Винятки як системний підхід до обробки помилок

FACULTY OF APPLIED MATHEMATICS AND INFORMATICS
LVIV UNIVERSITY

План на сьогодні

1. Обробка винятків. Оператори throw, try і catch
2. Винятки, Функції і розгортання стеку
3. Неперехоплені винятки і обробники catch-all
4. Класи-винятки і Спадкування
5. Повторна генерація винятків, функціональний try-блок
6. Ієрархія класів стандартних винятків
7. Нюанси та недоліки використання винятків

FACULTY OF APPLIED MATHEMATICS AND INFORMATICS
LVIV UNIVERSITY

Обробка винятків. Оператори throw, try і catch.

FACULTY OF APPLIED MATHEMATICS AND INFORMATICS
LVIV UNIVERSITY

Що таке виняток? Навіщо вони потрібні?

Виняток — це спеціальна подія, яка сигналізує про помилкову або нетипову ситуацію під час виконання програми, і дозволяє передати керування до обробника помилок.

Проблеми з кодами повернення:

- Значення помилок неочевидні (наприклад, -1, 0.0, false)
- Неможливо одночасно повернути результат і статус помилки
- Потрібно вручну перевіряти кожен виклик функції
- Неможливо використовувати return у конструкторах
- Ускладнюється структура програми, знижується читабельність

```
double divide(int a, int b) {
    return static_cast<double>(a)/b;
}
```

```
double divide(int a, int b, bool &success) {
    if (b == 0) {
        success = false;
        return 0.0;
    }
    success = true;
    return static_cast<double>(a)/b;
}
```

Як допомагають винятки?

- Винятки мови C++ забезпечують могутній і гнучкий засіб для коректного виходу з нетипових ситуацій.
- Винятки генеруються в результаті нестандартних ситуацій, що виникають при виконанні програми. З їх допомогою можна передавати керування з однієї частини програми в іншу.

Використання винятків C++ має такі переваги:

- Відділяють логіку обробки помилок від основного коду;
- Дають можливість обробляти помилки лише там, де це дійсно потрібно;
- Працюють у конструкторах і складних ініціалізаціях;
- Дозволяють писати більш безпечний та чистий код.

Генерація винятків.

Для сигналізування про помилку використовується оператор throw. Це називається генерацією винятку.

Приклад з життя:
Коли у баскетболі відбувається фол — арбітр свистить і гра зупиняється. Після штрафного кидка — гра продовжується. Подібно до цього, в C++ виконується сигнал про помилку — програма переривається звичний потік виконання, поки виняток не буде оброблений.

```
throw -1; // int
throw ENUM_INVALID_INDEX; // enum
throw "Negative number"; // C-style рядок
throw dX; // значення типу double
throw MyException("Fatal Error"); // об'єкт класу винятку
```

```
double divide(int a, int b) {
    if (b == 0) {
        throw "Error: Division by zero!";
    }
    return static_cast<double>(a) / b;
}
```

Перехоплення винятків

У C++ для перехоплення винятків використовується блок try:

- Блок try містить потенційно небезпечний код
- Якщо всередині блоку try виникає виняток (оператор throw), програма негайно припиняє виконання цього блоку
- Система починає пошук відповідного обробника (catch)

Приклад з життя:
Як у баскетболі, коли гравець порушує правила — арбітр свистить. Це зупиняє гру. Аналогічно, throw зупиняє виконання в блоці try. Далі — наступний етап: реакція на ситуацію, тобто обробка винятку.

```
try {
    // Код, який може згенерувати виняток
    throw -1;
}
```

```
try {
    int a = 10, b = 0;
    double result = divide(a, b);
    cout << "Result: " << result << endl;
}
```

Обробка винятків

Обробка винятків у C++ виконується в блоці catch, який розміщується відразу після try.

