Обектно-ориентирано програмиране (записки)

- 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
                                          казваме 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() {...тяло...};
}

namespace B
{
    f() {...тяло...};
}

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, рестриктиран 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
};
```

```
//Инициалицириране на променлива от тип color

int main()
{
    color a1 = color::red;   // 0, използаме 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

         //структура -> 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


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

//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
```

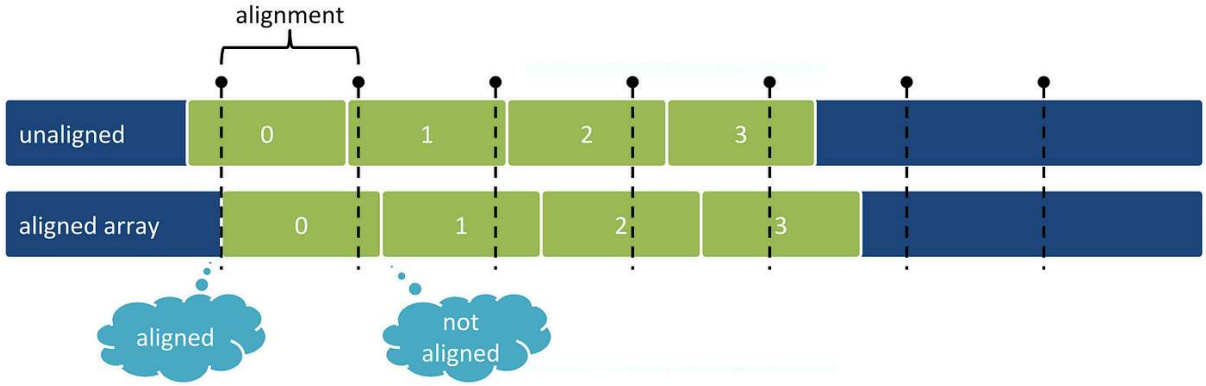
```
//Връщане на инстанция от функция

//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-а =>
                 //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-а
                 //=> 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

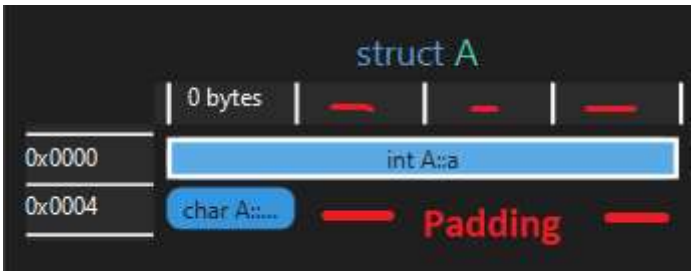
При **примитивни типове** (int, float, char….) **sizeof(<T>) == alignof(<T>)**

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 големината

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.
(int + char + 3 = 4 + 1 + 3 = 8 и също 8 % int = 0)

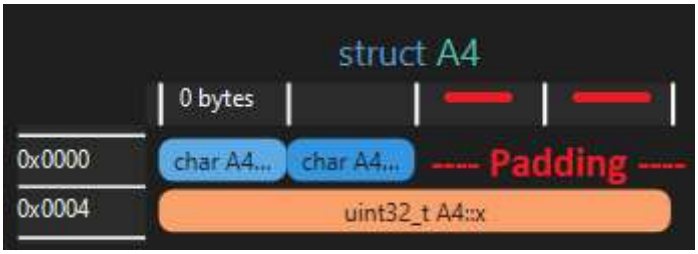
```
//масив от инстанции

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)
```

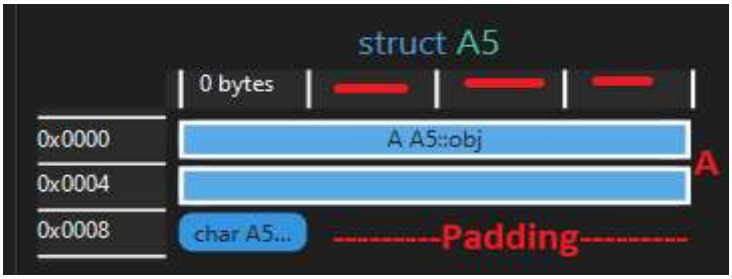
Пр. 3:

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

};

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



Пр. 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)
```

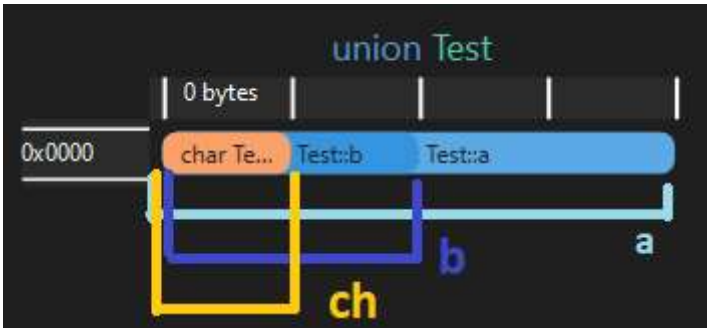
```
struct A7
{
    char ch;
    int arr[]; //ако е последен, можем да не даваме размер
               //тогава взема колкото място е останало
               //в случая имаме 3 байта padding, което не стига
               //за нито един int => не създаваме клетка на масива
};

//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

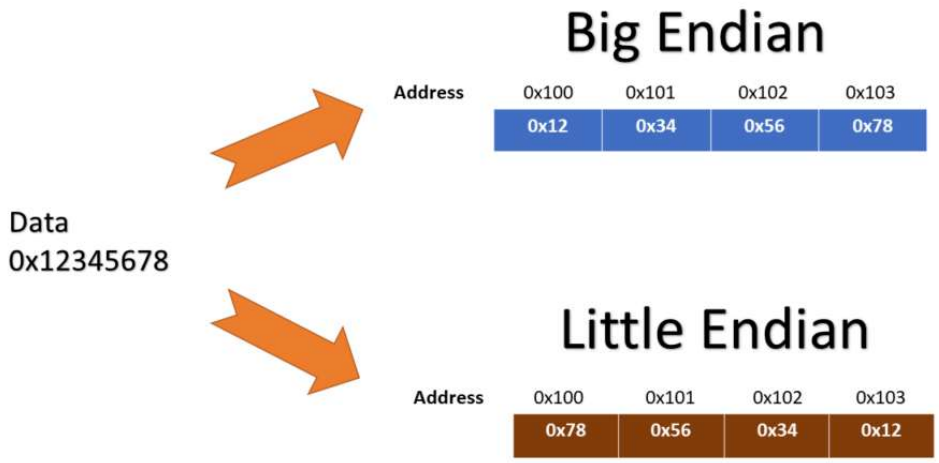
def.| последователност от полета, които заемат (споделят) една и съща памет

```
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; //K (K 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).