

# **Обектно-ориентирано програмиране (записки)**

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

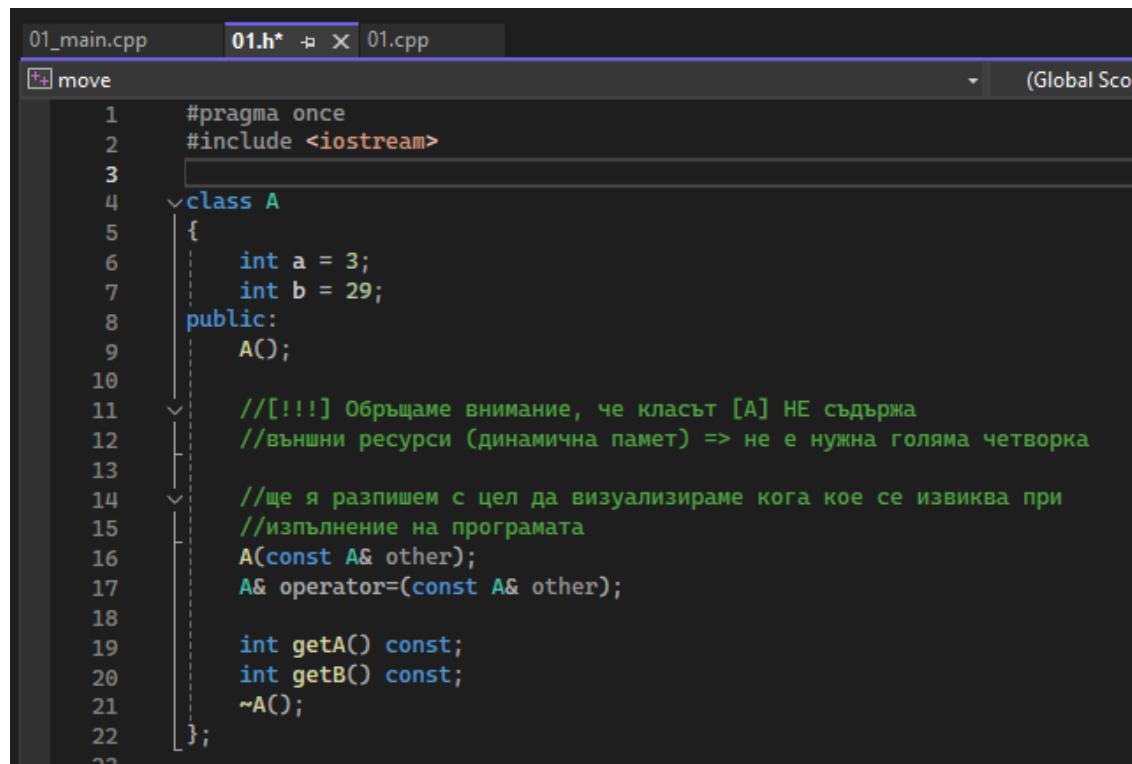
## Тема 09. Масиви от указатели към обекти. Move семантики.

**def.** Колекция от обекти - клас, който съдържа колекция от еднотипни обекти

Нека разгледаме със знанията досега как бихме направили функциите за манипулация на нашата колекция от обекти от тип [A] в класа [X]:

- add
- remove
- at
- pop\_back
- resize

Клас [A]



The screenshot shows a code editor window with three tabs at the top: "01\_main.cpp", "01.h\*", and "01.cpp". The "01.h\*" tab is active. The code editor displays the following C++ code:

```
01_main.cpp 01.h* 01.cpp
move (Global Scope)

1 #pragma once
2 #include <iostream>
3
4 class A
5 {
6     int a = 3;
7     int b = 29;
8 public:
9     A();
10
11    //!!! Обръщаме внимание, че класът [A] НЕ съдържа
12    //външни ресурси (динамична памет) => не е нужна голяма четворка
13
14    //ще я разпишем с цел да визуализираме кога кое се извиква при
15    //изпълнение на програмата
16    A(const A& other);
17    A& operator=(const A& other);
18
19    int getA() const;
20    int getB() const;
21    ~A();
22};
```

The code defines a class A with private members a and b, and public methods A(), operator=(const A& other), getA() const, and getB() const. It also contains two comments: one warning about the class not containing external resources and another explaining the purpose of the implementation details.

## Клас [X]

```
23
24     <class X
25     {
26         A* arr;          //колекция от еднотипни обекти
27         size_t size;   //текущ размер
28         size_t cap;    //максимален размер
29
30         void copyFrom(const X& other);
31         void free();
32
33     public:
34         X();
35
36         X(const X& other);
37         X& operator=(const X& other);
38
39         //интерфейс
40         void resize(size_t newCap);
41         void add(const A& obj);
42         void remove(size_t idx);
43         void pop_back();
44         const A& at(size_t index) const;
45
46         ~X();
47     };
48
49
```

Ще пропуснем разглеждането на функциите различни от петте изброени горе

-add

```
113     <void X::add(const A& obj)
114     {
115         //проверка, ако вече сме стигнали максималния
116         //възможен размер
117         if (this->size == this->cap)
118         {
119             throw std::out_of_range("X::add arr is full");
120         }
121
122         //добавяме новия елемент и увеличаваме текущия размер,
123         //тъй като успешно сме минали валидацията =>
124         //е възможно да добавим нов елемент
125         arr[size] = obj;
126         this->size++;
127     }
```

