

ИСКЛЮЧЕНИЯ

Механизмы нелокальной обработки ошибок. Гарантии безопасности.
Проектирование с учётом исключений

К. Владимиров, Syntacore, 2022
mail-to: konstantin.vladimirov@gmail.com

➤ Ошибки и исключения

❑ Гарантии безопасности

❑ Детали работы с памятью

❑ Проектирование с исключениями

Обработка ошибок в стиле C

- Определяется область целочисленных кодов ошибок:

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

- Как функция сигнализирует, что результат её исполнения это E_OK?

Обработка ошибок в стиле C

- Определяется область целочисленных кодов ошибок:

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

- Вернёт код ошибки

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

- Использует thread-local facility, например errno/GetLastError

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

- Вернёт error_t* в списке параметров.

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

Проблемы уже в C

- Стандартная функция.

```
long strtol(const char *str, char **str_end, int base);
```

- В случае, если конвертировать невозможно, возвращает 0.
- Действительно ли возвращать ноль хорошая идея?
- В случае, если число слишком большое, возвращает LONG_MAX и устанавливает `errno = ERANGE`.
- Часто ли вы проверяли возврат на `ERANGE`?

Проблема в C++

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

- Вы видите в чём проблема в этом коде?

Проблема в C++

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

- Не обработана ситуация когда malloc возвращает nullptr

Чем нам грозит эта ситуация?

```
MyVector v(100);
```

```
// тут объект v может оказаться в несогласованном состоянии
```

```
// v.arr_ = 0 т.к. память кончилась
```

```
// v.size_ = 100 т.к. конструктор никак не обработал ошибку
```

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

Попытка решения: iostream style

```
template <typename T> class MyVector {  
    T *arr_ = nullptr;  
    size_t size_, used_ = 0;  
    bool valid_ = true;  
  
public:  
    MyVector(size_t sz): size_(sz) {  
        arr_ = static_cast<T*>(malloc(sizeof(T) * sz));  
        if (!arr_) valid_ = false;  
    }  
  
    bool is_valid() const { return valid_; }  
  
    // .... и так далее ....
```

Обсуждение

- Покритикуйте решение в стиле потоков ввода-вывода.

```
MyVector v(1000);  
if (!v.is_valid())  
    return -1;  
  
// здесь используем v
```

- Кому оно нравится?

Копирование и присваивание

- Похоже 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 ли идеи получше?

Перегрузка операторов

- Делает вещи ещё хуже.

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

- Здесь неоткуда вернуть код возврата.
- И поскольку это отдельная функция, здесь негде сохранить goodbit
- Конечно мы всё ещё 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** выход информацией о случившемся.
- Но что вообще мы знаем о нелокальных переходах?

Типы передачи управления

- Локальная передача управления:
 - условные операторы.
 - циклы.
 - локальный goto.
 - прямой вызов функций.
- Нелокальная передача управления:
 - косвенный вызов функций (напр. по указателю).
 - возобновление/приостановка сопрогаммы.
 - **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 они и будут нас далее интересовать).

