

Лекція 5 (2023)

Об'єктно-орієнтоване програмування

Лектор: *Розова Людмила Вікторівна*

План лекції 5

- 1 Конструктори, передача параметрів при успадкуванні
- 2 Просте, множинне успадкування
- 3 Поліморфізм. Віртуальні функції

Матеріали курсу:

https://github.com/LRozova/OOP_ukr_2023

Передача параметрів конструкторам базового класу

Якщо необхідно передати параметри конструктору **базового класу**, використовується розширена форма оголошення конструктора **похідного класу**, в якій передбачена можливість передачі аргументів одному чи декільком конструкторам базового класу.

```
Конструктор_похідного_класу(список_аргументів)  
: конструктор_базового_класу(список_аргументів)  
{  
тіло конструктора похідного класу  
}
```

! Якщо базовий клас містить тільки конструктори з параметрами, то похідний клас має викликати в своєму конструкторі один з конструкторів базового класу.

```
#include <iostream>
using namespace std;
class base
{protected:
    int i;
public:
    base (int x)
    { i = x;
      cout << "Створення base-об'єкту.\n";
    }
    ~base() { cout<<"Знищення base-
об'єкту.\n"; }
};
```

```
class derived: public base
```

```
{ int j;
```

```
public:
```

```
// Клас derived використовує параметр x, а параметр y
```

```
//конструктору класу base.
```

```
derived(int x, int y): base(y)
```

```
{ j = x; cout << "Створення derived1-  
об'єкту.\n"; }
```

```
~derived()
```

```
{ cout <<"Знищення derived1- об'єкту.\n"; }
```

```
void show() { cout <<i<< " " << j << "\n"; }
```

```
};
```

```
int main()
```

```
{ derived ob(3, 4);
```

```
ob.show();
```

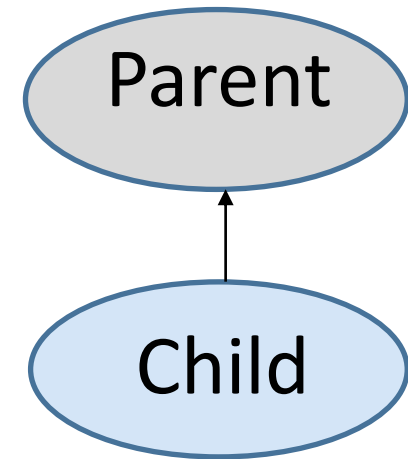
```
return 0;
```

```
}
```

```
Створення base-об'єкту.  
Створення derived1-об'єкту.  
4 3  
Знищення derived1- об'єкту.  
Знищення base- об'єкту.
```

Приклад простого успадкування

```
#include <iostream>
#include <string>
using namespace std;
class Parent //базовий клас
{
protected: //захищені елементи
    int age;
    string name;
    int getAge() {return age;}
public:
    Parent(string N,int A)
    {name=N; age=A;
    }
    void identify() { cout << "I am a Parent: «
        <<name<<" I am: "<<age<<endl; }
};
```



```
class Child : public Parent
```

```
{private:
```

```
    string ch_name;
```

```
    int ch_age;
```

```
public:
```

```
    Child(string CN, int CA, string N, int A)
```

```
        : Parent(N,A)
```

```
    {ch_name=CN; ch_age=CA; }
```

//заміщення або розширення методів базового класу

```
void identify()
```

```
    {Parent::identify(); // спочатку виконається для
```