## -remove

```
128     ✓void X::remove(size_t idx)
129     {
130         //проверка, ако индексът на елемента, който искаме да
131         //премахнем надхвърля текущия размер
132
133         //#[NOTE] не е нужно да проверяваме дали [idx] е
134         //отрицателно, тъй като типът [size_t] е [unsigned],
135         //т.е., ако подадем отрицателно число, то ще се превърне
136         //в положително такова
137         if (idx >= this->size)
138         {
139             throw std::out_of_range("X::remove invalid index");
140         }
141
142         //в случая извикването на деструктора не е нужно,
143         //тъй като A() не съдържа член-данни с динамична памет,
144         //ако имаше такива, щеше да е нужно да освободим паметта,
145         //преди да махнем елемента от масива
146
147         //![!] arr[idx].~A();
148         this->size--;
149
150         //тъй като премахваме елемент, е необходимо да запълним
151         //образувалата се празна позиция, като преместим всички елементи
152         //с една позиция наляво
153         for (size_t i = idx; i < this->size - 1; i++)
154         {
155             arr[idx] = arr[idx + 1];
156         }
157     }
158
159 }
```

## -at

```
160
161     ✓const A& X::at(size_t idx) const
162     {
163         //валидация, ако търсим елемент извън масива
164         if (idx >= this->cap)
165         {
166             throw std::out_of_range("X::at invalid index!");
167         }
168
169         return this->arr[idx];
170     }
```

- **pop\_back**

```
172     ~void X::pop_back()
173     {
174         // [Отново] в случая извикването на деструктора не е нужно,
175         // тъй като A() не съдържа член-данни с динамична памет,
176         // ако имаше такива, щеше да е нужно да освободим паметта,
177         // преди да махнем елемента от масива
178
179         // [!] this->arr[this->size - 1].~A();
180         this->size--;
181     }
182 }
```

- **resize**

```
85     ~void X::resize(size_t newCap)
86     {
87         // Идея:
88         // 01. Създаваме нов масив с новия максимален размер (newCap)
89         // 02. Запазваме елементите на стария [към който arr сочи] в новия масив [към който newArr сочи]
90         // 03. Освобождаваме паметта за стария масив [към който arr сочи]
91         // 04. Пренасочваме по-търъла [arr] към нова памет (тази на новия масив)
92         // 05. Обновяваме член-данната [cap] (запазваме новия максимален размер)
93
94         // 01.
95         A* newArr = new A[newCap] {};
96
97         // 02.
98         for (size_t i = 0; i < this->size; i++)
99         {
100             newArr[i] = this->arr[i];
101         }
102
103         // 03.
104         delete[] arr;
105
106         // 04.
107         arr = newArr;
108
109         // 05.
110         this->cap = newCap;
111     }
112 }
```

## Ограничения и проблеми:

01. Вече знаем, че при масиви от обекти винаги се извиква default-ния конструктор на типа на масива => на [A] му трябва default-ен конструктор

- **БЕЛЕЖКА:** може да се избегне ако заделим паметта за нещо друго предварително и върху нея извикаме желания от нас конструктор

**def.| Placement new** - приема вече заделена памет и извиква конструкторите върху нея

```
size_t objsCount = 4;
//01. заделяме паметта за нещо друго (създаваме масив от [objsCount] на брой обекта от тип [A])
//
//    операторът [operator new[]] служи за
//    01. съхранение на указания брой обекти от дадения тип
//    02. в комбинация с placement new можем да пропуснем автоматичната инициализация на обектите
//        (викането на default-ния конструктор A() за всяка клетка на масива член-данна на класа)
//    и вместо това само резервирате блока памет, което в бъдеще ни позволява да извикаме конструкторите на обектите ръчно
//
//    |
//    v
void* memory = operator new[](sizeof(A)* objsCount);
//    ^
//    |
//създаваме пойнър          размер на байтовете, които искаме да заделим,
//от тип [void]           в случая искаме да заделим sizeof(A) * objsCount байта,
//                        които ще ни стигнат за [objsCount] на брой обекти, т.е. 4
//пойнъра [memory]
//е [void], тъй като
//[void] пойнърите могат да
//сочат към всякакъв вид памет
```

```
A* array = (A*)memory; //след успешното предварително заделяне на паметта
//прехвърляме пойнъра [memory] от [void] към [A]
//и го присвояваме на масива, който ще използваме [arr]
```

С цел да извикаме различен конструктор различен от default-ния, ще създадем нов такъв, който се извиква с подаването на параметри

```
public:
    A();
    A(int a, int b) : a(a), b(b) {};
```

```

for (size_t i = 0; i < objsCount; i++)
{
    //вече имаме масивът [arr], за който сме заделили памет
    //достатъчна за 4 обекта от тип [A]
    //=> на първата итерация масивът ни има вида [] [] [] []

    //чрез оператора [new] и специфичен адрес, вече можем да
    //конкретизираме коя клетка на масива чрез кой конструктор на [A] да се създаде
    new (&array[i]) A(3, 4);
    //           ^           ^
    //           |           |
    // адресът на      викаме конструктора, с който искаме да
    // клетка [i]      създадем [i]-тата клетка

    //#[NOTE]оператора new + специфичен адрес = placement new
}

```

```

//след всички итерации масивът придобива вида [A(3, 4)] [A(3, 4)] [A(3, 4)] [A(3, 4)] =>
std::cout << array[0].getA() << std::endl; //3
std::cout << array[0].getB() << std::endl; //4

```

```

//тъй като сме заделили памет динамично, е необходимо
//да освободим паметта ръчно след като приключим работата си с [arr]

//имаме предвид, че ако [A] съдържаше външни ресурси (динамична памет)
//щеше да е необходимо да освободим първо паметта заделена за всяка клетка поотделно,
//преди да освободим паметта за [array]
//
//for (size_t i = 0; i < objsCount; i++)
//{
//    array[i].~A(); // извикване на деструктора за всяка клетка от тип [A]
//}

//освобождаваме паметта заделена с operator new[]
operator delete[](array);

