

# Наследяване

## Същност на наследяването - идеи и приложения

Нека разгледаме следните класове:

```
class Car
{
    private:
        char* maker;
        char* model;

        int horsepower;
        int seats;

    public:
        //Big 4
        void set_maker(const char*);
        void set_model(const char*);
        void set_horsepower(const int);
        void set_seat(const int);

        const char* get_maker() const;
        const char* get_model() const;
        const int get_horsepower() const;
        const int get_seats() const;
};
```

```
class Motorcycle
{
    private:
        char* maker;
        char* model;

        int horsepower;
        int capacity_luggage;

    public:
        //Big 4
        void set_maker(const char*);
        void set_model(const char*);
        void set_horsepower(const int);
        void set_capacity(const int);

        const char* get_maker() const;
        const char* get_model() const;
        const int get_horsepower() const;
```

```
const int get_capacity() const;
};
```

Забелязваме, че и двата класа имат член-данните `maker`, `model` и `horsepower` и съответните методи, но `Car` има `seats`, а `Motorcycle` има `capacity_luggage`. Тук изниква въпроса - възможно ли е да отделим общите части на двата класа и да ги използваме, без да се налага да ги пишем всеки път наново ?

Един вариант е да се използва композиция - да се създаде клас `Vehicle` и инстанция на този клас да бъде член-данна на `Car` и `Motorcycle`.

Другият вариант е `Car` и `Motorcycle` да **наследяват** `Vehicle`. Така те ще получат член-данните и методите на `Vehicle` като свои, без да се налага да пишем два пъти.

## Композиция vs Наследяване ?

Кога е да използваме наследяване и кога да използваме композиция ?

Тук въпросът е, каква връзка искаме да създадем между отделните класове. Ако искаме връзката да е "is-a" (т.е. `Car` is a `Vehicle`) - използваме наследяване. Ако искаме връзката да е "has-a", (`Car` has a `Vehicle` няма особенна логика, но `Car` has a `SteeringWheel` има) - използваме композиция.

Също така, използваме наследяването, когато новият клас, който ще наследява нещо, да добавя нови елементи (член-данни или методи) или пък да променя действието на някои методи.

## Как се дефинират производните класове ?

Нека отделим общите член-данни и методи на `Car` и `Motorcycle` в нов клас, `Vehicle`:

```
class Vehicle
{
    private:
        char* maker;
        char* model;

        int horsepower;
    public:
        Vehicle();
        Vehicle(const Vehicle& object_to_copy_from);
        Vehicle& operator=(const Vehicle& object_to_copy_from);
        void set_maker(const char* new_maker);
        void set_model(const char* new_model);
        void set_horsepower(const int new_horsepower);
        ~Vehicle();
};
```

Класовете `Car` и `Motorcycle` ще изглеждат по този начин:

```
class Car : public Vehicle
{
    private:
```

```
    int seats;

    public:
    Car()
    {
        seats = 1;
    }
    void set_seat(const int);

    const int get_seats() const;

};
```

```
class Motorcycle : public Vehicle
{
    private:
    int capacity_luggage;

    public:
    Motorcycle()
    {
        capacity_luggage = 0;
    }
    void set_capacity(const int);

    const int get_capacity() const;

};
```

Така вече и двата класа `Car` и `Motorcycle` имат член-данните и методи на `Vehicle`

Възникват няколко въпроса:

## Защо `Car` и `Motorcycle` нямат голяма четворка ?

Самите класове `Car` и `Motorcycle` нямат динамична памет. При извикване на copy-constructor или operator=, компилатора извиква съответните copy-constructor или operator= на `Vehicle`. Същото се случва и при деструкторите. Не е нужен и конструктор по подразбиране, макар че е добра практика да има такъв, защото при инициализация на `Car`, се извиква и конструктора по подразбиране на `Vehicle`.