- catch приймає параметр винятку — як у функції
- Винятки фундаментальних типів (int, double тощо) можна перехоплювати за значенням
- Винятки класів перехоплюють за const посиланням, щоб уникнути копіювання
- Якщо параметр не використовується, його ім’я можна опустити

Приклад з життя:
Як у баскетболі: поки не буде виконано штрафний кидок — гра не продовжиться.
Так само в C++: поки виняток не буде оброблений у catch — програма не відновить своє виконання.

```
try {
    throw -1;
}
catch (int a) {
    cerr << "We caught an int exception with value " << a << '\n';
}
```

```
int main() {
    try {
        int a = 10, b = 0;
        double result = divide(a, b);
        cout << "Result: " << result << endl;
    }
    catch (const char* msg) {
        cout << msg << endl;
    }

    return 0;
}
```

Використання throw, try і catch разом

Після генерації винятку у блоці **try** оператором **throw**, здійснюється пошук **catch** блоку з аргументом, що співпадає з типом винятку

- ❏ throw генерує виняток типу int зі значенням -1
- ❏ try виявляє виняток
- ❏ catch (int a) обробляє його і виводить повідомлення
- ❏ Після обробки виконання програми продовжується

```
int main() {
    try {
        throw -1;
    }
    catch (int a) {
        cerr << "We caught an int exception with value: " << a << '\n';
    }
    catch (double) {
        cerr << "We caught an exception of type double" << '\n';
    }
    catch (const string& str) {
        cerr << "We caught an exception of type string" << '\n';
    }
    cout << "Continuing our way!\n";
    return 0;
}
```

Як працюють винятки

Основні правила:

- ❏ Після throw виконання негайно переходить до найближчого try
- ❏ Якщо є відповідний catch, виняток обробляється
- ❏ Якщо обробника немає — пошук триває далі вгору по стеку
- ❏ Якщо жоден обробник не знайдено — програма аварійно завершується

Що зазвичай робить catch:

- ❏ Виводить повідомлення про помилку
- ❏ Повертає код або значення назад
- ❏ Генерує новий виняток

```
double a;
cin >> a;
try {
    if (a < 0.0)
        throw "Can not take sqrt of negative number";
    cout << "The sqrt of " << a << " is " << sqrt(a) << '\n';
}
catch (const char* e) {
    cerr << "Error: " << e << '\n';
}
```

*Увага: **компілятор не виконує неявні перетворення типів у `catch`.** Наприклад, `throw 'a'` не буде оброблено `catch(int)`.*

Винятки, Функції і розгортання стеку.

FACULTY OF APPLIED MATHEMATICS AND INFORMATICS
LVIV UNIVERSITY

Генерація винятків поза блоком try

Оператор throw не обов’язково має бути всередині try.
C++ підтримує розгортання стеку — механізм передачі винятку назад по стеку викликів, поки не буде знайдено відповідний catch.

```
double mySqrt(double a) {
    if (a < 0.0)
        throw "Can not take sqrt of negative number";
    return sqrt(a);
}

int main() {
    double a;
    cin >> a;
    try {
        double d = mySqrt(a);
        cout << "The sqrt of " << a << " is " << d << '\n';
    }
    catch (const char* e) {
        cerr << "Error: " << e << '\n';
    }
}
```

- В процесі виклику функції в стек потрапляє інформація про точку повернення, всі аргументи функції і локальні змінні функції.
- Після завершення функції – керування переходить в точку виклику і всі змінні функції зі стеку знищуються (деструктори).
- Якщо виконання функції припиняється генерацією винятку, то керування передається не у безпосередньо викликаючу функцію, а в точку try-блоку. Вся інформація зі стеку буде коректно знищена (деструктори). Цей процес наз. розгортанням стеку.

