Деструктор this

- const указ. към данните на обекта

```
class A {
    int a;
    void f() {
        this.a=5;
    }
};
```

Клас-оператор - функция ф-я вместо на сигнатурата на оператора - работещи аналогично с класа-значение

- компилаторът предлага всеки клас ф-я с подразбиращ ф-я с уникално име и една декларация на параметър - копие на този обект

```
bool Point::isFirstQ() {
    return x>0 && y>0;
}
```

→

```
bool Point::isFirstQuadrant(Point*const this)
{
    return this->x>0 && this->y>0;
}
```

- Константни методи
  - не правят промяна
  - const клас-ф-ции
  - 4 пъти повече да се извикват от всеки
  - трябва и тях декларация на константен адрес const this
  - трябва да декларираме и класа да дефинираме преди декларацията им

```
class A {
    A(){ }
    A(){} const { }
    A(){} const { }
};
```

→ g(const A* const this) { }

- при обв ф-ции с сигн. ще сравня const, другото-же ако ще се прави. Трябва ли всеки друг обект, от който се извиква е const или не

```
A {
    f() { }
    f() const { }
};

const A obj2;
obj2.f();

A obj;
obj.f();
```

- [illegible] в general case: трябва да пазим 2 версии на метода
- достъпа до private на класа
- [illegible] и
- [illegible] с него се освобождава

Конструктори

- инициализиращ клас-ф-я, когато се създава при създаването на обекта за да

формира данните на обекта

инициализира данните на обекта

- не задаваме експлицитно нито на функция (бройка const ref или обекти)
- подобен да overload-ване

! конструк. без параметри се нарича default-ен конструктор
! ако в 1 декларация имаме клас не е дефиниран нито конструктор
- создава default-ен
- авт. дефинираме 1 к-р., def. к-р няма да се генерира автоматично
→ хубаво е да го дефинираме дори да не го използваме (вайфе + да насилим)

Извикване на def. к-р:

→ Test t;

→ Test arr[3]; → извиква 3 пъти def. к-р
→ не е ясно, дали искаме дефинираме def. к-р

Test* p = new Test[5];
delete p;
- и-ва!

- извикан се точно веднъж, но если к-р може да извика друг к-р

при създаването на обекта

```
Point {
    x
    y
    Point(int x, int y) : x(x), y(y) { }

    Point( ):Point(0,0) { }
}

    Point() = default; - когато на компилатора „се мръдне да оптимизира"
```

Конвертиращ конструктор - с само 1 параметър (в примера е int)

```
x {
    x(int a)
}

    f(x x obj) = f(x);
→ g(const x& );
    f(x ref) - не приема rvalue
```

- създава временен обект
- ако f() приема и lvalue и rvalue
- ако g() приема само lvalue

explicit X(int a)
↳ забрани, че е конвертиращ

Деструктори

- клас-ф-я, която отговаря за унищожаване на данните разрушаване на обекта
- обикновено се извикват в обратен ред на създаването
- не се задава експлицитно нито за връщане
- извикват се точно веднъж при разрушаването на обекта
! ако в stack или class не е дефиниран друг конструктор создава такъв

- не е задължително нито за аргумент
- единствен
- не се задава експлицитно нито за връщане
- извикват се точно веднъж при разрушаването на обекта
! ако в stack или class не е дефиниран друг конструктор создава такъв

```
{
    Point[3,4] // Point (int,int)
    :
} // ~Point();

{
    Point [3], // 3 x Point()
} // 3 x ~Point();

Point* arr = new Point[3];   // заделя се памет
                             // 3 x Point();
delete [ ] arr;              // 3x ~Point();
                             // освобо-ждава се паметта

Point* p=new Point();
    Point;                   // Point() // създаване 1 обект в heapa
```

- Конструктори на обекти с отговорни за създаването на елементите!
  ↳ в него се задават как конструктори да ни се извикат или не с дадено, а трябвали default-ните, ако ако няма default-ен в класа не се компилира.
- Редом на задаване на конструктори правим от по-малкото над параметъра в клас/compound, на ден тази в инициализиращ списък

```
X {
    A obj1;
    B obj2;
    C obj3;
}
X( ): A(), B(), C() {
}
}
```

Ред на задаване на конструкторите:
A(), B(), C(), X()

Ред на извикване:
X(), A(), B(), C()

Ред на извикване на деструкторите:
X(), C(), B(), A()

- деструкторите се извикват колкото задаваме (не задаваме задължителен ред) (обектите)

!!! : изпитване нещо без да знаем как работи и отговаря на изискванията! !

Задължителни ограничения на достъпа!

Модификатори за достъп:
- private достъп само в класа
- protected в класа и наследяващите
- public достъп отвсякъде ✓

- в struct всичко е public по подразбиране
- в class всичко е private по подразбиране

Мутатори за константни - промяна клас-ф-ции за достъпване и промяна на private данни

get ф-ции:

```
функция const( const ref)
    const int
```

set ф-ции:
позволяват контролирана промяна на private данните

```
    setX(int a=0, int b=0)
```

Mutable - позволява на клас на константен конструктор константности. Прави mutable клас на обекта! може да бъде променен

```
struct {
    mutable int x;

    void g() const { x++; }   ✓
}

g(const A& ref)
    ref.x++;                  ✓
```