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 на 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"; }  
};
```

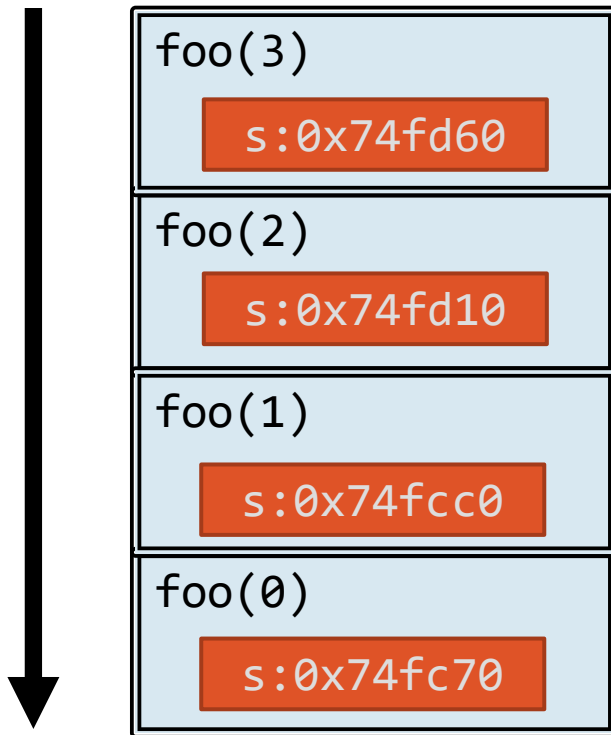
```
int foo(int n) {  
    UnwShow s;  
    if (n == 0) throw 1;  
    foo(n - 1);  
}
```

// ВЫЗОВ **внутри try-блока**

foo(4); // что на экране?

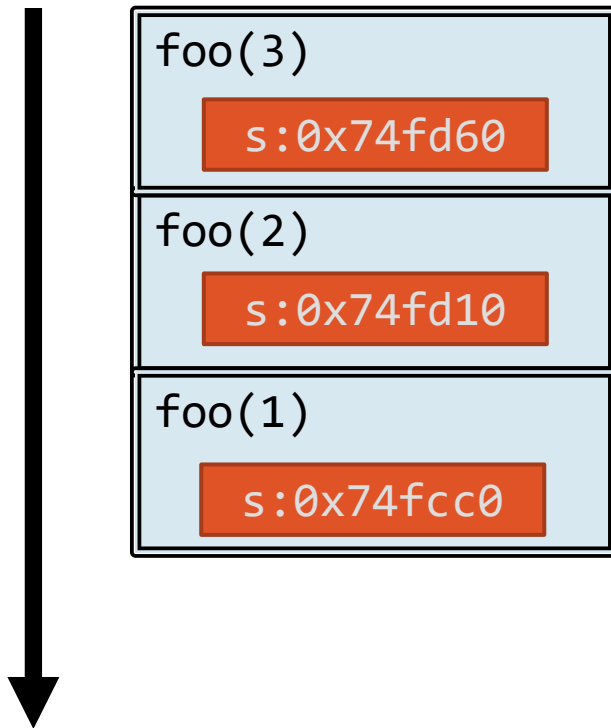
```
ctor  
ctor  
ctor  
ctor  
ctor  
dtor  
dtor  
dtor  
dtor  
dtor  
dtor  
тут программа входит в try блок
```

Раскрытие стека



```
#0 UnwShow::~~UnwShow(this=0x74fc70) at exc.cc:19
#1 0x0000000000401627 in foo(n=0) at exc.cc:29
#2 0x0000000000401627 in foo(n=1) at exc.cc:31
#3 0x0000000000401627 in foo(n=2) at exc.cc:31
#4 0x0000000000401627 in foo(n=3) at exc.cc:31
```

Раскрытие стека



```
#0  UnwShow::~~UnwShow(this=0x74fc70) at exc.cc:19
#1  0x0000000000401627 in foo(n=0) at exc.cc:29
#2  0x0000000000401627 in foo(n=1) at exc.cc:31
#3  0x0000000000401627 in foo(n=2) at exc.cc:31
```

Больше про throw

- Конструкция `throw <expression>` означает следующее:
 - Создать объект исключения.
 - Начать размотку 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 происходит 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 { throw Derived(); } 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; } // ok
```

- Пойманное исключение можно перевыбросить

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

Обсуждение

- Чуть раньше был приведён 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 errs_t::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

- Очевидная проблема здесь это срезка (уже рассматривалась ранее).

```
class MathErr { информация об ошибке };
class Overflow : public MathErr { расширение };

// где-то дальше

try {
    // тут много опасного кода
}
catch (MathErr e) { обработка всех ошибок } // slicing!
catch (Overflow o) { обработка переполнения }
```

Избегаем неприятностей

- Обсуждение: какие **ещё** 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 }
```

Но как избежать самобытности?

- Тут всё неплохо но хм... 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 интерфейс вы бы сделали у `std::exception`?

Обсуждение

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

```
struct exception {  
    exception() noexcept;  
    exception(const exception&) noexcept;  
    exception& operator=(const exception&) noexcept;  
    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 тёмные 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() << "\n"; }  
    catch(...) { std::cerr << "whoops!\n"; }  
}
```

Перехват всех исключений

- Используется троеточие (как в printf).

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

- Сама идея, что можно как-то осмысленно обработать любое исключение **очень** сомнительна.

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;  
}
```

- Ктонибудь предложит лучше?