```

## 02. При resize:

```
void X::resize(size_t newCap)
{
    //Идея:
    //01. Създаваме нов масив с новия максимален размер (newCap)
    //02. Запазваме елементите на стария [към който arr сочи] в новия масив [към който newArr сочи]
    //03. Освобождаваме паметта за стария масив [към който arr сочи]
    //04. Пренасочваме пойтъра [arr] към нова памет (тази на новия масив)
    //05. Обновяваме член-данната [cap] (запазваме новия максимален размер)

    //01.
    A* newArr = new A[newCap] {};

    //02.
    for (size_t i = 0; i < this->size; i++)
    {
        newArr[i] = this->arr[i];
    }

    //03.
    delete[] arr;

    //04.
    arr = newArr;

    //05.
    this->cap = newCap;
}
```

създават се нови обекти с default-ния конструктор на A(), т.е. тук отново имаме проблемът разгледан в 01., който може да се избегне по същия начин

(във for цикъла се извиква operator=, като запазваме старите данни в новия уговорен масив)

## 03. Бавни размествания:

- ако създадем нова функционалност, която да размества две клетки на масива член-данна, то това ще извика допълнително

**01 вариант** - един път копиращия конструктор, два пъти operator= и съответно веднъж деструктора на създадения чрез копиращия конструктор обект от тип [A]

**02 вариант** - един път конструктор на A (в примера default-ния), три пъти operator= и съответно веднъж деструктора на създадения чрез конструктора обект от тип [A]

```

public:
    X();
    X(const X& other);
    X& operator=(const X& other);

    //интерфейс
    void resize(size_t newCap);
    void add(const A& obj);
    void remove(size_t idx);
    void pop_back();
    const A& at(size_t index) const;
    void swap(size_t idxFirst, size_t idxSec);
    ~X();
};


```

## 01 вариант

```

void X::swap(size_t idxFirst, size_t idxSec)
{
    if (idxFirst >= this->size || idxSec > +this->size)
    {
        throw std::out_of_range("X::swap invalid index!");
    }

    //A temp = this->arr[idxFirst];
    //this->arr[idxFirst] = this->arr[idxSec];
    //this->arr[idxSec] = temp;

    std::swap(this->arr[idxFirst], this->arr[idxSec]);
}

```

before swap  
 Acopy()  
 Aop=  
 |Aop=  
 ~A()  
 after swap

## 02 вариант

```

void X::swap(size_t idxFirst, size_t idxSec)
{
    if (idxFirst >= this->size || idxSec > +this->size)
    {
        throw std::out_of_range("X::swap invalid index!");
    }

    A temp;
    temp = this->arr[idxFirst];
    this->arr[idxFirst] = this->arr[idxSec];
    this->arr[idxSec] = temp;
}

```

before swap  
 A()  
 Aop=  
 |Aop=  
 ~A()  
 after swap

04. махане на елемент на даден индекс (remove)

```

void X::remove(size_t idx)
{
    //проверка, ако индексът на елемента, който искаме да
    //премахнем надхвърля текущия размер

    //#[NOTE] не е нужно да проверяваме дали [idx] е
    //отрицателно, тъй като типът [size_t] е [unsigned],
    //т.e., ако подадем отрицателно число, то ще се превърне
    //в положително такова
    if (idx >= this->size)
    {
        throw std::out_of_range("X::remove invalid index");
    }

    //в случая извикването на деструктора не е нужно,
    //тъй като A() не съдържа член-данни с динамична памет,
    //ако имаше такива, щеше да е нужно да освободим паметта,
    //преди да махнем елемента от масива

    //![!] arr[idx].~A();
    this->size--;

    //тъй като премахваме елемент, е необходимо да запълним
    //образувалата се празна позиция, като преместим всички елементи
    //с една позиция наляво
    for (size_t i = idx; i < this->size - 1; i++)
    {
        arr[idx] = arr[idx + 1];
    }
}

```

```

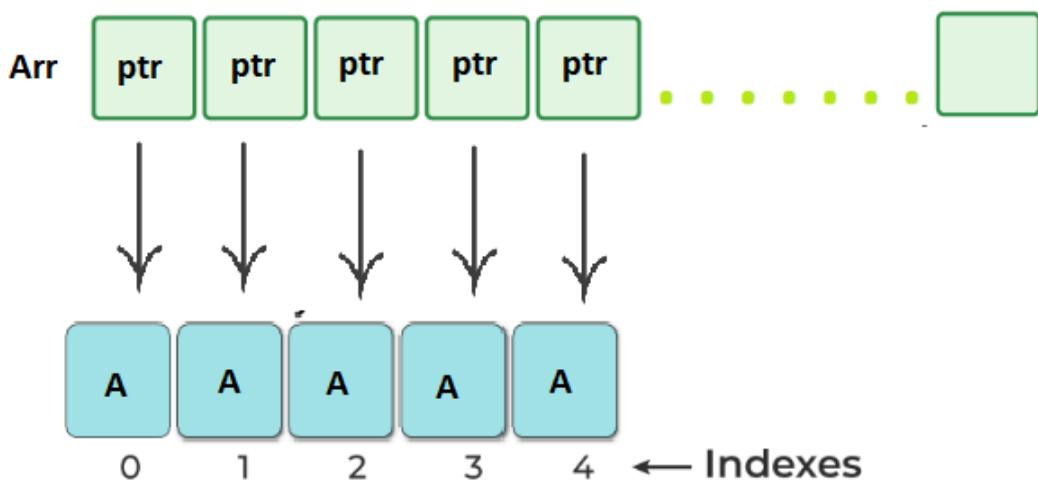
//с една позиция наляво
for (size_t i = idx; i < this->size - 1; i++)
{
    arr[idx] = arr[idx + 1];
}

```

Във **for** цикъла, тъй като е необходимо да слепим наново клетките след махане на една от тях, за да избегнем дупка между тях, ще трябва да извикваме многократно **operator=**

За да премахнем всички тези проблеми използваме:

## Колекция от указатели (към A)



Вместо масивът ни директно да се състои от обекти от [A], ще го направим така, че да се състои от **ПОЙНТЪРИ** към тях.

