Обектно-ориентирано програмиране (записки)

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

Тема 10. Наследяване

def| Der е наследник на **Base**, ако разширява неговите данни/поведение

Увод

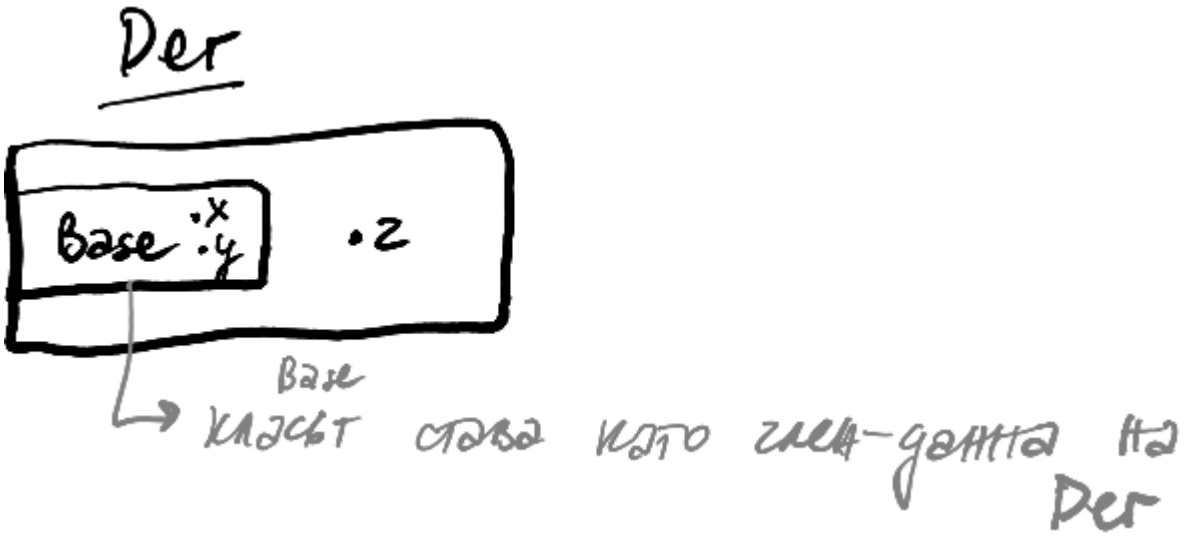
01.cpp*

inheritance

```cpp
#include <iostream>

//това ще е класът, който ще използваме
//за наследяване
class Base
{
public:
    int x;
    void f()
    {
        std::cout << "f()" << std::endl;
    }

private:
    int y;
    void g()
    {
        std::cout << "g()" << std::endl;
    }

};
```

```cpp

//синтаксис за наследяване:
//class <име на наследника>: <начин на наследяване> <име на наследения клас>
class Der : public Base
{
public:
    int z;
};

int main()
{
    //наследяването ни позволява да използваме
    //член-данни/член-функции на друг клас + още наши имплементации
    Der obj;
    obj.z++;
    obj.x++;
    obj.f();

    return 0;
}
```

Можем да си представим, че наследеният клас стана нещо като член-данна на наследника.

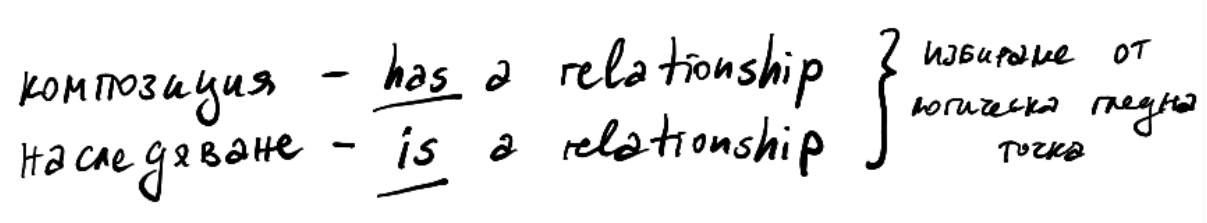


Композиция vs наследяване

```
//синтаксис за наследяване:
//class <име на наследника>: <начин на наследяване> <име на наследения клас>
class Der : public Base
{
public:
    int z;
};

//ще направим нов клас, който е композиция на
//[DerTest] и [Base]
class DerTest
{
public:
    Base b;
    int z;
};
```

```cpp
int main()
{
    //[Der] и [DerTest]
    //са идентични относто функционалност
    Der obj;
    obj.z++;
    obj.x++;
    obj.f();

    DerTest objTwo;
    objTwo.z++;
    objTwo.b.x++;
    objTwo.b.f();

    return 0;
}
```

Можем да направим извода, че разликата между композиция и наследяване е чисто логическа



Has-a Relationship (Отношение "**притежава**") **(композиция)**
Когато един клас включва един или повече обекти от други класове като член-данни. Това означава, че един клас **притежава** или съдържа обекти от други класове.

Кога да се използва: когато един клас съдържа или притежава друг клас като част от него, но не е този клас по природа. Например, ако имате клас **Кола** и клас **Двигател**, **Колата** има **двигател**, но **колата НЕ Е двигател**. Това означава, че **Кола** ще има обект **Двигател** в себе си, което му позволява да използва функционалността на двигателя, например да го пуска или спира.

