

Обектно-ориентирано програмиране (записки)

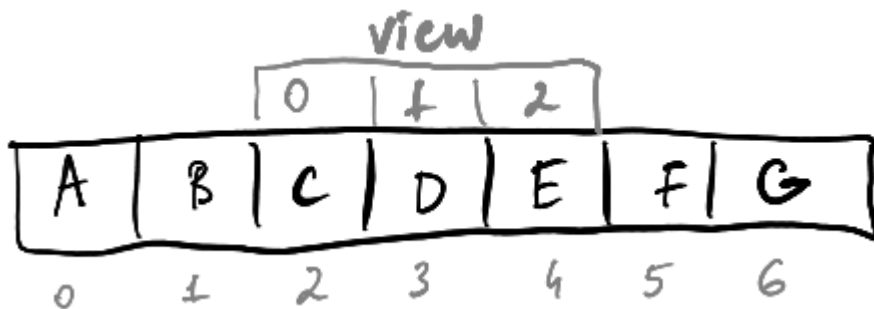
- **Марина Господинова**
- **Илиан Запрянов**

Тема 11.Изгледи и полиморфизъм

Изгледи

def| Клас, който се ползва за преглед на интервал от колекции

С други думи изгледите са инструменти, които действат като прозорци към данните. Те ни позволяват да виждаме и да манипулираме данни, които се съхраняват някъде другаде, без да създаваме излишни нови копия на тези данни.



```
ism (Global Scope)
#include <iostream>

int main()
{
    //за примера се налага да използваме [string], тъй като
    //обикновените [char] низове, които сме свикнали да
    //използваме не поддържат .substr()
    std::string str = "Hello, World";

    str.substr(7, 5);
    //      ^  ^
    //      |  |
    // начален | | брой
    // индекс -| |- символи

    //substr() връща нов [string], не променя този, с който сме извикали функцията,
    //което значи, че трябва да запазим резултата от .substr в нов [string]
    std::string res = str.substr(7, 5);

    //=> като извод се нуждаем да създадем нов [string]
    //и съответно да заделим нова памет за него,
    //а ние не искаме това

    std::cout << str << std::endl; //Hello, World (не се променя)
    std::cout << res << std::endl; //World (запазили сме резултата от .substr)

    return 0;
}
```

```
Hello, World
World
```

Идеята на **StringView** е чрез изглед към част от **string** да избегнем това излишно заделяне на памет и създаването на нов обект

https://github.com/Angeld55/Object-oriented_programming_FMI/tree/master/Week%2010/String%20and%20StringView

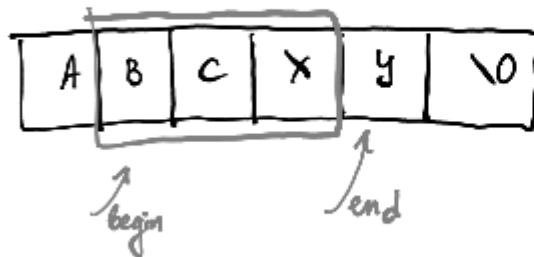
Има два варианта за член-данните на **StringView** 01

01. указател към началото на стринга + указател към края на стринга
02. указател към началото на стринга + дължина (брой символи)

Имаме предвид, че е прието подниза да се взима чрез интервал във вида
[start, end)

Бележки за **StringView**

String View → за изглед на част от string



1. указател за начало и
указател за край

2. указател за начало и
дължина

- $get\ size \rightarrow (end - start) / \underline{sizeof(T)}$ → ако не е char
с размер 1 byte

- два конструктора

- оператор $[]$ (size-t index) const

```
{ return - begin[index];  
}
```

- оператор $<<$

→ с този клас няма да заделяме нова памет

Полиморфизъм

def | Едно име на функция, но много различни имплементации

```
1  #include <iostream>
2
3  //[Пример] Function overloading
4  void f()
5  {
6      std::cout << "f()" << std::endl;
7  }
8
9  void f(int a)
10 {
11     std::cout << "f(int)" << std::endl;
12 }
13
14 void f(int a, int b)
15 {
16     std::cout << "f(int, int)" << std::endl;
17 }
18
19 int main()
20 {
21     int a = 3;
22     int b = 4;
23
24     //имаме няколко функции с едно и също име,
25     //но имплементациите са различни, т.е.
26     //имаме функция с името "f", която
27     // - не приема нищо
28     // - приема едно цяло число (int)
29     // - приема две цели числа (int, int)
30     f();
31     f(a);
32     f(a, b);
33
34
35     return 0;
36 }
```

def. Compile time polymorphism - по време на компилация се определя коя функция да се извика

- function overloading
- operator overloading

operator overloading - например при оператора за събиране (+), имаме много видове събирания. Например събирането на низове е различно от това на цели числа, едното очаква конкатенация, а другото сбор.