Разлики и плюсове:

```
~X::X()
{
    this->arr = new A*[8] {nullptr}; //слагаме default-на стойност на всеки поинтър
    this->size = 0;                 //в нашата колекция от поинтъри към [A]
    this->cap = 8;
}
```

-free

```
void X::free()
{
    //преди да освободим паметта за [arr]
    //трябва да освободим паметта за всяка динамично-заделена памет,
    //към която всеки елемент (поинтър) на [arr] сочи
    for (size_t i = 0; i < this->cap; i++)
    {
        delete this->arr[i];
    }

    delete[] this->arr;
}
```

### -copyFrom

```
void X::copyFrom(const X& other)
{
    this->size = other.size;
    this->arr = new A*[this->cap];

    for (int i = 0; i < this->cap; i++)
    {

        if (other.arr[i] == nullptr)
        {
            this->arr[i] = nullptr;
        }
        else
        {
            //насочваме пойнтьра към обект от тип [A],
            //инициализиран чрез копиращия конструктор на [A],
            //подавайки съдържанието на пойнтьра other.arr[i] (ДЕРЕФЕРИРАНЕ)
            this->arr[i] = new A(*other.arr[i]);
        }

        //НЕ дереферираме nullptr,
        //тъй като това ще доведе до [!UB!],
        //затова е необходимо ръчно, ако other.arr[i] е nullptr,
        //да кажем на this->arr[i], че също е nullptr
    }

    this->cap = other.cap;
}
```

### -add

```
void X::add(const A& obj)
{
    //проверка, ако вече стигнали максималния
    //възможен размер
    if (this->size == this->cap)
    {
        throw std::out_of_range("X::add arr is full");
    }

    //добавяме новия елемент и увеличаваме текущия размер,
    //тъй като успешно сме минали валидацията =>
    //е възможно да добавим нов елемент

    //тук вече имаме масив от пойнтьри към [A]
    //=> всяка клетка е пойнтьр

    arr[size] = new A(obj); //насочваме пойнтьра на клетка номер [size]
                            //към динамично-заделен обект, който се инициализира чрез
                            //копиращ конструктор (т.е. A(obj))
    this->size++;
}
```

### -remove

```
void X::remove(size_t idx)
{
    //проверка, ако индексът на елемента, който искаме да
    //премахнем надхвърля текущия размер
    if (idx >= this->size)
    {
        throw std::out_of_range("X::remove invalid index");
    }

    //освобождаваме паметта, заделена за обекта на индекс [idx]
    delete this->arr[idx];

    //правим пойнтьра nullptr, за да
    //не сочи към вече изчистена памет
    this->arr[idx] = nullptr;

    //манхали сме един елемент
    this->size--;
}
```

### -at

```
const A& X::at(size_t idx) const
{
    //валидация, ако търсим елемент извън масива
    if (idx >= this->cap)
    {
        throw std::out_of_range("X::at invalid index!");
    }

    //за разлика от предходния вариант, тук е нужно да дереференцираме пойнтьра,
    //стоящ на клетка номер [idx]
    //(т.е. да върнем стойността на пойнтьра, който се намира на тази клетка)

    return *(this->arr[idx]); //или &arr[idx]
}
```

### -pop\_back

```
void X::pop_back()
{
    //тъй като пойнтьра на последна позиция сочи към
    //динамично-заделена памет, е необходимо да я освободим

    //освобождаваме паметта, заделена за последния обект
    delete this->arr[size - 1];

    //правим пойнтьра nullptr, за да
    //не сочи към вече изчистена памет
    this->arr[size - 1] = nullptr;

    //махнали сме един елемент
    this->size--;
}
```

### -resize

```
void X::resize(size_t newCap)
{
    //Идея:
    //01. Създаваме нов масив от ПОЙНТЬРИ с новия максимален размер (newCap)
    //02. Запазваме ПОЙНТЬРИТЕ на стария в новия масив
    //03. Освобождаваме паметта за стария масив
    //04. Пренасочваме пойнтьра [arr] към нова памет (тази на новия масив)
    //05. Обновяваме член-данната [cap] (запазваме новия максимален размер)

    //01.
    A** newDataPtr = new A*[newCap]{nullptr};

    //02.
    for (int i = 0; i < this->cap; i++)
    {
        newDataPtr[i] = this->arr[i];
    }

    //03.
    delete[] this->arr;

    //04.
    this->arr = newDataPtr;

    //05.
    this->cap = newCap;
}
```