Is-a Relationship (Отношение "**е вид на**") **(наследяване)**
Когато клас **Der** наследява клас **Base**, казваме, че всеки обект от **Der** "**е вид на**" обект от **Base**. Това означава, че наследникът наследява **интерфейса** (публичните и защитените методи и свойства) на базовия клас.

Кога да се използва: Използва се, когато имаме два класа, и единият клас е подмножество на другия. Например, ако имате клас **Град** и клас **Пловдив**, **Пловдив** е подмножество на **Град**. Така че **Пловдив** ще наследи **Град** , защото **Пловдив** е **Град**.

Shadow

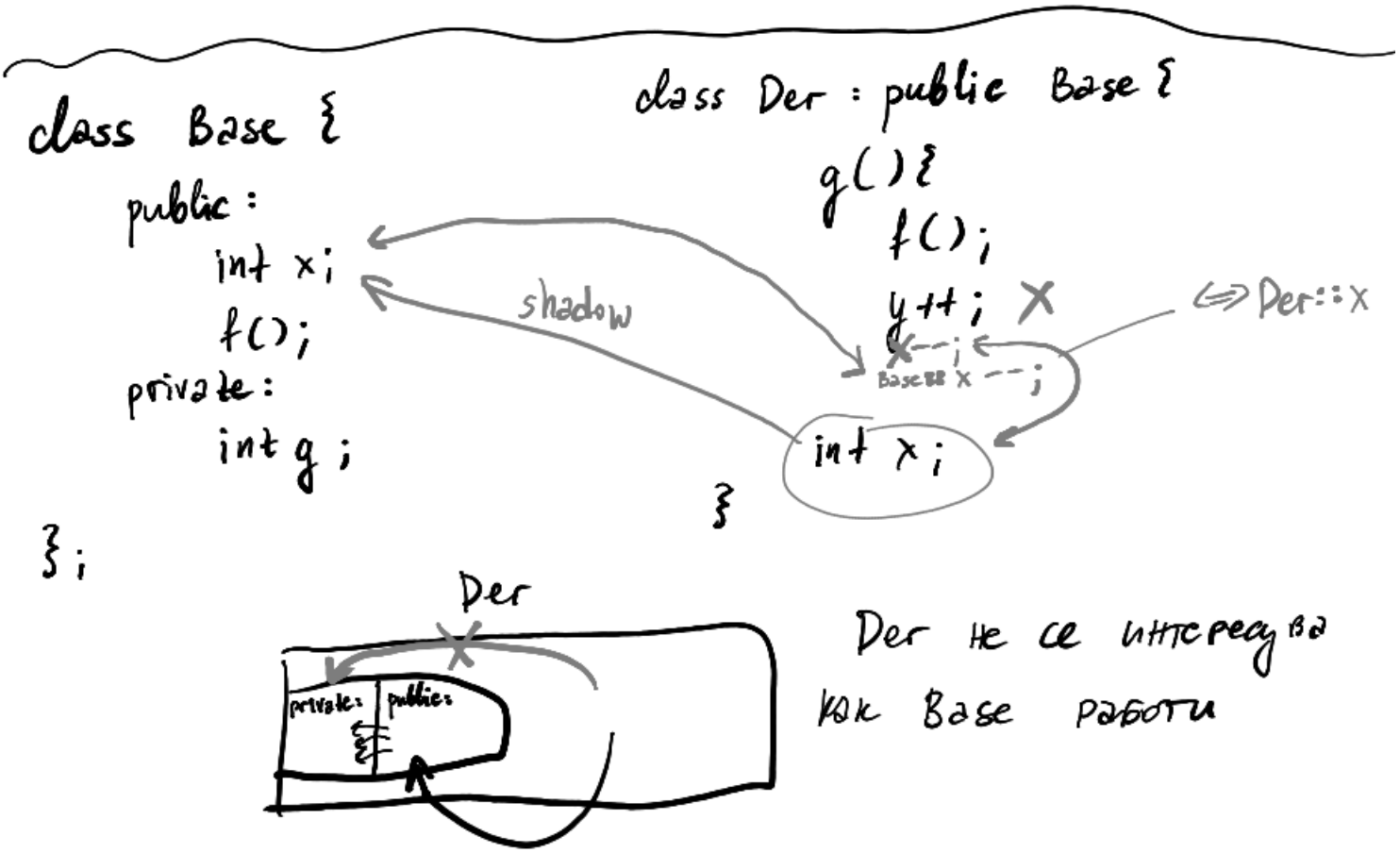
```cpp
//това ще е класът, който ще използваме
//за наследяване
class Base
{
public:
    int x;
    void f()
    {
        std::cout << "f()" << std::endl;
    }


private:
    int y;
};
```

```cpp
class Der : public Base
{
public:
    int x;
    void g()
    {
        f();
        //y++; //нямаме достъп до [y] (след малко ще видим защо)

        //първоначалните стойности на двете член-данни [x] (в Der и Base)
        std::cout << "[Der] -> [x]: " << x << std::endl;
        std::cout << "[Der] -> [Base] -> [x]: " << Base::x << std::endl;
        std::cout << std::endl;

        x--; // ще декрементира член данната [x] на класа [Der]

        //стойностите, след x--;
        std::cout << "[Der] -> [x]: " << x << std::endl;
        std::cout << "[Der] -> [Base] -> [x]: " << Base::x << std::endl;
        std::cout << std::endl;

        Base::x--; // ще декрементира член данната [x] на класа [Base]

        //стойностите, след Base::x--;
        std::cout << "[Der] -> [x]: " << x << std::endl;
        std::cout << "[Der] -> [Base] -> [x]: " << Base::x << std::endl;
    }

    //тоест при повтаряне на имена на променливи/функции при наследяване,
    //тези на наследникът "shadow-ват" тези на наследения клас
};

