

Обектно-ориентирано програмиране (записки)

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

Тема 04. Член-функции

def.] Член-функции наричаме

- всяка функция в тялото на структура/клас
- функции, които работят с член-данныте на обекта от дадена структура

```
#include <iostream>

struct Point
{
    int x;
    int y;
    bool isInFirstQ() //създаваме член-функция, която работи с полетата [x], [y]
    {
        //и проверява дали точката (x,y) е в първи квадрант

        return x >= 0 && y >= 0;
    }
};

int main()
{
    Point p{ 7, 9 }; //първи начин за инициализация

    p.isInFirstQ(); //достъпваме член-функциите на дадена инстанция/обект,
                    //както сме свикнали да достъпваме член-данныте

    Point p2 = { 2, 4 }; //втори начин за инициализация

    Point* p3 = new Point{ 3, 5 };

    std::cout << p3->isInFirstQ() << std::endl;
    std::cout << (*p3).isInFirstQ() << std::endl;

    delete p3; //[!]

    //напомняме начините за инициализация на променлива от тип Point (инстанция/обект)
    //както и достъпването на полета чрез пойнтър към инстанция/обект

    //напомняме: struct -> инстанция
    //           class -> обект

    return 0;
}
```

```
#include <iostream>

struct A
{
    void f()
    {
        std::cout << "f()" << std::endl;
    }

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

int main()
{
    A a; // макар и член-функциите f() и g() да са създадени вътре в структурата A
          // те се преобразуват във външни функции с ДОПЪЛНИТЕЛЕН параметър, който
          // е УКАЗАТЕЛ към обекта, върху който се извършва функцията [!] this [!]

    std::cout << sizeof(A) << std::endl; //тъй като казахме, че функциите се преобразуват
                                         //във външни, това означава, че те де факто не са
                                         //в A и не променят паметта, заделена за
                                         //структурата A, т.е.
                                         //A в случая е празна => sizeof(A) = 1

    return 0;
}
```

```

#include <iostream>

struct A
{
    int x = 7;
    void print()
    {
        std::cout << x << std::endl;
    }
};

int main()
{
    A a; //когато функцията print() се преобразува във външна функция с ДОПЪЛНИТЕЛЕН параметър,
    //който е УКАЗАТЕЛ към обекта, върху който се извършва функцията [!] this [!],
    //този поинтър е КОНСТАНТЕН, с цел винаги да сочим към обекта, който манипулираме

    //с други думи print() -> print(A* const this) [външна]
    //                                         ^
    //                                         |
    //                                         запазена дума

    a.print(); //създадохме член-функция, която отпечатва член-данната [x] на инстанцията [A]

    return 0;
}

```

Резюме:

Член-функциите:

- извикват се от обект (трябва да имаме такъв)
- работят директно с член-данныте (променливите, които са в обекта)
- Компилаторът преобразува всяка **член-функция** на дадена структура в обикновена функция с уникално име и един допълнителен параметър – **константен указател към обекта**.

```

//изключително важно при член-функциите е
//спазването на валидния преход non-const -> const
//и избягването на невалидния преход const -> non-const
//тъй като не можем да направим никаква константа на променлива
struct A
{
    int x = 10;
    void printInvalid()
    {
        std::cout << x << std::endl;
    }

    void printValid() const
    {
        std::cout << x << std::endl;
    }
};

void testF(const A& ref) //функцията testF приема константна референция
{
    //към променлива от тип A, но при викането на
    //функцията printInvalid()
    //вътре в тялото на testF, по никакъв начин не гарантираме, че КОНСТАНТНАТА
    //референция няма да бъде променена, т.е. няма да се компилира

    //ref.printInvalid();

    ref.printValid(); //обещаването, че функцията извикана в тялото на testF
    //няма да промени по никакъв начин КОНСТАНТНАТА референция
}
    // се извършва чрез ключовата дума [const], написана след изписването на
    //необходимите параметри за успешното изпълнение на функцията, т.е.
    //func(<параметри>) const {<тяло>}

A a;
testF(a); //testF -> printValid -> 10

```

```

//Резюме:
//[1] Константните член-функции не могат да променят член-данныте
//[2] Могат да се извикват от константни обекти/референции/указатели
struct A
{
    int x = 10;
    void printValid() const
    {
        std::cout << x << std::endl;
    }
};

