4. Указател this. Член-функции. Конструктори и деструктори. Извикване на конструктори и деструктори. Конвертиращи конструктори. Извикване на конструктори и деструктори при масиви (статични и динамични). Абстракция. Капсулация и. Модификатори за достъп. Наследяване.

Указател this

- указател към текущия обект
- всеки обект има точно 1 такъв
- използва се за извикване на член-данни / ф-ции на класа
- константен указател (не може да се мести)

Член-функции

- ф-ция в тялото на инстанция / клас
- работят директно с член-данните на класа
- извиква се от обект на класа

Пр:

```
struct A {
    int aa;
};

bool isBig(const A& a) {
    return a.aa > 1000;
}

A a;
a.isBig();
A* ptr = new A();
ptr->isBig();
delete ptr;
```

=>

```
struct A {
    bool isBig() {
        return aa > 1000;
    }
};
```

⚠ компилатора преобразува член-ф-ците на даден клас във външни ф-ции с допълнителен параметър-
const OBJ* this

Пр:

```
struct A {
    void f() {...}
};
```

компилация =>

```
struct A {...};
void A::f(const A* this) {...}
```

Константни член-ф-ции

- не променят член-данните
- в клас могат да извикат само други const ф-ции
- могат да се извикват само на const обекти

Пр:

```
struct A {
    void g();
    void p() const;
    void f() const {
        p(); ✓
        g(); ✗
    }
};
```

```
void m(const A& a) {
    a.g(); ✗
    a.p(); ✓
    a.f(); ✓
}
```

но

```
void g() {
    g(); ✓
    f(); ✓
    p(); ✓
}
```

- Вътре при компилация

```
struct A {
    void f();            → f(A* const this);   // this може да променя паметта
    void g() const;      → g(const A* const this);
};
```

// не може да променя член-данните

// не може да мести указателя

Жизнен цикъл на обекта

{ // заделя се памет (констр)
  Point p{}; - задават се стойности на променливите (конструктор)
} // изчистват се всички ресурси (дестр.)
// освобождава се паметта за обекта (компилатор)

Конструктор

- извиква се при създаване на обекта
- заделя ресурси и инициализира променливи (initialization list)
- няма тип на връщане
- свързан като класа
- Ако няма параметри се нарича default
- автоматично се гени от комп, ако няма констр. с параметри

Пр:
struct A {
};

Комп. →

struct A {
A() {}
};

Конструктор при влагане

struct X {};
struct B {};
struct C {};

struct X {
A a;
B b;
C c;
X() {}
}

- конст на X има отговорност за живота на A, B, C (композиция)
- X() {} - при неизвикване на конст автоматично се създава X(): A(), B(), C() {}
инициализационен списък
- реда на създаване е еднакъв ред на реда на запис в структурата
- при липса на констр в иниц. лист на някой обект се вика def, ако го няма → Compile time error

Деструктор
- освобождава всичките ресурси на обекта
- НЕ отговаря за изтриването на самия обект
- специална ф-ция
- не връща тип
- свиващи като името на класа с ~ отпред
- ако няма разписан компилатора ген. автоматично

Пр: struct D {
};

Комп. →

struct D {
~D() {...}
};

- можем да имаме много констр., но само един деструктор

Пр: struct A {
A();
A(int);
A(double);
~A();
}

Деструктор при наследяване

```
struct X {
    A a;
    B b;
    C c;
};
```

```
~X() {
    ...
}
```

3 ~C, ~B, ~A

- в точно обратен ред на конструиране, първо се извиква на X
- смисъла в това е като лък да "обелим" структурата и да избегнем странични ефекти от зависимости

Пр: Констр + Дест

Конвертиращ конструктор
- констр с точно един парам.
Пр: X(int a);

explicit
- казва, че този констр., не трябва да е конвертиращ и затова се използва трябва да се прилага static_cast

Пр:

```
Point(int val) : x(val), y(val)
printPoint(const Point& p)

implicit
printPoint(10); ✓
```

```
explicit Point(int val)..
printPoint(10); ✗
printPoint(static_cast<Point>(10));
```

Извикване на конструктор и деструктор при масиви
- статични

```
{
    A arr[10]; // 5x A();
}
// 5x ~A() // изтрива се най-скоро създадения първо
```

- динамични

```
{
    A* arr = new A[3]; // 3x A();
}
// сие се вика деструктор.

{
    A* arr = new A[3]; // 3x A()
    delete[] arr; // 3x ~A()
}
```

Модификатори за достъп (интерфейс)
- публични член-функции
- позволяват на потребителя да променя член-данни
- мутатори (set) - позволяват модификация, но под контрол
- селектори (get) - връщат копие/const "/const & на данните, няма как да бъдат променени

Пр:
```
struct Bank {
private:
    int balance;
public:
    int getBalance() const;
    void setBalance(int); //validation
```

Абстракция
- използване нещо без да се интересуваме как работи
- скриване на ненужни детайли

Пр: сбирате :)
Пр: ф-ции от stl-библиотеката

Капсулация
- ограничение на достъпа на потребителя до всички елементи на програмата
- използване на модификатори за достъп:
  а) private - достъп само в класа
  б) protected - достъп в класа и наследниците
  в) public - достъп от всякъде

⚠ Клас се различава от struct по дефолтната видимост на член-данните и дефолтния метод на наследяване

Пр: struct A : B {} // public
class C : D {} // private

struct A {} // public

class D {} // private

Mutable

- мотиращи член-данни
- ~~Все~~ член-данни, които могат да бъдат променяни дори и от const ф-ция

Пр: ~~struct~~ class Logger {

```
private:
    mutable int logCount;
public:
    void logMessage(string) {
        logCount++;
    }
};
```