# В предишния епизод...

На миналото упражнение се запознахме с концепциите за `constructor` и `copy constructor` . Най-важното, което направихме е да покажем, че съществуват специални методи на даден клас, които служат за определени дейности. Накратко:

- `constructor`
  - **основно приложение** - да се грижи за процеса на инициализация (заделяне на памет за член-данните на класа, задаване на начални стойности) на обектите на даден клас
  - **предефинирани конструктори** - един клас може да има много конструктори, които обаче се различават по брой и тип на параметрите, като разбира се всички конструктори имат едно и също име - името на класа
  - **подразбиращ се конструктор**
    - *явно дефиниран* - в тялото на класа се дефинира конструктор без параметри
    - *недефиниран* - я тялото на класа не е дефиниран конструктор без параметри и компилатора си го създава сам
- `copy constructor`(конструктор за присвояване)
  - **основно приложение** - извиква се, когато за инициализация на обект от даден клас искаме да ползване вече създаден обект от същият клас. Ако не се създаде от програмиста конструктор за присвояване, компилатора създава собствен. Проблем е обаче, когато се работи с адреси, указатели, стриймове и файлове (засега ще се съобразяваме, когато се натъкваме на адреси и указатели). Предефинирането на copy конструктора се налага най-често в ситуации на член-данни, които са указатели към динамична памет.
  - **синтаксис**

    ```
    //В main функцияОО
    Student first("Ivan", 66666);
    Student second = first; //тук се извиква copy constructor
    ```

    ```
    //В декларацията на класа
    Student(const Student & other_object);
    ```

    Малко тълкуване на параметъра, че не съм сигурен дали добре ви го представих:
    - *Student* - означава, че приема като аргунмент обект от тип Student
    - *&* - означава, че подава обекта по референция,т.е. ще работи не с копие на данните му, а с лично неговите данни в паметта
    - *const* - означава, че този обект, който е подаден като параметър не може да бъде променян
    - *other_object* - име за именуване на обекта в обхвата на функцията

# Оператор =

Досега можехме да се справим със следмите ситуации в C++:

```
Student first;//извиква се default конструктор

Student second("Ivan", 66666);//извиква се конструктор с параметри

Student third = Student("Pesho", 77777);//друг начин за извикване на конструктор
с параметри

Student fourth = third;//този синтаксис изисква извикването на copy конструктор
```

Обаче не сме говорили за това, какво ще се случи в следната ситуация:

```
Student fifth("Gosho", 88888);

Student sixth("Misho", 99999);

sixth = fifth;
```

Правим разлика между:

```
Student fourth = third;//тук се инициализира обект със стойностите на друг при
създаването му (извиква се copy конструктор)
```

и

```
sixth = fifth;//тук искаме да променим стойностите на един вече инициализиран
обект и да му присвоим стойностите на друг (ще видим, че тук се извиква от
компилатора предефинирания оператор =)
```

За да можем да присвояваме на вече създаден обект стойностите на друг и искаме да го правим както номрални хора (в C++ има безкрайни възможности за какви ли не извращения) е нужно да предефинираме оператора `=` . Предефиниране ще рече, че ще направим `=` да работи за обекти на класове, така както работи за `int`, `double` и компания. Понеже можем по всяко време да напишем:

```
int a = 0;
int b = 9;
b = a;
```

Но това може да се случи, тъй като `=` знае, че работи с `int` и знае как да работи с него.

Когато обаче сме си създали ние типовете данни, в случая нашите класове Student, `=` не знае как да работи с тях. Затова C++ ни дава възможност да предефинираме оператори, и да укажем какво точно да се случва, когато напишем `sixth = fifth` . Това важи не само за оператора `=`, а и за други оператори като `+`, `-` и т.н. Генерално за предефиниране на оператори по-късно, сега да се върнем само на предефиниране на оператор `=`.