Генерація винятків поза блоком try

Як це працює:

- throw виникає в mySqrt(), поза межами блоку try
- mySqrt() припиняє виконання
- Відбувається розгортання стеку: програма повертається в main(), де є блок try
- Виняток обробляється у відповідному catch

Переваги:

- Функція повідомляє про помилку, але не обробляє її самостійно
- Обробка делегується на вищий рівень — гнучкість і незалежність логіки обробки
- Різні програми можуть реагувати на виняток по-різному (вивести повідомлення, показати діалог, завершити роботу тощо)

```
double mySqrt(double a) {
    if (a < 0.0)
        throw "Can not take sqrt of negative number";
    return sqrt(a);
}

int main() {
    double a;
    cin >> a;
    try {
        double d = mySqrt(a);
        cout << "The sqrt of " << a << " is " << d << '\n';
    }
    catch (const char* e) {
        cerr << "Error: " << e << '\n';
    }
}
```

Неперехоплені винятки і обробники catch-all

FACULTY OF APPLIED MATHEMATICS AND INFORMATICS
LVIV UNIVERSITY

Неперехоплені винятки

Що станеться, якщо виняток ніхто не обробить?

Функція може згенерувати виняток (через throw) і передати його на обробку вище, але якщо жоден рівень не містить відповідного блоку catch, програма аварійно завершується.

Що відбувається при введенні -5:

- mySqrt(-5) генерує виняток типу const char*
- Стек розгортається, виклик повертається в main()
- У main() немає обробки винятку
- Програма завершується з неперехопленим винятком

Поведінка залежить від ОС:

- Повідомлення в консолі
- Діалогове вікно помилки
- Збій виконання

```
double mySqrt(double a) {
    if (a < 0.0)
        throw "Can not take sqrt of negative number";
    return sqrt(a);
}
int main() {
    double a;
    cin >> a;
    // Блок try відсутній
    cout << "The sqrt of " << a << " is " << mySqrt(a)
<< '\n';
}
```

Обробники всіх типів винятків

Функція може згенерувати виняток будь-якого типу, але ми не завжди знаємо її реалізацію. Як бути, якщо тип винятку невідомий?

Використати обробник всіх типів — catch (...)
Такий обробник перехоплює будь-який виняток, незалежно від його типу.

Важливо:

- catch (...) завжди має бути останнім у списку обробників
- Інакше конкретні catch не матимуть шансів спрацювати
- Часто використовується для запобігання аварійному завершенню

```
int main() {
    try {
        throw 7; // виняток типу int
    }
    catch (double a) {
        cout << "We caught a double: " << a << '\n';
    }
    catch (...) {
        cout << "We caught an exception of an
undetermined type!\n";
    }
}
```

Специфікації винятків (throw/noexcept)

Специфікація винятків — це оголошення функції з вказанням, які винятки вона може або не може генерувати.
Як це працює?

- Якщо функція позначена як noexcept (або throw() до C++11), але все ж генерує виняток — програма негайно завершується (у більшості реалізацій).
- Такі функції зобов'язані обробляти винятки самостійно.
- У разі порушення специфікації — розгортання стеку заборонено.

Специфікація винятків (throw(type)) застаріла, і її слід уникати

```
int doSomething() noexcept;        // не генерує винятків
int doSomething() throw(double); // може згенерувати лише double
int doSomething() noexcept(false);    // може згенерувати будь-який виняток
```

Використання noexcept

- Функції переміщення (move-конструктори, move-оператори =) зазвичай повинні бути noexcept, щоб стандартні контейнери (std::vector, std::list) могли ефективно їх використовувати.
- Якщо move не помічено як noexcept, стандартні контейнери можуть копіювати об’єкти замість переміщення, що впливає на продуктивність.

```
class MyClass {
public:
    MyClass(MyClass&& other) noexcept { /* move logic */ }
    MyClass& operator=(MyClass&& other) noexcept { /* move assignment logic */ return *this; }
};
```

Класи-винятки і Спадкування

FACULTY OF APPLIED MATHEMATICS AND INFORMATICS
LVIV UNIVERSITY

Класи-винятки

Проблема з фундаментальними типами:

- Винятки типу int, char* тощо не містять контексту
- Невідомо, звідки саме і чому стався збій
- Неможливо обробити винятки з різних місць по-різному

