НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «ОДЕСЬКА ЮРИДИЧНА АКАДЕМІЯ»

ФАКУЛЬТЕТ КІБЕРБЕЗПЕКИ ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

28 листопада 2025 року

КІБЕРБЕЗПЕКА В СУЧАСНОМУ СВІТІ: АКТУАЛЬНІ ВИКЛИКИ

МАТЕРІАЛИ VI МІЖНАРОДНОЇ НАУКОВО-ПРАКТИЧНОЇ КОНФЕРЕНЦІЇ

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Одеська юридична академія»
Факультет кібербезпеки та інформаційних технологій
Кафедра кібербезпеки

КІБЕРБЕЗПЕКА В СУЧАСНОМУ СВІТІ: АКТУАЛЬНІ ВИКЛИКИ

МАТЕРІАЛИ VI МІЖНАРОДНОЇ НАУКОВО-ПРАКТИЧНОЇ КОНФЕРЕНЦІЇ

28 листопада 2025 року, м. Одеса

Одеса, 2025

MINISTRY OF EDUCATION AND SCIENCE OF UKRAINE
National University «Odesa Law Academy»
Faculty of Cybersecurity and Information Technologies
Department of Cybersecurity

CYBERSECURITY IN TODAY'S WORLD: ACTUAL CHALLENGES

MATERIALS OF THE VI INTERNATIONAL SCIENTIFIC AND PRACTICAL CONFERENCE

November 28, 2025, Odesa

Odesa, 2025

УДК 004.056

Відповідальній редактор:

О. В. Дикий, декан факультету кібербезпеки та інформаційних технологій, кандидат юридичних наук, доцент

Матеріали видано в авторській редакції.
Повну відповідальність за достовірність та якість поданого матеріалу несуть учасники конференції, їхні наукові керівники, які рекомендували ці матеріали до друку.

Рекомендовано до друку
Вченою радою Національного університету
«Одеська юридична академія»
(протокол №3 від 29.11.2025 р.)

Матеріали Міжнародної науково-практичної конференції «Кібербезпека в сучасному світі: актуальні виклики» (м. Одеса, 28 листопада 2025 р.). Одеса, 2025. 287 с.

У конференції взяли участь студенти, аспіранти, молоді вчені, викладачі та науковці. Конференція проводиться на базі Національного університету «Одеська юридична академія».

Редакційна колегія:

ГОРБАЧЕНКО Станіслав, д.е.н., професор

СОКОЛОВ Артем, д.т.н., професор

АХМАМЕТЬЄВА Ганна, к.т.н., доцент

РАЗІНКІН Нікіта, асистент кафедри

© НУ «Одеська юридична академія», 2025

UDC 004.056

Editor-in-chief:

O. V. Dykyi, Dean of the Faculty of Cybersecurity and Information Technologies,
Candidate of Law, Associate Professor

The materials were published in the author's edition.
Full responsibility for the reliability and quality of the submitted material
bears the participants of the conference, their scientific supervisors,
who recommended these materials for publication.

Recommended for printing
by the Academic Council of the National University
«Odesa Law Academy»
(Minutes No. 3 dated 11/29/2025)

Materials of the International Scientific and Practical Conference «Cybersecurity in the Modern World: Current Challenges» (Odessa, November 28, 2025). Odesa, 2025. 287 p.

The conference was attended by students, postgraduates, young scientists, teachers and researchers. The conference is held at the National University «Odesa Law Academy».

