OSZTÁLY
[class]
vs.

STRUCT

Készítette: Vastag Atila

2019

Azon programozási nyelveknél, amelyeknél létezik struktúra (***struct***) és osztály (***class***) típus is, időről-időre talán felvetődik az a kellemes kérdés, hogy vajon adott helyzetben az egyiket, vagy inkább a másikat kellene-e használni? Itt most elsősorban olyan helyzetekre (feladatokra) gondolunk, amelyekhez egyébként bármelyik típus megfelelő lenne (mondjuk azért, mert mindkettőnek lehet adattagja, metódusa, konstruktora, …), de valamilyen oknál fogva nekünk mégis le kell tennünk a voksunkat az egyik mellett.

Izgalmas kérdés, vizsgáljuk meg hát a dolgot .NET környezetben.

A típusokat két nagy csoportra oszthatjuk:

- **referencia** (*reference*) típus család
  - random
  - tömbök (vektorok)
  - listák
  - rekordok
  - objektumosztályok

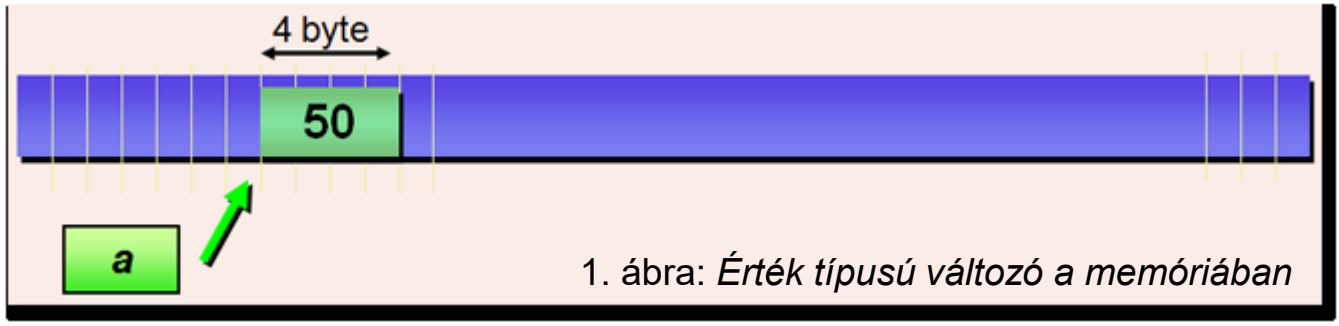
- **érték** (*value*) típus család
  - az összes egész szám típus (int, byte, ...)
  - az összes tört szám típus (double, float)
  - karakter típus (char)
  - logikai típus (bool)

Valójában, az érték típuscsaládba tartozik minden típus, ami a C# **struct** kulcsszavával előállított típus. A referencia típuscsaládba sorolhatóak a **class (**vagy **new)** kulcsszavával előállított típusok.

Érték típusok

Egy érték típusú változóról mindíg pontosan lehet tudni a memóriaigényét:

```
int a = 50;
double d = 3.2;
char c = 'x';
```



1. ábra: *Érték típusú változó a memóriában*

A memóriában az **a** változónak 4 byte foglalódik le, melyen majdan az értéket tárolni fogjuk. A program szövegében az **a** változó lényegében ezen memória címre mutat, ahol maga az érték tárolódik. Ez az oka annak, hogy ezen típuscsalád neve **érték** típusok.

Hasonló a helyzet a **d** változóval, csak az ő tárolási igénye 8 byte, a karakteré 2 byte. A **d=3.2** jelentése konkrétan: **másold be a 3.2 értéket a d változóhoz lefoglalt 8 byte területre**.

Vegyük észre, hogy az ilyen típusú változók esetén a típus neve máris mindent elárul a memóriaigényről. Az értékadó utasítás nélkül is tudni lehet a deklarációból, mi lesz a változó memóriaszükséglete:

Valamint fontos dolog, hogy ha egy változót deklarálunk, akkor a memóriában hely foglalódik neki, és amíg a változó meg nem szűnik, addig végig a memória ugyanazon a pontja tartozik hozzá. Ez érték típusú változók esetén egyszerű, hiszen pl. egy **double** számára minden esetben elég a 8 byte, az, hogy a program során többször is értéket adunk neki - nem okoz semmi problémát. Az új értékeket mindíg ugyanabba a 8 byte-ba kell bemásolni, felülírván a régi tartalmat.

