

Обектно-ориентирано програмиране (записки)

- **Марина Господинова**
- **Илиан Запрянов**

Тема 12. Множествено наследяване

В случаите, когато производният клас наследява директно повече от един базов клас, се казва, че наследяването е множествено.

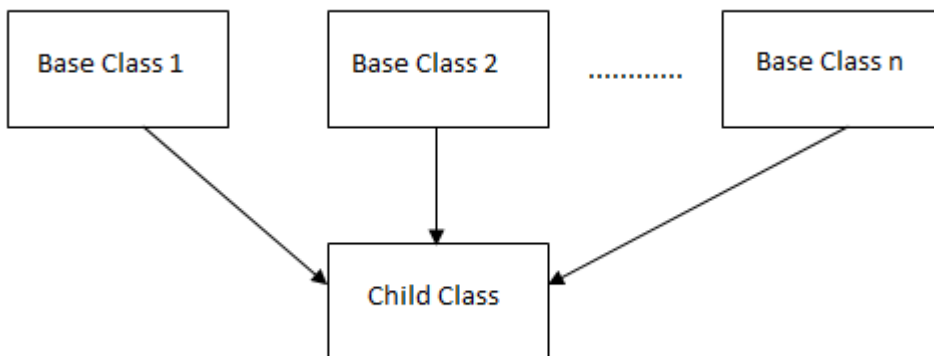
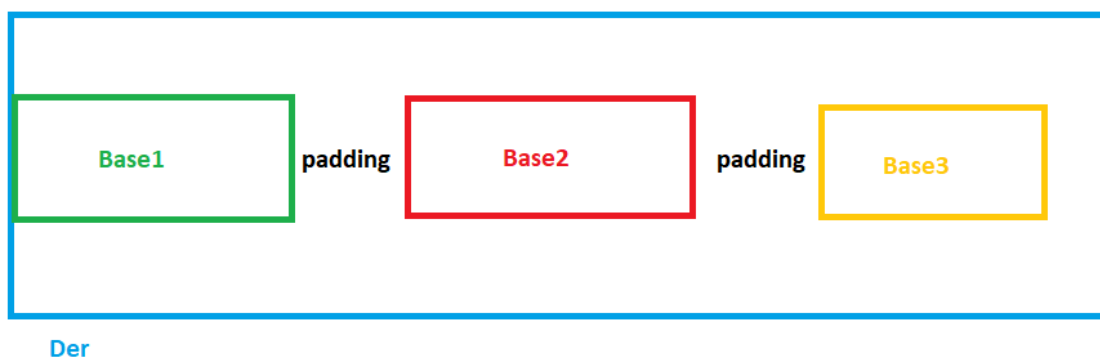


Fig: Multiple Inheritance

Вече свикнахме с идеята, че при стандартното наследяване, в началото на **Der** като скрита член-данна стои **Base**, това тук не се променя. Тоест, ако **Der** наследява **Base1**, **Base2**, **Base3**, то **Der** ще изглежда по следния начин:



Нека имаме следните три класа:

```
class Base1
{
public:
    Base1()
    {
        std::cout << "Base1()" << std::endl;
    }

    ~Base1()
    {
        std::cout << "~Base1()" << std::endl;
    }
};
```

```
class Base2
{
public:
    Base2()
    {
        std::cout << "Base2()" << std::endl;
    }

    ~Base2()
    {
        std::cout << "~Base2()" << std::endl;
    }
};
```

```
class Base3
{
public:
    Base3()
    {
        std::cout << "Base3()" << std::endl;
    }

    ~Base3()
    {
        std::cout << "~Base3()" << std::endl;
    }
};
```

```

class Der : public Base1, public Base2, public Base3
{
public:
    Der()
    {
        std::cout << "Der()" << std::endl;
    }

    ~Der()
    {
        std::cout << "~Der()" << std::endl;
    }
};

```

Чрез специален механизъм, който смята колко трябва да се отместят поинтьрите, всеки Base* може да намери своята част в Der:

```

int main()
{
    Der d;

    Base1* ptr1 = &d;
    Base2* ptr2 = &d;
    Base3* ptr3 = &d;

    std::cout << "ptr1:" << ptr1 << std::endl;
    std::cout << "ptr2:" << ptr2 << std::endl;
    std::cout << "ptr3:" << ptr3 << std::endl;

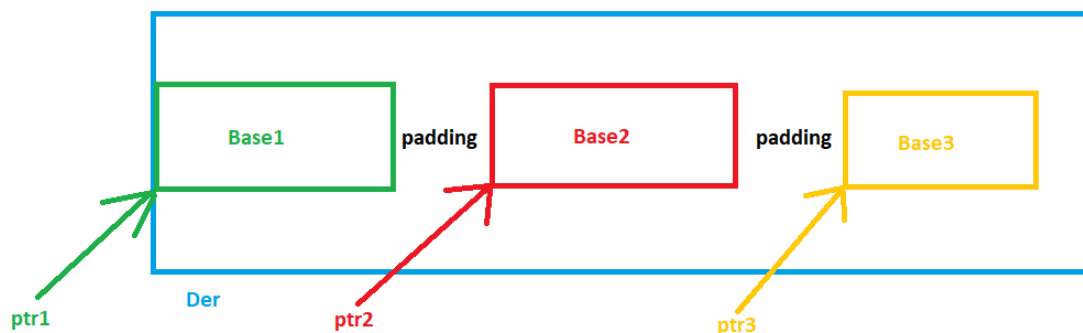
    return 0;
}

```

```

ptr1:000000BAA5F2F4E4
ptr2:000000BAA5F2F4E5
ptr3:000000BAA5F2F4E6

```



Нека **Der** има следния вид

```
class Der : public Base1, public Base2, public Base3
{
    A obj1;
    B obj2;
public:
    Der()
    {
        std::cout << "Der()" << std::endl;
    }

    ~Der()
    {
        std::cout << "~Der()" << std::endl;
    }
};
```

Виждаме, че **Der** е отговорен за създаването на **Base1**, **Base2**, **Base3**. Редът на викането на конструкторите не се различава по никакъв начин от вече разгледания.

```
};
class Der : public Base1, public Base2, public Base3
{
    A obj1;
    B obj2;
public:
    Der()
    {
        std::cout << "Der()" << std::endl;
    }

    ~Der()
    {
        std::cout << "~Der()" << std::endl;
    }
};
```

