HSE
Faculty of Computer Science

Конспект углубленного курса по C++

AUTHOR
Your Name

COURSE
Advanced C++
Fall 2023

Оглавление

Часть I

Лекция 05 – Ошибки. Исключения. Noexcept

Глава 1

Эволюция обработки ошибок: От C до C++17

Обработка ошибок — это механизм управления потоком выполнения при возникновении нештатных ситуаций. Любая операция, связанная с ресурсами (память, ввод-вывод), является потенциально сбойной.

Пример аллокации памяти:

- **Явный сбой:** Запрос объема памяти, превышающего физические возможности (32 ГБ на машине с 16 ГБ).
- **Скрытый сбой:** Фрагментация адресного пространства, когда суммарно свободной памяти достаточно, но нет непрерывного блока нужного размера.
- **Внешний сбой:** OOM-killer (Out of Memory killer) в Linux, который может завершить процесс при оверкоммите (overcommit).

До появления стандартизированных исключений в C++ (и в языке C) доминировали подходы, основанные на кодах возврата. Рассмотрим их эволюцию и архитектурные недостатки, которые привели к появлению механизма исключений.

Подход языка C: Коды возврата и errno

В языке C функции сообщают об ошибке через возвращаемое значение. Это создает семантическую неоднозначность: одно и то же значение (например, `int`) используется и как результат вычисления, и как индикатор статуса.

Глобальная переменная errno

Классический механизм UNIX — использование глобальной переменной `errno`. Функция возвращает специальное маркерное значение (например, `-1` или `NULL`), а код ошибки записывается в глобальную целочисленную переменную.

```
FILE* f = fopen("config.txt", "r");
if (f == NULL) {
    // Код ошибки в errno
    if (errno == ENOENT) {
        // Файл не найден
```

```
    } else if (errno == EACCES) {
        // Нет прав доступа
    }
}
```

Недостатки подхода: 1. **Потеря контекста.** `errno` — это просто число. Оно не сообщает, какой именно файл не удалось открыть или почему операция ввода-вывода была прервана. 2. **Игнорирование ошибок.** Программист обязан проверять результат *каждого* вызова. На практике это часто игнорируется (например, проверка результата `printf` или `close`). 3. **Проблемы многопоточности.** Изначально `errno` была глобальной, что делало невозможным её использование в multithreading. В современных стандартах это *thread-local* переменная, но архитектурная проблема разделения состояния остается.

Паттерн очистки ресурсов в ядре Linux

В отсутствие деструкторов (RAII) язык C требует ручной очистки ресурсов. Если функция захватывает несколько ресурсов (память, мьютексы, дескрипторы), обработка ошибок превращается в сложную задачу.

В ядре Linux стандартным паттерном является использование `goto` для обратной раскрутки инициализации. Это эмуляция деструкторов: метки располагаются в порядке, обратном захвату ресурсов.

```
int process_file(const char* path) {
    int err = 0;

    void* page = alloc_page();
    if (!page) return -ENOMEM;

    struct inode* node = get_inode(path);
    if (!node) {
        err = -ENOENT;
        goto out_free_page; // Прыжок к освобождению памяти
    }

    if (lock_inode(node) != 0) {
        err = -EIO;
        goto out_put_inode; // Прыжок к освобождению иноды
    }

    // Основная работа...

    unlock_inode(node); // Успешное завершение: освобождаем в прямом порядке
out_put_inode:
    put_inode(node);
out_free_page:
    free_page(page);

    return err;
}
```

Данный подход имитирует `switch-case` без `break`. Чем глубже произошла ошибка, тем «выше» по стеку очистки нужно прыгнуть. Это эффективно, но требует высокой дисциплины

и подвержено ошибкам при рефакторинге (copy-paste ошибок меток).

Подход Go и кортежи возврата (Tuple Returns)

Современные языки (например, Go) пытаются решить проблему неявности `errno`, возвращая пару значений: `(результат, ошибка)`.

```
// Пример на псевдокоде Go
file, err := os.Open("config.txt")
if err ≠ nil {
    return err
}
// Работа с file...
```

В C++ это можно эмулировать через возврат структуры или `std::pair`.

> **Product Type vs Sum Type**
>
> С точки зрения теории типов, возврат пары `(Value, Error)` — это **Тип-Произведение** (Product Type). Множество возможных состояний равно декартову произведению $S = V \times E$.
>
> Это создает семантически некорректные состояния:
>
> 1. `Value` валиден, `Error` валиден (Противоречие: успех и ошибка одновременно).
> 2. `Value` невалиден, `Error` невалиден (Отсутствие результата и отсутствие ошибки).
>
> Корректным подходом является **Тип-Сумма** (Sum Type), реализуемый через `std::variant<Valu Error>` или `std::expected` (C++23), где состояние может быть *либо* результатом, *либо* ошибкой, но не обоими сразу.

