Pázmány Péter Katolikus Egyetem
Információs Technológiai és Bionikai Kar

Bevezetés a Programozásba II

11. előadás

Adatszerkezetek megvalósítása

© 2014.05.12. Giachetta Roberto
groberto@inf.elte.hu
http://people.inf.elte.hu/groberto

Adatszerkezetek megvalósítása
Adatszerkezetek

- *Adatszerkezet*nek nevezzük adott típusú adatok valamilyen megadott struktúrában felépített halmazát
- A főbb adatszerkezetek a legtöbb programozási nyelvben definiáltak, ezeket *gyűjtemények*nek (*collection*) nevezzük
  - pl. tömb, verem, sor, lista, asszociatív tömb
  - az adatszerkezetek rendszerint szabványos osztályok, amelyeket a nyelvi könyvtárban valósítottak meg
    - lehetővé teszik többféle típus kezelését *sablon*okkal
    - könnyű használatot biztosítanak *operátorok* segítségével
    - kihasználják a *dinamikus memóriakezelés* adta lehetőségeket

Adatszerkezetek megvalósítása
Sablonok

- Sokszor előfordul, hogy egy adott osztályt, különösen adatszerkezetet több elemtípussal is meg akarunk valósítani (pl. számmal, szöveggel, vagy egyéb típussal)
  - ehhez több, művelethalmazában megegyező, de értékhalmazában különböző típus szükséges
- A megoldás az, hogy felveszünk egy behelyettesíthető típust, egy úgynevezett *sablon*t (*template*)
  - a sablont tetszőleges helyen felhasználhatjuk az osztályban
  - példányosításakor behelyettesítünk a használni kívánt konkrét típussal
  - használatát `template<class <név> >` jelöli az osztálynál

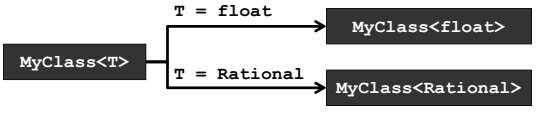
Adatszerkezetek megvalósítása
Sablonok

- Pl.:

```
template <class T>      // T sablon használata
class MyClass {
private:
   int _intValue; // rögzített típusú érték
   T _tValue; // T típusú érték, ahol T cserélhető
public:
   void tMethod(T param){ // T paraméterű metódus
      _tValue = param;
   }
};
…
MyClass<float> mtf;  // T helyettesítése float-tal
mtf.tMethod(3.5); // a paraméter float típusú lesz
```


Adatszerkezetek megvalósítása
Sablonok működése

- A sablonok biztosítják adatszerkezetek esetén a típusfüggetlenséget, hiszen így az elemtípus tetszőleges lehet, pl. `vector<float> fVector;`
- Sablonos típus létrehozásával igazából *típusmintát* hozunk létre, amely a sablon behelyettesítésével válik igazán típussá
  - azaz egy típusmintából több típus is létrejöhet
  - ekkor a sablonból fakadó hibák fordítási időben kiderülnek



Adatszerkezetek megvalósítása
Sablonok alkalmazása

- Mivel a sablonos típus nem igazi típus, csak minta, a kódja nem helyezhető el forrásfájlban, csak fejlécfájlban (ezért a sablonos típusok egy fájlból állnak)
- Minden osztálybeli hivatkozásnál a sablont is meg kell adnunk (vagy általános, vagy konkrét típussal)
- Amennyiben leválasztjuk a metódusok definícióját, a definícióban ismét kell jelölnünk a sablont, pl.:

```
template <class T>      // T sablon használata
class MyClass { … };

template <class T> // ismét jelöljük a sablont
void MyClass<T>::tMethod(T param){ … }
```

Adatszerkezetek megvalósítása
Sablonok lehetőségei

- Egy típus több sablonnal is rendelkezhet, ekkor azokat vesszővel választjuk el, pl.:
  `template <class T1, class T2> class MyClass { … };`
- A sablonok számos további lehetőséget kínálnak
  - nem csak típusok, de tetszőleges alprogramok megjelölhetőek sablonnal
  - értékek is megadhatóak, amik lehetnek sablonosak is
  - a sablonos művelet specializálható konkrét típusokra
- Egy másik lehetséges sablonmegoldás a generikus típus (*generic*, pl. Java), ahol a sablon behelyettesítése futási időben történik

