Виртуални функции, полиморфизъм, абстрактни класове

Преобразуване на класове

Едно основно качествено на наследяването е възможността за **представяне на един клас като друг**. Конкретно ще се спрем на два вида преобразуване

Производен към основен (Car -> Vehicle)

Нека имаме следния обект от тип `Car`:

```
Car a;
a.set_horsepower(380);
a.set_maker("Ferrari");
a.set_model("LaFerrari");
a.set_seat(4);
a.set_fuel_type("Diesel");
```

std::cout << std::endl;

и обект от тип `Vehicle`:

```
Vehicle b;
```

Възможно е да представим `Car` като обект от тип `Vehicle`:

```
b = a;
b.print();
```

Съответно метода `print()` ще изведе само името и модела на колата (`Vehicle` не притежава други член-данни)

Същото е възможно и с указатели:

```
Car* pointer_to_a = &a;
Vehicle* b = pointer_to_a;

b->print();
```

Основен към производен (Vehicle -> Car)

Такова преобразуване е невъзможно* (в C++ много малки неща за напълно невъзможни), защото член-данните на производния клас (в случая `Car`) остава неинициализирани.

Статично свързване

Всеки от класовете `Vehicle`, `Car` и `Motorcylce` имат метод `print()`. Когато създадем обекти от всеки един тип, този тип е свързан с правилния метод `print()`

```
Vehicle a;
a.print(); //Vehicle::print

Car b;
b.print(); //Car::print

Motorcycle c;
c.print(); //Motorcycle::print

Vehcile* p = &c;
p->print(); //Vehicle::print
```

Понякога обаче се налага да свържем даден обект от един тип с метод от друг клас

Виртуални функции

Нека променим `print` метода в `Car` и `Motorcycle`, така че да не използват `Vehicle::print()`:

```
void Car::print() const
{
    std::cout << "Maker: " << maker << std::endl;
    std::cout << "Model: " << model << std::endl;
    std::cout << "Horsepower " << horsepower << std::endl;
    std::cout << "Seats: " << seats << std::endl;
    std::cout << "Fuel type: " << fuel_type;
}
```

```
void Motorcycle::print() const
{

    std::cout << "Maker: " << maker << std::endl;
    std::cout << "Model: " << model << std::endl;
    std::cout << "Horsepower " << horsepower << std::endl;
    std::cout << "Luggage capacity: " << luggage_capacity;
}
```

И да направим указател от тип `Vehicle` към обект от тип `Car`:

```
Car b;
Vehicle* p = &b;

p->print();
```

Тук се използва `print()` на `Vehicle`, макар че указателя сочи към обект от тип `Car`

Как да оправим това ?

Можем да свържем **динамично** указателя към съответния метод, като направим `Vehicle::print()` **виртуален метод**. Това става по следният начин:

```
/*...*/
virtual void print() const;
/*...*/
```

Така, когато отново пуснем горния код, ще се извика `Car::print()`, независимо, че обекта е указател от тип `Vehicle`

Също така е прието, когато се **"пренаписва"** виртуален метод, той да бъде деклариран с ключовата дума `override`:

```
/*...*/
void print const override;
/*...*/
```

Правила на виртуалните функции

1. Виртуалните функции не могат да са `static` или `friend` на друг клас.
2. Виртуалните функции трябва да бъдат достъпвани чрез указател или псевдоним към базовия клас.
3. Прототипът на виртуалните функции трябва да бъде същия и в базовия, и в производния клас.
4. Винаги се дефинират в базовия клас и се предефинира в производния клас. Не е задължително този метод да е предефиниран в произовдния клас, в този случай, се използва метода от основиния.
5. Един клас може да има виртуален деструктор, но **не може** да има виртуален конструктор.

Виртуални деструктори

Да разгледаме следната ситуация:

```
class A
{
public:
    A()
    {
        std::cout << "Constructor of A called \n";
    }
```

```
    ~A()
    {
        std::cout << "Destructor of A called \n";
    }
};

class B: public A
{
public:
    B()
    {
        std::cout << "Constructor of B called \n";
    }
    ~B()
    {
        std::cout << "Destructor of B called \n";
    }
};
int main() {

    A* p = new B;
    delete p;

    return 0;же да има виртуален деструкт
}
```

Забелязваме, че тук **не се извиква** деструктора на `B`. Това е проблем, защото `B` не освобождава своята памет. За целта трябва да направим деструктора на `A` виртуален.

```
class A
{
public:
    A()
    {
        std::cout << "Constructor of A called \n";
    }
    virtual ~A()
    {
        std::cout << "Destructor of A called \n";
    }
};

class B: public A
{
public:
    B()
    {
        std::cout << "Constructor of B called \n";
    }
    ~B()
    {
        std::cout << "Destructor of B called \n";
    }
```

```
};
int main() {

    A* p = new B;
    delete p;

    return 0;
}
```

Полиморфизъм

Полиморфизмът се изразява в това **един клас да може да бъде представен като неговия родител**. Сега ще разгледаме няколко примера свързазни с полиморфизъм.

Хетерогенни контейнери

Полиморфизма ни позволява да държим различни класове, стига те да наследяват един клас. Можем да направим динамичен масив (или vector) от указатели към базовия клас. Това се получава чрез две основни възможности в C++: един клас да бъде представен като указател от тип класа който наследява и виртуалните функции.

```
std::vector<Vehicle*> a;

Car c;
a.push_back(c);
```

Абстрактни класове

Абстрактен клас е такъв клас, който не може да има създадени инстанции, и служи само като корен на дърво на наследяване. Абстрактен клас е клас, който има поне един виртуален клас = 0

```
virtual print() const = 0;
```