Eötvös Loránd Tudományegyetem
Informatikai Kar

# Objektumelvű alkalmazások fejlesztése

## 1. gyakorlat

## Dinamikus memóriakezelés

© 2011.09.22. Giachetta Roberto
groberto@inf.elte.hu
http://people.inf.elte.hu/groberto

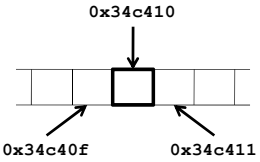
### Dinamikus memóriakezelés
**Memóriaszegmensek**

- A programok indításuk után bekerülnek a memóriába
- Az operációs rendszer lefoglal egy területet, amin elfér a teljes program, valamint elférnek a benne található értékek, ezt a memóriarészt *szegmens*nek nevezzük
- A programok csak a saját szegmensükön belül dolgozhatnak, csak ezt a memóriaterületet tudják elérni futás közben
- A szegmens mérete változhat futás közben, ha például több értéket akarunk eltárolni
- Lehetőségünk van a szegmensben található memóriahelyeket közvetlenül elérni, nem csak változóneveken keresztül
- Minden változónak tudjuk a helyét a memóriában, ez a *szegmensbeli címe*



### Dinamikus memóriakezelés
**Címzés a szegmensben**

- Képzeljük el a szegmenst, mint egy (nagyon hosszú, de véges) tömböt, ahol a cím a tömb indexe, egy hexadecimális (16-os számrendszerbeli) cím, amely megadja a szegmensen belüli elhelyezkedését (pontos bájtját) az adatnak
- Például a `0x34c410` jelentése a szegmens `34c410`-os címén található bájt, ami decimálisra átszámolva a 3 458 064. bájtot jelenti






### Dinamikus memóriakezelés
**Memóriacím lekérdezése**

- Minden változó létrehozásakor létrejön annak memóriabeli címe is
  - ezt C++-ban hasonlóan kezelhetjük, mint magát a változót
  - mivel a változó típusától függően több bájton is tárolódhat, mindig csak az első bájt címét kapjuk vissza
- Egy változó memóriacímét az `&` operátorral kérdezhetünk le, ez a *referenciaoperátor*
  - `&<változónév>` a változó első bájtjának memóriabeli címe
- Pl.: `int i = 128; cout << i << " " << &i;`
  - ekkor a kimenet lehet például:
    `128 0x22ff6c`



### Dinamikus memóriakezelés
**Műveletek referenciákkal**

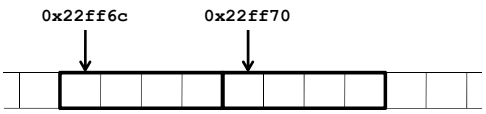
- Lehetőségünk van lépkedni a memóriában:
  - egy memóriacím is egy szám, amelyet növelhetünk, illetve csökkenthetünk (a `+`, `-`, `++`, `--` operátorokkal)
  - mivel memóriacímekkel dolgozunk, egy egyszeri növelés esetén a címérték nem eggyel fog nőni, hanem a cím a következő változó címét adja vissza
  - például egy `int` típusú változó esetében, amelyet a program 4 bájton tárol, ha eggyel megnöveljük a címet, akkor az érték 4-gyel fog nőni, 8 bájtos `double` esetén 8 bájttal nő
  - tehát a következő változó eléréséhez a változó referenciájához hozzá kell adni 1-et, a memóriában való lépkedéssel más változók értékeihez is hozzáférhetünk



### Dinamikus memóriakezelés
**Műveletek referenciákkal**

- Pl.: `int i = 128;`
  `cout << i << " " << &i << " " << &i+1;`

  Egy lehetséges kimenete:
  `128 0x22ff6c 0x22ff70`



## Dinamikus memóriakezelés
### Álnevek

- Referenciákat tekinthetjük *álnevek*nek (alias) is, azaz elérhetjük, hogy egy adott memóriaterületre több változónévvel is hivatkozhatunk.
- Egy változóhoz az álnév egy **`<típusnév>&`** típusú változó lesz.
  - miután létrehozunk egy változóhoz egy, vagy több álnevet, azok pontosan úgy fognak viselkedni, mint az eredeti változó
  - ha bármelyiket módosítjuk, akkor az összes változó értéke módosul
  - mindig kell neki kezdőértéket adni
  - pl.: `char ch; char& chr = ch;`
    `// egy karakter típusú változó memóriacíme`