Обсуждение

- Кажется есть одно место где мы не можем поймать исключение.

```
template <typename T> struct Foo {  
    T 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-block.

```
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 скорее экзотическая, но лучше знать чем не знать.

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.

- ❑ Ошибки и исключения

- Гарантии безопасности

- ❑ Детали работы с памятью

- ❑ Проектирование с исключениями

Вернёмся к исходной проблеме

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

Вернёмся к исходной проблеме

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

- Этот код можно упростить, так как по сути тут написан обычный new.

Вернёмся к исходной проблеме

```
template <typename T> class MyVector {  
    T *arr_ = nullptr;  
    size_t size_, used_ = 0;  
  
public:  
    // бросает bad_alloc  
    explicit MyVector(size_t sz): arr_(new T[sz]), size_(sz) {}  
  
    // .... тут всё остальное ....
```

- Задача: написать копирующий конструктор.

Пример Каргилла

- Все ли понимают что тут плохо?

```
template <typename T> class MyVector {  
    T *arr_ = nullptr;  
    size_t size_, used_ = 0;  
  
public:  
    MyVector(const MyVector &rhs) {  
        arr_ = new T[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?

```
template <typename T> class MyVector {  
    T *arr_ = nullptr;  
    size_t size_, used_ = 0;  
  
public:  
    MyVector(const MyVector &rhs) {  
        arr_ = new T[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"* [3]
- И в общем это до сих пор так, хотя прекрасные книги Саттера [5] и [6] сильно улучшили общую грамотность.

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

```
template <typename T>
T *safe_copy(const T* src, size_t srcsize) {
    T *dest = new T[srcsize];
    try {
        for (size_t idx = 0; idx != srcsize, ++idx)
            dest[idx] = src[idx];
    }
    catch (...) {
        delete [] dest;
        throw;
    }
    return dest;
}
```

Теперь конструктор копирования

```
template <typename T> class MyVector {  
    T *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?

```
template <typename T> class MyVector {  
    T *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

- Вы видите проблемы в этой реализации?

```
template <typename T> class MyVector {  
    T *arr_ = nullptr;  
    size_t size_, used_ = 0;  
  
public:  
    MyVector& operator= (const MyVector &rhs) {  
        if (this == &rhs) return *this;  
        delete [] arr_; // уже стёрли  
        arr_ = safe_copy(rhs.arr_, rhs.size_); // исключение  
        size_ = rhs.size_; used_ = rhs.used_;  
        return *this;  
    } // объект в неконсистентном состоянии
```


Оператор присваивания v2

```
template <typename T> class MyVector {  
    T *arr_ = nullptr;  
    size_t size_, used_ = 0;  
  
public:  
    MyVector& operator= (const MyVector &rhs) {  
        if (this == &rhs) return *this;  
        T *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

```
template <typename T> class MyVector {  
    T *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 оператор не бросает исключений и это хочется задокументировать.

Интерлюдия: noexcept

- Специальное ключевое слово `noexcept` документирует гарантию бессбойности 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 всё-таки вылетят, то это сразу `std::terminate`
- Вы не должны употреблять `noexcept` там где исключения всё же возможны.

Оператор присваивания: линия Калба

```
template <typename T> class MyVector {  
    T *arr_ = nullptr;  
    size_t size_, used_ = 0;  
public:  
    void swap(MyVector& rhs) noexcept;  
  
    MyVector& operator= (const MyVector &rhs) {  
        MyVector tmp(rhs); // тут мы можем бросить исключение  


---

  
        swap(tmp); // тут мы меняем состояние класса  
        return *this;  
    }  
}
```

- Это даёт строгую гарантию по присваиванию.

Подумаем про push?

