

Обектно-ориентирано програмиране (записки)

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

Тема 01. Пространства от имена (Namespace). Енумерации, структури и обединения.

Namespace

01. Какво е namespace?

- инструмент за избягване на конфликти на имена
- scope (област на действие), в който има дефинирани символи

Как се използва? (Пр.1)

```
namespace ns-name
{
    void f() {...тяло...};
    int global = 9;
}

f(); // не работи, защото не можем да достъпим функцията директно,
      // тя съществува в съответния namespace

ns-name::f(); //викане на функция с име "f", която се намира в
              //namespace с името "ns-name"

:: //ОПЕРАТОР ЗА РЕЗОЛЮЦИЯ
```

```

using namespace ns-name; // Служи за включване на всички имена от даден
                           // namespace (ns-name) в текущия scope.
                           // Това означава, че имаме право да
                           // използваме функции, класове, променливи и т.н.,
                           // дефинирани в namespace-а без да е нужно да
                           // назваме изрично къде се намират.

// може да се използва в глобалния scope и в scope-а на някаква
функция, тоест и двете са верни:
using namespace ns-name;
int main()
{
}

-----
int main()
{
    using namespace ns-name;
}

```

Недостатъци (Пр.2)

```

//[!] Проблем при namespace е, че въпреки, че е създаден с цел да
избегне конфликт на имена, все пак може да доведе до такъв конфликт

namespace A
{
    f() {...тило...};
}

namespace B
{
    f() {...тило...};
}

int main()
{
    using namespace A;
    using namespace B; //напомняме, че namespace не е задължително да е
                        //в глобалния scope

    f();           //конфликт на имена, компютърът не знае, коя функция "f" да
                    //изпълни

    //!НО!
    A::f();        //работи, защото сме подсказали коя от двете функции да се
                    //изпълни
}

```

```
//Преговор: namespace - папка от символи
-----
namespace A
{
    void f() {...} //функция с името f
    namespace B
    {
        void f() {...} //функция със същото име,
                        //но в друг namespace
    }
}

A::B::f(); //е валидно извикване на функцията f,
            //която е в namespace B, тъй като сме подсказали
            //коя от двете искаме да извикаме: A -> B -> f()

-----
namespace A
{
    f() {...}
    namespace B
    {
        f() {...}
    }

    using namespace B; //за разлика от първия пример, тук
                      //казваме, че ще използваме символите на
                      //namespace B в тялото на namespace A

    //кодът до тук ще се изпълни, но ако създадем нова функция
    g() { f(); } //не знае коя от двете функции f() да изпълни
                  //няма да се компилира (конфликт на имена)
}
```

Увод в типовете дефинирани от потребителя

01. Енумерация - enum

Дефиниция: тип, рестрикиран до домейн от стойности, които включват специално дефинирани константи. (**енумератори**)

```
//всеки енумератор (специално дефинирана константа) съответства на цяло
число

//всеки енумератор, на който не е дадена стойност, приема стойността на
предишния + 1
//ако не сме задали стойност на първия енумератор, той по подразбиране е
равен на 0

enum color
{
    red, //0
    blue, //1
    orange //2
};

enum nums
{
    a, // 0
    b, // 1
    c = 301,
    d // 302
};
```

```

//Инициализиране на променлива от тип color

int main()
{
    color a1 = color::red;    // 0, използваме оператора за резолюция, за да
                             // достъпим дадена константа в enum-а с името color

    int x = color::red;           //можем да ползваме неявно преобразуване от
                                  //енумератор към число

    if(a1 == color::red) //можем да сравняваме еднотипни енумератори
    {
        //тяло
    }

    a1++; //НЕ Е валидно, тъй като a1 е константа

    std::cout<<a1<<" "<<x; // 0 0
}

```

```

//[!] enum е unscoped, тоест енумераторите (специално дефинираните
//променливи) са глобални променливи
//това означава, че не можем да имаме еднакви имена в два различни enum{}

enum color
{
    red
    orange,
};

enum fruit
{
    orange,
}

[!] Кодът няма да се компилира

```

```
//можем да сравняваме два различни типа, което се нарича
//[!] НЕЛЕГАЛНО СРАВНЕНИЕ

enum color
{
    red,
    orange
};

enum animal
{
    dog,
    cat
};

int main()
{
    color c1 = color::orange;
    animal a1 = animal::cat;

    //![!] важно е да подчертаем, че енумераторите се преобразуват в
    //                                         int и след това се сравняват
    // (тук c1 и a1 се превръщат в числа от тип int равни на 1
    // => ще влезем в тялото на if-а

    if(c1 == a1)
    {
        //тяло
    }
}
```

```
//съществува enum class, което може да се каже, че е просто scoped enum,  
//тоест енумераторите са ограничени в scope-а на самия enum class,  
//което позволява два различни enum-а да имат енумератори с еднакви имена,  
//за разлика от enum  
  
enum class color  
{  
    orange  
};  
  
enum class fruit  
{  
    orange  
};  
  
[!] Кодът ще се изпълни.
```

```
//за разлика от enum, при enum клас няма неявно/имплицитно преобразуване  
  
int x = color::orange; // не може да се cast-не само  
  
int x2 = (int) (color::orange); // работи  
  
if(c1 == a1);           //не можем да сравняваме различни типове директно, тъй  
                      //като нямаме неявно преобразуване от енумератор към цяло число  
  
if((int)c1 == (int)a1){...} // работи
```

```
std::cout<<sizeof(e1);           // това е големината на целочислен тип, в
                                // чиито домейн стойности се побират енумераторите
                                // по подразбиране базовият тип на enum е int,
                                // тоест ще изведе 4,
enum class letters : char
{
    a,
    b,
}

std::cout<<sizeof(letters); // 1
```

```
//Преговор: enum -> enum (unscoped) / enum class (scoped)

enum A
{
    x; // глобално
}

int main()
{
    int x = 7; //в scope-а на main => няма конфликт на имена

    x++; //локалната променлива x, инициализирана в main, ще се увеличи с 1
          //с други думи локалната променлива x ще "shadow"-не глобалната и
          //глобалната променлива x няма да се промени

    A::x; //достъпваме глобалната
    A::x++; //ГРЕШКА, помним, ще енумераторите са константи и не можем да
    ги променяме
}

-----

enum A
{
    x; // глобално
}

enum class B
{
    x; // локално за scope-а на B
}

int main()
{
    int x; // локално за scope-а на main
}

//кодът ще се компилира, тъй като НЯМА конфликт на имена
//можем да кажем, че променливата x в main
// скрива" (shadow) глобалната променлива x в A, която пък
// скрива" (shadow) локалната променлива x в B
```

```
enum Test
{
    a = 0,
    b = 12,
}

//типът заемащ най-много памет в Test е int =>
//sizeof(Test) = sizeof(int) = 4

enum Test2
{
    a = 0,
    b = UINT_MAX + 1,
}

//тъй като b надхвърля int - грешен код ( зависи от компилатора)

enum Test3:char
{
    a = 8,
    b = 80000,
}

sizeof(Test3) = 1; //няма тип, надграждащ char => 1
//тъй като b надхвърля char- грешен код ( зависи от компилатора)
```

02. Структури - struct

01. Какво е struct?

- последователност от полета, които се пазят в определен ред

Как се използва? (Пр.1)

```
struct Test
{
    int a;
    char ch[10];
    bool b;
};

Test t1; //декларация на променлива от тип Test

        //структурата -> инстанция
        //класове -> обект
        //enum -> тип данни

//достъпване на поле и оператор за присвояване
t1.a = 10;

t1.ch = "BCD"; // [ERROR] не можем да ползваме оператор
                // за присвояване при масиви и това не се променя при struct

strcpy(t1.ch, "ABC"); //работи
t1.b = false;
```

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

//начини за инициализация на променлива от тип Test

//статично
Point P {3, 7};
Point P = {3, 7};

//динамично
Point* ptr = new Point{3,7};
delete ptr; // [!]
```

```
//подаване на инстанции (struct) във функции

//стандартно подаване

void f(Point P); //по този начин създаваме копие на инстанцията P,
                    //което означава, че отделяме допълнително памет
                    //за инстанцията и нейните полета
                    //затова, се опитваме да го избегнем, ако е възможно,
                    ////[!] за да спестим памет
```

```
//подаване по референция
```

```
void f(Point& P); //използваме вече заделената памет
                    //и не заделяме нова, по този начин пестим памет
```

```
//[!] ако няма да променяме инстанцията
//задължително ползваме const
```

```
void f(const Point&);
```

```
//подаване чрез пойнтър, тук се заделя допълнително памет само за пойнтъра
```

```
void f(Point* ptr);
```

```
//аналогично, ако не променяме нищо,
//използваме const
```

```
void f(const Point* ptr); //![!] НЕ може да променя данните,
                            //но може да променя адреса
                            //тоест по този начин ползваме
                            //пойнтъра ptr само за четене
```

```
//също:
```

```
//Point* const ptr -> може да променя данните
//                  -> ![!] НЕ може да променя адреса
```

```
//const Point* const ptr -> ![!] НЕ може да променя данните
//                  -> ![!] НЕ може да променя адреса
```

```

//можем да влагаме структури

struct Line
{
    Point beg; //полетата на инстанцията Line са
    Point end; //променливи от типа на друга инстанция (Point)
};

//![!] АБСТРАКЦИЯ - използваме нещо,
//без да се интересуваме как работи

//Пример за лоша абстракция
struct Triangle
{
    int x1; //всяка двойка трябва да е пакетирана
    int y1; //в отделна структура
    int x2;
    int y2;
    int x3;
    int y3;
};

```

```

//можем да създаваме масиви от инстанции

//статично

struct A
{
    int a;
}

A arr[10]; // всеки елемент в масива е променлива от тип A
            // и съответно му е необходима памет с размер sizeof(A);
            // => 10 * sizeof(A) е заделената памет за масива arr

//динамично

A* ptr = new A[n];
delete[] ptr; //при delete[] не пишем размера в скобите,
                //тъй като се създава допълнителна клетка,
                //която го пази и се намира преди първия елемент на масива

//тоест за разлика от статичния масив,
//тук заделената памет е n * sizeof(A) + sizeof(int)
//  
          ^
//  
          |
//  
          |   едната клетка, която
//          |   пази размера на масива
//
```

```
//Преговор: структура - последователност от полета в определен ред

//Декларация
A obj;
A* ptr = new A[n];

//Достъп до елементите
obj.x++;

//Следните са еквивалентни
(*ptr).x++;
ptr->x++;

//Масиви от инстанции
A arr[10];

A* ptr = new A[n];
delete[] ptr;
```

```
//подаване на инстанции във функция

//[NOTE] във всички правим копия

f1(A obj) {...} //тук създаваме копие => можем да извикаме
                    //функцията f1 във всички останали

f2(const A obj) {...} //тук създаваме константно копие
                    // => можем да извикаме f2 във всички останали
                    // тъй като можем да направим преход от
                    // неконстантна инстанция към константна

f3(A& ref) {...} //тук подаваме по референция =>
                    //не можем да извикаме f3 в [f2, f4, f6]
                    //тъй като преходът от константна инстанция
                    //към неконстантна е невъзможен (невалиден)

f4(const A& ref) {...} //по аналогичен начин на f2, само че тук
                    //референцията е константа
                    // => можем да извикаме f4 във всички останали
                    // тъй като можем да направим преход от
                    // неконстантна инстанция към константна

f5(A* ptr) {...} // аналогично на f3, само че подаваме пойнтьр, вместо да
взимаме по референция
f6(const A* ptr) {...} //аналогично на [f2 и f4], само че имаме константен
пойнтьр

//Резултат: [функция -> кои можем за извикаме]
//f1 -> f1, f2, f3, f4, f5, f6
//f2 -> f1, f2, f4, f6
//f3 -> f1, f2, f3, f4, f5, f6
//f4 -> f1, f2, f4, f6
//f5 -> f1, f2, f3, f4, f5, f6
//f6 -> f1, f2, f4, f6
```

```
//Връщане на инстанция от функция

//стандартно връщане на копие (работи)
A f()
{
    A obj;
    obj.data = ...;
    return obj;
}

//връщане на указател към копието (компилира се, но не е коректно)
A* f()
{
    A obj;
    return &obj; //връщаме адреса на локалната инстанция obj,
                 //но тъй като е локална нейната памет се освобождава
                 //в края на scope-a =>
                 //пойнтъра, който връщаме сочи към вече освободена памет
                 //=> не е коректно
}

//връщане по референция (компилира се, но не е коректно)
A& f()
{
    A obj;
    return obj; //връщаме референция по локалната инстанция obj,
                 //но тъй като е локална нейната памет се освобождава
                 //в края на scope-a
                 //=> референцията, която връщаме сочи към невалидна памет
                 //=> не е коректно
}
```

```

//динамично (работи, но ТРЯБВА да освободим паметта)
A* f()
{
    A* ptr = new A {...};
    return ptr; //връща указателя към инстанцията от тип A,
    //за която сме заделили памет динамично
    //=> съществува, докато не освободим паметта сами
    //=> съществува извън scope-а
    //=> работи, но трябва да освободим паметта,
    //за да избегнем възможни проблеми
}

//връщане по референция (работи, НО НЕ по уговорка)
const A& f()
{
    A obj;
    return obj; //когато връщаме по КОНСТАНТНА референция, животът на
    //инстанцията се удължава с един scope
    //=> работи, НО НЕ ГО ПРАВИМ по уговорка
}

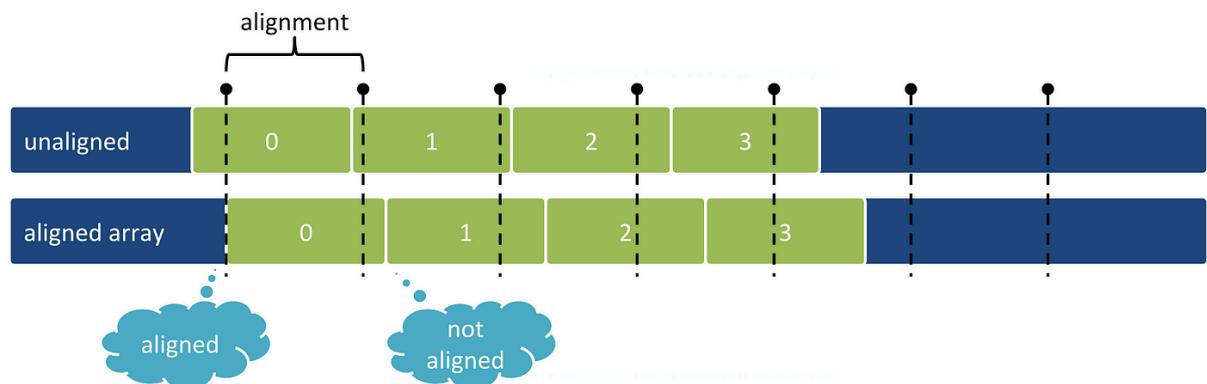
```

Размер на инстанцията

[!] Всеки тип има Alignment requirement

def.] Alignment requirement - разликата на 2 последователни адреса, на които можем да разположим дадена променлива

Искаме всяка променлива да я прочетем с едно четене на една дума



Напр.: int - 4 байта => 0, 4, 8, 12 ,16 , 20, 24... (потенциални адреси)
`alignof(int) = 4`

При **примитивни типове** (int, float, char....) `sizeof(<T>) == alignof(<T>)`

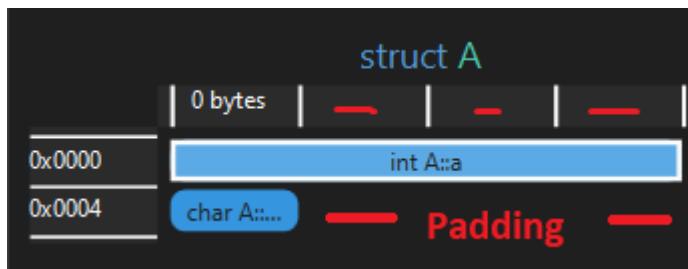
Ще въведем следните **правила** (по уговорка)

1. За да можем да правим масиви, големината на структурата трябва да се дели на най-голямата член-данна
2. Всяка примитивна член-данна трябва да е на адрес кратен на нейната големина

Пр. 1:

```
struct A1
{
    int a; //4 bytes
    char ch; //1 byte
}

//sizeof(A1) == 8
//alignof(A1) == 4
```



В този пример най-голямата член-данна е от тип int. Тя вече се намира на адрес кратен на нейната големина. След това имаме променлива от тип char, която се намира на адрес кратен на нейната големина. Правило номер 2 е изпълнено. Но, за да спазим правило номер 1, трябва да добавим още 3 байта, за да достигнем големина, която се дели на най-голямата член-данна.

($\text{int} + \text{char} + 3 = 4 + 1 + 3 = 8$ и също $8 \% \text{int} = 0$)

```

//масив от инстанции

A1 arr[2];

//sizeof(arr) == 16
//alignof(arr) == 4

//можем да си го представим като горния пример,
//само че ще имаме две инстанции, залепени една до друга
//=> (int + char + padding) + (int + char + padding) = (4
+ 1 + 3) + (4 + 1 + 3) = 16
//=> alignof(arr) все още е 4, тъй като най-големият тип
все още е int

```

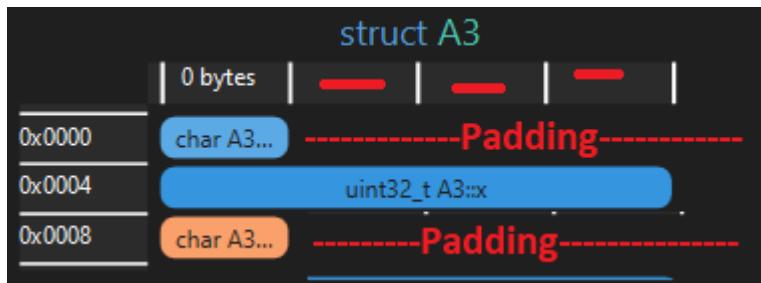
Пр. 2:

```

struct A3
{
    char ch;
    uint32_t x;
    char chTwo;
};

//sizeof: 12
//alignof: 4 (x)

```

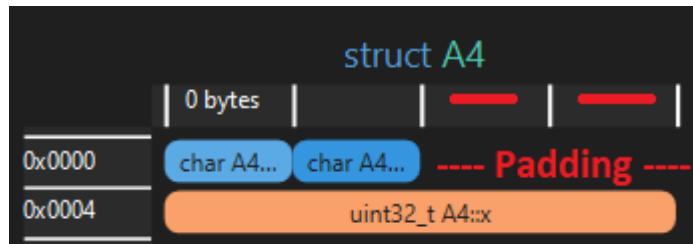


Пп. 3:

```
struct A4
{
    char ch1;
    char ch2;
    uint32_t x;

};

//sizeof: 8
//alignof: 4 (x)
```

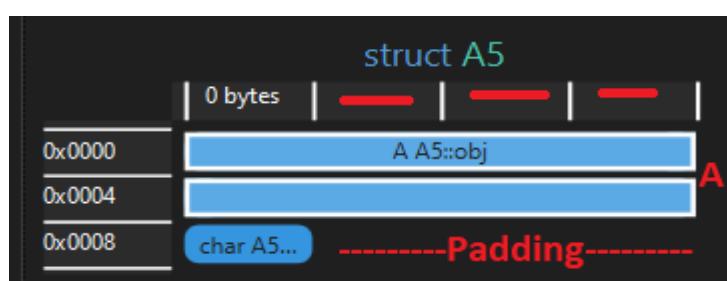


Пп. 4:

```
struct A
{
    uint32_t x;
    char ch;
};

struct A5
{
    A obj;
    char ch;
};

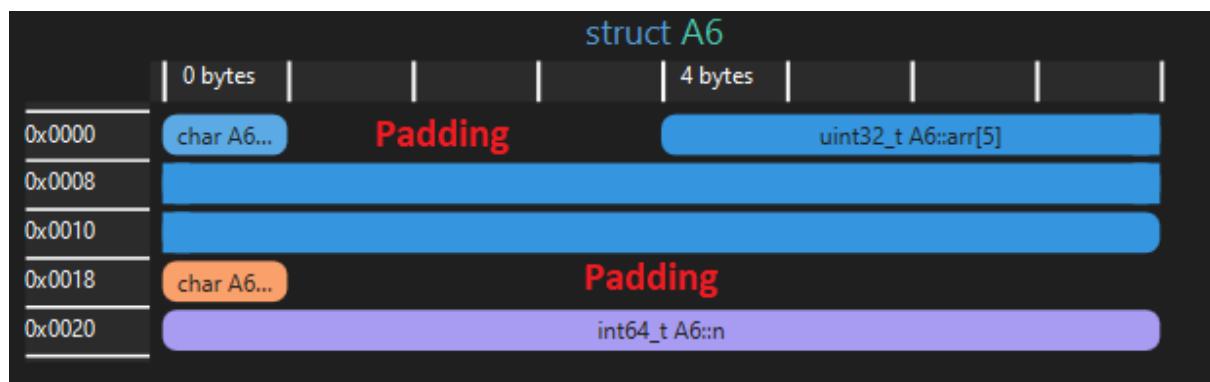
//sizeof: 12
//alignof: 4 (x)
```



Пр. 5:

```
struct A6
{
    char ch;
    uint32_t arr[5]; //третира се
като
                    //5 променливи
от тип uint32_t
    char ch2;
    int64_t n;
}

//sizeof: 40
//align: 8 (n)
```



```
struct A7
{
    char ch;
    int arr[]; //ако е последен, можем да не даваме размер
               //тогава взема колкото място е останало
               //в случая имаме 3 байта padding, което не стига
               //за нито един int => не създаваме клетка на масива
};

//sizeof: 4
//alignof: 4

struct Test
{
    int32_t a;
    char ch;
    char arr[]; //остават 3 байга padding, което стига
               //за 3 char-а => масива има 3 клетки
};

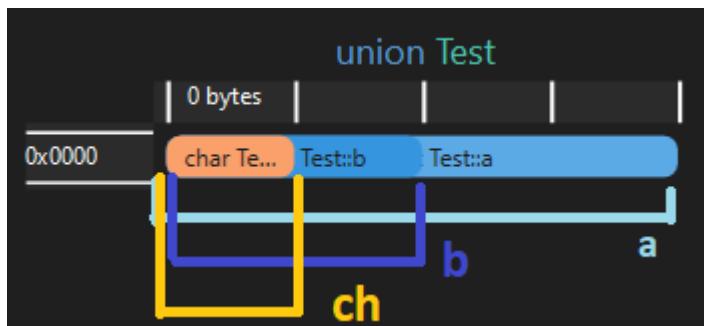
struct T
{
    char ch;
    int a[]; //няма да се компилира,
              //можем да пропуснем размера само АКО Е ПОСЛЕДЕН
    int b[];
};
```

03. Обединения - union

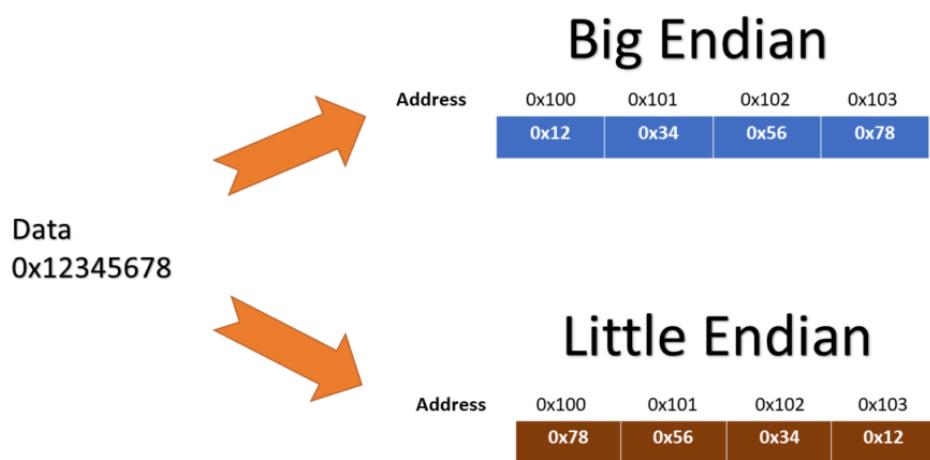
def.| последователност от полета, които заемат (споделят) една и съща памет

```
union Test
{
    int32_t a;
    char ch;
    int16_t b;
};
```

[Note] Една клетка е един байт



def.| Endianness - начин на подреждане на байтовете в една дума



Приемаме, че използваме Little Endian, тоест **отзад напред**

```

union T
{
    int a;
    char ch;
    short b;
}

int main()
{
    T obj;
    obj.a = 75;
}

//Най-големият тип е int =>
//ще имаме 4 споделени байта

//Little endian => [75] [0] [0] [0]
//          ^  ^  ^  ^
//          |  |  |  |  байтове
//          a  a  a  a
//          ch
//          b  b

std::cout << obj.ch; //К (К е буквата с ASCII код 75
std::cout << obj.b; //75

//ако искаме b да е различно от a
//трябва да дадем достатъчно голяма стойност на a,
//за да надхвърлим 16-те бита (short), които определят стойността на b,
//=> числото трябва да е >= 2^16 - 1

obj.a = 50000;
std::cout << obj.b; // != a

```

```
//различна интерпретация (полиморфизъм) в контекста на
//обединенията означава,
//че можем да тълкуваме една и съща област от паметта по
//различни начини,

union Person
{
    Student s;
    Teacher t;
};
```

```
//може да даваме ЕДНА default-на стойност

union Test
{
    int32_t a = 5;
    char ch ;
    int16_t b;
};
```

[!] Важно е да кажем, че **union** са предназначени за използване на точно едно поле. Ако достъпим поне 2 полета, то **UB (undefined behaviour)**.

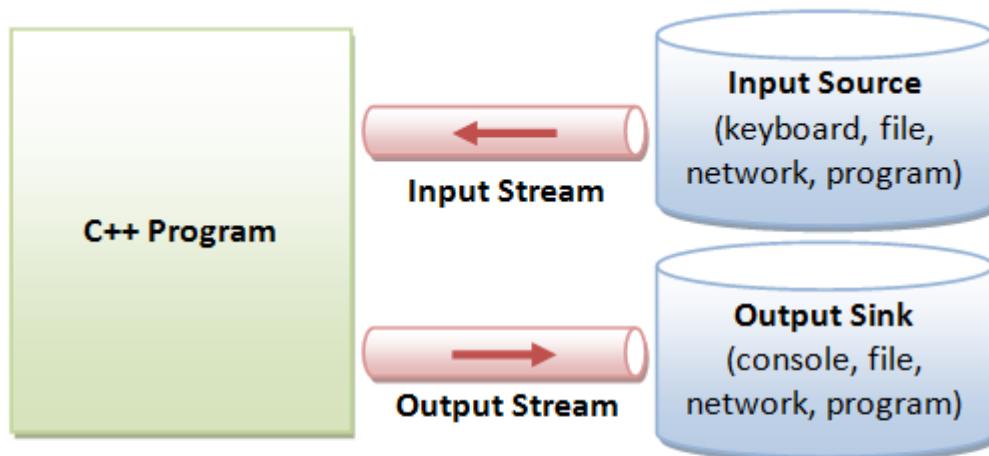
Примери, които трябва да напишем на теория

https://github.com/Angeld55/Object-oriented_programming_FMI/tree/master/Week%2001

Тема 02. Потоци и текстови файлове

def.| Поток - последователност от байтове "насочени" в определена посока

- При операциите за вход, байтовете идват от източник за вход (клавиатура, файл, мрежа или друга програма)
- При операциите за изход, байтовете данни излизат от програмата и се "вливат" във външно "устройство" (конзола, файл, мрежа или друга програма)
- Потоците служат като посредници между програмите и самите IO устройства по начин, който освобождава програмиста от боравене с тях.
- Потокът дефинира интерфейс с операции върху него, които не зависят от избора на IO устройство



Internal Data Formats:

- Text: `char`, `wchar_t`
- `int`, `float`, `double`, etc.

External Data Formats:

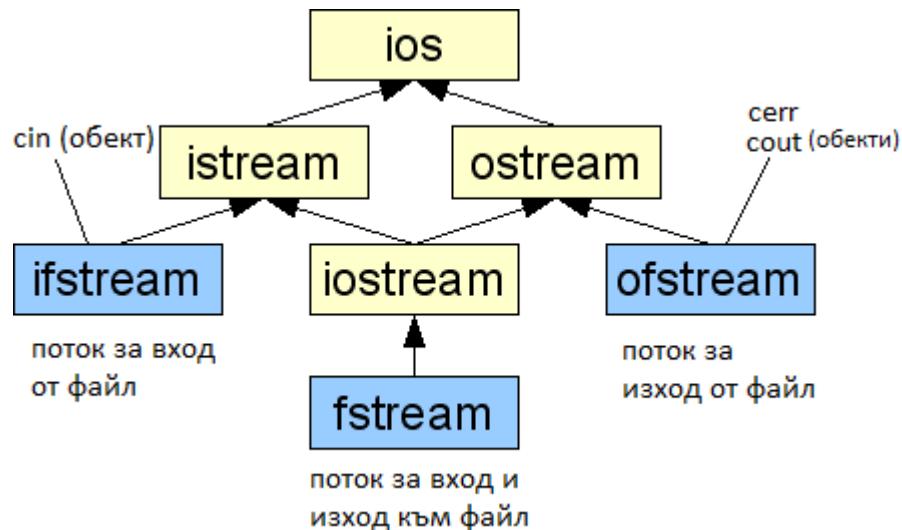
- Text in various encodings
(US-ASCII, ISO-8859-1, UCS-2, UTF-8, UTF-16, UTF-16BE, UTF16-LE, etc.)
- Binary (raw bytes)

За да извърши вход или изход, една C++ програма:

- Създава поток;
- Свързва потока с IO устройството (напр. конзола, клавиатура, файл, мрежа или друга програма);
- Извършва операции за вход/изход върху потока;
- Прекъсва връзка с потока;
- Освобождава потока;

Видове потоци:

- Потоци за вход
- Потоци за изход



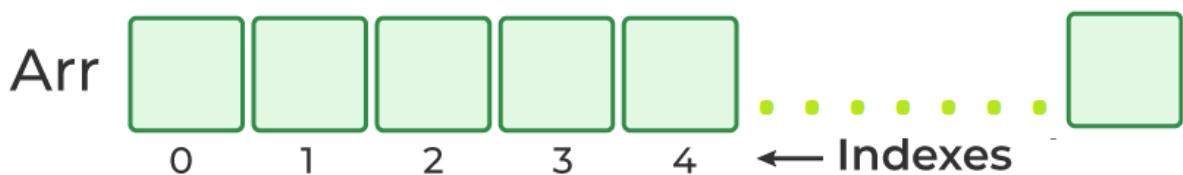
```
1 1 #include <iostream>
2 2 #include <fstream>
3
4 3 int main()
5 4 {
6 5     //отваряне на поток за вход към файл
7
8 6     std::ifstream fileIn("test.txt"); //важно е да отбележим, че вътрешно (скришно)
9 7         //      ^      ^      ^
10 //        |      |      |
11 // ifstream   име    файлът, към който
12 // (input)           отваряме потока
13
14
15 //аналогично е отварянето на поток за изход към файл
16
17 std::ofstream fileOut("test.txt"); //важно е да отбележим, че вътрешно (скришно)
18 //      ^      ^      ^
19 //      |      |      |
20 // ofstream   име    файлът, към който
21 // (output)           отваряме потока
22
23 fileIn.close();
24 fileOut.close(); //<-----|
25 //          //-----|
26 //          //-----|
27 //          //-----|
28 // в края на съответния scope, в който е отворен потока скришно се извиква .close(),
29 // но е добра практика да го пишем сами
```

Интерфейс

- "интерфейс" обикновено се отнася до набор от функции или методи, предоставени от езика за програмиране или библиотеката, които позволяват на програмиста да извършва операции върху файлове (в случая). Тези операции могат да включват отваряне и затваряне на файлове, четене от файлове, писане във файлове, манипулиране на позицията на четене/писане във файл и други подобни действия.

01. Потоци за изход

- можем да си го представим все едно отваряме един **БЕЗКРАЕН МАСИВ**



При потоците за изход имаме така наречения **put** указател, който се мести до първата свободна позиция

-> **форматиран изход** - отнася се до процеса на конвертиране на данните в четим за човека формат преди тяхното извеждане. (<<)

```
fileOut << <обект> << 37;  
//           ^           ^  
//           |           |  
// символ по символ      всеки обект се интерпретира  
// "поставя обекта"      по различен начин, 37 не е един символ
```

-> **неформатиран изход** - отнася до директното записване на данни в изходен поток без никаква промяна на формата им. Това означава, че данните се предават точно в същия вид, в който са представени в паметта, без да се конвертират в четим за човека формат (.put() .write())

```
char ch = 'a';
fileOut.put(ch); //използва се за записването САМО на един
символ

char str[4] = "abc";
fileOut.write((const char*) str, sizeof(str)); //ползва се при
//                                                 двоични файлове
//          ^           ^
//          |           |
//приема като аргумент      броят на байтовете, които
//                           искаме да запишем
//константен char поинтър
//(терминиращата нула не ни трябва)
```

-> **синхронизация** - тоест правим промени в буфера и след някакъв интервал от време ще се изсипе във файла

.flush() - принудително изпращане на съдържанието на буфера към крайния изход, гарантирайки, че всички междинно съхранени данни са били обработени

```
fileOut << 3;
fileOut << 3;
fileOut << 3;
fileOut << 3;
fileOut.flush(); //изсипваме и запазваме новите промени във
                  файла
```

-> позициониране - способността да управяваме текущата позиция във файл

```
fileOut.tellp(); //връща на коя позиция е пойнтърът в момента

//има два начина за използване на seekp

fileOut.seekp(3); //мести пойнтъра 3 позиции напред ОТ НАЧАЛОТО =>

std::cout << fileOut.tellp(); // 3
fileOut.seekp(3);
std::cout << fileOut.tellp(); // 3

