МІНІСТЕРСТВО ОСВІТИ ТА НАУКИ УКРАЇНИ
ХАРКІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ
РАДІОЕЛЕКТРОНІКИ

ЗВІТ

Практичної роботи №1

з дисципліни «Аналіз та рефакторинг коду»

на тему «Основні рекомендації написання коду на C++»

Виконав:
Тарасов Ростислав Максимович

Перевірив:
ст. викладач кафедри ПІ
Сокорчук Ігор Петрович

Харків 2025

1 ІСТОРІЯ ЗМІН

№	Дата	Версія звіту	Опис змін та виправлень
1	09.11.2025	1.0	Створено звіт
2	30.11.2025	1.1	Додано відеозапис

2 ЗАВДАННЯ

Доповідь на тему «Основні рекомендації написання коду на C++»

3 ОПИС ВИКОНАНОЇ РОБОТИ

– Рекомендація 1: Іменування та форматування

Використовуйте змістовні імена.

Дотримуйтесь єдиного стилю (наприклад, CamelCase для класів, snake_case для змінних).

– Рекомендація 2: Використання Сучасного C++

auto для складних типів (але не зловживайте).

Range-based for loops для ітерацій.

nullptr замість NULL.

– Рекомендація 3/4: Керування пам'яттю та RAII

Для запобігання витоків пам’яті і винятків:

"Обгортайте" ресурси (пам'ять, файли, м'ютекси) в об'єкти.

Використовуйте std::unique_ptr та std::shared_ptr.

– Рекомендація 5: unique_ptr vs shared_ptr

std::unique_ptr (унікальне володіння):

- Легкий, швидкий (нульові накладні витрати).
- Не можна копіювати, можна переміщувати (std::move).
- Використовуйте його за замовчуванням.

std::shared_ptr (спільне володіння):

- Використовує лічильник посилань.
- Ресурс живе, доки існує хоч один shared_ptr.
- Використовуйте, лише коли володіння справді має бути спільним.

– Рекомендація 6: Ефективна передача параметрів (const&)

Прості типи (int, double, char) — передавайте за значенням (by value).

Великі/складні об'єкти (string, vector, ваші класи) — передавайте за константним посиланням (const&).

– Рекомендація 7: Використання const

Використовуйте const скрізь, де це можливо.

Це захищає від випадкових змін.

Це чітко виражає ваші наміри.

Функції-члени, що не змінюють об'єкт, мають бути const.

– Рекомендація 8: Перевага STL

std::vector > C-style масиви (new int[10])

std::string > C-style рядки (char*)

std::array > C-style масиви (int arr[10])

<algorithm> > ваші власні sort, find

– Рекомендація 9: Керування заголовками

Заголовки (.h, .hpp) - це контракт вашого класу.

.cpp файли - це реалізація.

Правило 1: Включайте в .h файл лише те, що абсолютно необхідно.

Правило 2: Використовуйте попереднє оголошення (forward declaration) замість #include, де це можливо.

4 ВИСНОВКИ

– Було розглянуто ключові рекомендації та найкращі практики написання чистого, безпечного та ефективного коду мовою C++.

– Було проаналізовано поширені помилки (наприклад, ручне керування пам'яттю, копіювання великих об'єктів) та їхні сучасні вирішення (RAII, const&).

– Отримано навички застосування ідіоми RAII за допомогою розумних вказівників (std::unique_ptr, std::shared_ptr) для уникнення витоків пам'яті.

– Отримано навички використання можливостей сучасного C++ (C++11 та новіше), таких як range-based for loops, auto та nullptr, для підвищення читабельності коду.

– Було розглянуто важливість const-коректності та правильного керування файлами заголовків для покращення безпеки та швидкості компіляції проєкту.

5 ВИКОРИСТАНІ ДЖЕРЕЛА

1. Meyers S. Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14. O'Reilly Media, 2014. 334 p.

2. Stroustrup B. A Tour of C++ (Second Edition). Addison-Wesley Professional, 2018. 240 p.

3. C++ Core Guidelines / ed. Bjarne Stroustrup, Herb Sutter. URL: https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines (дата звернення: 09.11.2025).

