

Лекція 10.

Об'єктно-орієнтоване програмування.

Класи.



План на сьогодні

1

Що таке клас?

2

Специфікатори доступу

3

Конструктори класу

4

Оператори доступу

5

Статичні поля

6

Деструктори



FACULTY OF APPLIED
MATHEMATICS AND
INFORMATICS
LVIV UNIVERSITY

Парадигми програмування



FACULTY OF APPLIED
MATHEMATICS AND
INFORMATICS
LVIV UNIVERSITY

Парадигми програмування

Функціональне програмування

- ❖ Парадигма, що базується на функціях.
- ❖ Немає змінних станів, дані незмінні.
- ❖ Використання чистих функцій без побічних ефектів.
- ❖ Функції можуть бути передані як аргументи або повернені з інших функцій.

Об'єктно-орієнтоване програмування

- ❖ Парадигма, що базується на об'єктах та їх взаємодії.
- ❖ Інкапсуляція даних і методів в об'єктах.
- ❖ Наслідування та поліморфізм для повторного використання коду.
- ❖ Зміна стану через методи об'єктів.
- ❖ Основні принципи об'єктно-орієнтованого програмування **SOLID**.

Об'єктно-орієнтоване програмування

Нехай нам потрібно зберігати ширину, довжину, висоту кімнати для обчислення площі кімнати та її об'єму.

Функціональне програмування

Для розв'язування задачі можна визначити три змінні та дві функції:

```
double length, width, height;  
cin>> length>>width>>height;  
calcArea(length, width);  
calcVolume(length, width, height);
```

Об'єктно-орієнтоване програмування

Можна об'єднати всі дані та необхідні функції в одному об'єкті:

```
Room room;  
cin>>room;  
room.calcArea();  
room.calcVolume();
```

Що таке клас?

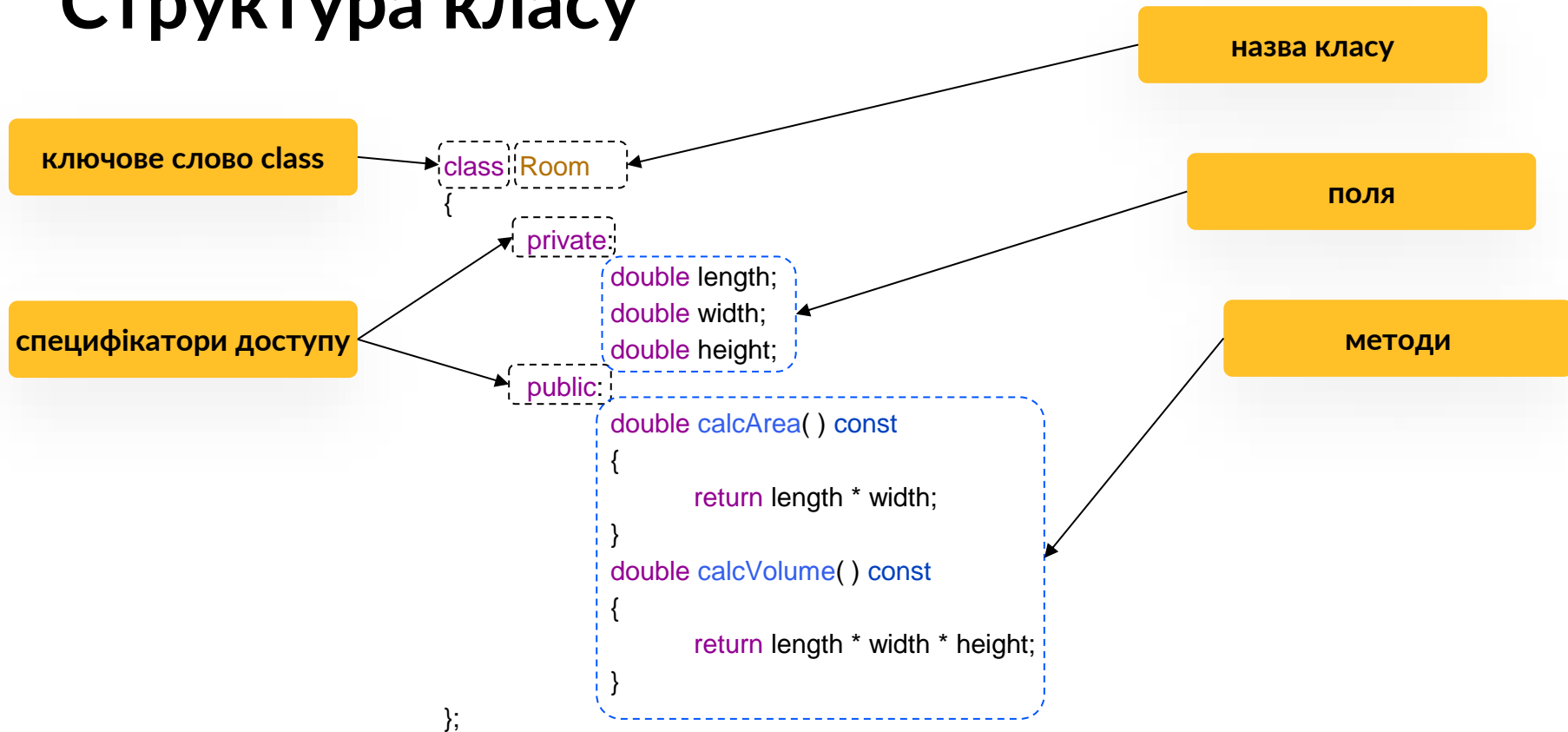
Що таке клас?