```

```
52  int main()
53  {
54      Der obj;
55      obj.g();
56
57      return 0;
58  }
```

```
[Der] -> [x]: -858993460
[Der] -> [Base] -> [x]: -858993460

[Der] -> [x]: -858993461
[Der] -> [Base] -> [x]: -858993460

[Der] -> [x]: -858993461
[Der] -> [Base] -> [x]: -858993461
```

Бележка: Base влияе на големината на Der, тъй като все пак е част от него, но Der не се интересува как работи Base.



Der не се интересува как Base работи

Видове наследяване

Модификатори за достъп

- **private**
- **protected**
- **public**

```
class X
{
private:
    int a;

//модификаторът [protected] се използва при наследяване
//и означава, че наследниците могат да достъпят [b], но [b]
//не може да бъде достъпено от външния свят, т.е.
//извън наследения и наследяващите класове
protected:
    int b;

public:
    int c;
};
```

```
//публично наследяване
class A : public X
{
    //тук [a] ще се наследи, но нямаме достъп до тази член-данна в [A],
    //тъй като е [private] в [X]

    //[b] -> protectd (тъй като [A] е наследник на [X] => имаме достъп до нея)

    //[c] -> public
};
```

```
//protected наследяване
class B : protected X
{
    //тук [a] ще се наследи, но нямаме достъп до тази член-данна в [A],
    //тъй като е [private] в [X]

    //[b] -> protected (ще си остане protected)

    //[c] -> protected (public член-данните стават protected при protected наследяване)
};
```

```
class C : private X
{
    //тук [a] ще се наследи, но нямаме достъп до тази член-данна в [A],
    //тъй като е [private] в [X]

    //[b] -> private (protected член-данните стават private при private наследяване)

    //[c] -> private (public член-данните стават private при private наследяване)
};

 //private наследяването ни дава извода, че:
class D : public C
{
    //от разгледания случай с член-данната [a] => нямаме достъп до нито една член-данна,
    //тъй като в [C] -> [a], [b] и [c] са станали [private]
};
```

Бележка:

- **при класовете наследяването по default е private**
- **при структурите наследяването по default е public**

Накратко

public - запазва всичко, т.е.
- **public -> public**
- **protected -> protected**
- **private -> нямаме достъп (тъй като е private в наследявания клас)**

protected - прави всичко protected, т.е.
- **public -> protected**
- **protected -> protected**
- **private -> нямаме достъп (тъй като е private в наследявания клас)**

private - прави всичко private, т.е.
- **public -> private**
- **protected -> private**
- **private -> нямаме достъп (тъй като е private в наследявания клас)**

Указатели/референции

При наследяване можем да “насочваме” указатели/референции от базовия клас към обекти към наследника

```cpp
//това ще е класът, който ще използваме
//за наследяване
class Base
{
public:
    int x;
    int y;

};

//наследниците, които ще използваме
class Der1 : public Base
{
public:
    int a;
};

class Der2 : public Base
{
public:
    int b;
};
```

```cpp
int main()
{
    Der1 obj;

    //както казахме, можем да си представим [Base]
    //като член-данна, стояща най-отгоре

    //тъй като обекта [Base] е в началото
    //=> преобразуването е успешно
    Base* ptr = &obj;
    Base& ref = obj;

    //[ptr] не подозира, че е част от нещо по-голямо
    //(в случая [Der1])

    //имаме достъп до член-данните на [Base]
    ptr->x++;
    ptr->y++;

    //но нямаме достъп до член-данните на [Der1],
    //тъй като [ptr] е пойнтър към [Base], а не [Der1]

    ptr->a++; // [ERROR]

    //т.е. губим конктретиката (същото е и при [ref])
    //(губим способността да достъпваме член-данни/член-функции на наследника)
    return 0;
}
```

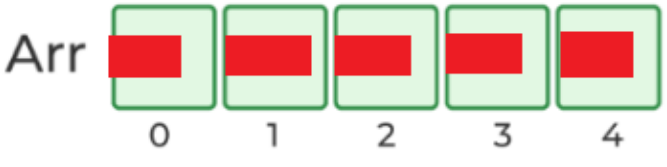
```cpp
void f(const Base* ptr)
{
    ptr->x;
    ptr->y;
}

int main()
{
    Base O1;
    Der1 O2;
    Der2 O3;

    f(&O1);

    //тъй като [Base] стои в началото,
    //можем да подадем обекти от тип [Der1] и [Der2]
    //на функцията f(), параметъра [ptr] ще вземе
    //само [Base]-а, намиращ се вътре в тях
    f(&O2);
    f(&O3);

    return 0;
}
```

```cpp
void f(Der1* ptr)
{
    //
    //
    //
}

int main()
{
    Der1 obj;
    Base* ptr = &obj;

    //[!Но!] обратното НЕ Е възможно,
    //с други думи нямаме casting нагоре, т.е.
    //преобразуване от наследник към наследяван клас

    f(ptr); //[ERROR]

    return 0;
}
```

```cpp
void f(const Base* arr, size_t size)
{
    arr[1]; //не намира следващия [Base],
            //а достъпва памет, която е част още
            //на първия наследник =>
            //
            //НЕ можем да вдигнем нивото на
            //абстракция в тази ситуация
}

int main()
{
    Der1 arr[3];
    f(arr, 3);


    return 0;
}
```

С други думи проблемът е, че масивът не изглежда по начина, по който сме свикнали, т.е.

