Тема 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. Модификатори за достъп. Mutable.

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

Пример:
```
Struct A{
    int aa;
};
bool isBig(const A& a){
    return a.aa > 1000;
}
```
=>
```
Struct A {
    bool isBig(){
    return aa>1000;
    }
}
```

```
A a;
a.isBig();
A* ptr=new A()
ptr->isBig; //синтактична захар за (*ptr).
delete ptr;
```

⚠ компилатора преобразува член-ф-циите на даден клас във външни ф-ции с допълнителен параметър ~~const~~ Obj $\overset{*const}{}$ this

Пример:

```
struct A{                                  struct A{..};   this се подава винаги
      void f(){..}   --композиция-->        void A::f(A* const){..}
}
```

Константни член-ф-ции

- не променят член-данните
- в ~~клас~~ константна инстанция могат да се извикат само други const ф-ции

Пример:

```
struct A{                          void m(const A& a){
   void g();                            a.g(); X
   void p() const;                      a.p(); V
   void f() const{                      a.f(); V
         g(); X                    }
         p(); V
   }, но
void g(){
      g(); V
      f(); V
      p(); V
}
```

-при компилация

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

↑ не може да променя член-данните   ↑ не може да мести указала

Жизнен цикъл на обекта

```
{
                  / заделя се памет (констр).
    Point p{}; — задават се стойности на променливите (констр.)
}  ← изчистват се външните ресурси (деср.)
   ↑ освобождава се паметта за обекта (компилатор)
```

Конструктор

- извиква се при създаване на обекта
- заделя ресурси и инициализира променливи (initialization list)
- няма тип на връщане
- същото име като класа
- Ако няма параметри се нарича default
- автоматично се генерира от компилатора, ако няма конструктор с параметри.

Пример:

```
struct A {          Компил.→      struct A {
};                                    A() {};
                                  };
```

Конструктор при влагане

```
struct A{};
struct B{}
struct C{}

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

- конструктор на X има отговорност за живота на A, B, C (с композиция)
- X(){...} - при не извикване на констр.

автоматично се създава:

$\mathrm{X}(): \underbrace{\mathrm{A}(), \mathrm{B}(), \mathrm{C}()}_{\text{инициализиращ списък}} \{\}$

- реда на създаване винаги е в еднакъв ред на запис в структурата
- при липса на конст в инициал. списък на някой обект се вика def, ако го няма -> compile time error.

Деструктор

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

Пример: struct D{ → комп. struct D{
}   ~D(){...}
                               }

можем да имаме много констр., но само 1 деструктор

Пример: struct A {
```
        A();
        A(int);
        A(double);
        ~A();
    }
```

Деструктор при влагане

```
Struct X {      ~X() {
    A a;          ;
    B b;        } ~C, ~B, ~A
    C c;
}
```

- в точно обратен ред на констр. първо се извиква на X
- смисъла в това е като лук да "обели" структурата и да избегнем странични ефекти от зависимост

Пример: Констр + Дестр

Конвертиращ констр.
- констр. с точно един параметър

Пример: x(int a);

explicit
- казва, че този конструктор не трябва да е конвертиращ и за да се използва трябва да се приложи static_cast

Пример:

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

```
explicit Point(int val)
printPoint(10); X

printPoint(static_cast<Point>(10));
```

Извикване на констр. и дестр. при масиви
- статични

```
{ A arr[10]; //10 x A()
} //10x ~A() // изтрива се първ най-скоро създ.
```

-динамични
```
{
  A* arr = new A[3]; // 3xA()
```
(!) не се викат дестр.

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

Интерфейс за достъп
- публични член-ф-ции
- позволяват на потребителя да променя член-данни
- мутатори (set) - позволява модификация, но под контрол
- селектори (get) - връща копие/конст*/конст& към данните, няма как да бъдат променени.

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

- използваме нещо без да се интересуваме как работи

- скриване на ненужни детайли

[illegible]

Капсулация

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

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

Пример

Struct A : B {..}
  public

class C : B
  private

Mutable

- мотираща член-данни
  - член-данни, които могат да бъдат променени дори и от const ф-ции

Пример:
```
class Logger{
    private
        mutable int logCount=0;
    public
        void logMessage(string) const {
                logCount++;
        }
}
```

Student.cpp

```
#include <iostream>
#pragma warning(disable : 4996)
bool isSmallLetter(char ch)
 {
      return ch>='a' && ch<='z';
  }
bool isCapitalLetter(char ch)
 {   return ch>='A'&& ch<='Z';

}
bool containsOnlySmallChars(const char* str)
{
      size_t len = strlen(str);
      for(size_t i=0;i<len; i++)
      {   if(!isSmallLetter( str[i]))
                return false;

      }
      return true;
}
constexpr int NAME_MAX_LEN=20;
constexpr int NAME_MIN_LEN=2;
constexpr int AGE_MIN=5;
constexpr int AGE_MAX=90;
constexpr int
```

```
class Student
{
    char name [NAME_MAX_LEN +1]="Unknown";
    int age=AGE_MIN;

    bool isValidAge(int age)
    { return age >= AGE_MIN && age <= AGE_MAX;

    }
    bool isValidName(const char* name)
    {   if(name==nullptr)
            return false;
        size_t nameLen=strlen(name);
          if(nameLen <= NAME_MIN_LEN||nameLen>=
                                        NAME_MAX_LEN)
             return false;
          if(!isCapitalLetter(*name))
             return false
        return containsOnlySmallChars(name+1);
    }
public:
  Student()=default;
  Student(const char* name, int age)
  { setName(name);
    setAge(age);
  }
 int getAge() const
 { return age;
 }
```

Student.cpp
№2

```
const char* getName() const
{
    return name;
}
void setName(const char* name)
{
        if(isValidName(name))
            strcpy(this->name, name);
        else
            strcpy(this->name, "Unknown");
}
void setAge(int age)
{
        if(isValidAge(age))
            this->age = age;
        else
            this->age = MIN_AGE;
}
int main()
{   Student s("Ivan", 33);
}
```