Eötvös Loránd Tudományegyetem
Informatikai Kar

Alkalmazott modul: Programozás

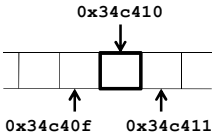
8. előadás

Strukturált programozás: dinamikus memóriakezelés

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

Dinamikus memóriakezelés
Memóriaszegmensek

- Az operációs rendszer minden futó program számára fenntart egy területet a memóriából, ezt nevezzük *memóriaszegmens*nek
  - minden program a saját szegmensében dolgozik, a szegmens mérete változhat futás közben
  - minden szegmensbeli memóriahely (bájt) rendelkezik egy sorszámmal, ez a *szegmensbeli memóriacíme*, amelyen keresztül elérhető a programban
- A memória(szegmens) tekinthető egy vektornak, a memóriacím pedig annak egy indexe (általában hexadecimálisan adjuk meg, pl. `0x34c410`)



Dinamikus memóriakezelés
Memóriacím lekérdezése

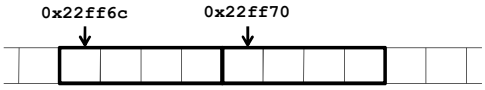
- Minden változó rendelkezik memóriacímmel
  - ezt C++-ban hasonlóan kezelhetjük, mint magát a változót
  - mivel a változó típusától függően több bájton is tárolódhat, mindig csak az első bájt címét kapjuk vissza
- Egy változó memóriacímét az `&` operátorral kérdezhetjük le, ez a *referenciaoperátor*, `&<változónév>` a változó első bájtjának memóriabeli címe
- Pl.:

```
int i = 128;
cout << i << " " << &i;
// lehetséges eredmény: 128 0x22ff6c
```


Dinamikus memóriakezelés
Műveletek memóriacímekkel

- Lehetőségünk van a memóriában történő „ugrásra”
  - a memóriacímet számként kezelhetjük, növelhetjük, illetve csökkenthetjük (a `+`, `-`, `++`, `--` operátorokkal)
  - azonban egy egyszeri növelés esetén a címérték nem eggyel fog nőni, hanem a következő változó címét adja vissza
- Pl.:

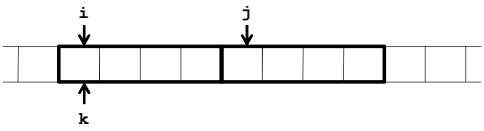
```
cout << i << " " << &i << " " << &i+1;
// lehetséges eredmény: 128 0x22ff6c 0x22ff70
```



Dinamikus memóriakezelés
Referencia változók

- A *referencia változók* (vagy álnevek) olyan változók, amelyek nem a változók értékét, hanem memóriacíműket másolják át, így ugyanarra a területre mutatnak a memóriában
- Pl.:

```
int i = 128;
int j = i; // egyszerű változó
int& k = i; // referencia változó
```



Dinamikus memóriakezelés
Mutatók deklarálása

- Egy még speciálisabb változótípus a *mutató* (*pointer*), amely memóriacímet tárol értékként
  - mutató létrehozásával egy új adatot viszünk a memóriába, amely másik adat memóriacímét tartalmazza
  - általánosabb célú, mint a referencia
- A mutató létrehozásakor meg kell adnunk, milyen típusú változó címét fogja eltárolni
  - egy típushoz a hozzá tartozó mutató típus a `<típusnév>*`
  - mutató létrehozása: `<típus> *<mutatónév>;`
  - pl.: `int* ip; // egy int-re mutató pointer`

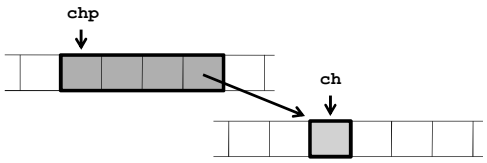
Dinamikus memóriakezelés
Mutatók használata

- A mutatók hasonlóan viselkednek, mint más változóink
  - értéket adhatunk nekik, élettartammal rendelkeznek
  - az értéküket lehet növelni, csökkenteni (`+`, `-`, `++`, `--`), ekkor a megfelelő memóriacímbeli objektumra ugranak
  - mutatókat nem csak változókra, hanem tömbökre és alprogramokra állíthatunk
  - a referenciaváltozókkal ellentétben nem kell nekik adni kezdőértéket
- A mutató mérete rögzített minden típusra, 32 bites architektúrában 4 byte (emiatt csak 4GB memória címezhető meg), 64 bites architektúrában 8 byte