Adatszerkezetek megvalósítása
Indexelés

- Saját típusainkhoz lehetőségünk van indexelő operátort készíteni
  - az indexelő `[ ]` operátorban egy tetszőleges típusú értéket adhatunk meg (az operátoron belül), ez kerül át a paraméterbe
  - amennyiben több értékkel szeretnénk indexelni (pl. mátrix esetén sor/oszlop), használhatjuk a `( )` funktor operátort, amely zárójelezéssel hívható meg, tetszőleges sok paraméter adható át
  - indexelés esetén külön kell ügyelni a beállítás, illetve a lekérdezés kérdésére, ezért az operátort túl kell terhelni (a túlterhelést a `const` kulcsszó fogja biztosítani)

Adatszerkezetek megvalósítása
Indexelés

- Pl.:

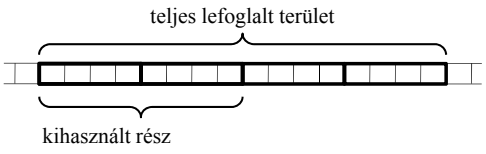
```
class MyClass {
private:
   int _values[100]; // értékek tömbje
public:
   int operator[](int index) const {
      // indexelő operátor lekérdezéshez
      return _values[index]; // értéket ad vissza
   }
   int& operator[](int index) {
      // indexelő operátor értékadáshoz
      return _values[index];
      // referenciát ad vissza
   }
   …
```


Adatszerkezetek megvalósítása
Dinamikus memóriakezelés

- A dinamikus memóriakezelés lehetővé teszi, hogy programfutás közben allokáljuk területeket a memóriából
  - a `new` és `delete` operátorok segítségével
  - több terület is foglalható egyszerre (dinamikus tömb)
  - a lefoglalt területeket mutatók segítségével kezeljük
- Saját típusokban is felhasználhatjuk a dinamikus memóriakezelés adta lehetőséget
  - elhelyezhetünk bennük mutatókat, mint mezőket, amelyekre dinamikusan allokálhatunk memóriaterületet
  - az allokálást többször is elvégezhetjük a programfutás során

Adatszerkezetek megvalósítása
Dinamikus méretezés

- A dinamikusan allokált tömb alkalmas arra, hogy futás közben szabályozzunk méretet
  - pl. tömb esetén allokálunk egy területet, amely egy részét fogja kitenni a vektor tényleges tartalma



  - probléma, hogy így akkor is nagy területet kell allokálni, ha jórészt kevéssé használjuk ki

Adatszerkezetek megvalósítása
Dinamikus méretezés

- A dinamikusan lefoglalt terület nem méretezhető át, de lehetőségünk van új területet foglalni, és arra lecserélni a régit
- A következő lépéseket kell elvégeznünk:
  1. új tömb dinamikus lefoglalása
  2. az elemek átmásolása az új tömbbe
  3. a régi tömb törlése, lecserélése az újra (mutató átállítás)
- Ez jelentős műveletigényt jelent, ezért ritkán alkalmazzuk, nem egyesével növeljük a méretet, hanem duplájára
- Fordítva is alkalmazható, felezhetünk is méretet (így nem foglalunk felesleges memóriát)

Adatszerkezetek megvalósítása
Dinamikus méretezés

- Pl.:

```
int* values = new int[10];
   // 10 méretű dinamikus tömb
… // 10 elemet behelyeztünk, szeretnénk 11-et is

int* t = new int[20]; // új tömb lefoglalása
for (int i = 0; i < 10; i++)
   t[i] = values[i]; // elemek áthelyezése

… // 11. elem behelyezése

delete[] values; // régi tömb törlése
values = t;
  // a mutató átállításával lecserélődik a tömb
```

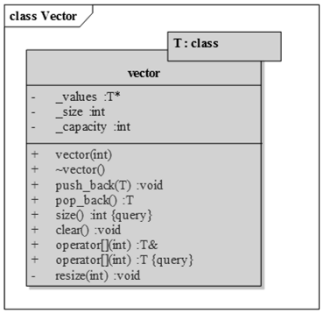

Adatszerkezetek megvalósítása
Példa

Feladat: Valósítsuk meg az intelligens dinamikus vektor (`vector`) típust, amelyet futás közben lehet bővíteni.