Одним із способів вирішення цієї проблеми є використання класів-винятків. **Клас-виняток** — це звичайний клас, який генерується в якості винятку

```
class ArrayException {
private:
    string m_error;
public:
    ArrayException(string error) : m_error(error) {}
    const char* getError() { return m_error.c_str(); }
};
```

Класи-винятки

Переваги:

- Можна передати точний опис помилки
- Можна створити окремі класи для різних виняткових ситуацій
- Обробники можуть перехоплювати винятки за типом посилання, а не значення — для уникнення копій

```
class ArrayException {
private:
    string m_error;
public:
    ArrayException(string error) : m_error(error) {}
    const char* getError() { return m_error.c_str(); }
};
```

```
class ArrayInt {
    int m_data[4];
public:
    int& operator[](int index) {
        if (index < 0 || index >= 4)
            throw ArrayException("Invalid index");
        return m_data[index];
    }
};

try {
    int value = array[7];
}
catch (const ArrayException& e) {
    cerr << "Array error: " << e.getError() << '\n';
}
```

Винятки і спадкування

У C++ винятки — це об'єкти

- Отже, можна використовувати успадковані класи як винятки.
- Обробник може ловити як свій тип, так і успадковані від нього.

Правило:

Завжди розміщуйте обробники дочірніх класів перед обробниками батьківських, інакше вони ніколи не виконаються.

```
class Parent {};
class Child : public Parent {};
try {
    throw Child();
}
catch (Parent& p) {
    cerr << "caught Parent";
}
catch (Child& c) {
    cerr << "caught Child";
}
//Результат: caught Parent
```

```
class Parent {};
class Child : public Parent {};
try {
    throw Child();
}
catch (Child& c) {
    cerr << "caught Child";
}
catch (Parent& p) {
    cerr << "caught Parent";
}
//Результат: caught Child
```

Приклад ієрархії класів винятків

```
class MyException {
public:
    virtual void printError() const {
        cout << "MyException: An error occurred!" << endl;
    }
    virtual ~MyException() {}
};

class DivideByZeroException : public MyException {
public:
    void printError() const override {
        cout << "DivideByZeroException: Cannot divide by zero!" << endl;
    }
};

class IndexOutOfBoundsException : public MyException {
public:
    void printError() const override {
        cout << "IndexOutOfBoundsException: Index out of range!" << endl;
    }
};

class BadMemAllocation : public MyException {
public:
    void printError() const override {
        cout << "BadMemAllocation: Memory allocation fails!" << endl;
    }
};

class NotInitialized : public MyException {
public:
    void printError() const override {
        cout << "NotInitialized: Vector is empty!" << endl;
    }
};
```

Повторна генерація винятків

FACULTY OF APPLIED MATHEMATICS AND INFORMATICS
LVIV UNIVERSITY

Відкладена обробка винятку

Припустимо функція хоче перехопити виняток, записати інформацію про нього, але при цьому залишити остаточну обробку на тому хто її викликав — однак вона не має способу повідомити про помилку, оскільки тип повернення не дозволяє цього зробити.

Ми не можемо безпечно повернути "індикатор помилки", бо будь-яке int — допустиме. Можна обробили виняток частково (лог), але caller не знає, що сталася помилка.

Можливі рішення:

- Повторно кинути виняток (throw;)
- Використати об'єкт-обгортку, що містить статус
- Змінити логіку: не перехоплювати виняток у цій функції

```
int getIntValueFromDatabase(Database* db, string table, string key) {
    assert(db);
    try {
        return db->getIntValue(table, key); // може кинути int
    }
    catch (int exception) {
        g_log.logError("getIntValueFromDatabase failed");
        // Але що тепер повернути? Будь-яке int — потенційно
допустиме
    }
}
```

Генерація нового винятку

Припустимо функція перехопила виняток, виконала попередню обробку (наприклад, логування), але не має змоги повернути caller-у інформацію про помилку через тип результату. Рішення — згенерувати новий виняток.