ред на извикване
на конструктори

ред на извикване
на деструктори

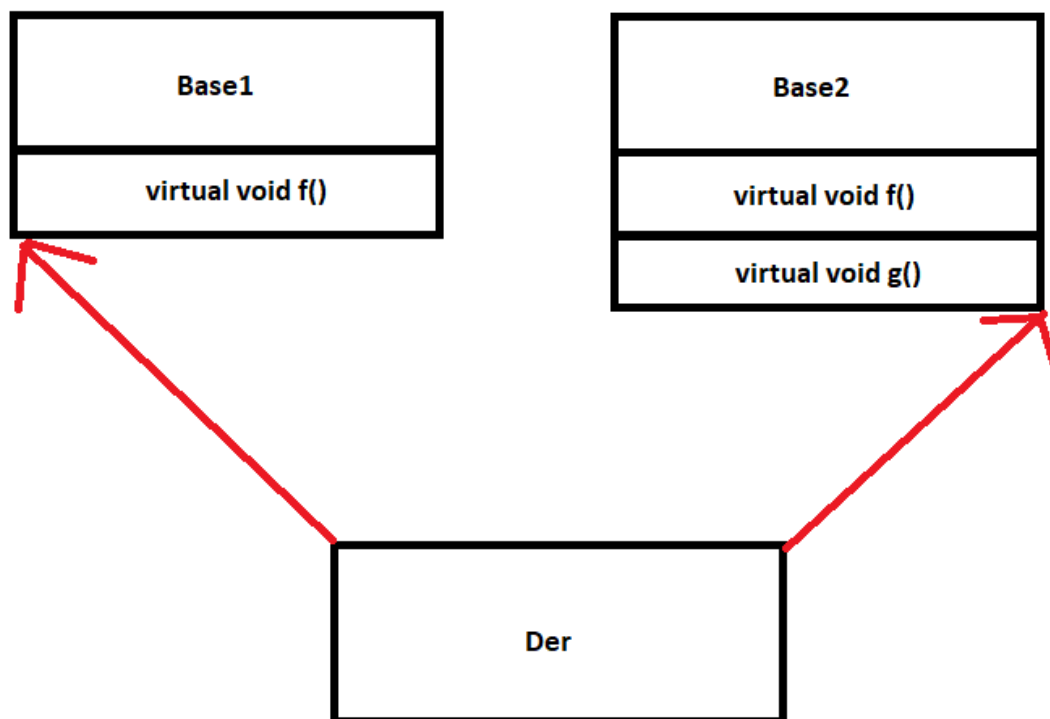
Microsoft Visual Studio De

```
Base1()
Base2()
Base3()
A()
B()
Der()
~Der()
~B()
~A()
~Base3()
~Base2()
~Base1()
```

Голямата шестика също не се различава при множествено наследяване. Напр.:

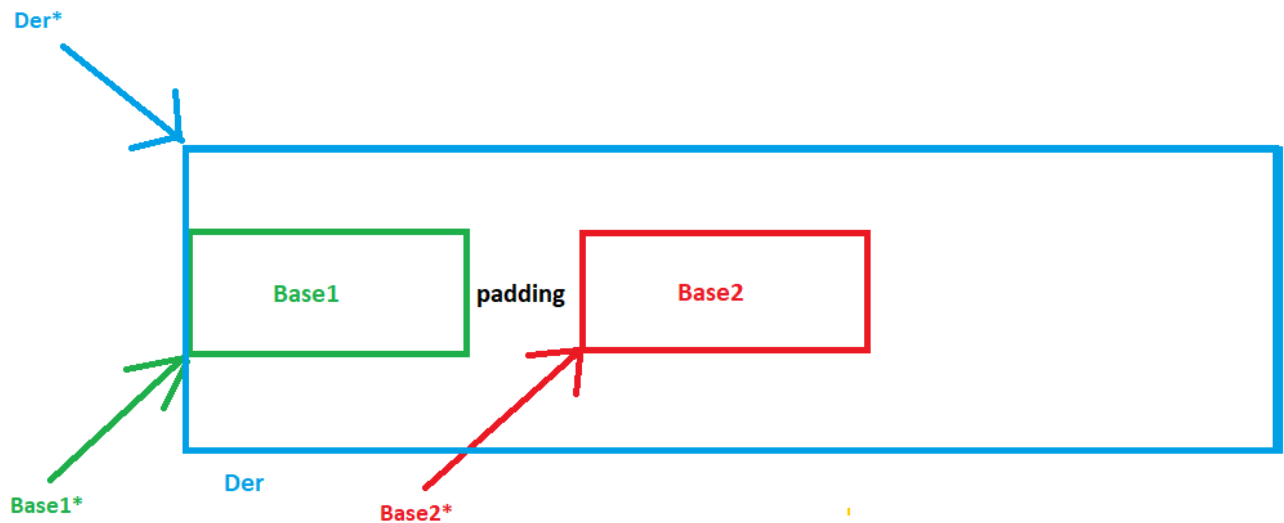
```
Der& Der::operator=(const Der& other)
{
    if (this != &other)
    {
        Base1::operator=(other);
        Base2::operator=(other);
        Base3::operator=(other);
        copyFrom(other);
        free();
    }
}
```

Виртуална таблица при множествено наследяване



Следната полиморфна йерархия ни задължава да презапишем функцията $f()$ в **Der**, тъй като ако не го направим ще стане двусмислица коя от двете да се извика
 \Rightarrow **Der** задължително трябва да **override**-ва функцията $f()$

Виртуалните таблици имат втори параметър - Δ , който смята колко трябва да отместим указателя, за да намерим обекта (отместването е в байтове)



Таблиците придобиват следния вис:



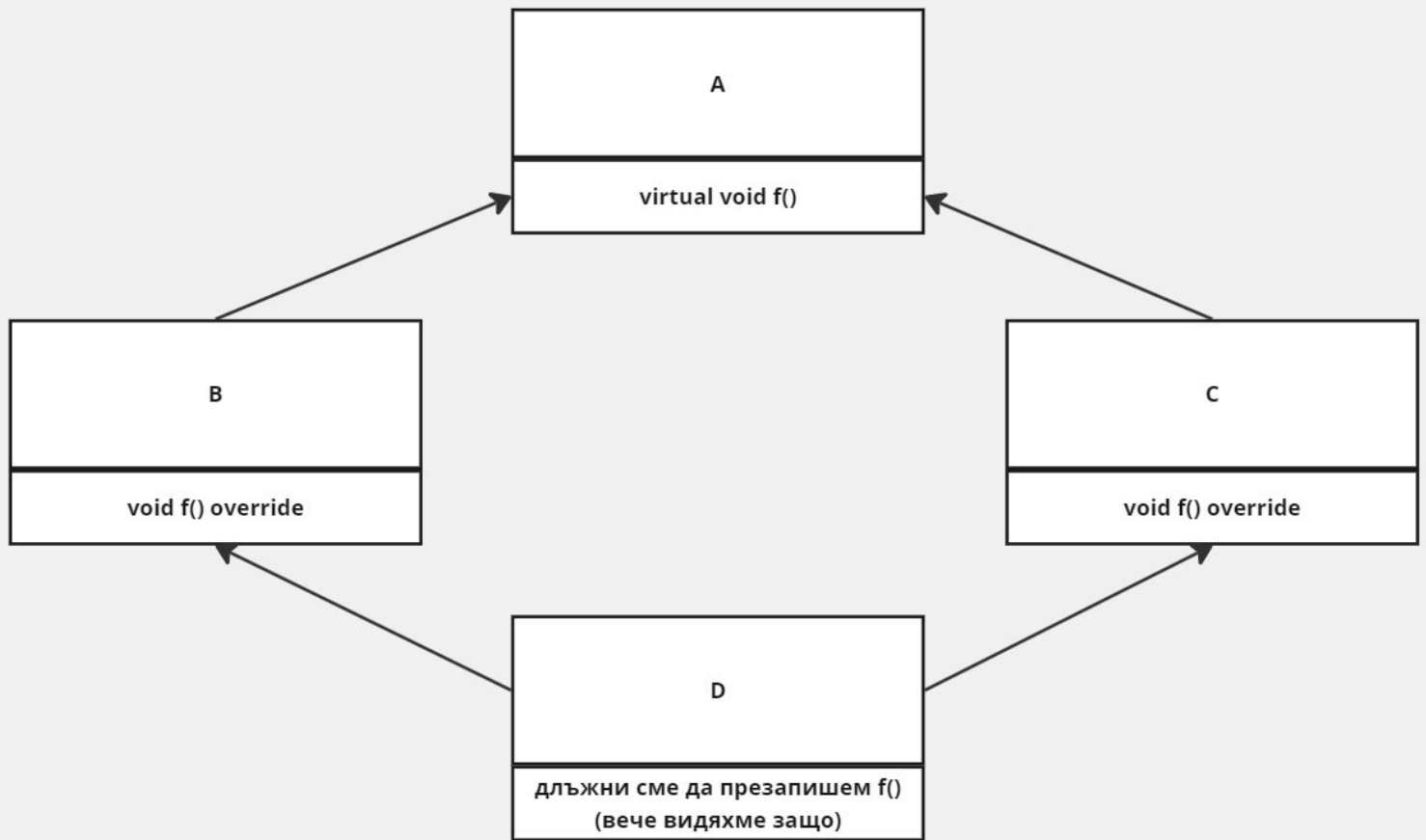
$\Delta(\text{Base2})$ - отместването на **Base2** от началото на **Der**

