Обектно-ориентирано програмиране (записки)

- **Марина Господинова**
- **Илиан Запрянов**

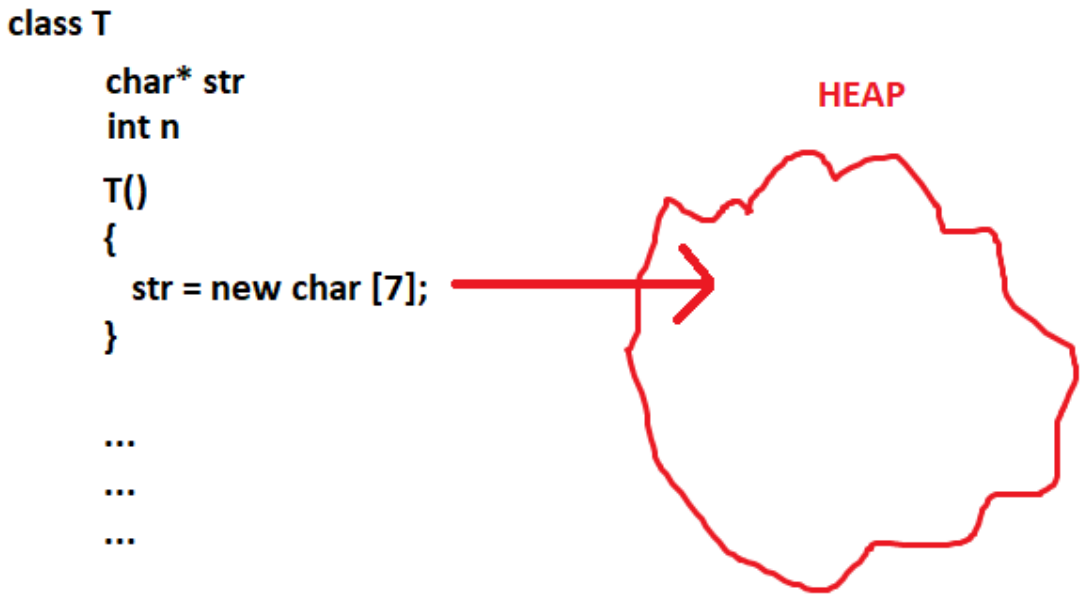
Тема 06. Голямата четворка

Димитриев

Проблем при генерираните от компилатора operator= и копиращ конструктор имаме при работа с динамична памет

```
//ще работим със следния код

class T
{
	char* str; //напомняме, че [str] е ПОЙНТЪР, а НЕ МАСИВ
	int n;

public:
	T()
	{
		str = new char[7]; //насочваме ПОЙНТЪРА [str] към динамичен масив
	}

	~T()
	{
		delete str;
	}
};


int main()
{
	T t1;

	T t2(t1); //викане на копиращия конструктор

	return 0;
}
```

Какво се случва всъщност при викането на копиращия конструктор?



От УП знаем, че пойнтъра str съдържа адреса на обекта, към който сочи (в случая масив от 7 елемента (с терминиращата 0), А НЕ СЪДЪРЖАНИЕТО НА МАСИВА. То тогава при извикването на копиращия конструктор (default) в t2 ще се изпълни скришно [!] t2.str = t1.str [!] , което ще запази адреса, към който сочи член-данната str на t1. Тоест член-данните на двата обекта сочат към едно и също място в паметта (към един и същи масив). Това копиране че нарича shallow copy. Защо това е проблем?

Тъй като имаме заделена динамична памет на член-данна, за чиито живот отговаря нашия обект, то по правило в деструктора на класа ще трябва да я освободим. Но какво се случва всъщност?

```
int main()
{
	T t1;
	T t2(t1); //копираме данните в [t2]

	//дотук видяхме, че двата пойнтъра в обектите сочат към един и същи адрес


	return 0;

} // вече знаем, че в края на scope-а се викат деструкторите на декларираните от нас статични обекти
//и че също така деструкторите се викат в обратен ред на конструкторите =>
//ще извикаме деструктора на [t2], след което този на [t1]
//[!НО!] когато извикаме деструктора на [t2], се освобождава паметта, заделена за масива, към който
сочи пойнтъра,
//това означава, че когато извикаме деструктора на [t1], той ще иска да освободи вече освободена памет
// (тъй като пойнтъра на [t1] сочи към същия масив) => грешка
```

За да решим този проблем, когато имаме динамична памет разписваме така наречената **“Голяма четворка”** (default конструктор, деструктор, operator=, копиращ конструктор). Не ги пишем, когато нямаме член-данна, работеща с динамична памет, за чиито живот трябва да се грижи обекта, в който се намира.