Недостатки подхода Go/Tuple в C++: 1. **Раздувание кода.** Проверка `if (err)` занимает 3-4 строки на каждый вызов. 2. **Накладные расходы.** Постоянное конструирование и копирование объектов ошибок, даже если они не нужны (happy path).

Современные оптимизации: std::from_chars

В C++17 был введен заголовок `<charconv>` и функции `std::from_chars`, которые используют подход возврата структуры, напоминающий C-style, но по причинам производительности.

```
#include <charconv>

struct from_chars_result {
    const char* ptr; // Указатель на первый нераспарсенный символ
    std::errc ec;    // Код ошибки (0, если успех)
};

// ...
auto [ptr, ec] = std::from_chars(begin, end, value);
```

```cpp
if (ec != std::errc()) {
    // Обработка ошибки
}
```

На заметку

Почему не исключения? Парсинг чисел — операция, которая часто может завершаться неудачей (например, валидация пользовательского ввода). Выброс исключения — это тяжелая операция (раскрутка стека, RTTI), которая может быть в 100-1000 раз медленнее простой проверки кода возврата. Для высоконагруженных низкоуровневых операций (парсинг JSON, логов) возврат структуры оправдан.

Фундаментальная проблема конструкторов

Ключевая причина введения исключений в C++ — это архитектура объектно-ориентированного программирования, в частности, конструкторов.

Конструктор не имеет возвращаемого значения. Он возвращает сам созданный объект. Если внутри конструктора происходит ошибка (например, `new` не смог выделить память под буфер вектора), у него нет штатного способа сообщить об этом вызывающему коду через `return`.

Антипаттерн: Двухфазная инициализация

До исключений использовался подход с разделением создания и инициализации:

```cpp
class Vector {
    int* data = nullptr;
public:
    Vector() {} // 1. Дешевое создание "пустого" объекта

    // 2. Метод, который может вернуть ошибку
    bool init(size_t size) {
        data = (int*)malloc(size * sizeof(int));
        return data != nullptr;
    }
};

// Клиентский код
Vector v;
if (!v.init(100)) {
    // Обработка ошибки
}
```

> **Важно!**
>
> Этот подход порождает "**Zombie Objects**" — объекты, которые существуют, но находятся в невалидном состоянии.
>
> - Программист может забыть вызвать `init()`.
> - Объект может быть передан в функцию, которая ожидает валидное состояние.
> - Требуется постоянная проверка флагов `is_initialized` внутри каждого метода.

RAII и Исключения

C++ решает эту проблему через идиому **RAII** (Resource Acquisition Is Initialization). Инвариант класса должен быть установлен в конструкторе. Если это невозможно (ошибка), конструктор должен прервать выполнение через выброс исключения.

```
class Vector {
public:
    Vector(size_t size) {
        data = new int[size]; // Если new упадет, полетит std::bad_alloc
        // Если мы здесь, объект ГАРАНТИРОВАННО валиден
    }
};
```

Это гарантирует, что в программе не существует "полуживых" объектов. Либо объект создан корректно, либо его создание было прервано исключением и память очищена (при условии корректного Stack Unwinding, который мы рассмотрим далее).

Резюме раздела

1. Коды возврата (C-style) приводят к игнорированию ошибок и смешиванию бизнес-логики с обработкой сбоев.
2. Подход "Результат + Ошибка" (Go) теоретически некорректен (Product Type) и многословен.
3. Исключения в C++ необходимы для корректной работы конструкторов и предотвращения появления Zombie Objects.
4. Исключения имеют накладные расходы, поэтому в узких местах (парсинг) применяются специальные API без исключений (`std::from_chars`).

Глава 2

Механика исключений: Stack Unwinding и Гарантии

Механизм исключений в C++ — это не просто альтернатива кодам возврата, а сложная инфраструктура времени выполнения (Runtime), тесно связанная с управлением памятью и временем жизни объектов. Понимание процесса «раскрутки стека» (Stack Unwinding) критически важно для написания корректного C++ кода.

Анатомия раскрутки стека (Stack Unwinding)

Когда оператор `throw` выбрасывает исключение, программа перестает выполняться линейно. Среда выполнения (C++ Runtime) начинает процесс поиска подходящего обработчика (`catch`), поднимаясь вверх по стеку вызовов. Этот процесс называется **Stack Unwinding**.

