Informatika 3
4. előadás

Tóth Dávid

Budapesti Műszaki és Gazdaságtudományi Egyetem

2025.03.10.

A `vector` osztály

- A `vector` osztály egy adatlista tárolására kínál a tömbnél rugalmasabb megoldást.
- Az egyik előnye a tömbhöz képest, hogy a méretét nem kell a deklaráláskor megadni, az módosítható bármikor (lehet elemet hozzáadni, törölni).
- Nem szükséges a `new` operátort használni, a memória foglalásáról és felszabadításáról a `vector` osztály gondoskodik.
- Egy másik előnye, hogy mivel osztály, így függvényparaméterként megadható.
- Mivel lekérdezhető ill. változtatható a mérete, így nem a felhasználónak kell gondoskodnia a megfelelő méretről egy függvénynek történő átadás esetén (de lehetőség erre is van).
- Az elemek elérése ugyanúgy történhet, mint egy tömb esetén.

A vector osztály

- A `vector` osztály a sztenderd könyvtár `<vector>` fejállományban (és az `std` névtérben) található.
- Egy `vector` minden eleme ugyanolyan típusú.
- Deklaráció: `vector<típus> változónév;`
- Egy másik lehetőség: `vector<típus> név(elemszám);`
- Az első esetben a `vector` kezdetben 0 elemű, míg a második esetben kezdetben 100 elemű.
- A második esetben a `vector` elemeit `int`, `double`, stb. típusok esetén 0-ra állítja a konstruktor, ha pedig osztály típusú változókat tárolunk, akkor meghívja azok default konstruktorát (itt fontos, hogy ez a konstruktor `public` legyen, különben a program nem fordul).
- Az elemek felsorolásával is inicializálható a `vector`:

```
vector<int> v = {0, 10, 20};
```

A vector osztály

- A vector-ban tárolt elemek elérése ugyanúgy történhet, mint a tömbök esetén (az indexelés 0-tól indul):

```
vector<int> v(100);
int last = v[99];  //a v vector utolsó eleme
```

- Ha v egy vector objektum, akkor az i-edik elem elérésére a v[i] szintaxis helyett használható az at(index) tagfüggvény is: v.at(i).
- A két kód ugyanazt az elemet adja vissza, de van közöttük lényeges különbség.
- Ha a v[i] szintaxist használjuk, akkor ügyelnünk kell rá, hogy az index a megfelelő tartományban legyen, a vector osztály ezt nem ellenőrzi.
- Ha az at tagfüggvényt használjuk, akkor az rossz indexelés esetén egy ún. kivételt dob, de ezt csak később tárgyaljuk részletesen.

A vector osztály

- Ha a vector mérete nem megfelelő, akkor megváltoztathatjuk azt a resize tagfüggvénnyel:

```
vector<int> v(100);
v.resize(110);
v[100] = -3; //rendben van, már létezik a 101. elem
```

- A fenti megoldás helyett használhatjuk a push_back tagfüggvényt, amely a paraméterként megadott elemet hozzáfűzi a vektor végéhez:

```
vector<int> v(100);
v.push_back(-3); //most már 101 elemű a vektor
```

A `vector` osztály

- A fenti esetben méret növelésénél vagy egy elem hozzáfűzésénél a következő történik a háttérben:
  - Mindkét esetben túl kicsi a vektor az új elem tárolásához.
  - A `vector` osztály lefoglal egy nagyobb memóriablokkot az adatok tárolására.
  - Átmásolja az objektumban korábban tárolt elemek az új memóriablokkba.
  - A `push_back` használata esetén az immár nagyobb vektor végére hozzáfűzi az új elemet.
- Lehetséges, hogy az új memóriablokk foglalásánál a szükségesnél nagyobb memóriát foglal az osztály, hogy a későbbi bővítéseknél ne kelljen újra foglalni és másolni.

A `vector` osztály

- Ezt a folyamatot a felhasználó is kontrollálhatja.
- A `size()` tagfüggvény visszaadja a vektor méretét. A visszatérési érték típusa `size_type` (ezt pl. `unsinged int`-té is konvertálhatjuk, de inkább használjuk a `size_type` típust).
- A `capacity()` tagfüggvény visszaadja a vektor számára lefoglalt memóriát. Szintén `size_type` a visszatérési típus.
- A lefoglalt memóriát a `reserve(size_type méret)` tagfüggvénnyel tudjuk állítani.
- A `resize` tagfüggvény a vektor méretét állítja.
- Ha a vektor sokszor bővül, egy előzetes memóriafoglalással megelőzhető sokszoros újbóli foglalás és másolás (gyorsítható az algoritmus).