Referencia típusok

A referencia típuscsalád valamely típusába tartozó változók esetén a helyzet sokkal bonyolultabb. Mennyi helyet foglal el az alábbi **t** változó?

```
int[] t = new int[20];
```

A válasz első körben egyértelműnek tűnik: 80 byte (20*4 byte).

Csakhogy tömböket úgy is lehet létrehozni, hogy a kezdőértékadás később történik meg:

```
int[] t;
//
// sok utasítás
//
t = new int[20];
```

Nyilván a fordítóprorgam nem várhatja meg, amíg a végrehajtás eléri azt a pontot, amikor már végre kiderül a tényleges memóriafoglalás (**new** utasítás sora), hanem a deklaráció pillanatában már memóriát kell foglalni a **t**-nek. Ez egyébként akkor is így van, amikor a deklaráció és a memóriafoglalás egybe van építve, ez akkor is két lépés egy sorban: a deklaráció és a memóriafoglalás.

És mennyi legyen akkor a **t** változó memóriaigénye? Hogyan számoljuk ki, amikor a deklarációból még nem derül ki a tényleges memóriaigény!

Valamint van még egy probléma: a vektorok esetén (akárcsak bármely más változók esetén) szabad magához a változóhoz más értéket rendelni futás közben. Ez tömbök esetén így néz ki:

```
int[] t = new int [20];
//
t = new int[40];
//
t = new int[10];
```

A különböző méretű vektorok memóriaigénye is más-más. Ezek az új vektorok nem feltétlenül férnek el a régebbi helyeken. De akkor mi legyen? Ugyanis a **t** változó a memória ugyanazon pontjára kell mutasson végig, ez a szabály. Nem lehet hogy minden egyes értékadás során más-más memóriacímre mutasson.

Referencia - a megoldás

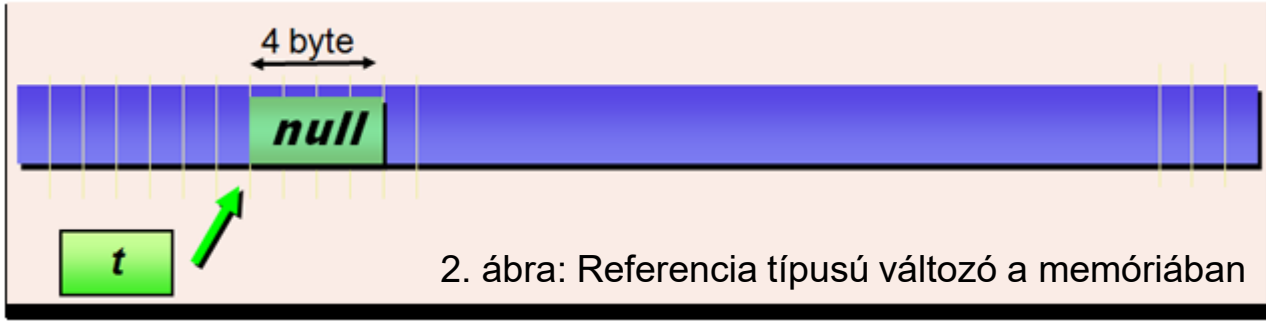
A megoldás valójában egyszerű, de nagyon fontos megérteni a megoldás működését, mert nagyon sok dologra ad magyarázatot a későbbiekben (pl. a tömbök paraméterátadását is ez a háttéműködés fogja magyarázni).

A fordítóprogram kettéválasztja a referencia típusú változók kezelését a hozzájuk tartozó tényleges adattárolástól. Két memóriaterület tartozik ténylegesen egy referencia típusú változóhoz:

- **elsődleges memóriaterület, mindig 4 byte, tartalma egy memóriacím**

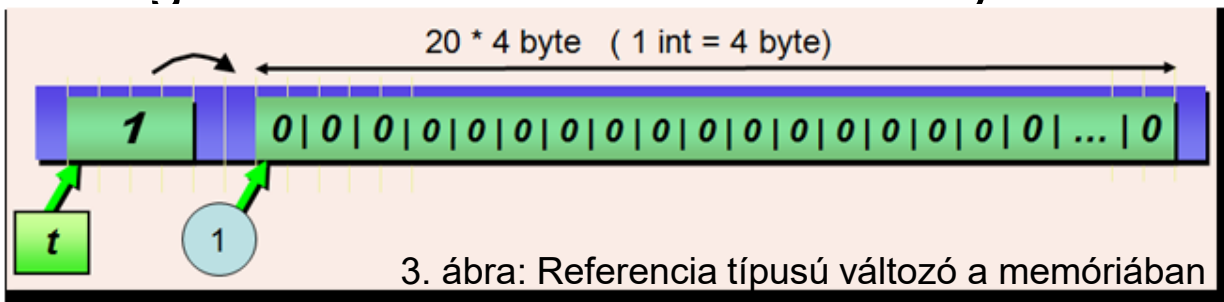
- **másodlagos memóriaterület, mérete változó, itt történik a tényleges adattárolás**

A **t** változó deklarációja során először csak 4 byte foglalódik le. Ez a 32 bites processzorok működésével magyarázható, amelyek esetén a memóriabeli címzés 32 biten írható fel, ezért a memóriacímek tárolásához elég 4 byte. Ezen a 4 byte-on kerül tárolásra későbbiekben a másodlagos memóriaterület címe.



2. ábra: Referencia típusú változó a memóriában

A **t = new int[20]** végrehajtása során kerül lefoglalásra a másodlagos memóriaterület, 80 byte. Ennek címe kerül be a **t**-hez tartozó 4 byte-ra, eképpen a **t** változó a 4 byte-t, a **t[0]** pedig a **t** változóhoz tartozó másodlagos memóriaterületen elhelyezkedő vektor 0. elemét jelöli.



3. ábra: Referencia típusú változó a memóriában

E pillanattól kezdve világos mi fog történni az alábbi utasítások hatására:

```
int[] t = new int [20];
//
t = new int[40];
//
t = new int[10];
```

A **t** változó végig ugyanott marad a memóriában, végig ugyanaz a 4 byte lesz, csak három különböző értéket tartalmaz. Először egy 20 elemű **int** vektor memóriacímét, majd valahol máshol lefoglalásra kerül egy 40 elemű vektornak hely, és azon pont memóriacíme kerül be, végül egy 10 elemű vektor memóriacíme kerül be a **t** változóhoz tartozó 4 byte területre.

Referencia kezdőértéke – null

Amikor deklarálásra kerül egy referencia típusú változó, akkor 4 byte memóriaigénye van. Amíg nem teszünk bele értéket (másodlagos memóriaterület címét) addig mi kerül tárolásra ezen a 4 byte-on? Egy speciális memóriacím, melyet **null**-nak nevezünk. Ezen **null** érték azt mutatja: **még nincs itt memóriacím**. Mi történne az alábbi utasítássorozat hatására?

```
int[] t;
t[0] = 12;
```

A válasz: fordítási hiba, a fordítóprogram is észrevenné, hogy ez a kód nem futhat le, nincs értelme.