ДОДАТОК А
Відеозапис

Посилання: https://youtu.be/jHHsqtBjn4A

Вступ (Слайд 1): 00:00-00:23

Мета (Слайд 2): 00:23-01:04

Рекомендація 1 (Слайд 3): 01:04-01:47

Рекомендація 2 (Слайд 4): 01:47-02:31

Рекомендація 3 (Слайд 5): 02:31-03:06

Рекомендація 4 (Слайд 6): 03:06-04:10

Рекомендація 5 (Слайд 7): 04:10-05:03

Рекомендація 6 (Слайд 8): 05:03-05:50

Рекомендація 7 (Слайд 9): 05:50-06:37

Рекомендація 8 (Слайд 10): 06:37-07:19

Рекомендація 9 (Слайд 11): 07:19-07:50

Загальний приклад (Слайд 12): 07:50-08:42

Висновки (Слайд 13): 08:42-09:13

Кінець (Слайд 14): 09:13-09:14

ДОДАТОК Б

Слайди презентації

Рисунок Б.1 – Титульний слайд

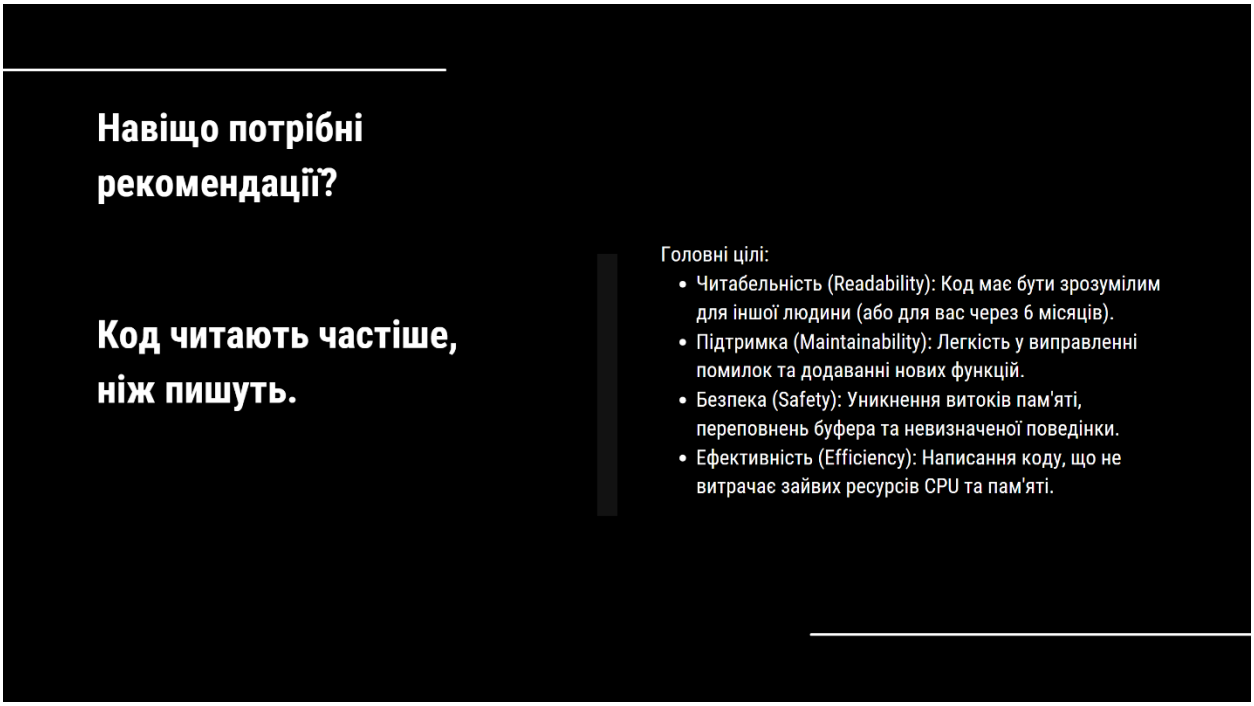

Рисунок Б.2 - Вступний слайд

Рекомендація 1. Іменування та форматування

- Використовуйте змістовні імена.
- Дотримуйтесь єдиного стилю (наприклад, CamelCase для класів, snake_case для змінних).
- Код — це комунікація.

```
// ПОГАНО
class usr {
    int d;
public:
    int cal(int v) { return d * v; }
};

// ДОБРЕ
class UserProfile {
public:
    // Імена чітко описують, що вони роблять
    int calculateScore(int value) const {
        return data_score_ * value;
    }
private:
    // Єдиний стиль (snake_case з підкресленням)
    int data_score_ = 0; // Завжди ініціалізуйте!
};
```