- Подумайте про push

```
template <typename T> class MyVector {  
    T *arr_ = nullptr;  
    size_t size_, used_ = 0;  
public:  
    void push(T new_elem);
```

- Может потребоваться реаллокация если `size_ == used_`

Kalb line

- При проектировании очень полезно провести в уме эту линию

```
void push(const T& t) {  
    if (used_ == size_) {  
        MyVector tmp(size_*2 + 1);  
        while (tmp.size() < used_)  
            tmp.push(arr_[tmp.size()]);  
        tmp.push(t);  
        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?

```
template <class T>
T copy(T const& orig) /* noexcept? */ {
    return orig;
}
```

Условный noexcept

- Некоторые функции непонятно аннотировать noexcept или нет?

```
template <class T>
T copy(T const& orig) /* noexcept? */ {
    return orig;
}
```

- Эта функция noexcept для int, но не для vector<int>.

Условный noexcept

- Некоторые функции можно различить простыми определителями.

```
template <class T>
T copy(T const& orig) noexcept(is_fundamental<T>::value) {
    return orig;
}
```

- `noexcept(true)` это всё равно что просто `noexcept`.
- `noexcept(false)` это его отсутствие а не обещание что функция точно что-то бросит.
- Решение рабочее, но недостаточно точное.

Оператор noexcept

- Для более тонкой настройки служит оператор noexcept.

```
template <class T>
T copy(T const& orig) noexcept(noexcept(T(orig))) {
    return orig;
}
```

- Оператор noexcept возвращает true или false в зависимости от вычисления выражения под ним на этапе компиляции.
- Разумеется выражение T(orig) выглядит так себе.

Оператор noexcept: альтернативы

- Очень часто, если хочется спросить, лучше спросить явно.

```
template <class T>
T copy(T const& orig) noexcept(
    std::is_nothrow_copy_constructible<T>::value) {
    return orig;
}
```

- Внутри этот определитель реализован через оператор noexcept и настоящее место этого оператора именно там в библиотечном коде.
- Тем не менее, какие-то детали о нём знать полезно.

Оператор noexcept: детали

- 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(foo(ThrowingCtor{})) == false);
```

- Возвращает false для constant expressions.
- Интересно, что разыменованное nullptr это вариант нормы для noexcept.

Обсуждение: `noexcept(false)`

- Любой деструктор по умолчанию `noexcept`
- Одним из способов позволить исключениям покидать деструктор является его пометка как `noexcept(false)`
- Вы должны быть осторожны, помечая так деструкторы потому что деструктор сам по себе используется в процессе размотки (см. пример)
- Вы можете проверить внутри деструктора идёт ли размотка посредством вызова `std::uncaught_exceptions()`

Извлечение из массива

- Безопасен ли этот код относительно исключений?

```
template <typename T> class MyVector {  
    T *arr_ = nullptr;  
    size_t size_, used_ = 0;  
  
public:  
    T pop() {  
        if (used_ <= 0) throw underflow{};  
        T result = arr_[used_ - 1];  
        used_ -= 1;  
        return result;  
    }  
}
```

Внезапная проблема

- Кажется, что всё хорошо
- Но что произойдёт в точке использования?

```
MyVector<SomeType> v;
```

```
// тут много кода
```

```
SomeType s = v.pop(); // исключение при копировании в s
```

- Тогда окажется, что объект уже удалён, но по месту назначения не пришёл и навсегда потерян

Извлечение из массива v2

- Тут правильное проектирование страхует от проблем

```
template <typename T> class MyVector {  
    T *arr_ = nullptr;  
    size_t size_, used_ = 0;  
  
public:  
    T top() const {  
        if (used_ <= 0) throw outofbounds{};  
        return arr_[used_ - 1];  
    }  
  
    void pop() {  
        if (used_ <= 1) throw underflow{}; used_ -= 1;  
    }  
}
```


Обсуждение

- Оказывается безопасность относительно исключений влияет на проектирование!
- Если это так, то почему бы сразу не спроектировать нечто, что нам удобно будет делать безопасным?
- Удивительно, но для этого нам надо будет посмотреть на тонкости работы с памятью