- ❖ **Клас** — це шаблон або структура, що описує властивості (поля) та поведінку (методи) об'єктів.
- ❖ Він визначає, які характеристики будуть у об'єктів, створених на його основі.

Основні компоненти класу:

- ❖ **Поля (атрибути)** — змінні, які зберігають стан об'єкта.
- ❖ **Методи** — функції, які визначають поведінку об'єкта.

Структура класу



Використання об'єктів класу

- ❖ Клас `Room` має поля: `length`, `width`, `height` та методи для обчислення площі і об'єму кімнати.
- ❖ Об'єкт `room1` створюється на основі класу `Room` і використовує методи `calcArea()` та `calcVolume()` для обчислень.

```
class Room
{
    public:
        double length;
        double width;
        double height;
        double calcArea() const
        {
            return length * width;
        }
        double calcVolume() const
        {
            return length * width * height;
        }
};
```

```
int main()
{
    Room room1;

    room1.length = 42.5;
    room1.width = 30.8;
    room1.height = 19.2;

    cout << "Area of Room = " << room1.calcArea();
    cout << "Volume of Room = " << room1.calcVolume();
    return 0;
}
```

Відмінності між класом і структурою в C++

- ❖ Поля та методи **структури** є публічними за замовчуванням.
- ❖ Поля та методи **класу** є приватними за замовчуванням.
- ❖ Як **класи** так і **структури** можуть мати суміш публічних, захищених і приватних членів, можуть використовувати наслідування та можуть мати функції-члени.

Рекомендовано використовувати:

- ❖ **struct** для об'єднання базових типів даних без будь-яких класоподібних функцій;
- ❖ **class** клас, коли необхідно мати нестандартні конструктори, приватні або захищені члени, оператори, тощо.

```
class X
{
    public:

    // ...
};
```

```
struct X
{
    // ...
};
```

Специфікатори доступу



FACULTY OF APPLIED
MATHEMATICS AND
INFORMATICS
LVIV UNIVERSITY

Специфікатори доступу

public – необмежений доступ; поля та методи класу доступні як в межах класу, так і поза визначенням класу

protected – поля та методи класу недоступні скрізь поза визначенням класу крім ієрархії похідних класів, дружніх функцій та класів

private – поля та методи класу можуть використовуватися лише в межах класу і є недоступним поза визначенням класу, крім дружніх функцій та класів

Інкапсуляція даних

В об'єктно-орієнтованому програмуванні **інкапсуляція** визначається як зв'язування разом даних і функцій, які ними маніпулюють.