$3+4$
"ABC" + "yx" } различен +

```
class Base
{
public:
    void f()
    {
        std::cout << "Base::f()" << std::endl;
    }
};

class Der : public Base
{
public:
    void f()
    {
        std::cout << "Der::f()" << std::endl;
    }
};

int main()
{
    Der d;

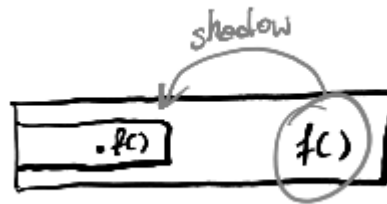
    //в случая, тъй като [Der] наследява [Base] публично,
    //то де факто имаме две едни и същи функции с еднакво име,
    //като имаме достъп и до двете

    //когато извикаме функцията f() на обект от тип [Der],
    //то това ще извика член-функцията f() на [Der], т.е.
    //функцията f() на [Der] ще "shadow"-не тази на [Base]
    d.f();

    //ако искаме да извикаме функцията f() на [Base], то
    //е необходимо да конкретизираме, че искаме да извикаме точно нея
    d.Base::f();

    return 0;
}
```

```
Microsoft Visual Studio
Der::f()
Base::f()
```



```
void k(Base* ptr) //[Base] pointer param
{
    ptr->f(); //[!] [ptr] е от тип [Base] =>
    //ще извика член-функцията f() на [Base]
}

void s(Der* ptr) //[Der] pointer param
{
    ptr->f(); //[!] [ptr] е от тип [Der] =>
    //ще извика член-функцията f() на [Der]
}

int main()
{
    Der d;

    //припомняме, че [Base] стои в началото на [Der]
    k(&d); //k() ще вземе само [Base] частта на [Der]
    s(&d); // k() ще вземе [Der]

    //[!] същото е и при подаване по референция

    return 0;
}
```

```
Microsoft Visual Studio
Base::f()
Der::f()
```

```

int main()
{
    Der d;

    //насочваме [Der] поинтер към [d]
    Der* ptr = &d;
    ptr->f(); //=> ще извика f() функцията на [Der]

    //насочваме [Base] поинтер към [d]
    Base* ptr2 = &d;
    ptr2->f(); //=> ще извика f() функцията на [Base]

    return 0;
}

```

Microsoft V

Der::f()
Base::f()

Статично свързване

def. | **Статично свързване**

- изборът на функция става при време на компилация
- определя се от: **типа на указателя/референцията**, от който се извиква функцията

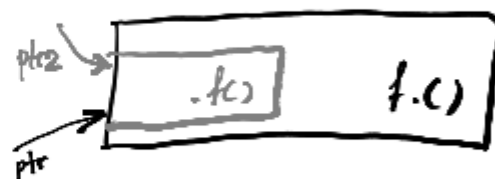
Предните примери са примери и за статично свързване

```

Der d;
Der* ptr = &d;
ptr->f(); // Der::f()

Base* ptr2 = &d;
ptr2->f(); // Base::f()

```



Динамично свързване

def. | **Динамично свързване**

- изборът на коя функция да се извика става по време на изпълнение на програмата (**run-time polymorphism**)
- чрез виртуални функции

```
class Base
{
public:
    void f()
    {
        std::cout << "Base::f()" << std::endl;
    }
};

class Der : public Base
{
public:
    void f()
    {
        std::cout << "Der::f()" << std::endl;
    }
};

int main()
{
    //припомняме следния пример

    Der d;
    d.f(); //ще се извика функцията f() на [Der]

    Base* ptr = &d; //ще вземе [Base] частта на [Der]
    ptr->f(); //ще извика функцията f() на [Base]

    //това обаче води до отрязване на някаква информация
    //при насочването на поинтъра [ptr] към [Der], тъй като
    //[ptr] е от тип [Base]

    //макар и да насочваме поинтъра [ptr] към [Der],
    //той взема [Base] частта, тоест имаме поинтър, който
    //де факто насочваме към [Der], но сочи към [Base]

    return 0;
}
```



```

1  #include <iostream>
2
3  class Base
4  {
5  public:
6      virtual void f()
7      {
8          std::cout << "Base::f()" << std::endl;
9      }
10 };
11
12 class Der : public Base
13 {
14 public:
15     void f()
16     {
17         std::cout << "Der::f()" << std::endl;
18     }
19 };
20