// класу Parent

```
    cout <<"And I am a Child: " <<ch_name<<
```

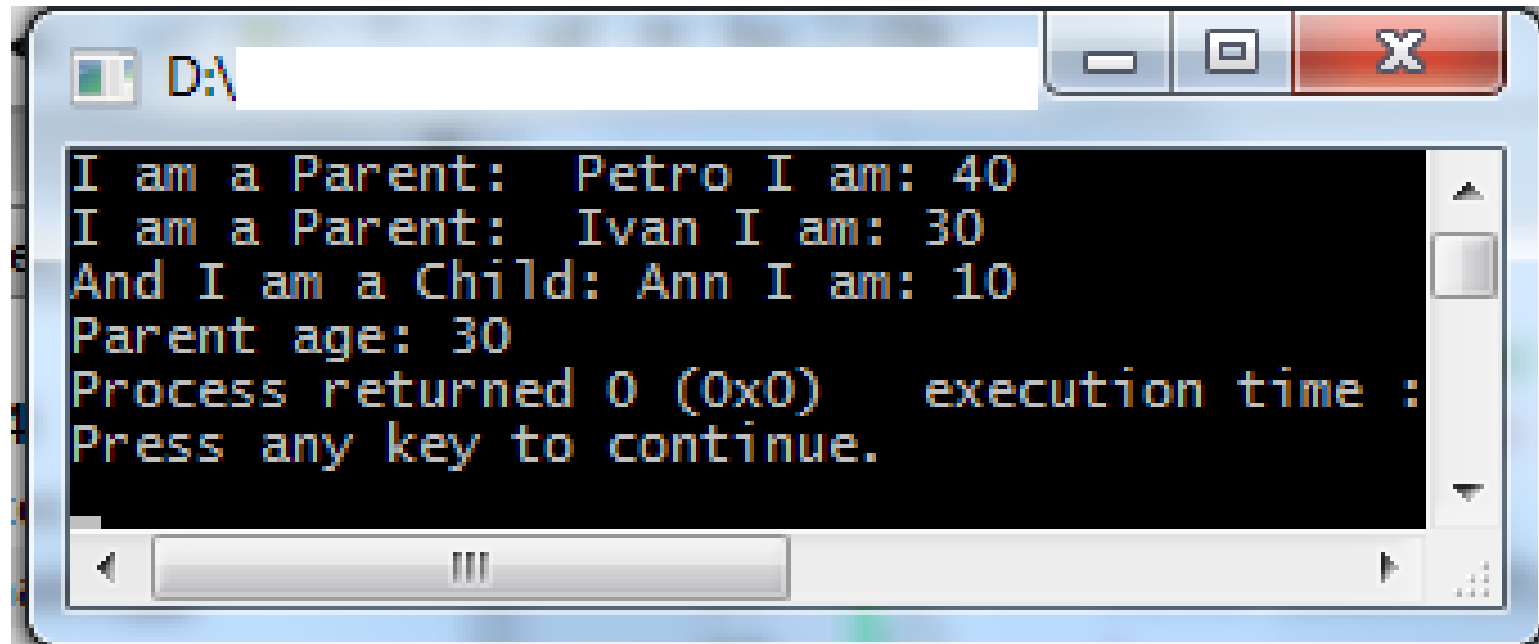
```
    "I am: " <<ch_age<<endl; // потім цей текст
```

```
    cout <<"Parent age: "
```

```
        <<getAge(); } // викликається для класу Parent
```

```
};
```

```
int main()  
{  
    Parent parent("Petro", 40);  
    parent.identify();  
    Child child("Ann", 10, "Ivan", 30);  
    child.identify();  
    return 0;  
}
```



```
I am a Parent: Petro I am: 40  
I am a Parent: Ivan I am: 30  
And I am a Child: Ann I am: 10  
Parent age: 30  
Process returned 0 (0x0) execution time :  
Press any key to continue.
```


Принцип підстановки

Успадкування встановлює між класами **відношення «є»**:
похідний клас є частиною базового класу

Усюди, де може бути використаний об'єкт базового класу

-при присвоюванні,

-при передачі параметрів ,

-при поверненні результату

замість нього можна використовувати об'єкт похідного класу.

Це положення має назву **«принцип підстановки»**.

