

# Лекція 2

## Об'єктно-орієнтоване програмування

Лектор: *Розова Людмила Вікторівна*

# План лекції 2

- 1 Типи конструкторів класу
- 2 Показчик this
- 3 Перевантаження функцій
- 4 Перевантаження операцій

Матеріали курсу:

<https://github.com/LRozova/oop1>

# Типи конструкторів класу

**Конструктор** – метод, який викликається при створенні об'єкта класу

✓ **Конструктор за замовчуванням.** При відсутності в класі будь-якого конструктора створюється компілятором

```
class Person
{
    string Name;
    int Age;
public:
    Person() {cout<<"Створено об'єкт класу Person";}
...
    Person obj1;
```

✓ **Конструктором з параметрами.** Передача початкових значень полям класу

```
Person(string n, int a) {Name = n; Age = a; }
```

У тому числі з параметрами за замовчуванням

```
Person(string n="Human", int a)
{ Name = n; Age = a; }
```

...

```
Person obj2("Bill", 20), obj3(22);
```

## ✓ Конструктор зі списком ініціалізації

```
Person(string n, int a): Name(n), Age(a)
{ cout<<"Конструктор зі списком ініціалізації"; }
Person("Mike", 30);
```

## ✓ Делегуючі конструктори

```
Person(string n)
{ Name = n; }
Person(int a): Person(string n)
{ Age=a; }
```

```
Person obj("John"); //викликається перший конструктор
//Person(string n);
```

```
Person obj("John",40); //викликається спочатку перший
конструктор, потім другий Person(int a);
```

## ✓ Конструктор копіювання

- Коли один об'єкт класу ініціалізується іншим (створюється копія об'єкту).
- Коли об'єкт передається в будь-яку функцію у якості параметра;
- Коли будь-яка функція повинна повернути об'єкт класу у результаті своєї роботи

class **Dot**

{ int \*ptr;

public:

**Dot()** // конструктор за замовчуванням

{

cout << "Звичайний покажчик"<<endl;

}

**Dot(const Dot &obj)** // конструктор копіювання

{

cout << "Конструктор копіювання"<<endl;

}

**~Dot()**

{ cout << "\nДеструктор\n";}

};

```
void funcShow(Dot object)
```

```
{ cout << “Функція приймає об’єкт класу, як параметр”<<endl;  
}
```

```
Dot funcReturnObject()
```

```
{ Dot object;  
  cout << “Функція повертає об’єкт класу”<<endl;  
  return object;  
}
```

```
int main()
```

```
{  
  Dot obj1; //створюємо об’єкт класу  
  funcShow(obj1); // передаємо об’єкт у якості параметра у функцію  
  funcReturnObject(); //функція повертає об’єкт  
  Dot obj2 = obj1; // ініціалізація об’єкта іншим при створенні  
}
```

# Показчик this

**Показчик *this*** — це показчик на поточний об'єкт даного класу, який викликає метод класу. Це неявний параметр кожного методу класу (окрім статичних).

Наприклад, `this->x`

В будь-якому методі показчик *this* можна використовувати наявно для посилання на об'єкт, що його викликає

```
class Point
```

```
{ int x,y;
```

```
  public:
```

```
    Point(int xx=0, int yy=0) {x=xx; y=yy;}
```

```
    int getX() {return this->x;} 
```

```
    int getY() {return this->y;} 
```

```
    Point &move(int x, int y)//якщо параметри мають таке ж
```

```
// ім'я, як і дані класу
```

```
    { this->x += x;
```

```
      this->y += y;
```

```
      return *this; }
```

```
};
```

```
int main()  
{  
    Point p1(20, 50);  
    p1.move(10, 5).move(10, 10); // викликає по чергово  
// метод move для об'єкту (переміщення координат об'єкту  
// Point.)  
// Це стає можливим завдяки тому, що метод move повертає  
// посилання на об'єкт, а не копіює його  
    p1.showCoords(); // x: 40   y: 65  
  
    return 0;  
}
```



# Основні принципи ООП

- **Інкапсуляція** – об'єднання даних і методів у єдине ціле (class);
- **Успадкування** – створення нових класів на базі існуючих
- **Поліморфізм** – «один інтерфейс, багато методів»