- a vektor sablonos lesz, primitív dinamikus tömbbel ábrázoljuk, amely automatikusan méreteződik
- mindig egy részét használjuk ki a teljes tömb kapacitásának (hasonlóan, mint a verem esetében), a konstruktor paraméterben kapja meg a kezdeti méretet
- a `push_back(T)` művelettel bővíthető, a `pop_back()` művelettel redukálható, a `clear()` művelettel üríthető, méretét a `size()` művelettel kérdezhetjük le
- az elemek elérését/beállítását a `[ ]` indexelő operátor biztosítja (ezért két változatban írjuk meg)

Adatszerkezetek megvalósítása
Példa

Tervezés:



Adatszerkezetek megvalósítása
Példa

Megoldás (`vector.hpp`):

```
template <class T>
vector<T>::vector(int size) {
   if (size > 10) // kezdeti kapacitás beállítása
      _capacity = size * 2;
   else
      _capacity = 10;

   _size = size; // méret átvétele
   _values = new T[_capacity];
      // tömb dinamikus létrehozása
}
```


Adatszerkezetek megvalósítása
Példa

Megoldás(`vector.hpp`):

```
template <class T>
void vector<T>::push_back(T value) {
   if (_size == _capacity) // ha már nincs hely
      resize(2 * _capacity);
      // átméretezés a duplájára

   _values[_size] = value;
      // behelyezzük az elemet
   _size++; // növeljük az elemszámot
}
```


Adatszerkezetek megvalósítása
Példányok másolása

- Értékek másolásakor két változatot különböztetünk meg:
  - *mély másolat* (*deep copy*): a teljes példány minden mezőjével lemásolódik a memóriában egy ugyanakkora memóriaterületre
    - lassú, mert teljes másolatot kell végezni
    - a másolaton végzett műveletek nem hatnak az eredetire
  - *sekély másolat* (*shallow copy*): a példány nem, csak annak hivatkozása másolódik a memóriában
    - gyors, csak a memóriacímet másoljuk
    - a másolaton végzett műveletek kihatnak az eredetire
    - ehhez használjuk a referenciaorerátort, illetve mutatókat

Adatszerkezetek megvalósítása
Példányok másolása

- Azonban dinamikus adatszerkezetek mély másolata problémás
  - az alapértelmezett mély másolat a mezőkről készít másolatot, *a mezőkre ráállított dinamikus területekről nem*
  - egy köztes állapotot kapunk, amely több a sekély másolatnál, de kevesebb a mély másolatnál, pl.:

```
vector<int> v1;
vector<int> v2 = v1; // mély másolat (reméljük)
v1.push_back(1);
v2.push_back(2); // a két hozzáadásnak
                 // függetlennek kéne lennie
cout << v1.pop_back() << endl;
   // eredménye: 2, vagyis mégse volt független
   // a két hozzáadás
```


Adatszerkezetek megvalósítása
Példányok másolása

- nem csupán téves viselkedéshez, de programhiba is keletkezhet a hibás másolás miatt, pl.:

```
vector<int> v1;
v1.push_back(1);
{  // programblokk
   vector<int> v2 = v1;
      // mély másolat a lokális változóba
}  // a v2 megsemmisül, lefut a destruktora,
   // vagyis törlődik a dinamikus tömb
cout << v1.pop_back() << endl;
   // programhiba, a tömb már nem létezik
```

- Ahhoz, hogy ezt megoldjuk, felül kell definiálnunk az alapértelmezett másolást

Adatszerkezetek megvalósítása
Másoló konstruktor

- A példányok másolását két művelet, a *másoló konstruktor* (*copy constructor*), illetve az *értékadás operátor* (`=`) valósítja meg
- A másoló konstruktor egy már létező példány alapján hoz létre újat
  - paraméterben megkapja egy ugyanolyan típusú példány referenciáját, törzsében elvégzi a másoló műveleteket
  - amennyiben nincs dinamikus tartalom a mezőkben, az alapértelmezett másoló konstruktor megfelelő
  - amennyiben van dinamikus tartalom, azt létre kell hozni, és az értékeket megfelelően belemásolni

Adatszerkezetek megvalósítása
Másoló konstruktor

- Pl.:

```
class MyClass {
private:
   int* _value;
public:
   MyClass(int v = 0){ // konstruktor
      _value = new int; *_value = v;
   }
   MyClass(const MyClass& other) { // másoló k.
      _value = new int;
      // a dinamikus taralom létrehozása
      *_value = *other._value; // érték másolása
   }
};
```


Adatszerkezetek megvalósítása
Másoló konstruktor

- A másoló konstruktor törzsében tud hivatkozni a másolandó példány mezőire
  - általában ezeket egyenként értékül adjuk az új objektumnak
  - ha mutató is van az mezők között, akkor az általa mutatott objektumot is le kell másolnunk
  - természetesen további inicializálásokat is végezhetünk
- A másoló konstruktor a következő esetekben fut le:
  - közvetlen meghívás: `MyClass b(a);`
  - deklaráció értékadással: `MyClass b = a;`
  - érték szerinti paraméterátadás

