Обработка на грешки

Изключения. Обработка на изключения. Йерархия на изключенията и примери. Изключения в конструктори и деструктори. Нива на exception safety. Модерен вариант - std::expected(C++23). std::optional(C++17). Примери.

д-р Тодор Цонков
todort@uni-sofia.bg

Какво е представляват изключенията?

Изключенията са механизъм за обработка на грешки при необичайни ситуации, които:
1)не са част от нормалния контролен поток
2)не могат или не е удобно да се обработят локално.
3)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)

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

Изключения - продължение

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

```
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 - по стойност и по референция

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-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

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

Пример:

```
class MyException : public std::exception {
private:
    std::string message;
public:
explicit MyException(const std::string& msg)
: message(msg) {}
    const char* what() const noexcept {
        return message.c_str();
    }
};
```

Йерархия на изключенията

Пример за използване в main()

```
int main() {
    try {
        throw MyException("Нещо се
объркa!");
    } catch (const MyException& e) {

        std::print("Caught MyException:
{}\n", e.what());

    } catch (const std::exception& e) {
        std::print("Caught std::exception:
{}\n", e.what());
    }
}
```

```
int getIntValueFromDatabase(Database* d,
std::string table, std::string key)
{
    try
    {
        return d->getIntValue(table, key); // throws
int exception on failure
    }
    catch (int exception)
    {
        // Write an error to some global logfile
g_log.logError("getIntValueFromDatabase
failed");
        throw exception;
    }
}
```

Изключения 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.

Следва пример в следващия слайд:

Изключения в конструктора

```
#include <string>
#include <stdexcept>
#include <print>

class BankAccount {
    std::string owner_;
    double balance_;
public:
    BankAccount(const std::string&
owner, double balance)
        : owner_(owner), balance_(balance)
    {
        if (owner.empty()) {
 throw std::invalid_argument("Owner name
cannot be empty");
        }
        if (balance < 0.0) {
            throw std::invalid_argument("Initial
balance cannot be negative");
        }

        std::print("Account created for {} with
balance {}\n", owner_, balance_);
    }

    double balance() const {
        return balance_;
    }
};
```

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.

Ако хвърлиш изключение от деструктор →
това автоматично води до std::terminate().

По време на unwinding runtime-ът:
1)разрушава обекти
2)освобождава ресурси
3)възстановява stack frame

Ако деструктор хвърли:

1)системата вече е в нестабилно състояние
2)няма ясен механизъм кое изключение да оцелее
3)няма безопасен recovery

Затова стандартът казва:
Ако по време на stack unwinding бъде хвърлено второ изключение → terminate.

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

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

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

No guarantee — няма обещание. ЛОШ ДИЗАЙН!

Примери за 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“).

```
class IntCollection {
  std::vector<int> data;
public:
  void add_two(int a, int b) {
    // 1. Работим върху копие
    std::vector<int> temp = data;
    temp.push_back(a);
    temp.push_back(b);
    // 2. Commit (noexcept операция)
    data.swap(temp);
  }
};
```

Примери за exception safety

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

```
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
    }
};
//memory leak - заделената памет няма да се възстанови
```

Error codes

Error codes са стойности, които функция връща, за да сигнализира дали операцията е успешна или е възникнала грешка.

Това е класическият C-подход, който в C++ продължава да съществува паралелно с exceptions.

Функцията връща:
0 или специална стойност → успех
различна стойност → конкретна грешка

```
#include <print>
int divide(int a, int b, int& result)
{
    if (b == 0)
        return 1; // error code
    result = a / b;
    return 0; // success
}

int main()
{
    int result{};
    int error = divide(10, 0, result);

    if (error != 0)
        std::print("Error code: {}\n", error);
}
```

Error codes

Предимства:

Предсказуем контролен поток, без runtime cost от stack unwinding

Подходящи за:

Embedded системи, Game engines

Недостатъци:

1)Смесва се бизнес логиката с грешките,

2)може да се пропусне грешка или да се изтрие погрешка възможен проблем.