## Dinamikus memóriakezelés
### Álnevek használata

- Pl.:
```
int i = 10;
int &j = i; // j ezentúl egyenértékű i-vel
int &k = j; // k ezentúl egyenértékű j-vel,
            // vagyis egyben i-vel is
i = 1000;   // i értékét beállítjuk, ezzel
   // beállítjuk minden álnevének értékét is
k = 2000;   // itt is mindhárom módosul
cout << j;  // 2000-t ír ki
```
- A későbbiek során teljesen mindegy, hogy `i`, `j`, illetve `k` melyikét írjuk le, az mind ugyanazt a változót jelenti
- Referenciák segítségével valósítjuk meg a *cím szerinti paraméterátadás*t, hiszen a formális paraméter ugyanarra a memóriaterületre fog írni, mint az aktuális paraméter



## Dinamikus memóriakezelés
### Mutatók deklarálása

- Van olyan változótípus, amelynek memóriacímet adhatunk meg értékül, ezek a *mutatók* (pointerek)
  - különböznek az álnevektől, mert önmagukban is adatok, amiket eltárolunk a memóriában, és az értékük módosítható
  - mutató létrehozásával egy új adatot viszünk a memóriába, amely egy másik adat memóriacímét tartalmazza
  - egy adatra nyilván több mutatót is ráállíthatunk
- A mutató létrehozásakor meg kell adnunk, milyen típusú változó címét fogja eltárolni, és ez onnantól nem változtatható
  - egy típushoz a hozzá tartozó mutató típus a **`<típusnév>*`**
  - mutató létrehozása: **`<típus> *<mutatónév>;`**
  - pl.: `int* ip; // egy int-re mutató pointer`



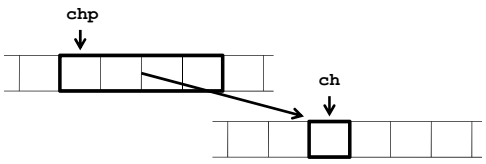
## Dinamikus memóriakezelés
### Mutatók használata

- A mutatók hasonlóan viselkednek, mint más változóink
  - értéket adhatunk nekik, élettartammal rendelkeznek
  - nem kell nekik adni kezdőértéket (ellenben az álnévvel), ekkor egy véletlenszerű címet fognak kezdetben tartalmazni
  - az értéküket lehet növelni, csökkenteni (`+`, `-`, `++`, `--`), ekkor a megfelelő memóriacímbeli objektumra ugranak
  - mutatókat nem csak egyértékű változókra, hanem tömbökre, függvényekre, rekordokra (tehát saját típusainkra) is állíthatunk
  - bárhol el lehet helyezni őket, ahol más változókat is (saját típusban mezőként, tömbelem típusaként, paraméterként, …)



## Dinamikus memóriakezelés
### Mutatók használata

- Mutató értékadására használhatjuk a referencia operátort, így ráállíthatjuk egy már létező változó memóriacímére
  - pl.:
```
char ch = 'a'; char *chp = &ch;
// a chp megkapja a ch karakter címét, innentől
// arra az adatra fog hivatkozni, amit a ch
// tartalmaz
```






## Dinamikus memóriakezelés
### Mutatók lekérdezése

- Amikor mutatók értékét kezeljük, akkor egy memóriacímet kapunk, ha az általa mutatott változó értékére vagyunk kíváncsiak, akkor ismét használnunk kell a `*` operátort
  - pl.:
```
char ch = 'a', *chp = &ch;
   // halmozott deklarálás
cout << chp;
   // lekérdezzük a chp tartalmát, azaz ch címét
   // tehát az eredmény a memóriacím
cout << *chp;
   // lekérdezzük a chp által mutatott változó
   // tartalmát, az eredmény 'a' lesz
cout << &chp;
   // lekérdezzük a chp mutató címét
```

## Dinamikus memóriakezelés
### Biztonságos használat

- A mutatók használata veszélyes, mert ha olyan mutatóra hivatkozunk, amelynek nem adtunk értéket, a program futási idejű hibát generál (*szegmenshiba*), erre mindig figyeljünk
- Nulla kezdőérték: ha nem akarunk kezdőértéket adni, és azt sem szeretnénk, hogy ez véletlenszerű legyen, akkor használhatjuk a **`NULL`** (`0`) memóriacímet
  - pl.: `int *ip = NULL; // vagy int *ip = 0;`
  - célszerű használni, mert így elkerülhetővé válik a mutató használata azelőtt, hogy valamilyen alkalmas értéket kapna
  - megfogalmazhatunk egy logikai lekérdezést:

```
if (ip)
{ /* ez az ág akkor hajtódik végre, ha ip
     nullától különböző értéket tárol*/ }
```