Adatszerkezetek megvalósítása
Értékadás operátor

- Amennyiben nem a változó deklarálásakor kap értéket, hanem később, az értékadás operátor fut le:
  `MyClass a, b; b = a;`
- Az értékadás operátor paraméterben kapja meg a másolandó példányt (konstans referenciaként), és hasonló műveleteket végez, mint a másoló konstruktor
  - fontos, hogy már léteznek a dinamikusan létrehozott értékek, így azokat törölni kell
  - ellenőrizni kell, hogy a paraméterben kapott változó nem-e saját maga (a memóriacím segítségével)
  - a többszörös értékadás használatához vissza kell adnia az aktuális példány (`*this`) referenciáját

Adatszerkezetek megvalósítása
Értékadás operátor

- Pl.:

```
class MyClass {
public:
   …
   MyClass& operator=(const MyClass& other){
      if (this == &other)
         // ha ugyanazt a példányt kaptuk
         return *this; // nem csinálunk semmit

      *_value = *(other._value);
      // különben a megfelelő módon másolunk
      return *this; // visszaadjuk a referenciát
   }
};
```


Adatszerkezetek megvalósítása
Típusok megvalósítása

- Amennyiben dinamikusan lefoglalt memóriaterületet használunk a típusban, minden esetben meg kell valósítani a destruktort, a másoló konstruktort, és az értékadás operátort
  - ha nincs dinamikus tartalom, akkor is előfordulhat, hogy használjuk őket
  - mindhárom művelet csak metódusként írható meg
  - az értékadás operátor teljesen független az értékmódosító operátoroktól (**+=**, ***=**, …), amelyek megvalósíthatók a típuson kívül is
  - ha a másolást, vagy értékadást rejtetté tesszük, azzal letiltjuk ezt a funkcionalitást (a leszármazott osztályokra is)

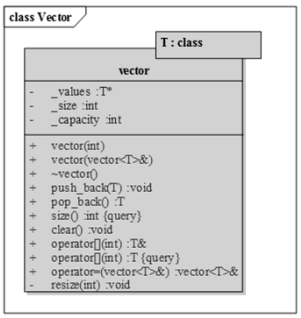
Adatszerkezetek megvalósítása
Példa

Feladat: Valósítsuk meg az intelligens dinamikus vektor (**`vector`**) típust, amelyet futás közben lehet bővíteni.

- a vektor sablonos lesz, primitív dinamikus tömbbel ábrázoljuk, amely automatikusan méreteződik
- a **`push_back(T)`** művelettel bővíthető, a **`pop_back()`** művelettel redukálható, a **`clear()`** művelettel üríthető, méretét a **`size()`** művelettel kérdezhetjük le
- az elemek elérését/beállítását a **`[ ]`** indexelő operátor biztosítja (ezért két változatban írjuk meg)
- biztosítjuk a megfelelő másolást a másoló konstruktor és az értékadás operátor megvalósításával

Adatszerkezetek megvalósítása
Példa

Tervezés:



Adatszerkezetek megvalósítása
Példa

Megoldás (`vector.hpp`):

```
template <class T>
vector<T>::vector(const vector<T>& other) {
   _capacity = other._capacity;
      // átmásoljuk az egyszerű értékeket
   _size = other._size;

   _values = new T[_capacity];
      // létrehozzuk a dinamikus tömböt

   for (int i = 0; i < _size; i++)
      _values[i] = other._values[i];
      // átmásoljuk a hasznos értékeket
}
```


Adatszerkezetek megvalósítása
Példa

Feladat: Használjuk fel a saját vektor típusunkat a grafikus felületű programcsomagunkban.

- csupán ki kell cserélnünk a hivatkozásokat a beépített vektorra
- emellett, az alkalmazásban már található dinamikusan foglalt tartalom, de a vezérlő valamely leszármazottjában is előfordulhat, nekünk egyelőre csak az **`Application`** osztállyal kell törődnünk
- ugyanakkor az alkalmazás, illetve a vezérlők másolása nem célravezető, hiszen minden vezérlő egy dedikált célt szolgál, alkalmazásból pedig csak egy van, ezért inkább tiltsuk le a funkcionalitást

Adatszerkezetek megvalósítása

Példa

Megoldás (`application.hpp`):

```
class Application // grafikus alkalmazás
{
…
private: // letiltjuk a másolást
   Application(const Application&) { }
   Application& operator=(const Application&) {
      return *this;
   }
      // ez a viselkedés úgyse kerül
      // végrehajtásra, ezért a törzs igazából
      // lényegtelen
…
}
```