-----
fileOut.seekp(3, std::ios::cur); //мести пойнтъра 3 позиции напред
//от текущата позиция

fileOut.seekp(0, std::ios::beg); //мести пойнтъра 0 позиции напред
//от началото

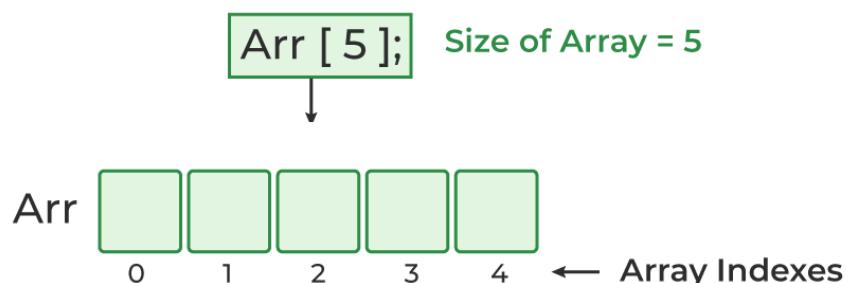
fileOut.seekp(-3, std::ios::end); //мести пойнтъра 3 позиции назад
//от края

// [NOTE] end ни праща след последния символ във файла
```

02. Потоци за вход

Приprotoците за вход имаме така наречения **get** указател, който се намира на текущата позиция за четене

-можем да си го представим все едно отваряме един **КРАЕН МАСИВ**



-> **форматиран вход**- се отнася до процеса на четене на данни от входен поток, където данните се интерпретират и конвертират в определени типове на данни ([>>](#))

```
fileIn >> <обект>;  
// ^  
// |  
// объект, който  
// знаем как да четем
```

```
int n;  
fileIn >> n; //вижда, че е int и чете като int
```

-> **неформатиран вход** - чете данните точно както са представени, без да променя съдържанието или да пропуска символи (.get() .getline())

```

//[!] Важно е да разграничаваме get() и getline()
//getline() прескача разделителя, който сме задали ('\n' по default)
//get() НЕ прескача разделителя, който сме задали (EOF по default)

is.getline(<buff>, <size>, 'X');
is.get(<buff>, <size>, 'X');
//-----
//Нека нашият ред е ABCX123
//getline() ще напълни buff с ABC\0, защото среща нашия разделител 'X',
//прескача го и на следващата входна операция X няма да бъде отразен, т.е

char strOne[10] = {};
std::cin.getline(strOne, 4, 'X');

char strTwo[10] = {};
std::cin.getline(strTwo, 10); //помним, че по default разделителя е нов ред

std::cout << "[STR ONE]: " << strOne << std::endl; //ABC
std::cout << "[STR TWO]: " << strTwo << std::endl; //123

//get() ще напълни buff с ABC\0, защото среща нашия разделител 'X',
//НЕ го и на следващата входна операция X ще бъде отразен, т.е

char strOne[10] = {};
std::cin.get(strOne, 4, 'X');

char strTwo[10] = {};
std::cin.get(strTwo, 10); //помним, че по default разделителя е EOF
                        //и cin е поток за вход насочен някъде (абстракция)

std::cout << "[STR ONE]: " << strOne << std::endl; //ABC
std::cout << "[STR TWO]: " << strTwo << std::endl; //X123

```

Като извод можем да кажем, че:

getline = **get** + **ignore** -> тоест местим пойнтьра до разделителя с помощта на **get**, и игнорираме разделителя с **ignore**, за да го прескочим
(както прави **getline**)

get = **getline** + **seekg**(-1, ios::curr) -> тоест местим пойнтьра до СЛЕД разделителя с помощта на **getline** и се връщаме назад с една позиция с помощта на **seekg**, за да се върнем на разделителя (където отиваме с **get**)

-> позициониране - способността да управяваме текущата позиция във файл

```
fileIn.tellg(); //връща на коя позиция е пойнтьрът в момента

//има два начина за използване на seekp

fileIn.seekg(3); //мести пойнтьра 3 позиции напред ОТ НАЧАЛОТО =>

std::cout << fileIn.tellg(); // 3
fileIn.seekg(3);
std::cout << fileIn.tellg(); // 3

-----
fileIn.seekg(3, std::ios::cur); //мести пойнтьра 3 позиции напред
                                //от текущата позиция

fileIn.seekg(0, std::ios::beg); //мести пойнтьра 0 позиции напред
                                //от началото

fileIn.seekg(-3, std::ios::end); //мести пойнтьра 3 позиции назад
                                //от края

// [NOTE] end ни праща след последния символ във файла

-----
//може да преместим пойнтьра, без да връщаме резултат
fileIn.ignore(<брой символи, които искаме да пропуснем>, <разделител>)
//          ^          ^
//          |          |
//          по default 1      по default eof
//          (unsigned -> не можем да
//           връщаме назад;
//
//          Напр.: не можем да игнорираме -1 символа
```

```
//форматирания вход прескача ' ', '\n', '/t'  
//или с други думи ги смята за излишни  
  
//но има една особеност  
int a;  
std::cin >> a; //когато std::cin прочете всичко, което му е казано приключва, тоест  
//ако отидем на нов ред след std::cin, новият ред няма да се прескочи  
  
char buff[1024];  
std::cin.getline(<размер>, buff); //разделителят е '\n', но след въвеждането на [a],  
//имаме нов ред, който не сме отразили => buff ще бъде празен,  
//тъй като веднага среща разделителя  
  
-----  
//в такива ситуации използваме ignore()  
  
int a;  
std::cin >> a;  
std::cin.ignore(); //прескачаме новия ред  
  
char buff[1024];  
std::cin.getline(<размер>, buff); //записваме в buff
```

```
//това няма да бъде проблем, ако
//1. разделителят ни не е '\n'
//2. не слагаме разделител между [a] и [buff]

//1.
int a;
std::cin >> a;

char buff[1024];
std::cin.getline(<размер>, buff, 'X');

//ако входът ни е 123\nabcX
std::cout<< a << " "; //123 -> чете числото

std::cout<< buff; //\nabc -> тъй като разделителят не е
//'\n', '\n' просто се записва в buff

//2.
int a;
std::cin >> a;

char buff[1024];
std::cin.getline(<размер>, buff);

//ако входът ни е 123abc, тоест без разделител между тях
std::cout<< a << " "; //123
std::cout<< buff; //abc

//очевиден е проблемът, че ако искаме [a] да е 12, а [buff] - 3abc,
//това е невъзможно без да разделим числото и стринга, тъй като по този начин
//записваме в [a], докато срещнем символ, който не е цифра
```

```
//за потоци имаме и командата .peek(),  
//която връща текущия символ и не мести указателя  
  
//Например, ако имаме файл със съдържание (32abc)  
  
std::fileIn.get(); //прескачаме "3"  
std::cout << std::fileIn.peak(); //намираме се на "2" и го отпечатваме  
  
//разликата между .peek() и .tellg() е  
//1. .peek() ни казва на кой символ сме и не мести указателя нататък  
//2. .tellg() ни казва на кой индекс сме във файла и не мести указателя нататък  
  
std::ofstream fileOut("test.txt");  
fileOut << "123";  
fileOut.close();  
  
std::ifstream fileIn("test.txt");  
  
std::cout << fileIn.tellg() << " "; //0 -> намираме се на нулевия индекс  
  
fileIn.get(); // прескачаме "1"  
  
std::cout << fileIn.peek() - '0' << " "; //2 -> намираме се на "2"  
  
std::cout << fileIn.tellg() << " "; //1 -> намираме се на първи индекс  
std::cout << fileIn.tellg() << " "; // 1 -> tellg() не мести указателя и все още  
сме на първи индекс  
  
fileIn.get();  
std::cout << fileIn.tellg() << " "; //2 -> намираме се на втори индекс, защото  
//get() мести указателя в случая с 1 позиция
```

Режими на работа

Режимите на работа представляват 8-битово число. Всеки бит има стойност 1, ако е вдигнат и 0, ако не е. Всеки бит отговаря за различно нещо. Например ако последният бит е вдигнат, това означава, че файлът е отворен за вход



```
//ofstream ofs(<име>, <число>)

std::ofstream fileOut("text.txt", 10);

//числото 10 в двоичен вид е 0000 1010 => ще бъде отворен за
конкатенация и изход
```



```

std::ofstream fileOutContain("text.txt");
fileOutContain << 1;
fileOutContain.close();

//ако имаме файл със съдържание (1) и отворил файлът за
конкатенация, то
std::ofstream fileOutConc("text.txt", 10); //конкатенация и изход

fileOutConc << 2;
fileOutConc.close();

//във файлът ще се запише съдържанието (12), тъй като чрез
конкатенацията
//сме отишли накрая и пишем след това, а не пишем отгоре на
съдържанието на файла,
//тоест не заменяме 1 с 2, както сме свикнали, а ги слепваме и
става 12

```

```

//с цел да не помним числа наизуст, за да вдигнем даден бит, за удобство за
//удобство са създадени специални флагове, които го правят

//ако искаме да вдигнем няколко бита, можем да ги изредим с побитовата
операция или ("|")
std::ofstream fileOut("text.txt", std::ios::app | std::ios::out);

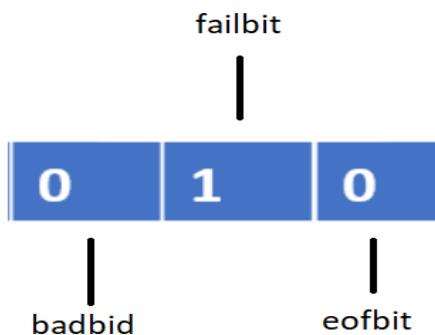
fileOut.close();

```

ios	Ефект:	
ios::in	Отваря файл за извлечане.	1
ios::out	Отваряне на файл за вмъкване. Допуска се вмъкване на произволни места във файла. Ако файлът съществува, съдържанието се изтрива.	2
ios::ate	Отваря за вмъкване и установява указателя rut в края на файла. Допуска вмъкване на произволни места.	4
ios::app	Отваря за вмъкване и установява указателя rut в края на файла	8
ios::trunc	Ако файлът съществува, съдържанието се изтрива.	16
ios::binary	Превключва режима от текстов в двоичен	32
ios::_Nocreate	Отваря за вмъкване, само ако файлът с указаното име съществува.	64
ios::_Noreplace	Отваря за вмъкване само ако файлът с указаното име не съществува.	128

Състояние на потока

Подобно на битовете за режим на работа, съществуват битове за състояние на потока, които също са число.



съществува и **goodbit**, който не е част от това число и е равен на 0, ако има операция, която не се е извършила успешно, това се проследява чрез другите флагове.

Флагове на състоянията на потока

Флаг:	Значение:
bad()	Има загуба на информация. Някоя операция за четене и писане не е изпълнена.
fail()	Последната входно/изходна операция е невалидна.
good()	Всички операции са изпълнени успешно.
clear()	Изчиства състоянието на потока (Вече good() ще върне истина).
eof()	Достигнат е края на файла.

Разликата между **fail()** и **bad()** е, че когато е вдигнат **badbit-a** има проблем с потока за вход или този за изход. Това означава, че сме загубили информация и води до невъзможно ползване на следващи операции. **Failbit-a** се вдига когато имаме проблем с логиката, например искаме да прочетем някакво число, но ни подават буква. Тоест не губим информация и са възможни следващи операции.

iostate value (member constant)	indicates	functions to check state flags					
		good()	eof()	fail()	bad()	rdstate()	
goodbit	No errors (zero value iostate)	true	false	false	false	goodbit	
eofbit	End-of-File reached on input operation	false	true	false	false	eofbit	
failbit	Logical error on i/o operation	false	false	true	false	failbit	
badbit	Read/writing error on i/o operation	false	false	true	true	badbit	

Когато е вдигнат **badbit-a**, то задължително е вдигнат и **failbit-a**. Това означава, че не можем директно да проверим дали **failbit-a** е вдигнат поради логически проблем. За да проверим това ни е нужна проверката

```
if(file.fail() && !file.bad())
{
    std::cout<<"Fail";
}
```

```
//Имаме командата .clear(), която изчиства състоянието на потока,
//fail, bad, eof стават 0

file.clear();
```

Stringstream

Stringstream представлява нещо като “**фалшив поток**” към даден **стринг**

```
std::stringstream ss("33"); //отваряме поток към стринг,
                           // ЧИЕТО СЪДЪРЖАНИЕ е "33"

//главната му полза е за по-бързо и ефективно прехвърляне
//на int в char arr и обратно
int a = 0;
ss >> a; //присвояваме стойност на [a], имплицитно превръща "33" в int
std::cout << a << std::endl; // 33
```

```
std::stringstream ssTwo("33Test"); //отваряме поток към стринг, ЧИЕТО СЪДЪРЖАНИЕ е "33Test"

int b = 0;
ssTwo >> b; //присвояваме стойност на [b], имплицитно превръща "33" в int,
//33, защото докато чете стига до символ, който не е цифра и спира

std::cout << b << " "; //33

char strTwo[10];
ssTwo >> strTwo; //в ssTwo ни остана "Test", и го записваме в strTwo
std::cout << strTwo << std::endl; //Test
```

```
//Забележете, че използването на stringstream е в доста специфични случаи, напр.:

std::stringstream ssThree("Test33"); //отваряме поток към стринг, ЧИЕТО СЪДЪРЖАНИЕ е "Test33"

int c = 0;
ssThree >> c; //присвояваме стойност на [c], но още първият символ не е цифра
//=> няма да му запазим нищо => ще се вдигне failbit-а, заради
//грешка, с която не се губи информация => трябва изчистим състоянието на потока

ssThree.clear(); //изчистваме потока

std::cout << c << " "; //0 (default-натаста стойност)

char strThree[10];
ssThree >> strThree; //в ssThree имаме "Test33", и го записваме в strThree
std::cout << strThree; //Test33
```

```
//можем да създаваме фалшиви потоци без нищо в тях
//и съответно в никакъв етап от нашата програма да запишем в тях
std::stringstream ss;

//пишем 5 в ss
ss << 5;

//тъй като ss е "поток", макар и фалшив,
//можем да ползваме интерфейса на потоците
ss.seekp(1, std::ios::beg);
ss.seekg(1, std::ios::beg);

ss.tellg();
ss.get();

char str[10];
ss.getline(str, 10);
```

Примери, които трябва да знаем:

https://github.com/Angeld55/Object-oriented_programming_FMI/tree/master/Week%2002

Тема 03. Двоични файлове и fstream

01. Двоичен файл

def| готов за зареждане в паметта (файлове с пряк достъп)

Двоичният файл записва информацията точно като е представена в паметта.

За да направим разликата, нека имаме числото 5.

При **стандартния файл**, той ще запише **"5"**. Докато при **двоичния** ще се запазят байтовете на числото 5, в случая **[0] [0] [0] [5]**, ако числото е от тип **int**.

Тестов файл - лесен за нас, труден за програмата

Двоичен файл - труден за нас, лесен за програмата

```
//познатата ни операция за изход при стандартните файлове <<,
//тук е .write(), а операцията за вход >>, тук е .read()

std::fstream inFile("text.txt", std::ios::binary);

int a = 5;
inFile.write((const char*)&a, sizeof(a));

//           ^
//           |           ^
//превръщаме обекта,           броят байтове, които
//който искаме да запишем     искаме да запишем
//CONST char масив, тъй като
//char е 1 байт

char buff[10];
inFile.read(buff, 10); // <- броят байтове, които искаме да запишем
//           ^
//           |
//превръщаме обекта,
//в който искаме да запишем
//в char масив, тъй като
//char е 1 байт
```

```

//-> структури с "външен ресурс"
struct Person
{
    char* name = nullptr;
    int age;
}

//тук, name е пойнтър към char, което означава, че може да сочи към
//низ от символи (string) съхраняван някъде в паметта.
//Това е типичен пример за "външен ресурс" - памет,
//която не е част от самата структура Person,
//но е от съществено значение за нейната коректна функционалност.

std::cout << sizeof(Person); //ще включи паметта отделена за ПОЙНТЬРА,
                           //а не самия масив

write(...&p, sizeof(P)); //така ще запишем указателя (адрес)
                        //и годините, което губи ползите от двоичния файл

//=> [nameSize] [name] [age], ще е нашето съдържание на файла,
   //т.е. запазваме дължината на името, самото име и годините

```

!!! Указателят е един **put и **get** са един и същи указатели (**не винаги е така**)**

```

std::fstream file(fileName);

std::cout << tellp() == tellg() << std::endl; // 1 (true)

//важно е да кажем, че ако преместим единия, то се мести и другия, т.е
file.seekg(1, std::ios::cur);

std::cout << tellp() == tellg() << std::endl; // 1 (true)

//При входна операция след изходна трябва да синхронизираме буфера

file << "a";
file.flush(); //синхронизация

//или seekg(tellg());

int num = 0;
file >> num;

```

Примери, които трябва да знаем:

https://github.com/Angeld55/Object-oriented_programming_FMI/tree/master/Week%2003

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 КОНСТАНТИ
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
    // (обикновено при массиви)
};
```

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

    //Point() {}

    Point() = default; //при инициализации член-данные ще
    //используем ключевую слово default
    //(поэтому образом делаются оптимизации)
};
```

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 А, В, С
    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 А, В и С
    //           |
    //След двоеточието изписваме кои конструктори на А, В, С искаме да се извикат,
    //ако не упоменем кой конструктор на А искаме да извикаме например, ще се извика
    //default-ният, а ако няма такъв, няма да се компилира, тъй като А 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 А, В, С, конструкторът на X ще се извика пръв, но няма да влезе в тялото му,
//докато А, В, С не бъдат създадени => редът на успешно изпълнение е А -> В -> С -> X

X() {} //DEFAULT + вика default-ните на А, В, С, 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) {}
};

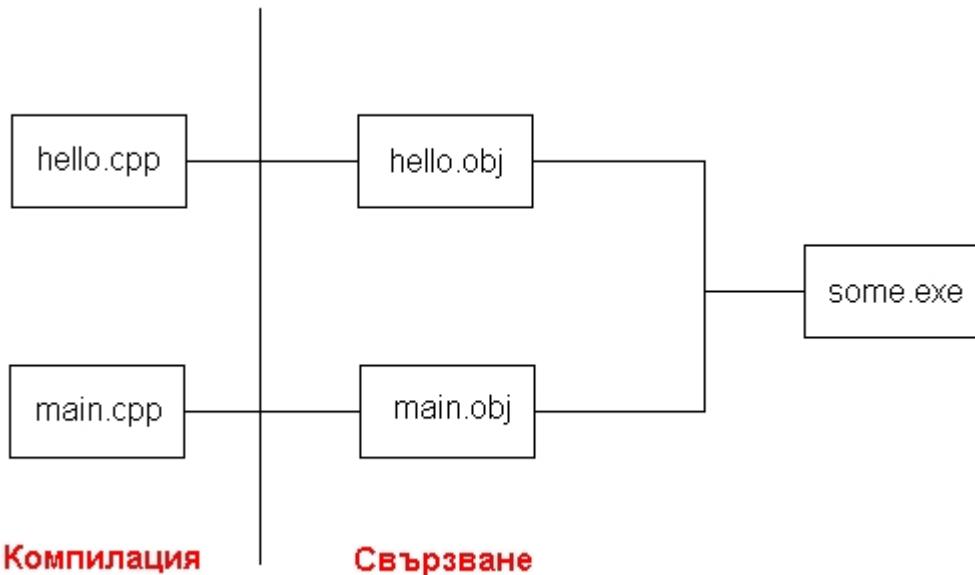
struct Y {
    A obj;
    Y() = default;
    Y(int x) : Y(), obj(x) {} //не знае дали да присвои default-натата стойност на obj,
}; //или да му присвои [x], т.е. не знае по какъв начин да инициализира obj,  

    //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-данината в текущия
```

Тема 05. Разделна компилация



02.cpp ✘ X 01.cpp

+ Project7

```
1 #include <iostream>
2
3 void f()
4 {
5     std::cout << "02" << std::endl;
6 }
```

02.cpp 01.cpp* ✘ X

+ Project7 (Global Scope)

```
1 #include <iostream>
2
3
4 void f(); //обещаваме, че в процеса на свързване (Linking)
5 //функцията f() ще бъде намерена (forward declaration)
6
7 //ако функцията не бъде намерена, ще се върне грешка при Linking процеса
8
9 int main()
10 {
11     f(); //тъй като функцията f() е дефинирана във файла 02.cpp
12     //при Linking-а, 01.cpp ще намери функцията f() във 02.cpp
13     //и ще я изпълни
14
15
16     return 0;
17 }
```

```
02.cpp* 01.cpp*
```

```
Project7
```

```
1 #include <iostream>
2
3 void f() //дефиниция на функцията f() в 02.cpp
4 {
5     std::cout << "02" << std::endl;
6 }
```

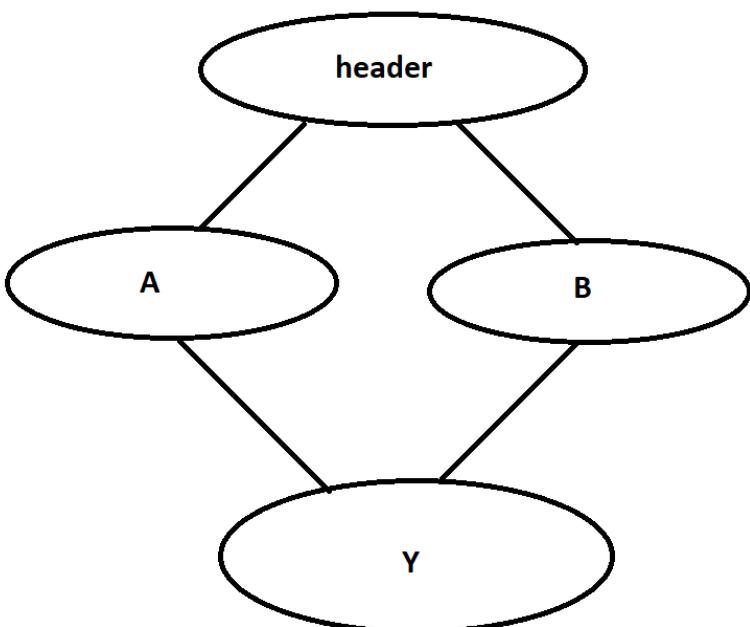
```
02.cpp* 01.cpp* (Global Scope)
```

```
Project7
```

```
1 #include <iostream>
2
3 void f() //дефиниция на функцията f() в 01.cpp
4 {
5     std::cout << "01" << std::endl;
6 }
7
8 int main()
9 {
10    f(); //тъй като функцията f() е дефинирана в 01.cpp и 02.cpp
11    //при Linking-а, ще се намерят две функции f() с еднакви имена
12    //=> конфликт на имена => няма да се компилира
13
14
15    return 0;
16 }
```

Ще отбележим, че:

- Forward декларациите (обещанията, че функции с дадени имена ще се намерят при Linking) ще ги слагаме в .h (header) файлове и когато искаме да ги използваме ще ги include-ваме в съответния .cpp файл.
- Ако променим един .cpp файл, то тогава няма да се променят останалите, а ще се използват техните .obj файлове
- При обекти/инстанции ще пишем декларациите и обещанията за функции в .h (header) файлове, а имплементацията им в .cpp



В следния пример, нека А и В наследяват даден header файл, а Y наследства А и В.

То тогава, Y include-ва 2 пъти header-а
(1 път от А и 1 път от В).

За да избегнем многочестотно включване,
използваме **#pragma once**,
което унищожава всяко
копие на даден header
и го include-ва точно веднъж.

1. Препроцесор - обработка на стрингове

Препроцесорът е първата стъпка в процеса на компилация. Той обработва всички файлове на препроцесора, които започват с #, като #include и #pragma. Примерно, когато използвате #include <iostream>, препроцесорът **замества този ред със съдържанието** на файла iostream, така че компилаторът да може да разбере и използва кода в него.

макроси - мини функции, които ни казват замести код с друг код

#define A 73 - всяко A се заменя с числото 73

```

Project: Project1
1  #include <iostream>
2
3  //define се използа при инициализация на константи
4  //или малки функции
5
6  #define MIN(a, b) ((a) < (b) ? (a) : (b))
7  #define PI 3.14
8  int main()
9  {
10     int x = MIN(5, 3); //3
11     std::cout << PI << std::endl; //3.14
12
13
14 }
15
16

```

Макросите **НЕ** заемат място в стека, по време на компилация всяко достъпване на макрос води до заместване със съответната стойност или функция.

int a = 73; **ЗАЕМА** място в стековата рамка

2. Синтактичен анализ

Тази стъпка анализира синтаксиса на изходния код, за да провери за синтактични грешки. Например, ако забравим да сложим точка и запетая (;) в края на израз, синтактичният анализ ще открие тази грешка.

a++; **X**

3. Семантичен анализ

След като кодът е синтактично правилен, семантичният анализ проверява дали той има смисъл. Например, ако опитаме да съберем число и низ, семантичният анализ ще открие, че това е невалидно.

A obj;
obj = 73; **X**

4. Междинни оптимизации

След семантичния анализ, компилаторът може да извърши редица оптимизации на междинния код, за да направи програмата по-бърза и/или да намали нейния размер. Това може да включва премахване на излишен код или оптимизиране на цикли.

Напр.:

```
if(10 > 3) //замества се с тялото на if-а
{
    <тяло>
}
```

5. Assembly code

Програмен език от по-ниско ниво и е специфичен за типа на процесора, към който е насочена програмата. Той представлява мост между високо ниво кода и машинния код.

6. Машинен код

Асемблерният код се компилира (компиляционен процес) до машинен код, който е директно изпълним от процесора. Машинният код е набор от 0 и 1. (компилирания код)

7. Линкване

След като всички изходни файлове са компилирани до машинен код, чрез Linking се комбинира този код с кода от външни файлове и генерира изпълним файл. Това е процесът на "свързване" на различни части от програмата и файловете, от които зависи, в единен изпълним файл.

8. Оптимизация - CPU-depend оптимизации

Това са оптимизации, специфични за конкретния тип процесор, към който е насочена програмата, с цел да се увеличи ефективността на програмата.

Композиция и агрегация

- взаимоотношения между обекти

Композиция

```
class A {};
class B {};
class C {};

class X
{
    A obj1;
    B obj2;
    C obj3;
};

//Композиция наричаме, когато жизненият цикъл на
// A, B, C се контролира от X, т.е.
// 1. Конструктора на X създава A, B, C
// 2. Деструктора на X унищожава A, B, C
```

```
class A {};
class B {};
class C {};

class X
{
    A* obj1;
    B& obj2;
    C obj3;
};

//Агрегация наричаме, когато
// 1. указателя или референцията сочи към обекти,
// които са самостоятелни
// 2. A, B могат да живеят извън рамките на X

//В този пример има АГРЕГАЦИЯ между
//X и A
//X и B
//
//В този пример има КОМПОЗИЦИЯ между
//X и C
```

```

class A {};
class B {};

class Y
{
    A* a;
    B& b;

    Y(A* ptr1, B* ptr2) : a(ptr1), b(*ptr2) {}

    ~Y() {} //не извиква деструкторите на A и B,
              //тъй като те са самостоятелни
};

```

```

class A {};
class B {};
class C {};

class Z
{
    A obj;
    B* ptr;
    C obj3;

    Z() //A(), C()
    {
        ptr = new B(); //B()

        //създаваме B (грижим се за живота на [Z])

        //макар и ptr да е поинтър към външен ресурс,
        //това също може да се разглежда като форма на композиция,
        //тъй като [Z] управлява живота на обекта [B], към който [ptr] сочи.
        //
        //отговорността за създаването и унищожаването на обекта [B] е на класа [Z].
    } //Z()

    ~Z()
    {
        delete ptr; //унищожаваме B (освобождаваме паметта; грижим се за живота на [Z])
    } //~Z(), ~B(), ~C(), ~A()
};

```

```
//нека имаме следния код

class Config
{
    int x = 4;
};

class App1
{
    Config* config = nullptr; //агрегация
public:
    App1(Config& config)
    {
        this->config = &config;
    }

    void run() { /*...*/ }
};

class App2
{
    Config* config = nullptr; //агрегация
public:
    App2(Config& config)
    {
        this->config = &config;
    }

    void run() { /*...*/ }
};

int main()
{
    //когато животът на [c] приключва заедно или след [a1], [a2]
    //е добър пример за агрегация, викаме конструктора, подавайки
    //външният ресурс към когото искаме да насочим пойнтър член-данната

    Config c;
    App1 a1(c);
    a1.run();

    App2 a2(c);
    a2.run();

} //в случая и двата обекта ще умрат след scope-а
```

```
int main()
{
    //когато животът на [c] приключва ПРЕДИ [a1], [a2]
    //е ЛОШ пример за агрегация

    //нека си представим, че сме разширили кода с default-ен конструктор,
    //(напомняме, че това е нужно, тъй като вече имаме наш конструктор и компилаторът
    //няма да създаде default-ен сам)

    //и имаме мутатор (setter) за член-данната [config]

    App1 a1; //викаме конструктора (default)
    {
        Config c;
        a1.set_config(c); //насочваме член-данната [config] към [c]

    } // [c] умира в края на scope-а, тоест ПРЕДИ [a1]

    //тогава нашата член-данна [config] сочи към вече освободена памет,
    //тъй като [c] вече е умряло, което очевидно е проблем

    a1.run();
}
```

Копиране на обекти

01. Копиращ конструктор

- приема обект от същия тип
- текущия става негово копие

```
1 //синтаксис
2
3 class X
4 {
5     //член-данни
6     public:
7     X(const X&); //копиращ конструктор
8     {             //приема обект от същия тип)
9
10    //тело на копиращия конструктор
11 }
12 };
```

!Ако не го създадем (разпишем), компилаторът ще създаде такъв сам!

```
class A
{
    int a;
};

class B
{
    int b;
};

class X
{
    int a;
    char ch;
    A obj; //обект
    B obj2; //обект
};
```

```
int main()
{
    X obj;
    X copy(obj); //в този пример не сме разписали
    //      ^      наш копиращ конструктор, т.е. както
    //      |      вече казахме, компилаторът ще ни създаде такъв
    //      |
    //извикваме копирация конструктор на X
    //той на свой ред извика копиращите конструктори на A и B,
    //а на променливите от примитивен тип (в случая int и char)
    //ще извика мястото им в паметта

    //копирацият конструктор е вид КОНСТРУКТОР, които вече разгледахме,
    //тоест реда на копирането зависи според реда на тяхното деклариране
    //в съответните класове

    //напомняме, че това е така, тъй като всяка следваща променлива/инстанция/обект
    //може да зависи от предходната променлива/инстанция/обект

    //в случая реда на деклариране е [a] [ch] [A obj] [B obj2]
    // [a] и [ch] се копират директно (тъй като са примитивни типове),
    // след което се извикват копирация конструктор на обекта [A] (заради [obj])
    // и накрая копирация конструктор на обекта [B] (заради [obj2])

    return 0;
}
```

```
class A
{
    int a;
};

class B
{
    int b;
};

class Y
{
    A obj1; //обект
    B obj2; //обект

public:
    //предният пример може да ни позволя да
    //направим следния извод:

    //декларацията ни на члед-данные е A -> B
    //=> след извикването на копиращия конструктор на Y
    //редът на копиране ще е A -> B

    Y(const Y& other) // при копиране сме извикали копиращия на Y
    {
        //това се случва всъщност
        //obj1(other.obj1); -> извикваме копиращия на A
        //obj2(other.obj2); -> извикваме копиращия на B
    }
};

int main()
{
    //продължавайки горния пример:

    //викане на копиращ конструктор

    A obj; //викане на DEFAULT конструктор
    A obj2(obj); //викане на КОПИРАЩИЯ конструктор, подавайки обекта [obj]

    return 0;
}
```

```
int main()
{
    A obj;

    f(obj); //функцията [f] приема копие от тип [A]
    //тоест [a] ще копира [obj] (вика се копиращия конструктор)

    g(obj); //функцията [g] приема РЕФЕРЕНЦИЯ към тип [A]
    //тоест [a] е референция на [obj]

    //копиращият конструктор НЯМА да се извика,
    //тъй като правим референция към [obj], а не го копираме

    return 0;
}
```

02. Оператор = (оператор за присвояване)

- приема обект от същия тип и текущия става негово копие
- текущият обект е съществувал преди това

```
A obj1;
A obj2;

obj1 = obj2; //obj1 вече Е СЪЩЕСТВУВАЛ
              //всички данни на obj2 се копират в obj1
```

Разликата между копиращия конструктор и оператор= е:

копиращия конструктор

01. копира данните

оператор=

01. изчиства текущите данни

02. копира данните

!Ако не създадем оператор= (разпишем), компилаторът ще създаде такъв сам!

```
class A
{
    int a;
};

class B
{
    int b;
};

class X
{
    A obj1; //обект
    B obj2; //обект
};

int main()
{
    X obj1;
    X obj2;

    obj2 = obj1; //извикване на оператор= (в случая default-ния такъв)

    //подобно на default-ния копиращ конструктор, тук се
    //извика първо оператор= на [X], след което и тези на [A] и [B]
    //(след [X] викането на операторите= зависи от реда на декларация на
    //член-данните (отново за примитивни типове просто се копират данните))

    return 0;
}
```

```

class A
{
    int a;
};

class B
{
    int b;
};

class X
{
    A obj1; //обект
    B obj2; //обект

public:
    X& operator=(const X& other) //забелязваме, че [operator=] връща РЕФЕРЕНЦИЯ
    {
        //това е така, за да можем да използваме [operator=] верижно

        //това нямаше да бъде възможно ако [operator=] беше [void],
        //тъй като нямаше да връщаме нищо:

        //obj3 = obj2 = obj1; щеше да доведе до компилационна грешка,
        //защото obj2 = obj1 няма да върне стойност, която може да се присвои на obj3

        //а връщането по референция, а не по копие, за да спестим ненужно копиране
        //(с връщане по копие [operator=] все още ще работи, но по-неективно)

        obj1 = other.obj1;
        obj2 = other.obj2;

        return *this;
    }
};

int main()
{
    X obj1;
    X obj2;

    obj2 = obj1; //извикване на оператор=

    X obj3;
    obj3 = obj2 = obj1; //верижно ползване на operator=

    return 0;
}