а има този вид:



Base

член-данните/функции на Der1

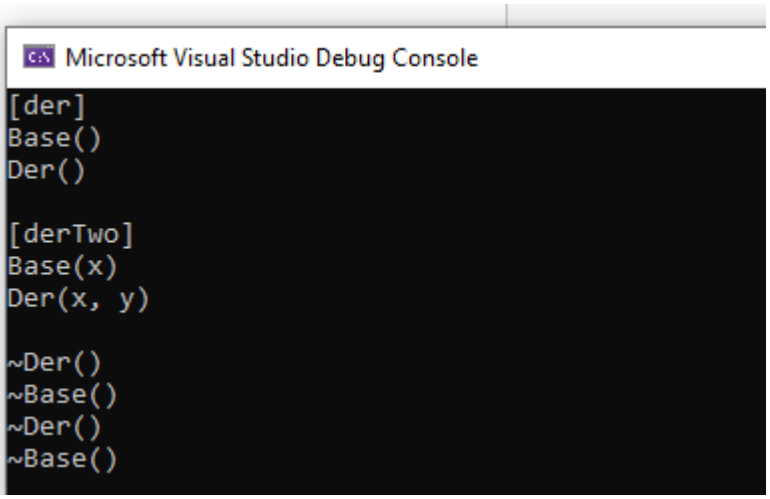
Конструктори при наследяване

- **конструкторът на наследника трябва да каже кой конструктор на базовия клас да се извика**
- **ако не каже, се извиква default-ният такъв**

```cpp
//ще използваме следните два класа
class Base
{
private:
    int x = 0;
public:
    Base()
    {
        std::cout << "Base()" << std::endl;
    }

    Base(int x)
    {
        std::cout << "Base(x)" << std::endl;
        this->x = x;
    }

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

```cpp
class Der : public Base
{
private:
    int y;
public:
    Der()
    {
        std::cout << "Der()" << std::endl;
    }

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

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

};
```

```cpp
int main()
{
    //Ще се извика default-ния конструктор на [Der], в който
    //не упоменаваме кой конструктор на [Base] да се извика =>
    //ще се извика default-ният на [Base]

    std::cout << "[der]" << std::endl;
    Der der;
    std::cout << std::endl;

    //Ще се извика конструктора на [Der], в който се
    //приемат два (int, int) параметъра
    //
    //В него също така упоменаваме, че искаме да извикаме конструктора Base(int),
    //т.е. казваме чрез кой конструктор да създадем [Base] =>
    //ще се извика конкретизирания от нас

    std::cout << "[derTwo]" << std::endl;
    Der derTwo(3, 4);
    std::cout << std::endl;

    return 0;
}
```



```cpp
//ще разширим програмата ни със следните три класа
class A
{
public:
    A()
    {
        std::cout << "A()" << std::endl;
    }

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

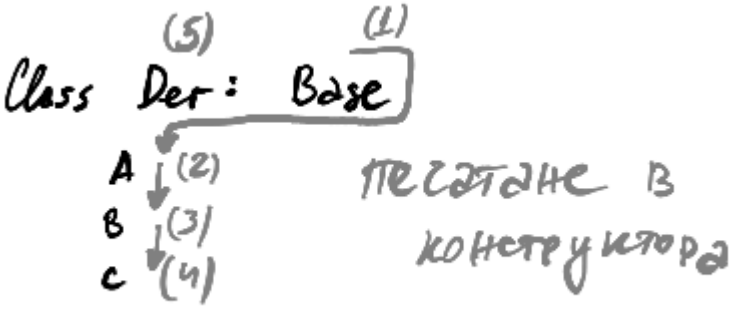
```cpp
class B
{
public:
    B()
    {
        std::cout << "B()" << std::endl;
    }

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

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

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

```cpp
//нека [Der] вече има следните член-данни
class Der : public Base
{
private:
    A a;
    B b;
    C c;
public:
    Der()
    {
        std::cout << "Der()" << std::endl;
    }

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

};
```

```cpp
int main()
{
    //При наследяване каквото и да правим първо винаги се изпълняват
    //конструкторите в НАСЛЕДЕНИЯ клас, след което тези в НАСЛЕДНИКА в реда,
    //в който сме свикнали, т.е. при създаването на [der], програмата ще види,
    //че [Der] е наследник на [Base] => ще се извика default-ния на [Base],
    //след което ще влезем в тялото на [Der], и ще извикаве default-ния на [Base],
    //който извиква заедно със себе си default-ните конструктори на [A], [B], [C]

    std::cout << "[der]" << std::endl;
    Der der;
    std::cout << std::endl;


    return 0;
}
```

```
Microsoft Visual Studio Debug Console
[der]
Base()
A()
B()
C()
Der()

~Der()
~C()
~B()
~A()
~Base()
```



Bonus:

```cpp
//ако добавим член-данна от тип [A] в [Base]
class Base
{
private:
    A a;
    int x = 0;
public:
    Base()
    {
        std::cout << "Base()" << std::endl;
    }

    Base(int x)
    {
        std::cout << "Base(x)" << std::endl;
        this->x = x;
    }

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

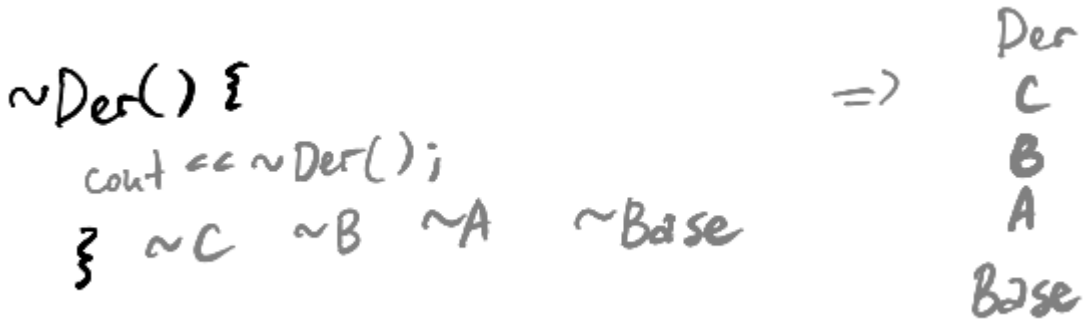
```cpp
int main()
{
    //Когато се извика default-ния конструктор на [Base], както
    //вече видяхме, той в себе си извиква default-ния на [A], както
    //сме свикнали да се държи програмата

    std::cout << "[der]" << std::endl;
    Der der;
    std::cout << std::endl;


    return 0;
}
```

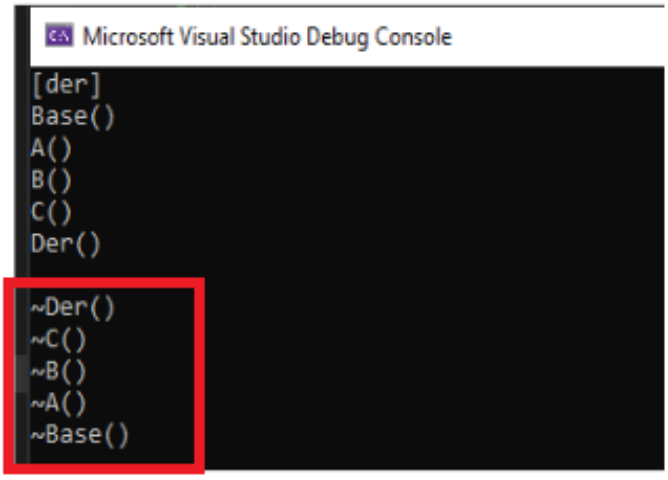
```
Microsoft Visual Studio Debug Console
[der]
A()
Base()
A()
B()
C()
Der()

~Der()
~C()
~B()
~A()
~Base()
~A()
```

Деструктори при наследяване

- **деструкторът на наследника извиква деструктора на базовия клас**

```cpp
int main()
{
    //с миналия пример подсказахме в какъв ред
    //се извикват деструкторите при наследяване

    //ако сме запомнили [Base] като скрита член-данна,
    //която стои най-отгоре, то редът е по стандартния начин

    //първо се извиква деструктора на ~Der(), след което
    //тези на всички член-данни в обратния ред, в който са декларирани, т.е.
    //ако сме си представили, че класът има следните член-данни в себе си:
    //Base  ^
    //A     |
    //B     |
    //C     | - ред на викане на деструкторите

    std::cout << "[der]" << std::endl;
    Der der;
    std::cout << std::endl;


    return 0;
}
```





Bonus:

```
//ако добавим член-данна от тип [A] в [Base]
class Base
{
private:
    A a;
    int x = 0;
public:
    Base()
    {
        std::cout << "Base()" << std::endl;
    }

    Base(int x)
    {
        std::cout << "Base(x)" << std::endl;
        this->x = x;
    }

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

```
int main()
{
    //след деструктора на [Base], следвайки реда
    //в предния пример, ще се извика и този на [A],
    //който да зачисти член-данната от тип [A] в класа [Base]

    std::cout << "[der]" << std::endl;
    Der der;
    std::cout << std::endl;


    return 0;
}
```

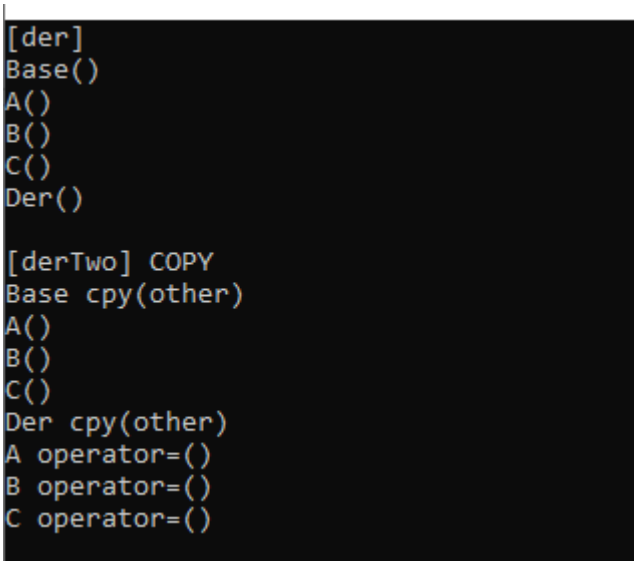
```
Microsoft Visual Studio Debug Console
[der]
A()
Base()
A()
B()
C()
Der()