```

```

//Разгледаният горе пример може да доведе до извода,
//че в КОНСТАНТНИТЕ член-функции в структура или клас могат
//да се извикват САМО член-функции на структурата/класа
//които са КОНСТАНТНИ
struct A
{
    void f();           //f() може да извика в себе си функциите g() и t(),
                         //тъй като не сме обещали, че няма да се променят,
                         //и функцията f() не държи на това, извиканите в нея функции
                         //да не променят член-данныте

    void g() const; //тук g() и t() могат да се извикат помежду си, но не можем да
    void t() const; //извикаме f() в тях, тъй като сме обещали, че нищо в инстанцията
};                      //няма да бъде променено във функциите g() и t(), което не се гарантира
                         //от функцията f()

                         //това означава, че можем да извикаме единствено t() в g() и
                         //обратното, тъй като g() обещава, че нищо няма да бъде променено
                         //и в нея викаме друга функция t(), която също обещава, че нищо
                         //няма да бъде променено (по същата причина в t() можем да извикаме g())

```

```

//можем да имаме две функции с еднакви имена,
//но едната да обещава нищо в инстанцията да не бъде променено

#include <iostream>

struct A
{
    void f()
    {
        std::cout << "non-const" << std::endl;
    }

    void f() const //обещаваме, че нищо няма да се промени
    {
        std::cout << "const" << std::endl;
    }
};

int main()
{
    //в този случай, вземането на решение коя от двете функции да се извика,
    //зависи единствено от вида на инстанцията/обекта

    A obj; //инстанцията е НЕКОНСТАНТНА => ще се извика функцията,
               //която НЕ ГАРАНТИРА, че нищо няма да се промени
    obj.f();

    const A obj2; //инстанцията е КОНСТАНТНА => ще се извика функцията,
                     //която ГАРАНТИРА, че нищо няма да се промени
    obj2.f();
}

```

```
#include <iostream>

struct A
{
    void f() const //обещаваме, че нищо няма да се промени
    {
        std::cout << "const" << std::endl;
    }
};

//вече знаем, че всяка член-функция се преобразува във външна,
//когато функцията е константна (не променя нищо) това означава,
//че освен пойнтъра (допълнителният параметър), инстанцията, към която сочи
//също е константна => можем да кажем, че

//void f() const ~ void f(const A* const this)
//  

//void f(const A* const this)
//    - this не може да бъде променен (винаги сочи към тази инстанция)
//    - данните на инстанцията не могат да бъдат променени
//    - т.е. константен указател към константна структура

int main()
{
    A obj;
    obj.f();

    return 0;
}
```

01. Конструktури и деструктури

```
//ВЪВЕДЕНИЕ
#include <iostream>

struct P
{
    int x;
    int y;
};

int main()
{
    P point{ 3,4 }; //при създаване на променлива от тип P,
                      //се извършват два основни процеса
                      // [1] заделя се памет за променливата
                      // [2] задават се стойности на променливите

    return 0;
}

} //в края на scope-а (при статични инстанции/обекти)
//се извършват също два процеса
// [1] паметта се освобождава
// [2] изчиства външните ресурси

//двета процеса обозначени с [2] се наричат съответно
//с КОНСТРУКТОР и ДЕСТРУКТОР
```

Жизнен цикъл на обект:

01. Създаване на обект в даден scope – заделя се памет и член-данныте се инициализират.
02. Достига се до края на scope-а (област).
03. Обектът и паметта, която е асоциирана с него се разрушава.