У даному прикладі приватні поля класу **Room** (**length**, **width**, **height**) не можуть бути змінені напряму з зовні класу. Для зміни значень використовується публічний метод **SetData()**, який забезпечує контрольований доступ до цих даних.

```
class Room
{
    private:
        double length;
        double width;
        double height;

    public:
        double calcArea() const
        {
            return length * width;
        }
        void SetData(double x, double y, double z)
        {
            length = x;
            width = y;
            height = z;
        }
};
```

```
int main()
{
    Room room1;

    room1.length = 42.5; // error
    room1.width = 30.8; // error

    room1.SetData(42.5, 30.8, 19.2);
    cout << "Area of Room = " << room1.calcArea();
    cout << "Volume of Room = " << room1.calcVolume();
    return 0;
}
```

помилка доступу

Методи класу

- ❖ **Метод** — це функція всередині класу, яка працює з даними об'єкта.
- ❖ **Конструктори** — відповідають за створення та ініціалізацію об'єктів.
- ❖ **Деструктори** — відповідають за звільнення пам'яті після видалення об'єкта.
- ❖ **Методи-модифікатори (set-методи)** — змінюють поля класу.
- ❖ **Методи-селектори (get-методи)** — повертають захищені поля класу, повинні бути визначені як **const**
- ❖ **Перевантажені оператори** — арифметичні, оператори введення та виведення.

Конструктор класу

Конструктори

Конструктор — це спеціальна функція, яка викликається автоматично під час створення об'єкта:

- ❖ з назвою класу;
- ❖ не має типу повернення;
- ❖ можна перевантажувати, як звичайні функції для створення об'єктів різними способами;
- ❖ мають специфікатор доступу (`private`, `protected`, `public`). В основному конструктори `public`.

Конструктор за замовчуванням

Конструктор без параметрів називається конструктором за замовчуванням

```
class Wall
{
    private:
        double length;
    public:
        Wall()
        {
            length = 0.0;
            cout << "Wall
              Constructor";
        }
};

int main()
{
    Wall wall;

    return 0;
}
```

коли створюється об'єкт `wall`, викликається конструктор `Wall()`. Значення `length` встановлюється під час виклику конструктора, і змінна `length` об'єкта набуває значення 0.0.

Примітка: Якщо в класі не визначено конструктора за замовчуванням та іншого конструктора (з параметрами або копіювання), C++ автоматично створить конструктор за замовчуванням і з порожнім тілом. У випадку якщо вже є визначений конструктор з параметрами то конструктор за замовчуванням автоматично не створюється.

Конструктор за замовчуванням - Приклад

Викликається автоматично при створенні масиву об'єктів класу

```
class Point {  
private:  
    int x;  
    int y;  
public:  
    Point() :x(0), y(0)  
    {  
        static int number = 0;  
        cout << "Point constructor was invoked:" << ++number << endl;  
    }  
};  
  
int main() {  
    Point p1;  
    Point* p2 = new Point();  
    Point mas[10];  
    Point* mas2 = new Point[5];  
}
```

Список ініціалізації полів

У всіх випадках викликається
конструктор за замовчуванням
автоматично

Конструктор з параметрами

Конструктор з параметрами називається параметризованим конструктором.

```
class Wall
{
    private:
        double length;
    public:
        Wall(double len)
        {
            length = len;
        }
};

int main()
{
    Wall wall(1.5);

    return 0;
}
```

- ❖ Відповідають за утворення об'єктів з ініціалізацією полів через аргументи, які передаються в конструктор
- ❖ Може бути декілька в класі для різного набору параметрів
- ❖ Компілятор сам вирішує який конструктор викликати в залежності від кількості і типів аргументів

Конструктор з параметрами - Приклад

```
class Wall {  
private:  
    double length;  
    double height;  
  
public:  
    // parameterized constructor to initialize variables  
    Wall(double len, double hgt) : length(len), height(hgt) { }  
  
    double calculateArea() {  
        return length * height;  
    }  
};
```

```
int main() {  
    Wall wall; // compilation error  
    Wall wall1(10.5, 8.6);  
    Wall wall2(8.5, 6.3);  
  
    cout << "Area of Wall 1: " << wall1.calculateArea() << endl;  
    cout << "Area of Wall 2: " << wall2.calculateArea();  
  
    return 0;  
}
```

Помилка компіляції, оскільки дефолтний конструктор відсутній та автоматично не створюється якщо є вже визначений конструктор з параметрами.

Uniform initialization {} - C++11



```
class Base {  
    char x;  
  
public:  
    Base(char a) : x{ a } { }  
  
    void print() { std::cout << static_cast<int>(x); }  
};
```

```
int main()  
{  
    Base b{ 300 }; // Using uniform initialization with {}  
    b.print();  
    return 0;  
}
```

By using uniform initialization with {} and initializing x with the provided value a, the compiler will perform stricter type-checking and issue a warning or error during compilation, indicating the narrowing conversion from int to char.