```

```
int main()
{
    X obj1;
    X obj2 = obj1; //обръщаме внимание, че за да се извика operator=
                  //има изискване и двата обекта да съществуват,
                  //тъй като [obj2] не съществува, а го създаваме в момента,
                  //това ще извика копиращия конструктор, където
                  //[[obj2]] ще се създава и ще му бъдат зададени стойностите на [obj1]

    return 0;
}
```

```
int main()
{
    X obj1;
    X obj2;

    obj1 = obj2; //тъй като [obj1] вече съществува, ще се извика [operator=],
                  //който по дефиниция ще изпълни следните две стъпки
                  //01. ще изтрие данните на [obj1]
                  //02. ще копира в [obj1] данните на [obj2]

    return 0;
}
```

```

int main()
{
    X obj1;

    obj1 = obj1; //тъй като [obj1] вече съществува, ще се извика [operator=],
                 //който по дефиниция ще изпълни следните две стъпки
                 //01. ще изтрие данните на [obj1] (левият обект)
                 //02. ще копира в [obj1] (левият обект) данните на [obj1] (десният обект)

    //това обаче води до очевиден проблем,
    //тъй като първата стъпка е да се изтрият данните на [obj1] (левият обект) =>
    //в [obj1] (левият обект) не можем да присвоим данните на [obj1] (десният обект),
    //тъй като обектите са един и същ, и с триенето на данните на левия са изтрити и тези на десния,
    //(тоест на левия обект искаме да присвоим изтрити данни)
}

```

```

class X
{
    A obj1; //обект
    B obj2; //обект

public:

    X& operator=(const X& other)
    {
        if (this != &other) //проверяваме дали обектите са различни,
                           //ако са, няма да има проблем при стъпка 01

            obj1 = other.obj1;
            obj2 = other.obj2;

    }

    return *this; //ако НЕ СА различни, просто ще върнем непроменения наш обект
                  //(няма да влезем в if-а)

                  //ако СА различни, ще върнем променения наш обект
    }
};

int main()
{
    X obj1;

    obj1 = obj1; //вече сме решили проблема

    return 0;
}

```

RVO/NRVO

```
//RVO и NRVO се прилагат автоматично от компилатора
//(целят да подобят ефективността на нашия код)

//RVO - return value optimization
//спестява копиращия конструктор
//(когато обекта на връщане НЯМА име)
A createA()
{
    return A(); //извиква се default-ният конструктор на A
}

A obj = createA() //очаква се да се извика копиращия конструктор,
//но това не се случва заради RVO оптимизацията,
//тоест влизаме във функцията, викаме default-ния
//конструктор на A и благодарение на RVO оптимизацията
//не се вика копиращия конструктор, въпреки това в [obj]
//се запазват данните

//NRVO - named return value optimization
//спестява копиращия конструктор
//(когато обекта на връщане ИМА име)
A createA()
{
    A obj; // извиква се default-ният конструктор на A

    return obj;
}

A obj = createA() //очаква се да се извика копиращия конструктор,
//но това не се случва заради NRVO оптимизацията,
//тоест влизаме във функцията, викаме default-ния
//конструктор на A и благодарение на NRVO оптимизацията
//не се вика копиращия конструктор, въпреки това в [obj]
//се запазват данните
```

Тема 06. Голямата четворка

Димитриев

Проблем при генерираните от компилатора operator= и копиращ конструктор имаме при работа с динамична памет

```
//ще работим със следния код

class T
{
    char* str; //напомняме, че [str] е ПОЙНТЪР, а НЕ МАСИВ
    int n;

public:
    T()
    {
        str = new char[7]; //насочваме ПОЙНТЪРА [str] към динамичен масив
    }

    ~T()
    {
        delete str;
    }
};

int main()
{
    T t1;

    T t2(t1); //викане на копиращия конструктор

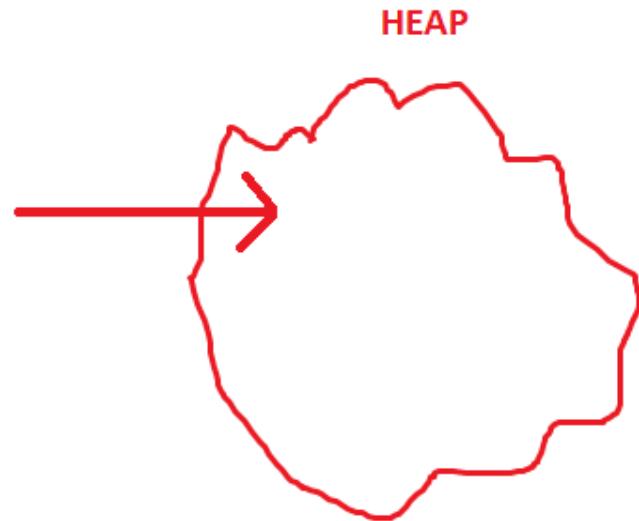
    return 0;
}
```

Какво се случва всъщност при викането на копиращия конструктор?

```
class T
```

```
    char* str  
    int n  
  
    T()  
    {  
        str = new char [7];  
    }
```

```
    ...  
    ...  
    ...
```



От УП знаем, че поинтъра **str** съдържа адреса на обекта, към който сочи (в случая масив от 7 елемента (с терминиращата 0), А НЕ СЪДЪРЖАНИЕТО НА МАСИВА. То тогава при извикването на копиращия конструктор (**default**) в **t2** ще се изпълни скришно **[!]** **t2.str = t1.str [!]**, което ще запази адреса, към който сочи член-данната **str** на **t1**. Тоест член-данныте на двата обекта сочат **към едно и също място в паметта (към един и същи масив)**. Това копиране че нарича **shallow copy**. Защо това е проблем?

Тъй като имаме заделена динамична памет на член-данна, за чийто живот отговаря нашия обект, то по правило в деструктора на класа ще трябва да я освободим. Но какво се случва всъщност?

```

int main()
{
    T t1;
    T t2(t1); //копираме данните в [t2]

    //дотук видяхме, че двата пойнтъра в обектите сочат към един и същи адрес

    return 0;
}

// вече знаем, че в края на scope-а се викат деструкторите на деклариралите от нас статични обекти
// и че също така деструкторите се викат в обратен ред на конструкторите =>
// ще извикаме деструктора на [t2], след което този на [t1]
//[!НО!] когато извикаме деструктора на [t2], се освобождава паметта, заделена за масива, към който
// сочи пойнтъра,
// това означава, че когато извикаме деструктора на [t1], той ще иска да освободи вече освободена памет
// (тъй като пойнтъра на [t1] сочи към същия масив) => грешка

```

За да решим този проблем, когато имаме динамична памет разписваме така наречената **“Голяма четворка”** (default конструктор, деструктор, operator=, копиращ конструктор). Не ги пишем, когато нямаме член-данна, работеща с динамична памет, за чито живот трябва да се грижи обекта, в който се намира.

Напомняме, че:

- **конструктор** - създава обект и инициализира член-данните му
- **копирацият конструктор** - създаване обект като копие на съществуващ обект (копира)
- **operator=** - копира стойностите на член-данните от един обект в друг вече съществуващ обект (трие + копира)
- **деструктор** - освобождава ресурсите, заети от обекта

Мои неща:

Разписване на голямата четворка

Освен конструктори и деструктор (които вече разгледахме) при наличието на динамични член-данни е необходимо пренаписването на допълнителни две функционалности (**копиращ конструктор** и **оператор=**).

```
class A
{
private:
    int x = 0;
    int y = 0;

    void copyFrom(const A& other)
    {
        this->x = other.x;
        this->y = other.y;
    }

public:
    //A(); -> това не се създава само

    A(int a, int b); //припомняме, че ако създадем конструктор,
                      //НЯМА да се създаде default-ен, а ако искаме такъв
                      //трябва да го разпишем САМИ

    A(const A& other) //конструктор, който приема като параметър ЕДИНСТВЕНО
    {
        //обект от същия тип

        //неговата основна функция е да запази стойностите на [other]
        //в НОВ обект (който досега не е съществувал) от същия тип

        copyFrom(other); //за разлика от default-ния конструктор, копиращият такъв,
                          //се създава винаги, независимо дали има други или не,
                          //но ни дава свобода да го презапишем с наша логика,
                          //тъй като не е предназначен да се справя с ДИНАМИЧНО заделени
                          //член-данни на класа

        //в случая НЯМА ДИНАМИЧНО заделена член-данна, тоест
        //default-ният копиращ конструктор може да се справи с
        //промяната на данните сам, затова в примера е излишно това
        //пренаписване на копиращия в конструктор, но го правим, за да
        //видим какво всъщност става отзад, когато не го разписваме
    }
};
```

```

class A
{
private:

    int x = 0;
    int* nums = nullptr;
    size_t numsSize = 0;

    void copyFrom(const A& other)
    {
        this->x = other.x; //копирането на [x] става по стандартен начин

        this->nums = new int[other.numsSize] {}; //тъй като искаме динамичен масив,
                                                //трябва да заделим такъв с размера на този,
                                                //в обекта, чито стойности искаме да вземем
                                                //и да пренасочим поинтъра към него (новия масив)

        for (unsigned i = 0; i < other.numsSize; i++)
        {
            nums[i] = other.nums[i]; //записване на стойностите
        }
        this->nunsSize = other.numsSize;
    }
}

-----
public:

A() = default; //когато имаме зададени default-ни стойности на член-данините
                //е достатъчно да кажем, че искаме да имаме default-ен, който да използва
                //зададените default-ни стойности на член-данините

A(int x, int* nums, size_t nunsSize) {/*тило*/};

A(const A& other)
{
    copyFrom(other);
}

~A()
{
    delete[] nums; //тъй като обектът е отговорен за живота на nums
                    //(инициализира се в рамките на този клас)
                    //трябва да освободим заделената памет за него в деструктора

    nums = nullptr; //след освобождаване на паметта, nums сочи към вече освободена памет
                    //и е добра практика да му кажем да сочи към нищо (nullptr)
}
};

```

```
class A
{
private:
    int x = 0;
    int* nums = nullptr;
    size_t numsSize = 0;

    void copyFrom(const A& other)
    {
        this->x = other.x;

        this->nums = new int[other.numsSize] {};
        for (unsigned i = 0; i < other.numsSize; i++)
        {
            nums[i] = other.nums[i];
        }
        this->nunsSize = nunsSize;
    }

    void free()
    {
        delete[] nums; //освобождаваме паметта заделена за [nums]
    }
}
```

```
public:

    A() = default;
    A(int x, int* nums, size_t numsSize) /*тило*/ ;
    A(const A& other)
    {
        copyFrom(other);
    }

    A& operator=(const A& other) //когато A съществува, се използва така наречения оператор
    {
        //за присвояване (който също съществува ВИНАГИ, но имаме
        //право да го презапишем с наша логика)

        if (this != &other) //проверяваме дали не присвояваме същия обект на текущия
        {
            //тъй като, ако се случи това ще освободим паметта и ще пренасочим пойнтьра
            //към вече освободена памет, а не същата

            free(); //ЗАДЪЛЖИТЕЛНО освобождаваме паметта преди да запишем новите данни,
            //тъй като в противен случай ще преместим пойнтьра към ново място
            //и ще имаме заделена памет, която не използваме и до която нямаме достъп
            //(memory leak)

            copyFrom(other);
        }
    }

    return *this;
}

~A()
{
    delete[] nums;
    nums = nullptr;
}
};
```

Пример за подробно разписана голяма четворка

https://github.com/Zapryanovx/FMI_2024_Object_Oriented_Programming/tree/main/00_demos/Big%20Four

Кога се извикват конструктор/деструктор/operator=/копиращ конструктор

01. Конструктор

https://github.com/Zapryanovx/FMI_2024_Object_Oriented_Programming/blob/main/00_demos/Big%20Four%20Calls/constructor.cpp

02. Копиращ конструктор

https://github.com/Zapryanovx/FMI_2024_Object_Oriented_Programming/blob/main/00_demos/Big%20Four%20Calls/copy.cpp

03. Оператор=

https://github.com/Zapryanovx/FMI_2024_Object_Oriented_Programming/blob/main/00_demos/Big%20Four%20Calls/op%3D.cpp

04. Деструктор

https://github.com/Zapryanovx/FMI_2024_Object_Oriented_Programming/blob/main/00_demos/Big%20Four%20Calls/destructor.cpp

Тема 07. Предефиниране на оператори. Приятелски класове и функции.

def оператор - функция със специален синтаксис

Операнд - нещото, върху което операторите извършват своето действие - константи, променливи, изрази, функции

Видове оператори в C++ има **3 вида оператори**:

Унарни оператори - действат върху един операнд

Примери:

- унарен минус (-), който променя знака на числото
- унарен плюс (+), който запазва знака
- логическо отрицание (!), което обръща булевата стойност (например, `!true` е `false`).
- `(++)` и `(--)`, които увеличават или намаляват стойността с единица

Бинарни оператори - действат върху два операнда

Примери

- оператори като събиране (+), изваждане (-), умножение (*), и деление (/).
- логически оператори като И (`&&`), ИЛИ (`||`).
- оператори за сравнение като равно (`==`), по-голямо (`>`), по-малко (`<`).

Тернарен оператор - има един единствен тернарен оператор (действа между три операнда)

- условният оператор, изразен като **условие ? израз1 : израз2**, където ако условието е истина (`true`), резултатът е `израз1`; ако е лъжа (`false`), резултатът е `израз2`

```
int a = 0;
int b = 0;

//унарни оператори
a++;
-b;
//бинарни оператори
a + b;
if (a && b) { /*...*/ };
//тернарният оператор
a > b ? std::cout << "plovdiv" : std::cout << "plovidiv";
```

Характеристики на оператори

Приоритет на операторите - реда, в който операциите се изпълняват (операторите с по-висок приоритет се изпълняват преди тези с по-нисък)

Например:

```
int a = 3;
int b = 4;
int c = 5;

// [*] > [+] (приоритет)
std::cout << a + b * c << std::endl; //23: 3 + 4 * 5 = 2 + 20 = 23
//           ^
//           |
//операторът между [b] и [c] се извършва преди този между [a] + [b] заради приоритета
```

Асоциативност - определя реда, в който операторите с еднакъв приоритет се изпълняват (всички оператори с еднакъв приоритет са или **ляво-асоциативни**, или **дясно-асоциативни**, т.е. няма оператори с еднакъв приоритет, при които единият да е ляво-асоциативен (от ляво надясно), а другият дясно-асоциативен (от дясно наляво))

- **ляво-асоциативни:** операторите се изпълняват от ляво надясно (оператори като събиране ('+'), умножение ('*'), логическо И ('&&')
- **дясно-асоциативни:** операторите се изпълняват от дясно наляво (оператори като събиране ('+='), умножение ('*='), логическо И ('&='))

```
int a = 3;
int b = 4;
int c = 5;

// [+] е ляво-асоциативен
//----->
std::cout << a + b + c << std::endl; //12: 3 + 4 + 5 = 7 + 5 = 12

// [=] е дясно-асоциативен
//<-----
a = b = c; //a = b = c -> b = c ([b] става 5)
//           -> a = b ([b] е станало 5, [a] става 5)

std::cout << a; // 5
```

Позиция - разположението на оператора спрямо operandите

- **префиксни** - оператори, които стоят **ПРЕД** operand-a
- **постфиксни** - оператори, които стоят **СЛЕД** operand-a
- **инфиксни** - оператори, които са разположени **МЕЖДУ ДВА ОПЕРАНДА**

```
int a = 3;
int b = 4;
int c = 5;

//префиксен оператор
++a;

//инфиксен оператор
a + b;

//постфиксен оператор
a++;

return 0;
```

Когато говорим за **обекти** (непримитивни типове), то нямаме дефинирани оператори като събиране ('+') и умножение ('*'), а трябва да ги дефинираме сами, което се извършва чрез създаването на нови функции (**външни и вътрешни за класа**), които симулират действието на даден оператор.

Ще използваме следната структура

```
struct A
{
    int a1 = 0;
    int a2 = 0;

public:
    A(int a1, int a2) : a1(a1), a2(a2)
    {
        ;
    }

    void printAll() const
    {
        std::cout << a1 << " " << a2 << std::endl;
    }
};
```

Първи начин на предефиниране на оператор за произволен оператор (който ще означим с \$) - **външна функция**

```
friend A& operator$(const A& other); //ще го разгледаме по-късно (декларация в тялото на [A])  
  
A operator$(const A& a1, const A& a2) //подаваме два обекта  
{  
    this->a1; // [X] функцията е външна, тоест нямаме достъп до член-данините на класа,  
    //тъй като щом е външна не сочи към свой собствен обект,  
    //а просто работи с такива обекти, което означава, че нито един от обектите  
    //формално не извиква оператора (тоест не се извиква от конкретен обект, тъй  
    //като функцията не се свързва (асоциира) с такъв)  
  
    //когато кажем, че функцията е friend (приятелска) това не променя това  
    //(ще разгледаме friend по-късно)  
  
    //логика, която да симулира действието на произволния оператор [$]  
    //  
    //  
}
```

Първи начин на предефиниране на оператор за произволен оператор (който ще означим с \$) - **вътрешна функция**

```
A& A::operator$(const A& other) //подаваме ЕДИН обект  
{  
    // (имаме предвид, че скришно се подава *this най-вляво),  
    // тъй като е ВЪТРЕШНА, означава че е член-функция =>  
    // има свой собствен обект, което означава,  
    // че ЛЕВИЯТ обект е този, който извиква оператора  
  
    this->a1 + other.a1; //имаме достъп до член-данините на this (текущия обект)  
  
    //логика, която да симулира действието на произволния оператор [$]  
    //  
    //  
}
```

Уговорка:

- операторите, които **ПРОМЕНЯТ** левия обект ще изнасяме във **ВЪТРЕШНИ** функции
- операторите, които **НЕ ПРОМЕНЯТ** левия обект ще изнасяме във **ВЪНШНИ** функции

При предефиниране на оператори трябва да се спазват **общоприети стандарти**

```
#include <iostream>

struct A
{
    int a1 = 0;
    int a2 = 0;

public:
    A(int a1, int a2) : a1(a1), a2(a2)
    {
        ;
    }

    void printAll() const
    {
        std::cout << a1 << " " << a2 << std::endl;
    }

    A& operator+=(const A& other);
};

A& A::operator+=(const A& other) //операторът [+=] ще симулира действието на прибавянето
                                //на стойностите на член-долните
{
                                //на десния обект към левия, тоест ЩЕ СЕ промени левия
                                //=> по уговорка ВЪТРЕШНА функция

                                //добавяме стойностите
                                //на втория обект към първия

    this->a1 += other.a1;          //връщането по референция (не void функция)
    this->a2 += other.a2;          // ни позволява така нареченото,
                                //chain-ване на оператори, т.е. ако имаме три обекта от
                                //тип [A] -> [first], [second], [third], то
                                //става валиден синтаксиса
                                //[[first] += [second]] += [third]

                                //напомняме, че ако връщаме по копие, chain-ването също
                                //е възможно, но връщаме по референция с цел да
                                //направим кода си по-ефективен

    return *this;
}
```

```

A operator+(const A& first, const A& second) //операторът [+] ще симулира действието събиране
                                                //между два обекта, тоест НЯМА ДА СЕ промени левия
{
                                                //=> по уговорка ВЪНШНА функция

    //тъй като вече сме предефинирали
    //оператора [+=] можем да го преизползваме

    //създаваме третия обект, който ще връщаме
    //директно чрез копиращия конструктор
    //(така данните му ще вземат стойностите на един от тях)
    A res(first);

    //напомняме, че при ВЪНШНО-предефинирани оператори, ще
    //връщаме ПО КОПИЕ, тъй като във функцията създаваме
    //трети обект, който да върнем, т.е. локален за функцията обект
    // => ще се изтреи след скоупа на функцията =>
    //ако върнем по референция, ще имаме референция към вече
    //несъществуващ обект, което ще доведе до [UB]

    //преизползваме [+=]
    // (променяме левия и му добавяме стойностите на десния)

    res += second;
    return res;
}

```

```

bool operator==(const A& first, const A& second) //левия не се променя => ВЪНШНА ФУНКЦИЯ
{
                                                //при предефиниране на operator== е прието да
                                                //връщаме тип [bool], тъй като операторът не е
                                                //предназначен за chain-ване =>
                                                // трябва просто да върнем [true], ако два обекта са
                                                //еквивалентни по даден стандарт и [false], ако не са
}

```

Особени случаи

1. Оператор<< и оператор>>

```
std::ostream& operator<<(std::ostream& os); //нека първоначално operator<< ни е обикновена
                                                //член-функция (напомняме, че уговорката тук НЕ важи,
                                                //тъй като е за оператори между два обекта от ЕДИН ТИП,
                                                //тука операторът е между ПОТОК и ОБЕКТ

std::ostream& A::operator<<(std::ostream& ofs) //връщаме по референция по аналогични причини на [=]
{
    //помним, че при член-функции като първи аргумент
    //винаги се подава скришно поинтъра [this], който
    //сочи към текущия обект, тоест можем да си представим
    //параметрите като (<обекта>, <потока>)

    //
    //какво променя това?

    return ofs << this->a1 << " " << this->a2; //връщаме потока след като запазим в него [a1], [a2]
}

int main()
{
    A a(2, 3);

    //тъй като скришно се подава this,
    //това означава, че ПЪРВО се подава this, ВТОРО - потока,
    //което води до обръщането на стандартния синтаксис, с който сме свикнали

    // [X] std::cout << a; (стала невалиден синтаксис)
    a << std::cout;
    // (this) << (<поток>) (реда, в който са ни параметрите на дефинираната функция operator<<)
    //      ^
    //      |
    //      оператор

    return 0;
}
```

```
//за да се придържаме към стандартния синтаксис  
//ще изнесем operator<< във външна функция, за да избегнем  
//скришното подаване на [this], което се извършва във всички  
//вътрешни функции, то тогава ще имаме параметри (<поток>, <обект>),  
//за разлика от вътрешната функция, която показвахме  
  
//за да достъпим член-данныте на обекта (които са private/protected)  
//във външната функция, ще кажем, че тя е friend (приятелска за обекта)  
  
friend std::ostream& operator<<(std::ostream& os, const A& obj);
```

```
std::ostream& operator<<(std::ostream& ofs, const A& obj) //връщаме по референция по аналогични  
{ //причини на [=] (по-ефективно chain-ване)  
  
    //не подаваме тайно [this], т.е. се придържаме към  
    //синтаксиса, на който сме свикнали  
  
    return ofs << obj.a1 << " " << obj.a2; //връщаме потока след като запазим в него [a1], [a2]  
}
```

```
int main()
{
    A a(2, 3);

    //тъй като направихме функцията външна,
    //това означава, че ПЪРВО се подава потока, ВТОРО - this,

    std::cout << a;
    // [X] a << std::cout; (стана невалиден синтаксис)
    // (<поток>) << (this) (реда, в който са ни параметрите на дефинираната функция operator<<)
    //           ^
    //           |
    //     оператор

    return 0;
}

//функцията за operator>> е аналогична
friend std::istream& operator>>(std::istream& is, A& obj); //за да се придържаме към
                                                               //стандартния синтаксис

std::istream& operator>>(std::istream& is, A& obj) // [NOTE] : забелязваме, че потоците, които
{                                                 // връщаме и подаваме са ostream и istream
    // (това е така, защото НЕ работим с файлове,
    // а с всякакви потоци (поддържащи вход/изход)

    return is >> obj.a1 >> obj.a2;
}
```

```
int main()
{
    A a(2, 3);

    std::cin >> a;

    return 0;
}
```

Предефиниране на оператори за имплементация и деплементация

a++ -> връща старата стойност (**a - 1**), т.е. увеличава **a** с единица, но функцията на оператора връща стойност **a - 1**

++a -> замества се с новото **a** (новата стойност), т.е. увеличава **a** с единица и оператора връща новата стойност на **a**

```
A& operator++(); //променяме обекта => вътрешни
A operator++(int);
```

```

//забелязваме, че prefix-ния ++ (++a) се връща по референция, докато
//postfix-ния ++ (a++) се връща по копие

//това идва от дефиницията на двета оператора в C++,
//която води до това, че prefix-ния може да се chain-ва,
//докато postfix-ния - не, т.е. [++++a] е валидно, но [a++++] - не

//идеята на postfix-ния е да върне старата стойност на [a],
//а на prefix-ния - новата =>
//postfix прави копие на [a] преди да го увеличи и връща КОПИЕТО,
//prefix увеличава [a] и го връща

A& A::operator++()
{
    this->a1++;
    this->a2++;

    return *this; //връщаме текущия обект с НОВИТЕ му стойности (така работи prefix-ния++)
}

A A::operator++(int) //int се нарича dummy parameter (параметър, който не се използва)
{
    //той на практика не изпълнява никаква функционална роля освен
    //да помогне за различаването между префиксната и постфиксната форма на
    //оператора, т.е. служи за отбелоязка, която ни помага
    //да различим [a++] и [++a]

    this->a1++;
    this->a2++;

    return A(this->a1 - 1, this->a2 - 1); //връщаме СТАРИТЕ стойности
                                                //(така работи postfix-ния++)
}

int main()
{
    A a(2, 3);

    std::cout << a++.a1 << " " << ++a.a2 << std::endl; //2 5 (първо връща старата стойност на
                                                               // [a1] и увеличава [a] с 1 (postfix)
                                                               //=> имаме предвид, че [a1], [a2] се
                                                               //увеличават с [1] и [a2] става [4],
                                                               //след което увеличаваме [a] още веднъж
                                                               // (prefix) => [a2] се увеличава с [1],
                                                               //става на [5] и го отпечатваме

    return 0;
}

```

Забележки:

- някои оператори **ТРЯБВА** да са член-функции - =, [], (), ->
- с () можем да викаме обектите като **функции** (въпрос за теория), т.е. **obj();**; е **викане на функция**
- **ограничения**
 - някой оператори **не могат** да се предефинират: ?: (тернарния оператор), :: (оператор за резолюция), . (оператор за достъп)
 - нови оператори **НЕ** могат да се създават (\$ в горния пример е невалиден оператор)
 - **НЕ** можем да предефинираме характеристиките на операторите - асоциативност/приоритет/позиция спрямо аргументите
 - оператора -> трябва да връща **обект (указател/референция/копие)**
 - при предефиниране на &&, || губим **early exit** (предварително изчисляване), т.е.
 - при **&&** не спира при първо грешно (не прави **false** оценка веднага)
 - при **||** не спира при първо вярно (не прави **true** оценка веднага)

Демото, разписано за предефиниране на оператори:

https://github.com/Zapryanovx/FMI_2024_Object_Oriented_Programming/tree/main/00_demos/Operators%20Redefinition

Приятелски класове/функции

def| класове/функции, които имат достъп до private имплементациите ни

```
#include <iostream>

class B
{
    A obj;
public:
    B()
    {
        obj.x = 7; //тъй като класът [B] е приятелски на [A]
    };           //можем директно да достъпим private член-данната [x] на [obj]

    friend void g();

    //...

};

class A
{
    int x;
public:
    A() = default;

    friend void f();
    friend class B;

    //friend void g(); //за да имаме достъп до private член-данните на [A] в [g]
                    //трябва да кажем, че тя е приятелска за този клас

    //...
};
```

```
void f()
{
    A obj;
    obj.x++; //аналогично на [B], във функцията [f] също можем да достъпваме private
              //член-данныте на [obj]
}

//приятелят на моя приятел НЕ Е мой приятел:

//класът [A] има приятелски клас [B], който има приятелска функция [g],
//това обаче НЕ ни дава достъп до private член-данныте на [A] във функцията [g]
void g()
{
    A obj;
    //obj.x++;
}

int main()
{
    return 0;
}
```

Тема 08. Статични член-дани. Изключения.

Static

- **Static локални променливи (статични променливи в тялото на функция)**
 - държи се в паметта на **глобалните/статичните променливи**
 - **static** променливите се създават в началото на програмата и се унищожават в края на програмата (а не в края на scope-а)
 - инициализира се **само веднъж** - при първото влизане в съответния scope и запазва стойността си дори след като излезе от scope-а

```
#include <iostream>

void increment()
{
    static int valueStatic = 0; //инициализира се само при първото викане
                                //на функцията [increment], след това единствено
                                //променя стойността (++value) и

    int valueAuto = 0;
    ++valueStatic;
    ++valueAuto;

    std::cout << valueStatic << " " << valueAuto << std::endl;
}

//тук се освобождава паметта, заделена за [valueAuto]
//(след края на scope-a)
```

```
int main()
{
    increment(); //първо викане на [increment]
    //01. [valueStatic] се инициализира
    //02. добавя се единица към [valueStatic]
    //03. [valueAuto] се инициализира
    //04. добавя се единица към [valueAuto]
    //=> valueStatic == 1 && valueAuto == 1

    increment(); //второ викане на [increment],
    // [valueStatic] е вече инициализирана
    //(тъй като е static, няма да се инициализира
    //наново, както сме свикнали да очакваме)
    //=> стъпка [01] ще бъде пропусната =>
    //02. добавя се единица към [valueStatic]
    //03. [valueAuto] се инициализира НАНОВО
    //04. добавя се единица към [valueAuto]
    //=> valueStatic == 2 && valueAuto == 1

    increment(); //3 1

    //напомняме, [static] променливите се държат в паметта на глобалните/статичните променливи,
    //но това НЕ означава, че valueStatic е глобална променлива

    //valueStatic++; // [X] когато променлива е декларирана като static в рамките на функция,
    //видяхме, че запазва стойността си между различните извиквания на функцията,
    //(тоест не се инициализира наново на всяко извикване)

    //![НО] въпреки това, тя е достъпна само в блока, в който е декларирана
    //(в случая скоупа на функцията [increment])

    return 0;
}

//тук се освобождава паметта, заделена за [valueStatic]
//(след края на програмата)
```

Димитриев

```
f()
{
    A obj;
    static B obj2; //static -> общ за всички
                    //извиквания на функцията
    C obj3;

    //...
    //логика
    //...
    //

}

int main()
{
    f(); //ще извика конструкторите на [A], [B], [C],
    //тъй като са локални, паметта ще се освободи в края на scope-а
    //![HO] помним, че static променливите се изтриват в края на изпълнението на
    //програмата => ще се извикат само деструкторите на [A], [C]

    f(); //помним, че статичните данни се инициализират само първия път,
    //след което запазват старата си стойност на всяко извикване
    //=> на второто извикване на функцията, [B] вече съществува
    //=> ще извикаме само конструкторите на [A], [C] и съответно
    //техните деструктори в края на scope-а на функцията

    return 0;
}

} //крайт на програмата => тук ще се унищожи [B]
```

Създаване на обекта: при първото извикване на функцията

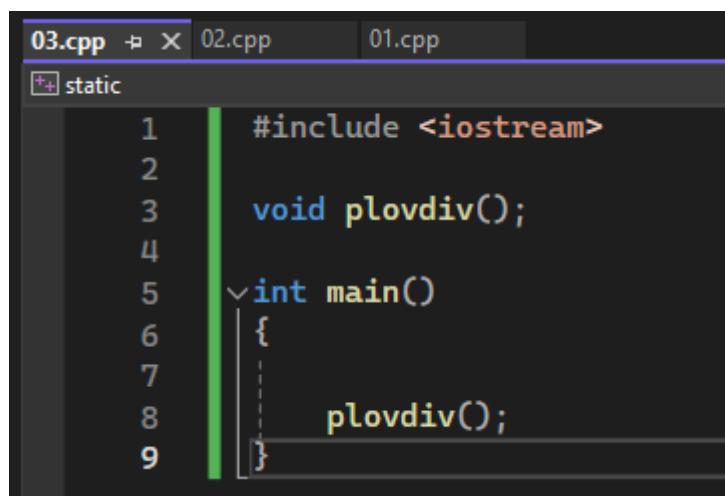
Изтриване на обекта: при излизане от програмата

Static функции

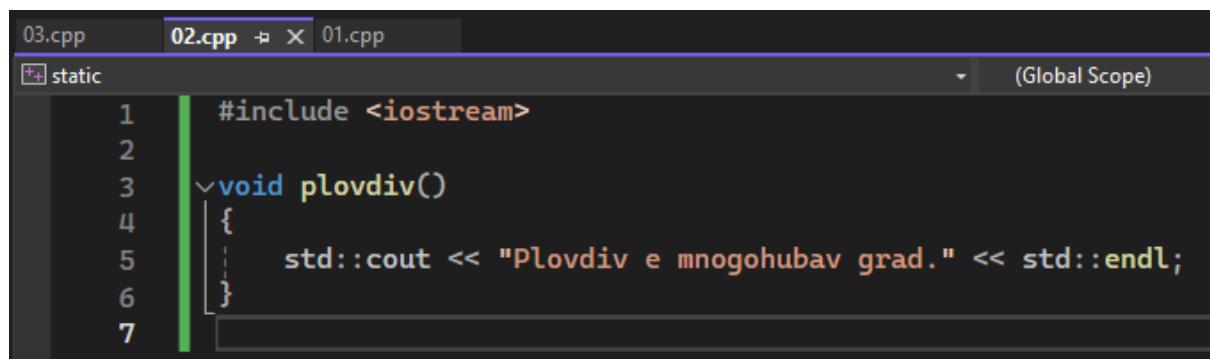
- **видимост** - когато функция е декларирана като **static** (извън всички класове), това ограничава нейната видимост само до файла, в който е дефинирана. Това предотвратява функциите от други файлове да извикват тази функция, дори ако имат функция със същото име.