A `vector` osztály

- További hasznos tagfüggvények:
  - `clear()`: törli a vektor összes elemét, és a méretét 0-ra állítja.
  - `empty()`: igaz értéket ad vissza, ha a vektor üres, hamisat különben.
  - `max_size()`: visszaadja a maximális vektorméretet.
- Ha ugyanolyan típusú elemeket tárolunk a `v1` és `v2` vektorokban, akkor a `v1 = v2` értékadás a `v2` tartalmát a `v1`-be másolja (a `v1` korábbi tartalma persze elvész).

Rendezett párok

- A `pair` osztály segítségével tárolhatunk rendezett párokat.
- Ez az osztály a `utility` fejállományban található.
- A deklarációnál <> közt meg kell adni a két tárolt elem típusát (ezek lehetnek különbözők):

```
pair<bme::Point, double> weighted_point;
```

- Ha nem inicializáljuk a változók értékét, akkor az osztály típusoknak kell legyen publikus default konstruktora.
- Deklarálásnál inicializálhatjuk a változók értékét:

```
using namespace bme;
pair<Point, double> wpoint(Point(1.0, 0.), 1.5);
```

Rendezett párok

- A `pair` tagváltozói (`first`, `second`) publikusak:

```
pair<int, double> p;
p.first = 2;
p.second = M_PI;
```

- A `make_pair` függvénnyel készíthetünk párokat, azokat értékül adhatjuk pair típusú objektumoknak:

```
pair<int, double> p;
p = make_pair(2, M_PI);
```

- A pair osztály jól használható visszatérési típusként, ha egy függvénynek két értéket kell visszaadnia (korábban referenciákkal oldottuk ezt meg).
- A párok összehasonlíthatók az == és != operátorokkal, ha pedig a < operátor mindkét elemére definiált a párnak, akkor ezzel a párt lexikografikusan rendezhetjük (először az első elemet hasonlítjuk össze, ha azok egyenlők, akkor a másodikat).

List

- Bemutatunk még egy tárolót, aminek a `vector` osztályhoz képest (is) vannak erősségei és gyengeségei.
- A `list` osztály a `list` fejállományban található, és adatok egy listáját tárolja lineáris struktúrában.
- A `list` minden eleme ugyanolyan típusú, ezt <> között kell megadni.
- Erősségei: hatékonyan lehet beszúrni vagy törölni elemet.
- Hátrányai: lassú az elemek elérése, pl. a 100. elem eléréséhez a lista elejéről végig kell menni a 100. elemig, nem lehet benne hatékonyan keresni elemet (a `vector`-ban sem).

List

- A deklarálásnál megadható a lista mérete:

  `list<int> li(100);`
- Használtó a `resize(méret)` tagfüggvény is az átméretezésre.
- A `vector` osztályhoz hasonló működéssel használhatók a `size()`, `clear()`, `empty()` tagfüggvények;
- Ha ugyanolyan típusú elemeket tárolunk a `li1` és `li2` listákban, akkor a `li1 = li2` értékadás a `li2` tartalmát a `li1`-be másolja (a `li1` korábbi tartalma elvész).

List

- A `list` osztályban az elemek elérése ún. *iterátorokon* keresztül történik.
- Egy iterátor a tároló egy elemére hivatkozik, hogy melyikre, az az iterátor állapotától függ. Úgy tekinthetünk az általa hivatkozott elemre, mint az "aktuális" elemre.
- Egy iterátor képes a következő vagy a megelőző elemre lépni.
- A `list` osztály egy iterátorának deklarációja:

```
list<int>::iterator it;
```

- Ha `const` listát szeretnénk bejárni, akkor konstans iterátort kell használni:

```
list<int>::const_iterator it;
```

- A deklaráció után az iterátor még nem mutat egy konkrét elemre, értéket kell adni neki.