Error List

Entire Solution 1 Error 1 Warning 0 of 4 Messages

	Code	Description
	C2398	Element '1': conversion from 'int' to 'char' requires a narrowing conversion
	C4309	'argument': truncation of constant value

Конструктор копіювання

Конструктор копіювання використовується для копіювання даних з одного об'єкта до іншого.

```
class Wall
{
    private:
        double length;
    public:
        Wall(const Wall& obj)
        {
            length = obj.length;
        }
};

int main()
{
    Wall wall1(2.5);
    Wall wall2(wall1);
    Wall wall3 = wall1;

    return 0;
}
```

- ❖ Генерує компілятор автоматично, якщо ми не визначимо самі та немає визначеного конструктора з параметрами.
- ❖ При відсутності полів-посилань чи вказівників і відповідній логіці класу достатньо згенерованого конструктора копіювання.
- ❖ Викликається автоматично при виклику функцій для копіювання аргументів-значень і повернення результату.

Конструктор копіювання - Приклад

```
class Point {  
    int x;  
    int y;  
public:  
    Point(int xx = 0, int yy = 0) : x(xx), y(yy) {}  
    Point(const Point& p) : x(p.x), y(p.y) {  
        static int counter = 0;  
        cout << "Copy constructor was invoked: " << ++counter << endl;  
    }  
};  
  
Point fun1(Point p) {  
    cout << "fun1 was invoked" << endl;  
    return p;  
}  
  
Point& fun2(Point& p) {  
    cout << "fun2 was invoked" << endl;  
    return p;  
}  
  
void main() {  
    Point p1;                // default constructor  
    Point p2(p1);             // copy constructor  
    Point p3 = p1;            // copy constructor  
    Point* p4 = new Point(p2); // copy constructor  
  
    cout << "Invoke fun1" << endl;  
    p2 = fun1(p1);             // copy constructor 2 times  
    cout << "Invoke fun2" << endl;  
    p2 = fun2(p1);  
}
```

Конструктор **explicit**

```
class Distance {
private:
    int meters;

public:
    // Constructor without explicit keyword
    Distance(int m) : meters(m) {
        cout << "Conversion constructor was invoked" << endl;
    }
    void display() const {
        cout << "Distance: " << meters << " meters" << endl;
    }
};

void printDistance(Distance d) {
    d.display();
}

int main() {
    Distance d1 = 10;    // Implicit conversion from int to Distance
    d1.display();

    printDistance(15);   // Implicit conversion from int to Distance
    return 0;
}
```

У C++ **explicit** (явний) конструктор використовується для запобігання неявним перетворенням до типу класу. Позначаючи конструктор як **явний**, ми гарантуємо, що конструктор можна використовувати лише з прямою ініціалізацією, а не з синтаксисом, подібним до присвоєння.

Конструктор **explicit**

```
class Distance {  
private:  
    int meters;  
  
public:  
    // Constructor without explicit keyword  
    explicit Distance(int m) : meters(m) {  
        cout << "Conversion constructor was invoked" << endl;  
    }  
    void display() const {  
        cout << "Distance: " << meters << " meters" << endl;  
    }  
};  
  
void printDistance(Distance d) {  
    d.display();  
}  
  
int main() {  
    Distance d1(10);           // Direct initialization is allowed  
    // Distance d2 = 20;       // Error: implicit conversion not allowed  
  
    // printDistance(15);      // Error: implicit conversion not allowed  
    printDistance(Distance(15)); // Explicit construction required  
    return 0;  
}
```

Конструктор **private**

Якщо конструктор є **приватним**, це означає, що ніхто, крім самого класу (і друзів), **не може створювати його екземпляри** за допомогою цього конструктора. Можна описати статичний методи, такий як `createObject()`, щоб створити екземпляри класу або створити екземпляри в якомусь іншому методі.

```
class TestClass
{
private:
    TestClass(){                // Restricting object creation
        cout << "Object created\n";
    }
public:
    static TestClass* createObject()    // A public method creates an object of this class
    {
        return new TestClass();
    }
};

int main()
{
    // TestClass a; // Compilation error: constructor is inaccessible
    TestClass* obj = nullptr;
    obj = TestClass::createObject();
}
```

Дані класу



FACULTY OF APPLIED
MATHEMATICS AND
INFORMATICS
LVIV UNIVERSITY

Оператори доступу до даних класу

Оператор . (крапка)

використовується для доступу до членів класу через об'єкт. Це дозволяє звертатися до полів або методів об'єкта напряму.