$\Delta(\text{Base1}) = 0$ - тъй като **Base1** е в началото на **Der**

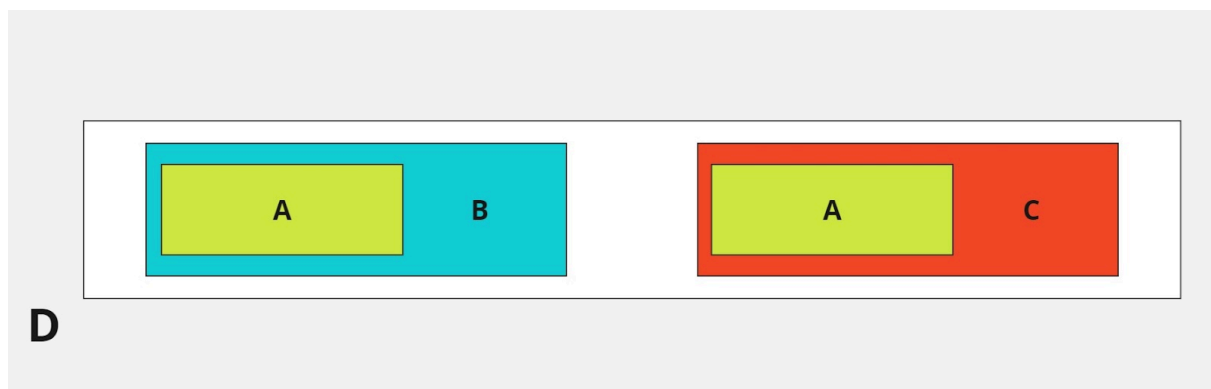
Напр.:

Във втората таблица, се намираме на $\Delta(\text{Base2})$ разстояние от началото. За да се върнем в **Der**, трябва да изминем това разстояние обратно $\Rightarrow -\Delta(\text{Base2})$

Диамантен “проблем”



При създаването на обект от тип D, той изглежда по следния начин



[!] Виждаме проблемът, че всеки обект от тип D води до създаването на две различни A, което е и същността на диамантения проблем

Виртуално наследяване

```
class X : virtual Y
```

- всеки наследник на X е длъжен да каже как да се създаде Y
- прехвърля се отговорността за създаването на Y надолу

В зависимост от типа на обекта, който създадем, се извиква различен конструктор на наследения виртуално. Напр.:

Нека това е класът, който ще наследяваме виртуално

```
class Y
{
public:
    Y(int x, int y, int z)
    {
        std::cout << "Y(int, int, int)" << std::endl;
    }

    Y(int x, int y)
    {
        std::cout << "Y(int, int)" << std::endl;
    }

    Y(int x)
    {
        std::cout << "Y(int)" << std::endl;
    }

    ~Y()
    {
        std::cout << "~Y()" << std::endl;
    }
};
```

```

class X : virtual public Y
{
public:
    X() : Y(3, 7) //Y конструктора с 2 параметъра
    {
        std::cout << "X()" << std::endl;
    }

    ~X()
    {
        std::cout << "~X()" << std::endl;
    }
};

```

```

class A : public X
{
public:
    A() : Y(3) //Y конструктора с 1 параметър
    {
        std::cout << "A()" << std::endl;
    }

    ~A()
    {
        std::cout << "~A()" << std::endl;
    }
};

```

```

class B : public A
{
public:
    B() : Y(3,7,4) //Y конструктора с 3 параметъра
    {
        std::cout << "B()" << std::endl;
    }

    ~B()
    {
        std::cout << "~B()" << std::endl;
    }
};

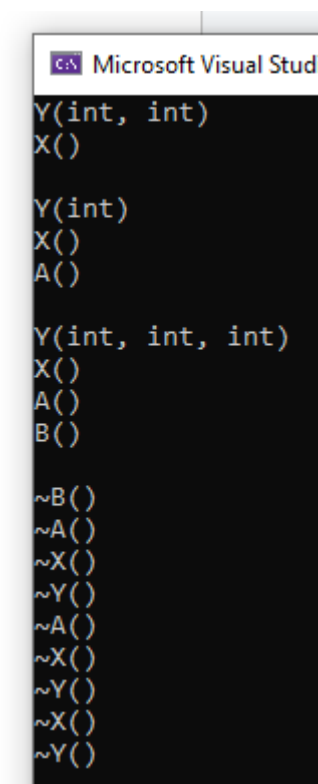
```

```

int main()
{
    X x; //X създава Y с 2 параметъра
    std::cout << std::endl;
    A a; //A създава Y с 1 параметър
    std::cout << std::endl;
    B b; //B създава Y с 3 параметъра
    std::cout << std::endl;
}

```

Не забравяме, че тъй като йерархията ни е $Y \rightarrow X \rightarrow A \rightarrow B$, то ще се извикат конструкторите и на по-горните класове както сме свикнали, заради наследяването. Напр.: A наследява X, което наследява Y \Rightarrow ще се извика първо конструктора на A, после този на X и ще отпечата **X()**, после този на Y и ще отпечата **Y()** и накрая ще се отпечата **A()**



```

Microsoft Visual Stud
Y(int, int)
X()

Y(int)
X()
A()

Y(int, int, int)
X()
A()
B()

~B()
~A()
~X()
~Y()
~A()
~X()
~Y()
~X()
~Y()