Ключевая особенность этого процесса: по мере выхода из каждой функции (scope), Runtime обязан корректно уничтожить все локальные объекты, созданные на стеке в этом блоке. Для каждого такого объекта вызывается деструктор.

> **RAII (Resource Acquisition Is Initialization)**
>
> Именно гарантия вызова деструкторов при раскрутке стека делает возможной идиому RAII. Если ресурс (память, файл, мьютекс) обернут в объект с деструктором, исключение не приведет к утечке. Если же используется «сырой» `new` без `delete` (который мог бы стоять далее по коду, но выполнение перепрыгнуло его), произойдет утечка памяти.

Рассмотрим класс `Noisy`, который сообщает о своем рождении и смерти:

```cpp
#include <iostream>

struct Noisy {
    int id;
    Noisy(int i) : id(i) {
        std::cout << "Ctor " << id << "\n";
    }
    ~Noisy() {
        std::cout << "Dtor " << id << "\n";
    }
};
```

```
12
13 void func() {
14     Noisy n1(1);
15     Noisy n2(2);
16     throw std::runtime_error("Error");
17     // Сюда управление никогда не дойдет
18     // Деструкторы n2 и n1 будут вызваны автоматически
19 }
```

При вызове `func()` вывод будет следующим:

```
Ctor 1
Ctor 2
Dtor 2  <-- Обратный порядок уничтожения
Dtor 1
```

Runtime гарантирует, что объекты уничтожаются строго в порядке, обратном их созданию (LIFO — Last In, First Out). Это важно, так как объект `n2` может зависеть от `n1`.

Деструкторы и спецификатор noexcept

До стандарта C++11 деструкторы могли выбрасывать исключения так же, как и любые другие функции. Однако практика показала, что это приводит к фатальным ошибкам в архитектуре приложений.

В современном C++ действует правило:

> **Важно!**
>
> Начиная с C++11, все деструкторы по умолчанию неявно помечены как `noexcept(true)`.

Это означает, что если вы попытаетесь выбросить исключение из деструктора и оно вылетит за его пределы, программа немедленно завершится вызовом `std::terminate()`.

Почему принято такое жесткое решение? Ответ кроется в механике взаимодействия двух исключений.

Катастрофа Double Fault: std::terminate

Представьте ситуацию: 1. В блоке `try` происходит ошибка, выбрасывается исключение `E1`. 2. Начинается раскрутка стека (Stack Unwinding). 3. Runtime находит на стеке локальный объект и вызывает его деструктор. 4. Внутри деструктора происходит сбой, и выбрасывается новое исключение `E2`, которое вылетает наружу.

В этот момент в одном потоке существуют два активных (unhandled) исключения: `E1` (которое еще не поймано) и `E2` (которое только что возникло). C++ Runtime не умеет обрабатывать две ошибки одновременно — непонятно, какую из них доставлять в `catch`, и как продолжать раскрутку.

Результат: Процесс немедленно убивается через `std::terminate()`. Никакие `catch` блоки не срабатывают, деструкторы остальных объектов не вызываются.

Пример кода «смертника»:

```cpp
struct Bomb {
    ~Bomb() {
        // Попытка выбросить исключение из деструктора
        throw std::runtime_error("Boom in dtor");
    }
};

int main() {
    try {
        Bomb b;
        // 1. Бросаем первичное исключение
        throw std::logic_error("Primary error");

        // 2. Начинается unwinding. Вызывается ~Bomb().
        // 3. Из ~Bomb вылетает "Boom".
        // 4. ДВА исключения -> std::terminate().
    } catch (...) {
        std::cout << "Never printed\n";
    }
}
```

Если убрать `throw "Primary error"`, программа все равно упадет (из-за `noexcept` по умолчанию на деструкторе), но уже по причине нарушения спецификации `noexcept`, а не из-за Double Fault. Если же явно пометить деструктор `noexcept(false)`, то одиночное исключение сработает, но двойное все равно убьет процесс.

Проблема очистки ресурсов (на примере std::ofstream)

Классический пример конфликта RAII и обработки ошибок — закрытие файла.

```cpp
class FileWriter {
    std::ofstream file;
public:
    ~FileWriter() {
        file.close(); // Может вернуть ошибку или кинуть исключение (если
        ↪ включено)
    }
};
```

Операция `close()` включает сброс буферов на диск (flush). Если диск переполнен, `close()` завершится с ошибкой.

- Если деструктор игнорирует ошибку (глотает исключение), пользователь теряет данные, не узнав об этом.
- Если деструктор выбрасывает исключение, программа рискует упасть с `std::terminate`, если `FileWriter` уничтожался в процессе обработки другой ошибки.