Накратко: обвързва се с **ЕДНА** компилационна единица (**.obj файл**) и не може да се използва от други файлове

Досега:



```
#include <iostream>
void plovdiv();
int main()
{
    plovdiv();
}
```



```
#include <iostream>
void plovdiv()
{
    std::cout << "Plovdiv e mnogohubav grad." << std::endl;
}
```

От разделната компилация и **линкинга** научихме, че по време на процеса на **линкинга** **03.cpp** ще открие дефиницията на функцията **plovdiv()**, която извиква във файла **02.cpp**. Тоест ще се отпечата съответния изход.

03.cpp 02.cpp 01.cpp

(Global Scope)

```
static
1 #include <iostream>
2
3 void plovdiv()
4 {
5     std::cout << "Plovdiv e mnogohubav grad." << std::endl;
6 }
7
```

Когато обаче кажем във файла **02.cpp**, че функцията е **static**, то тогава казваме, че тази функция може да се използва **CAMO** в този файл => Няма да бъде открита от файла **03.cpp** => **Linking** грешка, тоест не е открита дефиниция за тази функция

03.cpp 02.cpp 01.cpp

(Global Scope)

```
static
1 #include <iostream>
2
3 void plovdiv()
4 {
5     std::cout << "plovdiv 1" << std::endl;
6 }
7
```

03.cpp 02.cpp 01.cpp

(Global Scope)

```
static
1 #include <iostream>
2
3 void plovdiv()
4 {
5     std::cout << "plovdiv 2" << std::endl;
6 }
7
8 int main()
9 {
10
11     plovdiv();
12 }
```

За да оправим настъпилата **Linking** грешка, то трябва да напишем дефиниция за функцията **plovdiv()** и във файла **03.cpp**. След компилация ще се отпечата **“plovdiv 2”**, защото ще намери тази дефиниция. (вече казахме, че **static** функциите са достъпни **САМО** във файла, в който се намират)

Можем да забележим, че имаме две функции с **ЕДНАКВИ** имена и **ЕДНАКВИ** параметри, което не беше възможно досега, тоест можем да направим извода, че:

- **static** функциите ни помагат с **капсулатията** една функция да не може да бъде използвана от други файлове
- позволява ни декларацията на функция със **същото име и същите параметри** в друг файл (друг вариант е **namespace { f(); }**)

Static член-променливи

01. не е обвързана с конкретен обект, а с целия клас
02. всички обекти от класа използват една и съща инстанция
03. инициализира се извън класа

```
#include <iostream>

class A
{
public:
    static int x;

public:
    A() = default;
};

int A::x = 10; //статичните член-данни на класовете се инициализират извън класа,
               // (в глобалното пространство на файла, където класът е дефиниран)
               //
               //това се дължи на факта, че статичните член-данни принадлежат на класа,
               //а не на специфичен обект на този клас
               //
               //по този начин, те съществуват независимо от всякакви обекти, създадени от класа,
               // (те са външни) и са споделени от всички инстанции на класа.

int main()
{
    A obj1; //създаваме ДВА РАЗЛИЧНИ обекта
    A obj2;

    std::cout << obj1.x << " " << obj2.x << std::endl; //спрямо инициализацията -> 10 10

    obj1.x++; //увеличаваме член-данната [x] на [obj1] с 1

               //тъй като вече разбрахме, че статичните член-данни са свързани с КЛАСА
               //(тоест всички обекти от този клас, а НЕ СЪС СПЕЦИФИЧЕН такъв, то тогава следва,
               //че с увеличаването на член-данната [x] на [obj1], ще се увеличи
               //и член-данната [x] на [obj2], тъй като де факто [x] е обща член-данна
               // (споделена от всички инстанции на класа) за двата обекта

    std::cout << obj1.x << " " << obj2.x << std::endl; //11 11
}
```

Димитриев

```
//04.h

#include <iostream>

class A
{
    int a = 0;
public:
    A() = default;
};

class B
{
    int b = 0;
public:
    B() = default;
};

class X
{
    A obj;
    static B obj2; // [01] глобален обект, скрит в клас X
                    // (енкапсулация в клас X)
                    // терминът енкапсулация в случая означава:
                    // скриване на вътрешните данни на обекта от външния свят

                    // [02] не се дефинира в конструктора, а ИЗВЪН него, тоест
                    // конструктора на [X] няма да извика конструктора на [B]
                    // деструктора на [X] няма да извика деструктора на [B].
                    // [B] се изтрива сам след изпълнението на програмата

                    // [03] не се обвързва с конкретен обект от този клас,
                    // а с всички обекти от този клас

                    // [04] статичните член-данни не са нито агрегация, нито композиция

public:
    X() = default;
};
```

```
#include "04.h";

B X::obj2; // дефинираме static член-данната
            // чрез конструктора на [B]

int main()
{
    // В X::obj2; // [X] обръщаме внимание, че статичните член-данни
                // са глобални => не можем да инициализираме
                // [obj2] в scope-а на main
}
```

![B] НЕ влияе на големината на обекта [X]

```
//04.h
#include <iostream>

class A
{
    int a = 0;
public:
    A() = default;
};

class B
{
    int b = 0;
public:
    B() = default;
};

class X //можем да кажем, че [X] играе ролята на namespace
{
    //за [obj2]

public:
    A obj;
    static B obj2;

public:
    X() = default;
};
```

```
//04.cpp
#include "04.h";

int main()
{
    X obj;

    //достъпване на статична член-данна обект

    obj.obj2; //можем, но не го правим

    X::obj2; //просто глобален обект (както при namespace)

    return 0;
}
```

Static член-функция

01. не е обвързана с конкретен обект, а с целия клас
02. използва се за достъпване на статичните член-данни
03. няма указател **this**
04. не е нужен обект, за да се достъпи

```
#include <iostream>

class Test
{
public:
    int z = 5;
    //static int r = 10; // [X] не можем да инициализираме static член-данна в класа

    static const int P = 20; // ОСВЕН, ако не е константа
private:
    static int x; // статична член-данна
    int y;

public:

    int getXStatic()
    {
        return x;
    }

    int getY()
    {
        return y;
    }
}
```

```
static void f()
{
    x *= 2; // може да достъпва само статичните член-данни

    //y += 2;      // [X] НЕ можем да достъпваме НЕстатични член-данни
    //this->y;    // [X] в статичните функции нямаме указател [this]
    //getY();      // [X] НЕ можем да достъпим НЕстатични член-функции
    //this->getY(); // [X] в статичните функции нямаме указател [this]
    //z++;        // [X] НЕ можем да достъпваме НЕстатични член-данни

    //x += p; //можем да достъпим СТАТИЧНИТЕ член-данни [x], [p]
}

static void r()
{
    f(); // в СТАТИЧНИ член-функции можем да извикваме
          // единствено СТАТИЧНИ член-функции

    //g(); // [X] НЕ можем да извикваме НЕстатични член-функции

    std::cout << "r" << std::endl;
}

void g()
{
    r(); // в НЕстатични член-функции можем да извикваме
    f(); // както НЕстатични член-функции, така и СТАТИЧНИ член-функции
    k();
}

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

int Test::x = 3; // трябва да я инициализираме извън класа
```

```
int main()
{
    Test::f(); // не ни трябва обект, за да я извикаме

        // забелязваме, че [f] променя статичната член-данна [x] (x *= 2),
        // вече знаем, че статичните член-данни са общи за класа (всички обекти Test)

    Test a;      //=> новосъздадените обекти [a], [b] ще имат [x] == 3 * 2 == 6
    Test b;

    std::cout << b.getXStatic() << std::endl; //6

    a.f(); //макар и да фикаме [f] върху обекта [a], то това не променя факта,
           //че промяната на [x] за [a] е промяна на [x] за всички обекти от тип [Test]

    std::cout << b.getXStatic() << std::endl; // [x] == 6 * 2 == 12
}
```

Димитриев

```
#include <iostream>

class Z
{
    int x;
    static int y;
public:
    void g() //НЕстатичните функции имат достъп
    {           //до всички член-данни и член-функции

        f();
        x;
        y;
    }

    static void f() //СТАТИЧНИТЕ член-функции имат достъп
    {               //до СТАТИЧНИ член-данни и член-функции

        //СТАТИЧНИТЕ функции са външни функции (инициализират се извън класа)
        // (подобно на friend)
        //тоест не ни трябва обект, за да я извикаме

        y;
        //x;   // [X]
        //g(); // [X]
    }
};

int main()
{
    Z::f(); //можем да викаме СТАТИЧНИ член-функции без да създаваме обект, т.е.
            //СТАТИЧНИТЕ член-функции са ГЛОБАЛНИ функции част от класа

    //g(); // [X] НЕ можем да викаме НЕстатични член-функции без да създаваме обект

    return 0;
}
```

def| **Статичен клас** - клас, който има само статични член-данни

Изключения / Exceptions

def| Exception - сигнал, че има проблем

Приложение

Изключенията в C++ помагат при неочеквани или грешни условия, които се появяват по време на изпълнението на програмата. Чрез тях позволяваме на програмата да обработва грешки по зададен от нас начин.

Ключови думи, които ще използваме:

throw <обект>: когато възникне проблем в програмата, може да се "хвърли" изключение с помощта на ключовата дума **throw**, последвана от **<обект>**. Този **<обект>** може да бъде от всякакъв тип, но обикновено е от класа **std::exception** (напр. можем да "хвърлим" като **<обект>** числото **38**, както може и да "хвърлим" **std::runtime_error**).

try: след ключовата дума **try**, отваряме блок (**scope**), в който слагаме кода, който според нас е проблемен и за когото се прилагат изключенията. Тоест, в рамките на този **scope** се извършват операциите, които могат да хвърлят изключение.

catch: след блока **try** следват **един или повече** блокове **catch**, които улавят хвърлените изключения. Всеки блок **catch** обработва определен тип изключения. Кодът в блока **catch** определя как да се реши проблемът, предизвикан от изключението.

```
#include <iostream>

void error(const bool isValid)
{
    if (isValid)
    {
        std::cout << "no error" << std::endl;
        return;
    }
    else
    {
        //с ключовата дума [throw] сигнализираме, че
        //е възникнал някакъв проблем, а след него
        //"хвърляме" изключението, което искаме да обработим

        throw '2'; //char -> a = 30
        //throw "ASD"; //char масив -> a = 50
        //throw 412; // int -> [...] -> a = 60
    }
}
```

```

    int a = 0;

    try
    {
        //scope-a, в който слагаме кода,
        //в който може да възникне някакъв проблем

        error(true); //в първото извикване на функцията подаваме, че [isValid] е <true>
                    //=> всичко е наред, ще отпечата, че няма грешка и ще излезе от функцията

        error(false); //във второто извикване на функцията подаваме, че [isValid] е <false>
                    //=> нещо НЕ Е наред => ще "хвърлим" някаква грешка,
                    //за да сигнализираме, че нещо не е наред
    }

    //конструкцията, която обработва различните типове изключения
    //(влизаме тук, ако изобщо в [try] е хвърлено изключение

    catch (char ch) //ако обектът хвърлен след [throw] е от тип [char]
    {
        //помним синтаксиса throw <обект>
        a = 30;
    }
    catch (const char* str) //ако обектът хвърлен след [throw] е МАСИВ от тип [char]
    {
        a = 50;
    }
    catch (...) //с [...] се обозначава всичко, т.e int, char, bool, double, float...
                //дори ако е хвърлена инстанция на даден клас, но тъй като вече имаме
    {
        // [catch] за тип [char] напр., теоретически никога няма да влезе в този при
        //хвърлен [char], така че може да се каже, че хваща всичко останало

        a = 60;
    }

    std::cout << a << std::endl;
    return 0;
}

```

Приложение/Димитриев (github)

https://github.com/Angeld55/Object-oriented_programming_FMI/tree/master/Week%2008#exception-handling

Накратко

в **try** блока се намира проблемният код

ако се хвърли грешка, останалата част от **try** блока не се изпълнява

в **catch** блоковете грешката се обработва

може да има няколко **catch** блока, влизаме в този, който приема като параметър обекта, който е хвърлен (ако се хване грешката, проверката надолу спира)

catch (...) хваща всичко, ако до момента не е хванато - слага се накрая

Особености при инстанции на клас (Димитриев лекция)

```
#include <iostream>

// в следващите примери, ще използваме следните класове [X], [A], [B], [C]
// с цел визуализация, тъй като на лекции ги ползвахме наготово

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

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

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

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

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

```

void g()
{
    X obj; //X()

    throw 37; //хвърляме изключение (намерили сме проблем с кода)
    //функцията g() спира изпълнението си веднага при хвърлянето,
    //тоест НЯМА да създаде [obj2]
    //
    A obj2; //<-----| //след спирането на изпълнението, g() започва да търси
    //кой може да обработи грешката, като по пътя вика деструкторите
    //на всички обекти, създадени във функцията => се вика деструктора на [X]

    //грешката е прехвърляне на отговорност, тоест
    //в момента стека ни приема следния вид

    //g()
    // | функцията g() хвърля изключение, което се прехвърля в f(),
    // v тъй като g() е извикана в f()
    //f()
    // | програмата вижда, че f() не може да обработи изключението, хвърлено от g()
    // v и прехвърля отговорността изключението да бъде обработено нататък (на main)
    //main()

}

void f()
{
    A obj; //A()
    B obj2; //B()

    g(); //извикваме функцията g()

    //хвърлено е изключение (намерили сме проблем с кода)
    //от g() => функцията f() спира изпълнението си веднага при хвърлянето,
    //тоест НЯМА да създаде [obj3]
    //
    C obj3; //<-----| //подобно на разгледаното във функцията g()
    //след спирането на изпълнението, f() започва да търси
    //кой може да обработи грешката, като по пътя вика деструкторите
    //на всички обекти, създадени във функцията => се вика деструкторите на [A], [B]

    //междувременно не намира начин да обработи грешката и прехвърля
    //отговорността на main
}

```

```
113 int main()
114 {
115     f(); //извикваме функцията f()
116
117     //-[ЗАКЛЮЧЕНИЕ] Редът на действие е:
118     // f() -> A() -> B() -> g() -> X() -> throw 37 -> ~X() -> ~B() -> ~A() -> main()
119
120     return 0;
121 }
122 }
```

Тъй като в **main()** изключението не е обработено, сме длъжни да го направим, в противен случай кодът ни ще гръмне.

```
int main()
{
    try
    {
        //за примера ще използваме инстанции на класа [Y], [Z]
        //където класовете [Y], [Z] са идентични на останалите

        Y obj; //Y()

        f(); // --|   вече разглеждахме реда на действия в f()
        //
        //   тъй като f() не обработва изключението, хвърлено от g()
        //   то викането на f() ще хвърли изключение
        //
        //   проблемният код в [try] приключва изпълнението си,
        //   при първото хвърлено изключение, след което търси
        //   съответния [catch], за да го обработи
        //
        //   [!] Обръщаме внимание, че обектът от тип [Y] е създаден
        //   в тялото на [try] => ще се извика деструктора на [Y] след
        //   |-----|   като излезем от тялото на [try] и преди да влезем
        //   |-----|   в съответния [catch], който да обработи грешката
        //
        //предвид казано =>
        //#[obj2] няма да се
        //създаде
        Z obj2;           //|
        //|
    }                   //|
    catch (int x) // <-----|
    {
        std::cout << "error" << x;
    }

    return 0;
}
```



```
int main()
{
try
{
    //за примера ще използваме инстанции на класа [Y], [Z]
    //където класовете [Y], [Z] са идентични на останалите

    Y obj; //Y()

    f(); // --
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    // предвид казаното =>      | Допълнение [2]: ако имаме > 1 [catch], които могат да
    // [obj2] няма да се          | обработят грешката => ще се изпълни първият от тях
    // създаде                  | (този по-нагоре)
    Z obj2;                   ||
                           ||
    catch (int x) // <-----| {
        std::cout << "first error" << x; //| макар и двата [catch] да хващат [int]
    }                                //| ще се изпълни този, тъй като стои по-нагоре
                                       //| в изреждането на [catch]-овете ни
    catch (...)
    {
        std::cout << "second error"; //| ред на търсене на [catch], който
    }                                //| да обработи грешката ни

    return 0;
}
```

Всичко, казано дотук обобщава **Stack unwinding**, който има два случая:

- ако **НЯКОЙ** обработва грешката
- ако **НИКОЙ** не обработва грешката

(**std::terminate** -> незабавно прекратяване на изпълнението на програмата)

Stack unwinding

- при хвърляне на грешка **изпълнението на функцията се прекратява**
- програмата проверява дали текущата функция или някоя от извикващите функции нагоре по стека може да се справи с изключението (т.е. дали има **try-catch блок**)
- ако бъде намерен съответстващ **блок за обработка** на изключение, изпълнението се прескача от момента, в който е хвърлено изключението, до началото на съответстващия блок за обработка

[БЕЛЕЖКА]
(за “прескача”)

```
try
{
    //за примера ще използваме инстанции на класа [Y], [Z]
    //където класовете [Y], [Z] са идентични на останалите

    Y obj; //Y()

    f(); // --|
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //
    //предвид казаното =>
    // [obj2] няма да се
    // създаде
    Z obj2;           //|
}
                //|
catch (char ch) // <-----|
{
    std::cout << "error" << ch;
}
```

- това изиска **stack unwinding** (премахване на **текущата функция** от стека на повикванията) толкова пъти, колкото е необходимо, за да може функцията, обработваща изключението, да бъде най-горе в call stack-a

[БЕЛЕЖКА]: g() хвърля изключение и ни е **текущата функция**, премахваме я и f() ни става **текущата функция**. Тя не може да обработи изключението, премахваме **текущата функция f()** и отиваме в **main**. **Main** може да обработи изключението.

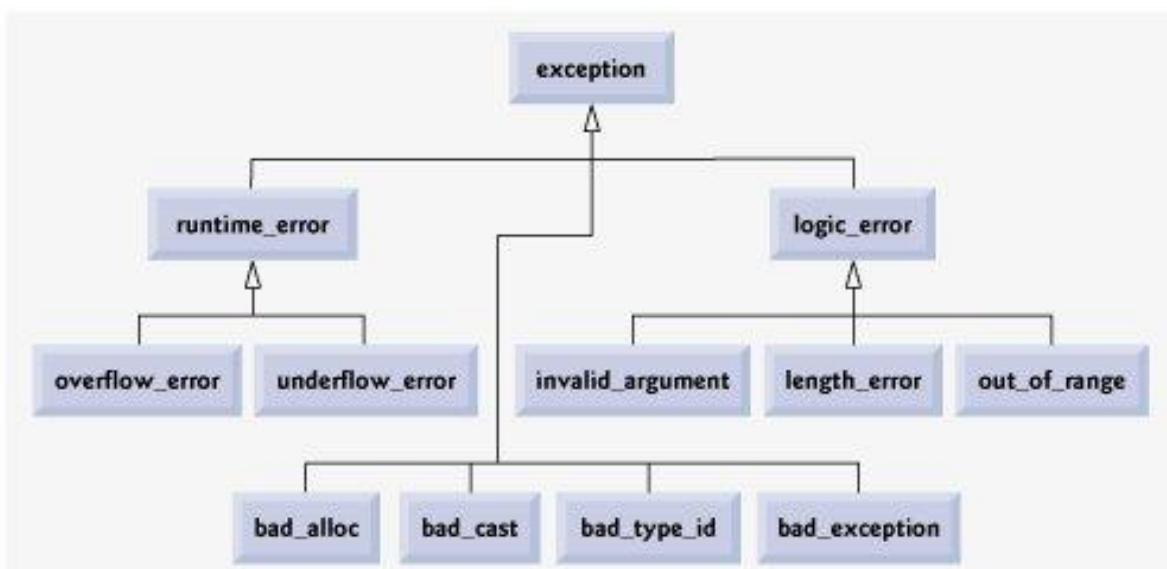
- когато текущата функция се премахне от call stack-a, всички успешно създадени локални променливи се унищожават както обикновено, но не се връща стойност

[БЕЛЕЖКА]: Викането на деструкторите от долу нагоре, което разгледахме (всички локални обекти/променливи се троят)

Exceptions vs. assert

- **assert** - системни/наши грешки
- assert(bool) - ако е **true**, ще спре програмата, ако е **false**, просто ще продължи напред
- **exception** - идеята е да сигнализираме на външния свят, че нещо се е случило некоректно и да обработим грешката (напр.: входни данни)

Видове грешки



std::bad_alloc - грешка при заделяне на памет (оператора **new**)

std::bad_cast - грешка при кастване (**преобразуване**)

std::runtime_error - грешка по време на изпълнение

std::logic_error - грешка, която наруши инвариантите на класа (условията, които трябва да са изпълнени) и може да бъде предотвратена

(напр.: рационално число число със знаменател 0)

std::out_of_range - грешка, наследник на **std::logic_error** (когато например излезем от масива)

Когато подреждаме catch блоковете, класовете по-нагоре в юерархията трябва да са по-надолу, защото са по-общи и хващат повече видове грешки. Тоест:

```
int main()
{
    try
    {
        Y obj; //Y()
        f();
        Z obj2;
    } //най-конкретната
    catch (std::bad_alloc) //|
    { //|
        // //|
    } //|
    catch (std::out_of_range) //out_of_range //|
    { //|
        // //|
        //е наследник на //|
        // //|
        //стое по-нагоре //|
    } //|
    catch (std::logic_error) //logic_error //|
    { //|
        // //|
        //е наследник на //|
        // //|
        //стое по-нагоре //|
    } //|
    //следните два [catch] пишем за всеки случай //|
    catch (std::exception) //|
    { //|
        // //|
    } //|
    catch (...) //|
    { //|
        // //|
        //v //|
        //към най-лошата (най-общата) //|
    } //|
} //помним, че [...] означава ВСИЧКО, т.е. //|
//сърържа и std::exception и неговите наследници
```

```
int main()
{
    try
    {
        Y obj; //Y()
        f();
        Z obj2;
    }                                //най-конкретната
    catch (std::bad_alloc)           //|
    {                                //|
        //                                //|
    }                                //|
    catch (std::out_of_range)        //|
    {                                //|
        //                                //|
        //е наследник на          //|
        //                           //logic_error и      //|
        //стои по-нагоре          //|
    }                                //|
    catch (std::logic_error)         //|
    {                                //|
        //                                //|
        //е наследник на          //|
        //                           //std::exception и //|
        //стои по-нагоре          //|
    }                                //|
    //следните два [catch] пишем за всеки случай //|
    //|
    catch (...)                      //|
    {                                //|
        //                                //|
    }                                //|
    catch (std::exception)           //|
    {                                //|
        //                                //|
        //                           //v
        //към най-лошата (най-общата)
    }
}
//от предходния извод следва, че [...] не може да стои преди друг [catch],
//защото ще го "замаскира"
```

```
int main()
{
    try
    {
        throw std::exception("ABC");
    }

    catch (const std::exception& e) //разбираме се, да приемаме обекти от НЕпримитивен тип
                                    //като константна референция в [catch], за да не ги копирате

        std::cout << e.what() << std::endl; //ABC, чрез e.what() можем да изведем съдържанието на грешката
    }

    catch (...)
    {
        //тук не можем да изведем съдържанието на грешката, тъй като [...]
        //може да приеме всичко и липсва параметър, който да улавя изключението
    }

    return 0;
}
```

Деструктор/конструктор

Деструктор

```
#include <iostream>

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

    ~A()
    {
        throw 37;
    }
};

void f()
{
    A obj;
    throw false; //хвърляме изключение [false] =>
                  //помним, че започват да се тряят всички създадени преди
                  //хвърлянето на изключението локални променливи =>
                  //ще се изтриве [obj] =>
                  //ще се извика деструктора на [A] в края на scope-a,
                  //който също хвърля изключение [37]
                  //=> имаме 2 хвърлени изключения и нито едното не е обработено
                  //=> програмата се терминира (std::terminate)
}
```

```
int main()
{
    try
    {
        //извикваме функцията f()
        f();
    }

    //не можем да обработим 2 изключения едновременно
    catch (int x)
    {
        //...
    }

    catch (bool is)
    {
        //...
    }

    return 0;
}
```

Извод - В **деструкторите** НЕ хвърляме изключения, защото ако този деструктор бъде извикан от друго изключение, програмата се терминира.

Конструктор

```
#include <iostream>

class A
{
public:
    A()
    {
        throw 37; //хвърляме изключение [37]

        //при хвърляне на изключения в конструктора се
        //извикват деструкторите на всички НАПЪЛНО построени
        //обекти в конструктора

        //![!] не се извиква деструктора на [A]
    }

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

int main()
{
    try
    {
        A obj; //A()
    }
    catch (...)
    {
        std::cout << "here" << std::endl;
    }

    return 0;
}
```

```
//в следващите примери, ще използваме следните класове [A], [B], [C], [X]
class A
{
public:
    A()
    {
        std::cout << "A()" << std::endl;
    }

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

class B
{
public:
    B()
    {
        std::cout << "B()" << std::endl;
        throw 37; //хвърляме изключение в конструктора на [B]
    }

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

```
32
33     \> class C
34     {
35         public:
36             C()
37             {
38                 std::cout << "C()" << std::endl;
39             }
40
41             ~C()
42             {
43                 std::cout << "~C()" << std::endl;
44             }
45     };
46
47     \> class X
48     {
49         A obj;
50         B obj2;
51         C obj3;
52     public:
53         X()
54         {
55             std::cout << "X()" << std::endl;
56         }
57
58         ~X()
59         {
60             std::cout << "~X()" << std::endl;
61         }
62     };
63
```

```
int main()
{
    X xObj;

    //създаваме обект от тип [X]
    //=> се извикват конструкторите в следния ред: [X] -> [A] -> [B] (както сме свикнали)
    //помним, че макар X() да се отпечатва последно, конструктора на [X] действително се извика първи
    //просто преди да стигнем до отпечатването викаме конструкторите на [A], [B]

    //когато стигнем до конструктора на [B] в него се хвърля изключение,
    //=> както казахме се извикват деструкторите на всички обекти създадени в конструктора
    //=> деструктора на [B] НЯМА да се извика, тъй като обекта НЕ е приключил създаването си успешно,
    //деструктора на [A] ще се извика, тъй като обекта е приключил създаването си успешно
    //деструктора на [X] няма да се извика, тъй като обекта НЕ е приключил създаването си успешно ([B] е хвърлил изключение)

    //=> A() B() ~A()

    return 0;
}
```

Извод - В конструкторите МОЖЕМ да хвърляме изключения, но не трябва да останат незатворени външни ресурси (динамична памет)

```
#include <iostream>

class A
{
    char* str;
public:
    A()
    {
        str = new char[10];

        throw 37; //вече разбрахме, че при хвърляне на
                   //изключение в конструктора,
                   //не се вика деструктора на този обект

                   //=> ~A() няма да се извика
                   //=> няма да се изчисти паметта,
                   // заделена за [str]

    }

    ~A()
    {
        delete[] str;
    }
};
```

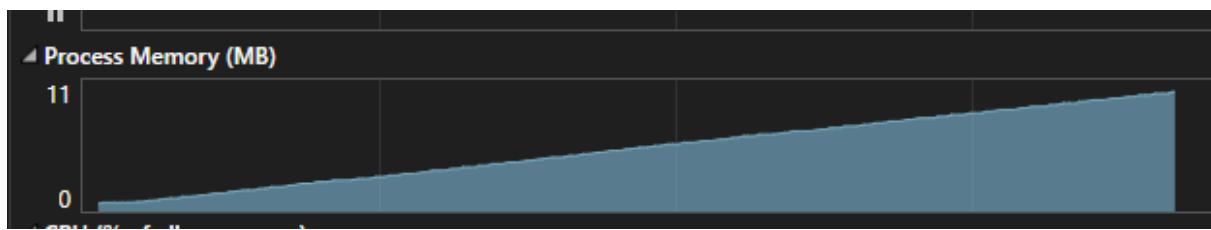
```

int main()
{
    //нека създадем безкрайно много обекти от тип [A]
    while (true)
    {
        //тъй като не се вика деструктора на [obj]
        //ще заделяме постоянно нова памет за член-данната [str]
        //без да я освобождаваме => memory leak
        try
        {
            A obj;
        }
        catch (int x) //#[catch] 37
        {
            std::cout << "err" << std::endl;
        }
    }

    return 0;
}

```

Ако проследим, заделянето на памет, ще видим, че на всяка итерация, паметта необходима на програмата се увеличава постоянно, това означава, че някъде имаме **memory leak** и не сме освободили паметта, заделена за стари данни => графиката на заделената памет потвърждава твърдението, че не се извиква деструктора на **[A]**, откъдето пък следва, че паметта за **[str]** никога не се освобождава



```

#include <iostream>

class A
{
    char* str;

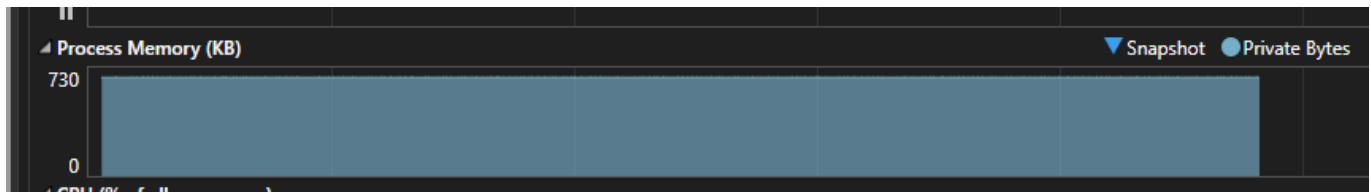
public:
    A()
    {
        str = new char[10];

        //тъй като примерът не е подходящ,
        //за да покажем, че трябва да изчистим паметта
        //сами, ще добавим някакво условие, което ако е изпълнено,
        //значи има проблем

        if (true) //някакво условие, което, ако е изпълнено
            //ще доведе до проблем
        {
            delete[] str;
            throw 37;
        }
    }

    ~A()
    {
        delete[] str;
    }
};

```



```
int main()
{
    //нека създадем безкрайно много обекти от тип [A]
    while (true)
    {
        //тъй като не се вика деструктора на [obj] ,
        //НО сме изчистили паметта, заделена за външните ресурси,
        //то НЯМАЕ memory leak
        try
        {
            A obj;
        }
        catch (int x) //#[catch] 37
        {
            std::cout << "err" << std::endl;
        }
    }

    return 0;
}
```

Какво прави оператор `new` в следните два сценария:

- **Успешна алокация (успешно създаване на динамичен масив):**

Ако `new` успешно задели памет, вече имаме създаден динамичен масив, готов за използване за нашите цели

- **Неуспешна алокация (неуспешно създаване на динамичен масив):**

Ако `new` не успее да задели памет, то хвърля изключение `std::bad_alloc`. Важно е да се отбележи, че в този случай `new` се погрижва за изчистването на паметта, която все пак е успяла да се задели.

```
#include <iostream>

class A
{
    char* str;
    char* strTwo;

public:
    A()
    {
        // [new] може да хвърли изключение
        str = new char[10];

        // ако втория масив [strTwo] хвърли изключение,
        // то паметта за него ще се изчисти сама
        // [!НО] няма кой да изчисти [str] => ще получим memory leak
        // ако се хвърли изключение при създаването на [strTwo]

        size_t size = -1; // максимална възможна стойност за size_t,
                           // която ще използваме, за да провокираме bad_alloc
                           // (системата няма достатъчно свободна памет, която да задели)

        strTwo = new char[size];
    }

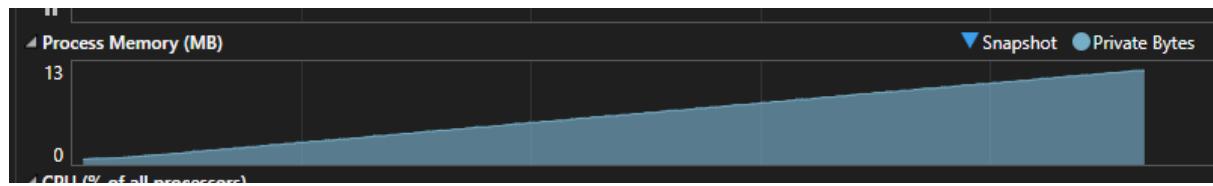
    ~A()
    {
        delete[] str;
        delete[] strTwo;
    }
};
```

```

int main()
{
    //нека създадем безкрайно много обекти от тип [A]
    while (true)
    {
        //тъй като [strTwo] хвърля изключение, [new] се грижи за
        //освобождаването на паметта, свързана със [strTwo],
        //но нямаме какво да изчисти [str] => ИМАЕЕ memory leak
        try
        {
            A obj;
        }
        catch (...)
        {
            std::cout << "err" << std::endl;
        }
    }

    return 0;
}

```



```
#include <iostream>

class A
{
    char* str;
    char* strTwo;

public:
    A()
    {
        str = new char[10];

        try
        {
            size_t size = -1;
            strTwo = new char[size];
        }

        catch (const std::bad_alloc& err)
        {
            delete[] str; //освобождаваме паметта,
                           //заделена за [str]

            throw; //хвърляме същата грешка
        }
    }

    ~A()
    {
        delete[] str;
        delete[] strTwo;
    }
};
```

```

int main()
{
    //нека създадем безкрайно много обекти от тип [A]
    while (true)
    {
        //тъй като [strTwo] хвърля изключение, [new] се грижи за
        //освобождаването на паметта, свързана със [strTwo],
        //И ВЕЧЕ сме се погрижили за изчистването на [str]
        // => НЯМАМЕ memory leak

        try
        {
            A obj;
        }
        //хващаме хвърленото изключение от A()
        catch (...)
        {
            std::cout << "err" << std::endl;
        }
    }

    return 0;
}

```



Аналогично, за три масива, ако имаме три масива, заделени динамично, които е възможно да доведат до хвърлянето на изключение

01. Ако [strOne] хвърли изключение => `new` ще се погрижи за паметта и не трябва да правим нищо допълнително
02. Ако [strTwo] хвърли изключение => `new` ще се погрижи за паметта за [strTwo] и ще трябва да изчистим паметта за [strOne] сами
03. Ако [strThree] хвърли изключение => `new` ще се погрижи за паметта за [strThree] и ще трябва да изчистим паметта за [strOne] и [strTwo] сами

```
#include <iostream>

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

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

```

class X
{
    A* ptr;
    char* arr;
public:
    X()
    {
        //01. заделя паметта за 10 A
        //02. вика 10 x A()
        //03. като някой хвърли изключение, вика деструкторите на миналите
        // Напр.: нека A[4] хвърли изключение
        // т.е. ~A[3], ~A[2], ~A[1], ~A[0]
        // (A[4] не се създава. защото е хвърлил изключение и
        //съответно не се вика деструктора му)
        //04. [new] се грижи в себе си да освободи паметта и осигурява,
        //че се изчиства паметта за незавършените обекти (тоест A[4], A[5] ... A[9])
        ptr = new A[10];

        //съответно деструктора ~X() не се извиква,
        //тъй като грешката е в конструктора X(),
        //чрез std::nothrow забраняваме на [new]
        //да хвърли изключение, ако създаването на [arr]
        //е НЕУСПЕШНО, вместо това, ако е НЕУСПЕШНО,
        // [arr] става [nullptr]
        arr = new (std::nothrow) char[30];
    }

    ~X()
    {
        delete[] ptr;
    }
};

```

```

int main()
{
    //
    //
    //
    //

    return 0;
}

```

Нива на exception safety (Семинар)

No throw guarantee (Гаранция за отсъствие на изключения)

Това е най-високото ниво на гаранция, при което се обещава, че дадена операция няма да хвърли изключение **ПРИ НИКАКВИ** обстоятелства. Тази гаранция често се осигурява при функции, които:

- използват само операции, които сами по себе си са no-throw.
- функции, които са деклариирани с noexcept (ключова дума, която гарантира, че функцията няма да върне изключение)
- поддържат състоянието непроменено при всякакви условия.

No throw guarantee

Можем да гарантираме за операциите, че ще успеят дори в ситуация на грешка. Ако грешка се случи тя ще бъде обработена вътрешно и потребителите на кода няма да разберат за нея.

Пример за такива функции обикновено са `move` конструкторите. Функция, която не хвърля грешки, се декларира като `noexcept`.