01. Конструктор

- Извиква се веднъж - при създаването на обекта.
- Има същото име като класа.
- Няма тип на връщане
- Можем да имаме няколко конструктора
- Конструктор, който е без параметри се нарича default-ен конструктор (конструктор по подразбиране)
- Задава стойности на член-данныте на class-а (в тялото си или чрез member initializer list)

```
#include <iostream>

struct A
{
    int a;
    int b;

    A(int x) { a = b = x; } //конструктор
    A(int aParam, int bParam) { a = aParam, b = bParam; } //конструктор
};

int main()
{
    A obj{ 3,4 };    //припомняме, че
                      // [1] заделя се памет за променливата
                      // [2] задават се стойности на променливите (извиква се конструктора)

    A obj2{ 3 };

    //тук [obj] ще приеме стойности чрез конструктора, който приема 2 параметъра
    //тук [obj2] ще приеме стойности чрез конструктора, който приема 1 параметър
    //тоест компютърът се ориентира сам кой конструктор за извика,
    //в зависимост от подадените параметри

    return 0;
}
```

```

#include <iostream>

struct A
{
    int a;
    int b;

    A()
    {
        a = 0;
        b = 0;
    }

    A(int x) { a = b = x; }
    A(int aParam, int bParam) { a = aParam, b = bParam; }
};

int main()
{
    A objs[5]; //при декларация на масив ВИНАГИ се извика default-ният конструктор
                //за всяка една клетка на масива => default-ният конструктор в случая
                //ще се извика 5 ПЪТИ

                //![ВАЖНО!] Когато направим наш конструктор, компютърът НЕ създава default-ен
                //тоест в случая заради другите 2 конструктора сме длъжни да разпишем и default-ния.
                //В противен случай няма да се компилира, поради твърдението в предходния коментар

                //ако НЯМАМЕ никакви наши конструктори в инстанцията, компютърът създава default-ен такъв
                //![!] само тогава

    return 0;
}

```

```

struct A
{
    //това се прави от компютъра
    //A()
    //{
    //    //////
    //}

    int a;
    int b;
};

int main()
{
    A obj; //нямаме конструктор => компютърът ще създаде такъв
    return 0;
}

```

```
#include <iostream>

struct A
{
    A()
    {
        ////  

    }

    int a;
    int b;
};

int main()
{
    A obj; //викаме default-ния конструктор  
  

    A obj2(); //това е деклариране на ФУНКЦИЯ, която връща A,  

              //НЕ ИЗВИКВА конструктор  
  

    return 0;
}
```

02. Деструктор

- Извиква се веднъж
- Няма тип на връщане
- Има същото име като класа със символа '~' в началото.
- Може да имаме САМО ЕДИН деструктор
- Затваря външни ресурси

```
#include <iostream>

struct A
{
    int a;
    int b;

    A()
    {
        /////
    }

    ~A()
    {
        /////
    }
};

int main()
{
    A obj; //викаме default-ния конструктор (A)

    return 0;
} //викаме default-ният деструктор (~A)
```

```
#include <iostream>

struct A
{
    int a;
    int b;

    A()
    {
        /////
    }

    ~A()
    {
        /////
    }
};

int main()
{
    A objs[3]; //вече знаем че при масиви default-ният конструктор
                //ще се извика 3 пъти (размера на масива)

    return 0;
} //по същия начин ще се извика default-ният деструктор (~A)
   //3 пъти (за всяка една клетка на масива)
```

```

#include <iostream>

struct A
{
    int a;
    int b;

    A()
    {
        /////
    }

    ~A()
    {
        /////
    }
};

int main()
{
    A* objs = new A[3]; //вече знаем че при масиви default-ният конструктор
                        //ще се извика 3 пъти (размера на масива)

    //тъй като масивът е ДИНАМИЧЕН, тоест трябва да изчистим паметта сами
    //няма да се извика деструктор в края на scope-а

    //когато освободим паметта, т.е.
    delete[] objs; //ще се извика 3 пъти деструктора (за всяка клетка)

    //delete objs; //ще се извика 1 път деструктора [!НО!] -> UB!!!

    return 0;
} //в случая не се извиква нищо

```

Резюме:

- **new** - заделя памет И ИЗВИКВА КОНСТРУКТОР
- **delete** - освобождава памет и ИЗВИКВА ДЕСТРУКТОР