~Der()
~C()
~B()
~A()
~Base()
~A()
```

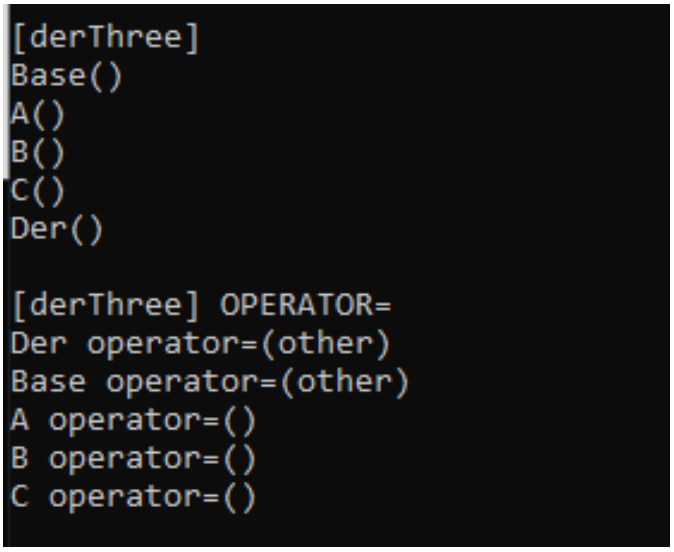
Копиране при наследяване

```
//[!] В наследниците трябва да се грижим САМО за член-данните/член-функциите
//на наследника => copyFrom() и free() копират и изчистват само тях

Der(const Der& other): Base(other) //използваме вече имплементирания копиращ консруктор на [Base],
                                   //за да копираме и [Base] частта, т.е.
                                   //за да осигурим че всички член-данни, наследени от базовия клас,
                                   //са коректно копирани

                                   //освен копиращия конструктор на [Base], ще се
                                   //извикат и default-ните на [A], [B], [C], тъй като
                                   //създаваме нов обект и не са създадени
{
    std::cout << "Der cpy(other)" << std::endl;
    copyFrom(other); //ще извика operator= на [A], [B], [C] (за да ги копираме)
}
```

```
Der& operator=(const Der& other)  //напомняме, че тук няма да се извикат
                                  //default-ните на [A], [B], [C], тъй като
                                  //вече обекта съществува (просто го манипулираме)
{
    std::cout << "Der operator=(other)" << std::endl;
    if (this != &other)
    {
        Base::operator=(other); //аналогично използваме вече имплементирания operator= на [Base],
                                //за да се погрижим за [Base] частта
        free();
        copyFrom(other);  //ще извика operator= на [A], [B], [C] (за да ги присвоим)
    }

    return *this;
}
```

```
int main()
{
    std::cout << "[der]" << std::endl;
    Der der;
    std::cout << std::endl;


    std::cout << "[derTwo] COPY" << std::endl;
    Der derTwo = der;
    std::cout << std::endl;

    std::cout << "[derThree]" << std::endl;
    Der derThree;
    std::cout << std::endl;

    std::cout << "[derThree] OPERATOR=" << std::endl;
    derThree = der;
    std::cout << std::endl;


    return 0;
}
```





Припомняме

- **вече видяхме, че можем изсмучем началото на наследника, т.е. при копиращия конструктор на Der, Base приема наследника и изсмуква началото му (частта на Base)**

- **В наследниците се грижим САМО за нещата, свързани с тях, а не се грижим за наследения клас**

- **copyFrom() е функция и в двата класа, но тъй като е private се избягва конфликта на имена + вече знаем, че Base НЕ подозира за съществуването на Der(shadow-ват се)**

- Копиране при наследяване


