Исключения

Введение в объектно-ориентированное программирование

17.04.2025

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++

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

- Вы видите в чём проблема в этом коде?

Проблема в C++

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

- Не обработана ситуация когда malloc вовращает 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.

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

Порождение исключения

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

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 occured");
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

https://en.cppreference.com/w/cpp/error/exception

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() « std::endl; }
catch(...) { std::cerr « "whoops!"; }
}
```

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.

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.

```
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.

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 ....
```

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
}
```

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_; // уже стерли
arr_ = safe_copy(rhs.arr_, rhs.size_); //исключение
size_ = rhs.size_; used_ = rhs.used_;
return *this;
}
```

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 ok, 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