Тема 1

Monday, April 1, 2024 11:50 PM

Тема 1

	Въпрос:
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 с задачата за N триъгълника.

Памет
- **Глобални (Статични)** - глобални (статични) променливи
- **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 нашия код, достъпва се с function pointers

ООП - програмна парадигма - когато една програмна система се моделира като набор от обекти, които взаимодействат помежду си

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 нашия 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 даден 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 до йн 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, тя е 0

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

отделение със ','

отделяме с ',' blue = 3, 113
orange 114
purple = red + blue; 113
}

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

enum Color {
orange
}

Няма да се компилира

enum Fruit {
orange
}

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
}

В повечето случаи това е нежелано поведение, защото въпреки че енумераторите съответстват на 0 и други стойности, те всъщност са съвсем различни неща.

Затова имаме и scoped enum - enum class

enum class Color {
orange
}

enum class Fruit {
orange
}

с ок

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

! големината на променлива от тип enum (sizeof(en))-

! 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 тип представянето в паметта е едно и също като при стат. масив, разликата е само семантична

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

→ отделяме с ';'

→ пишем ';' в края на дефиницията

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

Декларация на инстанция

```
Point p; // на член-данните се присвояват някакви 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;
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 копия. Вместо това, във ф-ии подаваме по референция (копирането само на адреса й е много - евтино)

f(Point& p)

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

double getDist(const Point& q, const Point& s)