Dinamikus memóriakezelés
Mutatók használata

- Mutató értékadására használhatjuk a referencia operátort, így ráállíthatjuk egy már létező változó memóriacímére, pl.:

```
char ch = 'a';
char *chp = &ch;
// lekérdezzük ch memóriacímét, amit értékül
// adunk chp mutatónak, innentől rámutat
```



Dinamikus memóriakezelés
Mutatók lekérdezése

- Amikor mutatók értékét kezeljük, akkor egy memóriacímet kapunk, amennyiben az általa mutatott változó értékére vagyunk kíváncsiak, akkor használnunk kell a `*` operátort, pl.:

```
char ch = 'a', *chp = &ch;
   // deklarálunk egy karakter változót és egy
   // mutatót
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

- Mutatók használata veszélyes lehet, ha olyan mutatóra hivatkozunk, amely nem mutat megfelelő helyre (pl. nem adtunk neki értéket, vagy amit adtunk, már megsemmisült), ekkor *szegmenshibá*t kapunk (futási időben)
- Célszerű a mutatónak létrehozáskor a nulla (`NULL`, `0`) értéket adni, mert így utólag ellenőrizhető lesz
  - pl.:

```
int *ip = NULL; // vagy int *ip = 0;
…
if (ip){
  /* ez az ág akkor hajtódik végre, ha ip
     nullától különböző értéket tárol */
}
```


Dinamikus memóriakezelés
Mutató, mint bejáró

- A mutatókat ráállíthatjuk tömbre is (pontosabban az első elemére), illetve használhatjuk tömbök bejárására is, így nem csupán indexeléssel férhetünk hozzá az adatokhoz
  - pl.:

```
int array[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[10], *p = array;
while (p != array + 10) cin >> *p++;
```


Dinamikus memóriakezelés
Memóriafoglalási lehetőségek

- Memóriahelyeket két módon foglalhatunk le (allokálhatunk):
  - *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 típust is, pl. `new double;`
    - a létrehozás visszaad egy memóriacímet, amelyen a változó elhelyezkedik
- A lefoglalással visszakapott memóriacímet megkaphatja egy mutató, pl.: `int *ip = new int;`

Dinamikus memóriakezelés
Memóriafoglalási lehetőségek

- szétválaszthatjuk a mutató deklarációját a hozzá tartozó memóriaterület lefoglalásától, pl.:
```
int *ip; // ekkor i még csak egy mutató
ip = new int; // új memóriaterület a mutatónak
```
- két hely kerül lefoglalásra a memóriában, egy a mutatónak, egy az értéknek
- egy mutató számára többször is lefoglalhatunk helyet, pl.:
```
int *ip = new int;
ip = new int; ip = new int;
```
- új memóriaterület foglalásakor a régi memóriaterület is bent marad a szegmensben, viszont a mutatón keresztül már nem lesz elérhető (de memóriaműveletekkel igen)

Dinamikus memóriakezelés
Memóriaterületek

- A programok a használat szempontjából három memóriaterületet 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 változók törlődnek elsőként a blokkból való kilépéssel
  - *kupac* (*heap*): a manuálisan lefoglalható memóriaterület, általában a legnagyobb részét képezi a szegmensnek
    - a tömbök és a szövegek is ide kerülnek

Dinamikus memóriakezelés
Memóriahely felszabadítás

- Ahogy lefoglalunk, úgy lehetőségünk van törölni is memóriahelyet a 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;
   // manuálisan lefoglaljuk a helyet
delete flp; // töröljük a lefoglalt helyet
```


Dinamikus memóriakezelés
Biztonságos dinamikus helyfoglalás

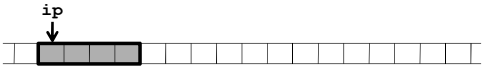
- Minden `new` operátornak kell rendelkeznie egy `delete` párral, azaz a dinamikusan létrehozott változókat törölni is kell
- A nem törölt, dinamikusan lefoglalt változók a mutató törlését követően is a memóriában maradnak, az ilyen területeket nevezzük *memóriaszemét*nek, pl.:
```
int *ip = new int;
// dinamikusan lefoglaltuk a memóriaterületet
ip = new int;
// ekkor az előző terület memóriaszemét lesz
```
- Sosem a mutatót, csak a dinamikusan lefoglalt területet kell manuálisan törölnünk (ha több mutató hivatkozik ugyanarra a területre, elég egyszer elvégeznünk a törlést)