Формат: `object.member`

```
int main()
{
    Wall wall;
    wall.calcArea();
    return 0;
}
```

Оператор -> (стрілка)

використовується для доступу до членів класу через вказівник на об'єкт. Він дозволяє працювати з полями і методами об'єкта, коли ви маєте вказівник на цей об'єкт.

Формат: `pointer->member`

```
int main()
{
    Wall* wallPtr = new Wall();
    wallPtr->calcArea();
    return 0;
}
```

Дані об'єкта і дані класу

```
class Wall
{
    public:
        static int count;
        Wall() {
            length = 0.0;
            cout << "Wall Constructor";
            count++;
        }
    private:
        double length;
};
Wall::count = 0;
void main() {
    cout << Wall::count << endl;
}
```

Нестатичні поля

- ❖ Внутрішній об'єкт (змінна) кожного екземпляра класу.
- ❖ Займає певний обсяг пам'яті в кожному об'єкті класу.
- ❖ Доступ — через об'єкт цього класу (через `this`).

Статичні поля

- ❖ Визначається зі специфікатором `static`.
- ❖ Статичне поле єдине для всіх екземплярів класу, не входить фізично до складу жодного з об'єктів класу.
- ❖ Доступ — через назву класу.
- ❖ Альтернатива глобальним об'єктам.

Статичні методи класу

```
class Box {
private:
    static int objectCount;
public:
    // Constructor definition
    Box(double l = 2.0, double b = 2.0, double h = 2.0) {
        cout << "Constructor called." << endl;
        length = l;
        breadth = b;
        height = h;

        // Increase every time object is created
        objectCount++;
    }

    double Volume() {
        return length * breadth * height;
    }

    static int getCount() {
        return objectCount;
    }
private:
    double length;
    double breadth;
    double height;
};
```

- ❖ Статичні методи не прив'язані до об'єкту та не мають прихованого вказівника this!
- ❖ Статичні методи можуть напряму звертатися до інших статичних членів (змінних або функцій), але не можуть напряму звертатися до нестатичних членів. Це пов'язано з тим, що нестатичні члени належать об'єкту класу, а статичні методи — ні!

```
// Initialize static member of class Box
int Box::objectCount = 0;
```

```
int main(void) {
    Box box1(3.3, 1.2, 1.5);    // Declare box1
    Box box2(8.5, 6.0, 2.0);    // Declare box2

    Box* arr = new Box[10];

    // Print total number of objects.
    cout << "Total objects: " << Box::getCount() << endl;

    delete[] arr;
    return 0;
}
```

Селектори, модифікатори

```
class Rectangle {
private:
    double length;
    double width;

public:
    Rectangle(double l, double w) : length(l), width(w) {}

    // Selector (Getter) for length
    double getLength() const {
        return length;
    }
    // Selector (Getter) for width
    double getWidth() const {
        return width;
    }
    // Mutator (Setter) for length
    void setLength(double l) {
        if (l > 0) { // Ensure positive length
            length = l;
        }
        else {
            std::cout << "Invalid length. Must be positive." << std::endl;
        }
    }
    // Mutator (Setter) for width
    void setWidth(double w) {
        if (w > 0) { // Ensure positive width
            width = w;
        }
        else {
            std::cout << "Invalid width. Must be positive." << std::endl;
        }
    }
    // Method to calculate area
    double calculateArea() const {
        return length * width;
    }
};
```

- ❖ **Селектор** – метод, визначений зі специфікатором `const`, який забороняє змінювати об'єкт, що його викликає. Можна викликати як через константний, так і через неконстантний об'єкти.
- ❖ **Модифікатор** – метод, що дозволяє (і передбачає) зміну об'єкта, який його викликає. Можна викликати лише через неконстантний об'єкт.