```
Der(const Der& other) : Base(other)
    copyFrom(other); → прилича само за Der
}
```

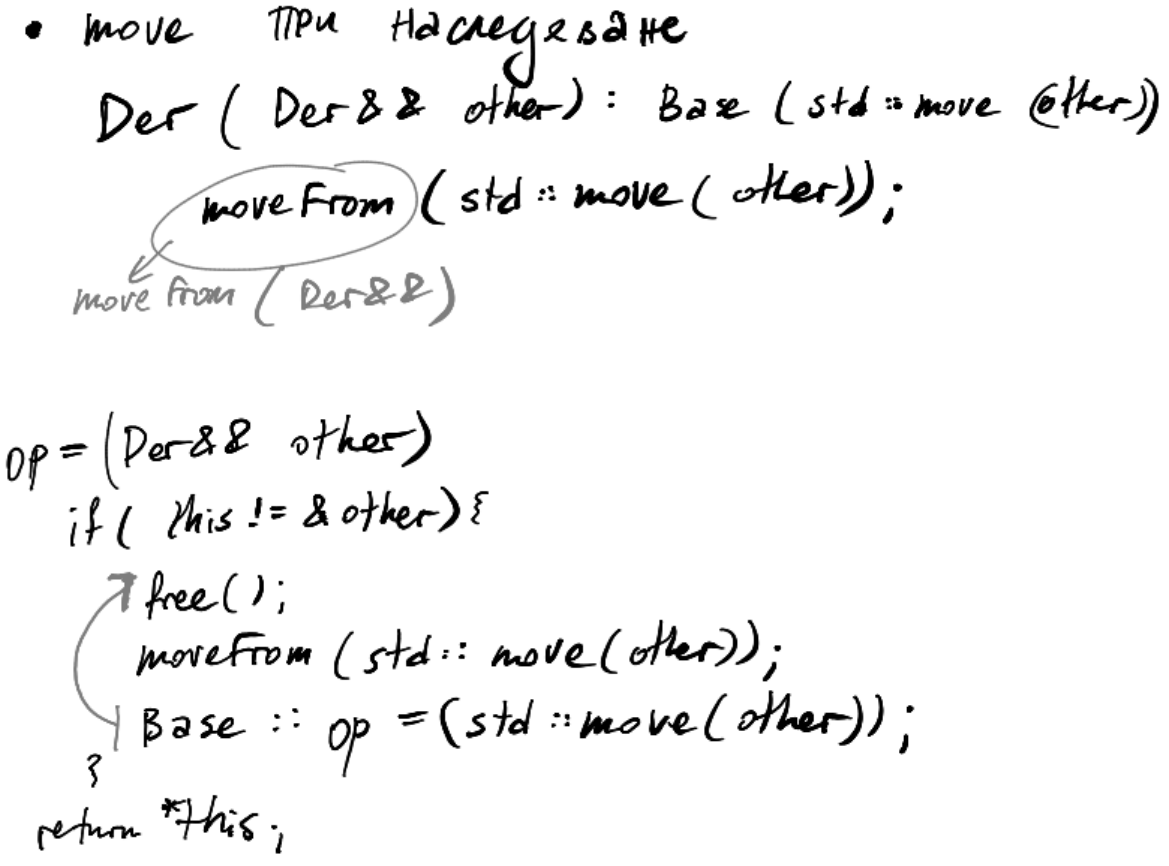
Base(const Base&)

взима само нужната Base

```
op=(const Der& other)
    if (this != &other) {
        free(); → само данните на Der
        copyFrom(other);
        Base::operator=(other);
    }
    return *this;
}
```

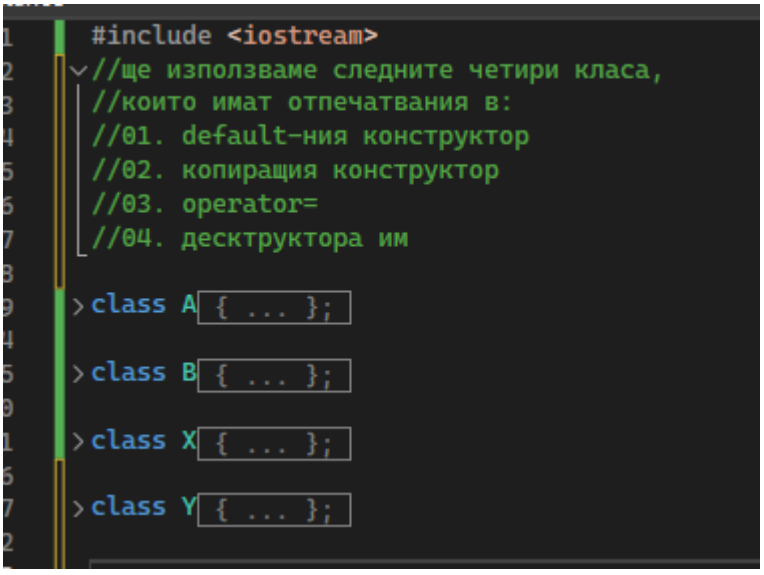
Move при наследяване

```cpp
//аналогично на копиращия конструктор и предишния operator=

Der(Der&& other) noexcept: Base(std::move(other)) //отново се грижим само за член-данни/функции
{                                                 //САМО на наследника

                                                  //за да се придържим към move семантиката, ще използваме
                                                  //и move конструктора на [Base]


    moveFrom(std::move(other));
}

Der& operator=(Der&& other) noexcept
{
    if (this != &other)
    {
        Base::operator=(std::move(other)); //за да се придържим към move семантиката, ще използваме
                                           //и operator= на [Base] (който КРАДЕ)
        free();
        moveFrom(std::move(other));
    }
    return *this;
}
```

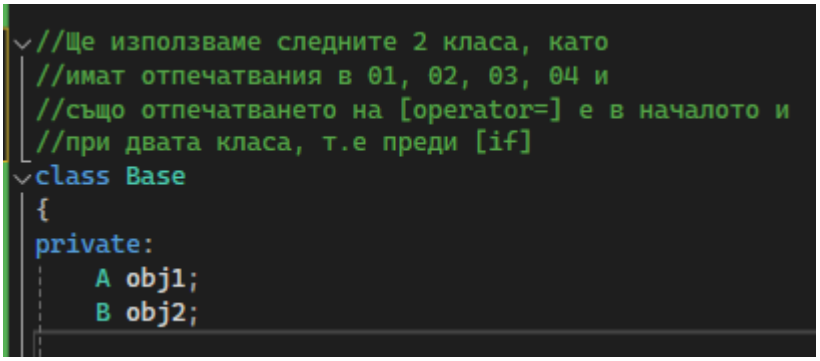


При теория задачи

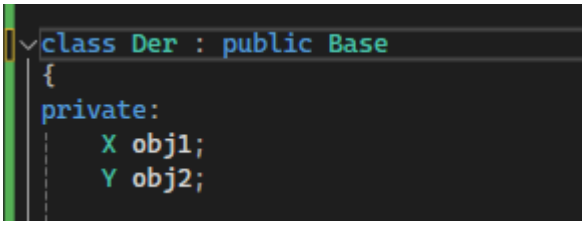
- **внимаваме дали отпечатването при operator= е в края или в началото (т.е. преди или след if-а)**