## Клас «Електрична кавомолка»:

Успадкування: створення класу «Електрична кавомолка» на базі класу «Механічна кавомолка»

Поліморфізм: перевизначення основних функцій роботи з кавомолкою в класі «Електрична кавомолка»

Інкапсуляція: приховування всієї начинки кавомолки від користувача. Лише взаємодія по інструкції

# Перевантаження функцій

**Перевантаження функцій** - це можливість визначати декілька функцій з одним і тим же ім'ям, але з різними параметрами.  
Реалізація принципу поліморфізму

```
int add(int a, int b)
```

```
{ return a + b; }
```

```
float add(float a, float b) //повинні відрізнятися типом параметрів
```

```
{ return a + b; }
```

```
int add(int a, int b, int c) //та кількістю параметрів
```

```
{ return a + b + c; }
```

Компілятор не буде відрізняти функції по типу повертаємого значення, тільки по параметрам функцій

```
int getRandomValue();
```

```
double getRandomValue();//однакові для компілятора
```

Якщо в класі декілька конструкторів – вони перевантаженні

# Дружні функції

**Дружня функція** - це функція, яка не є членом класу, але має доступ до членів класу, оголошеним в полях **private** або **protected**.

Дружня функція оголошується всередині класу, до елементів якого їй потрібен доступ, з ключовим словом **friend**.

Дружня функція може бути звичайною функцією або методом іншого раніше визначеного класу. Одна функція може бути дружньою відразу декільком класами.

```
class myclass {  
    int a, b;  
    public:  
        myclass(int i, int j) { a=i; b=j; }  
        friend int sum(myclass x);  
};  
int sum(myclass x)  
{  
    return x.a + x.b;  
}
```

В одному класі можуть бути визначені декілька дружніх функцій

Дружні функції можуть бути оголошені в іншому класі:

```
class Car; //прототип класу
```

```
class Man
```

```
{public:
```

```
    Man(string n)
```

```
    { name = n; }
```

```
    void driveCar(Car &a); // повертає назву авто, яким керує людина
```

```
    void setPrice(Car &a, int price); // встановлює ціну для авто
```

```
private:
```

```
    string name; // ім'я
```

```
};
```

```
class Car
```

```
{ friend void Man::driveCar(Car &);
```

```
    friend void Man::setPrice(Car &, int price);
```

```
public:
```

```
    Car (string carName, int carPrice)
```

```
    {name = carName; price = carPrice; }
```

```
private:
```

```
    string name; // назва авто
```

```
    int price; // ціна авто
```

```
};
```

# Дружні класи

**Дружні класи** - це класи, які мають доступ до всіх закритих членів класу, до якого вони є друзями.

```
class Car
```

```
{ friend class Man
```

```
.....
```

```
};
```

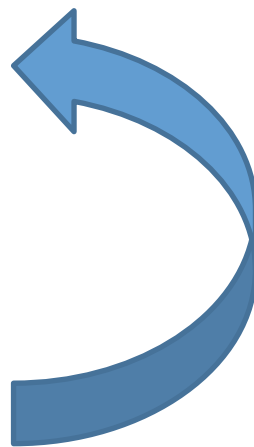
```
class Man
```

```
{.....
```

```
void driveCar(Car &a);
```

```
void setPrice(Car &a, int price);
```

```
};
```



**!!!** Але **class Car**, при цьому, не є другом **class Man**, тому не має доступу до його приватних елементів

# Перевантаження операцій

**Перевантаження операторів (operator overloading)** - це можливість застосовувати вбудовані оператори мови до різних типів, в тому числі і створених користувачем.

Переваги цього: Перевантаження операцій надає можливість використовувати власні типи даних як стандартні, це перетворює текст програми на інтуїтивно зрозумілий

➤ !Позначення власних операцій вводити не можна

**Перевантаження операцій здійснюється за допомогою методів спеціальної форми «функцій-операцій» за такими правилами:**

- при перевантаженні операцій зберігаються кількість аргументів, пріоритети операцій та правила асоціації (зліва направо чи справа наліво, як у стандартних типах даних);
- для стандартних типів даних перевизначати операції не можна;
- функції-операції не можуть мати аргументів за замовчуванням;
- функції-операції успадковуються (за винятком "=");
- функції-операції не можуть визначатися як static.