```
int f() noexcept
{
    return 2 + 3;
}
```

Също функции като `size()`, `empty()`, `capacity()` и тн имат `no throw guarantee`.

Примери за всеки споменат вариант:

```
#include <iostream>

//f() е no-throw функция,
//тъй като всички операции в нея
//няма как да хвърлят грешка
int f()
{
    int x = 3;
    int y = 5;
    int z = x + y;
    int d = z * y;
    //...
    //...

    return 3 + 1;
}
```

```
//g() гарантира, че няма да бъде хвърлена грешка
//чрез ключовата дума noexcept
int g() noexcept
{
    int arr[30] = {};

    //не можем да имаме
    //int* arr = new int[30];
    //тъй като това може да хвърли изключение,
    //което нарушава [noexcept] подсказката
    //(можем но трябва да го направим с [try-catch],
    //за да обработим грешката и да я премахнем)
    //...
    //...

    return 31;
}
```

```
//x() гарантира, че няма да бъде хвърлена грешка
//тъй като в себе си улавя всички възможни такива
int x()
{
    int* arr;
    char* ptr;

    //обработваме всички възможни изключения
    try
    {
        arr = new int[30];
    }
    catch (...)
    {
        arr = nullptr;
    }

    try
    {
        ptr = new char[30];
    }
    catch (...)
    {
        delete[] arr;
    }

    //...
    //...

    return 32;
}
```

```
int main()
{
    f();
    g();
    x();

    return 0;
}
```

Strong exception safety (Силна гаранция за безопасност при изключения)

Силната гаранция за безопасност при изключения означава, че операцията може да хвърли изключение, но ако това стане, състоянието на обекта ще остане непроменено.

```
void f(char*& str)
{
    //ако [new] хвърли изключение,
    //то [str] не се променя и остава
    //#[nullptr]
    try
    {
        size_t size = -1; //тъй като е size_t (unsigned),
                           //превърта от положителната страна

        str = new char[size]; //нямаме достатъчно памет
    }
    catch(...)
    {
        throw; //прехвърляме отговорността изключението
               //да се обработи на main()
    }
}

int main()
{
    char* str = nullptr;

    try
    {
        f(str); //въпреки, че [f] може да върне изключение,
                 //това изключение е обработено на по-високо ниво (в main)
                 //и няма да промени състоянието на програмата,
                 //(няма да се терминира)
                 //както и състоянието на [str] (ще си остане nullptr)
    }
    catch (...)
    {
        std::cout << !str << std::endl; //true (str е nullptr)
    }
}
```

strong exception safety

Възможно е да се хвърли грешка, но дори да се случи обектът няма да промени състоянието си. Пример за такава функция е функцията `push_back` от `std::vector`. При добавяне на 10 елемента проваленото добавяне на единадесети няма да повлияе на добавените елементи.

Basic exception guarantee (Основна гаранция за безопасност при изключение)

Ако функцията хвърли изключение програмата е във валидно състояние. Няма отечки на памет и всички инварианти на обектите са изпълнени. За разлика от **Strong exception safety**, тук не е гарантирано, че обекта няма да бъде променен, след хвърлянето на изключение

```
void f(char*& str)
{
    //ако [new] хвърли изключение,
    //то [str] винаги ще се промени
    try
    {
        size_t size = -1; //тъй като е size_t (unsigned),
                           //превърта от положителната страна

        str = new char[size]; //нямаме достатъчно памет
    }
    catch(...)
    {
        str = new char[5] {"asd"};
        throw; //прехвърляме отговорността изключението
               //да се обработи на main()
    }
}

int main()
{
    char* str = nullptr;

    try
    {
        f(str);
    }
    catch (...)
    {
        std::cout << !str << std::endl; //false, [str] е променен от f(),
                                         //тоест при хвърлено изключение,
                                         //функцията НЕ гарантира,
                                         //че [str] няма да бъде променен => basic
    }

    return 0;
}
```

Basic exception guarantee

Ако функцията хвърли изключение програмата е във валидно състояние. Няма отечки на памет и всички инварианти на обектите са изпълнени.

No exception guarantee (Няма гаранция за безопасност при изключения)

Не обещаваме нищо - ако програмата хвърли грешка можем да не сме във валидно състояние.

```
#include <iostream>

void f(char*& str)
{
    str = new char[5] {"ads"};

    size_t size = -1; //тъй като е size_t (unsigned),
                      //превърта от положителната страна

    char* strTwo = new char[size]; //нямаме достатъчно памет
}

int main()
{
    char* str = nullptr;
    f(str);

    //ще се хвърли изключение от [strTwo],
    //което не отработваме и програмата ще се терминира

    return 0;
}
```

```
#include <iostream>

void f(char*& str, size_t size)
{
    str = new char[size] {};
}

int main()
{
    //не обещаваме нищо за f()
    //можем да сме във валидно състояние,
    //можем и да не сме
    char* str = nullptr;

    f(str, 312); //валидно състояние
    f(str, -1); //невалидно състояние (хвърля се изключение)

    return 0;
}
```

No exception guarantee

Не обещаваме нищо - ако програмата хвърли грешка можем да не сме във валидно състояние.

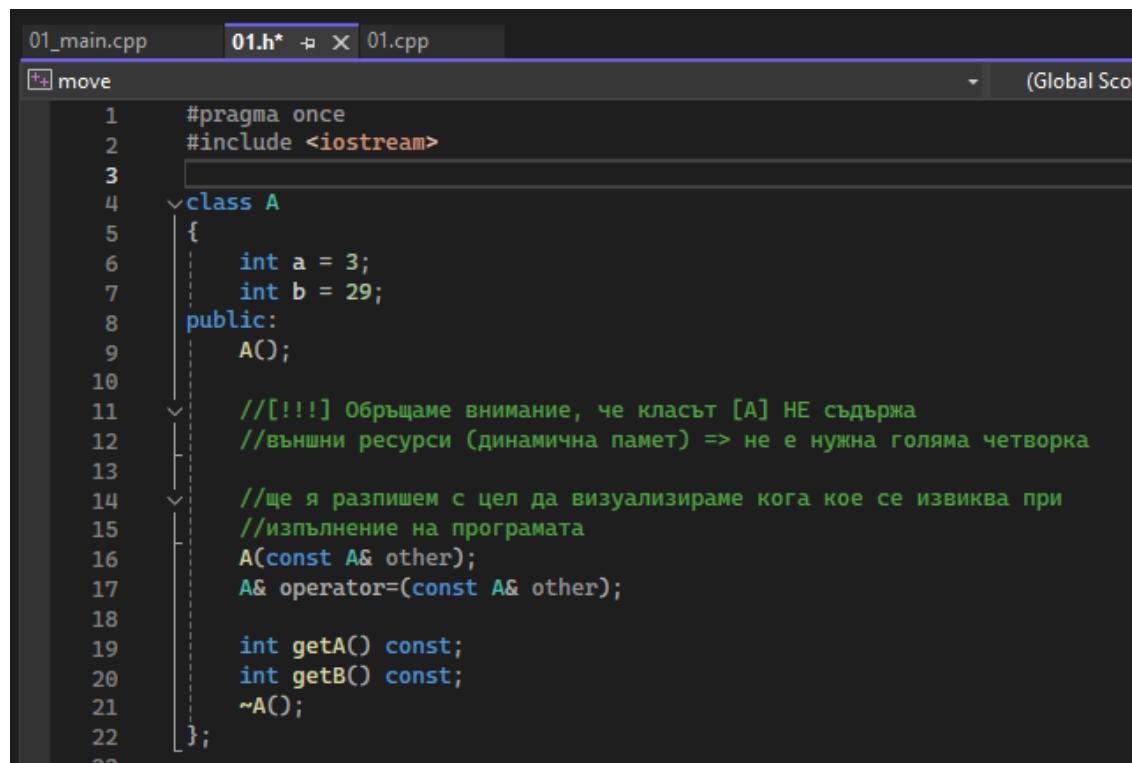
Тема 09. Масиви от указатели към обекти. Move семантики.

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: 01_main.cpp, 01.h*, and 01.cpp. The 01.h* tab is active, displaying 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 a note in lines 11-12 about not containing external resources and therefore not needing a large stack frame. The code editor interface includes a sidebar labeled "move" and a status bar indicating "(Global Scope)".

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     }
128
```

-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
143      //в случая извикването на деструктора не е нужно,
144      //тъй като A() не съдържа член-данни с динамична памет,
145      //ако имаше такива, щеше да е нужно да освободим паметта,
146      //преди да махнем елемента от масива
147
148      //![!] arr[idx].~A();
149      this->size--;
150
151      //тъй като премахваме елемент, е необходимо да запълним
152      //образувалата се празна позиция, като преместим всички елементи
153      //с една позиция наляво
154      for (size_t i = idx; i < this->size - 1; i++)
155      {
156          arr[idx] = arr[idx + 1];
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 пойнърът           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)
//ще 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 по-търпа [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=
 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.е., 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];
    }
}

```

```

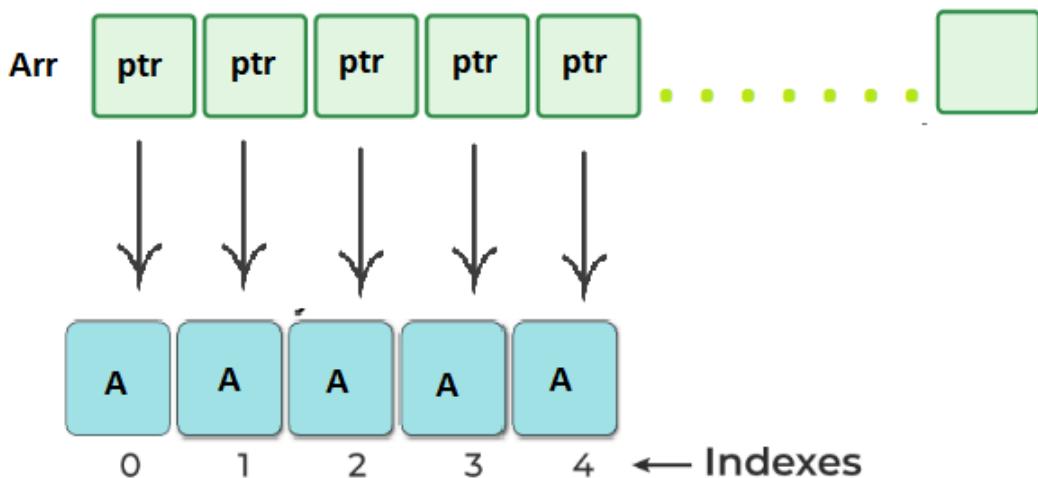
//с една позиция наляво
for (size_t i = idx; i < this->size - 1; i++)
{
    arr[idx] = arr[idx + 1];
}

```

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 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

    //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;
}
```

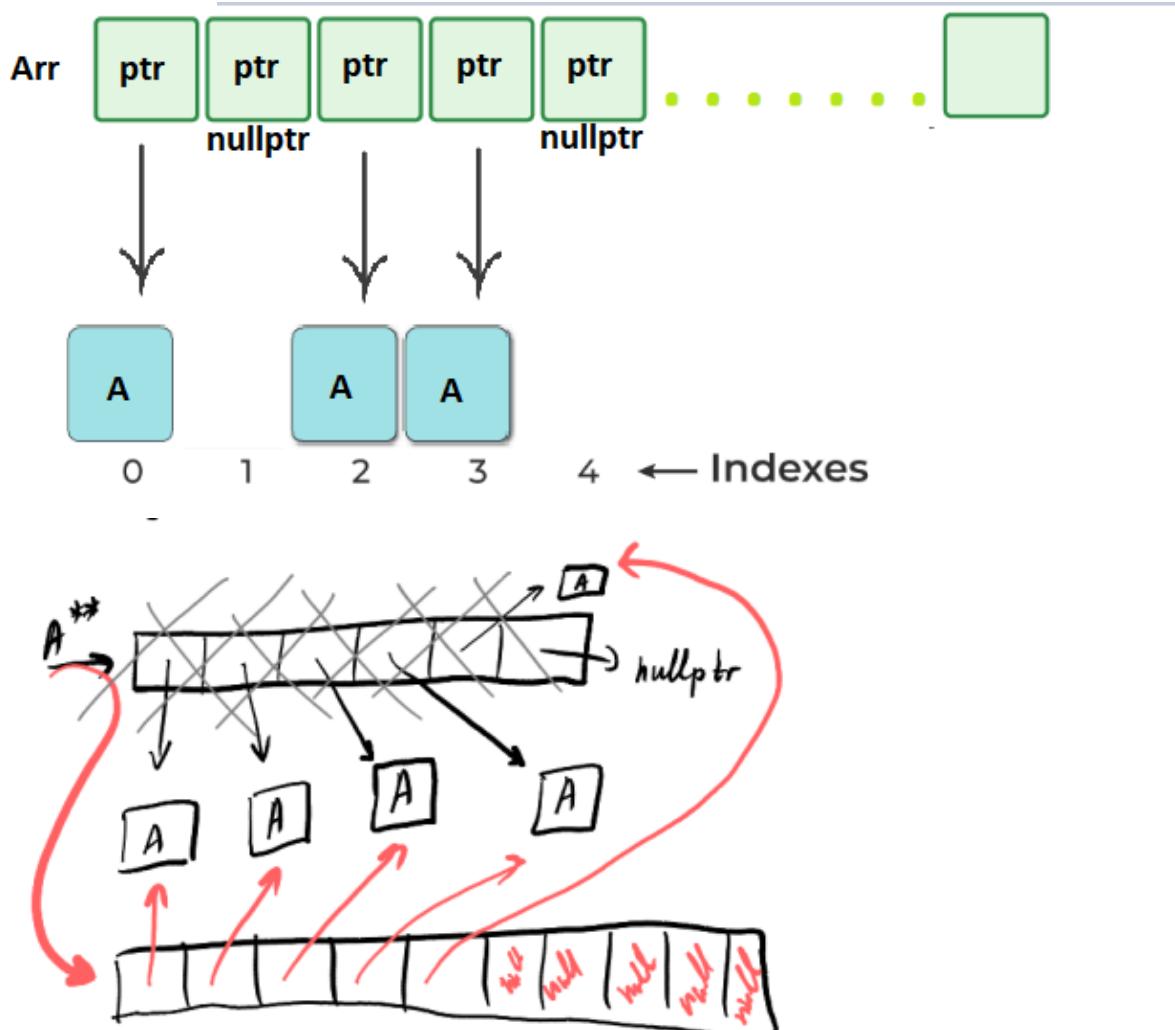
-swap

```
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-долната [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 (чиито 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
              //(заштото 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

move

```
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.e
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.е.
                                //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.е. 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 изисква 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 отпечатат, 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 rvalue
    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:

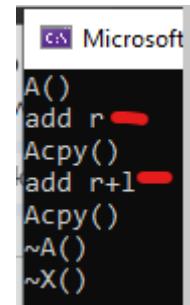
- `add(A&&)` -> очаква `rvalue`
- `add(const A&)` -> очаква `rvalue` ИЛИ `lvalue`

```
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;
```

Тема 10. Наследяване

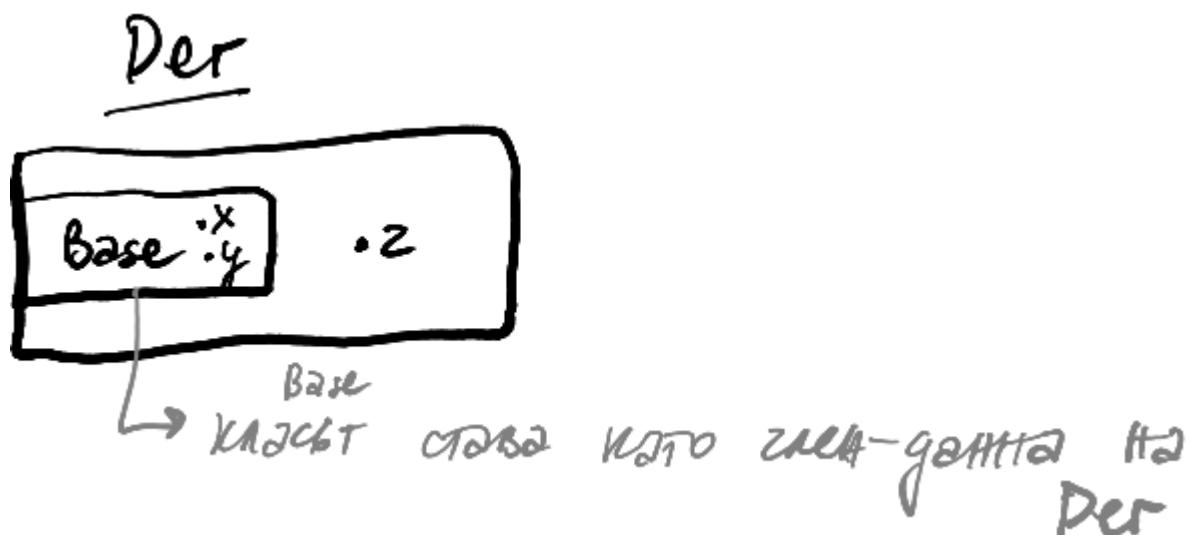
def **Der** е наследник на **Base**, ако разширява неговите данни/поведение

Увод

```
01.cpp* ✘ X
++ inheritance
1 #include <iostream>
2
3 //това ще е класът, който ще използваме
4 //за наследяване
5 class Base
6 {
7 public:
8     int x;
9     void f()
10    {
11        std::cout << "f()" << std::endl;
12    }
13
14 private:
15     int y;
16     void g()
17    {
18        std::cout << "g()" << std::endl;
19    }
20
21 };
22
23 
```

```
23
24 //синтаксис за наследяване:
25 //class <име на наследника> : <начин на наследяване> <име на наследения клас>
26 class Der : public Base
27 {
28 public:
29     int z;
30 };
31
32 int main()
33 {
34     //наследяването ни позволява да използваме
35     //член-данни/член-функции на друг клас + още наши имплементации
36     Der obj;
37     obj.z++;
38     obj.x++;
39     obj.f();
40
41     return 0;
42 }
```

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



Композиция vs наследяване

```
4 //синтаксис за наследяване:
5 //class <име на наследника>: <начин на наследяване> <име на наследения клас>
6 class Der : public Base
7 {
8     public:
9         int z;
10    };
11
12 //ще направим нов клас, който е композиция на
13 ////[DerTest] и [Base]
14 class DerTest
15 {
16     public:
17         Base b;
18         int z;
19    };
20
```

```

42     int main()
43     {
44         // [Der] и [DerTest]
45         // са идентични относно функционалност
46         Der obj;
47         obj.z++;
48         obj.x++;
49         obj.f();
50
51         DerTest objTwo;
52         objTwo.z++;
53         objTwo.b.x++;
54         objTwo.b.f();
55
56         return 0;
57     }

```

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

Композиция - has a relationship
Наследяване - is a relationship } извънне от
 логическа гледна
 точка

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

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

private:
    int y;
};
```

```
19  class Der : public Base
20  {
21  public:
22      int x;
23      void g()
24      {
25          f();
26          //y++; //нямаме достъп до [y] (след малко ще видим защо)

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

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

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

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

37          //стойностите, след Base::x--;
38          std::cout << "[Der] -> [x]: " << x << std::endl;
39          std::cout << "[Der] -> [Base] -> [x]: " << Base::x << std::endl;
40          std::cout << std::endl;
41      }

42      //тоест при повтаряне на имена на променливи/функции при наследяване,
43      //тези на наследникът "shadow-ват" тези на наследения клас
44  };
45
46
47  //тоест при повтаряне на имена на променливи/функции при наследяване,
48  //тези на наследникът "shadow-ват" тези на наследения клас
49  };
50
```

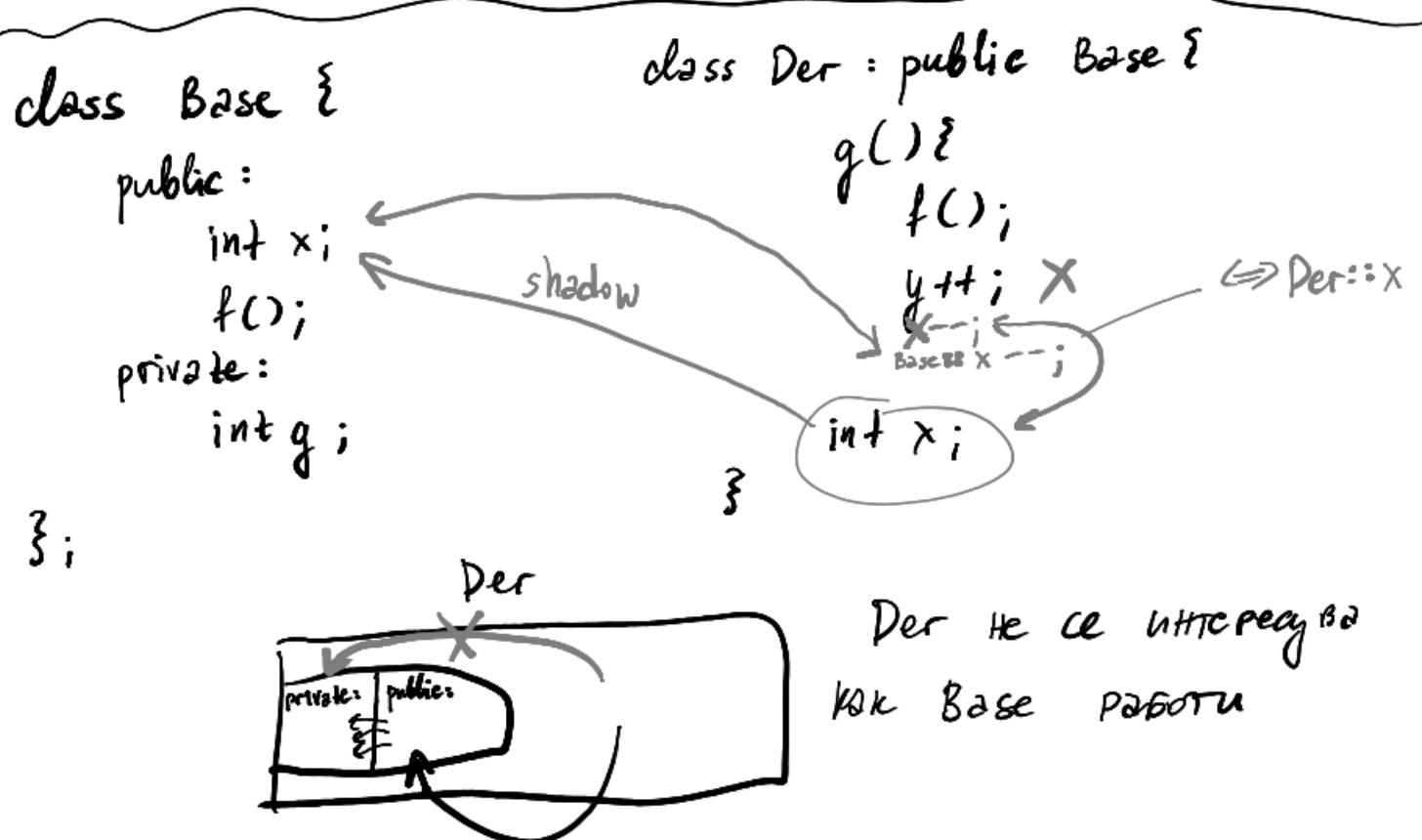
```

51
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.



Видове наследяване

Модификатори за достъп

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

```
3  class X
4  {
5      private:
6          int a;
7
8      //модификаторът [protected] се използва при наследяване
9      //и означава, че наследниците могат да достъпят [b], но [b]
10     //не може да бъде достъпено от външния свят, т.е.
11     //извън наследения и наследяващите класове
12     protected:
13         int b;
14
15     public:
16         int c;
17 }
```

```
//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] -> protected (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-данныте 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-данныте стават private при private наследяване)

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

//private наследяването ни дава извода, че:
class D : public C
{
    //от разгледания случай с член-днната [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

```
//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;
};
```

```
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-данините на [Base]
    ptr->x++;
    ptr->y++;

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

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

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

```
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;
}
```

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

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

    //![No!] обратното НЕ Е възможно,
    //с други думи нямаме casting нагоре, т.е.
    //преобразуване от наследник към наследяван клас

    f(ptr); //![ERROR]

    return 0;
}
```

```

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
}

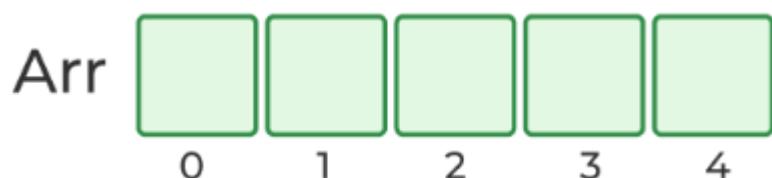
int main()
{
    Der1 arr[3];
    f(arr, 3);

    return 0;
}

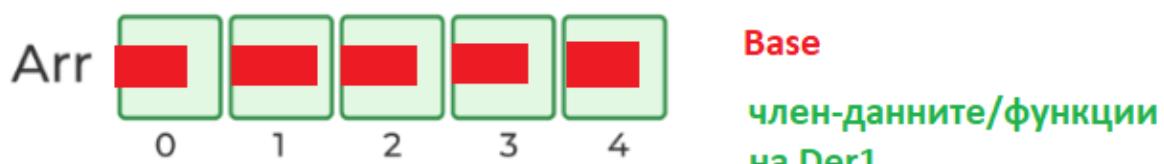
```



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



а има този вид:



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

```
//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;
    }
};
```

```
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;
    }
};
```

```
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;
}
```

```
[der]
Base()
Der()

[derTwo]
Base(x)
Der(x, y)

~Der()
~Base()
~Der()
~Base()
```

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

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

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

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

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

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

```

//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;
    }
};

```

```

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 свикали, 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;
}

```

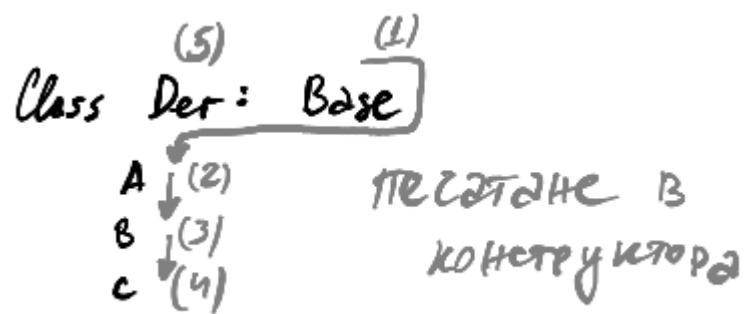
Microsoft Visual Studio Debug Console

```

[der]
Base()
A()
B()
C()
Der()

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

```



Bonus:

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

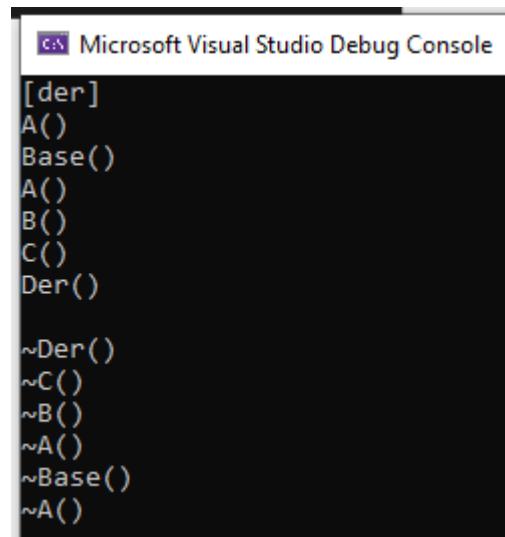
    Base(int x)
    {
        std::cout << "Base(x)" << std::endl;
        this->x = x;
    }

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

```
int main()
{
    //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;
}
```



The screenshot shows the Microsoft Visual Studio Debug Console window. The title bar says "Microsoft Visual Studio Debug Console". The console output is as follows:

```
[der]
A()
Base()
A()
B()
C()
Der()

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

Деструктори при наследяване

- деструкторът на наследника извиква деструктора на базовия клас

```
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 декларириани, т.е.
    //ако 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;
}
```

$\sim\text{Der}()$ {
 cout << $\sim\text{Der}()$;
}
 $\sim\text{C}$ $\sim\text{B}$ $\sim\text{A}$ $\sim\text{Base}$

\Rightarrow Der
 |
 C
 |
 B
 |
 A
 |
 Base

Microsoft Visual Studio Debug Console

```
[der]
Base()
A()
B()
C()
Der()

 $\sim\text{Der}()$ 
 $\sim\text{C}()$ 
 $\sim\text{B}()$ 
 $\sim\text{A}()$ 
 $\sim\text{Base}()$ 
```

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-данините/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 copy(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;
}
```

```
[der]
Base()
A()
B()
C()
Der()

[derTwo] COPY
Base copy(other)
A()
B()
C()
Der copy(other)
A operator=( )
B operator=( )
C operator=( )
```

```
[derThree]
Base()
A()
B()
C()
Der()

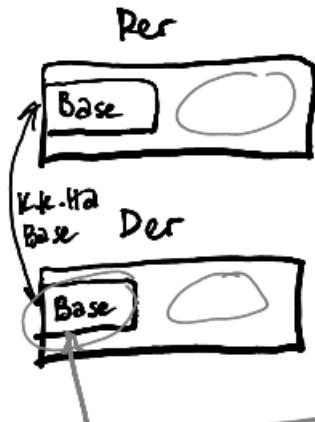
[derThree] OPERATOR=
Der operator=(other)
Base operator=(other)
A operator=( )
B operator=( )
C operator=( )
```

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

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)**

• KOTRUPAHE TIPU HACNEGRASHE



Der (const Der& other): Base (other)

copyFrom (other); → TIPU HACNE
como 32 Der

Base (const Base&)

Base (other)

?

BUMA como HYTHHE Base

```
op = (const Der& other)
if (this != &other) {
    free(); → une garantie que der
    copyFrom (other);
    { Base::operator= (other);
    return *this;
}
```

Move при наследяване

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

Der(Der&& other) noexcept: Base(std::move(other)) //отново се грижим само за член-данни/функции
{
    //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;
}
```

• move при наследяване

Der (Der&& other) : Base (std::move (other))

moveFrom (std::move (other));
moveFrom (Der&&)

op = (Der&& other)
if (this != &other) {
 free();
 moveFrom (std::move (other));
 Base :: op = (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-a**)

```
1 #include <iostream>
2 //ще използваме следните четири класа,
3 //които имат отпечатвания в:
4 //01. default-ния конструктор
5 //02. копиращия конструктор
6 //03. operator=
7 //04. десктруктора им
8
9 > class A{ ... };
10
11 > class B{ ... };
12
13 > class X{ ... };
14
15 > class Y{ ... };
16
17
```

```
//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;
```

```

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-данныте му
    //04. връщаме се в [Der], където се извикват конструкторите на член-данныте 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;

    //тъй като отпечатаването 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 копирання 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=()

```

```
int main()
{
    Der d1;
    std::cout << std::endl;
    Der d2;
    std::cout << std::endl;
    d1 = d2;
    std::cout << std::endl;

    return 0;
} //върху разгледахме викането на деструкторите
//=> ~Der() ~Y() ~X() ~Base() ~B() ~A()
//01. вика се конструктора на [Der]
//02. викат се деструкторите на член-данныте му една по една
//03. стига до зачистването на въображаемата ни член-дансна [Base]
//04. викат се деструкторите на член-данныте 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()
```

```

int main()
{
    //ако изместим отпечатването на operator= 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;
}

```

```

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

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