## Dinamikus memóriakezelés
### A referencia, mint mutató

- A referencia lényegében nem más, mint egy korlátozott felhasználású mutató, amely azonban mindig garantáltan biztonságos
- Ugyanakkor a referencia használata is kiválthat szegmenshibát, lokális változó értékének visszaadásakor, pl.:

```
int& BadFunction(){
   // cím szerint adja vissza a változót
   int value = 1;
   return value;
} // itt a lokális változó megsemmisül
// …
int val = BadFunction();
cout << val << endl;
   // szegmenshiba, a változó már nem létezik
```



## Dinamikus memóriakezelés
### Konstans mutatók és referenciák

- Referencia, illetve mutató változók is lehetnek konstansok
  - referencia esetén az érték nem módosítható:
    `<típus> const &<név> = <változó>;`
  - mutató esetén kétféle módon is korlátozhatjuk a használatot
    - lehet a mutatott érték konstans, ekkor nem változtatható a hivatkozott változó értéke, de a mutatót átállíthatjuk másik memóriacímre:
      `<típus> const *<név>;`
    - lehet a mutató konstans, ekkor nem állítható át másik memóriacímre, de a mutatott érték változtatható:
      `<típus> * const <név> = <változó>;`
    - lehet a mutató és a mutatott érték is konstans:
      `<típus> const * const <név>;`



## Dinamikus memóriakezelés
### Konstans mutatók és referenciák

- Pl:

```
double d1 = 10, d2 = 50;
double const &d1r = d1; // referencia konstansra
double const * d1p1 = &d1; // mutató konstansra
double * const d1p2 = &d1; // konstans mutató
double const * const d1p3 = &d1;
   // konstans mutató konstans értékre
d1r = 100; // HIBA, az érték nem módosítható
*d1p1 = 50; // HIBA, az érték nem módosítható
*d1p2 = 50; // az érték módosítható
*d1p3 = 50; // HIBA
d1p1 = &d2; // átállíthatjuk más memóriacímre
d1p2 = &d2; // HIBA, a mutató nem állítható át
d1p3 = &d2; // HIBA
```



## Dinamikus memóriakezelés
### Mutatóra állított mutatók és referenciák

- Mivel a mutatók is értékek a memóriában, rájuk is lehet mutatót állítani
  - ekkor jeleznünk kell, hogy a mutató célja is mutató, azaz halmoznunk kell a `*` jelet
  - tetszőleges szintig lehet mutatókat megcímezni ilyen módon
  - pl.:

```
int value = 0; int *intp = &value;
int **intpp = &intp; // mutatóra állított mutató
cout << **intpp; // kiírja value értékét
```

- Hasonlóan referencia is állítható mutatóra, így a mutató is használható cím szerinti paraméterátadáskor, pl.:

```
int *&intpref = intp;
cout << *intpref; // kiírja value értékét
```



## Dinamikus memóriakezelés
### Memóriafoglalási lehetőségek

- Memóriahelyeket kétféleképpen foglalhatunk le:
  - *automatikusan*: változó létrehozásakor lefoglalódik hozzá egy memóriahely is, ezt nem befolyásolhatjuk
  - *manuálisan* (*dinamikusan*): lehetőségünk van explicit megadni a kódban, hogy lefoglalunk egy a memóriahelyet
    - ehhez a **`new`** operátort használjuk, és meg kell adnunk a típusát is, pl. `new double;`
    - a létrehozás visszaad egy memóriacímet, amelyet a helyfoglalás megkapott (illetve annak az első bájtját)
- A lefoglalással visszakapott memóriacímet megkaphatja egy mutató, így később tudunk hivatkozni arra a címre
  - pl.: `int *ip = new int;`

## Dinamikus memóriakezelés
### Dinamikus memóriafoglalás

- Így szétválaszthatjuk a változó deklarációját a hozzá tartozó memóriaterület lefoglalásától:

```
int *ip; // ekkor i még csak egy mutató
ip = new int; // új memóriaterület a mutatónak
```

- Ekkor két hely kerül lefoglalásra a memóriában, egy a mutatónak, egy az értéknek
- Többször is lefoglalhatunk helyet egy mutatónak, pl.:

```
int *ip = new int;
ip = new int;
ip = new int;
```

