Тема 1.

Пространства от имена (namespace). Енумерации, структури и обединения. Видове енумерации и разлики.
Работа с инстанции: Инициализация, достъп до елементите, влагане, работа с функции. Работа с масиви.
Размер на обекти/инстанции. Подравняване и отместване. Endianness и проверка за big/little endian.
Пример със задачата за N триъгълника

Namespace (пространство от имена)

- инструмент за избягване на конфликти на имената
- scope, в който има дефинирани символи
- използваме → чрез using namespace *ns*
  → чрез оператора за резолюция :: за конкретен обект (пр. ns-name :: f())
  → чрез using *std::cout* за конкретен обект в целия scope

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

Енумерации. Видове и разлики

1. **enum** (тип изброен) - тип, рестриктиран до домейн от стойности, които включват специално дефинирани константи (енумератори)

- всеки енумератор съответства на цяло число; ако не е указано, то е предния + 1; първият има стойност 0, ако не е дадена друга

- enum е unscoped

пр.
```
enum t {
    a,          → 0
    b,          → 1
    c = 5,      → 5
    d = 4,      → 4
    e = c+d     → 9
};
```

- извикване t :: a
- енумераторите са глобални променливи

```
enum color {       проблем       enum fruit {
    orange,       <------->          orange
    red                           };
};
```

-има имплицитно / неявно преобразуване от енумератор към число

```
color c1 = color::orange;
animal a1 = animal::cat;
if(c1==a1) {...}
```

* позволява сравняване на несравними в действителност неща, затова е препоръчително използването на enum class

2. <u>enum class</u> - scoped enum

```
enum class color{              enum class fruit{
    orange                         orange
};                             };
```

няма сравнение ⟷
if(c1==a1) не работи

int x = color::orange; → не работи, защото няма неявно преобразуване, би било възможно чрез кастване : int x = (int)(color::orange);

Размер на enum:

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

sizeof(Test) = sizeof(int) = 4

enum Test 2: char{
  a=8,
  b = ~~8~~0000 не е char
};

sizeof(Test2) = sizeof(char) = 1

sizeof(enum) = размерът на най-малкия целочислен тип данни, в който могат да се поберат стойностите на енумераторите

Структури (struct)

- последователност от полета, които се пазят в определен ред

```
struct Point {
    int x;
    int y;
}
Point P {3,7};      или   Point P = {3,7};
P.x = 10;   → достъп до елемент/
              също (*ptr).x ≡ ptr->x
```

→ може да създаваме структури и динамично

```
Point *ptr = new Point {3, 4};
delete ptr;
```

→ влагане на структури - пример за абстракция

```
struct Line {
    Point beg;
    Point end;
}
```

→ подаване на инстанции във ф-ии
- по (константна) референция
- по (константен) указател

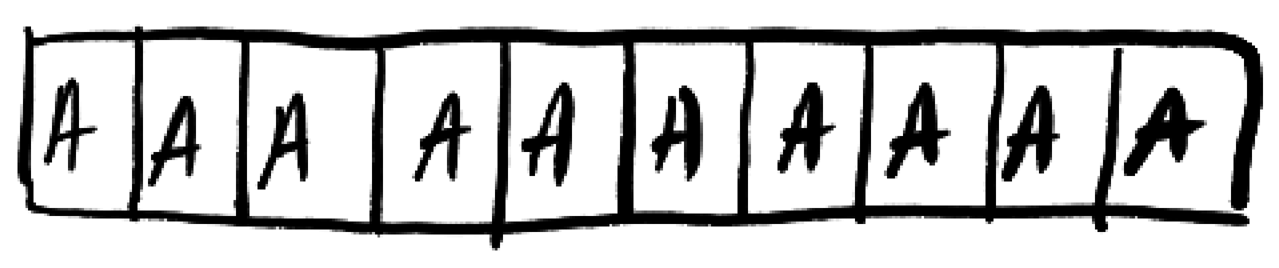
$>$ const, ако не променяме инстанциите

- по копие - избягваме!