При цьому зворотне невірно:

Наприклад,

будь-який **студент** (похідний) є **людиною** (базовий клас),

але

будь-яка **людина** (базовий) **НЕ** є **студентом** (похідний)

```
class X {  
    {protected:  
        int x;  
    public:  
        X(int i) // Конструктор X  
        {x=i;  cout<< "Constructor X \n";}  
        ~X() //Деструктор X  
        {cout<< "Destructor X \n"; }  
        void show ()  
        {cout<< " x =" << x << "\n";}  
};  
  
class Y {  
    protected:  
        int y;  
    public:  
        Y (int j) // Конструктор Y  
        {y=j;  cout<< "Constructor Y \n"; }
```

Базовий клас X



Базовий клас Y



Похідний клас Z

`~Y () //Деструктор Y`

```
{cout<< "Destructor Y \n";}
void show ()
{cout<< " y = "<< y << "\n";}
};
```

`//похідний клас від X і Y`

class Z: public X, public Y

{protected:

int z;

public:

Z (int i, int j):X(i), Y(j)

`// Конструктор класу Z передає параметри конструкторам X і Y`

```
{cout<< "Constructor Z \n";}
```

`~Z () //Деструктор Z`

```
{cout<< "Destructor Z \n";}
```

void show ()

```
{cout<< x*y<< " ="<< x << "*" <<y<< "; \n";}
```

```
};
```

```

int main()
{
Z zobj (3, 5); //Створення об'єкта класу Z та передача
//параметрів конструкторам класів X і Y
zobj.show(); //Виклик методу show() класу Z
zobj.X::show(); //Виклик методу show() класу X
zobj.Y::show(); //Виклик методу show() класу Y
return 0;
}

```

Увага!

Успадкування надає безліч переваг, але має бути ретельно спроектовано щоб уникнути проблем, можливість для яких воно відкриває.

```

D:\
Constructor X
Constructor Y
Constructor Z
15 =3*5;
  x =3
  y = 5
Destructor Z
Destructor Y
Destructor X

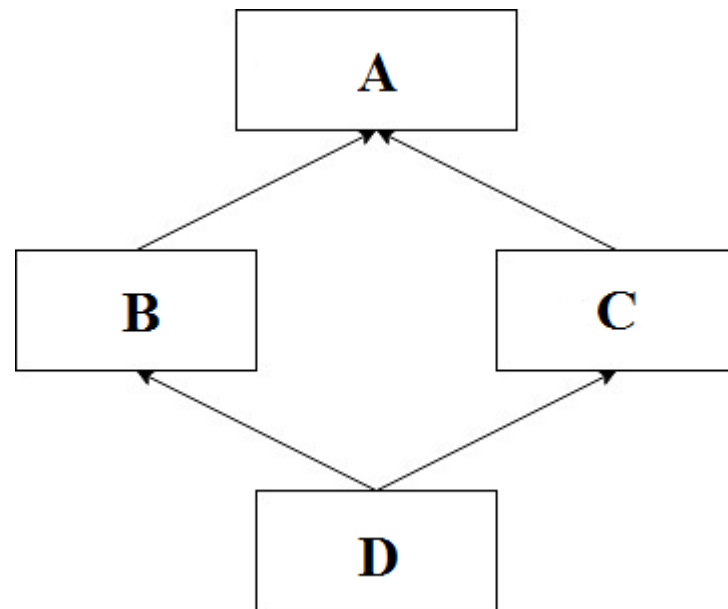
Process returned 0 (0x0)
Press any key to continue.

```

Проблеми, які можуть виникнути при множинному успадкуванні

- Виникнення неоднозначності, коли кілька батьківських класів мають метод з одним і тим же ім'ям, а в дочірньому класі цей метод не перевизначений.

- «Проблема ромба»



Поліморфізм

Поліморфізм (от греч. *polýs* — много и *morphé* — форма) означає можливість приймати різноманітні форми, зберігаючи суть, або одночасно приймати різноманітні форми.

Поліморфізм в програмуванні проявляється, наприклад, в перевантаженні функцій, операторів і операцій.

При **успадкуванні** поліморфізм — можливість об'єктів різних класів, що пов'язані відносинами успадкування, реагувати по різному під час виклику одного й того-самого методу.