Деструктор класу



FACULTY OF APPLIED
MATHEMATICS AND
INFORMATICS
LVIV UNIVERSITY

Деструктори

- ❖ **Деструктор** — це спеціальний тип методу класу, який виконується при видаленні об'єкта класу. У той час як конструктори призначені для ініціалізації класу, деструктори призначені для очищення пам'яті після нього.
- ❖ Коли об'єкт автоматично виходить з області видимості або динамічно виділений об'єкт явно видаляється за допомогою ключового слова `delete`, викликається деструктор класу (якщо він існує). Для простих класів (тих, які тільки ініціалізують значення базових типів змінних-членів) деструктор не потрібен, так як C++ автоматично виконує очищення самостійно.
- ❖ **Деструктор необхідно визначати для звільнення пам'яті у випадку якщо об'єкт класу містить динамічно виділені ресурси (in heap).**

Визначення деструктора в класі

- ❖ Деструктор повинен мати те ж ім'я, що і клас, зі знаком тильда (~) на самому початку;
- ❖ Деструктор не може приймати аргументи;
- ❖ Деструктор не має типу повернення.

```
class Point {  
private:  
    int x;  
    int y;  
public:  
    Point() : x(0), y(0){ }  
    ~Point()  
    {  
        static int counter = 0;  
        cout << "Point destructor was invoked:" << ++counter << endl;  
    }  
};  
int main() {  
  
    Point p;  
    Point* p2 = new Point();  
    Point mas[5];  
    Point* mas2 = new Point[5];  
  
    cout << "delete p2: " << endl;  
    delete p2;  
  
    cout << "delete mas2: " << endl;  
    delete[] mas2;  
  
    cout << "main() exit: " << endl;  
}
```

Деструктор

```
delete p2:  
Point destructor was invoked:1  
delete mas2:  
Point destructor was invoked:2  
Point destructor was invoked:3  
Point destructor was invoked:4  
Point destructor was invoked:5  
Point destructor was invoked:6  
main() exit:  
Point destructor was invoked:7  
Point destructor was invoked:8  
Point destructor was invoked:9  
Point destructor was invoked:10  
Point destructor was invoked:11  
Point destructor was invoked:12
```

Деструктор **private**

Приватний деструктор може використовуватись для заборони видалення об'єкту користувачем. Об'єкт може видалити лише сам себе або видалений дружнім класом або функцією.

Приватні конструктори та деструктори є досить корисними під час реалізації патерну проектування [фабрика](#). Об'єкти, як правило, створюються/видаляються статичним учасником або другом класу.

Object is created on Heap

```
class PrivateCD
{
public:
    static PrivateCD* create(/* args */) // Factory method
    {
        return new PrivateCD(/* args */);
    }
    static void destroy(PrivateCD* ptr) // Factory method
    {
        delete ptr;
    }
private:
    PrivateCD(/* args */) {
        cout << "Private constructor was invoked" << endl;
    }
    ~PrivateCD() {
        cout << "Private destructor was invoked" << endl;
    }
};

int main()
{
    //PrivateCD m;           // error: ctor and dtor are private
    //PrivateCD* mp = new PrivateCD(); // error: private ctor
    PrivateCD* mp = PrivateCD::create(); // OK
    //delete mp;             // error: private dtor
    PrivateCD::destroy(mp);  // OK
}
```

Object is created on Stack

```
class PrivateCD
{
private:
    PrivateCD(int i) : _i(i) {
        cout << "Private constructor was invoked" << endl;
    };
    ~PrivateCD() {
        cout << "Private destructor was invoked" << endl;
    };
    int _i;
public:
    static void TryMe(int i)
    {
        PrivateCD p(i);
        cout << "inside PrivateCD::TryMe, p._i = " << p._i << endl;
    };
};

int main()
{
    PrivateCD::TryMe(8);
}
```

Правило трьох (C++)

Правило трьох («Закон великої трійки») — практичне правило в C++, яке каже, що якщо в класі визначений один з таких методів, то, найпевніше, в ньому мають бути визначені всі три:

- ❖ Деструктор (англ. destructor)
- ❖ Конструктор копіювання (англ. copy constructor)
- ❖ Оператор присвоєння копіюванням (англ. copy assignment operator)

Ці три особливі функції-члени, можуть автоматично створюватися компілятором у випадку, якщо програміст не визначив їх явно (насправді, це не завжди так, наприклад, якщо один з атрибутів класу константа і, таким чином, вимагає явної ініціалізації). Якщо один з них має бути визначений програмістом, значить версія створена компілятором не підходить, а це означає, що, найпевніше, версії компілятора не підходять і для двох інших функцій.

Дякую!



FACULTY OF APPLIED
MATHEMATICS AND
INFORMATICS
LVIV UNIVERSITY