## Може ли даден клас да наследява два или повече класа ?

Да, това става по следният начин:

```
class A { /* ... */ };
class B { /* ... */ };

class C : A, B
{
    /* ... */
};
```

## Области на видимост и достъп

Забелязваме, че `Car` и `Motorcycle` наследяват `Vehicle` като `public`. Също както член-данните и методите, наследните класове могат да бъдат `public`, `protected` и `private`

Тип на наследяване\Област на видимост на компонента в основния клас	public	protected	private
public	public	protected	private
protected	protected	protected	private
private	private	private	private

Забелязваме следните няколко неща:

- Всички член-данни и методи, които са `private` в основния клас, винаги се наследяват като `private` (и ние нямаме директен достъп до тях).
- Ако наследним някой клас като `private` (което се случва ако изрично не напишем `protected` или `public` пред класа който ще наследяване), всички член-данни на родителския клас ще бъдат наследени като `private`
- `protected` ни позволява да тези член-данни да са видим за производните класове, но не и извън тях

## Достъп до член-данни и методи на наследен клас

Ако имаме достъп до дадени член-данни или методи (т.е. да са `public` или `protected` спрямо горната таблица), можем да ги използваме чрез `име_на_клас : член_данна` или `име_на_клас : метод()`

## Задача 1

Към класът `Vehicle` ще добавим метод `print()`, който извежда на конзолата марката, модела и конските сили на превозното средство. Да се добави метод `print()` и в `Car`, `Vehicle`, които да принтират цялата налична информация за превозните средства.

## Задача 2

Да се напише клас `Scooter` , който да наследява `Motorcycle` , но има само 30 конски сили. Да добавя нови член-данни - `max_capacity_of_driver` и `color` . Да се напишат подходящи методи.

Когато имаме наследяване на класове, голямата четворка не се наследява. Ще разгледаме всеки случай по отделно:

## Наследяване на конструктори

Когато наследим клас, без да кажем кой конструктор на наследения клас да се извика, винаги се извиква този по подразбиране. Ако искаме да извикаме конкретен конструктор, това става по следният начин:

```
Car::Car() : Vehicle()
{
    seats = 0;
}
```

Нека създадем конструктор с параметри на `Vehicle` , `Car` и `Motorcycle` :

```
Vehicle::Vehicle(const char* new_maker, const char* new_model, const int new_horsepower)
{
    maker = new char[strlen(object_to_copy_from.maker) + 1];
    strcpy(maker, object_to_copy_from.maker);

    model = new char[strlen(object_to_copy_from.model) + 1];
    strcpy(model, object_to_copy_from.model);

    horsepower = object_to_copy_from.horsepower;
}
```

Съответните конструктори с параметри на `Car` и `Motorcycle` ще изглеждат по следният начин:

```
Car::Car(const char* new_maker, const char* new_model, const int new_horsepower, const int new_seats) : Vehicle(new_maker, new_model, new_horsepower)
{
    seats = new_seats;
}
```

Ще разгледаме следните под-случаи:

Основен клас\Произведен клас	Има конструктор	Няма конструктор
Няма default-ен конструктор, има такъв с параметри	Трябва задължително да се извика конструктора с параметри	Грешка
Има няколко конструктора, включително и default-ен	Вика се default-ния, ако изрично не е посочен друг	Компилятора създава default-ен конструктор

Редът, по който се извикват конструкторите в основния и наследеният клас е следният - първо конструкторите на основните класове, после на производните .



```
lyubo@lyubo-Lenovo-Z710: /mnt/C044EDEE44EDE6DE/OOP_2019/WorkDir/bin$ ./main
Vehicle constructor called
Car constructor called
lyubo@lyubo-Lenovo-Z710: /mnt/C044EDEE44EDE6DE/OOP_2019/WorkDir/bin$
```

## Наследяване на сору конструктор

Правилата за обикновенните конструктори, също важат и за сору.

Ако искаме да извикаме сору конструктор на основен клас, това се случва по следният начин:

```
Car::Car(const Car &object_to_copy_from): Vehicle(object_to_copy_from)
{
    seats = object_to_copy_from.seats;
}
```

Отново имаме следните случаи:

Основен клас\Производен клас	Има сору конструктор	Няма сору конструктор
Няма сору конструктор	Използва се default-ен	Използват се default-ни от компилатора
Има сору конструктор	Трябва задължително да се извика конструктора с параметри	Компилатора създава default-ен