```
#include <iostream>
//ще използваме следните четири класа,
//които имат отпечатвания в:
//01. default-ния конструктор
//02. копиращия конструктор
//03. operator=
//04. десктруктора им

class A { ... };

class B { ... };

class X { ... };

class Y { ... };
```



```
//Ще използваме следните 2 класа, като
//имат отпечатвания в 01, 02, 03, 04 и
//също отпечатването на [operator=] е в началото и
//при двата класа, т.е преди [if]
class Base
{
private:
    A obj1;
    B obj2;
```



```
class Der : public Base
{
private:
    X obj1;
    Y obj2;
```

```cpp
int main()
{
    //Вече разгледахме викането на конструкторите при наследяването

    //Припомняме:
    //01. извиква се конструктора на наследника [Der]
    //02. извиква се конструктора на наследения клас [Base]
    //03. в него се извикват default-ните конструктори на член-данните му
    //04. връщаме се в [Der], където се извикват конструкторите на член-данните му

    //[Der] се извиква първи, но се отпечатва последен, тъй като преди да стигнем
    //до отпечатването се извикват всички останали конструктори

    Der d1;
    std::cout << std::endl;
    Der d2;
    std::cout << std::endl;

    //тук тъй като отпечататването е преди [if-a], то първото нещо, което ще се отпечата е
    //< Der operator=(other) >, след което, тъй като не са едни и същи обекти, влизаме в тялото на [if-a]
    //и извикваме < Base operator=(other) >. В него чрез copyFrom() на [Base] извикваме operator= на [A], [B], връщаме се
    //в копиращия конструктор на [Der] и извикваме copyFrom функцията на [Der], която извиква operator= на [X], [Y]

    d1 = d2;
    std::cout << std::endl;
    return 0;
}
```

```
A()
B()
Base()
X()
Y()
Der()

A()
B()
Base()
X()
Y()
Der()

Der operator=(other)
Base operator=(other)
A operator=()
B operator=()
X operator=()
Y operator=()
```

```cpp
int main()
{

    Der d1;
    std::cout << std::endl;
    Der d2;
    std::cout << std::endl;
    d1 = d2;
    std::cout << std::endl;


    return 0;

} //вече разгледахме викането на деструкторите
  //=> ~Der() ~Y() ~X() ~Base() ~B() ~A()
  //01. вика се конструктора на [Der]
  //02. викат се деструкторите на член-данните му една по една
  //03. стига до зачистването на въображаемата ни член-данна [Base]
  //04. викат се деструкторите на  член-данните на [Base] една по една

  //това ще се повтори два пъти, тъй като сме създали два обекта от тип [Der]
```

```
~Der()
~Y()
~X()
~Base()
~B()
~A()
~Der()
~Y()
~X()
~Base()
~B()
~A()
```

```cpp
int main()
{
    //ако изместим отпечатването на op= на [Der] и [Base]
    //след if-а, то разликата е, че просто
    //< Der operator=(other) > и < Base operator=(other) >
    //ще се отпечатат след извикването на operator= на техните член-данни
    //и съответно отпечатването на operator= на техните член-данни

    Der d1;
    std::cout << std::endl;
    Der d2;
    std::cout << std::endl;
    d1 = d2;
    std::cout << std::endl;


    return 0;

}
```

```cpp
Der& operator=(const Der& other)
{
    if (this != &other)
    {
        Base::operator=(other);
        free();
        copyFrom(other);
    }

    std::cout << "Der operator=(other)" << std::endl;
    return *this;
}
```

```cpp
Base& operator=(const Base& other)
{
    if (this != &other)
    {
        free();
        copyFrom(other);
    }

    std::cout << "Base operator=(other)" << std::endl;
    return *this;
}
```

```
A operator=()
B operator=()
Base operator=(other)
X operator=()
Y operator=()
Der operator=(other)
```

В този случай се отпечатват първо всички **operator=**, които се извикват от член-данните на **Der**. Това се случва и с **Base**. А не първо този, извикан от самите тях

отпечатване в началото (в нар. на if)

отпечатване в края (след if)

Base

A obj1;

B obj2;

Der

X obj1;

Y obj2;

по дефо:
Base = op=
free
copyFrom

{ Der d1; [A(), B(), Base(), X(), Y(), Der()

} ~Der(), ~Y(), ~X(), ~Base, ~B(), ~A()]

{

Der d1; [конструктори

Der d2; [конструктори

d1 = d2; OP=Der, OP=Base, OP=A, OP=B, OP=X, OP=Y (в началото печатане)

} 2x деструктори] (в края печатане)

OP=A OP=B OP=Base, OP=X, OP=Y
OP=Der

+ още 2 случая