A programozás alapjai 2.

Hivatalos segédlet a negyedik laborhoz.

Tartalomjegyzék

Dinamikus memóriakezelés

Az osztályok bevezetésével egy komoly problémával szembesülünk. A ***malloc***-cal történő memóriafoglalás **nem hívja meg az osztály konstruktorát**.

Ez nekünk miért baj? A memóriafoglalás után külön be kell állítanunk minden tagváltozó értékét külön-külön. Lehet akár nagyon sok tagváltozónk is, amiknek a beállítgatása rengeteg felesleges művelettel jár, és ezt minden egyes objektumnál meg kéne csinálnunk. Na nem! Pont erre lehetne használni a konstruktort, hogy csak átadogatjuk neki a paramétereket, és minden tagváltozónak kezdőértéket adjon.

A megoldás erre a malloc-free páros helyett két új művelet: a **new** és a **delete**. Szintaxisuk:

```
int *x = new int;
int *x_array = new int[10];

delete x;
delete[] x_array;
```

Figyelem, *new*-val foglalt memóriaterületet mindig *delete*-tel szabadítunk fel, *new[]*-val foglalt memóriaterületet pedig mindig *delete[]*-tel! Ha ettől eltérünk, a program viselkedése nem definiált, bármi történhet (felrobbanni azért nem fog a géped, de érdekes hibák jöhetnek elő).

Érdemes kiemelni: C++ programon belül sose használjátok a malloc-free párost, helyette new-delete.

Mindezek mellett, a **new típusbiztos**, miközben a malloc egyáltalán nem az.

Összefoglalás: new/delete vs malloc/free

Tulajdonság	new/delete	malloc/free
Honnan foglal memóriát?	Free Store	Heap
Visszatérési érték	teljesen típusos pointer	void*
Hiba esetén mit csinál?	hibát dob (ld. később: std::bad_alloc); sose ad vissza NULL-t	NULL-t ad vissza
Honnan tudja a szükséges méretet?	compiler számolja ki	byte-okban kell megadni (sizeof(típus)*elemszám)
Hogyan kezel tömböket?	közvetlenül megadható tömbméret (pl. new int[10])	manuális számítást igényel (ld. előző pont)
Átméretezés (reallocating)	nem beépített (nincs is rá szükség)	reallocate() függvénnyel megoldható, de nem hív copy konstruktort
Túlterhelhető? (ld. köv. labor: viselkedés újradefiniálható)	igen	nem
Meghívja a konstruktort, destruktort?	igen	nem

A dinamikus memóriaterület egy másik fajtája: Free Store

Egyike a két dinamikus memóriafoglalást biztosító memóriaterületeknek. Különlegessége, hogy az **objektum életciklus ideje kevesebb is lehet, mint amennyi eltelik a new és a delete között**. A többi időben (tehát amikor nem él az objektum, de a memória le van foglalva) void*-on keresztül ugyan lehet **módosítani** a területet, de az **objektum nem-statikus változóin, metódusain keresztül nem**.

Dinamikus tagváltozó

Egy osztály rendelkezhet dinamikusan foglalt tagváltozókkal. Ezeknek memóriát általában a konstruktorban foglalunk (ha már tudjuk, mekkora memóriaterület szükséges - pl. default konstruktorban nem adott meg semmit a felhasználó). A memória **felszabadítását** mindig a **destruktorban** végezzük el.

Ha egy osztályban van dinamikusan kezelendő tagváltozó, mindig írjunk **másoló konstruktort**.

Másoló konstruktor - Miért is deep copy?

Tegyük fel, hogy van egy “MyString” osztályunk, amiben dinamikusan tárolunk egy karaktertömböt. Legyen egy példány belőle a “magic” nevű MyString, aminek a dinamikus tömbjében “my magic” szerepel.

Hozzunk létre másoló konstruktorral egy “bigMagic” nevű MyString típusú objektumot. Most ehhez ne deep copy-t használjunk, hanem a “bigMagic” pointer-ét állítsuk át a “my magic” tömb kezdőcímére.

Nyugodtan használhatjuk így őket minden gond nélkül egy darabig. Konkrétan addig, amíg meg nem szűnik pl. a “magic” nevű objektumunk, mivel ilyenkor fel kell szabadítani a dinamikusan foglalt memóriaterületet is, hogy ne legyen memóriaszivárgás. Ekkor viszont a “bigMagic” objektum már olyan memóriaterületre fog hivatkozni, ami nem tartozik a programunkhoz, bárkinek kioszthatja az operációs rendszer. Ez ugyebár tilos.

Na de akkor mit lehetne tenni? Ne szabadítsa fel a “magic” nevű objektum a dinamikus memóriaterületet? Akkor ki fogja? Annak kéne, aki utoljára használja, mivel akár 10 objektum is másolhatta volna. De honnan tudjuk, hogy ki az utolsó?

Na ezeket a problémákat elkerülendő használunk deep copy-t. Amikor a “bigMagic” objektumunk másoló konstruktora meghívódik, lefoglal egy **teljesen új memóriaterületet**, és átmásolja bele a “my magic” karaktertömböt. Innentől kezdve a két objektum által használt tömbök teljesen **függetlenek**. Ha az egyik objektum megszűnik, a másik a saját tömbjén nyugodtan dolgozhat, ahogy az elvárt.

Másoló konstruktor paramétere miért mindig konstans referencia?

Amikor paraméterül egyszerűen adunk át egy objektumot egy függvénynek (nem pointerként és nem referenciaként), az objektumról másolat készül, és azt kapja meg a függvény.

Ha a másoló konstruktor nem referenciaként kapná meg a paramétert, akkor ahhoz meg kellene hívni a paraméter objektum másoló konstruktorát. Ami ugyanez a függvény, tehát megint csak nem referenciaként veszi át a paramétert, ezt is le kell másolni. Végtelen másoló konstruktort hívó rekurzióba futottunk, ami elég kellemetlen.

MyClass(MyClass otherObject) {} → paraméter átvételéhez kellene a másoló konstruktor, aminek a paraméterének az átvételéhez kellene a másoló konstruktor, aminek a paraméterének az átvételéhez …

Emellett azért konstans, mert azt az objektumot, amit lemásolunk, nem szeretnénk még véletlenül sem módosítani.

Ajánlott irodalom

- http://stackoverflow.com/questions/240212/what-is-the-difference-between-new-delete-and-malloc-free