```

```

int main()
{
    //ако искаме да нямаме това поведение, т.е. да
    //отрязваме информация, това става чрез ключовата дума
    //[!] virtual

    //тя ни позволява решението коя функция да бъде извикана
    //да става по време на изпълнение, т.е. вместо
    //предварително да види какъв е типа на пойнтъра/референцията,
    //компиляторът гледа към какво сочи/с какво е свързана и извиква
    //най-конкретната функция

    Der d;
    d.f(); //стандартно извикване на функцията f()

    Base* ptr = &d; //макар [ptr] да е пойнър от тип [Base]
    //      [d] е от тип [Der]

    //компиляторът по време на изпълнение ще види, че пойнтъра
    //соchi към обект от тип [Der] и ще извика функцията f() на [Der]
    //(функцията f() в типа на обекта към когото сочи)

    ptr->f(); //повтаряме: въпреки, че пойнтъра е от тип [Base],
    //компиляторът ще види, че той сочи към [Der] =>
    //ще се извика функцията f() на [Der]

    return 0;
}

```

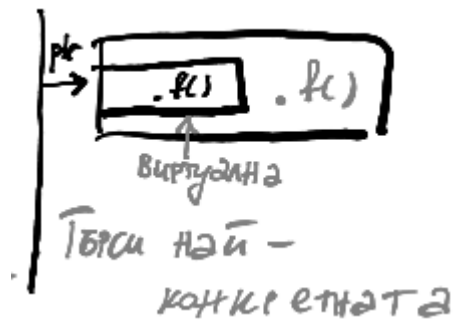
Microsoft Visual St

```

Der::f()
Der::f()

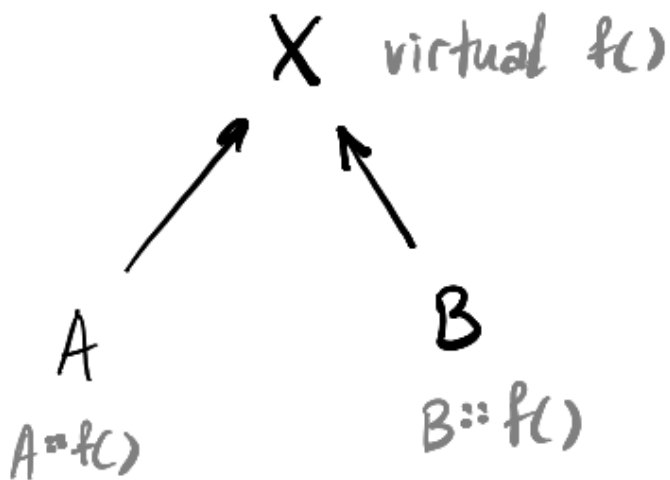
```

Можем да кажем, че **виртуалните функции** търсят най-конкретната функция и изпълняват нея



В случая видяхме, че сочим към **Der**, **Der** има **f()** в себе си => извиква нея. Ако **Der** **НЯМАШЕ** такава функция компилаторът щеше да продължи да търси най-конкретната нагоре по йерархията.

Нека имаме следната йерархия



```
class X
{
public:
    virtual void f()
    {
        std::cout << "X::f()" << std::endl;
    }
};
```

```

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

class B : public X
{
public:
    void f()
    {
        std::cout << "B::f()" << std::endl;
    }
};

```

```

void func(X* ptr)
{
    //подавайки [temp], не можем
    //да бъдем сигурни коя от двете
    //функции ще се извика
    ptr->f();
}

int main()
{
    X* temp = nullptr;

    //rand() се смята по време на изпълнение
    //=> не знаем [temp] към какъв обект сочи предварително
    //(от тип [A] или [B])
    if (rand() % 2 == 0)
    {
        temp = new A();
    }

    else
    {
        temp = new B();
    }

    func(temp);
    delete temp;

    return 0;
}

```

! Понякога е run-time не знаем коя ще се извика

?? $A::f$, $B::f$, ($X::f$ - ако има още един клас ~~члвр.~~)

```
ymorphism (Global Scope)
1  #include <iostream>
2
3  class Base
4  {
5  public:
6      virtual void f()
7      {
8          std::cout << "Base::f()" << std::endl;
9      }
10 };
11
12 class Der: public Base
13 {
14 public:
15     void f() override //добра идея е да използваме ключовата дума [override],
16                       //която сигнализира, че презаписваме виртуална функция
17
18                       //при грешка в синтаксиса дава компилационна грешка
19     {
20         std::cout << "Der::f()" << std::endl;
21     }
22 };
23 int main()
24 {
25
26     return 0;
27 }
```