Всичко до тук

```
struct Test
{
    char* str;

    void f() const
    {
        str[0] = 'A'; //тъй като str е поинтър към външен ресурс,
    }                      //компилаторът го позволява макар и f() да е const,
                           //но по уговорка НЕ го правим

};
```

```
#include <iostream>

struct Test
{
    //преобразуване като външна функция
    void g() {}           // -> void g(Test* const)
    void f() const {} // -> void f(const Test* const)

};

void f1(const Test& t)
{
    t.f();
    //t.g(); не ни гарантира, че нищо няма да се промени
}

void f2(const Test t)
{
    t.f();
    //t.g(); не ни гарантира, че нищо няма да се промени
    //      без значение, че правим копие, а не подаваме по референция
}
```

```
struct Point
{
    int x, y; //-----|-----|
    //           |       |
    //           v       v
    Point(int x, int y): x(x), y(y) {} //на член-данные [x], [y] присваиваме
    //           |       |     ^     ^      параметрите [x] [y]
    //           |       -----|       |
    //           -----|-----|
    Point() : Point(0, 0) {} //може един конструктор да извика друг
    // (обыкновено при массивы)

};
```

```
struct Point
{
    int x = 0, y = 0;

    //Point() {}

    Point() = default; //при инициализации член-данные ще
    //используются ключевое слово default
    // (поэтому можно использовать оптимизацию)
};
```

Конвертиращ конструктор

def. Конструктор, който приема точно един параметър

```
#include <iostream>

struct A
{
    int x = 0;

    A(int x) //конвертиращ конструктор
    {
        /**
     }
};

void f(const A& obj)
{
    std::cout << "here" << std::endl;
}

int main()
{
    A a(3); //създаваме променлива от тип a
    f(a);

    f(3); //забелязваме, че макар f() да има за параметър тип A,
           //можем да подадем число

           //това се дължи на КОНВЕРТИРАЩИЯ КОНСТРУКТОР, тъй като, когато подадем
           //различен тип от искания, компилаторът търси начин да превърне подадения тип
           //в желания тип. В случая подаваме число, но искаме тип A

           //Компилаторът вижда КОНВЕРТИРАЩИЯ КОНСТРУКТОР, който иска ЕДИН параметър
           //(в случая число) и позволява на компилатора да създаде сам временна променлива
           //от тип A със стойност 3, след което успешно влизаме в тялото на функцията

    return 0;
}
```

```

#include <iostream>

struct A
{
    int x = 0;

    A(int x) //конвертиращ конструктор
    {
        /**
     }
};

void f(A& obj)
{
    std::cout << "here" << std::endl;
}

int main()
{
    A a(3); //създаваме променлива от тип а
    f(a);

    f(3); //когато махнем const от параметъра на функцията f()
           //това не ни позволява да подаваме rvalue (в случая число),
           //тъй като const е нещото, което ни позволява да създаваме временни обекти.
           //В противен случай би било възможно функцията да промени временен обект,
           //което искаме да ограничим да не е възможно

    return 0;
}

```

Извод:

- **g(const X&)** -> позволява да приемаме **lvalue** и **rvalue**
- **t(X& ref)** -> позволява да приемаме само **lvalue**

```
#include <iostream>

struct A
{
    int x = 0;

    explicit A(int x) //ключовата дума explicit се използва
    {                //само за функции с 1 параметър

        //чрез explicit казваме, че
        //това не е конвертиращ конструктор
    }
};

void f(const A& obj)
{
    std::cout << "here" << std::endl;
}

int main()
{
    A a(3);

    f(a);
    f(3); //макар и f() да приема const, тоест временните инстанции
           //да са възможни, вече нямаме конвертиращ конструктор и
           //няма как да прехвърлим числото 3 в тип A и член-данна x == 3

    return 0;
}
```

Абстракция. Капсулация. Модификатори за достъп. Селектори и мутатори. Mutable.

01. Абстракция

- използваме нещо без да се интересуваме как работи
- скриване на междуинните детайли

02. Капсулация

- ограничаване на достъпа

03. Модификатори за достъп

- `private` - достъп само в класа
- `protected` - достъп в класа и наследниците му
- `public` - неограничен достъп

```
struct A //или class
{
public:
    /**
 */
private:
    /**
 */
};

struct B
{
    /**
     *no default - public
};

class C
{
    /**
     *no default - private
};

//(!) ЕДНА от основните разлики между struct и class е,
//че при struct, достъпът по default е неограничен, докато
//в class всичко по default е private
```

Уговорка:

- големи обекти - **class**
- малки обекти - **struct**

04. Селектори и мутатори

- селектор - `get()`
- мутатор - `set()`

`get()` - връща копие/константна референция/константен указател

`set()` - позволява промяна, но под контрол

Уговорка:

- примитивни типове - **копие**
- обекти - **константна референция/пойнтър**

05. Mutable

- `mutable` член-данни
- могат да бъдат променяни дори и в константни функции

Mutable член-данните не влияят на видимото състояние на обекта.