Editorial Board:

GORBACHENKO Stanislav, Doctor of Economics, Professor

SOKOLOV Artem, Doctor of Engineering, Professor

AKHMAMETYEVA Anna, Candidate of Engineering, Associate Professor

RAZINKIN Nikita, Assistant

© National University "Odesa Law Academy", 2025

VI Міжнародна науково-практична конференція

«Кібербезпека в сучасному світі: актуальні виклики»

28 листопада 2025 року

ТЕМАТИЧНІ НАПРЯМКИ КОНФЕРЕНЦІЇ:

Секція 1. Алгоритмічні та технічні аспекти кібербезпеки

Секція 2. Штучний інтелект та інформаційні технології

Секція 3. Управлінські, соціальні та психологічні аспекти взаємодії з кіберпростором

Секція 4. Нормативно-правові засади кібербезпеки та захисту інформації

E-mail: kiberprostirconf@gmail.com (Для питань та тез)

VI International Scientific and Practical Conference

«Cybersecurity in today's world: actual challenges»

28 November 2025 year

THEMATIC DIRECTIONS OF THE CONFERENCE:

Section 1. Algorithmic and technical aspects of cybersecurity

Section 2. Artificial intelligence and information technologies

Section 3. Management, social and psychological aspects of interaction with cyberspace

Section 4. Regulatory and legal principles of cybersecurity and information protection

E-mail: kiberprostirconf@gmail.com (Для питань та тез)

ІДІОМА RAII ЯК ЕВОЛЮЦІЙНА ВІДПОВІДЬ НА ПРОБЛЕМИ РУЧНОГО УПРАВЛІННЯ ПАМ'ЯТТЮ В C++

Д'якова Ксенія

Національний університет «Одеська юридична академія»,
студентка факультету кібербезпеки та інформаційних технологій

Мова програмування C++ історично поєднує низькорівневу потужність C із засобами об'єктно-орієнтованого програмування. Однією з її ключових особливостей є прямий контроль над динамічною пам'яттю, що дозволяє створювати високопродуктивні системи, але водночас породжує складні та небезпечні помилки. У ранніх версіях C++ (до C++11) управління пам'яттю здійснювалося вручну – через оператори new та delete, що вимагало від програміста абсолютної дисципліни та точності. Витоки пам'яті, висячі вказівники, подвійне звільнення – ці помилки не лише ускладнювали розробку, а й створювали серйозні ризики для стабільності та безпеки програмного забезпечення [1].

Ручне управління пам'яттю в C++ передбачає, що програміст самостійно відповідає за виділення та звільнення ресурсів. Оператор new створює об'єкт у купі (heap), а delete – знищує його та повертає пам'ять системі. Якщо delete не викликається, виникає витік пам'яті – ресурс залишається захопленим, але недоступним, що поступово призводить до вичерпання доступної пам'яті [2]. Особливо небезпечними є ситуації, коли програма достроково виходить з функції, минаючи виклик delete (рис. 1), або коли вказівник продовжує посилатися на вже звільнену область – так званий висячий вказівник (рис. 2). У першому демонстраційному прикладі показано, як достроковий вихід з функції без виклику delete призводить до витоку пам'яті, а в другому прикладі – доступ до висячого вказівника – програма продовжує працювати, створюючи ілюзію коректності, хоча фактично виконується код з невизначеною поведінкою (Undefined Behavior), що може призвести до краху програми або порушення безпеки. Висячі вказівники перетворюють програму на "мінне поле" невизначеної поведінки. Спроба доступу до звільненої пам'яті може не викликати миттєвої помилки, створюючи ілюзію коректної роботи, але водночас пошкоджуючи дані або створюючи вразливості, які проявляться значно пізніше і в зовсім іншій частині коду. Діагностика таких помилок є одним із найскладніших завдань у програмуванні. Наслідки таких помилок часто не проявляються одразу, що ускладнює їх діагностику.

```
#include <iostream>
#include <string>

// Використовуємо стандартний простір імен для спрощення
```

```
using namespace std;

// 1. Допоміжний клас для демонстрації управління ресурсами
class Resource {
private:
    string name;
public:
    Resource(string n) : name(n) {
        cout << "  [+] RESOURCE '" << name << "' CREATED." << endl;
    }
    ~Resource() {
        cout << "  [-] RESOURCE '" << name << "' DESTROYED." << endl;
    }
    void use() {
        cout << "  [*] Using resource '" << name << "'." << endl;
    }
};

// 2. Функція, що демонструє проблему
void demonstrate_memory_leak(bool condition) {
    cout << "Entering function 'demonstrate_memory_leak'..." << endl;
    Resource* ptr = new Resource("Leaked Resource");

    if (condition) {
        cout << "  Condition is true. Early return!" << endl;
        // ПОМИЛКА: Ми виходимо з функції, не викликавши delete.
        return;
    }

    ptr->use();
    delete ptr;
    cout << "Leaving function 'demonstrate_memory_leak'..." << endl;
}

// 3. Головна функція для запуску
int main() {
    demonstrate_memory_leak(true); // Викликаємо з умовою, що спричиняє витік
    cout << "\nAfter function call. 'Leaked Resource' was never destroyed." << endl;
    return 0;
}
```