**Синтаксисът** е следният:

```
Student& operator=(const Student& other_object);
```

```
Student& Student::operator=(const Student& other){
    if(this != &other){  //проверка дали не присвояваме един и същ обект
        delete[] this->name;
        //тъй като текущият обект (сочен от this) е вече инициализиран, трябва
динамичната памет, към която сочи this->name да бъде освободена

        this->name = new char[strlen(other.name) + 1];
        strcpy(name, other.name);
        this->fn = other_object.fn;
    }
    return *this;  //връщаме текущият обект,
}
```

- *operator=* - име на функцията, което е запазено в езика `c++` и обозначава точно, че тази функция предефинира `оператор =`
- функцията логично е от тип `Student&`, тъй като връща като резултат текущият обект с новоприсвоените му стойности
- аргументът е същият като на copy конструктора и това е съвсем логично, понеже те правят почти едно и също нещо - присвояват на текущия обект стойностите на обекта, който е подаден като аргумент (разликата е че copy-то се извиква при инициализиране на нов обект, а оператор = - при промяна на стойностите на вече инициализиран обект)
- накрая в имплементацията се връща текущият обект: `return *this;`

**Приликите** между `copy конструктор` и `оператор =` продължават:

- ако не са явно дефинирани от нас, компилатора си създава такива по подразбиране, които работят ефективно, освен в случаите, когато работим с динамична памет, файлове, стриймове и др.

**Разликите** между `copy конструктор` и `оператор =`:

- `copy конструктор` се извиква от компилатора, когато искаме да инициализираме за пръв път обект със стойностите на друг обект от същия тип
- `оператор =` се извиква от компилатора, когато искаме на вече инициализиран обект да му присвоим стойностите на друг обект от същия тип

## Пример за добре написан клас Student с "голяма четворка":

```
class Student{
private:
    char * name;
    int fn;
public:
    Student();
    Student(const char* name, int fn);
    Student(const Student& other_oject);
    Student& operator=(const Student& other_object);
    ~Student();
    void print() const;
};
```

```
Student::Student(){
    this->name = new char[1];
    strcpy(this->name, "");
    this->fn = 0;
}

Student::Student(const char* name, int fn){
    this->name = new char[strlen(name) + 1];
    strcpy(this->name, name);
    this->fn = fn;
}

Student::Student(const Student& other_object){
    this->name = new char[strlen(other_object.name) + 1];
    strcpy(this->name, other_object.name);
    this->fn = other_object.fn;
}

Student& Student::operator=(const Student& other){
    if(this != &other){  //проверка дали не присвояваме един и същ обект
        delete[] this->name;
        //тъй като текущият обект (сочен от this) е вече инициализиран, трябва
динамичната памет, към която сочи this->name да бъде освободена

        this->name = new char[strlen(other.name) + 1];
        strcpy(name, other.name);
        this->fn = other_object.fn;
    }
    return *this;  //връщаме текущият обект,
}

Student::~Student(){
    delete[] name;
}


void Student::print() const{
    cout << name << " | " << fn << "\n";
}
```

```
int main(){
    Student a;  //констурктор по подразбиране
    Student b("gosho", 32);  //конструктор с параметри
    Student c = b;  //copy конструктор
    Student d("pesho", 42);  //конструктор с параметри

    Student * p = new Student("sasho", 52);  //указател към динамичен обект от
класа Student, създаден посредством оператор new и заделен в динамичната памет

    a.print();
    b.print();
    c.print();
    d.print();
    p->print();
```

```
    c = d;
    c.print();

    delete pointer;
    return 0;
}
```

# Деструктор

## Жизнен цикъл

Можем да си представим целия цикъл на живот на един обект по следния начин:

- Обектът се създава - заделя се място в паметта за член-данните на обекта (посредством констроктури, copy конструктор)
- Пирсвояват му се стойности - на тези член-данни могат да се присвояват стойности (конструктори, copy конструктор, оператор =)
- Обектът се разрушава - заделената памет за член-данните на обекта се освобождава и обекта вече не е "жив" - не е в паметта (деструктор)

Досега говорихме как се създава и инициализира един обект, как му се присвояват стойности. Сега трябва да се каже и как се разрушава.

Казвали сте по УП, а и да сте забравили, сега ви казвам, че обхватът или животът на всяка променлива или обект, приключва или когато приключи изпълнението на функцията (излезем от нейния scope), в която са заделени статично, или когато се освободи динамичната памет с delete или delete[], ако са динамично заделени.