Напомняме, че:

- **конструктор** - създава обект и инициализира член-данните му
- **копиращият конструктор** - създаване обект като копие на съществуващ обект (копира)
- **operator=** - копира стойностите на член-данните от един обект в друг вече съществуващ обект (трие + копира)
- **деструктор** - освобождава ресурсите, заети от обекта

Мои неща:

Разписване на голямата четворка

Освен конструктори и деструктор (които вече разгледахме) при наличието на динамични член-данни е необходимо пренаписването на допълнителни две функционалности (**копиращ конструктор** и **оператор=**).

```
class A
{
private:

	int x = 0;
	int y = 0;

	void copyFrom(const A& other)
	{
		this->x = other.x;
		this->y = other.y;
	}

public:
	//A(); -> това не се създава само

	A(int a, int b); //припомняме, че ако създадем конструктор,
			 //НЯМА да се създаде default-ен, а ако искаме такъв
			 //трябва да го разпишем САМИ

	A(const A& other) //конструктор, който приема като параметър ЕДИНСТВЕНО
	{		  //обект от същия тип

			//неговата основна функция е да запази стойностите на [other]
			//в НОВ обект (който досега не е съществувал) от същия тип

	    copyFrom(other); //за разлика от default-ния конструктор, копиращият такъв,
			     //се създава винаги, независимо дали има други или не,
			     //но ни дава свобода да го презапишем с наша логика,
			     //тъй като не е предназначен да се справя с ДИНАМИЧНО заделени
			     //член-данни на класа

			     //в случая НЯМА ДИНАМИЧНО заделена член-данна, тоест
			     //default-ният копиращ конструктор може да се справи с
			     //промяната на данните сам, затова в примера е излишно това
			     //пренаписване на копиращия в конструктор, но го правим, за да
			     //видим какво всъщност става отзад, когато не го разписваме
	}
};
```

```
class A
{
private:

	int x = 0;
	int* nums = nullptr;
	size_t numsSize = 0;

	void copyFrom(const A& other)
	{
		this->x = other.x; //копирането на [x] става по стандартен начин

		this->nums = new int[other.numsSize] {}; //тъй като искаме динамичен масив,
							 //трябва да заделим такъв с размера на този,
						//в обекта, чиито стойности искаме да вземем
						//и да пренасочим пойнтъра към него (новия масив)

		for (unsigned i = 0; i < other.numsSize; i++)
		{
			nums[i] = other.nums[i]; //записване на стойностите
		}
		this->numsSize = other.numsSize;
	}
```

```
-----------------------------------------------------------------------------------------------
public:

	A() = default; //когато имаме зададени default-ни стойности на член-данните
		       //е достатъчно да кажем, че искаме да имаме default-ен, който да използва
		       //зададените default-ни стойности на член-данните

	A(int x, int* nums, size_t numsSize) {/*тяло*/ };

	A(const A& other)
	{
		copyFrom(other);
	}

	~A()
	{
		delete[] nums; //тъй като обектът е отговорен за живота на nums
			       //(инициализира се в рамките на този клас)
			       //трябва да освободим заделената памет за него в деструктора

		nums = nullptr; //след освобождаване на паметта, nums сочи към вече освободена памет
				//и е добра практика да му кажем да сочи към нищо (nullptr)
	}
};
```

```
class A
{
private:

	int x = 0;
	int* nums = nullptr;
	size_t numsSize = 0;

	void copyFrom(const A& other)
	{
		this->x = other.x;

		this->nums = new int[other.numsSize] {};
		for (unsigned i = 0; i < other.numsSize; i++)
		{
			nums[i] = other.nums[i];
		}
		this->numsSize = numsSize;
	}

	void free()
	{
		delete[] nums; //освобождаваме паметта заделена за [nums]
	}
```

```
-----------------------------------------------------------------------------------------------------------
public:

	A() = default;
	A(int x, int* nums, size_t numsSize) {/*тяло*/ };
	A(const A& other)
	{
		copyFrom(other);
	}

	A& operator=(const A& other) //когато А съществува, се използва така наречения оператор
	{                            //за присвояване (който също съществува ВИНАГИ, но имаме
	                             //право да го презапишем с наша логика)

		if (this != &other) //проверяваме дали не присвояваме същия обект на текущия
		{                   //тъй като, ако се случи това ще освободим паметта и ще пренасочим пойнтъра
		                    //към вече освободена памет, а не същата

			free(); //ЗАДЪЛЖИТЕЛНО освобождаваме паметта преди да запишем новите данни,
			        //тъй като в противен случай ще преместим пойнтъра към ново място
			        //и ще имаме заделена памет, която не използваме и до която нямаме достъп
			        //(memory leak)

			copyFrom(other);
		}

		return *this;

	}

	~A()
	{
		delete[] nums;
		nums = nullptr;
	}
};
```

Пример за подробно разписана голяма четворка

https://github.com/Zapryanovx/FMI_2024_Object_Oriented_Programming/tree/main/00_demos/Big%20Four

Кога се извикват конструктор/деструктор/operator=/копиращ констуктор

1. **Конструктор**

https://github.com/Zapryanovx/FMI_2024_Object_Oriented_Programming/blob/main/00_demos/Big%20Four%20Calls/constructor.cpp

2. **Копиращ конструктор**

https://github.com/Zapryanovx/FMI_2024_Object_Oriented_Programming/blob/main/00_demos/Big%20Four%20Calls/copy.cpp

3. **Оператор=**

https://github.com/Zapryanovx/FMI_2024_Object_Oriented_Programming/blob/main/00_demos/Big%20Four%20Calls/op%3D.cpp

4. **Деструктор**

https://github.com/Zapryanovx/FMI_2024_Object_Oriented_Programming/blob/main/00_demos/Big%20Four%20Calls/destructor.cpp