- Új memóriaterület foglalásakor a régi memóriaterület is bent marad a szegmensben, de a mutatón keresztül már nem elérhető (de memóriaműveletekkel igen)



## Dinamikus memóriakezelés
### Memóriahely felszabadítás

- Ahogy lefoglalunk, úgy lehetőségünk van törölni is memóriahelyet programunkban
  - az automatikusan lefoglalt memória törlését a program magától végzi, ezt nem befolyásoljuk
  - a manuálisan létrehozott memóriahelyeket nekünk kell törölnünk, vagy a program végéig a memóriában maradnak
  - a törlésre a **delete** operátor szolgál
  - pl.:

```
float* flp = 0; // flp nem hivatkozik semmire
flp = new float;
   // flp már hivatkozik egy memóriaterületre
delete flp; flp = 0;
   // flp ismét nem hivatkozik semmire
```



## Dinamikus memóriakezelés
### Biztonságos dinamikus helyfoglalás

- Minden **new** operátornak kell rendelkeznie egy **delete** párral, azaz a dinamikusan létrehozott változókat törölni is kell
- A nem törölt változók használatuk után is foglalják a memóriát, bár már nincs mutató rájuk állítva, az ilyen területeket nevezzük *memóriaszemét*nek, pl.:

```
int *ip = new int;
ip = new int;
   // az előző terület memóriaszemét lesz
```

- A memóriaszemét különösen veszélyes, ha ciklikusan kerül lefoglalásra
- Ha két mutató hivatkozik ugyanarra a területre, akkor csak az egyiket töröljük, de a másikkal se hivatkozzuk a későbbiekben a változóra (különben szegmenshiba lép fel)



## Dinamikus memóriakezelés
### Többszörös dinamikus foglalás

- Egyszerre több memóriahelyet is lefoglalhatunk azonos típusból, ekkor azok egymás után helyezkednek el a memóriában, pl.:

```
int *ip = new int[5];
// öt memóriahely lefoglalása
```

  - A törléshez **delete** operátornak jelölnünk kell, hogy több helyről van szó a **[]** operátorral, pl.:

```
delete[] ip;
```

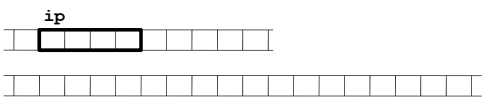
  - ha véletlenül lefelejtjük a tömb jelölést, akkor csak az első érték törlődik, a többi a memóriában marad
  - a törlés után a mutató továbbra is használható, de az értékek elvesznek



## Dinamikus memóriakezelés
### Példa

- Pl.:

```
int *ip = NULL; // mutató létrehozása
```

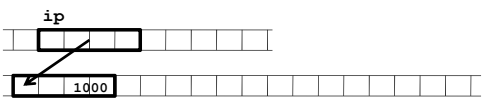





## Dinamikus memóriakezelés
### Példa

- Pl.:

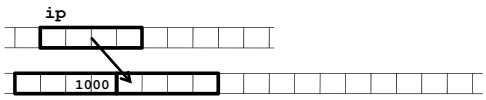
```
int *ip = NULL; // mutató létrehozása
ip = new int; *ip = 1000; // új érték
```



## Dinamikus memóriakezelés
**Példa**

- Pl.:

```
int *ip = NULL; // mutató létrehozása
ip = new int; *ip = 1000; // új érték
ip = new int; // új érték, az előző megmarad
```

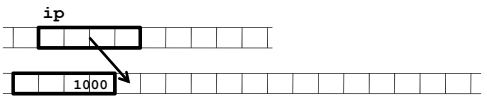





## Dinamikus memóriakezelés
**Példa**

- Pl.:

```
int *ip = NULL; // mutató létrehozása
ip = new int; *ip = 1000; // új érték
ip = new int; // új érték, az előzőből
              // memóriaszemét lesz
delete ip; // memóriahely törlése,
           // ip-ben megmarad a cím
```

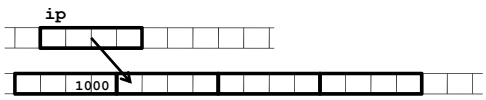





## Dinamikus memóriakezelés
**Példa**

- Pl.:

```
int *ip = NULL; // mutató létrehozása
ip = new int; *ip = 1000; // új érték
ip = new int; // új érték, az előzőből
              // memóriaszemét lesz
delete ip; // memóriahely törlése,
           // ip-ben megmarad a cím
ip = new int[3]; // 3 memóriahely foglalása
```