### Статично заделена памет

Може да започнем с прост пример с примитивни типове данни за това какво се случва в паметта (от асистента на IV-та група):

```
int f(){
    int a = 5;
    //3-ти момент
    return a;
}

int main(){
    //1-ви момент
    int b;
    //2-ри момент
    b = f();
    //4-ти момент
}

//5-ти момент
```

Нека да видим във всеки от моментите какво се случва в паметта:

- 1ви момент - няма заделена памет
- 2ри момент - 4B (заделена е памет само за `b`)

- 3ти момент - 8B (заделена е памет за `a` и `b`)
- 4ти момент - 4B (функцията в която е декларирана `a` приключва изпълнение и паметта за всички променливи декларирани в нейното тяло е освободена, затова остава заделената памет само за `b` )
- 5ти момент - няма заделена памет, програмата е приключила и при края на main функцията, всяка памет заделена в нейното тяло е освободена

**Динамично заделена памет**

**!!!Винаги изтривайте динамичната памет, която сте заделили!!!**

При използваме операторите `new` и `new []` за заделяне на динамична памет и нейното освобождаване с `delete` и `delete []` е малко по-различно.

Динамично заделената памет няма обхват, тя си стои алокирана някъде в паметта, без да я интересува коя функция приключва изпълнението си. Например:

```
void f(){
    char * mem = new char[25];
}

int main(){
    f();
    return 0;
}
```

Паметта, заделена във функцията f() си стои в heap-а и въобще не я бърка дали f() е приключила изпълнение. Тя си стои и си чака някой да я "delete-не".

От друга страна, за указателя `char * mem` важат правилата за статично заделените променливи (в случая съхранява като стойност адреса на динамично заделената памет) и при приключване на изпълнението на функцията f(), паметта на `mem` се освобождава. Така изпадаме в ситуация в която имаме заделена динамична памет, която чака някой да я събори, но нямаме указател, който сочи към нея, достъпа ни към нея вече не съществува. Затова е добре винаги да се грижим за паметта, която е заделена динамично и за решение на този проблем има два варианта:

- да ползваме динамичната памет локално, тоест да я изтрием на края на функцията, в която е зададена:

  ```
  void f(){
      char * mem = new char[25];
      //..
      delete[] mem;
  }

  int main(){
      f();
      return 0;
  }
  ```

- да върнем адреса й и да можем да работим с нея в друга функция ():

```
char * f(){
    char * mem = new char[25];
    //..
    return mem;
}  //тук самата променлива mem, която пази адреса, ще бъде премахната от паметта, но не и динамично заделената памет

int main(){
    char * p = f();
    //..
    delete[] p;
    return 0;
}
```

## И вече дойде време за деструктора...

Деструкторът е метод на класа, който се грижи за това паметта, която е заделена за един обект от този клас, да бъде освободена. Той трябва да се погрижи всички член-данни на обекта да си освободят паметта, както и ако използваме някъде заделена памет асоциирана с член-данните на няшия обект, тя да бъде, както се казва, по европейски, цивилизовано изтрита.

Както вече надявам се свикнахте, по подобие на другите специални методи на класа, и деструкторът, ако не е дефиниран от нас, компилаторът си задава default-ен деструктор, който сам се грижи за всичко описано по-горе успешно, но не и когато става въпрос за динамична памет. Затова може да не се дефинира деструктор, когато не обвързваме някоя от член-данните на обекта с динамична памет, но ако го правим е задължителен.

Ще ползвам класа Student от по-горе (за удобство и от мързел):

**Декларацията** е следната:

```
~Student();
```

Името на деструктора е същото като това на класа само че с `~` опред, както и без параметри. Един клас може да има само един конструктор.

**Дефиницията** е следната:

```
Student::~Student(){
    delete[] name;
}
```

понеже имаме член-данна указател, който при създаването на обекта задаваме да сочи към динамична памет с оператор `new []`, е нужно да се погрижим за нея в тялото на деструктора.

Относно **извикването** на деструктора, не е нужно да се притесняваме - той се извиква автоматично от компилатора когато ще завърши живота на обект (при излизане от scope, в който е бил създаден обект на класа, ако е създаден статично, или при изикване на опеатор `delete` или `delete []` върху динамично създаден обект на класа)

**С това вече сме покрили материала за така наречената ГОЛЯМА ЧЕТВОРКА (навсякъде е различно, на английски май беше Rule of Three)**