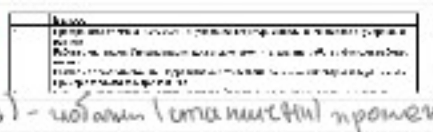
Тема 1



Глобална (Статична) - глобални (статични) променливи

Памет
- Стек - локални, нестатични пром., заделя се при комп.
- Динамична - "свободната" памет, която може да се заяви по всяко време от изпълнението на програмата
- Програмен код - там където се пази компилирания код, достъпва се с function pointers

ООП - програмна парадигма - когато една програмна система се моделира като набор от обекти, които взаимодействат помежду си
• Абстракция • Капсулация • Наследяване • Полиморфизъм

Namespace

- инструмент за избягване на конфликти на имена
- пространство от имена

```
namespace ns{
    void f() { }
    int global = 9;
}
```

Може:

```
namespace A {
    namespace B{
        f() }
}
```

Извикване:

`using namespace ns;` - вкарва всичко от namespace-а в текущия код. Позволява използването на членове на namespace-а без използването на оп. за резолюция

`using ns::f;` - позволява използването на даден член на namespace-а в дадения scope, без да налага използването на "::"

`ns::f;` - извиква еднократно даден член на namespace-а

`::` - оператор за резолюция

! Когато използваме namespace-ове, съдържащи елементи с еднакви имена, кодът ще се компилира, докато не използваме тези елементи

Енумерации

Enum - тип данни, дефиниран от програмиста, който е ограничен до домейн от стойности, представляващи специално дефинирани константи (енумератори)

! Всеки енумератор съответства на цяло число
- Ако не е зададена стойност, то тя е тази на предния +1
- Ако на първия не е зададена ст-т, то е 0

```
enum Color {
    red,              //0
    blue = 3,         //3
    orange            //4
    purple = red + blue;   //3
}
```

отделяне със ','

! enum е unscoped - енумераторите са нещо като глобални променливи, т.е. ще имаме достъп до тях от целия код

```
enum Color {
    orange
}

enum Fruit {
    orange
}
```

Няма да се компилира

При unscoped enum имаме имплицитно преобразуване от енумератор към число

Пример:

```
enum Color {
    orange = 1
}

enum Animal {
    cat = 1
}

Color c1 = Color::orange;
Animal a1 = Animal::cat;

if ( c1 == a1) {
    ще се изпълни
}                       V
```

В повечето случаи това е нежелано поведение, защото въпреки че енумераторите съответстват на едни и същи стойности, те значат съвсем различни неща.

Затова имаме и scoped enum - enum class

```
enum class Color {          enum class Fruit {
    orange                      orange
}                           }                 е ОК
```

- ако искаме да присвояваме стойността на енумератор на променлива от какъвто и да е друг тип или да сравняваме енумератори трябва да ги кастнем експлицитно към необходимия тип

! големината на променлива от тип enum (sizeof(e1)) - големината на целочислен тип променлива, в чийто домейн от стойности се побират енумераторите

Структури

- последователност от полета, които се пазят в определен ред в паметта
- член-данните могат да бъдат от различен тип
- когато са от еднакъв представянето в паметта е едно и също като при стат. масив, разликата е само семантична

```
struct Point {              Point* ptr = new Point {3,4};
    int x = 0;                 :
    int y = 0;              ! delete ptr;
};
```

отделяне с ';'

пишем ';' в края на дефиницията

Декларация на инстанция

```
Point p; // на член-данните се присвояват техните default-ни стойности
Point p1 = {1,1} // p.x=1  p.y=1
Point p2;
p2.x = 2;
p2.y = 3;
```

Достъп до член данните: <името на инстанцията>.<името на член-данната>

Можем да създаваме динамични обекти:

`Point* pointPtr = new Point();`

Достъп до член данните:

```
(*pointPtr).x = 5;   } еквивалентни
pointPtr -> y = 10;
```

Влагане на структури

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

Масиви от инстанции

```
Point arr[10]; //създава масив с 10 инстанции от тип Point
Point* ptr = new Point[n]; //създава указател към масив от инстанции тип Point в heap-а
```

- големината на масив от инстанции е

```
struct A{ };
n * sizeof(A) + sizeof(int) - защото
```

Подаване във функции

(Структурата е тежък тип, затова избягваме да създаваме копие. Вместо това, във ф-ии подаваме по референция (копирането само на адреса й е много по-евтино))

`f(Point& p)`

Важно е, когато няма да променяме инстанцията във функцията да я подаваме по константна референция

`double getDist(const Point& g, const Point& s)`

```
f1 (A obj)          |1|2|3|4|5|6|
f2 (const A obj)    |1|2|~~3~~|4|~~5~~|6|
f3 (A& ref)         |1|2|3|4|5|6|
f4 (const A& ref)   |1|2|~~3~~|4|~~5~~|6|
f5 (A* ptr)         |1|2|3|4|5|6|
f6 (const A* ptr)   |1|2|~~3~~|4|~~5~~|6|
```

във функции, които няма да променят параметрите си не можем да извикваме такива, които ще ги променят

няма да се компилира

• Връщане на инстанции от ф-я

```
A f() {                        A&/* f() {
    A obj;    V връща копие        A obj;    X връщаме референция (ук-л) към нещо, което вече не съществува
    return obj;                    return obj;
}                              }     → умира в края на scope-а

A* f() {
    A* ptr = new A {...}; V
    return ptr; → живее, докато не го изтрием
}
```

Размер на инстанции

Alignment requirement - разликата на два последователни потенциални адреса, на които можем да разполагаме дадена променлива

потенциален адрес - адрес кратен на големината на типа (по този начин ОС по-лесно се ориентира в паметта)

! Размерът на инстанцията се определя по следните правила
- Големината на структурата трябва да се дели на размера на най-голямата примитивна член-данна
- Всяка примитивна член данна трябва да е на адрес, кратен на големината ѝ

```
struct A {
    char ch;                alignof(A)=8 //=sizeof(int64_t)
    uint32_t arr[5];        sizeof(A)=40
    char ch2;
    int64_t n;
};
```

```
|ch|x|x|x|     |     |     |     |     |ch2|x|x|..|x|n|n|...|n|
0        4   7 8  11 12 15 16 19 20 23 24        31 32      39
 padding                                  padding
```

padding - празни клетки, с които се подравняват член-данните

```
struct B {
    int32_t a;
    char ch;
    char arr[ ];
};
```

Когато последното поле е масив можем да не му подаваме големина. Той ще заеме останалото място в структурата

Unions

- последователност от полета, които споделят една и съща памет
- предназначени са за използване на точно едно от полетата им (първото или последното) (при достъпване на 2 полета undef. beh.)
- sizeof() - размера на най-голямата променлива
- по този начин можем да интерпретираме една и съща памет по различни начини (имаме полиморфизъм)

Endianess

- начин на подреждане на байтовете на една дума в паметта

Big endian - най-старшият байт е пръв

Little endian - най-младшият байт е пръв

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

Проверка за big/little endian:

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

Въпроси:
- Какво става, когато в стр няма място за масив, на който не сме задали големина
- енумератори с еднакви имена в enum и enum class