Рисунок Б.3 – Іменування та форматування

Рекомендація 2. Використовуйте Сучасний C++ (C++11 та новіше)

```
#include <vector>
#include <iostream>
int main() {
    // ПОГАНО (C++03 стиль)
    std::vector<int> numbers = { 1, 2, 3, 4, 5 };
    for (std::vector<int>::iterator it = numbers.begin();
        it != numbers.end();
        ++it) {
        std::cout << *it;
    }
    int* p = NULL; // NULL - це просто (void*)0 або 0

    // ДОБРЕ (C++11/17 стиль)
    for (int number : numbers) { // Набагато чистіше
        std::cout << number;
    }
    int* ptr = nullptr; // nullptr - це окремий тип
}
```

- Забудьте про C++03. Стандарти C++11, 14, 17, 20 додали мові безпеки та виразності.
- auto для складних типів (але не зловживайте).
- Range-based for loops для ітерацій.
- nullptr замість NULL.

Рисунок Б.4 – Використання сучасного C++

Рекомендація 3. Проблема: Ручне керування пам'яттю

```cpp
// ПОГАНО (Небезпечно!)
void legacyCode(bool condition) {
    MyObject* obj = new MyObject(); // 1. Виділили ресурс

    if (condition) {
        throw std::runtime_error("Щось пішло не так!");
        // 2. Стався виняток...
    }

    obj->doWork();
    delete obj; // 3. ...цей рядок НІКОЛИ не виконається!
    // МИ ОТРИМАЛИ ВИТІК ПАМ'ЯТІ.
}
```

- Ручні new та delete — головне джерело помилок у C++.
- Проблема: Витоки пам'яті (Memory Leaks).
- Проблема: Винятки (Exceptions).

Рисунок Б.5 – Ручне керування пам’яттю

Рекомендація 4. Рішення: RAII та Розумні вказівники

- RAII (Resource Acquisition Is Initialization) — "Захоплення ресурсу є ініціалізація".
- "Обгортайте" ресурси (пам'ять, файли, м'ютекси) в об'єкти.
- Використовуйте std::unique_ptr та std::shared_ptr.

```cpp
#include <memory> // Не забудьте!
#include <stdexcept>

// ДОБРЕ (Безпечно за RAII)
void modernCode(bool condition) {
    // 1. Ресурс керується об'єктом unique_ptr
    auto obj = std::make_unique<MyObject>();

    if (condition) {
        throw std::runtime_error("Щось пішло не так!");
        // 2. Стався виняток...
    }

    obj->doWork();
    // 3. 'delete' не потрібен!
    // Пам'ять звільниться АВТОМАТИЧНО,
    // коли 'obj' вийде з області видимості.
}
```

Рисунок Б.6 – RAII та розумні вказівки

Рекомендація 5. unique_ptr vs shared_ptr

- std::unique_ptr (унікальне володіння)
  - Легкий, швидкий (нульові накладні витрати).
  - Не можна копіювати, можна переміщувати (std::move).
  - Використовуйте його за замовчуванням.
- std::shared_ptr (спільне володіння)
  - Використовує лічильник посилань.
  - Ресурс живе, доки існує хоч один shared_ptr.
  - Використовуйте, лише коли володіння справді має бути спільним.

```cpp
#include <memory>

// 1. unique_ptr: за замовчуванням
auto u_ptr = std::make_unique<Widget>(1);
// auto u_ptr2 = u_ptr; // ПОМИЛКА КОМПІЛЯЦІЇ (добре!)
auto u_ptr_moved = std::move(u_ptr); // ОК, володіння передано

// 2. shared_ptr: коли потрібно спільне володіння
auto s_ptr = std::make_shared<Widget>(2);
auto s_ptr2 = s_ptr; // ОК, лічильник = 2
// 'Widget(2)' буде видалено, коли s_ptr та s_ptr2 вийдуть з області видимості
```

Рисунок Б.7 - unique_ptr vs shared_ptr

Рекомендація 6. Ефективна передача параметрів

Прості типи (int, double, char) — передавайте за значенням (by value).