```

```

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

OPERATORE B HAZARDO (B træ. Ha if)

OPERATORE B KPAR (creg if)

Base

A obj1

B obj2

Der

X obj1

Y obj2

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

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

{ Der d2; [konstruktor]

Der d2; [kommunikation]

d1 = d2; OP = Der, OP = Base, OP = A, OP = B, OP = X, OP = Y
(B HAZARDO negation)

{ 2x geschrieben } (B KPAR negation)

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

+ ouje & copy2par

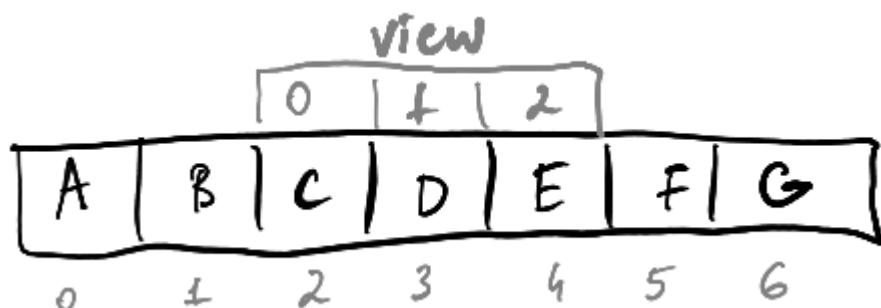
no getp:
base = op =
free
copyFrom

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
#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

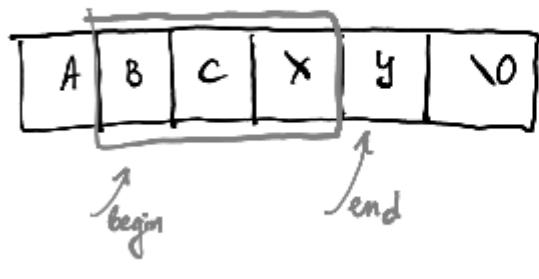
Има два варианта за член-данные на **StringView** 01

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` → $(\text{end} - \text{start}) / \text{sizeof}(T)$ → ~~ако не e char~~
с размер ~~1 byte~~
 - `два конструктора`
 - `operator [] (size_t index) const`

```
{ return -begin[index]; }
```
 - `operator <<`
- с този клас нямаме 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 компиляция 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;

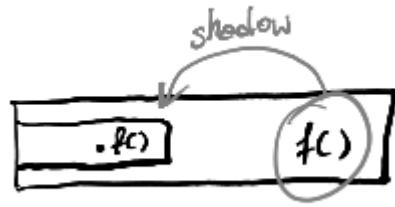
    //в случая, тъй като [Der] наследява [Base] публично,
    //то де факто имаме две едни и същи функции с еднакво име,
    //като имаме достъп и до двете

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

    //насочваме [Base] поинтър 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

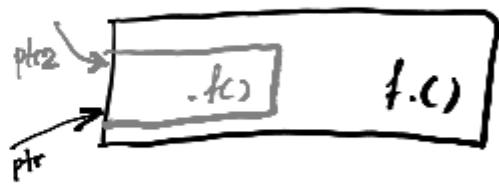
Предните примери са примери и за статично свързване

```

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 пойнтьра [ptr] към [Der], тъй като
    // [ptr] е от тип [Base]

    //макар и да насочваме пойнтьра [ptr] към [Der],
    //той взима [Base] частта, тоест имаме пойнтьр, 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 пойнтьра/референцията,
    //компилаторът 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 пойнтър от тип [Base]
                    //![d] е от тип [Der]

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

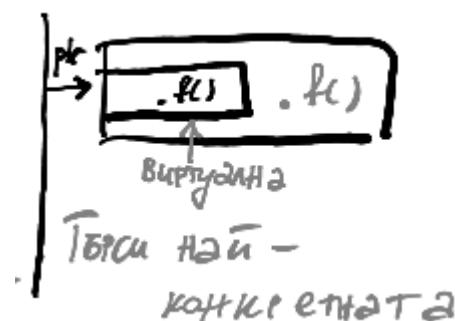
    ptr->f(); //повтаряме: въпреки, че пойнтьра е от тип [Base],
                //компилаторът ще види, че той сочи към [Der] =>
                //ще се извика функцията f() на [Der]

    return 0;
}
```

Microsoft Visual St

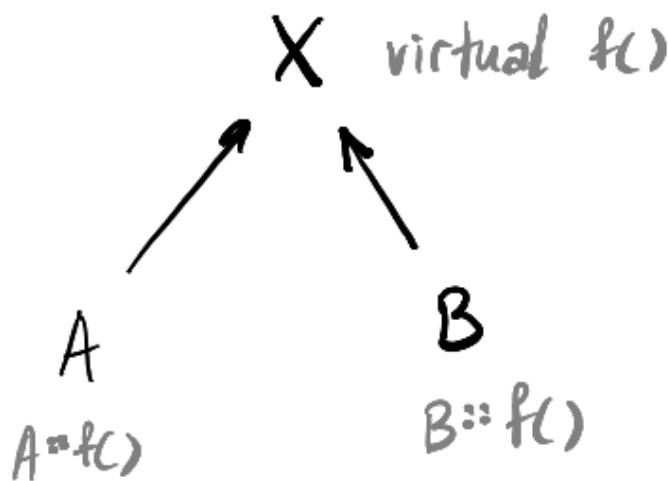
```
Der::f()
Der::f()
```

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;
}
```

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

? A::f , B::f , (X::f - и то и да е един и същ клас напр.)

```
ymornism                                         (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 презаписвам виртуална функция
17                         //при грешка в синтаксиса дава компилационна грешка
18     {
19         std::cout << "Der::f()" << std::endl;
20     }
21 };
22
23 int main()
24 {
25     return 0;
26 }
```

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 пойнтьра,
    //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]
    //=> С също е абстрактен клас
};
```

```
#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 иерархията
- 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;
    }

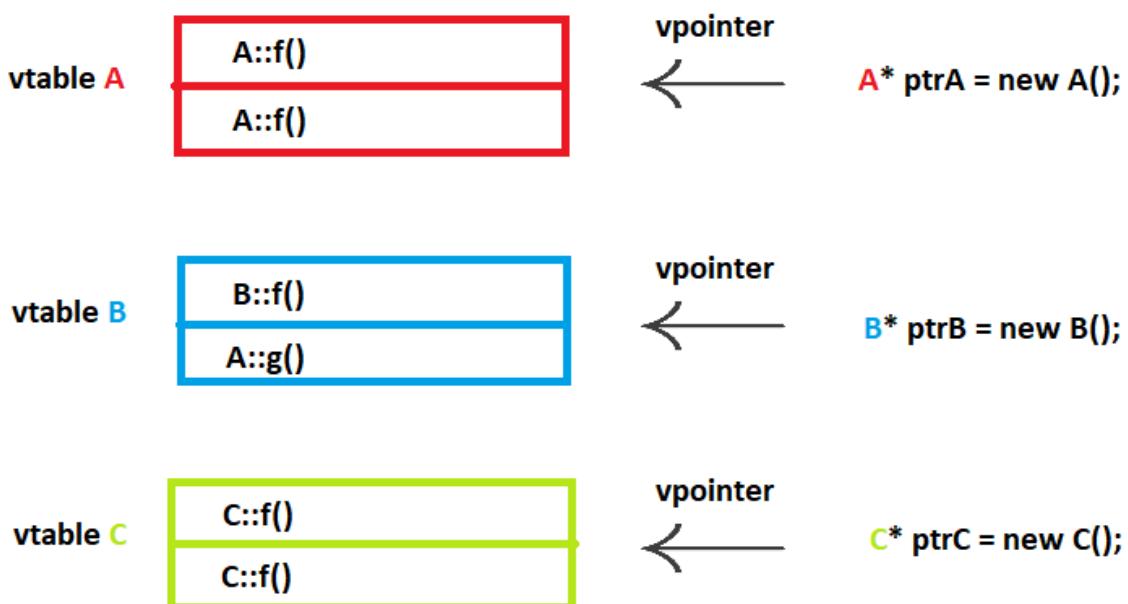
    void g() override
    {
        std::cout << "C::g()" << std::endl;
    }
};
```

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 пойнтър, 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 пойнтър, 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 пойнтъри, 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



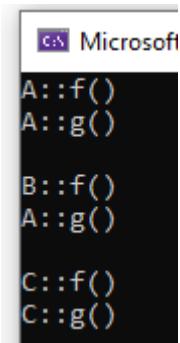
```
int main()
{
    A* ptrA = new A();
    ptrA->f();
    ptrA->g();
    std::cout << std::endl;

    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;
}
```



Тема 12. Множествено наследяване

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

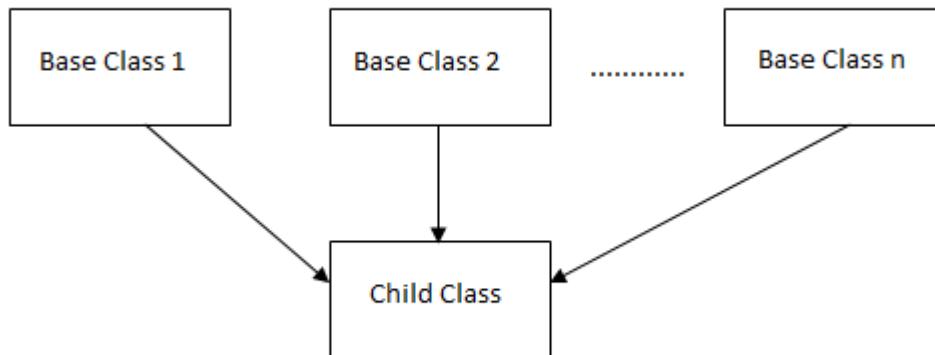
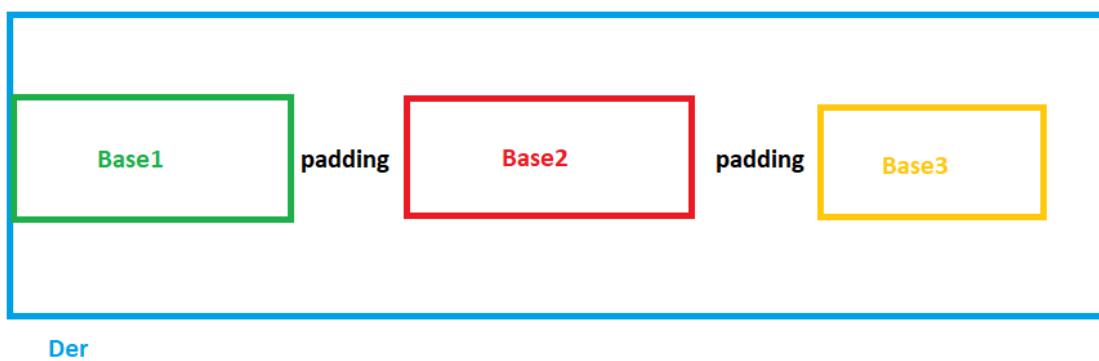


Fig: Multiple Inheritance

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



Нека имаме следните три класа:

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

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

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

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

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

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

```

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

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

```

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

```

int main()
{
    Der d;

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

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

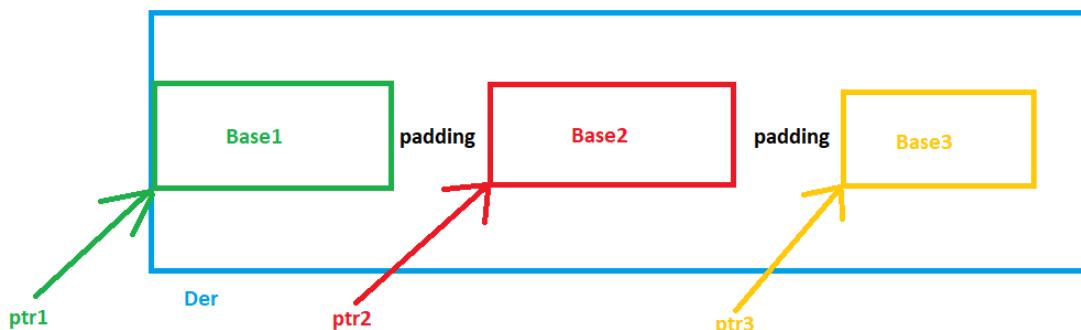
    return 0;
}

```

```

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

```



Нека **Der** има следния вид

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

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

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

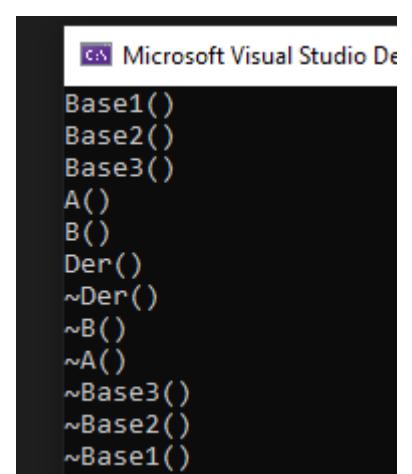
```
};

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

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

ред на извикване
на конструктори

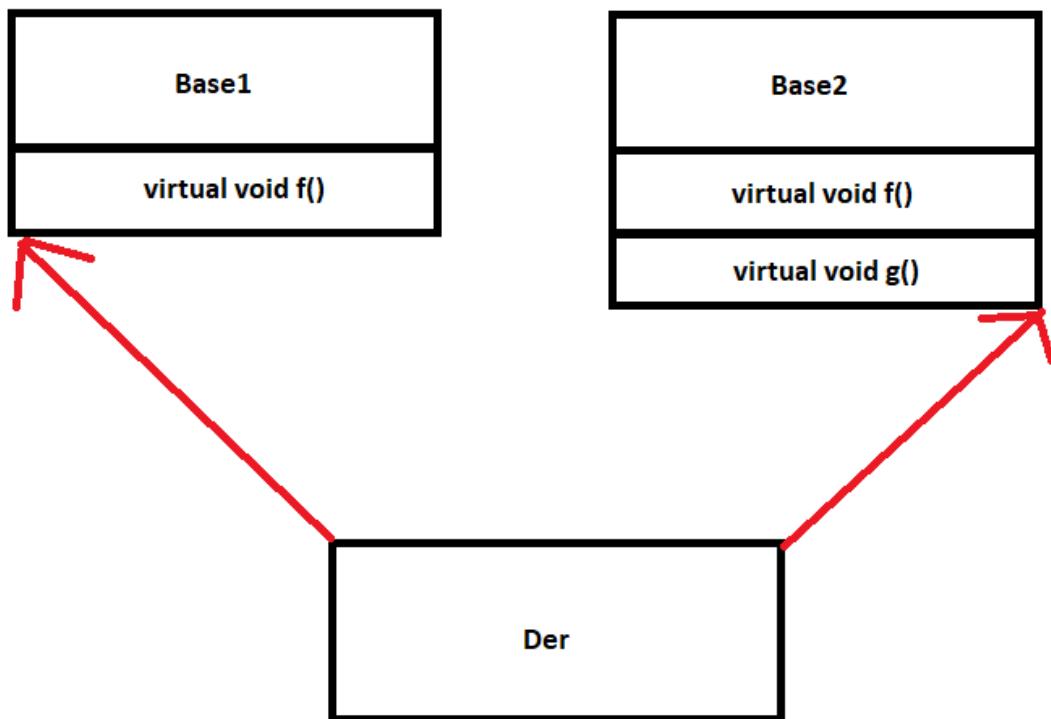
ред на извикване
на деструктори



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

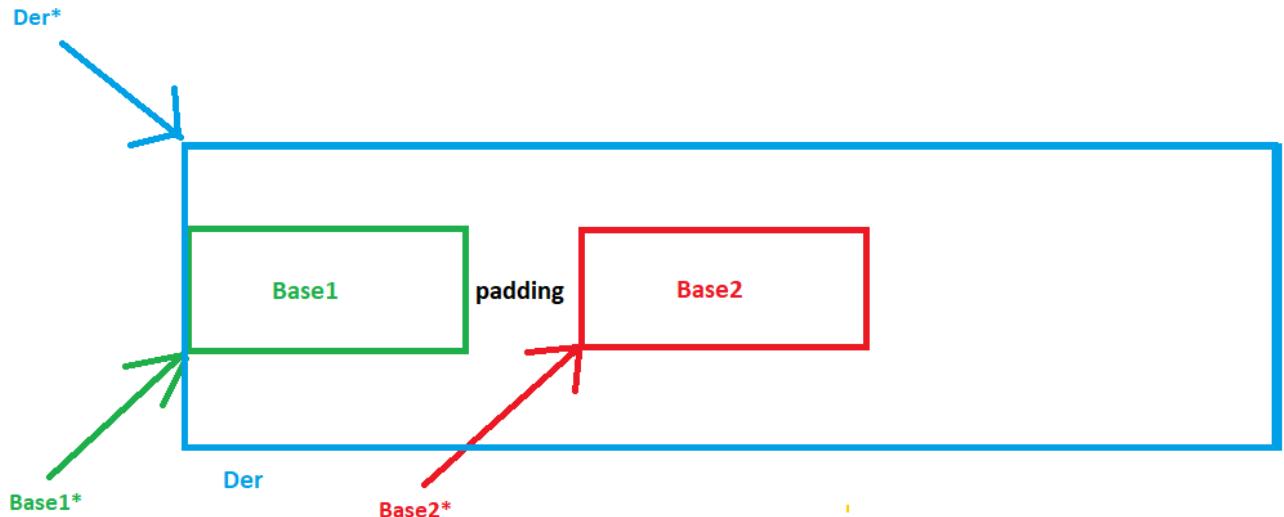
```
Der& Der::operator=(const Der& other)
{
    if (this != &other)
    {
        Base1::operator=(other);
        Base2::operator=(other);
        Base3::operator=(other);
        copyFrom(other);
        free();
    }
}
```

Виртуална таблица при множествено наследяване

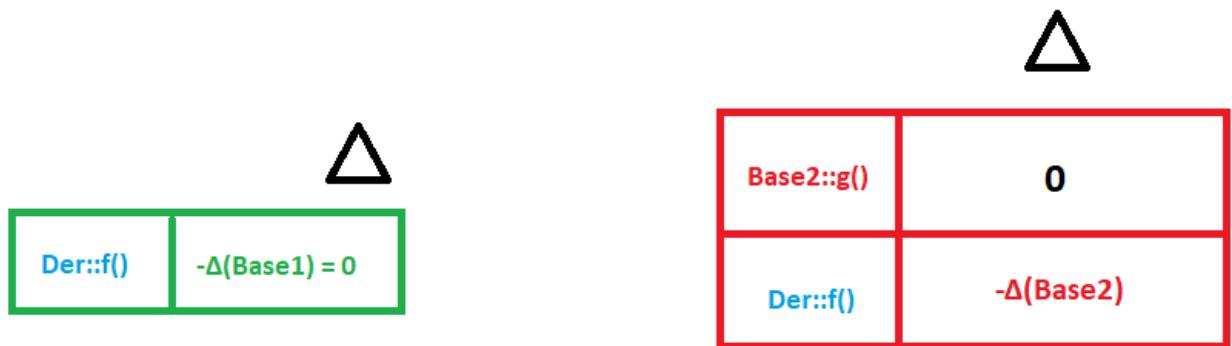


Следната полиморфна йерархия ни задължава да презапишем функцията `f()` в `Der`, тъй като не го направим ще стане двусмислица коя от двете да се извика
 \Rightarrow `Der` задължително трябва да override-ва функцията `f()`

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



Таблиците придобиват следния вид:



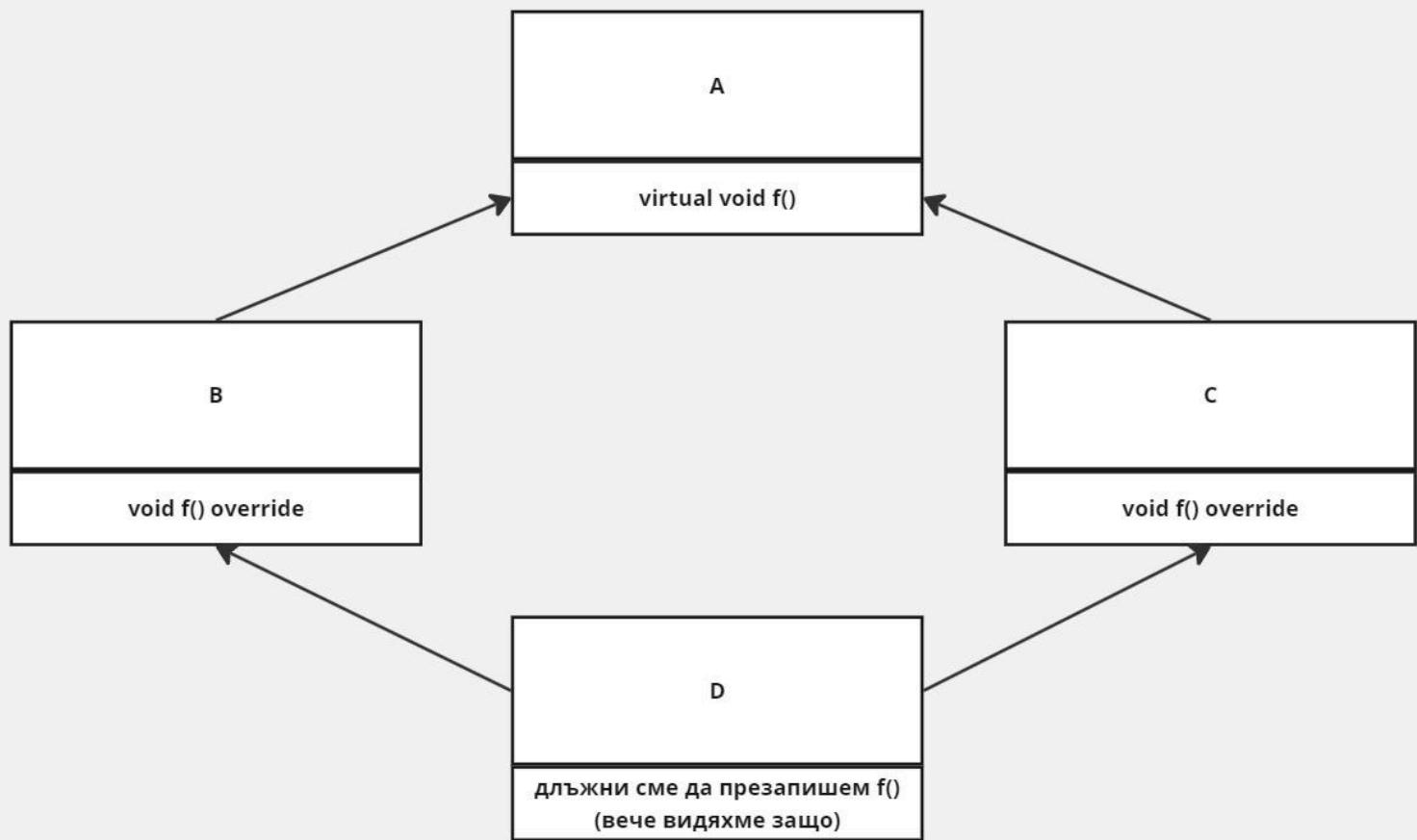
$\Delta(\text{Base2})$ - отместването на `Base2` от началото на `Der`

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

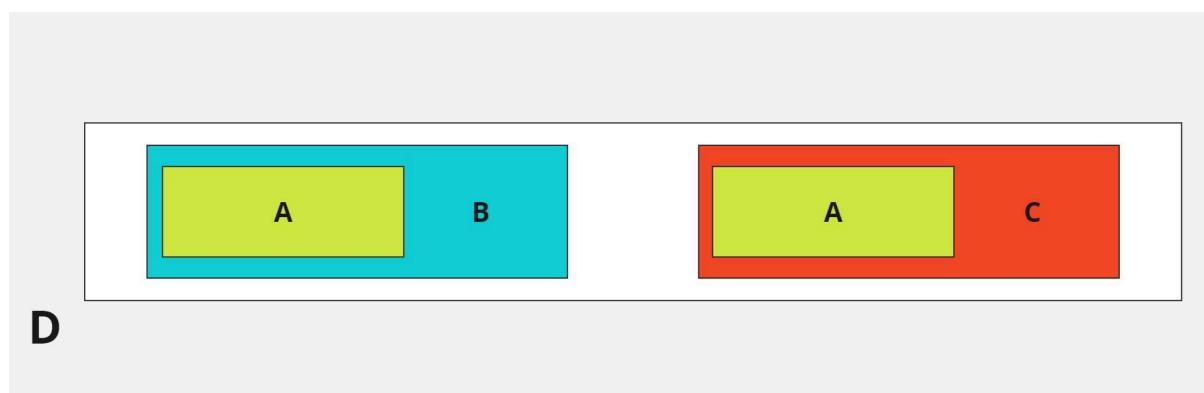
Напр.:

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

Диамантен “проблем”



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



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

Виртуално наследяване

```
∨class X : virtual Y
```

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

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

Нека това е класът, който ще наследяваме виртуално

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

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

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

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

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

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

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

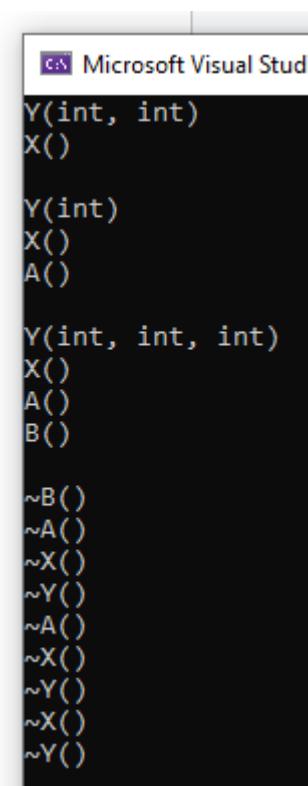
    ~A()
    {
        std::cout << "~A()" << std::endl;
    }
};
```

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

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

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

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



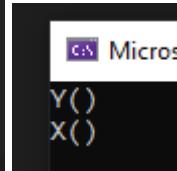
Можем да забележим, че винаги се извика първо конструктора на виртуално наследения клас, след което тези на базовите, независимо дали наследения виртуално клас е в началото, средата или края на йерархията ни (при деструкторите е същото, но в обратен ред, както сме свикнали)

