Тема 4

Член-функции. Конструктор и деструктор. Извикване на конструктори и деструктори. Конструктор и деструктор при композиция на обекти. Модификатори за достъп. Капсулация

Член-функциите са функции, които работят с член-данните на обекта от дадена структура/класи се извикват от обекта. Компилаторът преобразува всяка член-функция в обикновена функция с уникално име и един допълнителен параметър-указател към обекта. Има константни член-функции, които не променят член-данните на обекта и се указват чрез записването на ключовата дума const в декларацията и в края на заглавието в дефиницията им. ~~Те не променят~~ Могат да се извикват от константни обекти. Има член функции, които се създават автоматично (освен, ако не сме си написали нашите такива):

- Конструкторът е член-функция, която се извиква веднъж - при създаването на обекта. Не й се указва експлицитно тип на връщане. Връща константна референция. Конструкторът има същото име като класа. Можем да имаме повече от един конструктор, но трябва да имаме задължително поне един. Конструктор без параметри се нарича default-ен (по подразбиране). Когато нямаме default-ен, той се създава автоматично, ако нямаме друг конструктор.
- Деструкторът се извиква веднъж при изтриването на обекта. При него също не се указва експлицитно тип на връщане - има същото име като класа със символа '~' преди това. Деструкторът няма параметри. Дефинира се по default.

Пример за извикване на конструктори и деструктори:

```
struct Test
{
  Test()
  {
   cout << "Object is created" << endl;
  }
 ~Test()
  {
   cout << "Object is destroyed" << endl;
  }
 int a;
 int b;
};
```

```
int main()
{
  while()
  {
     Test t; // Object is created
     {
       Test t2; // Object is created
     } // Object is destroyed (t2)
  } // Object is destroyed
}
```

Когато имаме динамична памет, задължително извикваме експлицитно деструктора с ключовата дума delete

Пример:

```
int main()
{
  Test* t = new Test; // Object is created
}
```

В този случай няма да се извика деструктор, който да освободи заделената динамична памет, а ще се извика само деструкторът на указателя, което ще доведе до теч на memory leak.

Всички конструктори извикват в началото конструктор по подразбиране на член данните:

```
struct A; struct B; struct C;
struct X
{
 A objA;
 B objB;
 C objC;
 X()
{ -> A(), B(), C()
  cout<< "X()" << endl;
}
~X()
{
 cout << "~X()" <<endl;
}~C(), ~B(), ~A()
```

} Всеки един от тези трябва да има default-ен конструктор, за да се построи тази инстанция

Можем от даден конструктор да извикаме други конструктори на член-данни

```
struct A                struct B
{ int x; char ch;       { A obj;
  A(int x, char ch)       ----
}                       }
```

Това няма да се компилира, защото default-ният конструктор на B ще се опита да извика default-ният конструктор на A, а той няма такъв

Има два начина да се поправи:
- да се направи default-ен конструктор на A
- да накараме конструктора на B да извиква създадения в A конструктор, който не е default-ен.

Това става по следния начин:

```
B(...) : obj(3,'c');
```

Понякога искаме потребителите да нямат достъп до всички член-данни и методи на даден клас. Това е така, защото техната програма може да доведе до неочаквано поведение на нашата програма. Принципът за капсулация ни помага като позволява да определим кои методи и атрибути е нужно да се използват от потребителите на нашия клас. Това става чрез така наречените модификатори за достъп:
- private - член-данните не са достъпни извън тялото на класа/структурата
- public член-данните са достъпни за всеки
- protected - член-данните са достъпни за ~~всеки~~ само в текущия клас и наследниците

Ако не е зададен модификатор за достъп, то в класа всичко е private (by default), а в структурата всичко е public (by default). Съществуват и при наследяването - public при структура (by def.) и private при клас (by default).