Функція-операція, що використовуються для перевантаження, містить ключове слово ***operator***, за яким слідує знак операції, яку треба перевизначити:

```
<тип> operator <операція> (<список параметрів>)  
{ <тіло функції> }
```

Функцію-операцію можна визначити трьома способами:

- ✓ як метод класу;
- ✓ як «дружню» функцію класу;
- ✓ як звичайну функцію.

У двох останніх випадках функція повинна мати хоча б один аргумент, який має тип класу, покажчик чи посилання на клас.

## Операції, які **можуть** бути перевантажені:

+   -   \*   /   %   ^   &   |   ~   !   =   <   >   +=   -=  
 \*=   /=   %=   ^=   &=   <<   >>   ==   !=   <=   >=  
 &&   ||   ++   --   ->   []   ()   new   delete

## Операції, які **НЕ** можуть бути перевантажені:

.   .\*   ?   :   ::   #   ##   sizeof

| Операція                         | Рекомендована форма перевантаження     |
|----------------------------------|--|
| Всі унарні операції              | Зовнішня функція/ friend / метод класу |
| =   []   ()   ->                 | Метод класу                            |
| +=   -=   *=   /=   %=   &=   ^= | Метод класу                            |
| Інші бінарні операції            | Зовнішня функція/ friend / метод класу |
| <<   >>                          | Зовнішня функція / friend              |



# Перевантаження операторів з використанням методів класу

Можливо перевантажувати як унарні операції,  
наприклад, ++, --, -, +

Так і бінарні оператори:

наприклад, +, -, =, \*, /

Перевантаження унарної операції методом класу:

- функція-оператор не має параметрів;
- операція виконується над об'єктом, який генерує виклик цього методу через неявно переданий покажчик *this*.

Розглянемо перевантаження унарних операцій на прикладі операція **інкременту (префіксна форма)**.

```
class Point
{
    int x, y, z;
public:
    ...
    Point operator++();
};
Point Point::operator++()
{
    ++x; ++y; ++z; //інкремент координат x, y, z
    return *this; //повертання значення
}
...
Point a(1, 2, 3);
++a;
```

## Операція інкременту (постфіксна форма).

```
class Point
```

```
{
```

```
    int x, y, z;
```

```
public:
```

```
    Point operator++(int notused) ;
```

```
...};
```

```
Point Point::operator++(int notused)
```

```
{Point temp = *this; //збереження вихідного значення
```

```
    x++; //інкремент координат x, y, z
```

```
    y++; z++;
```

```
    return temp; //повертання вихідного значення
```

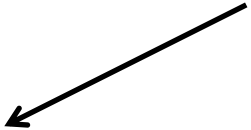
```
}
```

```
...
```

```
Point a(1, 2, 3);
```

```
a++;
```

Вказівка компілятора,  
що це постфіксна форма



## Перевантаження бінарних операцій

- В цьому випадку функція-оператор має тільки один параметр.
- Інший параметр передається неявним чином за допомогою покажчика *this* на об'єкт, для якого викликається функція-оператор.
- Об'єкт, що знаходиться справа від знаку операції, передається методу як параметр

Розглянемо на прикладі *бінарної операції* +

```
class Point
{   int x,y,z;
public:
    Point operator+ (Point op2);
};
Point Point::operator+(Point op2)
{
    Point temp;
    temp.x=x+op2.x;
    temp.y=y+op2.y;
    temp.z=z+op2.z;
    return temp;
}
.....
Point a(1,2,3), b(2,3,4), c;
c=a+b;...
```

x,y,z відповідають this->x, this->y, this->z для об'єкту, що викликає операцію (який зліва)

Розглянемо на прикладі *бінарної операції* >

```
class Person
{
    string Name;
    int age;
public:
    ...

    bool operator >(const Person &Man) {
        if( age > Man.age() )
            return true;
        return true; }

};

...
Person M1 ("Ivan", 25), M2 ("Petro", 30);
if (M1>M2) ...//порівняння за полем age
```

Дякую за увагу!