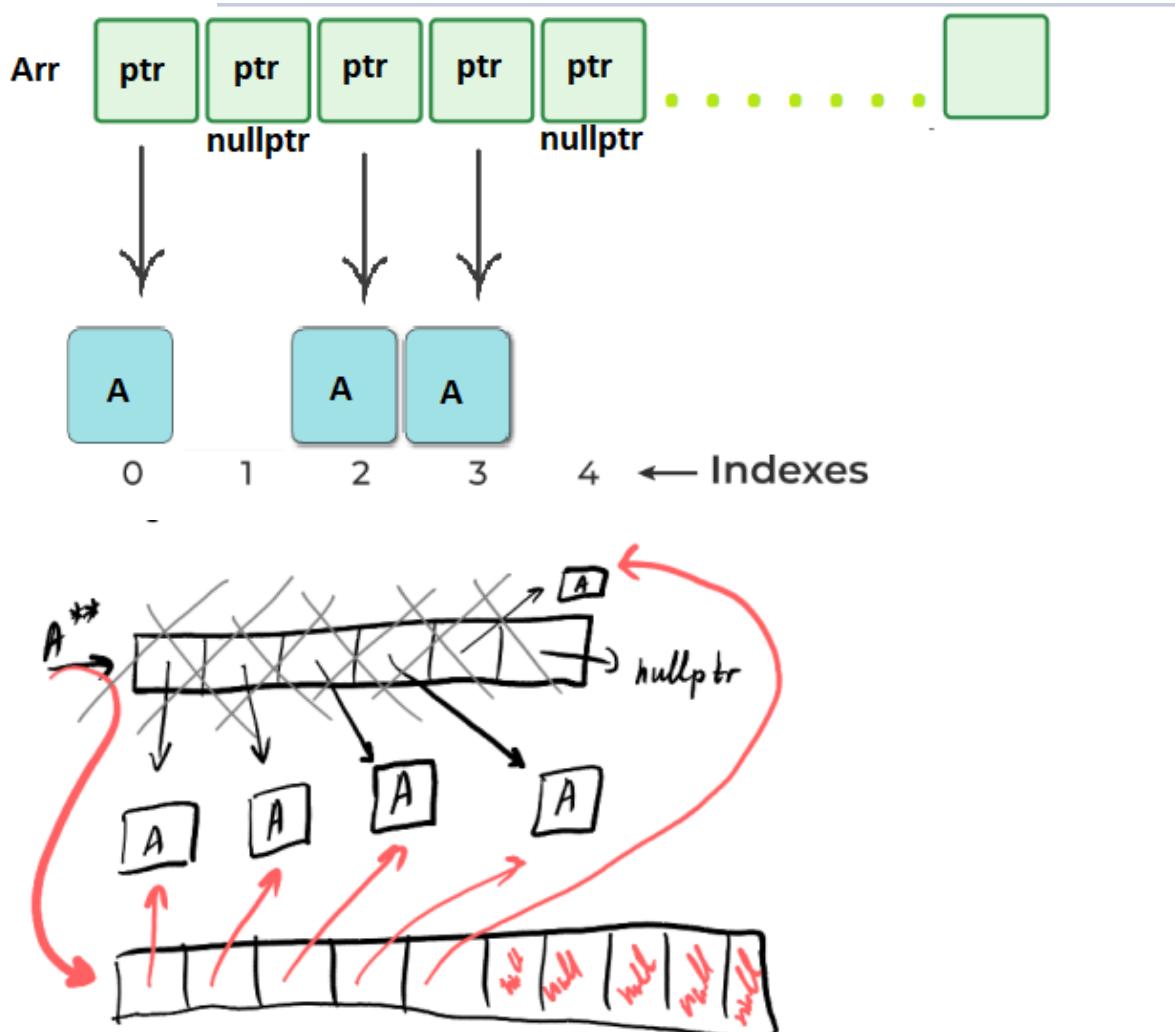
## -swap

```
void X::swap(size_t idxFirst, size_t idxSec)
{
    if (idxFirst >= this->size || idxSec > +this->size)
    {
        throw std::out_of_range("X::swap invalid index!");
    }

    //разменяме пойнтьрите
    std::swap(this->arr[idxFirst], this->arr[idxSec]);
}
```

## Плюсове:

01. Вече не ни трябва default-ен конструктор на A() и не е обходимо да избягваме изикването му, тъй като елементите вече са ни пойнтьри, а **НЕ** обекти от тип A
02. Позволяват ни да имаме празни позиции (**nullptr**). Тоест за разлика от миналата реализация на задачата, тук **НЕ Е** необходимо слепване, този вариант е валиден:



03. Бързи swap-ове - за разлика от миналата реализация тук **НЕ ИЗВИКВАМЕ НИТО КОНСТРУКТОРИ, НИТО КОПИРАЩИ КОНСТРУКТОРИ, НИТО operator=**, тъй като разменяме **ПОЙНТЪРИ**, а не **ОБЕКТИ**, тоест разместването става значително по-бързо.

```
void X::swap(size_t idxFirst, size_t idxSec)
{
    if (idxFirst >= this->size || idxSec > +this->size)
    {
        throw std::out_of_range("X::swap invalid index!");
    }

    //разменяме пойнтърите
    std::swap(this->arr[idxFirst], this->arr[idxSec]);
}
```

Copy()  
before swap  
after swap

04. Нямаме проблемът при **resize**, вече не създаваме нови обекти от тип **A**, а **пойнтъри** към тях

```
void X::resize(size_t newCap)
{
    //Идея:
    //01. Създаваме нов масив от ПОЙНТЪРИ с новия максимален размер (newCap)
    //02. Запазваме ПОЙНТЪРИТЕ на стария в новия масив
    //03. Освобождаваме паметта за стария масив
    //04. Пренасочваме пойтъра [arr] към нова памет (тази на новия масив)
    //05. Обновяваме член-данната [cap] (запазваме новия максимален размер)

    //01.
    A** newDataPtr = new A*[newCap]{nullptr}; //масив от ПОЙНТЪРИ!!!

    //02.
    for (int i = 0; i < this->cap; i++)
    {
        newDataPtr[i] = this->arr[i];
    }

    //03.
    delete[] this->arr;

    //04.
    this->arr = newDataPtr;

    //05.
    this->cap = newCap;
}
```

A\* - в общия случай (използва се повече, поради възползването от **locality**)

**locality** (оптимизация) - възползваме се, че обектите са на **съседни адреси**

Когато достъпим даден елемент от масив (например), то **не** четем само неговите данни, но и тези в дадена околност (чийто размер не знаем) отляво и отдясно => ако достъпим още един елемент, който попада в range-а на първия, вече ще сме му прочели данните и просто ще ги вземем

Index: 0	1	2	3	4	5
Arr[0]	arr[1]	arr[2]	arr[3]	arr[4]	arr[5]