Великі/складні об'єкти (string, vector, ваші класи) — передавайте за константним посиланням (const&).

```cpp
#include <iostream>
// ПОГАНО (Дороге копіювання 'user_data')
void processData(std::string user_data) {
    // Тут створюється ПОВНА копія рядка.
    // Якщо рядок великий - це дуже повільно.
    std::cout << user_data;
}

// ДОБРЕ (Без копіювання)
void processData(const std::string& user_data) {
    // Передається лише посилання (адреса)
    // 'const' гарантує, що функція
    // випадково не змінить оригінал.
    std::cout << user_data;
}
```

Рисунок Б.8 – Ефективна передача параметрів

Рекомендація 7. Використовуйте const

- Використовуйте const скрізь, де це можливо.
- Це захищає від випадкових змін.
- Це чітко виражає ваші наміри.
- Функції-члени, що не змінюють об'єкт, мають бути const.

```
// ПОГАНО
class DataManager {
    int id;
public:
    int getID() { // Не 'const'
        id = 5; // Компілятор це ПРОПУСТИТЬ, хоча не мав би
        return id;
    }
};

// ДОБРЕ
class DataManager {
    int id_;
public:
    // 'const' обіцяє, що функція не змінить об'єкт
    int getID() const {
        id_ = 5; // ПОМИЛКА КОМПІЛЯЦІЇ (добре!)
        return id_;
    }
};
```

Рисунок Б.9

Рекомендація 8. Надавайте перевагу STL

```
// ПОГАНО (C-style, небезпечно)
void C_style() {
    int* list = new int[5];
    list[0] = 1;
    // ...
    // А якщо забудемо delete[] list?
    // А якщо вийдемо за межі [0..4]?
}

// ДОБРЕ (STL, безпечно, зручно)
#include <vector>
#include <algorithm>
void STL_style() {
    std::vector<int> list = { 1, 5, 2, 4, 3 };
    list.push_back(6); // Легко додати
    // Пам'ять керується автоматично (RAII)
    std::sort(list.begin(), list.end()); // Є готові алгоритми
}
```

- STL (Standard Template Library) — ваш набір інструментів.
- Не винаходьте колесо.
- std::vector > C-style масиви (new int[10])
- std::string > C-style рядки (char*)
- std::array > C-style масиви (int arr[10])
- <algorithm> > ваші власні sort, find

