# 7. előadás

## 6. előadáson nem voltam.

Dinamikus objektumok és élettartam szabályok voltak.

## Vizsgák

ETR-ben meghirdetve, NEPTUN-ba vagy át lesz importálva vagy nem.

## Tanár úr nem lesz csütörtök-pénteken.

Kurzusmail ment az érintetteknek. Engem nem érint.

## Élettartam szabályok

Volt:

* Automatikus változók
* Lokális statikusok
* Globális / névtér / osztály statikus tag
* Dinamikus objektumok

## Dinamikus objektumok ismétlés

HEAP-en allokált objektumok.

Aminek a méretét nem ismerjük fordításidőben.

Pl.: int**\*** p **=** **new** int**[**k**];**

k méretű tömböt van olyan fordító, ami engedi, de nem szabványos.

$p$ lokális változó, pointer, méretét ismerjük. Azt a névtelen tömböt,, amire mutat, nem ismerjük.

Nekünk kell garantálni, hogy fel legyen szabadítva destruktor​ral: **delete[]** p**;**

int**\*** q **=** **new** int**(**k**);** //Ez egy darab int, aminek értéke legyen k. Ezt **delete** q**;**-val töröljük.

Ha elveszik $p$, nem tudjuk felszabadítani. Memóriaszivárgás (memory leak) lép fel. Elkezd swap​pelni a winchester majd meghal a gép.

Memóriaszivárgás észlelésére, felderítésére van eszköz.

Ha kétszer próbálunk felszabadítani valamit, az Runtime​Error-ban hasal el.

Null pointer törlése nem okoz gondot.

### Dinamikus mátrix lefoglalása és felszabadítása

//k×k mátrix lefoglalása:

int**\*\*** m**=new** int**\*[**k**];**

**for(**int i**=**0**;**i**<**k**;++**i**){**

 m**[**i**]=new** int**[**k**];**

**}**

//k×k mátrix felszabadítása:

**for(**int i**=**0**;**i**<**k**;++**i**){**

 **delete[]** m**[**i**];**

**}**

**delete[]** m**;**

## Tömbelem

Tömb létrehozásakor jön létre.

Megszűnik, mikor a tömb megszűnik.

## Osztály adattag

Létrejön, mikor a tartalmazó objektum létrejön.

Megszűnik, mikor a tartalmazó objektum megszűnik.

## Temporális (ideiglenes) objektumok

Bonyolultabb kifejezés elkezd futásidőben kiértékelődni. A további kiértékelés során felhasználásra kerülnek.

x**=**a**+**b**+**c**+**d**;**↓
x**=(**a**+**b**)+**c**+**d**;**//(a+b) temporális objektumba kiértékelődik, jele T
↓
x**=(**T**+**c**)+**d**;**
↓
...

Nagy mátrixokkal való műveletek esetén ez erősen kellemetlen.

Létrejönnek a részkifejezés kiértékelésekor.

Megszűnnek, amint kiértékelődött a teljes kifejezés.

Ez hiba forrása lehet:

std**::**string a**,** b**;**

const char**\*** p**=(**a**+**b**).**c\_str**();**//Létrejön egy temporális konkatenált string (a+b) és ennek első karakterére mutató pointert (c\_str()) eltárolunk p-be. Ez csak a kifejezés végéig élt, megszűnik. (a+b) kiértékelése után.

**if(**strlen**(**p**)<**20**){**hiba!**}**

## Gyakorlás: char\* answer(char\* q);

Paraméterként kapott kérdést kiírja, vár egy válaszra a felhasználótól, majd visszaadja ezt a választ eredményként.

Felmerülő hibák:

char**\*** answer**(**char**\*** q**){**

 std**::**cout**<<**q**;**

 char ans**[**80**];**

 std**::**cin**>>**ans**;**//Ha 80-nál több karaktert írunk, további adatokat ír felül.

 **return** ans**;**//Hiba! Lokális változóra pointert visszaadni nem lehet.

**}**

Ha globális változóba kiraknánk $ans$-t, a program futásáig végig a memóriában maradna. De ez nem elegáns. Sőt, ha többször hívjuk meg, nem működik:

std**::**cout**<<**answer**(**"How are you?"**);**//Működik.

std**::**cout**<<**answer**(**"How are you?"**)<<**answer**(**"What's your name?"**);**//Nem működik a várt módon. Kétszer írja ki ugyan azt. (A második kérdésre kapott választ.)

Lokális statikus (static char ans**[**80**];**): Ugyan túléli a függvényhívást, de ugyan abba a hibába esik bele, mint a fenti globális példa.

Foglaljunk tárterületet **new**-val: char**\*** ans**=new** char**[**80**];**

Ez memory leak. Nem tudjuk meghívni rá a **delete**-t.

Trükközni lehet: char**\*** p**=**answer**(**"Leak?"**);** /\*...\*/ **delete[]** p**;** de ez előbb-utóbb el lesz felejtve.

### Mi a megoldás (C++)

Karaktertömb helyett string. Konstruktora, destruktor​a és másolója végzi el a leak-mentesítést.

std**::**string answer**(**char**\*** q**){**

 std**::**cout**<<**q**;**

 std**::**string ans**;**

 std**::**getline**(**std**::**cin**,** ans**);**

 **return** ans**;**

**}**

### Ha nincs string (C)

Bementi "eredmény paraméterbe" írjuk a választ.

## Paraméter átadás

### Érték szerinti

Új lokális változó jön létre. Információ-áramlás csak az alprogram (függvény) irányába folyik.

void f**(**int x**);**//x formális paraméter

f**(**4**);**//4 aktuális paraméter

C-ben csak érték szerinti paraméterátadás volt. (Cím szerinti átadást pointerrel oldották meg. De az a pointer érték szerint került átadásra: scanf**(**"%d"**, &**x**);**)

### Cím (referencia) szerinti

void inc**(**int**&** x**){++**x**;}**

C++-ban referenciák (álnevek) jelentek meg:

int a**=**5**;**

int**&** r**=**a**;**//referencia

r**=**7**;** → a**==**7

Ha nagy objektumot adunk át, költséges lenne lemásolni.

Sőt, lehet, hogy épp az eredeti objektumon akarunk dolgozni.

C++-ban nincs null referencia.

Nem kell pointerrel járó kódcsúnyító címlekérés, dereferálás. Nincs referencia-​arithmetika.

Kötelező inicializálni.

Speciális eset: konstans referencia. void f**(**const int**&** x**);**

### Eredmény szerinti, érték/eredmény szerinti, név szerinti, stb.

Jövő héten.

Előadás vége.