exc1, my_exc2 {};  
  
int main() {  
    try { throw your_exc3(); }  
    catch(std::exception const &e) {std::cout << e.what();}  
    catch(...) {std::cerr << "whoops!" << std::endl; }  
}
```

ПЕРЕХВАТ ВСЕХ ИСКЛЮЧЕНИЙ

Используется троеточие

```
try {  
    // тут много опасного кода  
} catch(...) {  
    // тут обрабатываются все исключения  
}
```

Сама идея, что можно как-то осмысленно обработать любое исключение
очень сомнительная

НЕЙТРАЛЬНОСТЬ

Функция называется нейтральной относительно исключений, если она не ловит чужих исключений

Хорошо написанная функция в хорошо спроектированном коде как минимум нейтральна

У меня проблема!
`throw MyException()`

А я испорчу вам праздник
`try{что-то}`
`catch(...) {}`

Я знаю, как решить проблему
`try{что-то}`
`catch(MyException &e) {обработка}`

ПЕРЕВЫБРОС

- Единственное разумное применение catch-all – это очистка критического ресурса и перевыброс исключения
- На самом деле даже разумность этого варианта под сомнением

```
int *critical = new int[10000]();  
try {  
    // тут много опасного кода  
}  
catch(...) {  
    delete [] critical;  
    throw;  
}
```

Кто-нибудь предложит лучше?

ОБСУЖДЕНИЕ

Кажется, есть одно место, где мы не можем поймать исключение

```
struct Foo {  
    S x_, y_;  
    Foo(int x, int y) : x_(x), y_(y) { // <-exception in x_(x)  
        try {  
            // some actions  
        }  
        catch(std::exception &e) {  
            // some processing  
        }  
    }  
};
```

С одной стороны, вроде и не нужно ловить. Или может быть нужно?

TRY-БЛОКИ УРОВНЯ ФУНКЦИЙ

Мы можем завернуть всю функцию в try блок

```
int foo() try { bar(); }  
catch(std::exception &e) { throw;}
```

В том числе и конструктор

```
Foo::Foo(int x, int y) try : x_(x), y_(y) {  
    // some actions  
}  
catch(std::exception &e) {  
    // some processing  
}
```

Техника весьма экзотическая, но о ней лучше знать, чем наоборот

CATCH УРОВНЯ ФУНКЦИЙ

На уровне функций catch входит в scope функции

```
int foo(int x) try { bar(); }
```

```
catch(std::exception &e) {  
    std::cout << x << ":" << e.what() << std::endl; // ok  
}
```

Увы, try-block на main не ловит исключения в конструкторах глобальных объектов

ИСКЛЮЧЕНИЯ ДЛЯ ЛУЧШЕГО КОДА

Преимущества

- Текст не замусоривается обработкой кодов возврата или `errno`, вся обработка ошибок отделена от логики приложения
- Ошибки не игнорируются по умолчанию. Собственно они не могут быть проигнорированы

Недостатки

- Code path disruption – появление в коде неожиданных выходных дуг
- Некоторый оверхед на исключения

ВЕРНЕМСЯ К ИСХОДНОЙ ПРОБЛЕМЕ

```
class MyVector {  
    double *arr_ = nullptr;  
    size_t size_, used_ = 0;  
public:  
    MyVector(size_t sz) : size_(sz) {  
        arr_ = static_cast<double*>(malloc(sizeof(double) * sz))  
        // и что здесь делать?  
    }  
    // тут все остальное
```

Теперь вполне ясно как эта ошибка вообще может быть обработана

ВЕРНЕМСЯ К ИСХОДНОЙ ПРОБЛЕМЕ

```
class MyVector {  
    double *arr_ = nullptr;  
    size_t size_, used_ = 0;  
public:  
    explicit MyVector(size_t sz) : size_(sz) {  
        arr_ = static_cast<double*>(malloc(sizeof(double) * sz))  
        if (!arr) {  
            // и что здесь делать?  
        }  
    }  
    // тут все остальное
```

Теперь вполне ясно как эта ошибка вообще может быть обработана

ВЕРНЕМСЯ К ИСХОДНОЙ ПРОБЛЕМЕ

```
class MyVector {  
    double *arr_ = nullptr;  
    size_t size_, used_ = 0;  
public:  
    explicit MyVector(size_t sz) : size_(sz) {  
        arr_ = static_cast<double*>(malloc(sizeof(double) * sz))  
        if (!arr) {  
            throw std::bad_alloc();  
        }  
    }  
    // тут все остальное
```

Этот код можно упростить, так как по сути тут написан оператор new

ВЕРНЕМСЯ К ИСХОДНОЙ ПРОБЛЕМЕ

```
class MyVector {  
    double *arr_ = nullptr;  
    size_t size_, used_ = 0;  
public:  
    // бросает bad_alloc  
    explicit MyVector(size_t sz) : arr_(new double[sz]),  
        size_(sz) {}  
    // тут все остальное
```

Задача: написать копирующий конструктор

ПРИМЕР КАРГИЛЛА

Все ли понимают, что тут плохо?