```

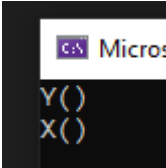
Можем да забележим, че винаги се извиква първо конструктора на виртуално наследения клас, след което тези на базовите, независимо дали наследения виртуално клас е в началото, средата или края на йерархията ни (при деструкторите е същото, но в обратен ред, както сме свикнали)

```

class X : virtual public Y
{
public:
    X() //Y() default
    {
        std::cout << "X()" << std::endl;
    }

    ~X()
    {
        std::cout << "~X()" << std::endl;
    }
};

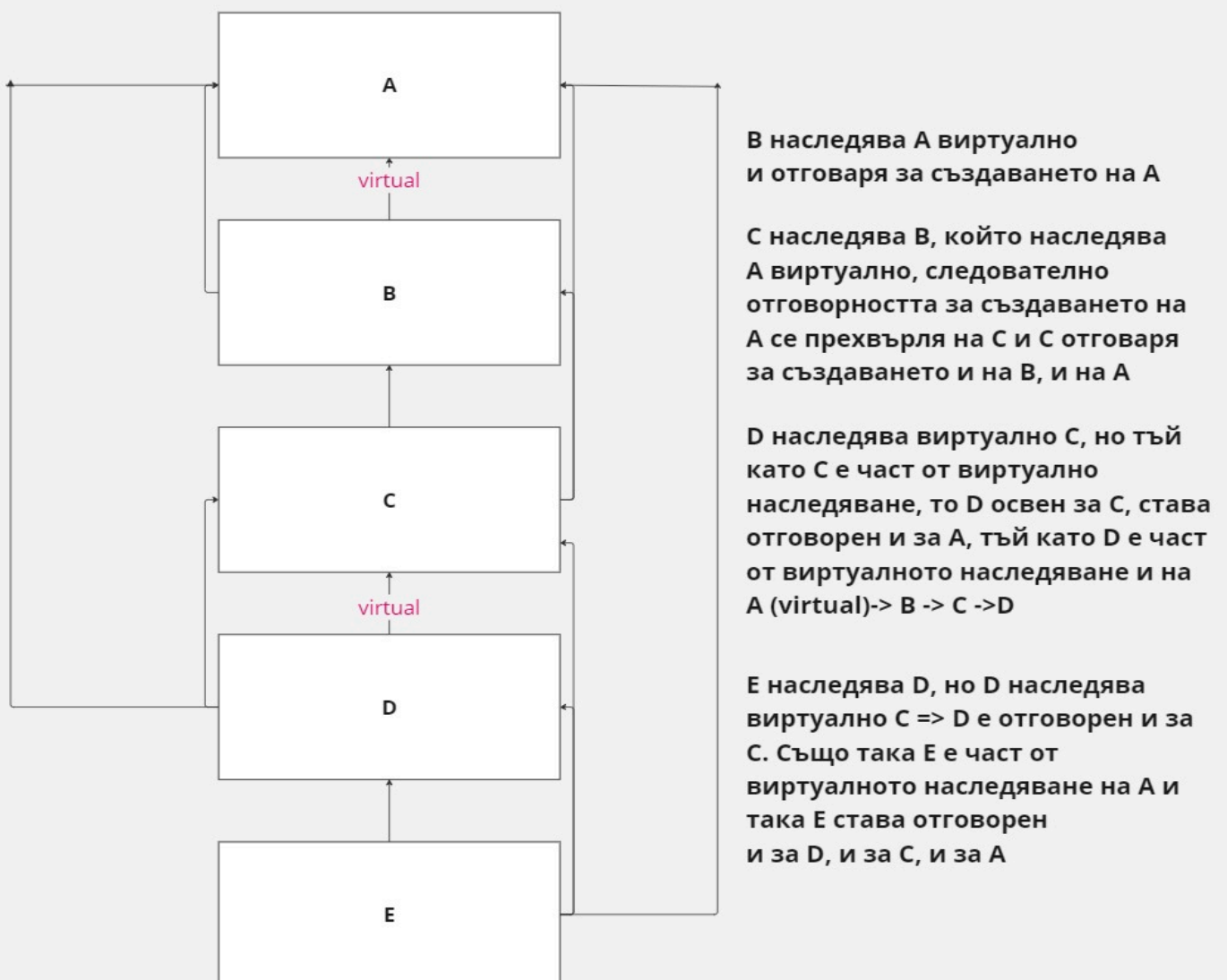
```



Ако пропуснем да кажем как да се създаде Y, то ще потърси default-ния конструктор на Y, ако Y няма такъв ще получим **компилационна грешка**

Използваме виртуално наследяване, защото очакваме обектът Y да се споделя и от други наследници

Пример за друга такава йерархия:



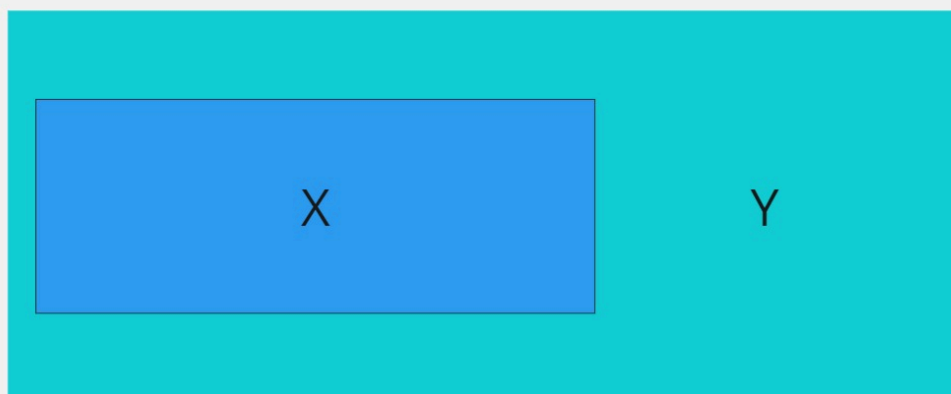
```

1  #include <iostream>
2
3  class X
4  {
5      //
6  };
7
8  class Y : virtual X
9  {
10     //
11 };
12
13 int main()
14 {
15     Y obj;
16
17     return 0;
18 }
19
20

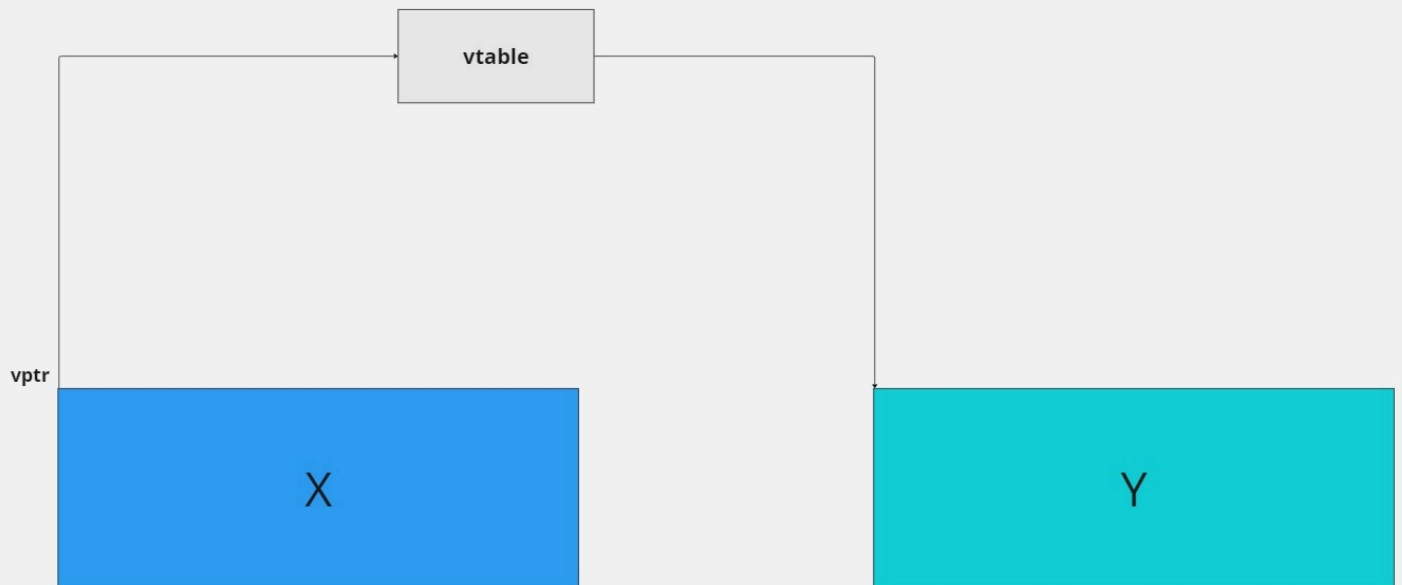
```

При виртуалното наследяване, **за разлика** от неvirtуалното, частта на базовия клас не стои в началото на наследника, както бяхме свикнали досега.