пр. void read(const Point& p)

→ Масиви от инстанции

```
struct A {};
A arr[10];
```



↑
статично

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

↑
динамично

- Размер на инстанции

• ∀ структура има alignment requirement - разликата м/у адресите на 2 съседни член-данни; той се определя от размера на най-голямата примитивна член-данна

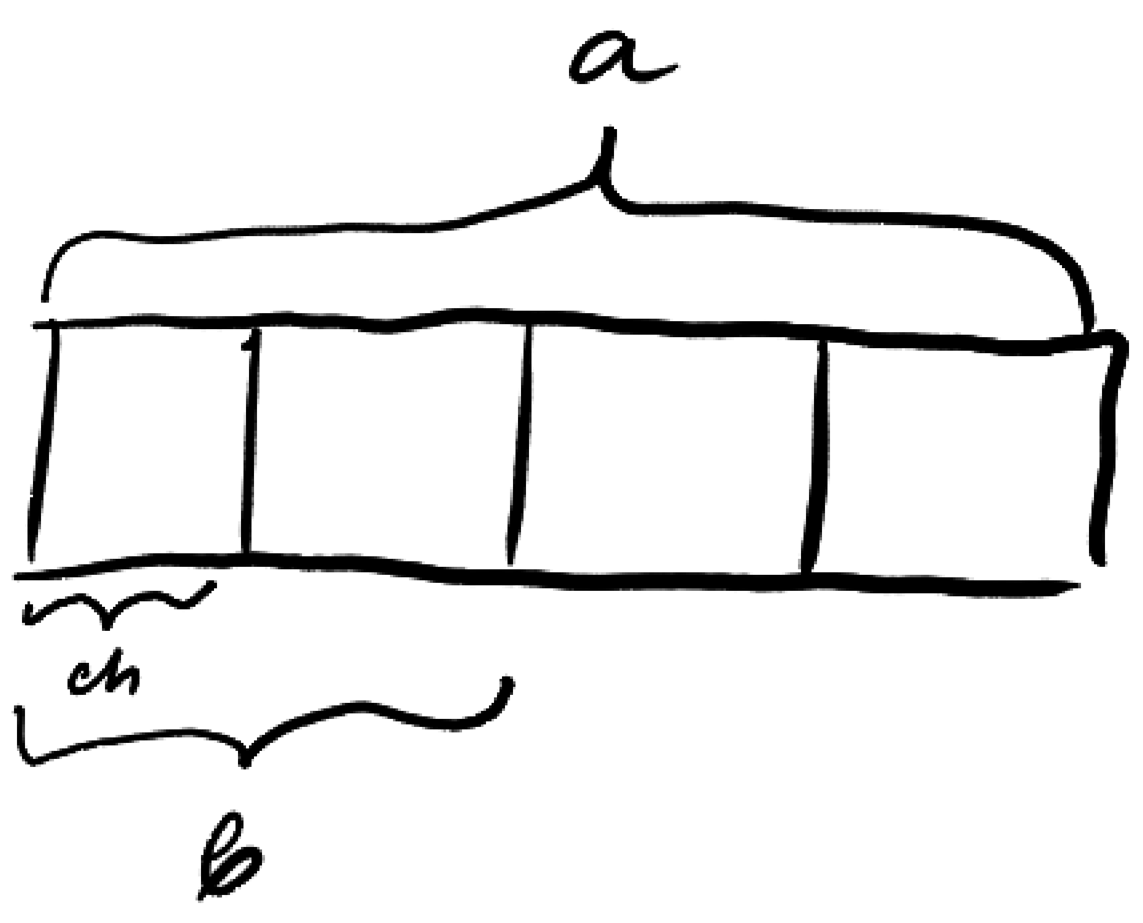
• ∀ примитивна член-данна трябва да е на адрес кратен на големината ѝ

→ празните клетки след подравняването се наричат padding
→ ако има масив като последен елемент на структурата можем да не му подаваме размер (запълва останалото място)

Обединения (union)

- Последователност от полета, които споделят/заемат една и съща памет

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



→ Test obj;
obj.a;

→ sizeof(union) = размерът на най-големия тип данни

→ чрез union постигаме полиморфизъм
→ предназначени за използване на точно едно поле

Endianness - начин на подредбата на байтовете

- little endian - най-старшият байт е най-отпред
- big endian - най-старшият байт е най-отзад

проверка:

```
bool isLittleEndian(){
    union endiannessTest{
        uint32_t n = 1;
        unsigned char bytes[4];
    } myTest;
  return myTest.bytes[0];
}
```