- ❑ Ошибки и исключения
- ❑ Гарантии безопасности
- Детали работы с памятью
- ❑ Проектирование с исключениями

Глобальные операторы

- В языке C для выделения памяти служат функции malloc и free

```
void *p = malloc(10);  
free(p);
```

- В языке C++ этим занимаются операторы new и delete
- При этом в отличии, от, скажем, оператора +, у них есть **глобальные формы**
- Когда вы пишете new-expression для встроенного типа, он будет истолкован именно как вызов глобального оператора

```
int *n = new int(5); // выделение + конструирование  
n = (int *) ::operator new(sizeof(int)); // только выделение
```

Глобальные операторы

- Вы можете переопределить глобальные операторы и изменить поведение всех классов, которые ими пользуются

```
void *operator new(std::size_t n) {  
    void *p = malloc(n); if (!p) throw std::bad_alloc{};  
    printf("Alloc: %p, size is %zu\n", p, n);  
    return p;  
}
```

- Теперь что мы ожидаем увидеть на экране при создании списка из одного элемента?

```
std::list<int> l;  
l.push_back(42);
```

Обсуждение

- Мы отделяем вызов конструкторов от выделения памяти, но что если конструктор выбросит исключение?

```
struct S {  
    S(); // десятый конструктор кинет исключение  
    ~S();  
};
```

```
S *sarr = new S[20];
```

- Сколько тут будет конструкторов и деструкторов, если мы знаем, что `new[]` даёт строгую гарантию?

Формы глобальных операторов

- Основные формы все в чём-то похожи на `malloc`

```
void *operator new(std::size_t);  
void operator delete(void*) noexcept;  
void *operator new[](std::size_t);  
void operator delete[](void*) noexcept;
```

- Предусмотрены также дополнительные варианты с семантикой `noexcept`

```
void *operator new(std::size_t, const std::nothrow_t&) noexcept;  
void *operator new[](std::size_t, const std::nothrow_t&) noexcept;
```

- Пока что должно быть не слишком понятно как их использовать.

Небросающий new

- Если для new-expression не передано аргументов, она раскрывается просто

```
p = new int{42};  
p = (int *) ::operator new(sizeof(int)); *p = 42;
```

- Если аргументы переданы, они ставятся в конец глобального оператора

```
p = new (nothrow) int{42};  
p = (int *) ::operator new(sizeof(int), nothrow); *p = 42;
```

- Специальный аргумент `std::nothrow` типа `std::nothrow_t` показывает, что мы не хотим бросать исключение
- Тогда нам надо возвращать нулевой указатель при неудаче

Размещающий new

- Поскольку аллокация/деаллокация это операторы, они могут быть переопределены
- Но есть непереопределяемый глобальный оператор

```
void* operator new(std::size_t size, void* ptr) noexcept;  
void* operator new[](std::size_t size, void* ptr) noexcept;
```

- Он называется размещающим new и ему не соответствует никакого delete, потому что всё что он делает это размещает объект в сырой памяти

Работа с размещающим new

- Работа с памятью отделена от работы с объектом в памяти

```
void *raw = ::operator new(sizeof(Widget), std::nothrow);  
if (!raw) { обработка }
```

```
Widget *w = new (raw) Widget;
```

```
// .... тут использование w ....
```

```
w->~Widget();
```

```
::operator delete(raw);
```

- Обсуждение: может ли это помочь проектированию безопасных контейнеров?

Переопределение new и delete

- Замечательным свойством new и delete является возможность переопределить их не глобально, а на уровне своего класса

```
struct Widget {  
    static void *operator new(std::size_t n);  
    static void operator delete(void *mem) noexcept;  
};
```

- Теперь для класса Widget будут использоваться его собственные операторы, а не глобальные
- При этом, в отличие от глобального, размещающий new тоже может быть переопределён

Работа с пользовательским классом

- new с исключением при исчерпании памяти

```
Widget *w = new Widget; // возможно bad_alloc
```

- new с возвратом нулевого указателя

```
Widget *w = new (std::nothrow) Widget;  
if (!w) { обработка }
```