«Один інтерфейс – багато методів»

Віртуальні функції

Реалізація динамічного *поліморфізму* в C++ здійснюється завдяки поєднанню успадкування і **віртуальних функцій (методів)** — функцій, що оголошуються в базовому класі з використанням ключового слова *virtual* і перевизначаються в одному або декількох похідних класах. Таким чином, кожний похідний клас може мати власну версію віртуальної функції.

```
virtual    тип    ім'я_функції    (параметри)  
{  
}
```

Розглянемо приклад 1

```
#include <iostream>
using namespace std;
class A
{ public:
    void what() // оголошення функції what()
    { cout << "BASE class\n "; }
};
class A1 : public A
{ public:
    void what() // перевизначення функції what() для класу A1
    { cout << "1_DERIVED class\n "; }
};
class A2 : public A1
{ public:
    void what() // перевизначення функції what() для класу A2
    { cout << "2_DERIVED class\n "; }
};
```



```

int main() {
    A base_object;
    A *p;
    A1 a1_object;
    A2 a2_object;
    p = &base_object; // встановлюємо покажчик на об'єкт
// базового класу
    p->what(); //викликаємо метод who()
    p = &a1_object; // встановлюємо покажчик на об'єкт
// класу A1
    p->what(); //викликаємо метод who()
    p = &a2_object; // встановлюємо покажчик на об'єкт
// класу A2
    p->what(); //викликаємо метод who()
    return 0;
}

```

```

BASE class
BASE class
BASE class

Process returned 0 (0x0)   executi
Press any key to continue.

```

Розглянемо приклад 2

```
#include <iostream>
using namespace std;
class A
{ public:
    virtual void what() // оголошення віртуальної функції
    { cout << "BASE class\n "; }
};
class A1 : public A
{ public:
    //перевизначення функції what() для класу A1 слово virtual не обов'язкове
    void what() //override
    { cout << "1_DERIVED class\n "; }
};
class A2 : public A1
{ public:
    void what() // перевизначення функції what() для класу A2
    { cout << "2_DERIVED class\n "; }
};
```

```

int main() {
    A base_object;
    A *p;
    A1 a1_object;
    A2 a2_object;
    p = &base_object; // встановлюємо покажчик на об'єкт
// базового класу
    p->what(); //викликаємо метод who() для класу A
    p = &a1_object; // встановлюємо покажчик на об'єкт
// класу A1
    p->what(); //викликаємо метод who() для класу A1
    p = &a2_object; // встановлюємо покажчик на об'єкт
// класу A2
    p->what(); //викликаємо метод who() для класу A2
    return 0;
}

```

```

BASE class
1_DERIVED class
2_DERIVED class

Process returned 0 (0x0)
Press any key to continue

```

- Важливим моментом забезпечення ідеї поліморфізму є те, що звернення до віртуальної функції відбувається **через покажчик** базового класу, який використовується в якості посилання на об'єкт похідного класу.
- У такому випадку компілятор C++ автоматично визначає, яку саме версію віртуальної функції (методу) потрібно викликати, по типу об'єкта, що адресується цим покажчиком, такий вибір відбувається під час виконання програми. При цьому викликається *«найбільш дочірній метод»*.
- За наявності кількох похідних класів при посиланні покажчика базового класу на різні об'єкти цих похідних класів будуть виконуватися різні версії віртуальної функції (метода).
- Також віртуальну функцію можна викликати як будь-яку іншу компонентну функцію.
- **Поліморфний клас** — клас, що включає віртуальну функцію

Правила оголошення та використання віртуальних функцій-методів

21

1. Віртуальна функція може бути тільки методом класу.
2. Кількість і тип параметрів віртуальних функцій у базовому та похідних класах повинні точно збігатися і мати однакові прототипи, на відміну від перевантажених функцій.
3. Віртуальна функція успадковується.
4. Будь-який перевантажений метод класу можна зробити віртуальним, наприклад перевантаження операцій.
5. Статичні методи не можуть бути віртуальними.
6. Конструктори не можуть бути віртуальними.
7. Деструктори можуть (частіше - повинні) бути віртуальними - це гарантує коректне звільнення пам'яті через покажчик базового класу.

Дякую за увагу!