Решение: Предоставить явный метод `close()`, а в деструкторе оставить «аварийную» логику.

```
void manual_close() {
    file.close();
    if (file.fail()) throw std::ios_base::failure("Write failed");
}

~FileWriter() {
    try {
        if (file.is_open()) file.close();
    } catch (...) {
        // Логируем, но НЕ бросаем дальше
        std::cerr << "Error closing file in dtor\n";
    }
}
```

Это компромисс: ответственный пользователь вызывает `manual_close()` и обрабатывает ошибки. Забывчивый пользователь полагается на деструктор, который гарантирует отсутствие утечек дескрипторов, но может "проглотить" ошибку записи.

Проверка активных исключений: std::uncaught_exceptions

Иногда действительно необходимо выполнить опасную операцию в деструкторе, но только если это безопасно (т.е. если мы не находимся в процессе раскрутки стека).

Для этого существует функция `std::uncaught_exceptions()` (обратите внимание на множественное число, C++17). Она возвращает количество активных исключений в текущем потоке.

> **На заметку**
>
> В C++98 существовала функция `std::uncaught_exception()` (единственное число), возвращавшая `bool`. Она была признана архитектурно ошибочной и удалена в C++20, так как не позволяла корректно работать во вложенных сценариях (например, когда одно исключение уже обрабатывается, и внутри `catch` бросается новое).

Паттерн безопасного выброса из деструктора (используется крайне редко, например, в транзакционной памяти или сложных базах данных):

```
class Transaction {
public:
    ~Transaction() noexcept(false) { // Разрешаем исключения
        if (std::uncaught_exceptions() == 0) {
            // Стек спокоен, можно кидать исключения.
            // Например, коммит транзакции, который может упасть.
            commit_or_throw();
        } else {
            // Мы уже летим из-за ошибки.
            // Кидать новое нельзя -> terminate.
            // Поэтому делаем только безопасный откат.
            rollback_silent();
        }
    }
```

```
15 };
```

Резюме раздела

1. **Stack Unwinding** — механизм автоматического вызова деструкторов при возникновении исключения. Это основа RAII.
2. **Деструкторы по умолчанию noexcept**. Выброс исключения из деструктора при наличии другого активного исключения приводит к `std::terminate`.
3. Не стройте логику программы на исключениях в деструкторах. Деструктор должен заниматься освобождением ресурсов, которое (в идеале) не должно фейлиться.
4. Если необходимо сообщить об ошибке при закрытии ресурса, используйте отдельный метод (например, `close()`) и вызывайте его явно перед разрушением объекта.

Глава 3

Продвинутая работа с исключениями: Транспортировка и Slicing

Работа с исключениями в C++ не ограничивается простыми блоками `try-catch`. При построении сложных архитектур (например, асинхронных очередей задач или плагинных систем) возникает необходимость сохранять состояние ошибки, передавать его между потоками и корректно обрабатывать полиморфные иерархии исключений.

Физическое расположение исключений

Когда выполняется инструкция `throw`, среда выполнения (Runtime) должна аллоцировать память под объект исключения. Возникает вопрос: где именно живет этот объект?

1. **Не на стеке.** Стек раскручивается (unwinding) в процессе поиска обработчика, поэтому объект исключения не может быть локальной переменной.
2. **Не в статической памяти.** Исключения могут быть рекурсивными или возникать в нескольких потоках одновременно.

Фактически, компиляторы (в соответствии с ABI, например, Itanium C++ ABI) выделяют память в отдельной области кучи (heap). Это накладывает специфические требования:

- Операция `throw` может потребовать динамической аллокации.
- Если памяти нет (бросается `std::bad_alloc`), Runtime использует заранее зарезервированный "аварийный буфер", чтобы гарантировать возможность сообщить об ошибке нехватки памяти.

Время жизни этого объекта управляется Runtime. Он существует до тех пор, пока последний обработчик `catch` не завершит свою работу с ним.

Проблема срезки (Object Slicing)

Одной из самых коварных ошибок в C++ является перехват исключений по значению, а не по ссылке. Это приводит к потере полиморфизма и данных, известной как **Object Slicing** (срезка объекта).