```

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

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

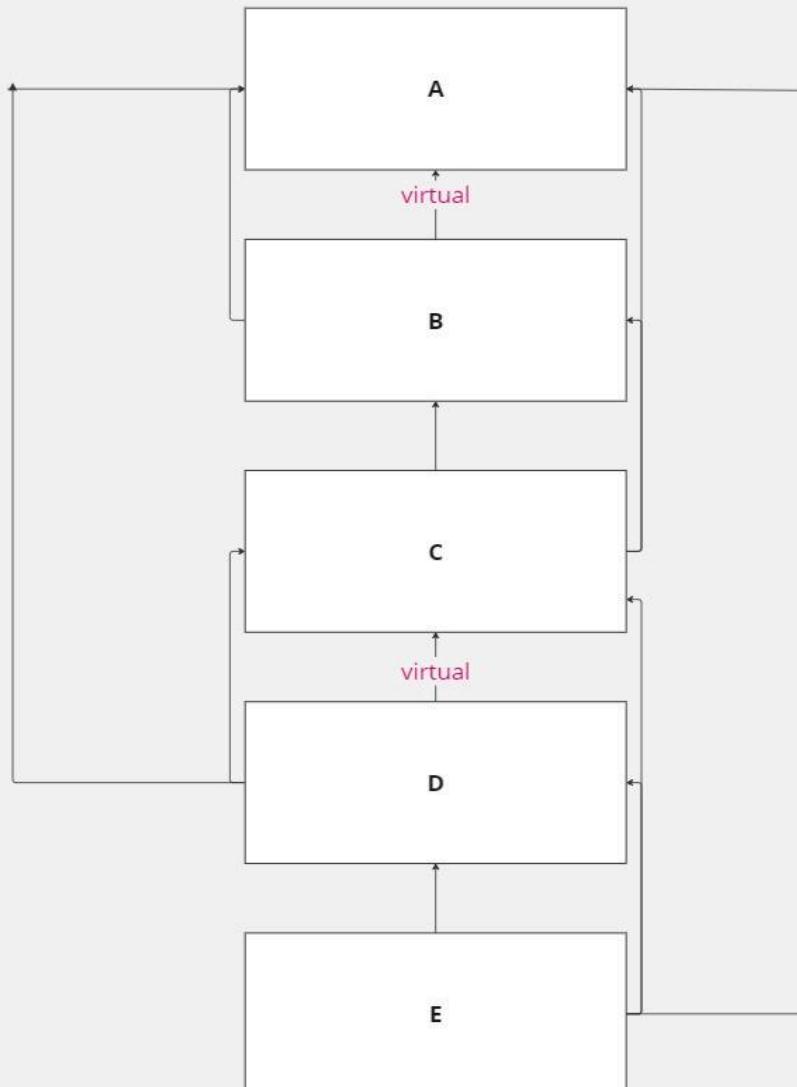
```



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

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

Пример за друга такава йерархия:



В наследява А виртуално и отговаря за създаването на А

С наследява В, който наследява А виртуално, следователно отговорността за създаването на А се прехвърля на С и С отговаря за създаването и на В, и на А

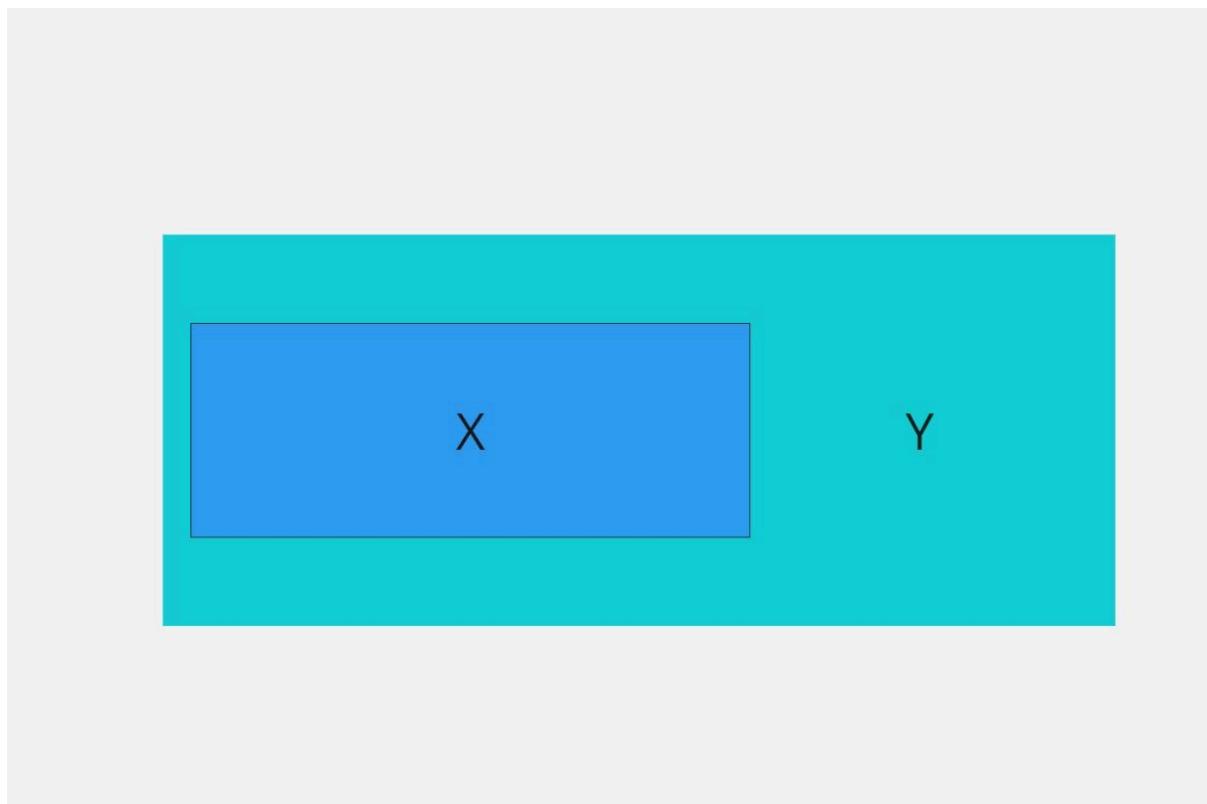
Д наследява виртуално С, но тъй като С е част от виртуално наследяване, то Д освен за С, става отговорен и за А, тъй като Д е част от виртуалното наследяване и на А (virtual)-> B -> C ->D

Е наследява D, но D наследява виртуално С => D е отговорен и за С. Също така Е е част от виртуалното наследяване на А и така Е става отговорен и за D, и за С, и за А

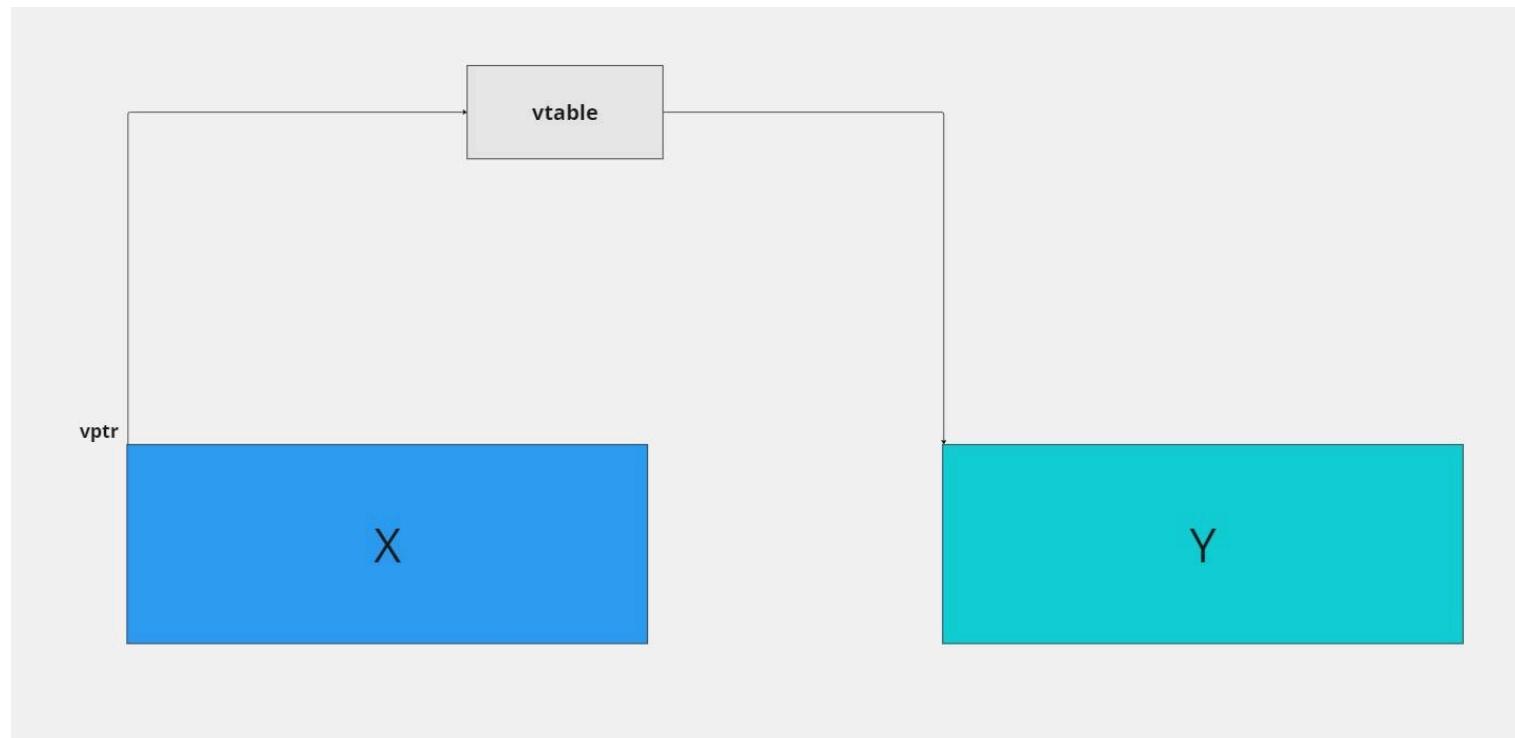
```
project14
1 #include <iostream>
2
3 class X
4 {
5     // ...
6 }
7
8 class Y : virtual X
9 {
10    // ...
11 }
12
13 int main()
14 {
15     Y obj;
16
17     return 0;
18 }
```

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

Стандартното наследяване:

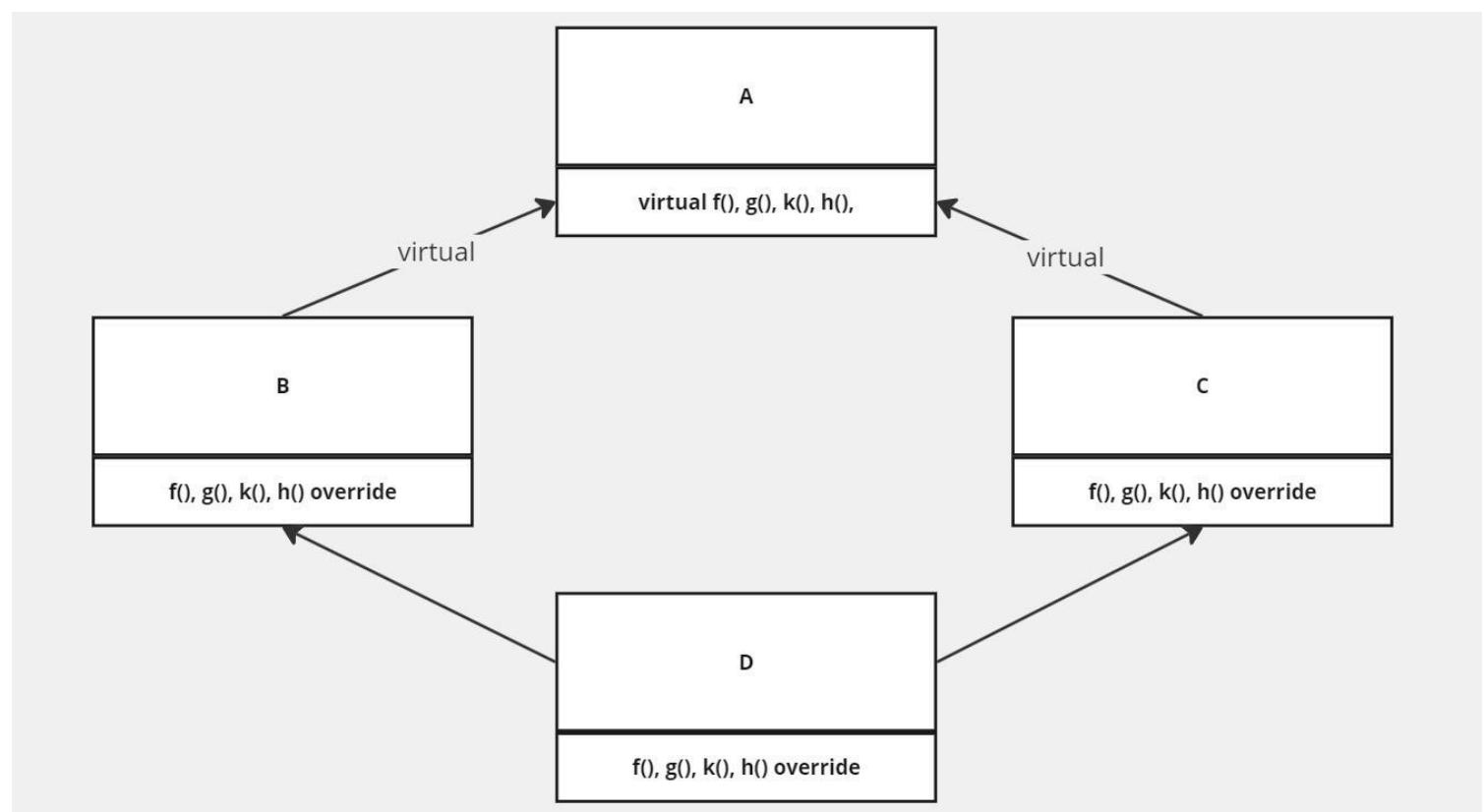


Виртуално наследяване



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

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



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

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

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

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

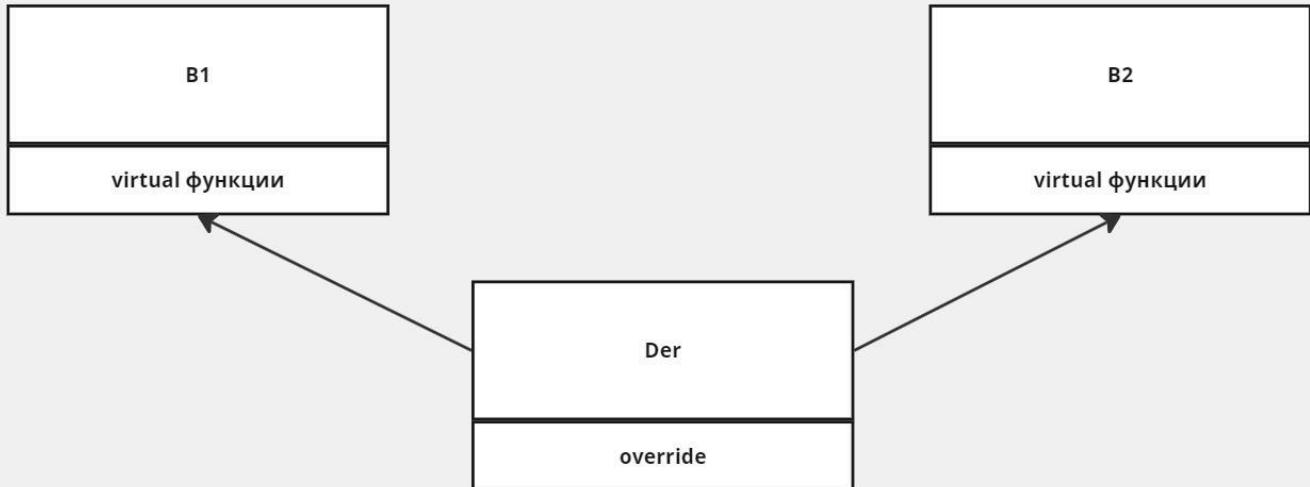
class C : virtual public A
{
public:
    C() : A()
    {
        std::cout << "A()" << std::endl;
    }

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

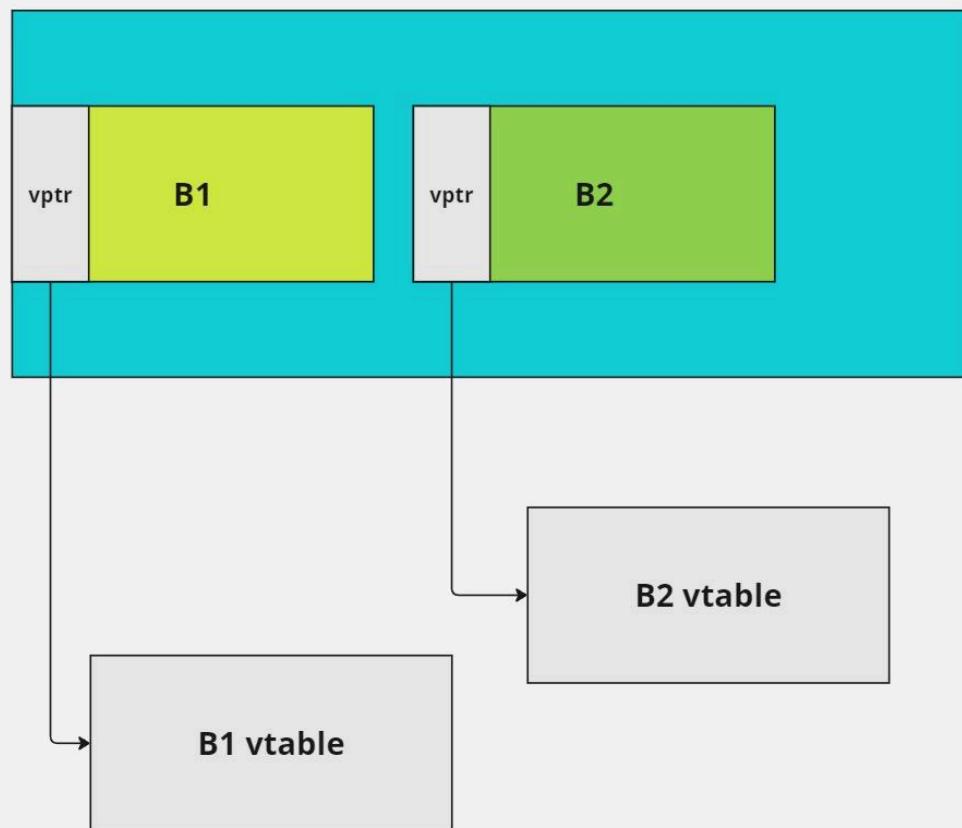
class D : public B, public C
{
public:
    D()
    {
        std::cout << "D()" << std::endl;
    }

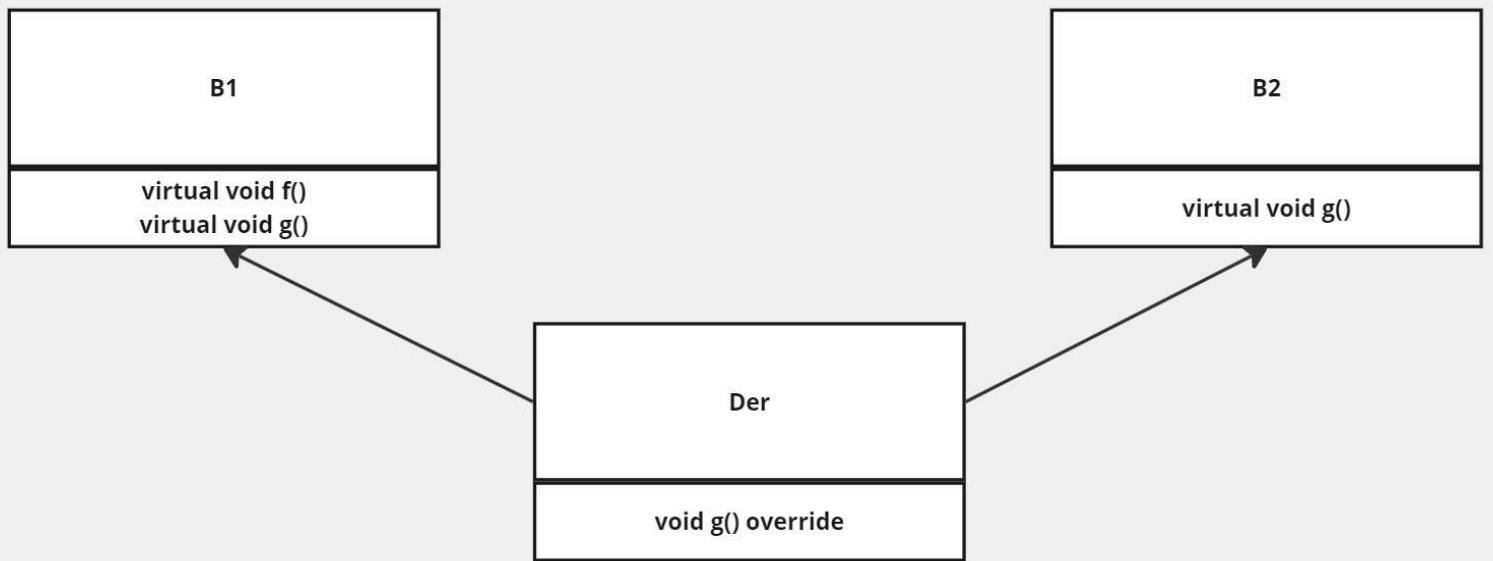
    ~D()
    {
        std::cout << "~D()" << std::endl;
    }
};
```

Още примери

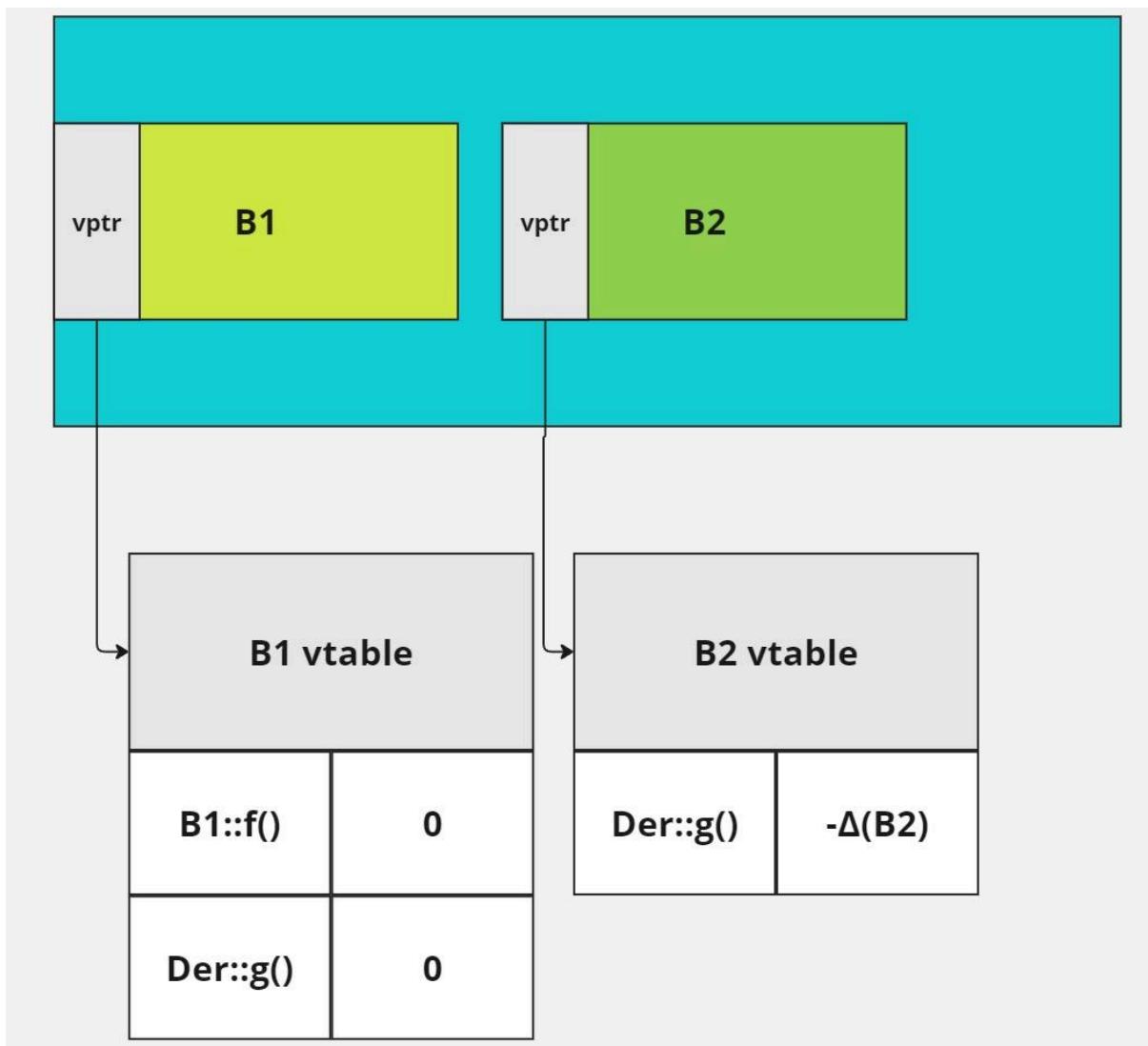


В тази йерархия, **Der** придобива следния вид:

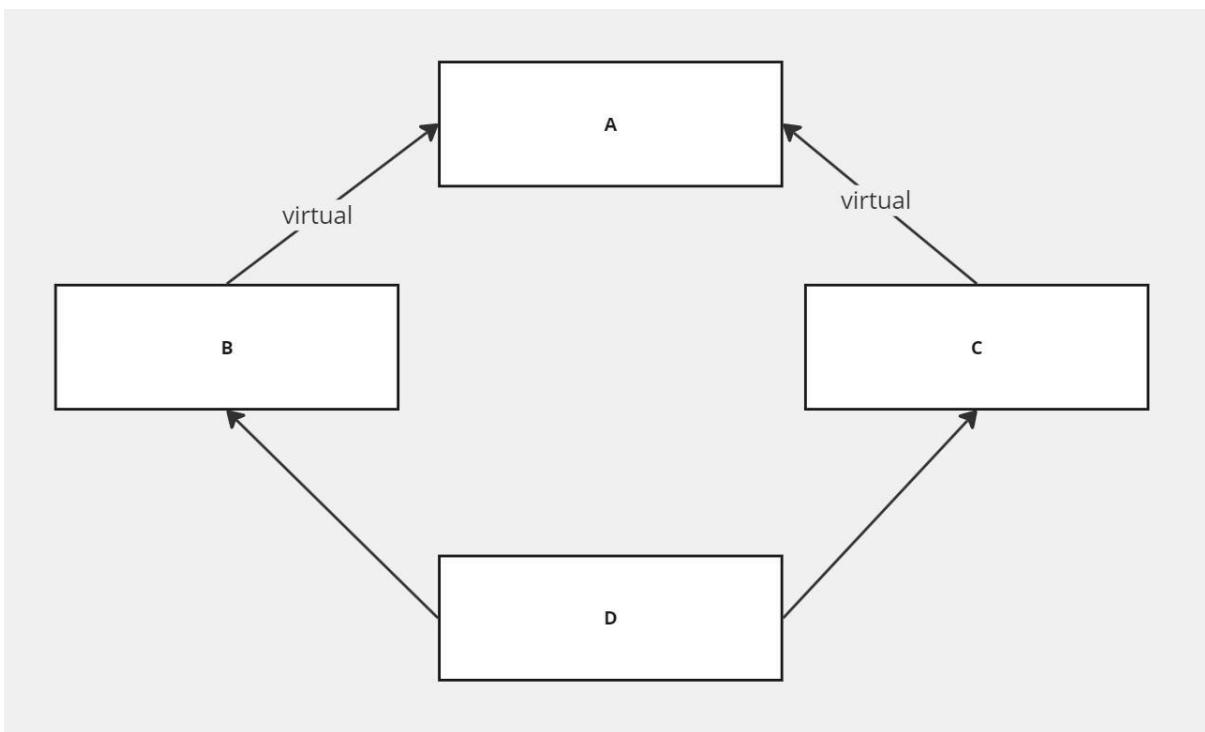




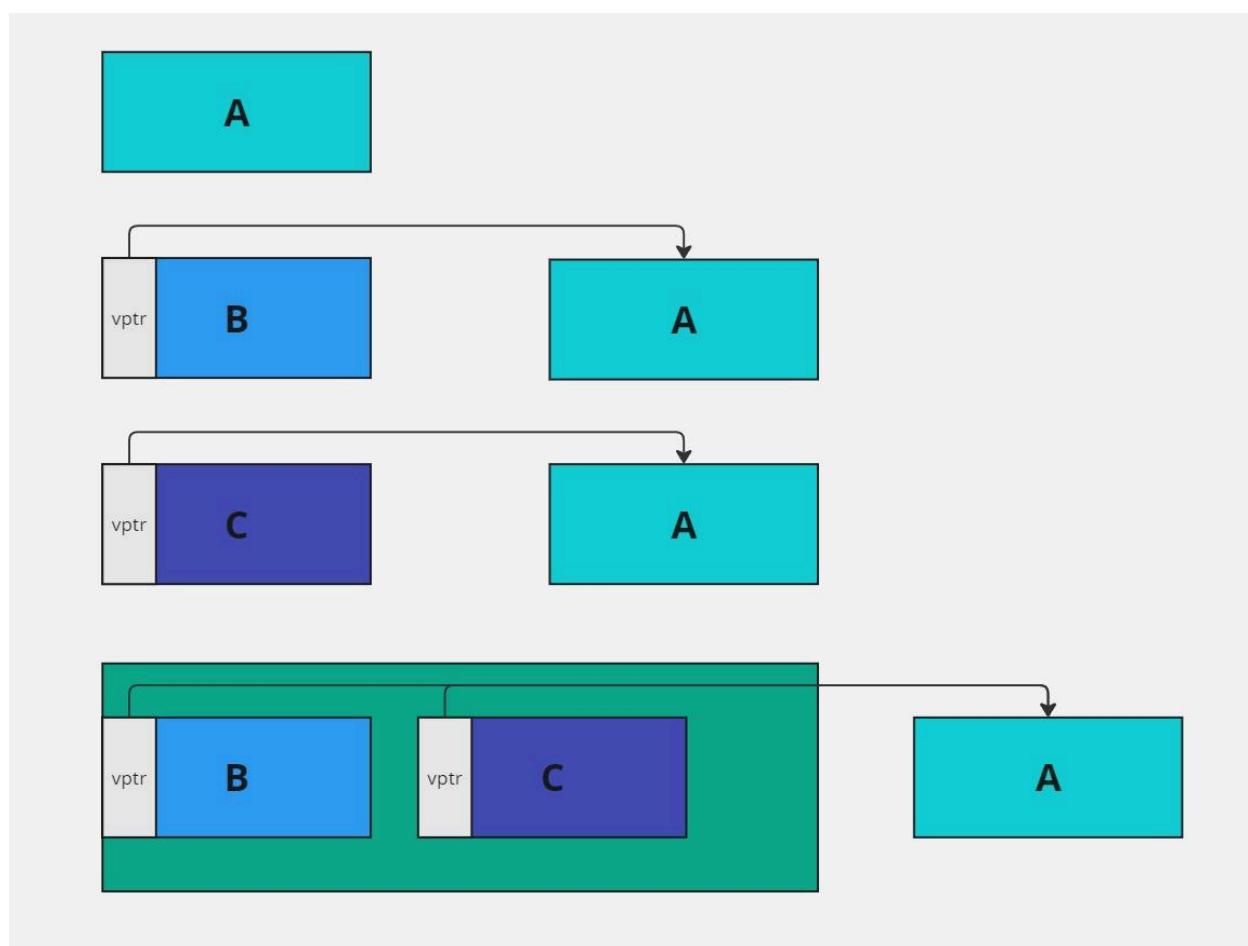
В тази йерархия, **Der** придобива следния вид:

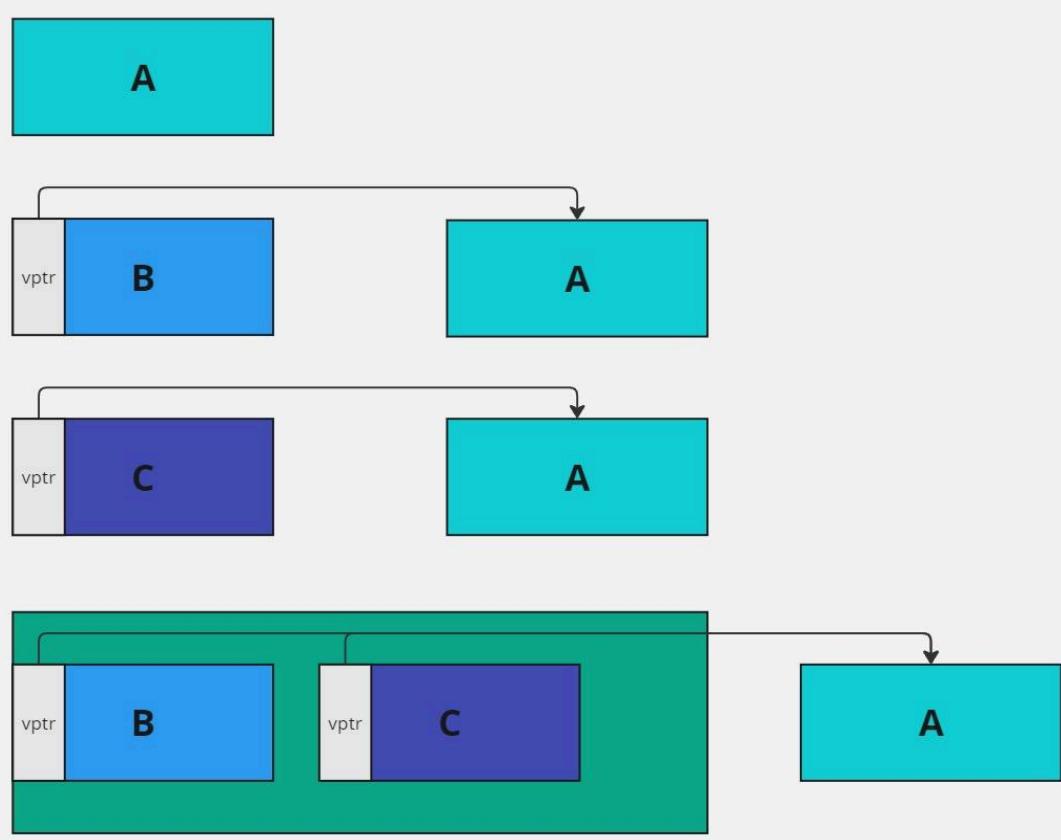
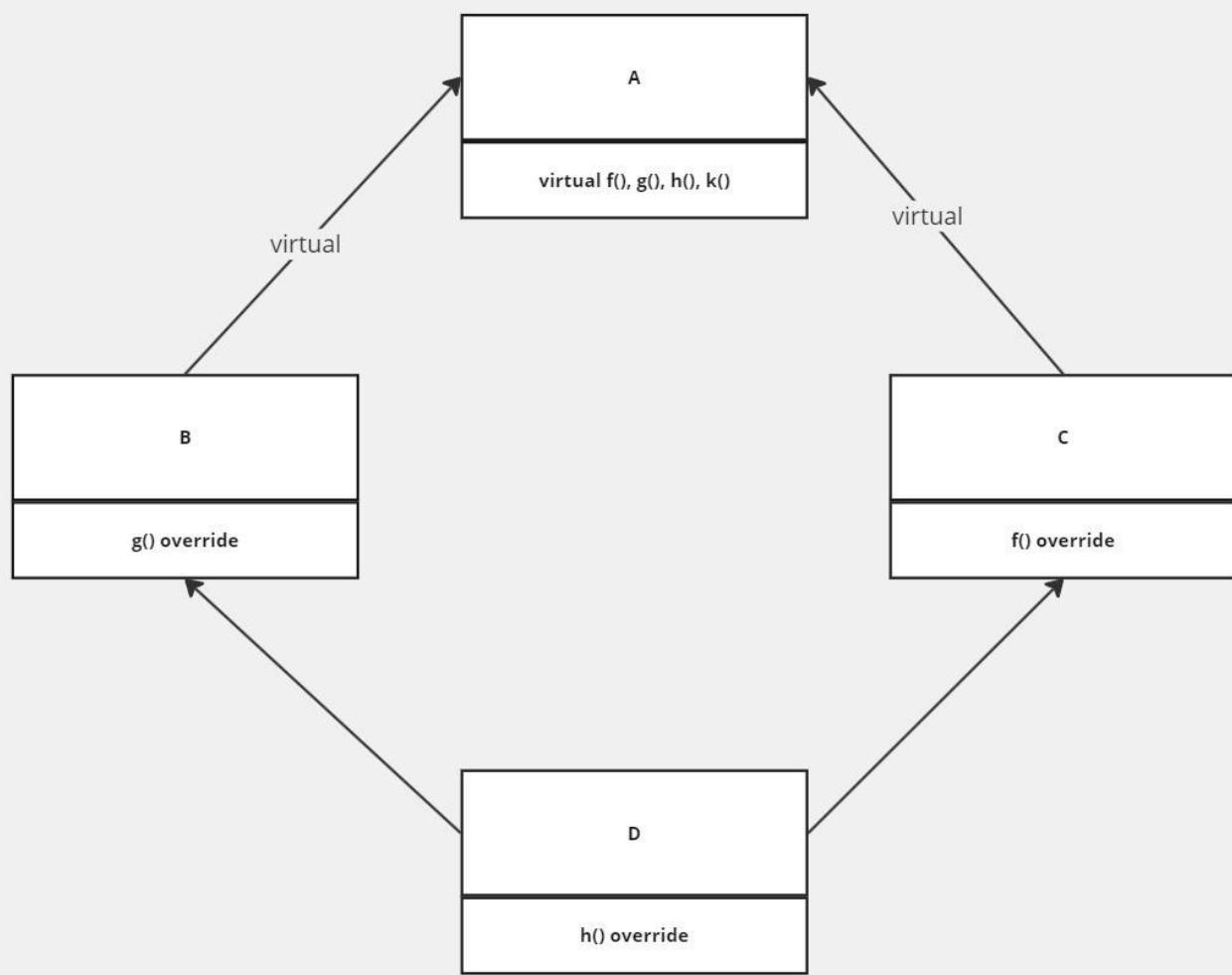


[!] Ако **Der** има виртуална функция, която я няма в **B1** и **B2**, то тя също ще участва във виртуалните таблици



Членовете на тази йерархия ще изглеждат по следния начин:





vtable of D (ако имаме A* ptr)

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

Хетерогенен контейнер

def.| Хетерогенен контейнер

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

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

Копиране

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

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

Ако **A** наследява **Base**, така би изглеждала презаписаната функция **close()** в интерфейса на **A**. Чрез копирация конструктор създаваме ново **A**, което копира данните на ***this**.

Триене

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

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

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

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

def.| Visitor Pattern

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

Тема 13. Шаблони

def| Шаблон

- Клас/функция с общо предназначение спрямо типа ,
(необходима функция на тип)
- шаблон = параметричен полиморфизъм

Със сегашните знания функцията **max()** между числа и стрингове ще реализираме по следния начин:

```
#include <iostream>

int max(int a, int b)
{
    return a > b ? a : b;
}

const std::string& max(const std::string& lhs, const std::string& rhs)
{
    return lhs > rhs ? lhs : rhs;
}

int main()
{
    std::cout << max(3, 9) << std::endl;
    std::cout << max("ABC", "XY") << std::endl;
    return 0;
}
```

Но благодарение на шаблоните, можем да спестим дублирането на код:

```
#include <iostream>

//обозначаване на функция, че е темплейтна
template <typename T> //или template <class T>

//навсякъде типа, който не знаем се заменя с T
const T& max(const T& lhs, const T& rhs)
{
    return lhs < rhs ? rhs : lhs;
}

int main()
{
    std::cout << max<int>(3, 9) << std::endl;
    std::cout << max<std::string>("ABC", "XY") << std::endl;

    //<int> и <std::string> генерират нов код, където T е заменено
    //със съответния тип

    return 0;
}
```

```
#include <iostream>

template <typename T>
const T& max(const T& lhs, const T& rhs)
{
    return lhs < rhs ? rhs : lhs;
}

class A {};

int main()
{
    A obj1, obj2;

    //уловка
    max<A>(obj1, obj2);
    // ^
    // |
    // |
    // тъй като синтаксиса е max [<] A > ...
    // се очаква A да има предефиниран оператор <

    return 0;
}
```

[!] Всички колекции трябва да имат дефиниран оператор <

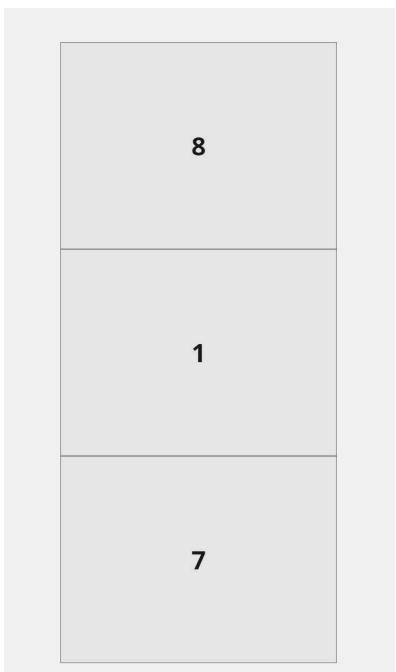
[!] Тъй като шаблонните функции/класове се генерират по време на компилация и типовете за шаблоните трябва да са известни по време на компилация, то можем да кажем, че шаблоните са **параметричен полиморфизъм**

Пример **Stack**

- абстрактна структура от данни
- имаме дефинирано поведение
(знаем какво трябва да стане, но нямаме дефинирана имплементация)

```
int main()
{
    //стек, работещ с int-ове
    Stack<int> s;
    s.push(7);
    s.push(1);
    s.push(8);

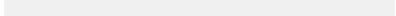
    return 0;
}
```



чрез функцията **push()** добавяме нови елементи в нашия стек

```
    s.pop(); //8
    s.pop(); //1
    s.pop(); //7

    return 0;
}
```



чрез функцията **pop()** премахваме
най-горния елемент в стека ни
(тоест последния добавен)

Искаме максимален капацитет

- капацитетът да е шаблон
- типа на стека също да е шаблон

```
template <typename T, unsigned N>
class Stack
{
    T data[N]; //създаваме масив от тип T с размер N

    //![!] вече сме видяли как се създават масиви
    //и можем да направим извода, че задължаваме T да има
    //default-ен конструктор
};
```

```
int main()
{
    Stack<int, 33> s1;
    Stack<int, 7> s2;
    Stack<char, 3> s3;

    //създадохме три различни стека
    //с различни типове данни, с които работят,
    //както и различни размери

    //![Note] int заменя T, 33 заменя N
    //
    //или първият аргумент заменя T, вторият заменя N

    return 0;
}
```

Проблем при разделна компилация

main вижда .h файлът и ще замести T с подходящия тип, но възниква проблем, тъй като .h не вижда .cpp файла и не може T да се замести с подходящия тип в .cpp => компилаторът няма да може да генерира необходимия код за шаблона, тъй като не разполага с информацията за неговата имплементация.

