Основні рекомендації написання коду для мови програмування

C++

Виконав ст. гр. ПЗПІ-22-1 Токар Денис

Вступ

C++ — це одна з найпотужніших і універсальних мов програмування, яка використовується для розробки програмного забезпечення, ігор, і високопродуктивних додатків. Її ключовою особливістю є поєднання об'єктно-орієнтованого та процедурного підходів, що надає розробникам гнучкість у виборі стилю написання коду.

При написанні коду на C++ важливо дотримуватися загальноприйнятих правил, стандартів і стилів програмування, таких як наприклад **C++ Core Guidelines**, що сприятиме полегшенню співпраці в команді, підвищенню читабельності коду та зниженню кількості помилок.

Динамічне виділення пам'яті.

```cpp
// Виділення пам'яті
int* ptr = new int(42);

// Використання
std::cout << *ptr << std::endl;

// Видалення
delete ptr;
```

Витік пам'яті.

```
void leakyFunction() {
    int* ptr = new int(42);
    // Забули видалити ptr
}

int main() {
    for (int i = 0; i < 1000000; i++) {
        leakyFunction();
    }
    // Програма може вичерпати всю доступну пам'ять!
}
```

RAII (Resource Acquisition Is Initialization)

RAII - ідіома C++, яка гарантує правильне звільнення ресурсів. Ресурси прив'язуються до часу життя об'єктів

```
class ResourceManager {
private:
    int* resource;
public:
    ResourceManager() : resource(new int(42)) {}
    ~ResourceManager() { delete resource; }
};

void function() {
    ResourceManager rm; // Автоматично видалиться при виході з області видимості
}
```

Розумні покажчики.

Автоматично управляють пам'яттю та реалізують ідіому RAII.

Основні типи розумних покажчиків:

- std::unique_ptr: для ексклюзивного володіння ресурсом.
- std::shared_ptr: для розділеного володіння ресурсом.
- std::weak_ptr: для неволодіючого посилання на ресурс.

```
void function() {
    std::unique_ptr<int> ptr = std::make_unique<int>(42);
    // ptr автоматично видалиться при виході з області видимості
}
```

Переваги та недоліки ручного управління пам'яттю.

Переваги:

- Повний контроль над життєвим циклом об'єктів;
- Передбачувана продуктивність;
- Відсутність пауз на збирання сміття;
- Можливість оптимізації під конкретні сценарії.

Недоліки:

- Більша складність коду;
- Ризик витоків пам'яті та помилок сегментації;
- Необхідність ретельного відстеження виділення/звільнення пам'яті;
- Збільшення часу розробки та налагодження.

Необхідно писати читабельний код

Основні принципи написання читабельного коду:

- Використовуйте зрозумілі назви для змінних, функцій і класів.
- Уникайте "магічних чисел" - замініть їх іменованими константами.
- Розбивайте довгі функції на менші, кожна з яких виконує одне конкретне завдання.

```
int a;  // Погана назва змінної
a = 5;

int userAge; // Хороша назва змінної
userAge = 5;
```

```
// Поганий приклад
int calculateArea(int length, int width) {
    return length * width * 3; // 3 - "магічне число"
}

// Хороший приклад
const int HEIGHT_FACTOR = 3; // Іменована константа

int calculateArea(int length, int width) {
    return length * width * HEIGHT_FACTOR;
}
```

Необхідно писати читабельний код

Використовуйте однакові відступи.
Дотримуйтесь єдиного стилю в усьому проекті.
Правильно розставляйте пробіли.

```
// Поганий приклад використання відступів і пробілів
for( int i= 0;i<10;i++){ if(condition){ doSomething(); }}
```

```
// Поганий приклад
for(int i=0;i<10;i++){
if(condition){
doSomething();
}else{
doSomethingElse();
}}

// Хороший приклад
for (int i = 0; i < 10; i++) {
    if (condition) {
        doSomething();
    } else {
        doSomethingElse();
    }
}
```

Використовуйте стандартні бібліотеки C++.

```cpp
// Поганий приклад
int numbers[] = {1, 2, 3, 4, 5};
int sum = 0;
for (int i = 0; i < 5; i++) {
    sum += numbers[i];
}

// Хороший приклад
std::vector<int> numbers = {1, 2, 3, 4, 5};
auto sum = std::accumulate(numbers.begin(), numbers.end(), 0);
```

Використовуйте std::string замість рядків у стилі C.

```
// Поганий приклад
char name[10] = "John";
strcat(name, " Doe");

// Хороший приклад
std::string name = "John";
name += " Doe";
```

Правильно обробляйте винятки.

```
// Поганий приклад
if (riskyOperation() == ERROR) {
    // Обробка помилки
}
// Може пропустити деякі помилки

// Хороший приклад
try {
    riskyOperation();
} catch (const std::exception& e) {
    std::cerr << "Error: " << e.what() << std::endl;
}
```

Використовуйте посилання замість показчиків.

```
void processObject(MyClass* obj) {
    if (obj) {
        // Працювати щось з obj
    }
    // Необхідно перевірити на null
}

void processObject(const MyClass& obj) {
    // Працювати з obj
}
```

Висновки

- Управління пам'яттю є критично важливим аспектом програмування на C++. Розуміння динамічного виділення пам'яті, запобігання витокам пам'яті та використання ідіоми RAII є фундаментальними для створення надійного коду.
- Читабельність коду є ключовим фактором для довгострокового успіху проекту. Використання зрозумілих імен змінних, послідовного форматування та належних коментарів полегшує розуміння та підтримку коду.
- Використання сучасних можливостей C++ значно підвищує безпеку, ефективність та якість коду, автоматизуючи багато аспектів управління ресурсами, підвищуючи безпеку та зручність роботи.

Використані джерела:

- **Coding Best Practices for C++ -** Richard Bellairs - https://www.perforce.com/blog/qac/3-coding-best-practices-cpp#three-01
- **C++ Core Guidelines –** isocpp - https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines#main
- **C++ Coding Standards** - Herb Sutter and Andrei Alexandrescu - https://micro-os-plus.github.io/develop/sutter-101/
- **Dynamic memory allocation C++** - Akash Gupta, Geeks for Geeks - https://eng.libretexts.org/Courses/Delta_College/C_-_Data_Structures/03%3A_Arrays/3.01%3A_Dynamic_memory_allocation