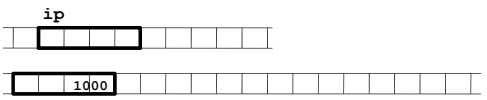





## Dinamikus memóriakezelés
**Példa**

- Pl.:

```
int *ip = NULL; // mutató létrehozása
ip = new int; *ip = 1000; // új érték
ip = new int; // új érték, az előzőből
              // memóriaszemét lesz
delete ip; // memóriahely törlése,
           // ip-ben megmarad a cím
ip = new int[3]; // 3 memóriahely foglalása
delete[] ip; ip = 0; // törlés és kinullázás
```






## Dinamikus memóriakezelés
**A primitív dinamikus tömb**

- A többszöri memóriafoglalással lényegében egy tömböt hozhatunk létre, amely a primitív tömb dinamikus megfelelője
  - az elemei elérhetőek a `[ ]` operátorral, 0-tól indexelve
  - működése lényegében megegyezik a statikus dinamikus tömbével, de paraméterben megadható változó is méretnek
  - pl.:

```
int size; cin >> size;
int* array = new int[size];
   // a beolvasott méretű lesz a tömb
for (int i = 0; i < size; i++)
   cin >> array[i]; // elemek bekérése
// …
delete[] array; // tömb törlése
```



## Dinamikus memóriakezelés
**Tömbelem címzés**

- A tömbelem címzés nem más, mint a memóriában való címmódosítás, hiszen a memória egyes címei a + operátorral is elérhetőek
  - azaz `a[i]` leírható `*(a+i)` formában is, hiszen a tömb címével a változó mennyiséggel arrébb lévő memóriacím értékét akarjuk kiolvasni
  - ez az oka, hogy mindent 0-tól indexelünk, mivel azt fejezzük, mennyivel lépjünk a kezdőcímhez képest a memóriában
  - mivel az összeadás kommutatív, ezért az index és a tömbnév fel is cserélhető a kifejezésben, pl.:

```
float* a = new float[10];
cin >> 5[a]; // ugyanaz, mint a[5]
```

## Dinamikus memóriakezelés
### Többdimenziós tömbök

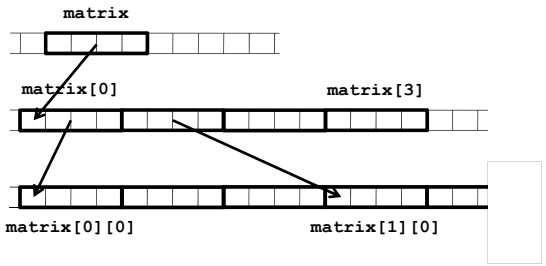
- Lehetőségünk van többdimenziós tömbök létrehozására is a tömbök tömbje módszerrel
  - azaz mutatókra mutatót állítunk, így a külső tömbünk fogja tartalmazni a mutatókat, amelyek a mátrix soraira hivatkoznak
  - létrehozzuk a mutatókat tároló tömböt, majd utána mindegyikre felfűzzük az értékeket tároló tömböt, tehát egy ciklusra van szükségünk, pl.:

```
float** matrix = new float*[4];
// 4 sora lesz a mátrixnak
for (int i = 0; i < 4; i++)
   matrix[i] = new float[3];
   // 3 oszlopa lesz a mátrixnak
```



## Dinamikus memóriakezelés
### Többdimenziós tömbök

- a mátrix megjelenése a memóriában:






## Dinamikus memóriakezelés
### Többdimenziós tömbök

- a létrehozást követően az indexelés és a sorok hozzáférése a megszokott módon történik, pl.:

```
cin >> matrix[3][2]; // elem bekérése
cout << **matrix; // mátrix 1. sorának 1. eleme
float* row = matrix[3];
   // sor átadása egy mutatónak
```

- törléskor külön kell törölnünk minden sort, majd a mutatókat tartalmazó tömböt, pl.:

```
for (int i = 0; i < 4; i++)
   delete[] matrix[i]; // sorok törlése
delete[] matrix; // mutatók törlése
```

- ugyanez megvalósítható magasabb dimenziókban is, pl.:

```
float*** m3d = new float**[4]; // …
```



