Обработка на грешки

титуляр на курса: д-р Тодор Цонков (ttsonkov@gmail.com)

От какво ще се състои настоящата лекция?

Изключения. Обработка на изключения. Йерархия на изключенията и примери. Изключения в конструктори и деструктори. Нива на exception safety. Модерен вариант - std::expected(C++23). Пример с клас Student.

Какво представляват изключенията?

Изключенията са механизъм за обработка на грешки при необичайни ситуации, които:
не са част от нормалния контролен поток
не могат или не е удобно да се обработят локално.
Изискват прекъсване на текущото изпълнение и прехвърляне на контрола нагоре по call stack-а

Примери:
1) неуспешно заделяне на памет
2) невалиден вход
3) файл не може да бъде отворен
4) логическа грешка (out-of-range, invalid state)

Синтаксис

```
throw
```

1)Хвърля обект (не тип!)
2)Обектът се копира или премества в специална exception област
3)Типът на обекта определя кой catch блок ще го улови

пример: `throw std::runtime_error("File not found");`

Какво представляват изключенията?

try

Ограничава блок код, в който може да възникне изключение
Сам по себе си не обработва нищо – само дефинира защитена зона

```
try {
    risky_operation();
}

catch{
}
```

1)Улавя изключения по тип
2)Може да има няколко catch блока
3)Първият съвпадащ тип се изпълнява

Синтаксис

```
catch (const std::invalid_argument& e) { }
catch (const std::exception& e) { }
catch (...) { } // всичко останало
```

⚠️ Редът е критичен – от най-специфичен към най-общ.

Когато се хвърли изключение:

1)Текущата функция прекъсва
2)Започва размотаване на стека (stack unwinding)
3)Всички локални обекти се унищожават
4)Търси се първият catch, който може да улови изключението

Изключения - улавяне по стойност vs reference

Добра практика

Лоша практика!
catch (std::exception e) // slicing!
Добро!
catch (const std::exception& e)

1)избягва object slicing
2)не прави излишни копия
3)позволява полиморфизъм

catch (...)

Catch-all: catch (...)
Улавя всяко изключение
Полезно за:
boundary слоеве (main, thread entry, framework hooks)
логване и cleanup

❌ Лоша практика:

1)да се използва без повторно хвърляне

2) да „поглъща“ грешки

Кога да ползваме изключения?

1)Грешки, които не са част от нормалната логика
2)конструктори (нямат return value)
3)нарушения на класа
4)boundary между слоеве (I/O, OS, DB).

Кога да не ползваме изключения?

Неподходящо е в следните ситуации:

1)нормална последователност на кода
2)очаквани резултати (например find() в контейнер)
3)performance-critical inner loops

Причини:
1) throw е много скъпа операция (stack unwinding, RTTI, metadata)
2)времето за обработка е непредсказуемо нарушават worst-case execution time (WCET)

Rethrow и std::current_exception

```
try {
// ...
} catch(...) {
    auto eptr = std::current_exception();
    // прехвърли, запази или логни, после повторно хвърли
    std::rethrow_exception(eptr);
}
```

Йерархия на изключенията

`std::exception` (базов)

`std::logic_error` (грешки в логиката на програмата)

Примери: `std::invalid_argument`, `std::out_of_range`

`std::runtime_error` (външни грешки по време на изпълнение)

Примери: `std::overflow_error`, `std::underflow_error`

Добра практика: улавяйте специфични типове, а не “...” или базов клас, освен когато е нужно

Собствени exception класове

```
class MyError : public std::runtime_error {
public:
  explicit MyError(const std::string& msg) :
std::runtime_error(msg) {}
};

int divide(int a, int b) {
    if (b == 0) {
        throw MyError("Division by zero is not allowed");
    }
    return a / b;
}

int main() {
    try {
        std::cout << divide(10, 0) << '\n';
    }
    catch (const MyError& e) {
        std::cerr << "MyError caught: " << e.what() <<
'\n';
    }
    catch (const std::exception& e) {
        std::cerr << "std::exception caught: " <<
        e.what() << '\n';
    }
}
```

Изключение в конструктора

Конструктор може да хвърли изключение, когато обектът не може да бъде валидно конструиран.

Ако конструктор хвърли:

самият обект не се счита за създаден
деструкторът му НЕ се извиква
деструкторите на вече конструираните член-данни се извикват автоматично

`std::runtime_error` се хвърля
`file_` (`ifstream`) се унищожава автоматично
няма `resource leak` благодарение на `RAII`

📌 Добра практика: ако обектът не може да съществува в невалидно състояние – хвърляй в конструктора.

Хвърляне в конструктор

```cpp
#include <fstream>
#include <stdexcept>

class FileReader {
std::ifstream file_; // RAII

public:

explicit FileReader(const std::string& path) {
  file_.open(path);
  if (!file_) {
      throw std::runtime_error("Cannot open file: " +
        path);
      }
}

  std::string readLine() {
      std::string line;
      std::getline(file_, line);
      return line;
   }
};
```

Изключения в деструктора

Деструкторите не трябва да хвърлят изключения.

Ако деструктор хвърли по време на stack unwinding (докато вече има активно изключение) → std::terminate().

В modern C++ деструкторите са noexcept(true) по подразбиране.

