

Лекція 2

Об'єктно-орієнтоване програмування

Лектор: *Розова Людмила Вікторівна*

План лекції 2

- 1 Конструктор класу. Типи конструкторів
- 2 Деструктор класу
- 3 Показчик this
- 4 Дружні функції. Дружні класи
- 5 Перевантаження операторів та операцій

Матеріали курсу:

https://github.com/LRozova/OOP_ukr_2022

Повторимо!

- ✓ Основні принципи ООП:
інкапсуляція, поліморфізм, успадкування.
- ✓ Інкапсуляція та абстракція при створенні класу.
- ✓ Клас: дані + методи
- ✓ Режим доступу до елементів класу:
private, public, protected
- ✓ Клас та його об'єкти
- ✓ Сетери, гетери класів
- ✓ Многофайлові проєкти.

Конструктор класу

Конструктор - метод класу, який служить **для створення і ініціалізації екземпляра** класу (об'єкта). Ім'я конструктора завжди збігається з ім'ям класу

- ✓ **Ім'я конструктора – це ім'я класу.** Отже, компілятор відрізняє конструктор від інших методів класу

`Employee () ;`

- ✓ Виконується автоматично в момент створення об'єкта
- ✓ Для створення екземпляра класу потрібно, щоб конструктор був методом типу *public*
- ✓ Конструктор не має типу, не повертає жодного значення, навіть типу `void`
- ✓ **Клас може мати кілька конструкторів** з різними параметрами для різних видів ініціалізації
- ✓ Конструктори не успадковуються

Деструктор класу

Деструктор - метод класу, який служить для видалення екземпляра класу.

- ✓ **Ім'я деструктора** складається з символу ~ (тильда) і імені класу

```
~Employee ();
```

- ✓ Деструктор не має аргументів і значення, що повертається.
- ✓ Деструктор викликається автоматично, коли об'єкт виходить з області видимості:
 - для локальних об'єктів - при виході з блоку, в якому вони оголошені;
 - для глобальних - як частина процедури виходу з main;
 - для об'єктів, заданих через покажчики, деструктор викликається неявно при використанні операції delete.

```
//Файл CreateAndDestroy.h
class CreateAndDestroy
{
public:
    CreateAndDestroy(int); // Конструктор
    ~CreateAndDestroy(); // Деструктор

private:
    int data;
};
//-----Конец файла-----

//Файл CreateAndDestroy.cpp
#include <iostream>
using namespace std;
CreateAndDestroy::CreateAndDestroy(int value)
{
    data = value;
    cout << "Object " << data << " constructor" << endl;
}
CreateAndDestroy::~~CreateAndDestroy()
{
    cout << "Object " << data << " destructor" << endl;
}
//-----Кінець файлу-----
```

```
// Файл main.cpp
#include <iostream>
#include "CreateAndDestroy.h"
using namespace std;
CreateAndDestroy first(1);
int main()
{   cout << "main: Hello" << endl;
    CreateAndDestroy second(2);
    cout << "main: second created"<< endl;
    CreateAndDestroy* third = new CreateAndDestroy(3);
    cout << "main: third created"<< endl;
    delete third;
    cout << "main: third deleted" << endl;
    cout << "Exit from main" << endl;
    return 0;
}
```

```
Object 1 constructor
main: Hello
Object 2 constructor
main: second created
Object 3 constructor
main: third created
Object 3 destructor
main: third deleted
Exit from main
Object 2 destructor
Object 1 destructor
```

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, int a=30)
{ Name = n; Age = a; }
```

```
...
Person obj2("Bill", 20), obj3("Tom");
```


✓ Конструктор зі списком ініціалізації

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

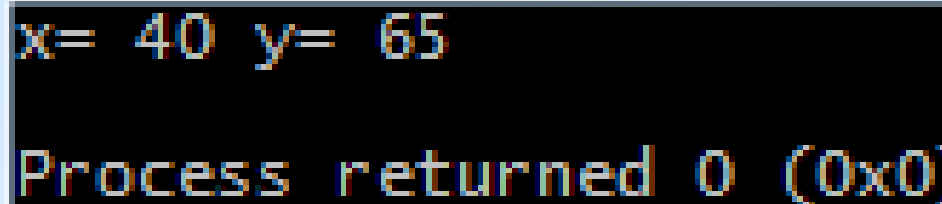
```
      this->y += y;
```

```
      return *this; }
```

```
};
```

```
int main()
{
    Point p1(20, 50);
    p1.move(10, 5).move(10, 10);//викликає по чергово
// метод move для об'єкту
// відбувається переміщення координат об'єкту Point
// Це стає можливим завдяки тому, що метод move повертає
// посилання на об'єкт, а не копіює його
    cout<<"x="<<p1.getX()<<" y= "<<p1.getY()<<endl;

    return 0;
}
```



```
x= 40 y= 65
Process returned 0 (0x0)
```

Перевантаження функцій

Перевантаження функцій - це можливість визначати декілька функцій з одним і тим же ім'ям, але з різними параметрами. Це один з проявів 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 перевантажені

Дружні функції

Дружня функція - це функція, яка не є членом класу, але має доступ до членів класу, оголошених 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. **Не має покажчик this!**

```
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: Приклад

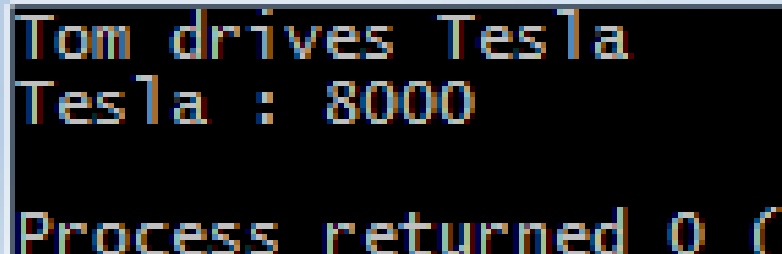
```
class Car; //прототип класу
class Man
{public:
    Man(string n)
    { name = n; }
    void driveCar(Car &a); // повертає назву авто, яким керує людина
    void setPrice(Car &a, int price); // встановлює ціну для авто
private:
    string name; // ім'я
};

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
};
```



```
void Man::driveCar(Car &a)
{
    cout << name << " drives " << a.name << endl;
}
void Man::setPrice(Car &a, int price)
{
    if (price > 0)
        a.price = price;
}
int main()
{
    Car tesla("Tesla", 5000);
    Man tom("Tom");
    tom.driveCar(tesla);
    tom.setPrice(tesla, 8000);
    cout << tesla.getName() << " : " << tesla.getPrice() << endl;

    return 0;
}
```



```
C>
Tom drives Tesla
Tesla : 8000
Process returned 0 (0 minutes, 0 seconds)
```

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 операції **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
```

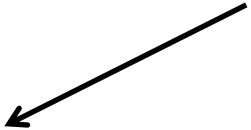
```
}
```

```
...
```

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

Дякую за увагу!