Рисунок Б.10 – Перевага STL

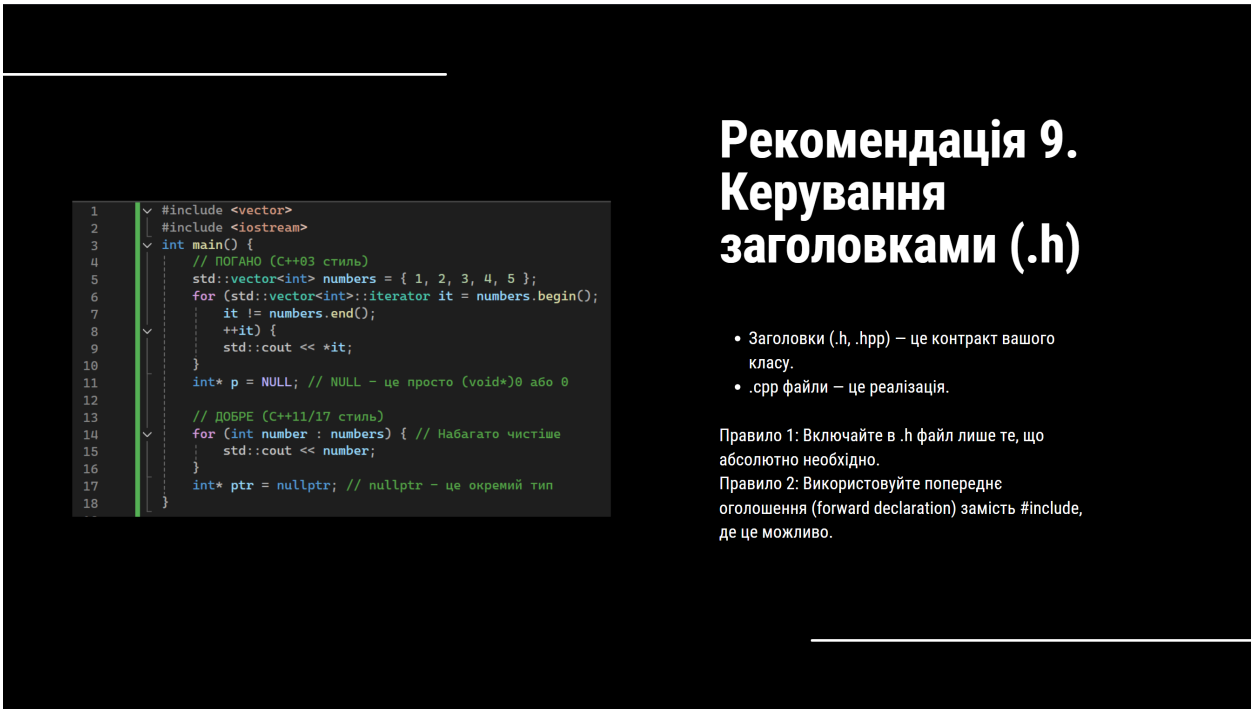

Рисунок Б.11 – Керування заголовками

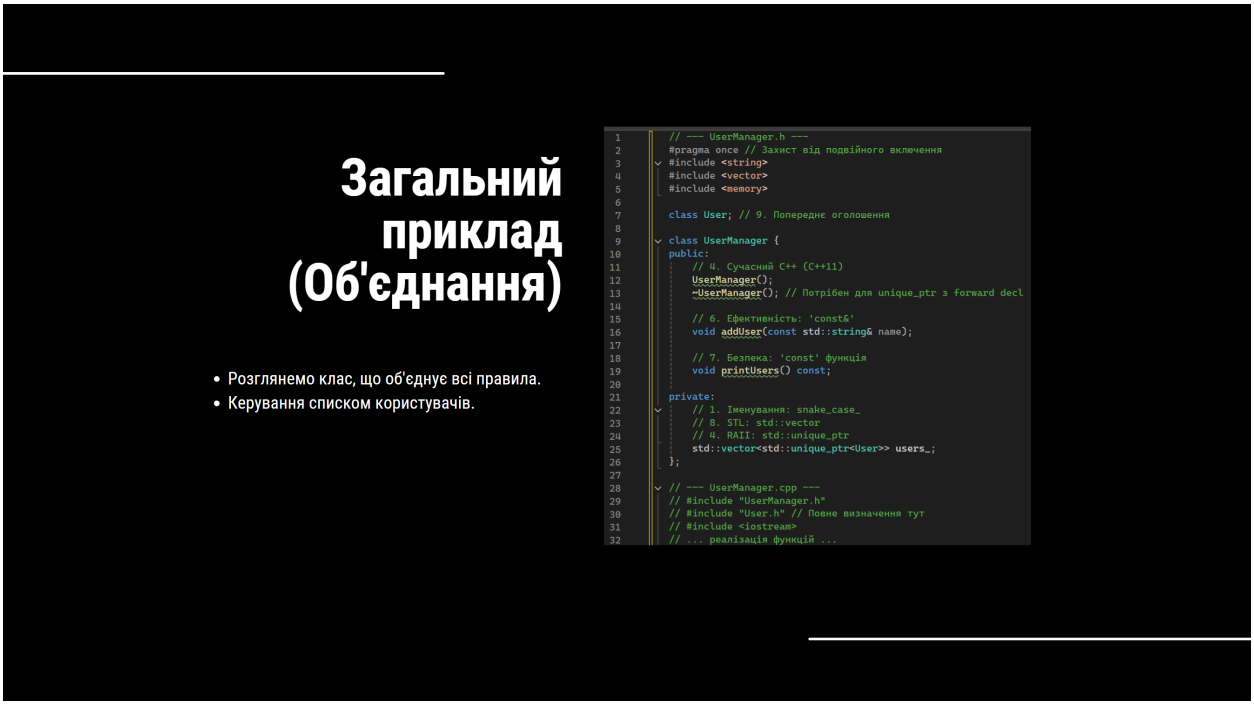

Рисунок Б.12 – Загальний приклад

Висновки

- Пишіть код для людей, а не лише для компілятора.
- RAII — це ваш головний інструмент. std::unique_ptr — ваш друг.
- Надавайте перевагу STL (vector, string) над C-style підходами.
- Використовуйте const та const& агресивно.
- Керуйте заголовками (.h), щоб прискорити компіляцію.