```
class MyVector {  
    double *arr_ = nullptr;  
    size_t size_, used_ = 0;  
public:  
  
    MyVector(const MyVector &rhs) {  
        arr_ = new double[rhs.size];  
        size_ = rhs.size_; used_ = rhs.used_;  
        for (size_t i = 0; i != rhs.size_; ++i)  
            arr_[i] = rhs.arr_[i];  
    }  
}
```

ПРИМЕР КАРГИЛЛА

Все ли понимают, что тут плохо?

```
class MyVector {  
    double *arr_ = nullptr;  
    size_t size_, used_ = 0;  
public:  
  
    MyVector(const MyVector &rhs) {  
        arr_ = new double[rhs.size]; // здесь утечка памяти  
        size_ = rhs.size_; used_ = rhs.used_;  
        for (size_t i = 0; i != rhs.size_; ++i)  
            arr_[i] = rhs.arr_[i]; // если здесь исключение  
    }
```


БЕЗОПАСНОСТЬ ОТНОСИТЕЛЬНО ИСКЛЮЧЕНИЙ

- Код, в котором при исключении могут утечь ресурсы, оказаться в несогласованном состоянии объекты и прочее, называется небезопасным относительно исключений
- Каргилл писал: *“I suspect that most members of the C++ community vastly underestimate the skills needed to program with exceptions and therefore underestimate the true costs of their use”*
- И в общем это до сих пор так, хотя прекрасные книги Саттера сильно улучшили общую грамотность программистов.

ГАРАНТИИ БЕЗОПАСНОСТИ

- Базовая гарантия: исключение при выполнении операции может изменить состояние программы, но не вызывает утечек и оставляет все объекты в согласованном (**но не обязательно предсказуемом**) состоянии
- Строгая гарантия: при исключении гарантируется **неизменность состояния** программы относительно задействованных в операции объектов (commit/rollback)
- Гарантия бессбойности: функция не генерирует исключений (noexcept)

БЕЗОПАСНОЕ КОПИРОВАНИЕ

```
S *safe_copy(const S* src, size_t srclsize) {  
    S *dest = new S[srclsize];  
    try {  
        for (size_t idx = 0; idx != srclsize; ++idx)  
            dest[idx] = src[idx];  
    }  
    catch(...) {  
        delete [] dest;  
        throw;  
    }  
    return dest;  
}
```

ТЕПЕРЬ COPY CTOR

```
class MyVector {  
    double *arr_ = nullptr;  
    size_t size_, used_ = 0;  
public:  
  
    MyVector(const MyVector &rhs) :  
        arr_(safe_copy(rhs.arr_, rhs.size_)),  
        size_(rhs.size_), used_(rhs.used_) {}  
};
```

Следующий шаг – оператор присваивания

Вероятно теперь, когда у нас есть `safe_copy`, нам будет совсем просто?

ОПЕРАТОР ПРИСВАИВАНИЯ

Вы видите проблемы в этой реализации?

```
class MyVector {  
    double *arr_ = nullptr;  
    size_t size_, used_ = 0;  
public:  
  
    MyVector& operator=(const MyVector &rhs) {  
        if (this == &rhs) return *this;  
        delete [] arr_;  
        arr_ = safe_copy(rhs.arr_, rhs.size_);  
        size_ = rhs.size_; used_ = rhs.used_;  
        return *this;  
    }  
}
```

ОПЕРАТОР ПРИСВАИВАНИЯ V2

```
class MyVector {  
    double *arr_ = nullptr;  
    size_t size_, used_ = 0;  
public:  
    MyVector& operator=(const MyVector &rhs) {  
        if (this == &rhs) return *this;  
        double *narr = safe_copy(rhs.arr_, rhs.size_);  
        delete [] arr_;  
        arr_ = narr;  
        size_ = rhs.size_; used_ = rhs.used_;  
        return *this;  
    }  
}
```

Теперь ок, но это как-то хрупко и подвержено случайным проблемам

ВНЕЗАПНО SWAP

Вы видите проблемы в этой реализации?