Рисунок 1 – Код-приклад "вистріл в ногу"

```
#include <iostream>
#include <string>

// Використовуємо стандартний простір імен для спрощення
using namespace std;

// 1. Той самий допоміжний клас
class Resource {
private:
    string name;
public:
    Resource(string n) : name(n) {
        cout << "  [+] RESOURCE '" << name << "' CREATED." << endl;
    }
```

```
    ~Resource() {
        cout << "  [-] RESOURCE '" << name << "' DESTROYED." << endl;
    }
    void use() {
        cout << "  [*] Using resource '" << name << "'." << endl;
    }
};

// 2. Функція, що демонструє проблему
void demonstrate_dangling_pointer() {
    cout << "\nEntering function 'demonstrate_dangling_pointer'..." << endl;
    Resource* ptr1 = new Resource("Dangling Test");
    Resource* ptr2 = ptr1;

    cout << "  Deleting ptr1..." << endl;
    delete ptr1; // Пам'ять звільнена, деструктор викликано.

    cout << "  Trying to use ptr2..." << endl;
    // КАТАСТРОФА: Доступ до вже звільненої пам'яті.
    ptr2->use();

    cout << "Leaving function 'demonstrate_dangling_pointer'..." << endl;
}

// 3. Головна функція для запуску
int main() {
    demonstrate_dangling_pointer();
    return 0;
}
```

Рисунок 2 – Код для демонстрації висячого вказівника

Саме ці виклики стали рушієм еволюції мови C++ у напрямі безпечного управління ресурсами. Пошук рішення привів до формулювання ідіоми RAII – Resource Acquisition Is Initialization – яка докорінно змінила підхід до роботи з пам’яттю. RAII передбачає, що захоплення ресурсу (наприклад, пам’яті) відбувається в конструкторі об’єкта, а його звільнення – у деструкторі. Це дозволяє прив’язати життєвий цикл ресурсу до життєвого циклу об’єкта. Якщо об’єкт створено на стеку, його деструктор гарантовано викликається при виході з області видимості – незалежно від того, чи це нормальне завершення функції, чи виняткова ситуація [3].

На рис. 3 показано, як створення об’єкта Resource на стеку забезпечує автоматичне звільнення пам’яті без явного виклику delete. Це усуває ризик витоку пам’яті навіть у випадках дострокового виходу з функції або винятків.

```
#include <iostream>
#include <string>

using namespace std;

// Допоміжний клас, що імітує ресурс, який потрібно створювати та знищувати
class Resource {
```

```
public:
    // Конструктор "захоплює" ресурс
    explicit Resource(const string& name) : name_(name) {
        cout << "[+] RESOURCE '" << name_ << "' CREATED." << endl;
    }

    // Деструктор "звільняє" ресурс
    ~Resource() {
        cout << "[-] RESOURCE '" << name_ << "' DESTROYED." << endl;
    }

    void use() const {
        cout << "[*] Using resource '" << name_ << "'." << endl;
    }

private:
    string name_;
};

// Функція, що демонструє автоматичне управління життєвим циклом об'єкта
void demonstrate_raii_on_stack() {
    cout << "\nEntering function 'demonstrate_raii_on_stack'..." << endl;
    {
        cout << "  Entering inner scope..." << endl;
        // 1. Створення об'єкта на стеку
        // Викликається конструктор Resource
        Resource res("Stack-based RAII");

        // 2. Використання ресурсу
        res.use();

        cout << "  Leaving inner scope..." << endl;
        // 3. Вихід з області видимості
        // Деструктор для 'res' викликається автоматично
    }
    cout << "Leaving function 'demonstrate_raii_on_stack'..." << endl;
}

int main() {
    demonstrate_raii_on_stack();
    return 0;
}
```

