3. Gyakorlat

A new és a delete operátorok használata

A dinamikus változóknak a **new** operátorral foglalunk helyet a memóriában, megadva annak típusát.

```
int*p = new int(1);
```

A **new** operátorral létrehozott változók a **heap**-re kerülnek. Az itt létrehozott változókat dinamikus változóknak is szokás nevezni. A heap segítségével nagy szabadságra tehetünk szert, azonban ez a szabadság nagy felelősséggel is jár. A heapen nincs a lefoglalt területnek neve, így mindig szükség lesz egy mutatóra, hogy tudjunk rá hivatkozni. Ha egyszer lefoglalunk valamit a heap-en, gondoskodni kell arról, hogy felszabadítsuk. Az egyik leggyakoribb hiba a dinamikus memóriakezelésnél, ha a memóriát nem szabadítjuk fel.

Felszabadításuk nem automatikusan történik a scope-ból való kilépéskor, mint a lokális változóké, hanem a fejlesztőnek kell gondoskodni arról, hogy a lefoglalt tárterületet felszabadítsa. Amennyiben ezt nem teszi meg **memory leak**(memóriaszivárgás) lesz a programban.
A **delete** operátor felszabadítja a **new** operátor által lefoglalt tárterületet.

```
delete p;
```

A C-ben megszokott `malloc(..)` és `free(..)` műveletek függvényhívások voltak, ám a C++ban a fenti két művelet *operátor*ként van megadva. További különbség, hogy a `malloc(..)` függvény visszatérési értéke `void*` típusú volt, azaz tetszőleges memóriaterületre mutathatott. A kapott értéket egy explicit cast segítségével szokás konvertálni (C-ben ez nem szükséges, csak C++-nál). Így azonban lehetőség lenne értelmetlen foglalásokra is, mint pl:

```
int* p = malloc(sizeof(char));
```

Mivel a C/C++ szabvány csak annyit követel meg, hogy a char mérete legalább 8 bit, az int pedig minimum 16 bit legyen, ezért nincs garancia arra, hogy a hívás után a **p** pointer "elég nagy" memóriaterületre mutat.

C++-ban ezzel szemben a **new** operátor már típusozott - így az `int* p = new char;` utasítás nem fordul le, mivel a típusok nem egyeznek.

További előnye a **new** operátornak, hogy a memória foglaláson kívül az adott típus (struktúra/osztály) konstruktorát is meghívja, így egyből inicializálni is lehet az adott értéket. Ezzel párban a **delete** operátor a destruktort fogja meghívni az adott objektumon.

Abban az esetben, ha egy *n* elemű tömbnek szeretnénk helyet foglalni a **new** operátorral, jeleznünk kell a fordítónak, mégpedig a [] operátor segítségével.

```
int*pArray = new int[5];
```

ezesetben, ha a fentebb említett `delete pArray;`-t használjuk még mindig memória szivárgás lesz a programunkban. Miért? Mert a `pArray` nevű változó egy mutató, ami a tömb első elemére mutat, így egy sima `delete pArray;` nem elég az egész tömb törléséhez.

Megoldás: jelezzük a fordítónak, hogy nem csak egy elemet, hanem egy egész tömböt szeretnénk törölni. Ezt szintén a []operátor segítségével tehetjük meg.

```
delete[] pArray;
```

Kitérő: **pointer aritmetika**

Csak tömbökön értelmezett. Tömbön kívül nem definiált viselkedés! Tömbön való túlindexelés szintén nem definiált viselkedés.

```
int main()
{
      intp[] = {1,2,3};
}
```

Egy tömb adott elemére több módon is hivatkozhatunk:

*(p+3) == *(3+p) == p[3] == 3[p]

Tekintsünk egy kétdimenziós tömböt:

```
int main()
{
    int t[][3] = {{1,2,3}, {4,5,6}};
}
```

Az első [] jelek között nincs méret megadva, mert a fordító az inicializáció alapján meg tudja állapítani. A második dimenzió méretének megadása viszont kötelező.

t[1][0] == *(*(t+1)+0) == *(1[t]+0) == 0[1[t]] == 0[*(t+1)] == *(t+1)[0] == 1[t][0]

Copy constructor (másoló konstruktor)

A fordító sok kódot generál a class-unkba, struct-unkba: konstruktoron és destruktoron kívül még **másoló konstruktor**t is. A másoló konstruktor egy olyan konstruktor, melynek egyetlen paramétere egy azonos típusú objektum. Ez alapértelmezetten minden adattagot lemásol az adott adattag másoló konstruktora segítségével. Primitív típusoknál ez bitről bitre másolást jelent. Azonban, ha az osztályunk dinamikusan létrehozott adattagokat is tartalmaz a fordító által létrehozott másoló konstruktor már nem fogja tudni rendesen elvégezni a feladatát. Ugyanis nem fogja dinamikusan létrehozni az új változót csak egy mutatót ami az eredeti objektum dinamikus változójára fog mutatni. Ennek pedig az a következménye, hogy az egyik objektumból képesek leszünk egy másik objektum privát adattagját módosítani, továbbá, ha az egyik osztálynak lefut a destruktora, ami felszabadítja a dinamikus változó által foglalt tárhelyet, amikor a következő osztály destruktora is lefut szintén megpróbál felszabadítani egy már felszabadított tárhelyet ami futási idejű hiba. Ez azért van mert a fordító által létrehozott másoló konstruktor csak egy ún. sekély másolatot (**shallow copy**) fog készíteni, azaz a dinamikus foglalású változóknak nem az értéke csak a hivatkozása lesz lemásolva. Ez esetben saját másoló konstruktor írása szükséges, ami mély másolatot (**deep copy**) készít.

A copy konstruktor nem értékadó operátor!

Assignment operator (értékadó operátor)

Hasonló a helyzet, mint a másoló konstruktor esetében, az alapértelmezetten létrehozott értékadás operátor is meghívja az egyes tagok értékadó operátorait, amik a primitív típusok esetén bitről bitre másolnak. Dinamikus típusok másolásához szükség van saját értékadó operátor írására.

RAII (Resource Acquisition is Initialization)

Ha olyan struktúrát írtunk, mely gondoskodik arról, hogy minden dinamikusan lefoglalt területet felszabadít, mindent csak egyszer töröl, azt is jó sorrendben, akkor egy RAII osztályt írtunk. Ennek lényege, hogy az adott osztály a megfelelő erőforrásokat lefoglalja magának, majd a destruktor gondoskodik az erőforrások felszabadításáról. Minden erőforrást egy stack-en lévő objektumhoz kötünk, mivel azok garantáltan automatikusan fel fognak szabadulni, a destruktoruk le fog futni. Bjarne Stroustup híres mondása, hogy a C++ szemétgyűjtéssel rendelkező nyelv, mert nem generál szemetet.