- размещающий new

```
void *raw = ::operator new(sizeof(Widget)); // возможно bad_alloc  
  
// только конструирование в готовой памяти  
Widget *w = new (raw) Widget;
```

Обсуждение

- Что вы думаете о таком операторе присваивания?

```
T& T::operator=(T const& x) {  
    if (this != &x) {  
        this->~T();  
        new (this) T(x);  
  
    }  
    return *this;  
}
```

Обсуждение (Stepanov assignment)

- Что вы думаете о таком операторе присваивания?

```
T& T::operator=(T const& x) {  
    if (this != &x) {  
        this->~T();  
        new (this) T(x); // исключение тут  
                        // и дальше dtor при размотке  
    }  
    return *this;  
}
```

- Алекс Степанов написал его в одной из первых реализаций `std::vector` и эта ошибка там **была незамеченной 6 лет**.

- ❑ Ошибки и исключения
- ❑ Гарантии безопасности
- ❑ Детали работы с памятью
- Проектирование с исключениями

Отделённая реализация

- Идея для проектирования ваших классов с учётом исключений это разделить функциональность:
 - Класс, работающий с сырой памятью
 - Используя объекты этого класса внешний класс, работающий с типизированным содержимым
- Для этого часто используется управление памятью вручную через нестандартные формы `new` и `delete`

Полезные хелперы

- Создание копии объекта в сырой памяти

```
template <typename T> void copy_construct(T *p, const T& val) {  
    new (p) T(val);  
}
```

- Разрушение такого объекта без освобождения памяти

```
template <typename T> void destroy(T* p){  
    p->~T();  
}
```


Обсуждение

- Что можно сказать о возможных исключениях в следующем коде, деконструирующем содержимое forward-итерируемого контейнера?

```
template <typename FwdIter>
void destroy(FwdIter first, FwdIter last) {
    while (first != last)
        destroy(&*first++);
}
```

Обсуждение

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

```
template <typename FwdIter>
void destroy(FwdIter first, FwdIter last) {
    while (first++ != last)
        try {
            destroy(&*first);
        }
        catch(...) {
            и что здесь делать?
        }
}
```

Правило для деструкторов

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

Буфер для вектора

- Ключевой момент: конструктор и деструктор

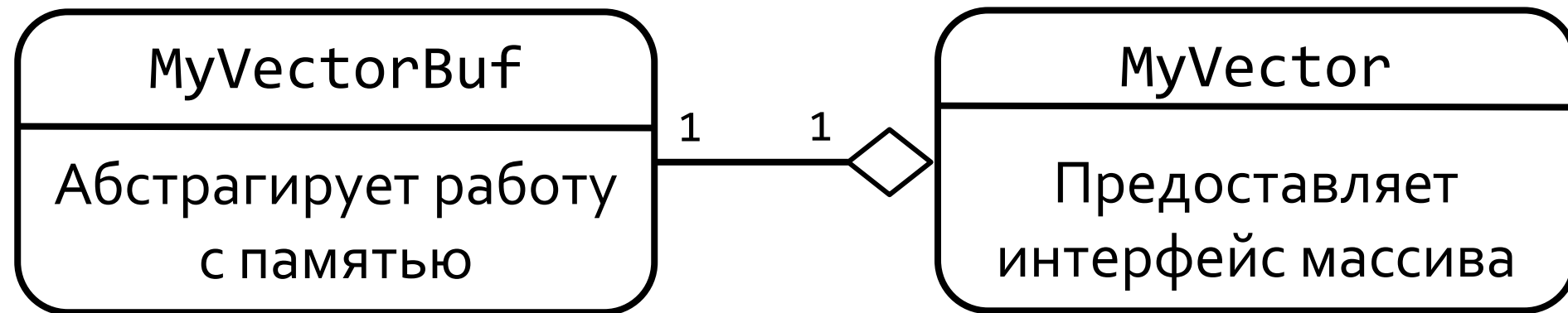
```
MyVectorBuf(size_t sz = 0) :  
    arr_((sz == 0) ?  
        nullptr :  
        static_cast<T*>(::operator new(sizeof(T) * sz))),  
    size_(sz), used_(0) {}  
  
~MyVectorBuf() /* noexcept */ {  
    destroy(arr_, arr_ + used_);  
    ::operator delete(arr_);  
}
```