Рассмотрим иерархию ошибок:

```cpp
struct ErrorBase {
    virtual const char* what() const { return "Base Error"; }
    virtual ~ErrorBase() = default;
};

struct DatabaseError : ErrorBase {
    int errorCode;
    DatabaseError(int code) : errorCode(code) {}

    const char* what() const override { return "DB Error"; }
};
```

Если мы выбросим `DatabaseError`, но поймаем его как `ErrorBase` (по значению), произойдет следующее:

```cpp
void unsafe_handler() {
    try {
        throw DatabaseError(505);
    } catch (ErrorBase e) { // ОШИБКА: Перехват по значению!
        // 1. Создается новый объект типа ErrorBase.
        // 2. Вызывается копирующий конструктор ErrorBase(const ErrorBase&).
        // 3. Поля DatabaseError (errorCode) отбрасываются.
        // 4. vptr теперь указывает на таблицу виртуальных функций Base.

        std::cout << e.what(); // Выведет "Base Error", а не "DB Error"
    }
}
```

> **Механика Slicing**
>
> При копировании объекта-наследника в переменную базового типа происходит физическое копирование только той части памяти, которая относится к базовому классу. Дополнительные поля наследника игнорируются. Указатель на таблицу виртуальных функций (`vptr`) инициализируется значением для базового класса, полностью стирая полиморфное поведение.

> **Важно!**
>
> Всегда перехватывайте исключения по константной ссылке: `catch (const ErrorBase& e)`. Это предотвращает копирование и сохраняет полиморфизм (вызов `e.what()` вернет "DB Error").

Механика повторного выброса (Rethrow)

Частый сценарий: перехватить ошибку, залогировать её и пробросить дальше для обработки на уровне выше. Здесь существует критическая разница между `throw e;` и `throw;`.

Ошибочный проброс (throw e)

```
try {
    // ... код ...
} catch (const ErrorBase& e) {
    log_error(e);
    throw e; // ОШИБКА: Повторная срезка!
}
```

Даже если `e` — это ссылка на `DatabaseError`, инструкция `throw e` создает *новый* объект исключения. Тип этого нового объекта определяется статическим типом переменной `e` (то есть `ErrorBase`). Мы снова теряем информацию о том, что это была ошибка базы данных.

Корректный проброс (throw)

```
try {
    // ... код ...
} catch (const ErrorBase& e) {
    log_error(e);
    throw; // КОРРЕКТНО: Проброс текущего активного исключения
}
```

Инструкция `throw;` (без аргументов) сообщает Runtime: «Возьми тот самый объект исключения, который сейчас обрабатывается (со всеми его полиморфными свойствами), и запусти процесс раскрутки стека дальше». Копирования не происходит, тип сохраняется.

Транспортировка исключений между потоками

Исключения привязаны к стеку текущего потока (Thread Local). Вы не можете выбросить исключение в одном потоке и автоматически поймать его в другом (например, в `main`).

Для решения этой задачи в C++11 был введен механизм захвата и транспортировки исключений: `std::exception_ptr`.

Это «умный указатель» (аналог `std::shared_ptr`) для объектов исключений. Он копирует (или продлевает жизнь) объекту исключения, позволяя сохранить его в переменную и передать в другой контекст.

Основные примитивы

1. `std::current_exception()` — вызывается внутри блока `catch`. Возвращает `std::exception_p` указывающий на текущую ошибку. Если исключений нет, возвращает `nullptr`.
2. `std::rethrow_exception(ptr)` — принимает указатель и выбрасывает исключение заново.

Реализация паттерна Worker-Result

Рассмотрим, как это работает на примере самодельного аналога `std::future`:

```cpp
#include <exception>
#include <thread>
#include <iostream>

std::exception_ptr globalException = nullptr;

void worker() {
    try {
        // Имитация работы
        throw std::runtime_error("Failure in worker thread");
    } catch (...) {
        // 1. Не знаем тип исключения, но можем его захватить
        globalException = std::current_exception();
        // Теперь объект исключения живет в куче и удерживается указателем
    }
}

int main() {
    std::thread t(worker);
    t.join();

    if (globalException) {
        try {
            // 2. Пробрасываем захваченное исключение в текущем потоке
            std::rethrow_exception(globalException);
        } catch (const std::exception& e) {
            std::cout << "Caught from worker: " << e.what() << "\n";
        }
    }
    return 0;
}
```

На заметку

`std::exception_ptr` корректно работает с любыми типами исключений, даже если они не наследники `std::exception` (например, `throw 42;`). Однако, чтобы обработать их после `rethrow_exception`, вам все равно понадобится соответствующий `catch`.

Резюме раздела

- Объекты исключений живут в специальной области памяти (не стек), их время жизни управляется Runtime.
- Перехват по значению (`catch (Base e)`) разрушает полиморфизм (Object Slicing). Всегда используйте ссылки.
- Для проброса исключения используйте `throw;` (без аргументов), чтобы избежать повторной срезки.
- Для передачи ошибок между потоками используйте пару `std::current_exception` и `std::rethrow_exception`.