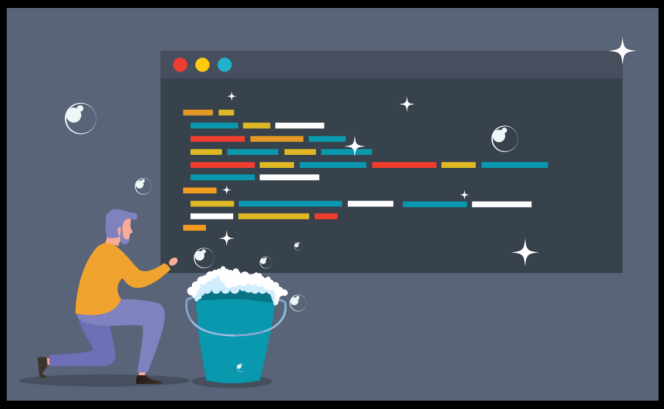

Рисунок Б.13 - Висновки

Тарасов Ростислав

ПЗПІ-23-8

ДЯКУЮ ЗА УВАГУ

Рисунок Б.14 – Кінець

ДОДАТОК В

Програмний код

В.1 Рекомендація 1

```
// ПОГАНО
class usr {
int d;
public:
int cal(int v){return d*v;}
};

// ДОБРЕ
class UserProfile {
 public:
     // Імена чітко описують, що вони роблять
     int calculateScore(int value) const {
         return data_score_ * value;
     }
 private:
     // Єдиний стиль (snake_case з підкресленням)
     int data_score_ = 0; // Завжди ініціалізуйте!
 };

```

В.2 Рекомендація 2

```
// ПОГАНО (C++03 стиль)
std::vector<int> numbers = {1, 2, 3, 4, 5};
for (std::vector<int>::iterator it = numbers.begin(); it != numbers.end();
++it) {
     std::cout << *it;
}
int* p = NULL; // NULL - це просто (void*)0 або 0

// ДОБРЕ (C++11/17 стиль)
for (int number : numbers) { // Набагато чистіше
```

```
10      std::cout << number;
11  }
12  int* ptr = nullptr; // nullptr - це окремий тип
13
```

В.3 Рекомендація 3

```
1  // ПОГАНО (Небезпечно!)
2  void legacyCode(bool condition) {
3      MyObject* obj = new MyObject(); // 1. Виділили ресурс
4
5      if (condition) {
6          throw std::runtime_error("Щось пішло не так!");
7          // 2. Стався виняток...
8      }
9
10      obj->doWork();
11      delete obj; // 3. ...цей рядок НІКОЛИ не виконається!
12                  // МИ ОТРИМАЛИ ВИТІК ПАМ'ЯТІ.
13  }
14
```

В.4 Рекомендація 4

```
1  #include <memory> // Не забудьте!
2
3  // ДОБРЕ (Безпечно за RAII)
4  void modernCode(bool condition) {
5      // 1. Ресурс керується об'єктом unique_ptr
6      auto obj = std::make_unique<MyObject>();
7
8      if (condition) {
9          throw std::runtime_error("Щось пішло не так!");
10          // 2. Стався виняток...
11      }
12
13      obj->doWork();
14      // 3. 'delete' не потрібен!
15      // Пам'ять звільниться АВТОМАТИЧНО,
```

```
16      // коли 'obj' вийде з області видимості.
17  }
18
```