```
class MyVector {  
    double *arr_ = nullptr;  
    size_t size_, used_ = 0;  
public:  
    void swap(MyVector &rhs) {  
        std::swap(arr_, rhs.arr_);  
        std::swap(size_, rhs.size_);  
        std::swap(used_, rhs.used_);  
    }  
}
```

Вроде бы этот метод не бросает исключений и это хочется задокументировать

ИНТЕРЛЮДИЯ: NOEXCEPT

Специальное ключевое слово `noexcept` документирует гарантию бесбойности для кода

```
void swap(MyVector &rhs) noexcept {  
    std::swap(arr_, rhs.arr_);  
    std::swap(size_, rhs.size_);  
    std::swap(used_, rhs.used_);  
}
```

- При оптимизациях компилятор будет уверен, что исключений не будет
- Если они все-таки вылетят, то это сразу `std::terminate`
- Вы не должны употреблять `noexcept` там, где исключения возможны

ЛИНИЯ КАЛБА

```
class MyVector {  
    double *arr_ = nullptr;  
    size_t size_, used_ = 0;  
public:  
    void swap(MyVector &rhs) noexcept;  
    MyVector& operator=(const MyVector &rhs) {  
        if (this == &rhs) return *this;  
        MyVector tmp(rhs); // тут мы можем бросить исключение  
  
        swap(tmp); // тут мы меняем состояние класса  
        return *this;  
    }  
}
```

Это дает строгую гарантию по присваиванию

ПОДУМАЙТЕ ПРО PUSH

```
class MyVector {  
    double *arr_ = nullptr;  
    size_t size_, used_ = 0;  
public:  
    void push(double new_elem);  
};
```

Может потребоваться реаллокация, если `size_ == used_`

ЛИНИЯ КАЛБА

При проектировании очень полезно провести в уме эту линию

```
void push(double new_elem) {  
    if (used_ == size_) {  
        MyVector tmp(size_ * 2 + 1);  
        while(tmp.size() < used_)  
            tmp.push(arr_[tmp.size()]);  
        tmp.push(new_elem);  
        swap(*this, tmp); // операция noexcept  
        return;  
    }  
    // и так далее
```

Выше этой линии
инварианты класса
неизменны

Ниже этой линии
операции не кидают
исключений

УСЛОВНЫЙ NOEXCEPT

Некоторые функции непонятно аннотировать `noexcept` или нет?

```
S copy (const S &original) /* noexcept? */ {  
    return original;  
}
```

ОПЕРАТОР NOEXCEPT

Оценивает каждую функцию, задействованную в выражении, но не вычисляет выражение

```
struct ThrowingCtor {ThrowingCtor(){} };  
void foo(ThrowingCtor) noexcept;  
void foo(int) noexcept;  
assert(noexcept(foo(1)) == true);  
assert(noexcept(ThrowingCtor{}) == false);
```

Возвращает false для constant expressions

ОБСУЖДЕНИЕ

Возможна критика: что если деструктор выбросит исключение? Попробуем от этого защититься

```
void destroy(FwdIter first, FwdIter last) {  
    while(first++ != last)  
        try {  
            destroy(&*first);  
        }  
        catch(...) {  
            // что тут делать?  
        }  
}
```

ПРАВИЛО ДЛЯ ДЕКТРУКТОРОВ

- Исключения не должны покидать деструктор
- По стандарту исключение, покинувшее деструктор, если при этом остались необработанные исключения, приводит к вызову `std::terminate` и завершению программы.

РЕКОМЕНДУЕМАЯ ЛИТЕРАТУРА

1. Bjarne Stroustrup – The C++ Programming Language (4th Edition), 2013
2. Grady Booch – Object-oriented Analysis and Design with Applications, 2007
3. Скотт Мейерс, Эффективный современный C++: 42 способа улучшить ваше использование C++11 и C++14
4. Joshua Gerrard – The dangers of C-style casts, CppCon, 2015
5. Ben Deane – Operator Overloading: History, Principles and Practice, CppCon, 2018
6. Titus Winters – Modern C++ Design, CppCon 2018
7. Дональд Кнут, Искусство программирования. Том 1. Основные алгоритмы / Ю.В. Козаченко. - 3-е изд – Москва, Санкт-Петербург: ВИЛЬЯМС, 2018. – 721 с.
8. Дональд Кнут, Искусство программирования. Том 2. Получисленные алгоритмы / Ю.В. Козаченко. - 3-е изд – Москва, Санкт-Петербург: ВИЛЬЯМС, 2018. – 743 с.
9. Дональд Кнут, Искусство программирования. Том 3. Сортировка и поиск / Ю.В. Козаченко. - 3-е изд – Москва, Санкт-Петербург: ВИЛЬЯМС, 2018. – 767 с.