Пример

Ще използваме следните два класа със следните имплементации

```
class Base
{
public:
    virtual void f()
    {
        std::cout << "Base::f()" << std::endl;
    }

    Base()
    {
        f();
    }

    ~Base()
    {
        f();
    }

    void g()
    {
        f();
    }
};
```

```
class Der : public Base
{
public:
    void f() override
    {
        std::cout << "Der::f()" << std::endl;
    }
};
```

```

int main()
{
    //Създаваме обект от тип [Der], който наследява [Base],
    //тоест се викат default-ните конструктори на [Der] и [Base]
    //
    //вече знаем, че default-ният конструктор на [Der] ще извика този на [Base] =>
    //ще влезем в тялото на Base(), където се извиква функцията f(), но тъй като
    //обектът ни от тип [Der] още не е напълно създаден, то ще се извика функцията
    //f() на класа [Base], а не тази на [Der]

    Der d; //Base::f()

    //тук вече обектът ни от тип [Der] вече е напълно създаден и ще се извика
    //функцията f() на [Der]

    d.g(); //Der::f()

    //разбрахме, че когато имаме виртуална функция се гледа към какво сочи поинтъра,
    //а не от какъв тип е => макар [ptr] да е от тип [Base], [ptr] сочи към обект от тип [Der]
    //и ще се извика функцията f() на [Der]

    Base* ptr = &d;

    ptr->g(); //Der::f()

    return 0;
} //извикват се деструкторите по стандартния начин ~Der() ~Base()
//но след като обектът ни от тип [Der] вече е изтрит и се извика деструктора на [Base]
//вътре в тялото си ~Base() извиква f(), тъй като [Der] вече е изтрит, то ще се извика тази на [Base]
//=> ще се отпечата Base::f()

```

def. | Чисто виртуална функция (pure virtual function)

- функция, която няма имплементация
- предназначена да бъде пренаписана от наследниците

def. | Абстрактен клас

- клас, който има поне една чисто виртуална функция и е предназначен за наследяване
- ако чисто виртуалната функция не се разпише от наследник и той става абстрактен

```

class Base //абстрактен клас
{
public:
    virtual void f() = 0;
};

class A: public Base
{
public:
    void f() override
    {
        //
    }
};

class B : public Base
{
public:
    void f() override
    {
        //
    }
};

class C : public Base //тъй като [C] не override-ва
//функцията f() на [Base]
{
    // //=> C също е абстрактен клас
};

```

(global scope)

```
#include <iostream>

class A
{
public:
    virtual void f() = 0 //чисто виртуалните функции могат да имат тяло,
                        //но НЕ можем да я извикаме извън класовете
    {
        std::cout << "A::f()" << std::endl;
    }
};

class B : public A
{
public:
    void f() override
    {
        A::f(); //извиква се чрез оператора за ре
    }
};

int main()
{
    B b;
    b.f(); //A::f()

    return 0;
}
```


Ключовата дума **final**

- указва, че дадена виртуална функция не може да се презаписва надолу по йерархията
- за класове - указва че даден клас не може да се наследява

```
class Base //абстрактен клас
{
public:
    virtual void f() = 0;
};

class B: public Base
{
public:
    void f() override
    {
        //
    }
};

class C final: public B //[C] не може да бъде наследен
{
public:
    void f() override
    {
        //
    }
};

class D : public C //[X]
{
    //
};
```

class C
[C] не може да бъде наследен
Size: 8 bytes

```
class Base //абстрактен клас
{
public:
    virtual void f() = 0;
};

class B: public Base
{
public:
    void f() override
    {
        //
    }
};

class E : public B
{
public:
    void f() override final //f() не може да бъде презаписана от наследник на [D]
    {
    }
};

class F :public E
{
public:
    void f() override //[X]
    {
    }
};
```

inline virtual void F::f() override
Search Online
cannot override 'final' function "E::f" (declared at line 21)

Виртуални таблици

Намирането на подходящата функция, която трябва да се извика се случва чрез **виртуални таблици** - “масив от указатели към функции”

- всеки клас, който има виртуални функции има своя виртуална таблица - в нея пише коя функция трябва да се извика
- всеки обект има виртуален указател като допълнителна член-данна (в началото на класа), която сочи към виртуалната таблица на класа и влияе на размера му (8 байта за 64-битова система)
- Невиртуалните функции не са в тези виртуални таблици
- Деструкторът, от друга страна, е в тази таблица, затова задължително при полиморфизъм деструкторът е виртуален, в противен случай - memory leak (достатъчно е да кажем само на този на класа най-отгоре на йерархията да бъде виртуален, останалите ще се направят по подразбиране)

```
class Base
{
public:
    virtual void f();
    virtual ~Base(); //(!] при полиморфизъм деструктора е virtual (поне този на базовия клас)
};

class B: public Base
{
public:
    void f() override
    {
        //
    }

    //ще се направи virtual сам, тъй като този на [Base] е виртуален
};
```