- Виняток int перехоплюється й обробляється локально (логування)
- Далі функція перекидає новий виняток типу char вище в стек викликів
- Цей новий виняток не буде перехоплений у цьому ж блоці catch

```
int getIntValueFromDatabase(Database* db, string table,
string key) {
    assert(db);
    try {
        return db->getIntValue(table, key);
    }
    catch (int exception) {
        g_log.logError("getIntValueFromDatabase failed");
        throw 'q'; // новий виняток типу char
    }
}
```

Функціональний try-блок

Функціональний try-блок

У тілі конструктора ми можемо обробити виняток.
Але якщо виняток виникає в списку ініціалізації — звичайний try вже не допоможе.
В такому випадку використовується функціональний try-блок

- try додається після заголовка конструктора, перед списком ініціалізації
- catch стоїть на тому ж рівні, що і конструктор
- Перехоплює винятки з:
  - списку ініціалізації
  - тіла конструктора

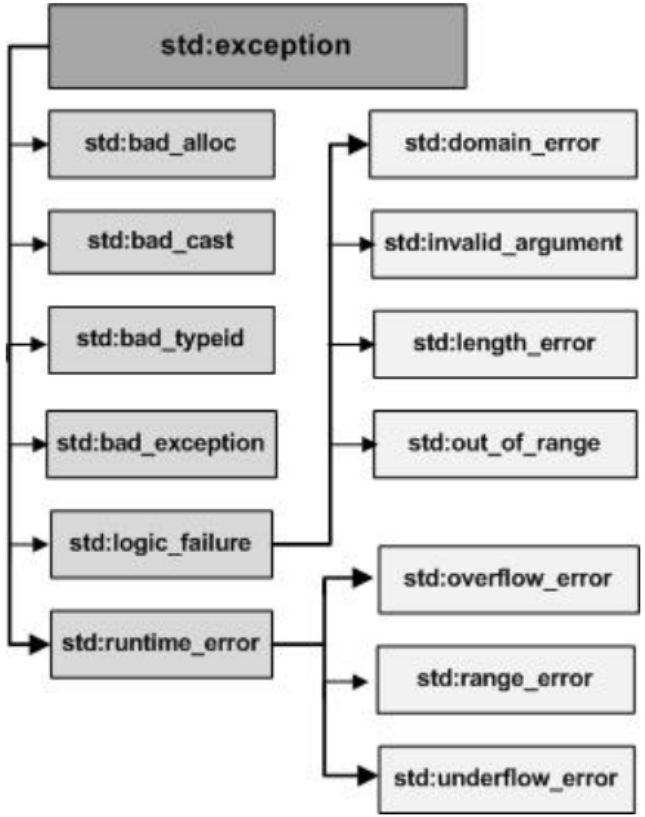
```
//  Конструктор A генерує виняток
class B : public A {
public:
    B(int x) try : A(x) {
        // Тіло конструктора
    }
    catch (...) {
        cerr << "Construction of A failed\n";
        // Якщо нічого не згенерувати — виняток буде повторно кинуто автоматично
    }
};
// Результат: Construction of A failed
```

Ієрархія класів стандартних винятків

FACULTY OF APPLIED MATHEMATICS AND INFORMATICS
LVIV UNIVERSITY

Ієрархія класів стандартних винятків

- **bad_alloc** // помилка виділення пам'яті
- **bad_cast** // неможливість перетворення типу (dynamic_cast)
- **bad_exception** // неочікуваний виняток logic_failure // винятки етапу компіляції
- **domain_error** // математичні винятки, напр, корінь з відємного
- **invalid_argument** // недозволені аргументи
- **lenght_error** // помилки при зміні розміру string, vector, і тп.
- **out_of_range** // доступ до неіснуючого елемента контейнера
- **runtime_error** // винятків етапу виконання overflow_error, underflow_error // математичні помилки переповнення
- **range_error** // помилки розрахунку меж



Ієрархія класів стандартних винятків

Усі винятки, що виникають у Стандартній бібліотеці C++, успадковуються від базового інтерфейсного класу std::exception.

Базовий клас: std::exception
Оголошений у заголовку <exception>, містить віртуальний метод what() для повернення текстового опису помилки.