Ако вземем за пример този масив и достъпим **arr[3]**, то прочитаме освен неговите данни и тези на останалите елементи (ако са достатъчно близо в паметта). Тоест ако достъпим **arr[2]** след **arr[3]**, то вече имаме данните на **arr[2]** и просто ги взимаме

A\*\* - “**празни места**” или “**бързи swap-ове**” (в условието на контролно)

# Move семантики

## Типове стойности в C++

### 01. lvalue

def.| lvalue - име на съществуваща променлива/функция

```
#include <iostream>

int main()
{
    int a;
    a = 4; //операторът за присвояване се нуждае от променлива (lvalue) от лявата си страна
           //в този случай това е променливата [a] – името на вече съществуваща променлива

    //обикновено lvalue-тата могат да стоят отлявата страна на оператора за присвояване

    //казваме обикновено, тъй като има изключения, например
    const int x = 10;
    x = 20; // [x] е константа и не може да бъде променена
             // (тоест да стои вляво на оператора)

    return 0;
}
```

```
#include <iostream>

//lvalue може да са и функции, връщащи референция
int x = 3; //глобална променлива

int& f()
{
    return x; //функцията f() връща референция към променливата [x],
               //която по дефиниция е lvalue
}

int main()
{
    f() = 4; //вече казахме, че f() връща референция към [x],
              //следователно f() може да стои от лявата страна на оператора
              //за присвояване, тъй като върнатата референция е lvalue
              //(зашпото е референция към lvalue)

              //казваме, че f() = 4, то тогава и [x] става 4,
              //тъй като f() е върната референция към [x] и казваме
              //на референцията, че има нова стойност 4

              //от УП знаем, че когато променим референцията
              //се променя и променливата, към която сочи

    std::cout << x << std::endl; //4

    return 0;
}
```

```

#include <iostream>

int x = 3;

int getRef()
{
    return x;
}

int main()
{
    int x = 5;

    // [4] и [x+1] не са lvalue, тоест не са имена на
    // вече съществуваща променлива и не могат да стоят вляво
    // на оператора за присвояване

    4 = x;           //Error <-----
    (x + 1) = 4;    //Error
    //
    //
    //

    //ако премахнем връщането на референция на вече
    //разгледаната от нас функция getRef(),
    //то вече тя не връща референция към lvalue,
    //а стойността на [x], която НЕ е lvalue -----
}

getRef() = 4;    //Error

```

## 02. rvalue

израз, който произвежда временна стойност, която обикновено се използва от дясната страна на оператора за присвояване (**не е def**)

**rvalue = prvalue + xvalue**

### prvalue

- > литерали: 73, nullptr, "ABC", false
- > извикване на функция, която връща копие

```
#include <iostream>

int x = 3;

int getRef()
{
    return x;
}

int main()
{
    int i = 42; // [42] е prvalue (литерал)

    // разгледаната от нас функция [!int!] getRef(),
    // връща копие на стойността на
    // [x] => е prvalue

    getRef() = 4; // Error

    i = getRef(); // напомняме, че rvalue може да стои САМО
                  // отляво на оператора за присвояване

    return 0;
}
```

### xvalue

- > expiring value
- > обекти към края на жизнения цикъл

**Бележка:** това обикновено е обект, който е в процес на унищожаване и чиито ресурси могат да бъдат прехвърлени (използва се в **move семантиката**).

08.cpp\* 07.cpp\* 06.cpp 05.cpp 04.cpp 03.cpp delete\_scalar.cpp text.cpp

(Global Scope)

```
1 #include <iostream>
2
3 int f(int& a) //функцията приема променлива по референция
4 {
5     return a;
6 }
7
8 int g(const int& b) //функцията приема променлива по КОНСТАНТНА референция
9 {
10    return b;
11 }
12
13
14 int main()
15 {
16     //това са двата вида подавания по референция,
17     //които сме разглеждали досега
18
19     //тъй като CONST удължава живота на [rvalue],
20     //то тогава можем да подаваме [rvalue]
21     //на КОНСТАНТЕН параметър на дадена функция, т.е
22     int x = 3;
23     f(3); //ERROR параметърът [a] НЕ е константен => НЕ МОЖЕ да му дадем [rvalue]
24     f(x);
25
26     //Извод: функцията f() работи единствено с lvalue
27
28     int y = 4;
29     g(4); //параметърът [b] е константен => МОЖЕ да му дадем [rvalue]
30     g(y);
31
32     //Извод: функцията g() работи както с lvalue, така и с rvalue
33
34     return 0;
35 }
```

**def.] rvalue reference** - може да се прикачи единствено за **rvalue**

(референция, която позволява референция към **rvalue**)

```
#include <iostream>

int k(int&& a) //rvalue reference
{
    return a;
}

int main()
{
    //може да има в случаи в които искаме
    //наша функция да работи единствено с [rvalue]

    //това е възможно благодарение на [rvalue reference],
    //тоест референция към [rvalue]

    //синтаксис: [&&], например:
    int x = 5;
    int&& xRRef = x;    //ERROR] помним, че не може да имаме референция към [lvalue]
                        //при използването на [rvalue reference]

    int&& xRRef = std::move(x); //[[Въведение] към std::move:
                                //благодарение на std::move, можем да превърнем
                                //lvalue (в случая [x]) в xvalue, т.e.
                                //така можем да закачим референция към [x],
                                //защото xvalue е вид rvalue

    k(x); //ERROR] k() изисква [rvalue], тоест НЕ МОЖЕМ да му подадем [x], защото [x] е lvalue
    k(3); //k() изисква [rvalue], тоест МОЖЕМ да му подадем числото 3, тъй като то е rvalue

    k(std::move(x)); //k() изисква [rvalue], тоест МОЖЕМ да му подадем std::move(x),
                     //тъй като std::move(x) превръща x в rvalue,
                     //т.e. std::move(x) прави [x] подходяща за свързване към rvalue reference,
                     //![!] Извод: функцията k() очаква rvalue, както и
                     //очаква подадената променлива/константа да не се използва след функцията

    return 0;
}
```