```
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;
}
```

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.

```
std::expected<int, std::string> divide(int a, int b)
{
    if (b == 0)
        return std::unexpected("Division by zero");

    return a / b;
}

int main()
{
    auto result = divide(10, 2);
    if (result)
        std::print("Result = {}\n", *result);
    else
        std::print("Error: {}\n", result.error());
     return 0;
}
```

std::expected - пример

```
enum class ParseError
{
  InvalidFormat, OutOfRange
};
std::expected<int, ParseError>
parse_int(std::string_view sv)
{
  int value{};
  auto [ptr, ec] = std::from_chars(sv.data(),
sv.data() + sv.size(), value);

 if (ec == std::errc::invalid_argument)
    return
std::unexpected(ParseError::InvalidFormat);
  if (ec == std::errc::result_out_of_range)
    return
std::unexpected(ParseError::OutOfRange);
  return value;
}

int main()
{
  auto r = parse_int("123");
  if (r)
    std::print("Parsed: {}\n", *r);
  else
    std::print("Parse failed\n");
  return 0;
}
```

std::optional

std::optional е шаблонен клас в C++17, който представя стойност, която може да съществува или да липсва.

Идеята е:
„Имам обект от тип T, но понякога няма смислена стойност.“ std::optional<T>

Той заменя:
1)nullptr проверки
2)магически стойности (например -1)
3)ръчни флагове bool hasValue

```
std::optional<int> find_even(int x)
{
    if (x % 2 == 0)
        return x;
    return std::nullopt;
}

int main()
{
    auto result = find_even(10);

    if (result)
        std::print("Value: {}\n", *result);
    else
        std::print("No value\n");
}
```

std::optional

```
#include <optional>
#include <string>
#include <print>

class Student
{
    std::string name;
    std::optional<std::string> middle_name;
public:
    Student(std::string n, std::optional<std::string>
m = std::nullopt)
        : name(std::move(n)),
middle_name(std::move(m)) {}
    void print() const
    {
        if (middle_name)
            std::print("{} {} \n", name,
*middle_name);
        else
            std::print("{}\n", name);
    }
};
```

std::optional архитектурна дефиниция

✅ Подходящо за:

1)функции, които може да нямат резултат
2)optional полета в обект
3)lazy initialization
4)частично конструирани обекти

❌ Не е подходящо за:

грешки → използвай std::expected
колекции → използвай празен контейнер
ownership → използвай std::unique_ptr

Формална дефиниция:

std::optional<T> е обектна обвивка, която моделира nullable value semantics, без да използва указатели или специални стойности, като осигурява безопасен и типово коректен механизъм за представяне на липсваща стойност.

std::optional<T> означава:
„T е част от модела, но понякога отсъства.“
Не означава:
„Това е обект, който живее другаде.“

Сравнение с Java и C#

В C++ exceptions са гъвкави и могат да хвърлят всякакви типове, но компилаторът не следи тяхното обработване, което може да доведе до пропуснати грешки.

В Java има строг контрол чрез checked exceptions, което гарантира обработка, но води до повече boilerplate код и runtime overhead.

В C# exceptions са типизирани и лесни за използване, но няма checked exceptions, така че програмата може да пропусне обработка, а всички exceptions са на heap.

```
public class Example {
   // Функция, която хвърля checked exception
   public static int divide(int a, int b) throws
ArithmeticException {
      if (b == 0) {
         throw new ArithmeticException("Division by
zero!");
       }
      return a / b;
   }
   public static void main(String[] args) {
      try {
         int result = divide(10, 0); // ще хвърли exception
         System.out.println("Result: " + result);
      } catch (ArithmeticException e) {
         System.out.println("Caught exception: " +
e.getMessage());
      } finally {
         System.out.println("Cleanup if needed");
      }
   }
}
```

Въпроси?

Благодаря за вниманието!

Допълнителни материали: learncpp.com глава 27