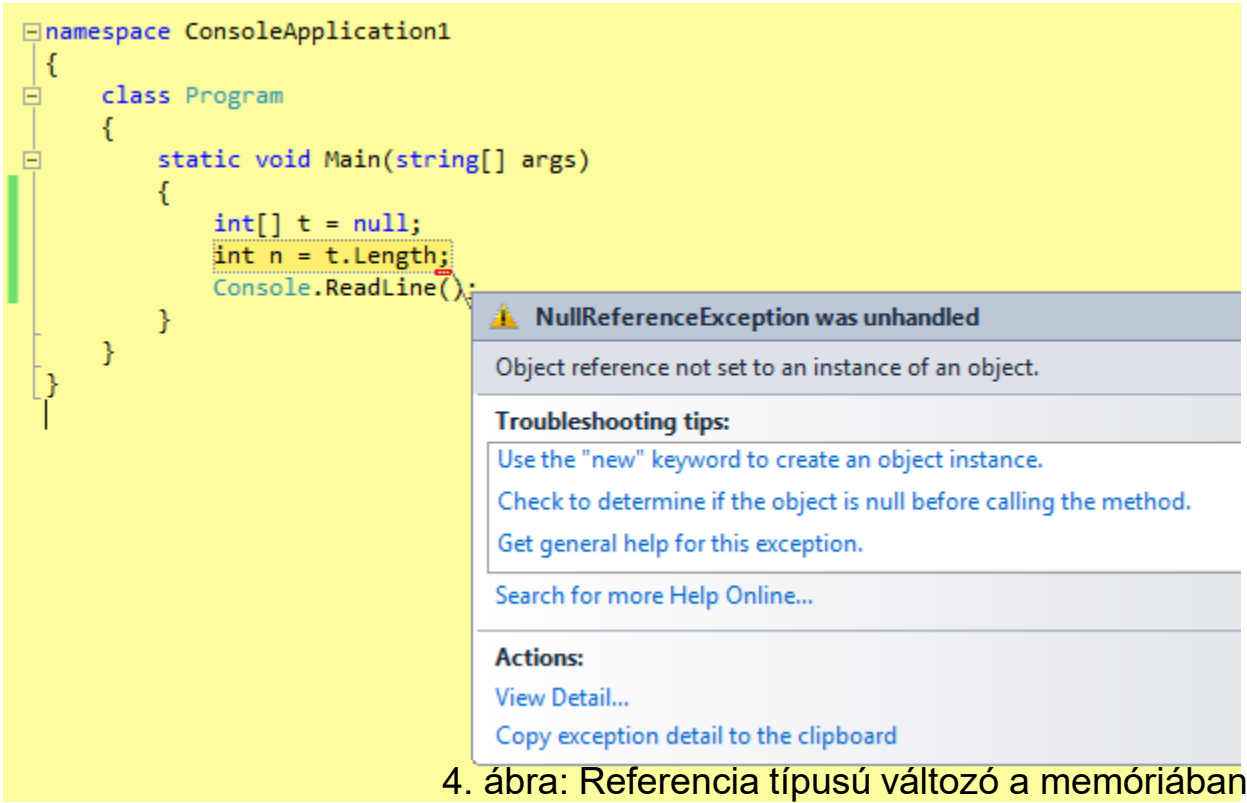
A **t** változónak van memóriaterülete, a 4 byte, de mivel még nem tettünk bele értéket, ott a **null** kezdőérték van induláskor. A **t[0]** végrehajtása azonban a másodlagos memóriaterület jelenlétét igényelni - mely nincs. Ezért ez hiba. Ezt ki lehet cselezni:

```
int[] t = new int[10];
t = null;
t[0] = 12;
```

Ekkor a fordítóprogram felületessége segíthet. A fordítóprogram azt látja, hogy vannak **t**-re vonatkozó értékadó utasítások, ezért **elhiszi**, hogy a **t[0]** időpontjára már rendben lesz **t** értéke. Persze nem lesz, hiszen a **t=null** hatására mesterségesen visszahelyezzük **t**-t a problémás állapotba, megint csak nem lesz másodlagos memóriaterület hozzárendelve. Ezért ez csak futás közben lesz problémás. A **t[0]**-ról futás közben fog kiderülni, hogy nem végrehajtható, ezért a program futása ezen a ponton le fog állni.

null reference error

Nyilván nem szabad egy **null** értékű vektor esetén lekérdezni annak **.Length** tulajdonságát. A vektor aktuális hosszának lekérdezéséhez ugyanis vektor kell. Ha egy referencia típusú változónknak **null** az aktuális értéke, akkor a vele kapcsolatos műveletek esetén **null reference exception error** hibát kapunk (*null érték esetén nem végrehajtható*).

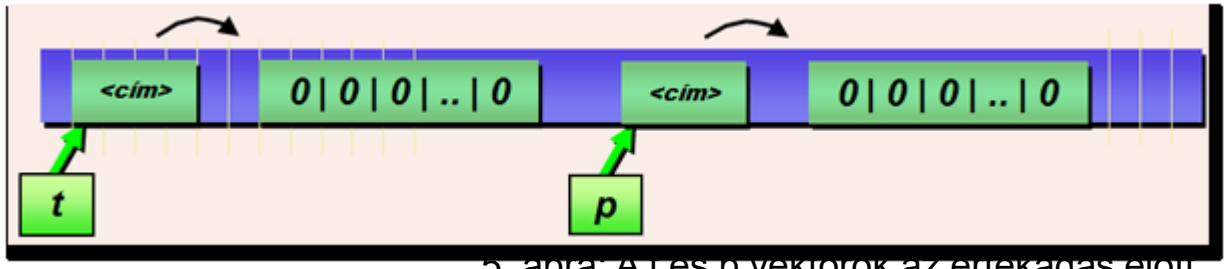


4. ábra: Referencia típusú változó a memóriában

Értékadás két referencia típusú változó között

Mi történik, ha két referencia típusú változónk között direkt értékadó utasítást hajtunk végre:

```
int[] t = new int[20];
int[] p = new int[30];
```