Нека вземем следните три класа:

```
class A
{
public:
    virtual void f()
    {
        std::cout << "A::f()" << std::endl;
    }

    virtual void g()
    {
        std::cout << "A::g()" << std::endl;
    }
};
```

```
class B : public A
{
public:
    void f() override
    {
        std::cout << "B::f()" << std::endl;
    }
};
```

```
class C : public B
{
public:
    void f() override
    {
        std::cout << "C::f()" << std::endl;
    }

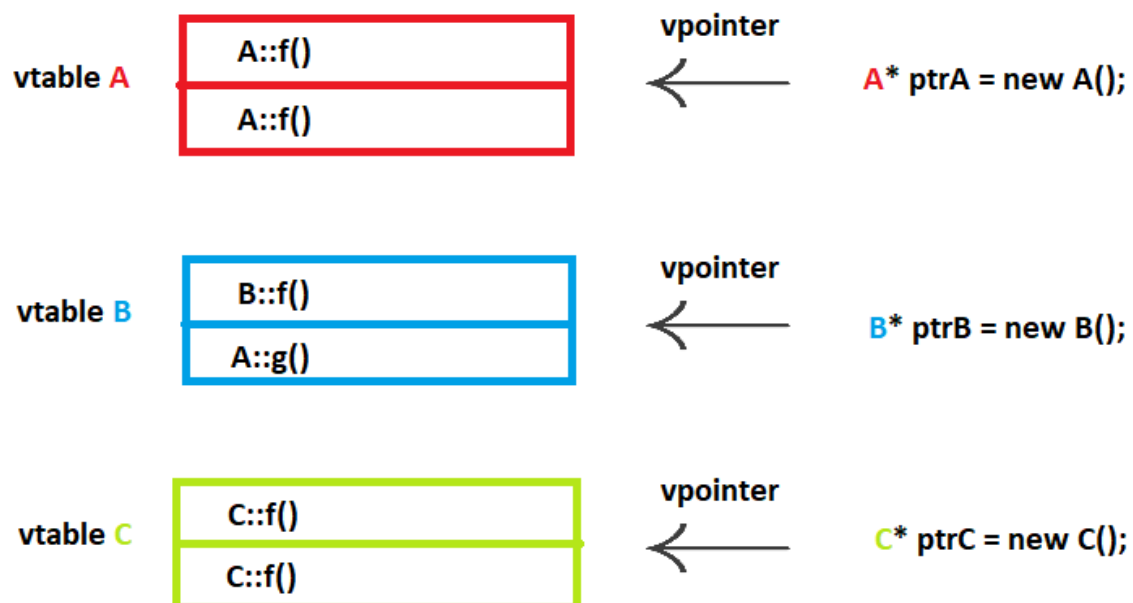
    void g() override
    {
        std::cout << "C::g()" << std::endl;
    }
};
```

Виждаме, че класът **B** не презаписва функцията `g()` на **A**, въпреки че е виртуална. С видяното до тук можем да направим извода, че когато извикаме функцията `g()` чрез поинтър, сочещ към обект от тип **B**, то ще се извика функцията `g()` на **A**, тъй като е най-конкретната, връщайки се нагоре по йерархията, защото **B** няма такава.

Виждаме, че класът **C** презаписва функцията `g()` на **A**. С видяното до тук можем да направим извода, че когато извикаме функцията `g()` чрез поинтър, сочещ към обект от тип **C**, то ще се извика функцията `g()` на **C**, тъй като е най-конкретната, връщайки се нагоре по йерархията, защото **C** има такава.

Това се случва, благодарение на виртуалните таблици и техните виртуални поинтъри, които можем да визуализираме по следния начин:

- всеки клас, който има виртуални функции, има своя виртуална таблица. В нея пише коя функция трябва да се извика
- всеки обект има виртуален указател, който сочи към виртуалната таблица на класа



```
int main()
{
    A* ptrA = new A();
    ptrA->f();
    ptrA->g();
    std::cout << std::endl;

    A* ptrB = new B();
    ptrB->f();
    ptrB->g();
    std::cout << std::endl;

    A* ptrC = new C();
    ptrC->f();
    ptrC->g();

    delete ptrA;
    delete ptrB;
    delete ptrC;

    return 0;
}
```

Microsoft

A::f()

A::g()

B::f()

A::g()

C::f()

C::g()