## Move реализация

```
//вече видяхме семантиката за открадването на данни,
//спестяващо ненужни извиквания на конструктори и operator=

//за да видим как се реализира тази семантика,
//ще използваме следния клас
#pragma once
#include <iostream>

class MyString
{
public:
    MyString();
    MyString(const char* data);

    MyString(const MyString& other);           //copy конструктор
    MyString(MyString&& other) noexcept;        //move конструктор - конструктор за
                                                //"открадване" на данните

    MyString& operator=(const MyString& other); //copy operator=
    MyString& operator=(MyString&& other) noexcept; //move operator=

    size_t getCapacity() const;
    size_t getSize() const;
    ~MyString();

private:
    explicit MyString(size_t stringLength);
    void resize(unsigned newAllocatedDataSize);

    void free();
    void copyFrom(const MyString& other);
    void moveFrom(MyString&& other);

    char* _data;
    size_t _size;
    size_t _allocatedDataSize;
};

};
```

```
#include "MyString.h"
#include <cstring>
#include <algorithm>
#pragma warning (disable : 4996)

static unsigned roundToPowerOfTwo(unsigned v)
{
    v--;
    v |= v >> 1;
    v |= v >> 2;
    v |= v >> 4;
    v |= v >> 8;
    v |= v >> 16;
    v++;
    return v;
}

static unsigned dataToAllocByStringLen(unsigned stringLength)
{
    return std::max(roundToPowerOfTwo(stringLength + 1), 16u);
}

MyString::MyString() : MyString("") {}

MyString::MyString(const char* data)
{
    _size = std::strlen(data);
    _allocatedDataSize = dataToAllocByStringLen(_size);
    _data = new char[_allocatedDataSize];
    std::strcpy(_data, data);
}

MyString::MyString(size_t stringLength)
{
    _allocatedDataSize = dataToAllocByStringLen(stringLength);
    _data = new char[_allocatedDataSize];
    _size = 0;
    _data[0] = '\0';
}
```

## Move и copy конструктор

```
MyString::MyString(const MyString& other)
{
    copyFrom(other);
}

MyString::MyString(MyString&& other) noexcept
{
    //помним, че въпреки че [other] е [rvalue reference],
    //#[other] се третира като [lvalue] в тялото на функцията
    //=> е нужно да използваме std::move, за да
    //го преобразуваме към [xvalue]

    // [Извод]: извън тялото на функцията, [other] се третира като [rvalue reference]
    // и може да бъде свързан с временни обекти, докато в тялото се третира като [lvalue]
    // и изиска std::move за използване на move семантиката
    moveFrom(std::move(other));
}
```

При забраната на **move** конструктора се забранява и **copy** и обратното (същото важи и за **operator=**)

```
//copy конструктор
MyString(const MyString& other) = default;

//move конструктор - конструктор за
//"открадване" на данните
MyString(MyString&& other) noexcept = delete;
```

Тогава, ако искаме единия да го има, а другия не, трябва изрично да го кажем (същото важи и за **operator=**)

## Move и copy operator=

```
MyString& MyString::operator=(const MyString& other)
{
    if (this != &other) {
        free();
        copyFrom(other);
    }
    return *this;
}

MyString& MyString::operator=(MyString&& other) noexcept
{
    if (this != &other)
    {
        free();
        moveFrom(std::move(other));
    }
    return *this;
}
```

```
void MyString::free()
{
    delete[] _data;
}
```

## copyFrom и moveFrom

```
void MyString::moveFrom(MyString& other)
{
    //КРАДЕМ данните на [other]
    _data = other._data;

    //(!) Важно е да кажем на [other._data], че е
    //nullptr, за да нямаме два различни обекта,
    //сочещи към една и съща [data]
    other._data = nullptr;

    _size = other._size;
    other._size = 0;

    _allocatedDataSize = other._allocatedDataSize;
    other._allocatedDataSize = 0;
}

void MyString::copyFrom(const MyString& other)
{
    //КОПИРАМЕ данните на [other]
    _allocatedDataSize = other._allocatedDataSize;

    // [Сравнение]: за разлика от [moveFrom], тук
    // вместо да вземем директно [data] на [other]
    // ние заделяме НОВ масив и КОПИРАМЕ СЪДЪРЖАНИЕТО на [other.data]

    // [Note]: трябва ни нов масив, за да нямаме два различни обекта,
    // сочещи към две различни места в паметта

    // [Извод]: при [move семантиката] резултатът е един обект
    // със съдържание [other.data], при [копиране] имаме два различни
    // обекта с едно и също съдържание (това на other.data), но сочещи
    // към различни места в паметта
    _data = new char[_allocatedDataSize];
    std::strcpy(_data, other._data);
    _size = other._size;
}
```

### Извод за std::move -

01. преобразува **[lvalue]** в **[xvalue]**
02. декларираме, че от подаденото **[lvalue]** можем да крадем, защото няма да се използва след функцията