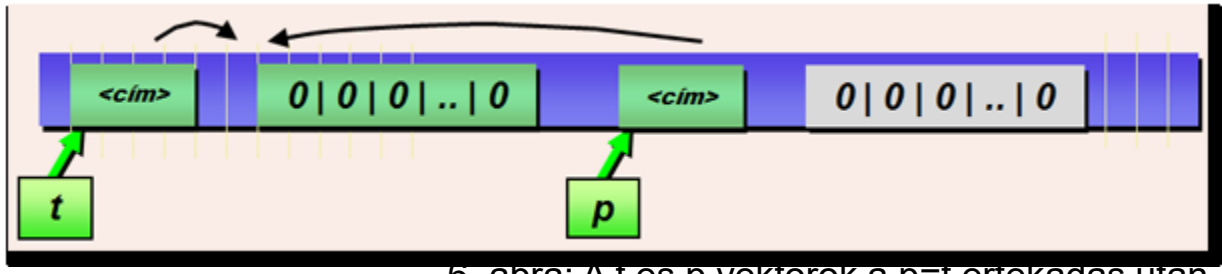


5. ábra: A t és p vektorok az értékadás előtt

```
p = t;
p[0] = 12;
Console.WriteLine("{0}",t[0]);
```

```
int[] t = new int[20];
int[] p = new int[30];
p = t;
```

A közvetlen értékadó utasítás (**p=t**) nincs hatással a másodlagos memóriaterületekre. Mivel a **p** változó valójában egy 4 byte-os memóriacím, a **t** is, így a **p=t** értelmezése nagyon egyszerű: a **t**-ben lévő 4 byteos memóriacímet másoljuk át a **p** 4 byte-os tároló helyére. Tehát 4 byte átmásolása történik. A másodlagos memóriaterületek nincsenek érintve. A **p=t** után a **p**-ben is ugyanaz a memóriacím lesz, mint a **t**-ben, így ha lekérdeznénk a **p.Length**-et, akkor ugyanúgy **20**-t kapnánk mint a **t.Length** értéke. Ha a **p[0]** elemnek értéket adunk, akkor az valójában a **t[0]** elem is.



6. ábra: A t és p vektorok a p=t értékadás után

```
p[0] = 12;
Console.WriteLine("{0}",t[0]);
```

Persze érdekes kérdés, mi történik az eredetileg **p**-hez tartozó 30 elemű, 120 byte helyigényű vektorral. Beragadt a memóriába? Hogy mi történik olyankor, mikor egy memóriaterületet elvesztünk, arra a **Garbage Collector** (röviden G.C.) működése adja meg majd a választ.

Előrebocsájtva a jó hírt: az elveszett memóriaterület automatikusan felszabadításra kerül!

Osztályok:
• Örökölhetőek
• Referens típusúak (pointer)
• A mutató lehet **null** értékű is [azaz nem foglalhat le memóriát]
• Példányonként (objektumonként) új másodlagos memórát foglal el

Strukturák:
• Nem örökölhető
• Értéktípusok (a memóriában értékként folgalnak el helyet)
• Paraméterkor értéktípusként adódnak át (mint pl. az int típusú változó)
• A mutató *nem* lehet **null** értékű [mivel nincs másodlagos memóriafoglalás]
• Példányonként (objektumonként) nincs másodlagos memória – kivétel, ha: egy objektum (referencia típus) adattagjaként jelenik meg (része az objektumnak)

Osztály és Strukturák:
• Mindkettő öszetett adatípus, mely logikailag összetartozó adatokat tartalmaz
• Mindkettő tartalmazhat függvényeket és eseményeket
• Mindkettő támogadja az interfaceket

http://aries.ektf.hu/~hz/wiki7/mprog1ea/ref_es_ertek

https://stackoverflow.com/questions/13049/whats-the-difference-between-struct-and-class-in-net

https://putabout.wordpress.com/2009/02/02/struktura-vagy-osztaly/