- проблем с разделната компилация - .h не вижда .cpp и не може да бъде генериран .cpp файлове за конкретния тип, указан от друг файл

Решения

1. Не използваме разделна компилация и пишем всичко в един .hpp файл

```
main.cpp          A.hpp
Project18
1 #include <iostream>
2
3 template <typename T>
4 class A
5 {
6 public:
7     void f();
8 }
9
10 template <typename T>
11 void A<T>::f()
12 {
13     std::cout << "A::f()" << std::endl;
14 }
```



```
ct18
1 #include <iostream>
2 #include "A.hpp"
3
4 int main()
5 {
6     A<int> a;
7
8     a.f();
9 }
10
11 // Инстанциране на типове - имплицитно ще кажем с какви типове ще
12 // използваме класа (с какви типове да генерира кода)
```

A.h A.cpp main.cpp

Project18

```
1 #pragma once
2 #include <iostream>
3
4 template <typename T>
5 class A
6 {
7 public:
8     void f();
9 };
10
11
```

A.h A.cpp main.cpp

Project18 (Global Scope)

```
1 #include "A.h"
2
3 template <typename T>
4 void A<T>::f()
5 {
6     std::cout << "A::f()" << std::endl;
7 }
8
9 //експлицитно инстанциране на класа А за конкретни типове
10 template A<int>;
11 template A<char>;
12
```

A.h A.cpp main.cpp

Project18

```
1 #include <iostream>
2 #include "A.h"
3
4 int main()
5 {
6     A<int> a;
7
8     a.f();
9
10    return 0;
11}
```

Темплейтна специализация

def| клас/функция с различно поведение спрямо типа

Темплейтната специализация ни позволява да дефинираме специални версии на шаблоните за конкретни типове. Това е полезно, когато общата реализация на шаблона не е подходяща за даден тип и е необходима специфична реализация.

Напр.:

```
#include <iostream>

template <typename T>
class A
{
public:
    void f()
    {
        std::cout << "A::f()" << std::endl;
    }
};

//създаваме класът A, специално за типа int
template <>
class A<int>
{
public:
    void f()
    {
        std::cout << "int A::f()" << std::endl;
    }
};

int main()
{
    A<double> a1;
    a1.f(); //A::f()

    A<int> a2;
    a2.f(); //int A::f()

    return 0;
}
```

```
#include <iostream>

template <typename T>
void print(const T& value)
{
    std::cout << "print " << value << std::endl;
}

// специален print за тип int
template <>
void print<int>(const int& value)
{
    std::cout << "print int " << value << std::endl;
}

int main()
{
    print(3.14); //print
    print(42);   //print int
    print("Hello"); //print

    return 0;
}
```

def Обекти , които се държат като функции

- обекти на клас, който има предефиниран operator(), който смята резултат от функция

Умни указатели

def Обивващи класове на указателите, които менажират паметта на обектите, към които сочат; не се интересуваме от триенето, а го прави указателя (`delete`)

Съществува `auto_ptr`, който е стар и не се използва, затова няма да го разглеждаме

Unique_ptr

- имаме точно 1 указател за точно един обект
- трябва да предефинираме операторите `*`, `->`
- изтрити са копирация конструктор и оператор=
- разписани са move конструктор и move оператор=

Създаване:

Unique_ptr<A> ptr(new A(...))

//пойнтър към динамично заделен обект A

В истинския `unique_ptr` има функция `make_unique`, тя комбинира създаването на обекта и управлението на паметта, т.е. се спестява употребата на `new`, тъй като `make_unique` се грижи за заделянето на памет

Напр.:

Unique_ptr<A> ptr = make_unique<A>(3, 7, 9, 'A');

//за разлика от първото закачане на пойнтъра, тук няма нужда от new

```
#include <iostream>

class A{};

int main()
{
    std::unique_ptr<A> ptr(new A()); //уникален пойнтър към динамично
                                     //заделен обект

    std::unique_ptr<A> ptrTwo = std::make_unique<A>(); //спестява new

    return 0;
}
```

Shared_ptr

- имаме много указатели за един обект
- имаме copy и move семантики
- при първия указател се заделя обекта
- при изтриването на последния указател към обекта се изтрива и самия обект
- има указател към брояч, който следи броя на shared_ptr към един обект
- броят на указателите **[!]** НЕ пазим в статична член-данна, а като указател към брояч, за да е един и същи за всички копия на обекта ни

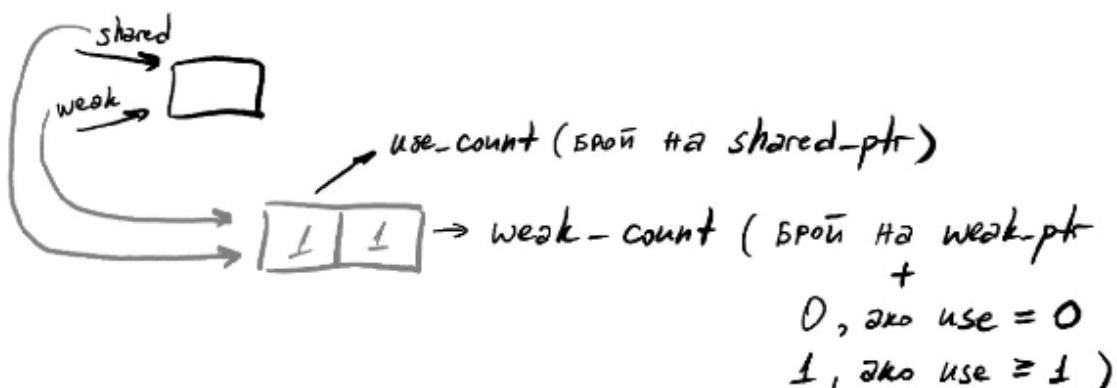
```
int main()
{
    std::shared_ptr<A> sp1(new A(2, 3));
    std::shared_ptr<A> sp1 = std::make_shared<A>(2, 3);

    return 0;
}
```

Синтаксисът за shared_ptr от stl е същият като този за unique_ptr

Weak_ptr (not-owning ref)

- указател към обект, който е менажират от shared_ptr
- не влияе на триенето и създаването
- възможно е промотиране от weak в shared
- ако обектът бъде преждевременно изтрит, имаме достъп до брояча на shared_ptr
- weak_ptr трябва да има проверка дали обектът още е жив



- Обектът се тряе, когато use-count = 0
→ Броят се тряе, когато weak-count = 0

[!] weak_count = брой на weak_ptr + 1/0 , за да сме сигурни, че имаме достъп до брояча и до проверката дали обектът още е жив

Тема 14. Type casting. SOLID principles. Design patterns

Type casting

def| когато искаме да преобразуваме от един тип в друг

```
int main()
{
    int x = 7;
    double y = x; //преобразуващо от int към double

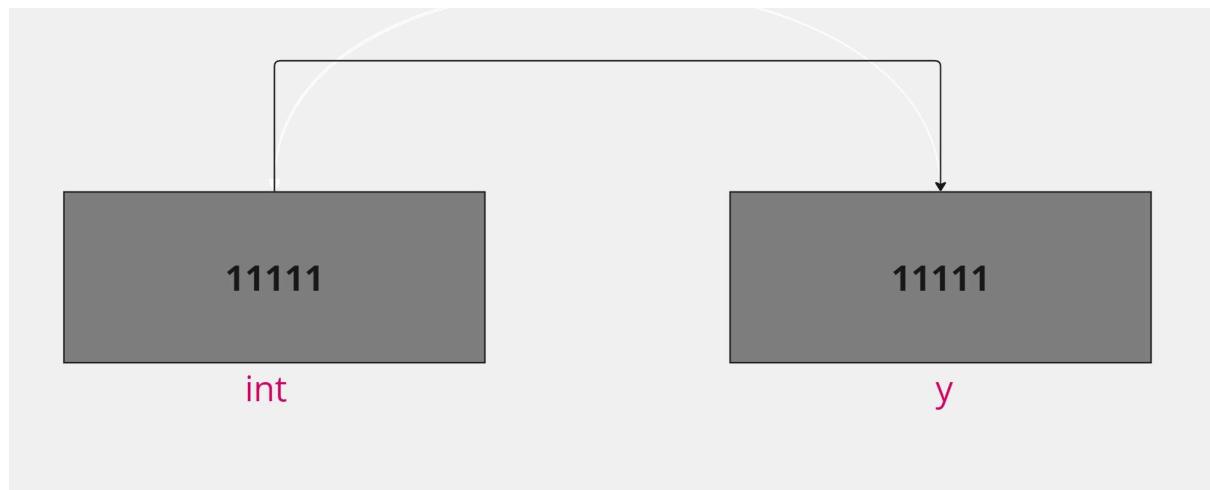
    return 0;
}
```



```
int main()
{
    int x = -1;
    unsigned y = x;

    std::cout << y << std::endl;
    //тук ще се опечата MAX на тип int
    //заради побитовите им стойности

    return 0;
}
```



```
#include <iostream>

class Base{};
class Der:public Base{};

int main()
{
    Der* d = new Der();
    Base* b = d; // смява на Base* b = (Base*)
                  // тоест C-style cast (неявно касвдане)

    return 0;
}
```

Static_cast

- шаблонна функция е
- използваме го само когато сме сигурни в типа, в който преобразуваме
- използваме го за преобразуване на примитивни типове
- използваме за upcasting (`Der* -> Base*`)
- не прави runtime check, тоест ако възникне грешка - crash/undefined behaviour
- използва се много в стари С библиотеки

```
class Base
{
public:
    int virtual getType();
};

class A:public Base
{
public:
    int getType() override {/* */};
};

class B :public Base
{
public:
    int getType() override {/* */};
};

void f(Base* ptr)
{
    if (ptr->getType() == 1)
    {
        A* obj = static_cast<A*>(ptr); //downcasting
        //в случая, сме сигурни, че обектът е от тип A
        //и можем да cast-нем надолу
    }

    /*...*/
}
```

```
int main()
{
    Base* ptr = new A();
    B* b = static_cast<B*>(ptr); //(!) това ще крашне или ще доведе до UB
    //((нямаме проверка, която да види, че ptr НЕ сочи
    //към обект от тип B, а към обект от тип A)
    return 0;
}
```

```

void f(int x, void* ptr)
{
    int* data = static_cast<int*>(ptr); //за примитивни типове
}

int main()
{
    int x = 0;
    void* ptr = &x;
    f(x, ptr);

    return 0;
}

```

Dynamic_cast

- преобразуваме, но с runtime check (която става чрез виртуалната таблица, където се пази типа), т.е. ако работим с указатели и преобразуването е неуспешно, но dynamic_cast ще върне nullptr, но ако работим с референции, ще хвърли std::bad_cast
- не се използва за upcasting
- използва се основно за downcast, когато не знаем какъв е типа
- по-бавно от static_cast

```

class Base
{
    void virtual g(); //dynamic_cast може да downcast-ва единствено
                      //когато имаме поне една виртуална таблица
};

class A: public Base{};
class B: public Base{};

void f(Base* ptr)
{
    if (A* aPtr = dynamic_cast<A*>(ptr)) //ще се замести с aPtr и ще провери
                                              //дали е nullptr
    {
        //
    }

    else if (B* bPtr = dynamic_cast<B*>(ptr)) //аналогично
    {
        //
    }
}

```

```
void g(Base& ref)
{
    A& refA = dynamic_cast<A&>(ref); //при неуспешен cast хвърля std::bad_cast
```

Const_cast

- **не го използваме**
- идеята е манипулиране на константност на обекти, с които работим чрез указатели и референции
- за да премахнем const трябва да сме сигурни, че при създаването си този обект не е бил const
- ако първоначално е бил деклариран като const => **Undefined Behaviour** (константите седят на друго място в паметта)

```
void f(const int* ptr) //забранено е да го променяме (const)
{
    int* n = const_cast<int*>(ptr); //махаме константността

    (*n)++;
}

int main()
{
    int x = 10; //първоначално не е const

    f(&x);

    std::cout << x << std::endl; //11

    return 0;
}
```

```
void f(const int* ptr)
{
    int* n = const_cast<int*>(ptr); //(!) UB, константите имат специално място в паметта
                                      //и не можем да ги местим от там
}

int main()
{
    const int x = 10; //(!) първоначално Е const

    f(&x);

    return 0;
}
```

Reinterpret_cast

- използва се за преобразуването на pointer от даден тип към pointer от друг тип, дори типовете да не съвпадат (не прави проверка)
- реинтерпретация на памет - работи като Union
- използва се, когато искаме да работим с битовете (напр. двоични файлове)

```
class X
{
public:
    int a;
    int b;
};

class Y
{
public:
    char ch[5];
};

int main()
{
    X* ptrX = new X();

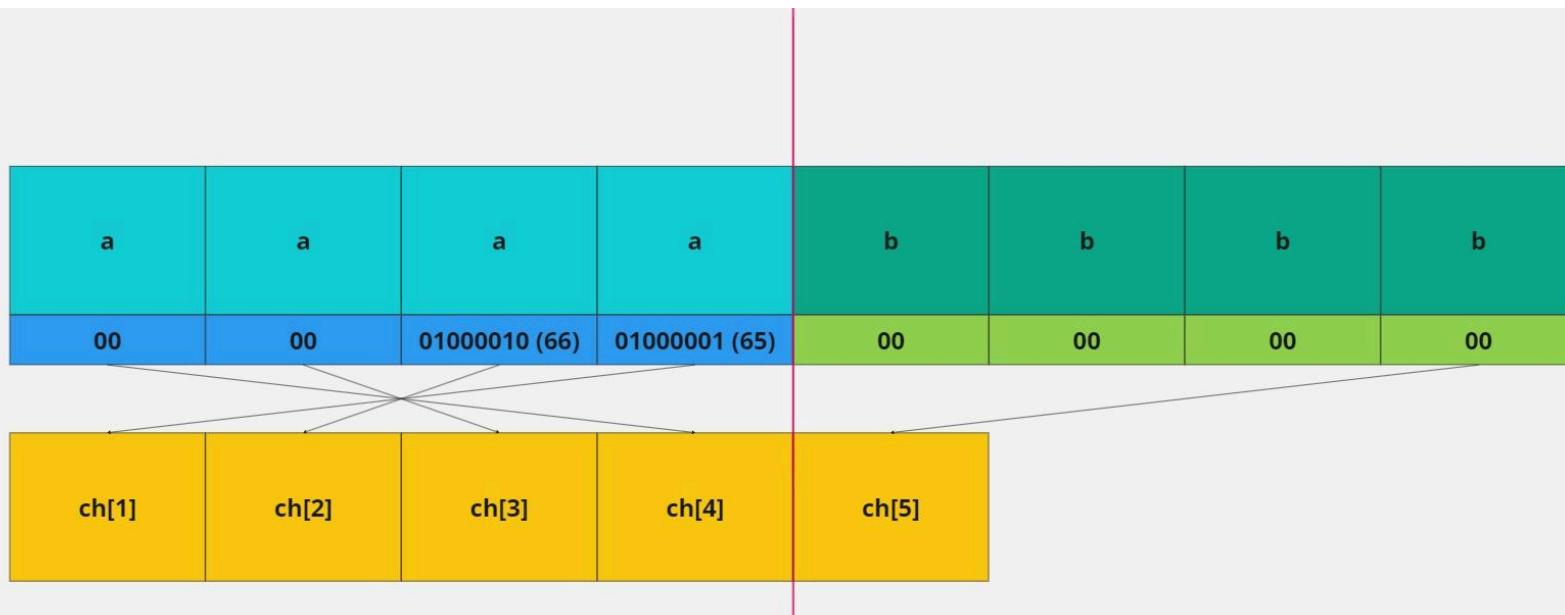
    ptrX->b = 4294967106; //11111111 11111111 11111111 01000010
    ptrX->a = 16961; //00000000 00000000 01000010 01000001

    Y* ptrY = reinterpret_cast<Y*>(ptrX);

    std::cout << ptrY->ch[4] << std::endl;
    std::cout << ptrY->ch << std::endl;

    return 0;
}
```

[!] Подобно на Union



```

#include <iostream>
#include <fstream>

int main()
{
    std::ofstream ofs("file.dat", std::ios::binary);

    int x = 5;
    ofs.write(reinterpret_cast<const char*>(&x), sizeof(x)); //всичко тук ще използваме .write()
                                                                // .read()
    return 0;
}

```

C-style cast

- изпълнява последователно:

→ const - cast
 → static - cast
 → static - cast + const - cast
 → reinterpret - cast
 → reinterpret - cast + const - cast

[!] рано или късно
 някой от кастовете ще
 успее

```

#include <iostream>
#include <fstream>

class A
{
public:
    int a;
    int b;
};

class B: public A
{
public:
    char ch[5];
};

int main()
{
    A* ptr2 = new A();
    A* ptr = (B*)ptr2; //C-style cast

    return 0;
}

```

SOLID Principles

Single responsibility principle - 1 компонент има точно 1 отговорност

cohesion - доколко компонентите са свързани
(разделяме ги в класове по strong cohesion)

Open-closed principle - отворен за разширение, но затворен за модификация (класът да не се променя при добавянето на наследник)

Liskov substitution principle - трябва да използваме указатели/референции от базовия клас, без да се интересуваме към кой наследник е насочен

Interface segregation - потребителите не трябва да разчитат на интерфейс, който не използват (правим класове с точно и ясно предназначение)

Dependency Inversion principle - модулите от високо ниво не трябва да зависят от модулите на ниско ниво (класовете трябва да зависят от интерфейси и абстрактни класове, не от конкретни класове и функции)

Design Patterns

def| **Design Patterns**

- обобщени практики (добри)
- решения на често възникващи проблеми
(не специфичен код, а концепция за решение)

3 вида:

1. **creational patterns** - осигуряват създаването на обекти, като скриват логиката по тяхното създаване (factory)
2. **structural patterns** - начин за създаване на по-сложни обекти, използвайки инструменти като наследяване и композиция
3. **behavioral patterns** - комуникация между обектите (visitor)

Singleton

- creational pattern
- осигурява само една инстанция за даден клас, към която има глобален достъп (пр. StringPool)
- private конструктор и deleted copy constructor и operator=
- функция getInstance()

Плюсове

- имаме само една инстанция на класа
- имаме глобален достъп до нея
- обектът се инициализира при първото достъпване на обекта (lazy initialization)

Минуси

- може да доведе до многонишково програмиране
- много често се определя като антипатърн
 - > не може да се тества при използване на друг код
 - > обвързва се с конкретна инстанция

Factory

- creational pattern
- статична функция (може в клас), която на база подаден аргумент връща инстанция на клас

Factory Method

- creational pattern
- предоставя интерфейс с точно един create method
- всеки наследник на този базов клас презаписва имплементацията на create, в която разписва какъв обект да върне

Base Factory
virtual Base* create() const = 0

Der1 Factory Der2 Factory
override ...

Abstract Factory

- представлява интерфейс с по един create за всички обекти, които създава
- обектите са свързани логически

Prototype (clone)

- creational pattern
- създаване на копие на обект (от полиморфна йерархия) без да се интересуваме какъв е типът

Composite

- structural pattern
- композиране на обекти в дървовидна структура
- листа и междинни обекти
- BooleanExpression

Flyweight Pattern

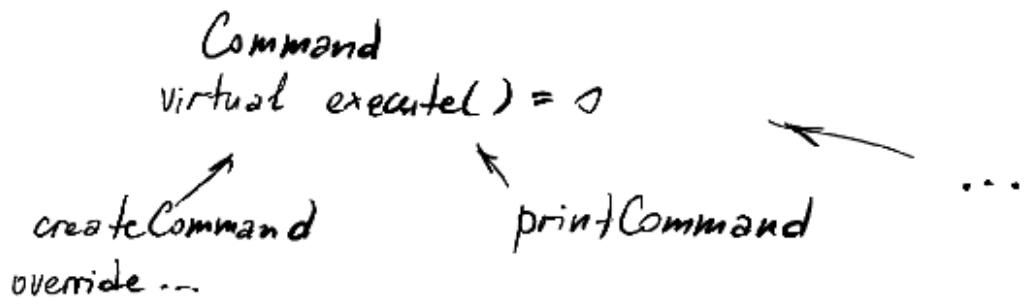
- structural pattern
- събира повече обекти в паметта като споделя общите им ресурси
- подобрява бързодействието, ако създаването е тежка операция
- StringPool

Iterator Pattern

- behavioral pattern
- начин за работа с колекция без да се интересуваме каква е тя
- има итератор - указател към конкретен елемент
- *it - връща елемент, op++, op--, op!=, op==
- колекциите трябва да имат следния интерфейс
 - > begin() - връща итератор към началото
 - > end() - връща итератор към края

Command Pattern

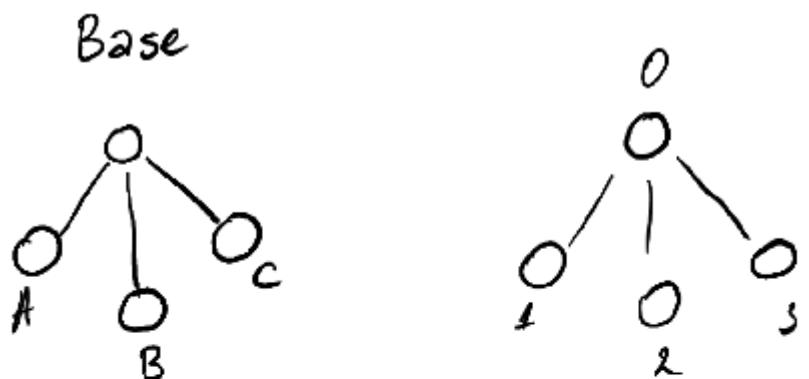
- behavioral
- програма, която получава заявки



*Command * curr = CommandFactory(...)*
curr → execute()

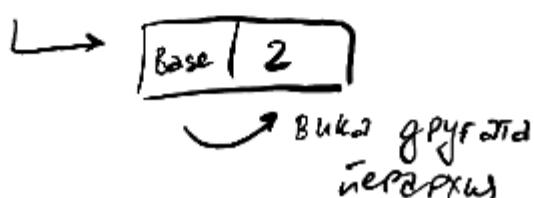
Visitor Pattern

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



f (Base, O*)*

Разширение



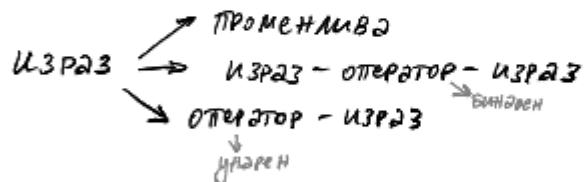
Тема 15. Дърводидна структура от обекти от полиморфна йерархия.

Калкулатор за булеви изрази:

$$(p \vee q) \Rightarrow s$$

израз булева интерпретация

- Тautология - израз, който винаги е истина
 $a \vee \neg a$
- Противоречие - израз, който винаги е лъжа
 $a \wedge \neg a$



$$\begin{cases} VAR \rightarrow a | b | c | \dots | z \\ S \rightarrow VAR | S1S | SVS | S \Rightarrow S | \gamma S \end{cases}$$

булева интерпретация: $F: VAR \rightarrow \{\text{true}, \text{false}\}$

дърво: свързан ацикличен граф

Практикуми

Pract Group 1: https://github.com/GeorgiTerziev02/Object-oriented_programming_FMI/

Pract Group 2: https://github.com/LyubenGeorgiev/OOP_2024_GROUP_2/

Pract Group 3: https://github.com/desiish/OOP_Pract_2023_2024

Pract Group 4: https://github.com/hrisge/Object_Oriented_Programming_Pracitcum_Course

Pract Group 5: <https://github.com/pepe2405/OOP-SI-Practicum>

Pract Group 6: <https://github.com/plamena-ilieva/Object-oriented-programming-23-24>

Още неща които трябва да знаем

```
double a;
std::cin >> a;
std::cout << a / 0; //при деление на тип double на 0
                    //възможни резултати са:
                    //+inf
                    //-inf
                    //not a number (nan)

// тип double също не се сравняват с оператора ==,
// а проверяваме дали abs() <= някакъв епсилон

int b;
std::cin >> b;
std::cout << b / 0; //при деление на тип int на 0
                    //се спира процеса на програмата
```

```
//constexpr -> обещание, че няма да се променя и се знае при compile time  
//const -> обещание, че няма да се променя,  
// но може да се приема и от конзолата (compile + run)  
  
//typedef <T> name [...] -> променливата ни вече е от тип <T> с [...]  
//Напр.: typedef char CharArray[100]; -> имаме тип CharArray, който е  
масив от 100 chara
```

```
//-> структури с "външен ресурс"  
struct Person  
{  
    char* name = nullptr;  
    int age;  
}  
  
//тук, name е пойнтър към char, което означава, че може да сочи към  
//низ от символи (string) съхраняван някъде в паметта.  
//Това е типичен пример за "външен ресурс" - памет,  
//която не е част от самата структура Person,  
//но е от съществено значение за нейната коректна функционалност.
```

```
// '\n' -> 2 байта  
//  
// ios::binary     .get()  
// -x             -xy  
// -y  
  
//основна разлика между ios::binary и get() е,  
//че get() взима едновременно байтовете при стандартен файл,  
//докато ios::binary - един по един
```

```
#include <iostream>

int global = 1;

int f()
{
    return global--;
}

int g()
{
    return global;
}

int main()
{
    std::cout << f() + g() << std::endl;

// [!] UB - не се знае коя от двете функции ще се изпълни първо,
// а това влияе върху резултата, ако f() се изпълни първо ще даде резултат
// 1(f) + 0(g) = 1, ако g() се изпълни първо, ще даде резултат 1(g) + 1(f) = 2

    return 0;
}
```

Дефиниции

Тема 01

def| Namespace

- инструмент за избягване на конфликти на имена
- scope (област на действие), в който има дефинирани символи (папка от символи)

def| Енумерация - enum

- тип, рестрикиран до домейн от стойности, които включват специално дефинирани константи. (енумератори)

def| Структури - struct

- последователност от полета, които се пазят в определен ред

def.| Alignment requirement

- разликата на 2 последователни адреса, на които можем да разположим дадена променлива

def.| Обединения - union

- последователност от полета, които заемат (споделят) една и съща памет

def.| Endianness

- начин на подреждане на байтовете в една дума

Тема 02

def.| Поток

- последователност от байтове "насочени" в определена посока

def.| Интерфейс (не сме го писали като def)

- набор от функции или методи (които можем да използваме, за да манипулираме даден обект/инстанция)

def.| Синхронизация (не сме го писали като def)

- правим промените в буфера и след някакъв интервал от време ще се изсипе във файла

Тема 03

def.| Двоичен файл

- готов за зареждане в паметта (**файлове с пряк достъп**)

Тема 04

def.| Член-функции

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

def.| Абстракция

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

def.| Капсулация

- ограничаване на достъпа

def.| Mutable (не сме го писали като def)

- член-дани, които могат да бъдат променяни дори от **const** функции

def.| Конструктор (не сме го писали като def)

- извиква се при създаването на обекта
- има същото име като класа/структурата
- няма тип на връщане
- можем да имаме много конструктори
- конструктор, който няма параметър се нарича default-ен конструктор

def.| Деструктор (не сме го писали като def)

- отговаря за затварянето на външни ресурси
- точно един, ако нямаме компилаторът ще създаде такъв
- също име като структурата само че ~ + име

Тема 05

def.| Макроси (не сме го писали като def)

- замести парче код с друго парче код (стрингообработка)

def.| Копиращ конструктор (не сме го писали като def)

- приема обект от същия клас и текущия става негово копие
- ако не го разпишем, то компилаторът създава такъв

def.| Оператор= (не сме го писали като def)

- приема обект от същия тип и текущия става негово копие
- текущият обект е съществувал преди това
- ако не го разпишем, то компилаторът създава такъв

Тема 06

def| Голямата четворка (не сме го писали като def)

- default констр. + деструктор + копиращ констр. + operator=

Тема 07

def| Оператор (унарни/бинарни/тернарен)

- функция със специален синтаксис

def| Приятелски класове/функции

- класове/функции, които имат достъп до private имплементациите ни

Тема 08

def| Static локални променливи (не сме го писали като def)

- държи се в паметта на глобалните/статичните променливи
- променяме продължителността на локалната променлива от automatic (т.е. до края на scope-а) на static (променливата се създава в началото на програмата и се унищожава в края на програмата, точно като глобалните променливи)
- инициализира се само веднъж - при първото влизане в съответния scope и запазва стойността си дори след като излезе от scope-а

def| Static функции (не сме го писали като def)

- обвързва се с една компилационна единица и не може да се използва от други файлове
- тоест не можем да използваме тази функция в друг .cpp файл

def| Static член-данна на клас (не сме го писали като def)

- не е обвързана с конкретен обект, а с целия клас
- всички обекти от класа използват една и съща инстанция
- инициализира се извън класа

def| Static член-функция на клас (не сме го писали като def)

- не е обвързана с конкретен обект, а с целия клас
- използва се за достъпване на статичните член-данни
- няма указател **this**
- не е нужен обект, за да се достъпи

def| Статичен клас (не сме го писали като def)

- клас, който има само статични член-данни

def| Exception

- сигнал, че има проблем

def| Stack unwinding (не сме го писали като def)

- при хвърляне на грешка изпълнението на функцията се прекратява
- програмата проверява дали текущата функция или някоя от извикващите функции нагоре по стека може да се справи с изключението (има **try-catch** блок)
- ако бъде намерен съответстващ блок за обработка на изключение, изпълнението се прескача от момента, в който е хвърлено изключението, до началото на съответстващия блок за обработка
- това изисква **stack unwinding** (премахване на текущата функция от стека на повикванията) толкова пъти, колкото е необходимо, за да може функцията, обработваща изключението, да бъде най-горе в call stack-а
- когато текущата функция се премахне от call stack-а, всички успешно създадени локални променливи се унищожават както обикновено, но не се връща стойност

01. ако някой обработва грешката
02. ако не е обработена -> **std::terminate**

Тема 09

def.| Колекция от обекти

- клас, който съдържа колекция от еднотипни обекти

def.| Placement new

- приема вече заделена памет и извиква конструкторите върху нея

def.| lvalue

- име на съществуваща променлива/функция

def.| prvalue

- литерали: **73, nullptr, “ABC”, false**
- извикване на функция, която връща копие

def.| xvalue (expiring value)

- обекти към края на жизнения си цикъл

def.| rvalue

- **prvalue + xvalue**

def.| rvalue reference

- **int&&**
- референция, която може да се прикачи единствено към **rvalue**

def.| Move конструктор

- конструктор за “открадване” на данни

Тема 10

def.| Наследяване

- Der е наследник на Base, ако разширява неговите данни/поведение

Тема 11

def.| Изглед

- клас, който се ползва за преглед на интервал от колекции

def.| Полиморфизъм

- едно име на функция, но много различни имплементации

def.| Compile time polymorphism

- по време на компилация се определя коя функция да се извика

def.| Статично свързване

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

def.| Динамично свързване

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

def.| Чисто виртуална функция (pure virtual function)

- функция, която няма имплементация
- предназначена да бъде преписана от наследниците
- може да има тяло

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

def.| Виртуална таблица - “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 юрархията да бъде виртуален, останалите ще се направят по подразбиране)

Тема 12

def| Множествено наследяване

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

def| Хетерогенен контейнер

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

def| Visitor Pattern

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

Тема 13

def| Шаблон

- клас/функция с общо предназначение спрямо типа „
(необходима функция на тип)
- шаблон = параметричен полиморфизъм

def| Темплейтна специализация

- клас/функция с различно поведение спрямо типа

def| Обекти , които се държат като функции

- обекти на клас, който има предефиниран operator(), който смята резултат от функция

def| Умни указатели

- обвиващи класове на указателите, които менажират паметта на обектите, към които сочат; не се интересуваме от триенето, а го прави указателя (delete)

Тема 14

def| Type casting

- когато искаме да преобразуваме от един тип в друг

def| Design Patterns

- обобщени практики (добри)
- решения на често възникващи проблеми
(не специфичен код, а концепция за решение)