Стандартното наследяване:

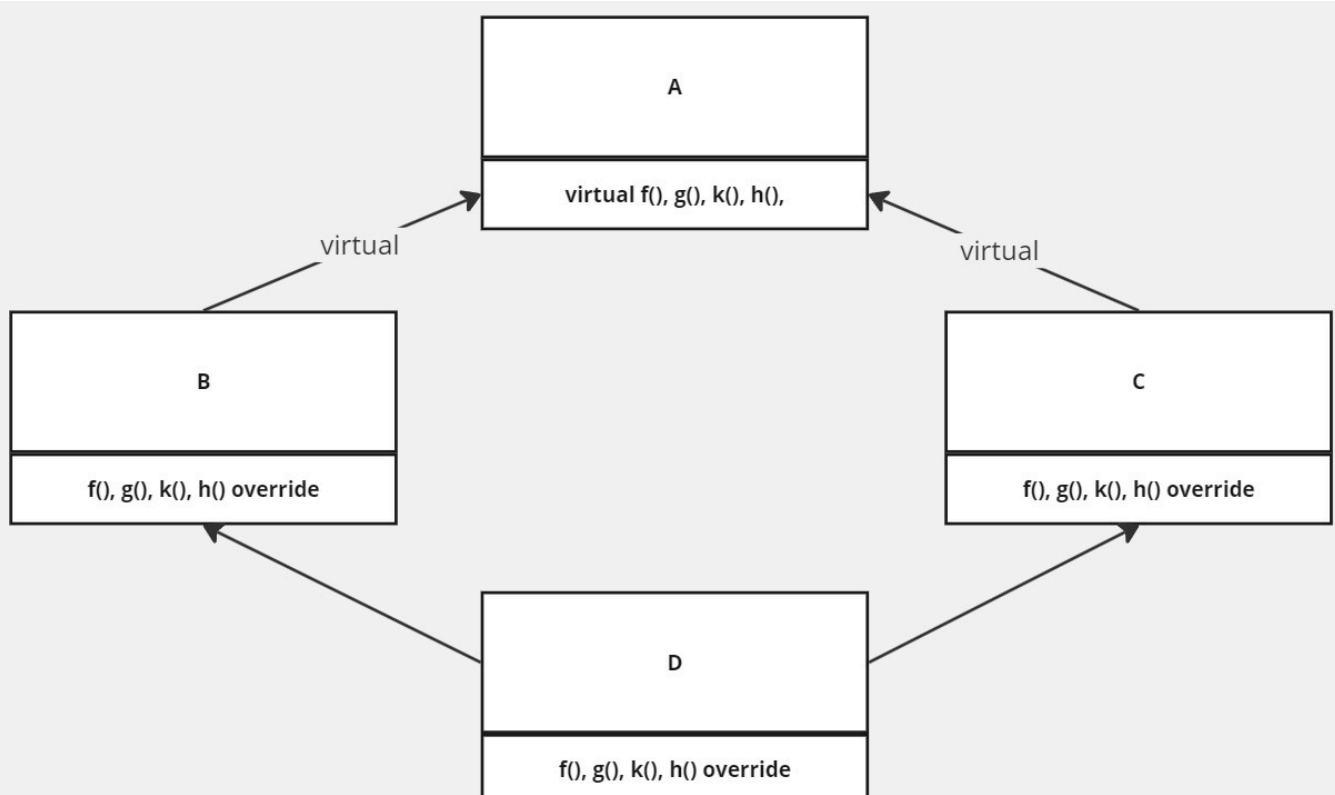


Виртуално наследяване



При виртуално наследяване, се добавя виртуален указател (vptr) към виртуалната таблица (vtable), която съдържа информация за местоположението на виртуалния базов клас (Y) в паметта.

След като разбрахме какво представлява виртуалното наследяване, можем да видим, че именно то е решението на диамантения проблем, с който се срещнахме, реализирайки следната йерархия:



```

class A
{
public:
    A()
    {
        std::cout << "A()" << std::endl;
    }

    ~A()
    {
        std::cout << "~A()" << std::endl;
    }
};

class B : virtual public A
{
public:
    B() : A()
    {
        std::cout << "B()" << std::endl;
    }

    ~B()
    {
        std::cout << "~B()" << std::endl;
    }
};

```

```

class C : virtual public A
{
public:
    C() : A()
    {
        std::cout << "A()" << std::endl;
    }

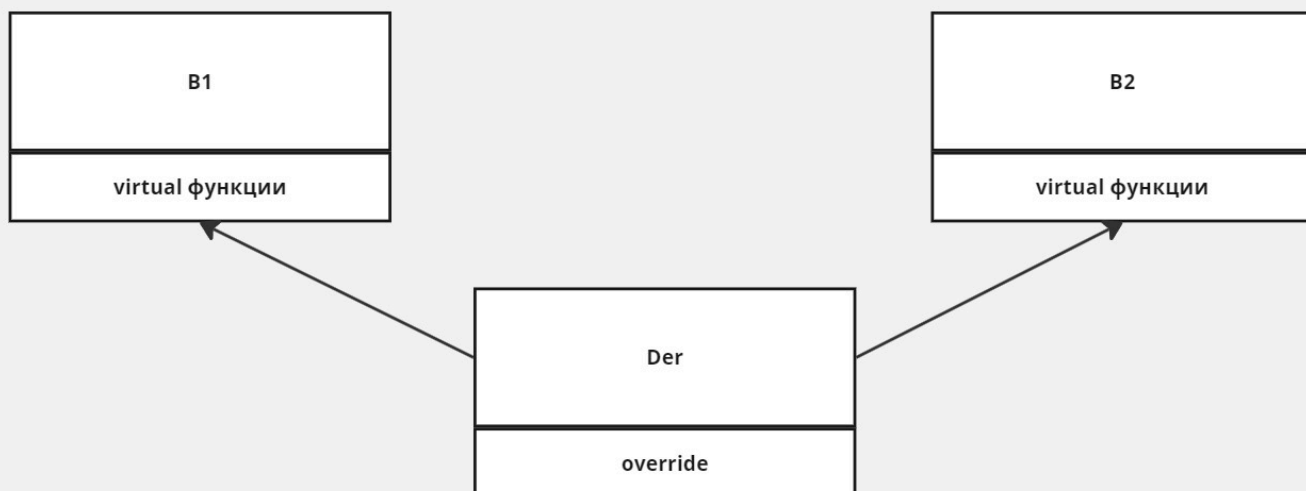
    ~C()
    {
        std::cout << "~A()" << std::endl;
    }
};

class D : public B, public C
{
public:
    D()
    {
        std::cout << "D()" << std::endl;
    }

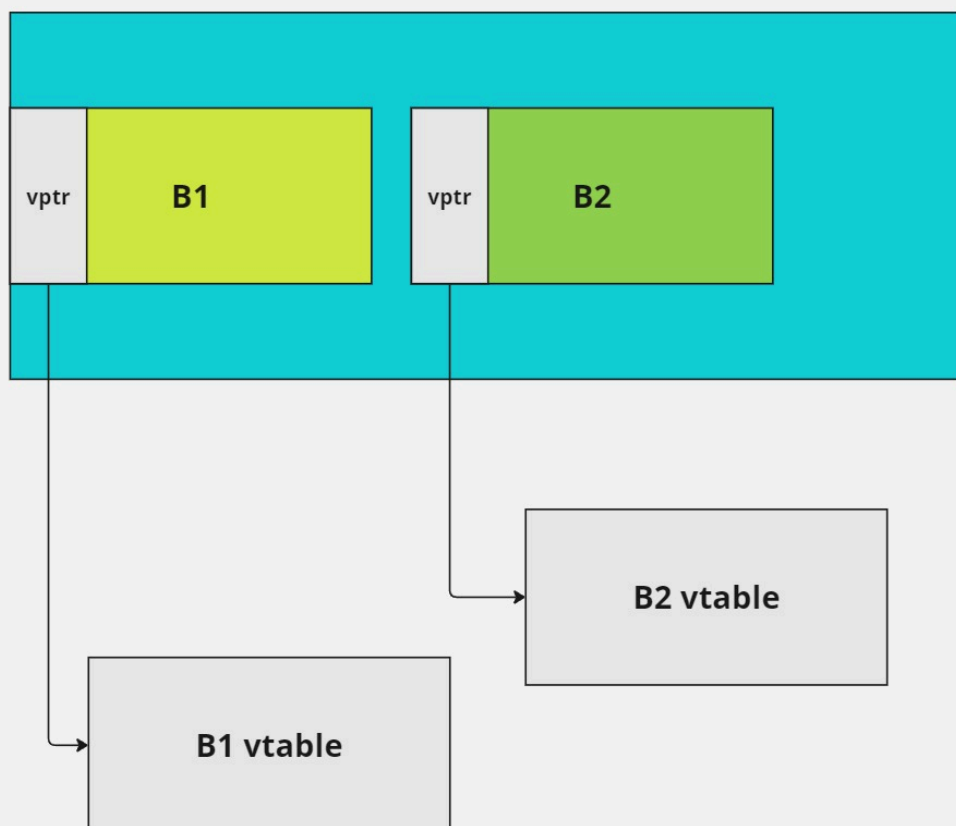
    ~D()
    {
        std::cout << "~D()" << std::endl;
    }
};

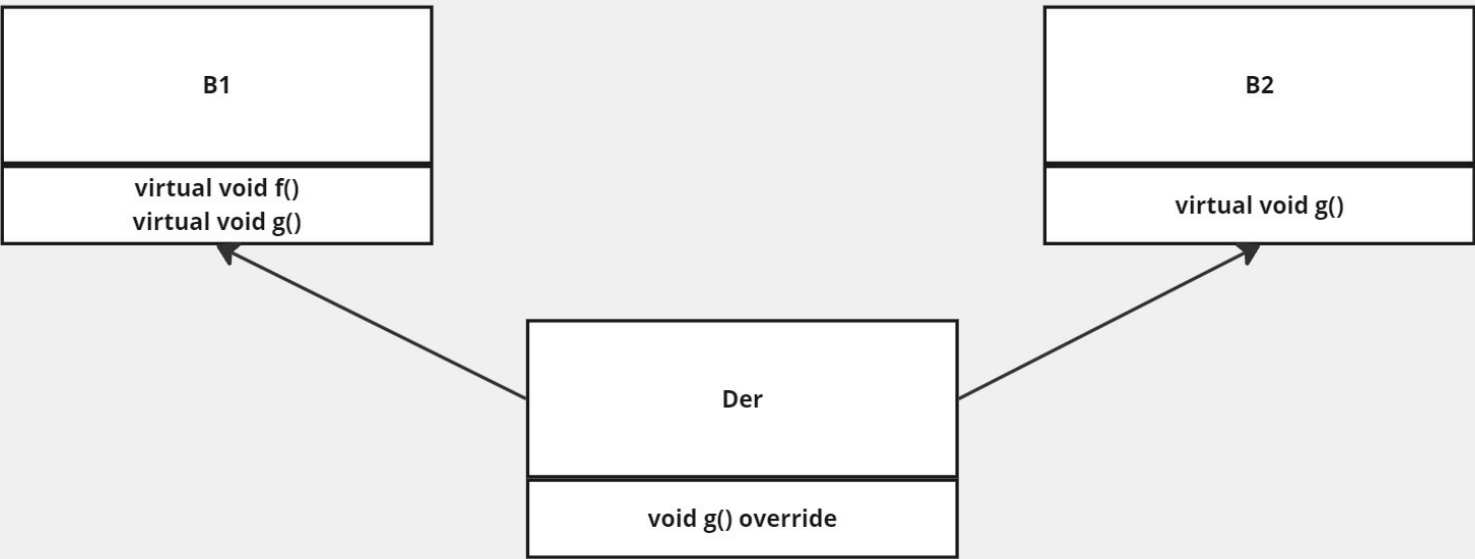
```