Безпосередні нащадки std::exception:

- std::bad_alloc — помилка виділення пам’яті
- std::bad_cast — помилка dynamic_cast
- std::bad_exception — згенеровано unexpected-обробником
- std::bad_function_call — виклик об'єкта std::function, що не містить функції
- std::bad_typeid — typeid на нульовому вказівнику
- std::bad_weak_ptr — некоректне створення shared_ptr з weak_ptr
- std::ios_base::failure — помилки потокового вводу/виводу

Ієрархія класів стандартних винятків

std::logic_error:

- std::domain_error — неправильне значення у математичній функції
- std::future_error — помилка при використанні std::future
- std::invalid_argument — передано некоректний аргумент
- std::length_error — перевищено дозволену довжину
- std::out_of_range — доступ поза межами контейнера

std::runtime_error:

- std::overflow_error — арифметичне переповнення
- std::range_error — загальна помилка діапазону
- std::system_error — помилки пов’язані з операційною системою
- std::underflow_error — арифметичне зменшення з точністю втрати

Інтерфейсний клас exception

Усі винятки, які генерує Стандартна бібліотека C++, успадковуються від базового класу exception.

exception — це інтерфейсний клас з віртуальною функцією what(), який дозволяє обробляти всю ієрархію стандартних винятків.

```
#include <exception>
int main() {
    try {
        string s;
        s.resize(-1); // викликає std::bad_alloc
    }
    catch (exception& exception) {
        cerr << "Standard exception: " << exception.what() << '\n';
    }
}
// Результат: Standard exception: string too long
```

```
try {
    // код з використанням стандартної бібліотеки
}
catch (bad_alloc& exception) {
    cerr << "You ran out of memory!\n";
}
catch (exception& exception) {
    cerr << "Standard exception: " << exception.what() << '\n';
}
// Перший catch обробляє std::bad_alloc
// Другий — усі інші виключення стандартної бібліотеки
```

Інтерфейсний клас exception

```
class exception {
 public:
  exception( );
  exception(const char *const&);
  exception(const char *const&, int);
  exception(const exception&);
  exception& operator=(const exception&);
  virtual ~exception( );
  virtual const char * what( ) const;
```

Використання стандартних винятків напряму

Клас std::exception сам по собі не використовується для генерації винятків, але можна використовувати його дочірні класи, якщо вони відповідають вашій ситуації

```
#include <stdexcept>
int main() {
    try {
        throw runtime_error("Bad things happened");
    }
    catch (exception& exception) {
        cerr << "Standard exception: " << exception.what() << '\n';
    }
}
// Результат: Standard exception: Bad things happened
```

```
void testInvalidArgument() {
    try {
        // Спроба передати некоректний аргумент до конструктора
        throw std::invalid_argument("Invalid argument passed to function!");
    } catch (const std::invalid_argument& e) {
        cout << "Caught exception: " << e.what() << endl;
    }
}
```

```
void testBadAlloc() {
    try {
        // Викликаємо виняток, якщо не вдається виділити пам'ять
        throw std::bad_alloc();
    } catch (const std::bad_alloc& e) {
        cout << "Caught exception: " << e.what() << endl;
    }
}
```

Власні класи-винятки на основі exception

Ви можете створювати власні класи-винятки, які успадковують exception. Потрібно перевизначити метод what(), який повертає текст опису помилки. Починаючи з C++11, what() має бути позначений як noexcept.

```
class ArrayException : public exception {
private:
    string m_error;
public:
    ArrayException(string error) : m_error(error) {}
    const char* what() const noexcept {
        return m_error.c_str();
    }
};

class ArrayInt {
    int m_data[4];
public:
    int& operator[](int index) {
        if (index < 0 || index >= 4)
            throw ArrayException("Invalid index");
        return m_data[index];
    }
};
```

```
int main() {
    ArrayInt array;
    try {
        int value = array[7];
    }
    catch (ArrayException& exception) {
        cerr << "An array exception occurred (" << exception.what() << ")\n";
    }
    catch (exception& exception) {
        cerr << "Some other std::exception occurred (" << exception.what() << ")\n";
    }
}
```

Нюанси та недоліки використання винятків

Очищення пам’яті при винятках

Коли виникає виняток, подальші інструкції не виконуються. Ресурси, які були виділені до винятку, можуть не звільнитися.