Пример на хвърляне в деструктор

Хвърляне в деструктор

```
//ЛОШ ПРИМЕР!
class BadLogger {
public:
~BadLogger() {
    if (!flush()) {
        throw std::runtime_error("Flush failed");
    }
  }
};

//Правилно!
class SafeLogger {
public:
~SafeLogger() noexcept {
    try {
        flush();
    } catch (const std::exception& e) {
            log_error(e.what()); // логване, но НЕ
хвърляме
       }
 }
};
```

Нива на Exception Safety

No-throw (nothrow) – операцията не хвърля (стойностна гаранция за не-хвърляне).
Характеристики:
1)маркира се с noexcept
2)позволява оптимизации от компилатора
3)критично важно за деструктори, swap, move операции

Strong guarantee (commit-or-rollback) – или операцията завършва успешно, или няма ефект (атомарно поведение).
Гаранция:
или операцията завършва успешно или състоянието на програмата остава напълно непроменено
Поведение, подобно на транзакция.
Изисква:
работа върху временни обекти

Basic guarantee – след изключение данните на обекта са запазени, но състоянието може да е променено (без изтичане на ресурси).

No guarantee – няма обещание. ЛОШ ДИЗАЙН!

Описание на exception safety

1) No-throw guarantee- никога няма да хвърли exception

```
void swap_ints(int& a, int& b) noexcept {
    int tmp = a;
    a = b;
    b = tmp;
}
```

2) Strong exception guarantee -
Ако възникне изключение → състоянието остава непроменено („commit or rollback“).

```
struct Account {
    int balance;
};
```

//Голяма част от STL контейнери я гарантират.

Пример

```
void transfer(Account& from, Account& to, int amount) {
    Account from_backup = from;
    Account to_backup = to;

    try {
        if (from.balance < amount) throw
std::runtime_error("insufficient");
        from.balance -= amount;

        // Тук може да хвърли (примерно логване, мрежа, база)
        if (amount > 10000) throw
std::runtime_error("reporting error");

        to.balance += amount;
    }
    catch (...) {
        // rollback – strong exception guarantee
        from = from_backup;
        to = to_backup;
        throw;
    }
}
```

Обобщение

3) Basic exception guarantee
Ако възникне изключение:
програмата остава в валидно, но не дефинирано състояние
няма resource leaks

```
#include <vector>
void add_element_basic
(std::vector<int>& v, int x) {
    v.push_back(x); // ако хвърли → v е валиден, но размерът може да е променен
}
```

4) **No exception safety** (Никаква гаранция)

Пример

```
#include <iostream>

struct Bad {
    int* data;

    Bad() {
        data = new int[10]; (1)
        throw std::runtime_error("Boom");//(2)leak
    }

    ~Bad() {
        delete[] data; // (3) никога няма да се извика
    }
};
```

Какво се случва?

1) Заделена памет
2) Изключение в конструктора

3) Деструкторът не се извиква → memory leak

4) Състоянието може да е счупено и да не се възстанови

Error codes.

Функцията връща стойност, която указва успех или тип грешка
Класически подход в C и системното програмиране

Обикновено:
int / enum
bool + out параметър

Предимства

Предсказуем контролен поток, без runtime cost от stack unwinding
Подходящи за:
Embedded системи, Game engines

Недостатъци

```
ErrorCode process_file(const std::string& path)
{
    ErrorCode err = open_file(path);
    if (err != OK) return err;

    err = read_header();
    if (err != OK) return err;

    err = read_data();
    if (err != OK) return err;

    err = write_data();
    if (err != OK) return err;

    return OK;
} //Смесва се бизнес логиката с грешките, може да се пропусне грешка или да се изтрие погрешка
```

std::expected

Какво е std::expected<T, E>?
Въведен в C++ 23 - модерна версия.

Тип, който съдържа:
валидна стойност T или грешка E

Compile-time гаранция:

„Тази функция може да се провали“

std::expected<int, ErrorCode> - първата част е при успех, втората при грешка.

```cpp
std::expected<Data, ErrorCode> process_file() {
    auto file = open_file();
    if (!file) return std::unexpected(file.error());

    auto header = read_header(*file);
    if (!header) return std::unexpected(header.error());

    return read_data(*header);
}

// поддържа функционален стил - монади
auto res =
    open_file("a.txt")
    .and_then(read_header)
    .and_then(parse_data)
    .transform(to_json);
```

Обобщение

Error codes → работят, но са примитивни
Exceptions → мощни, но скрити и скъпи

std::expected = най-доброто от двете:
1)Явни грешки
2)Без runtime overhead
3)Модерен API

❌ Не е идеален, когато:

Грешката е наистина изключителна

Искаме автоматично propagation през много нива на кода

Пример с класа Student

```
Student::Student(std::string name, std::vector<int>
grades)
    : name_(std::move(name)), grades_(std::move(grades))
{}

const std::string& Student::name() const noexcept {
    return name_;
}

const std::vector<int>& Student::grades() const noexcept
{
    return grades_;
}

std::expected<double, ErrorCode>
Student::average_grade() const {
    if (grades_.empty())
        return std::unexpected(ErrorCode::NoGrades);

    double sum = std::accumulate(grades_.begin(),
grades_.end(), 0.0);
    return sum / grades_.size();
}
```

Въпроси?

Благодаря за вниманието!