19 В.5 Рекомендація 5

```
1  // 1. unique_ptr: за замовчуванням
2  auto u_ptr = std::make_unique<Widget>(1);
3  // auto u_ptr2 = u_ptr; // ПОМИЛКА КОМПІЛЯЦІЇ (добре!)
4  auto u_ptr_moved = std::move(u_ptr); // ОК, володіння передано
5
6  // 2. shared_ptr: коли потрібно спільне володіння
7  auto s_ptr = std::make_shared<Widget>(2);
8  auto s_ptr2 = s_ptr; // ОК, лічильник = 2
9  // 'Widget(2)' буде видалено, коли s_ptr та s_ptr2 вийдуть з області
видимості
10
```

В.6 Рекомендація 6

```
1  // ПОГАНО (Дороге копіювання 'user_data')
2  void processData(std::string user_data) {
3      // Тут створюється ПОВНА копія рядка.
4      // Якщо рядок великий - це дуже повільно.
5      std::cout << user_data;
6  }
7
8  // ДОБРЕ (Без копіювання)
9  void processData(const std::string& user_data) {
10      // Передається лише посилання (адреса)
11      // 'const' гарантує, що функція
12      // випадково не змінить оригінал.
13      std::cout << user_data;
14  }
15
```

В.7 Рекомендація 7

```
1  // ПОГАНО
2  class DataManager {
```

```
3       int id;
4   public:
5       int getID() { // Не 'const'
6           id = 5; // Компілятор це ПРОПУСТИТЬ, хоча не мав би
7           return id;
8       }
9   };
10
11   // ДОБРЕ
12   class DataManager {
13       int id_;
14   public:
15       // 'const' обіцяє, що функція не змінить об'єкт
16       int getID() const {
17           id_ = 5; // ПОМИЛКА КОМПІЛЯЦІЇ (добре!)
18           return id_;
19       }
20   };
21
```

В.8 Рекомендація 8

```
1   // ПОГАНО (C-style, небезпечно)
2   void C_style() {
3       int* list = new int[5];
4       list[0] = 1;
5       // ...
6       // А якщо забудемо delete[] list?
7       // А якщо вийдемо за межі [0..4]?
8   }
9
10   // ДОБРЕ (STL, безпечно, зручно)
11   #include <vector>
12   #include <algorithm>
13   void STL_style() {
14       std::vector<int> list = {1, 5, 2, 4, 3};
15       list.push_back(6); // Легко додати
16       // Пам'ять керується автоматично (RAII)
17       std::sort(list.begin(), list.end()); // Є готові алгоритми
```

```
18  }
```

В.9 Рекомендація 9

```
1  // ПОГАНО (UserProfile.h)
2  // Тягне за собою ВЕСЬ код <string> та "Logger.h"
3  // у кожен файл, який включає "UserProfile.h"
4  #include <string>
5  #include "Logger.h" // Logger тут не потрібен
6
7  class UserProfile {
8  private:
9      std::string name_;
10      Logger* logger_; // <-- Вказівник
11  };
12
13  // ДОБРЕ (UserProfile.h)
14  #include <string> // string потрібен, бо він є членом
15
16  class Logger; // <-- Попереднє оголошення. Цього достатньо!
17
18  class UserProfile {
19  private:
20      std::string name_;
21      Logger* logger_; // Компілятор знає, що 'Logger' - це тип
22  };
23  // #include "Logger.h" додається лише в UserProfile.cpp
24
```

В.10 Загальний приклад

```
1  // UserManager.h
2  #pragma once // Захист від подвійного включення
3  #include <string>
4  #include <vector>
5  #include <memory>
6
7  class User; // 9. Попереднє оголошення
8
```

```
9  class UserManager {
10 public:
11     // 4. Сучасний C++ (C++11)
12     UserManager();
13     ~UserManager(); // Потрібен для unique_ptr з forward decl
14
15     // 6. Ефективність: 'const&'
16     void addUser(const std::string& name);
17
18     // 7. Безпека: 'const' функція
19     void printUsers() const;
20
21 private:
22     // 1. Іменування: snake_case_
23     // 8. STL: std::vector
24     // 4. RAII: std::unique_ptr
25     std::vector<std::unique_ptr<User>> users_;
26 };
27
28 // UserManager.cpp
29 // #include "UserManager.h"
30 // #include "User.h" // Повне визначення тут
31 // #include <iostream>
32 // ... реалізація функцій ...
33
```