Приклад: файл не закривається

```
try {
    openFile(filename);
    writeFile(filename, data); // тут виникає виняток
    closeFile(filename);       // не виконається!
}
catch (FileException& exception) {
    cerr << "Failed to write to file: " << exception.what()
<< '\n';
}
```

Рішення:

```
try {
    openFile(filename);
    writeFile(filename, data); // тут виникає виняток
    closeFile(filename);       // не виконається!
}
catch (FileException& exception) {
    closeFile(filename);
    cerr << "Failed to write to file: " << exception.what()
<< '\n';
}
```

Звільнення пам’яті при винятках

Коли виникає виняток, подальші інструкції не виконуються. Ресурси, які були виділені до винятку, можуть не звільнитися.

Приклад: витік пам’яті

```
try {
    Person* alex = new Person(...);
    processPerson(alex); // може
згенерувати виняток
    delete alex;
}
catch (PersonException& exception) {
    cerr << "Failed to process person\n";
    // alex вже недоступний
}
```

Рішення: винести alex за межі try

```
Person* alex = nullptr;
try {
    alex = new Person(...);
    processPerson(alex);
    delete alex;
}
catch (PersonException& exception) {
    delete alex;
    cerr << "Failed to process person\n";
}
```

Винятки і деструктори

Чому не можна генерувати винятки в деструкторах?

- На відміну від конструкторів, у яких виняток може вказувати на невдале створення об'єкта, у деструкторах винятки генерувати не можна.
- Якщо під час розгортання стеку (тобто обробки винятку) деструктор згенерує ще один виняток, компілятор не знатиме, який з них обробляти.
- Це призводить до негайного аварійного завершення програми.

Рекомендація:

- Уникайте генерації винятків у деструкторах.
- Якщо потрібно повідомити про помилку — запишіть її в лог або скористайтеся іншим безпечним механізмом.

Винятки і деструктори

```cpp
class BadDestructor {
public:
    ~BadDestructor() {
        cout << "Destructor called!" << endl;
        throw runtime_error("Exception from destructor!"); // ПОГАНА ПРАКТИКА
    }
};

void func() {
    try {
        BadDestructor obj;
        throw runtime_error("Exception inside function");
    }
    catch (const exception& e) {
        cout << "Caught exception: " << e.what() << endl;
    }
}

int main() {
    try {
        func();
    }
    catch (...) {
        cout << "Unhandled exception!" << endl;
    }
    return 0;
}
```

```cpp
class SafeDestructor {
public:
    ~SafeDestructor() noexcept { // noexcept запобігає поширенню винятку
        try {
            throw runtime_error("Exception in destructor");
        }
        catch (const exception& e) {
            cerr << "Caught exception in destructor: " << e.what() << endl;
        }
    }
};
```

Проблеми з продуктивністю

Чи впливають винятки на швидкодію?

Так, але не завжди (*zero-cost exception handling* в GCC).

- Винятки збільшують розмір виконуваного файлу
- Додають додаткові перевірки під час виконання
- Найбільші витрати виникають під час генерації винятку:
    - запускається розгортання стеку
    - виконується пошук відповідного обробника catch

Коли краще уникати винятків?

- ❏ У системах реального часу (наприклад, вбудовані пристрої, авіація, автоіндустрія) час виконання коду має бути передбачуваним.
- ❏ У критичних секціях коду (напр., драйвери, операційні системи).
- ❏ Обробка винятків може спричинити затримки, які важко оцінити, тому винятки там зазвичай не використовуються.

Дякую

FACULTY OF APPLIED MATHEMATICS AND INFORMATICS
LVIV UNIVERSITY