## Кое кога се извиква

```
#include "02.h"
#include "MyString.h"

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

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

int main()
{
    //Помним, че при функции приемащи [rvalue reference]
    //не можем да приемам lvalue (в случая [obj])
    //A obj;
    //f(obj); [X]

    f(A()); //01. ще създаде обект от тип A чрез default-ния конструктор
            //02. ще подадем създадения обект на f()
            //03. ще отпечатата, че успешно сме влязли в тялото на f()
            //04. ще се извика деструктора на създадения обект от тип A(),
            //      тъй като той съществува единствено в scope-а на функцията
            //=> A() f() ~A()

    //можем да подадем lvalue само, ако
    //го преобразуваме в xvalue
    MyString str;
    g(std::move(str)); //преобразуваме чрез std::move

    return 0;
}
```

```

#include "MyString.h"

int main()
{
    std::cout << "[1]" << std::endl;
    MyString s1 = "ABC"; //инициализираме обекта [s1] директно с "ABC"
                        //това автоматично се съпоставя с конструктора,
                        //който приема const char* като аргумент
                        //=> ще се отпечата MyString(data)

    std::cout << "[2]" << std::endl;
    MyString s2 = std::move(s1); //инициализираме обекта [s2] с преобразуван в [xvalue] - [s1]
                                //това автоматично се съпоставя с конструктора,
                                //който приема [rvalue reference] като аргумент
                                //=> ще се отпечата MyString move()

    std::cout << "[3]" << std::endl;
    MyString s3; //MyString()

    s3 = std::move(s2); //присвояваме на обекта [s3], преобразуван в [xvalue] - [s2]
                        //това автоматично се съпоставя с operator=,
                        //който приема [rvalue reference] като аргумент
                        //=> ще се отпечата MyString move operator=

    std::cout << "[4]" << std::endl;

    return 0;
} //всички създадени обекти се изчистват => 3x ~MyString

```

```

[1]
MyString(data)
[2]
MyString move()
[3]
MyString()
MyString move operator=
[4]
~MyString()
~MyString()
~MyString()

```

```

#include "MyString.h"

void f(MyString str)
{
    std::cout << "f()" << std::endl;
}

int main()
{
    std::cout << "[1]" << std::endl;

    f(MyString("ABC")); //създаваме обект от тип MyString чрез съответния конструктор
                        //и го подаваме на функцията, след нейното изпълнение изтриваме
                        //създадения обект чрез деструктора ~MyString

                        //=> MyString(data) f() ~MyString

    std::cout << "[2]" << std::endl;
    MyString s1 = "ABC"; //MyString(data)

    f(s1); //подаваме по копие => правим копие на [s1]
            //чрез копиращия конструктор, след което
            //след изпълнението на функцията изтриваме новосъздадения
            //обект (копието, което сме направили)

            //=> MyString(data)
            //  MyString copy()
            //  f()
            // ~MyString()

    std::cout << "[3]" << std::endl;
    MyString s2 = s1; //MyString copy()

    f(std::move(s2)); //създаваме нов обект чрез move конструктора,
                      //тъй като сме преместили s2 в [xvalue]
                      //съответно новия обект се трябва след изпълнението на функцията

            //=> MyString copy()
            //  MyString move()

    return 0;
}

} //всички създадени обекти се зачистват => 3x ~MyString

```

```

[1]
MyString(data)
f()
~MyString()
[2]
MyString(data)
MyString copy()
f()
~MyString()
[3]
MyString copy()
MyString move()
f()
~MyString()
~MyString()
~MyString()

```

```
//[!] Въпрос за теория

#include "02.h"

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

int main()
{
    A obj; // A()

    g(std::move(obj)); //тъй като g() приема референция (двойна)
                      //=> няма да се извика нищо, тъй като не трябва
                      //да създаваме нов обект, а да работим с подадения

    //=> A() g() ~A()

    //g(obj); //не се компилира (очаква rvalue)

    return 0;
} // ~A() за [obj]
```

 Micro

```
A()
g()
~A()
```

## Още

```
#include "02.h"

void g(A&& b)
{
    //
}

void f(A&& a) // [a] се третира като [lvalue] в scope-а на функцията,
                // въпреки, че е референция към rvalue
{
    //g(a); // [ERROR] g() очаква [rvalue] => не можем да подадем [a] директно
    g(std::move(a)); // g() очаква [rvalue] =>
                      // трябва да използваме std::move, за да подадем [a]
}

int main()
{
    f(std::move(A()));
    return 0;
}
```

Да се върнем на разгледаната функция **add** в началото

```
void X::add(const A& obj)
{
    //проверка, ако вече сме стигнали максималния
    //възможен размер
    if (this->size == this->cap)
    {
        throw std::out_of_range("X::add arr is full");
    }

    //добавяме новия елемент и увеличаваме текущия размер,
    //тъй като успешно сме минали валидацията =>
    //е възможно да добавим нов елемент
    arr[size] = new A(obj);

    this->size++;
}

void X::add(A&& obj)
{
    //проверка, ако вече сме стигнали максималния
    //възможен размер
    if (this->size == this->cap)
    {
        throw std::out_of_range("X::add arr is full");
    }

    arr[size] = new A(std::move(obj)); //вместо за извикваме постоянно копиращия
                                    //конструктор, може чрез move конструктора
                                    //просто да откраднем данните
    //arr[size] = new A(obj);

    this->size++;
}
```

Вече имаме две функции с името `add`, където:

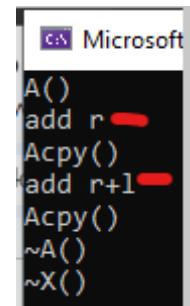
- `add(A&&)` -> очаква `rvalue`
- `add(const A&)` -> очаква `rvalue` ИЛИ `lvalue`

```
X xOne;
A a;

//в зависимост коя е по-конкретната функция,
//ще се извика нея

//в случая add(A&&) очаква [rvalue]
//=> е конкретна функция за [rvalue] =>
//при подаване на такова ще се извика тя,
//а не тази, която може да работи и с двете
xOne.add(std::move(a));

//тъй като няма конкретна функция за [lvalue]
//ще се извика тази, която работи и с двете
xOne.add(a);
```



```
//за move ще използваме ключовата дума [noexcept]
//[noexcept] – казваме, че тази функция няма да throw-ва
// – казваме, че няма заделяне на памет/логика
MyString(MyString&& other) noexcept;
MyString& operator=(MyString&& other) noexcept;
```