Използват се **САМО** в краен случай и трябва да се аргументираме защо

```

#include <iostream>

class A //помним, че класовете по default са private
{
    //капсуляция (чрез модификаторите за достъп),
    //казваме кое искаме да е достъпно и кое не

    //модификатор за достъп
private:
    mutable int num; //не искаме пряк достъп до член-данините
    int numTwo;

    //модификатор за достъп
public:
    A(int num) : num(num) {}

    //селектор
    int getNum() //не е необходимо да подаваме нищо, искаме
    {
        //да достъпим член-данината [num], която е достъпна
        //в рамките на класа

        return num;
    }

    //мутатор
    void setNum(int a) //подаваме число, което искаме да присвоим на
    {
        //член-данината [a]
        //функцията може да е void или bool в зависимост от
        //начина, по който ще валидирате дали данните са валидни

    }
    void f() const // <-
    {
        // |
        ++num; // | макар и да сме обещали да не променяме инстанцията
        ++numTwo; // | променливата [num] е mutable => можем да променим само
    } // | и единствено нея, т.е. не можем да променим numTwo =>
    // | няма да се компилира
};


```

Конструктори и деструктори при композиция

```
struct A
{
    int a;
    A() {} //default
    A(int n) : a(n){}
};

struct B
{
    int b;
    B() {} //default
    B(int n) : b(n){}
};

struct C
{
    int c;
    C() {} //default
    C(int n) : c(n){}
};
```

```
struct X
{
    A a; // -----
    B b; // | ред на викане на конструкторите на A, B, C
    C c; // v
    //

    X(int a) : a(a + 1), b(a + 2), c(a+3){} //при композиция на инстанции,
    //           ^                               най-външният е отговорен за създаването на останалите
    //           |                               в случая X е отговорен за A, B и C
    //           |

    //След двоеточието изписваме кои конструктори на A, B, C искаме да се извикат,
    //ако не упоменем кой конструктор на A искаме да извикаме например, ще се извика
    //default-ният, а ако няма такъв, няма да се компилира, тъй като A не може да се
    //създаде. Редът на създаване НЕ зависи от начина, по който сме ги избрали след
    //двоеточието, а от начина, по който сме ги ДЕКЛАРИРАЛИ, тъй като някоя инстанция
    //може да е зависеща от предишната => гледаме ----

    //тъй като X отговаря за A, B, C, конструкторът на X ще се извика пръв, но няма да влезе в тялото му,
    //докато A, B, C не бъдат създадени => редът на успешно изпълнение е A -> B -> C -> X

    X() {} //DEFAULT + вика default-ните на A, B, C, тъй като не сме упоменали кои да извика
};
```

```

struct X
{
    A a; // ^ <-----  

    B b; // | ред на викане на деструкторите на A, B, C  

    C c; // |  

    //  

    ~X(){} // при композиция на инстанции,  

    // триенето започва в обратна посока, тоест отвътре-навън  

    // в случая X е отговорен за A, B и C => ~C, ~B, ~A, ~X  

    // Деструкторът на X ще се извика пръв, но X няма да се изтрие, докато ~C, ~B, ~A не се изпълнят  

    // Редът на триене ЗАВИСИ САМО от начина, по който сме ги ДЕКЛАРИРАЛИ, => гледаме -->  

};
```

```

struct A {
    A(int x) {}
};

struct Y {
    A obj;
    Y() = default;
    Y(int x) : Y(), obj(x) {} //не знае дали да присвои default-натата стойност на obj,
} ; //или да му присвои [x], т.е. не знае по какъв начин да инициализира obj,  

    //тъй като и двата конструктора след [:] инициализират obj  

    //=> конфликт => няма да се компилира

    //с други думи няма как да прехвърлим отговорността на друг конструктор
    //и едновременно с това да инициализираме член-данната в текущия
```