Още примери

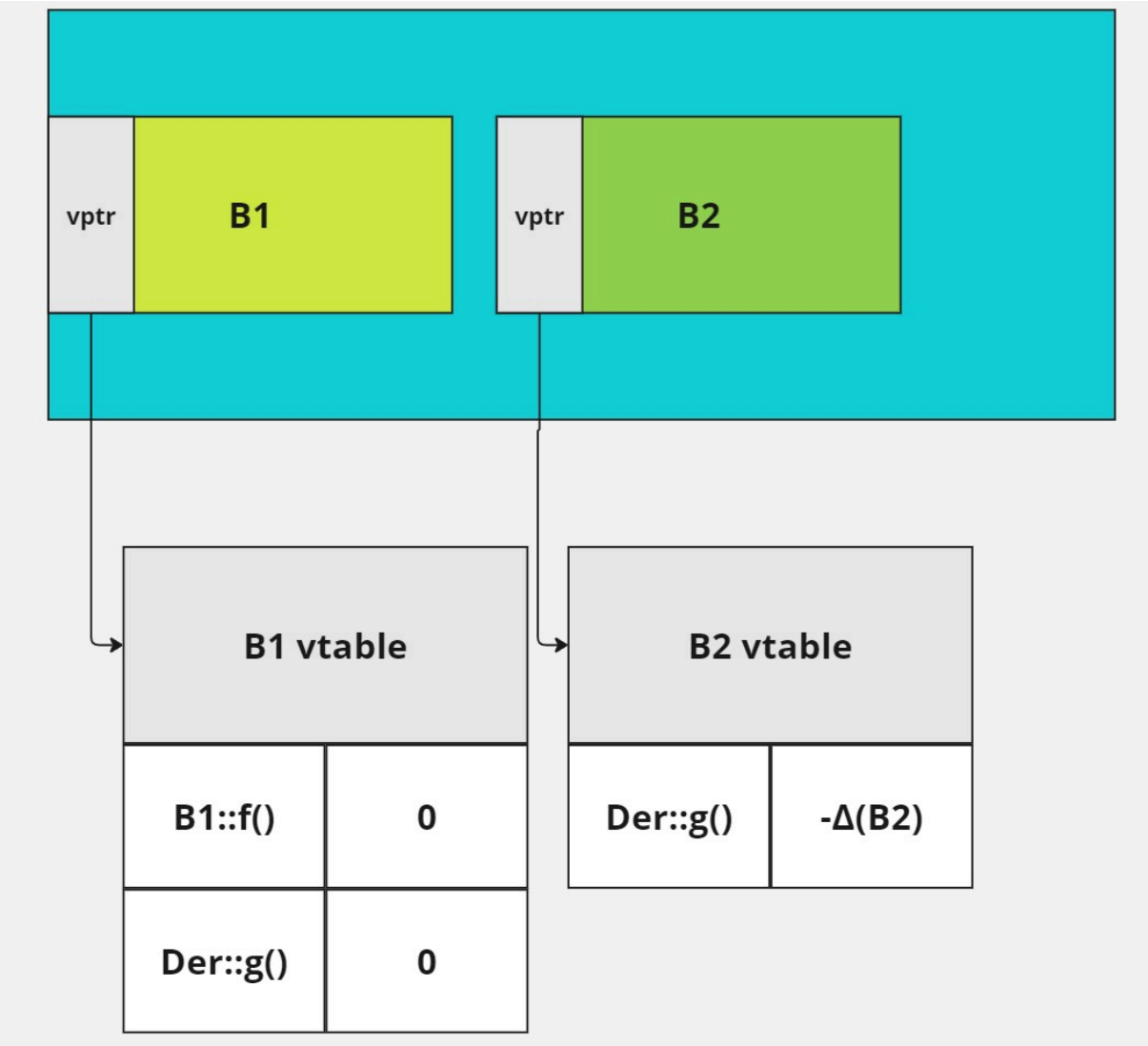


В тази йерархия, **Der** придобива следния вид:

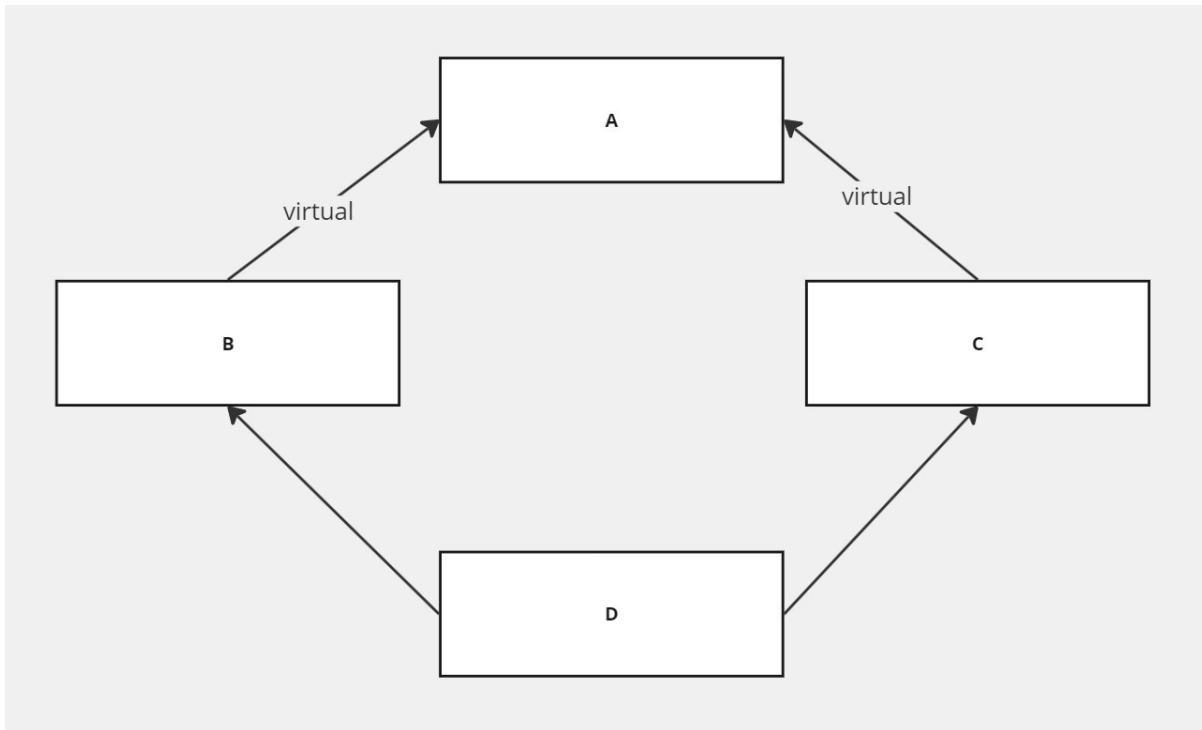




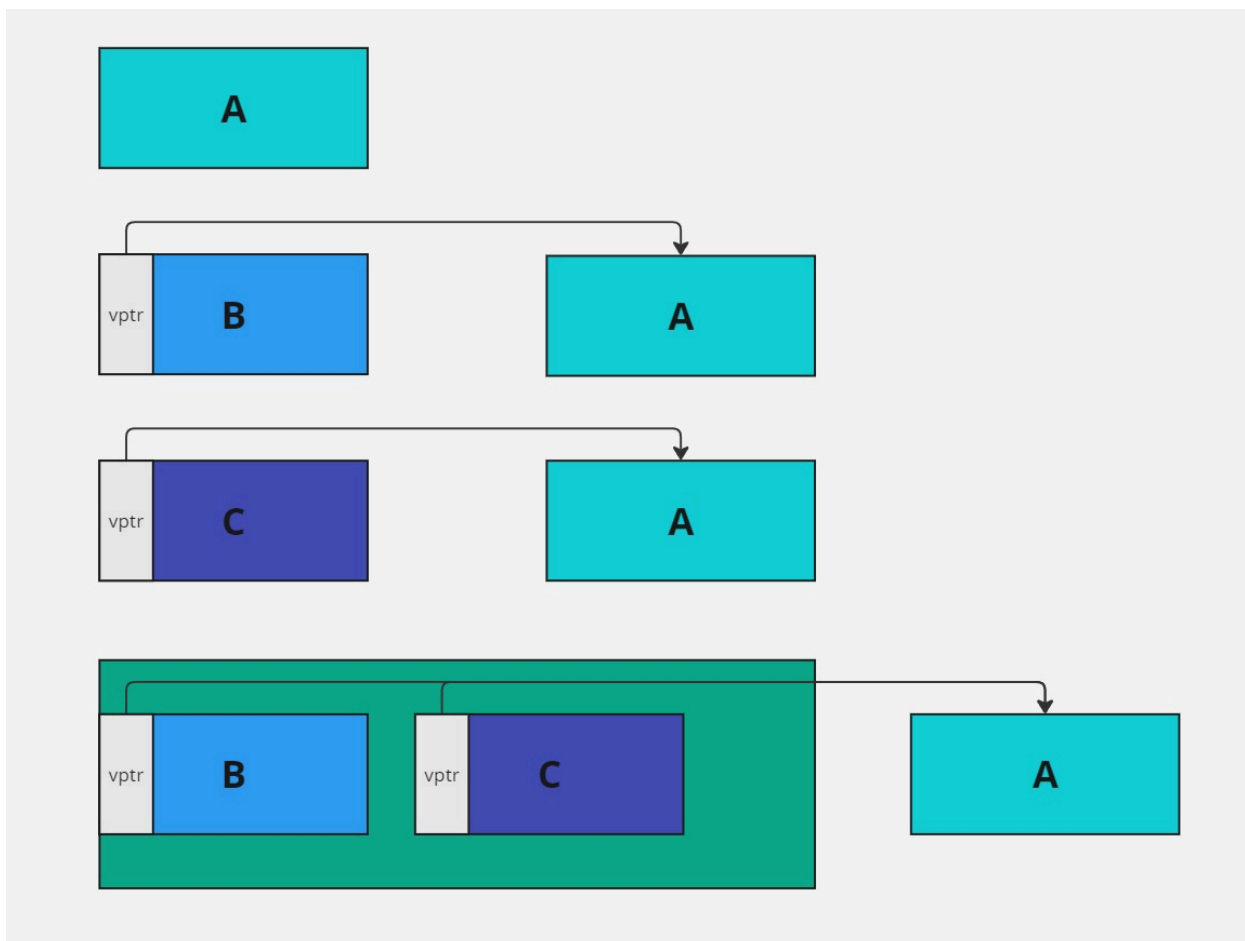
В тази йерархия, **Der** придобива следния вид:

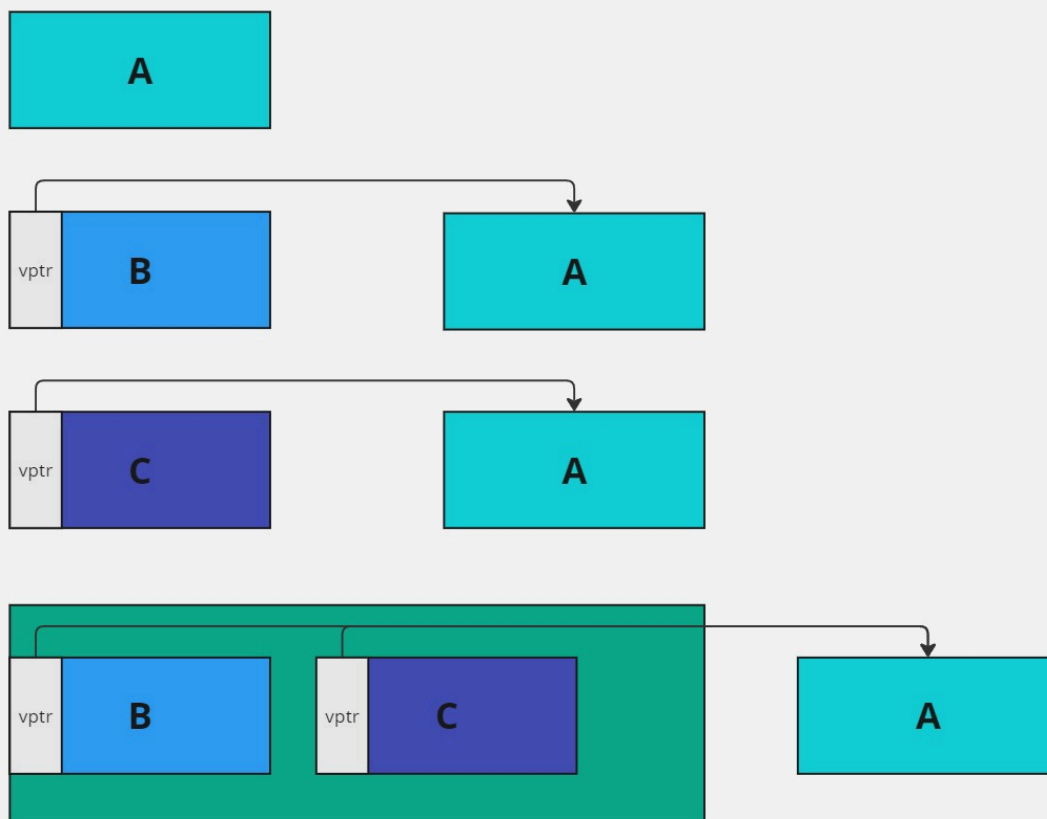
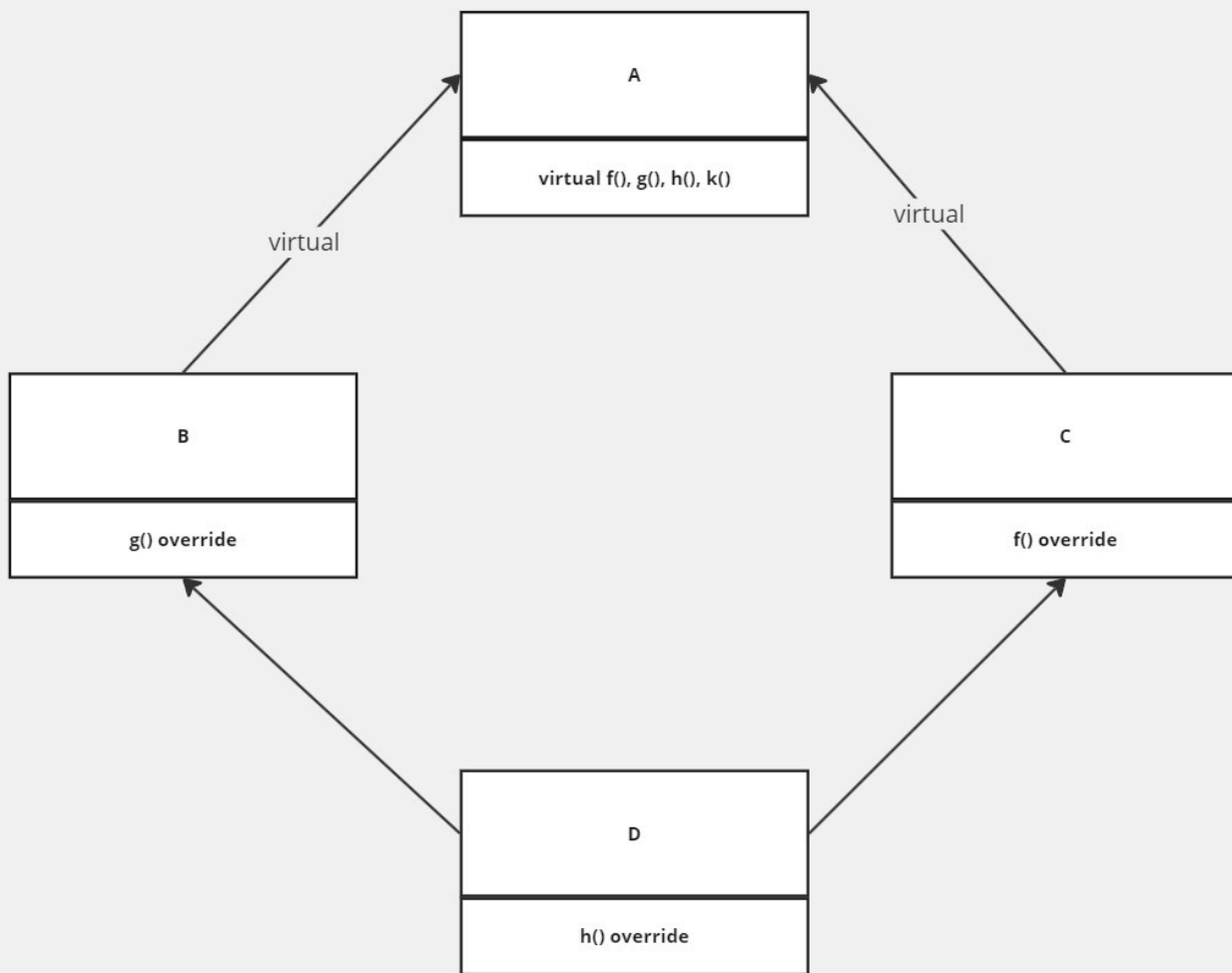


[!] Ако **Der** има виртуална функция, която я няма в B1 и B2, то тя също ще участва във виртуалните таблици



Членовете на тази йерархия ще изглеждат по следния начин:





vtable of D (ако имаме A* ptr)

	Δ
A::k()	0
B::g()	$-\Delta(A)$
C::f()	$-\Delta(A) + \Delta(C)$
D::h()	$-\Delta(A)$

Хетерогенен контейнер

def. | **Хетерогенен контейнер**

- клас, който съдържа колекция от указатели към абстрактен клас и се грижи за менажирането на паметта

Причината да имаме нужда от хетерогенния контейнер е, че не можем да създаваме обекти от абстрактния клас. Затова ще използваме колекция от пойнтери от тип **Base**, които сочат към обекти от наследниците и извикват техните функции

Копиране

Копирането на обекти става чрез клониране - това е функция, която връща копие на себе си (или с други думи динамично заделен обект от този тип)

```
Base* clone {  
    return new A(*this);  
}
```

Ако **A** наследява **Base**, така би изглеждала презаписана функцията `clone()` в интерфейса на **A**. Чрез копиращия конструктор създаваме ново **A**, което копира данните на `*this`.

Триене

Възползвайки се от това, че деструкторът е виртуален, виртуалната таблица ще се справи с намирането на обектите и не ни интересува техния тип. Така, че триенето е същото като при предните масиви от указатели, които сме срещали. Пример:

```
void Farm::free()  
{  
    for (size_t i = 0; i < animalsCount; i++)  
        delete animals[i]; //не се инт. какъв обект е. (вирт дестр)  
    delete[] animals;  
}
```

Първо освобождаваме заделената памет, към която сочи всеки един пойнтер от масива, след което и самия масив

Тъй като **има конкретика** само във factory, тоест хетерогенния контейнер не знае типа на обектите си, тяхното **разпознаване** може да стане чрез допълнителна член-данна в базовия клас, която пази типа на обекта или `dynamic_cast`

def. | Visitor Pattern

- когато си взаимодействат обекти от полиморфна йерархия (например член-функция на наследник, на която се подава обект от друг наследник)