Тема 6

Динамична памет в класовете. Голямата тетворка

Във всяка структура/клас се генерират автоматично:
- конструктор по подразбиране (ако не е имплементиран друг конструктор)
- копиращ конструктор (ако не сме имплементирали наш)
- оператор= (ако не сме написали сами оператор=)
- деструктор (ако не сме написали наш деструктор)

Тези четирите заедно се наричат "Голямата тетворка"

Да разгледаме следната структура:

```
struct Test
{
  A obj1;
  B obj2;
  C obj3;
};
```

Ако поне функциите от голямата тетворка не са дефинирани, компилаторът ще създаде такива.

Пример:

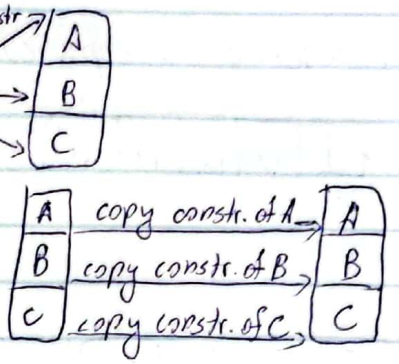
```
int main()
{
 Test object1; - извиква def. constr.
 Test object2(object1); - извиква copy constr.
 object1 = object2; - извиква operator=
} - извиква destr. x2
```

Кодът се компилира успешно и функциите имат правилно поведение

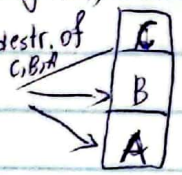
Как работят дефинираните от компилатора функции.
Всяка от тези функции извиква рекурсивно същите функции върху член-данните

Пример

```
int main()
{
  Test object;
  Test object2(object); -> copy constr. of Test
}
```



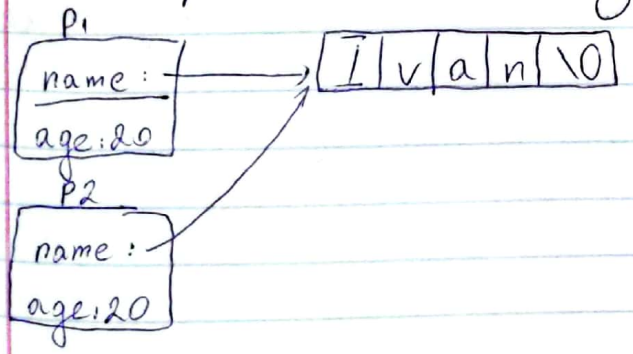
-> destr. of Test



Възниква проблем при функциите, генерирани от компилатора, когато в структурата/класа има заделена динамична памет
Да разгледаме следния код:

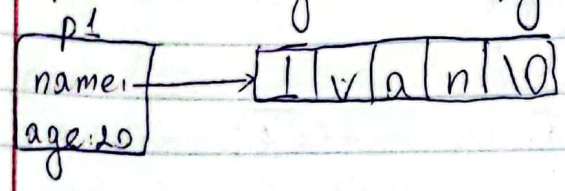
```
struct Person
{
  char* name;
  int age;
  // конструктор с параметри
};
int main()
{
  Person p1("Ivan", 20);
  Person p2(p1);
}
```

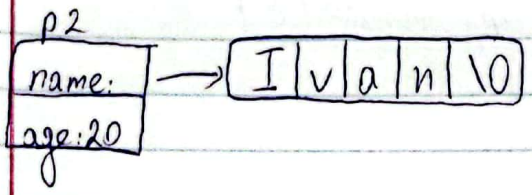
Компилаторът генерира копиращ конструктор с неправилно поведение:



Това е shallow copy. В p2 са се копирали стойностите на p1, което довежда до споделяне на една и съща динамична памет. В тази ситуация ще трябва да имплементираме експлицитно copy constr., operator= и destructor, защото генерирането от компилатора не работят правилно.

Правилното поведение на copy constr. е:





При дефинирането на оператор = трябва да се прави проверка за случаят p=p, тоест да присвояваме нашата информация върху нята отново. Тъй като първоначално трием данните, а после ги копираме, в този случай е възможно да се получи, че искаме да присвоим информация, която току-що е била изтрита, което би било пагубно за програмата.

Добра практика е трирането и копирането да бъдат в отделни функции, за да няма повтаряне на код. Така при всички класове, които използват динамична памет, тези функции изглеждат по един и същ начин. Разликата е в имплементацията на free (функция за трие) и copy-ctor (функция за копиране).

(*) Затова задължително преди да освободим сбора памет и да копираме нещо ново, правим проверка дали ще копираме различни данни

if (this != &other) - this е указател, а other обект, затова приемаме other по референция