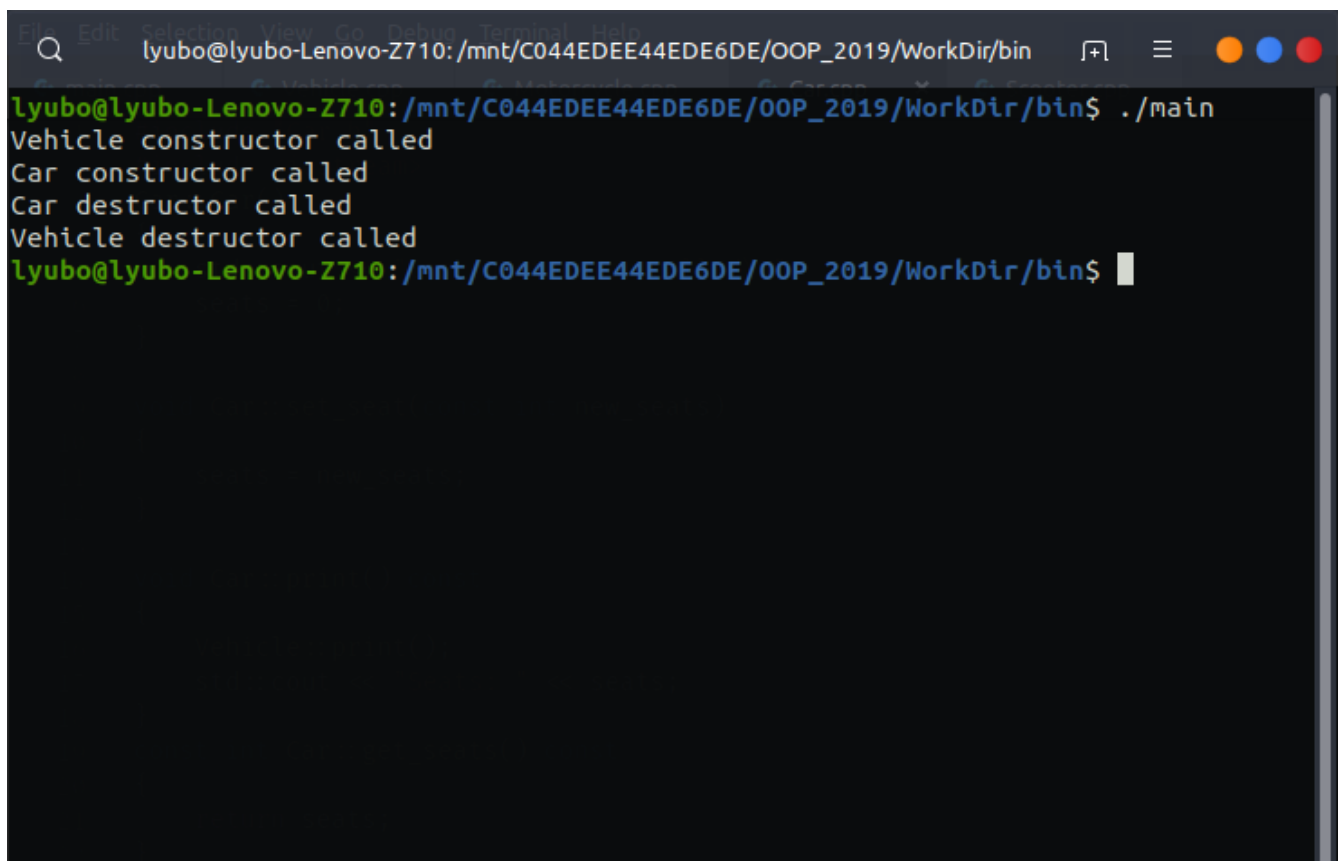
## Наследяване на оператор=

Извикването на оператор= при наследеният клас става по следният начин:

```
Car& Car::operator=(const Car &object_to_copy_from){  
    seats = object_to_copy_from.seats;  
    Vehicle::operator=(object_to_copy_from);  
  
}
```

## Наследяване на деструктори

Редът на изпълняване на деструкторите е обратен на този при конструкторите.



```
lyubo@lyubo-Lenovo-Z710: /mnt/C044EDEC44EDE6DE/OOP_2019/WorkDir/bin$ ./main  
Vehicle constructor called  
Car constructor called  
Car destructor called  
Vehicle destructor called  
lyubo@lyubo-Lenovo-Z710: /mnt/C044EDEC44EDE6DE/OOP_2019/WorkDir/bin$
```

## Задача 3:

Да се добави член-данна `fuel_type`, отговаряща за типа гориво на колата (petrol, diesel, etc) в класа `Car`

## Задача 4:

Да се направи реализация на `MyString`, който наследява `MyVector<char>`