## Dinamikus memóriakezelés
### Mutató, mint bejáró

- A mutatók használhatóak tömbök, vagy más adatszerkezetek bejárására is, így nem csupán indexeléssel férhetünk hozzá az adatokhoz, pl.:

```
int *array = new int[10];
for (int* p = array; p != array + 10; p++)
    // mutató használata indexelés helyett,
    // ugyanúgy 10 lépést teszünk meg
    cin >> *p; // tömb eleminek feltöltése

// ugyanez rövidebben:
int *array = new int[10], *p = array;
while (p != array + 10)
   cin >> *p++;
   // a léptetést a beolvasással együtt végezzük
```



## Dinamikus memóriakezelés
### Memóriaterületek

- A programok a használat szempontjából három területet különböztetnek meg:
  - *globális terület* (*global*): konstansok és globális változók, amelyek a program futása során mindig jelen vannak
  - *verem* (*stack*): a lokális változók, amelyeket automatikusan hoztunk létre
    - működésében olyan, mint egy verem, mert mindig az utolsó blokkban létrehozott változó törlődik elsőként a blokk végeztével
  - *kupac* (*heap*): a manuálisan lefoglalható memóriaterület, általában a legnagyobb részét képezi a szegmensnek
    - a tömbök és szövegek is ide kerülnek



## Dinamikus memóriakezelés
### Saját típusok dinamikus kezelése

- Saját típusainkat is létrehozhatjuk, illetve törölhetjük dinamikusan:

```
<típusnév> *<mutatónév> = new <típusnév>;
delete <mutatónév>;
```

  - amennyiben a saját típusunk konstruktorparaméterekkel rendelkezik, azokat meg kell adnunk a létrehozáskor (kivéve ha van 0 paraméteres konstruktor):

```
<mutatónév> = new <típusnév>(<paraméterek>);
```

- Továbbra is lehetőségünk van hivatkozni a típusunk adattagjaira `(*<mutatónév>).<mezőnév>` formában
  - a zárójel az operátor precedencia miatt kell
  - mivel ez elég összetett jelölés, lehet egyszerűsíteni a `->` operátorral: `<mutatónév>-><mezőnév>`

## Dinamikus memóriakezelés
### Saját típusok dinamikus kezelése

- Pl.:

```
struct Demo{
   int Value;
   void Print() { cout << Value; }
   Demo() { Value = 0; } // konstruktorok
   Demo(int v) { Value = v; }
};
// …
Demo *d1, *d2; // mutató létrehozása
d1 = new Demo; // példányosítás konstruktorral
d1->Print(); // 0, ugyanez: (*d1).Print()
d2 = new Demo(10); // paraméteres konstruktorral
d2->Print(); // 10
delete d1; delete d2; // törlések
```



## Dinamikus memóriakezelés
### Saját típusok dinamikus kezelése

- Mivel a típusainkban is elhelyezhetünk mutatókat (esetleg más saját típusra is), ez a hivatkozás halmozódhat
  - pl.:

```
struct Demo{ int Value; };
struct AnotherDemo{
   Demo* Pointer; // mutató saját típusra
};
// …
AnotherDemo* ad = new AnotherDemo; // új érték
ad->Pointer = new Demo; // új érték
   // ugyanez: (*ad).Pointer = new Demo;
ad-> Pointer->Value = 0;
   // egyik mutatón keresztül hivatkozhatunk egy
   // másik mutató által hivatkozott értékre
```



## Dinamikus memóriakezelés
### Előnyök

- A változók élettartama független a rájuk állított mutatótól, így a példányokat nem kötik a blokkok, tetszőleges ponton létrehozhatóak és törölhetőek a programban
- Paraméterátadásnál, amennyiben referenciákat, vagy mutatókat adunk át, nem másolódik le a teljes érték, hanem csak a memóriacíme, így nem foglalunk le feleslegesen memóriát a foglalással
  - ez különösen összetett típusokra, vagy tömbökre érvényes, hiszen rájuk is ráállíthatóak mutatók, pl.:

```
vector<Demo> vect;
vector<Demo>* vectp = &vect; // mutató vektorra
vectp->push_back(Demo());
   // elérés a mutatón keresztül
```



## Dinamikus memóriakezelés
### Előnyök

- Pl. ha egy rekord a memóriában 10kbyte helyet foglal, és van 1000 rekordunk, akiket egy vektorban tárolunk, és ezeket mind átmásolnánk a memóriában:
  - a `vector<Demo>` másolása közel 1000 * 10 kbyte = 10Mbyte plusz memóriát igényel
  - a `vector<Demo*>` másolása 1000 * 4 byte = 4 kbyte plusz memóriát igényel
  - `vector<Demo>*` (azaz a teljes vektorra egy mutató ráállítása) csak 4 byte plusz memóriát igényel