Dinamikus memóriakezelés
Többszörös dinamikus foglalás

- Egyszerre több memóriahelyet is lefoglalhatunk azonos típusból a `[]` operátorral, ekkor azok egymás után helyezkednek el a memóriában, pl.:
```
int *ip = new int[5];
// öt memóriahely lefoglalása
```
  - törlésnél a `delete` operátornak jelölnünk kell, hogy több helyről van szó, szintén a `[]` operátorral, pl.:
```
delete[] ip;
```
  - ha törlésnél elfelejtjük a 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

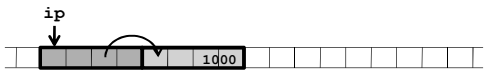
- Pl.:

```
int *ip = NULL; // mutató létrehozása
ip = new int; // memóriahely foglalás
```

Dinamikus memóriakezelés
Példa

- Pl.:

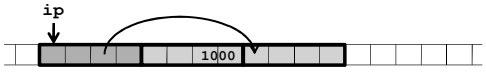
```
int *ip = NULL; // mutató létrehozása
ip = new int; // memóriahely foglalás
*ip = 1000; // érték beállítása
```



Dinamikus memóriakezelés
Példa

- Pl.:

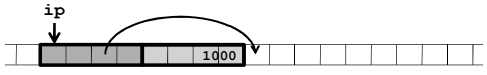
```
int *ip = NULL; // mutató létrehozása
ip = new int; // memóriahely foglalás
*ip = 1000; // érték beállítása
ip = new int;
   // új foglalás, az előzőből memóriaszemét lesz
```



Dinamikus memóriakezelés
Példa

- Pl.:

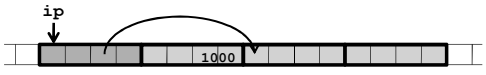
```
int *ip = NULL; // mutató létrehozása
ip = new int; // memóriahely foglalás
*ip = 1000; // érték beállítása
ip = new int;
   // új foglalás, az előzőből memóriaszemét lesz
delete ip;
   // memóriahely törlése, ip-ben megmarad a cím
```



Dinamikus memóriakezelés
Példa

- Pl.:

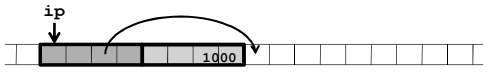
```
int *ip = NULL; // mutató létrehozása
ip = new int; // memóriahely foglalás
*ip = 1000; // érték beállítása
ip = new int;
   // új foglalás, az előzőből memóriaszemét lesz
delete ip;
   // memóriahely törlése, ip-ben megmarad a cím
ip = new int[2]; // 2 memóriahely foglalása
```



Dinamikus memóriakezelés
Példa

- Pl.:

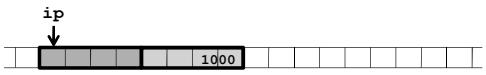
```
int *ip = NULL; // mutató létrehozása
ip = new int; // memóriahely foglalás
*ip = 1000; // érték beállítása
ip = new int;
   // új foglalás, az előzőből memóriaszemét lesz
delete ip;
   // memóriahely törlése, ip-ben megmarad a cím
ip = new int[2]; // 2 memóriahely foglalása
delete[] ip; // törlés
```



Dinamikus memóriakezelés
Példa

- Pl.:

```
int *ip = NULL; // mutató létrehozása
ip = new int; // memóriahely foglalás
*ip = 1000; // érték beállítása
ip = new int;
   // új foglalás, az előzőből memóriaszemét lesz
delete ip;
   // memóriahely törlése, ip-ben megmarad a cím
ip = new int[2]; // 2 memóriahely foglalása
delete[] ip; // törlés
ip = NULL; // újbóli kinullázás
```



Dinamikus memóriakezelés
A primitív dinamikus tömb

- A többszöri memóriafoglalással tulajdonképpen 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]; // tömb lefoglalása
for (int i = 0; i < size; i++)
   cin >> array[i]; // elemek bekérése
…
delete[] array; // tömb törlése
```


Dinamikus memóriakezelés
Tömbelem címzés

- A tömbelem indexelés igazából a memóriában való címelérés, és egyenértékű a `+` operátor használatával
  - azaz `a[i]` leírható `*(a+i)` formában is, vagyis a tömb kezdőcímétől (`a`) szeretnénk továbblépni indexnyi (`i`) helyet a memóriában, és az ottani értéket lekérdezni
  - ezért indexelünk 0-tól, mivel a tömb kezdőcíme egyben az első elem címe
  - pl.:

```
float* a = new float[10];
cin >> a[5];
// ugyanez: cin >> *(a+5);
// ugyanez: cin >> 5[a]; (kommutativitás miatt)
```


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 mutató átállítható:
      `<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; // konstans referencia
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, rájuk is lehet mutatót állítani
  - ekkor jeleznünk kell, hogy a mutató célja is mutató, azaz halmoznunk kell a `*` jelet
  - 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
```

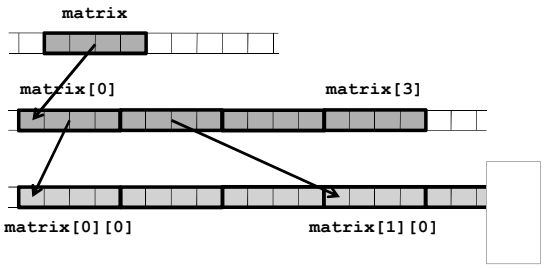
Dinamikus memóriakezelés
Többdimenziós tömbök

- Ezzel a módszerrel lehetőségünk van többdimenziós tömbök (mátrixok) létrehozására is
  - tömbök tömbjeként, 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*[width];
// 4 sora lesz a mátrixnak
for (int i = 0; i < width; i++)
   matrix[i] = new float[height];
   // 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 < width; i++)
   delete[] matrix[i]; // sorok törlése
delete[] matrix; // mutatók törlése
```

- ugyanez megvalósítható magasabb dimenziókban is, pl.:

```
float*** m3d = new float**[width];
```


Dinamikus memóriakezelés
Dinamikusan foglalt mezők

- Természetesen típusok mezői is lehetnek mutatók, és allokálhatunk nekik dinamikusan memóriaterületet
  - ezt általában a konstruktorban végezzük
  - de az így létrehozott értékeket manuálisan kell törölni, különben a példány megsemmisülése után is megmarad
  - a törlést akkor kell elvégezni, amikor a példány törlődik
- A típuspéldány megsemmisítéséért felelős műveletet nevezzük *destruktor*nak
  - a destruktor automatikusan lefut, amikor törlődik a változó (lokális változó esetén a blokk végére érünk, dinamikus létrehozás esetén meghívjuk a `delete` műveletet)

Dinamikus memóriakezelés
Destruktor

- A destruktorban olyan utasításokat tárolunk, amelyek „kitakarítják” az általunk használt memóriaterületet
  - csak a dinamikusan foglalt mezőket kell törölnünk
  - ha nincs dinamikusan foglalt mező, akkor nem szükséges
- A destruktor a `~<típusnév>` nevet kapja, mindig publikus, nincs típusa, nincs paramétere, ezért nem túlterhelhető:

```
class <típus> {
public:
   <typusnév>() { … } // konstruktor
   ~<típusnév>() { … } // destruktor
   …
};
```


Dinamikus memóriakezelés
Destruktor

- Pl.:

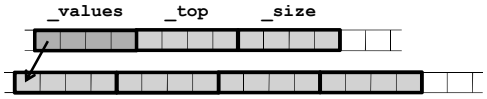
```
struct MyType {
   MyType(){ cout << "Hello!" << endl; }
   ~MyType(){ cout << "Byebye!" << endl; }
};

int main(){
   MyType mt; // itt fut le a konstruktor
   return 0;
} // itt fut le a destruktor

// eredménye:
// Hello!
// Byebye!
```

Dinamikus memóriakezelés
Példa

Feladat: Módosítsuk úgy a verem típust, hogy paraméterben lehessen megadni a maximális elemszámot.

- dinamikus tömbre van szükségünk (`T* _values`), amely méretét a konstruktorparaméter adja meg, amit szintén eltárolunk (`int _size`)
- a törlés miatt szükségünk lesz destruktor műveletre is

Tervezés:



Dinamikus memóriakezelés
Példa

Megoldás:

```
template <class T> // elemtípus sablonja
class Stack { // verem típus
private:
   T* _values; // értékek mutatója
   int _top; // a tetőelem indexe
   int _size; // méret eltárolása
public:
   enum Exceptions { BAD_SIZE, STACK_FULL,
                     STACK_EMPTY };
      // kivételek felsorolási típusa
   …
};
```


Dinamikus memóriakezelés
Példa

Megoldás:

```
template <class T>
Stack<T>::Stack(int size) {
   if (size <= 0) // ellenőrizzük a paramétert
      throw BAD_SIZE;
   _values = new T[size];
      // tömb dinamikus létrehozása
   _top = 0; // kezdetben 0 elem van a veremben
   _size = size;
}
template <class T>
Stack<T>::~Stack() {
   delete[] _values; // dinamikus tömb törlése
}
```