Рисунок 3 – Демонстрація ідіоми RAII для об'єкта, створеного на стеку

RAII не лише вирішує проблему витоків, а й унеможливлює доступ до звільненої пам’яті, оскільки об’єкт знищується автоматично, і вказівник на нього більше не існує. Це усуває клас помилок, пов’язаних із висячими вказівниками. RAII є фундаментом сучасного C++ і реалізується через стандартні засоби, зокрема, розумні вказівники (std::unique_ptr, std::shared_ptr), які автоматично керують ресурсами. У поєднанні з винятками, шаблонами та контейнерами STL, RAII дозволяє створювати безпечний, надійний та масштабований код [4].

Особливо важливим є застосування RAII у системах, де стабільність і контроль над ресурсами є критичними: розробка ігрових рушіїв, високочастотний трейдинг, системне програмування, вбудовані системи, наукові обчислення. У таких середовищах помилка управління пам’яттю може

призвести до катастрофічних наслідків, тому автоматизація цього процесу є не просто бажаною, а необхідною. Також RAII сприяє покращенню читабельності та підтримуваності коду. Програміст більше не повинен вручну відстежувати, де і коли викликати delete, що зменшує когнітивне навантаження та ризик помилок. Це особливо важливо в командній розробці, де код часто змінюється і доповнюється різними учасниками.

Хоча RAII ідеально працює для об'єктів на стеку, у багатьох задачах потрібне створення об'єктів у динамічній пам'яті, наприклад, при побудові дерев, списків або фабрик об'єктів. У таких випадках RAII реалізується через обгортки – розумні вказівники, які інкапсулюють динамічний ресурс і керують ним автоматично. Це дозволяє поєднати гнучкість динамічної пам'яті з безпекою автоматичного управління.

У сучасному C++ (починаючи з C11) ідіома RAII стала стандартом. Її підтримують всі основні бібліотеки, а нові засоби – такі як std::move, std::make_unique, std::make_shared – роблять її ще зручнішою. Це свідчить про те, що RAII не лише вирішила проблему ручного управління пам'яттю, а й стала основою нової парадигми програмування в C.

Тим самим ідіома RAII є еволюційною відповіддю на фундаментальні проблеми ручного управління пам'яттю в C++. Вона забезпечує автоматичне, безпечне та передбачуване управління ресурсами, усуває ризики витоків пам'яті, висячих вказівників та невизначеної поведінки. RAII стала основою сучасного C++ і є критично важливою для розробки надійного програмного забезпечення в системах з високими вимогами до стабільності та продуктивності.

Список використаних джерел:

1. Stroustrup B. Quotes. *Goodreads*. URL: https://www.goodreads.com/author/quotes/64947.Bjarne_Stroustrup
2. Meyers S. Effective C++: 55 Specific Ways to Improve Your Programs and Designs. Addison-Wesley, 2011. 297 p. URL: https://ptgmedia.pearsoncmg.com/images/9780321334879/samplepages/0321334876.pdf
3. RAII. *cppreference.com*. URL: https://en.cppreference.com/w/cpp/language/raii.html
4. ISO/IEC 14882:2024(E) – Programming Language C++ 23. URL: https://isocpp.org/std/the-standard

Ключові слова: управління пам'яттю, C++, витоки пам'яті, висячі вказівники, ідіома RAII.
Keywords: memory management, C++, memory leaks, dangling pointers, RAII idiom.

Науковий керівник: к.т.н., доцент Трофименко О.Г.

ЗМІСТ