Собственно вектор

- Тут демонстрация MyVector

Общий вывод и картинка

- Проектирование с использованием исключений в итоге позволяет упростить и улучшить код, структурируя его с чётким распределением ответственности



- В реальной libstdc++ вектор тоже будет устроен по такому принципу

Обсуждение

- Приведенный ранее метод push не очень эффективен

```
void MyVector::push(const T& t) {  
    if (used_ == size_) {  
        MyVector tmp (size_*2 + 1);  
        while (tmp.size() < used_)  
            tmp.push(arr_[tmp.size()]); // копирование  
        tmp.push(t);  
    }
```

- Можем ли мы вместо этого использовать перемещение?

Первая проблема: константность

- Нам придётся немного дублировать чтобы не снимать константность

```
void MyVector::push(const T& t) { T t2(t); push(move(t2)); }
```

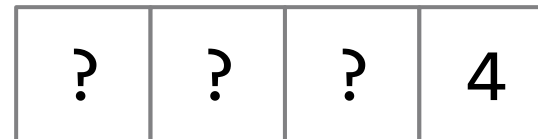
```
void MyVector::push(T&& t) {  
    if (used_ == size_) {  
        MyVector tmp (size_*2 + 1);  
        while (tmp.size() < used_)  
            tmp.push(std::move(arr_[tmp.size()])); // перемещение  
        tmp.push(std::move(t));  
    }
```

- Тут всё хорошо?

Вторая проблема: линия Калба

- Идея сделать его более эффективным использует move
- Но это порождает проблемы: мы портим состояние arr

```
void MyVector::push(T&& t) {  
    if (used_ == size_) {  
        MyVector tmp (size_*2 + 1);  
        while (tmp.size() < used_)  
            tmp.push(std::move(arr_[tmp.size()])); // если тут throw?  
        tmp.push(std::move(t));  
        swap(*this, tmp);  
    }  
    // всё остальное
```



Решение

- Перемещающие конструктор и присваивание не должны бросать исключений

```
MyVector(MyVector &&rhs) noexcept = default;
```

```
MyVector& operator=(MyVector &&rhs) noexcept = default;
```

- При этом если они неправильные или их нет, помещение в контейнер становится менее эффективным

```
void MyVector::push (const T& t) {  
    if (std::is_nothrow_move_assignable<T>::value)  
        push_move(t);  
    else  
        push_copy(t);  
}
```

Смещение линии Калба

- Случай с копированием

```
MyVector tmp(size_*2 + 1);  
while (tmp.size() < used_) tmp.push(arr_[tmp.size()]);  
tmp.push(t);
```

```
swap(*this, tmp);
```

- Случай с перемещением

```
MyVector tmp(size_*2 + 1);
```

```
while (tmp.size() < used_) tmp.push(move(arr_[tmp.size()]));  
tmp.push(move(t));  
swap(*this, tmp);
```

Обсуждение

- Исключения влияют на проектирование
- Использование перемещающих конструкторов влияет на проектирование
- Кажется приходит время обсудить проектирование

Литература

1. ISO/IEC, "Information technology – Programming languages – C++", ISO/IEC 14882:2017
2. The C++ Programming Language (4th Edition)
3. Tom Cargill, Exception handling: a false sense of security, C++Report '1994
4. David Abrahams, Exception-safety in generic components '1998
5. Herb Sutter, Exceptional C++: 47 engineering puzzles, programming problems, and solutions, Addison-Wesley, 2000
6. Herb Sutter, More exceptional C++: 40 new engineering puzzles, programming problems, and solutions, Addison-Wesley, 2002
7. Jon Kalb, Exception Safe code (3 parts), CppCon'2014
8. Arne Mertz, Modern C++ features – keyword `noexcept`, blog post, Jan'2016
9. Niall Douglas, Mongrel Monads, ACCU'2017
10. Nico Brailovsky, [Exception handling internals](#)