List

- A `list` objektum `begin()` tagfüggvénye visszaadja az első elemre mutató iterátort:

```
list<int> li(100);
list<int>::iterator it = li.begin();
```

- A fenti értékadás után az `it` iterátor a `li` első elemére mutat (feltéve, hogy a lista nem üres).
- A `list` objektum `end()` tagfüggvénye visszaadja az utolsó elem utáni helyre mutató iterátort:

```
list<int> li(100);
list<int>::iterator it = li.end();
```

Az `it` iterátor a `li` utolsó eleme utánra mutat.

List

- `it++` vagy `++it` lépteti az iterátort a következő elemre.
- `it--` vagy `--it` lépteti az iterátort az előző elemre.
- Több lépéssel is léptethetjük az iterátort: pl. `it+5` az iterátor által mutatott elemtől 5 lépésre található elemre mutató iterátort ad vissza.
- Az iterátor által mutatott elem `*it` (pl. `*it = 5` egy értékadás).

List

- Beszúrás `list`-be:

```
list<int> li;
int e = 1; // a lista leendő eleme
list<int>::iterator it;
```

  - `li.push_front(e)` - beszúrás a lista elejére
  - `li.push_back(e)` - beszúrás a lista végére
  - `li.insert(it,e)` - beszúrás az `it` által mutatott elem elé
- `li.push_front(e)` és `li.insert(li.begin(),e)` ugyanazt jelenti
- `li.push_back(e)` és `li.insert(li.end(),e)` ugyanazt jelenti

List

- Törlés `list`-ből:

```
list<int> li;
list<int>::iterator it;
int e = 2; //eltávolítandó elem
```

  - `li.pop_front()` - első elem törlése
  - `li.pop_back()` - utolsó elem törlése
  - `li.erase(it)` - az `it` által mutatott eleme törlése
  - `li.remove(e)` - az e-vel egyenlő összes elem törlése

List

- Törölhetjük az összes olyan elemet, amik eleget tesznek egy általunk megadott feltételnek.
- Ehhez hozzunk létre egy `bool` értéket visszaadó függvényt:

```
bool is_even(int n){
    return n % 2 == 0;
}
```

- Ezután a páros elemek törlése:

```
li.remove_if(is_even);
```

List

- A lista elemei rendezhetők, ha a $<$ operátor definiálva van a lista elemeinek típusán:

```
list<int> li;
...
li.sort();
```

- Ha a lista rendezett, a `unique()` tagfüggvény törli a többszörös értékeket.

```
li.sort();
li.uqinue();
```

List

- A listát egy `for` ciklus és egy iterátor segítségével bejárhatjuk:

```
for (list<int>::iterator it = li.begin();
        it != li.end(); ++it)
{
    cout << *it << " ";
}
```

- A C++11 óta a "tartomány alapú `for` ciklus" egy kényelmes eszköz az iterátorral rendelkező tárolók bejárására:

```
list<típus> li;
...
for (típus i : li)
{
    cout << i << " ";
}
```

List

- Ha a lista osztályokat tartalmaz vagy az elemeket felül akarjuk írni, akkor a tartomány alapú `for` ciklus változójánál referenciát használunk, hogy ne másolódjon a listaelem:

```
list<bme::Point> li(100);
for (bme::Point& p : li)
{
   p = bme::Point(1.0,1.0);
}
```

- Ha a lista konstans, és referenciát használunk a for ciklusban, akkor a ciklusváltozónak is konstansnak kell lennie:

```
const list<bme::Point> li(100);
for (const bme::Point& p : li)
{
    cout << p << " ";
}
```

A vector osztály iterátora

- A vector objektumban is használhatunk iterátort:

```
vector<típus>::iterator it;
```

A vector osztály iterátora

- A vector objektumban is használhatunk iterátort:

  ```
  vector<típus>::iterator it;
  ```

- A begin(), end(), erase, insert, pop_back függvények és a ++it, --it operátorok hasonlóan működnek, mint a list esetén.

A vector osztály iterátora

- A vector objektumban is használhatunk iterátort:

  vector<típus>::iterator it;
- A begin(), end(), erase, insert, pop_back függvények és a ++it, --it operátorok hasonlóan működnek, mint a list esetén.
- Egy vector esetén az n indexű elemre mutató iterátor elérhető a vect.begin()+n szintaxissal (ahol vect egy vector objektum).

A `vector` osztály iterátora

- A `vector` objektumban is használhatunk iterátort:

  `vector<típus>::iterator it;`
- A `begin()`, `end()`, `erase`, `insert`, `pop_back` függvények és a `++it`, `--it` operátorok hasonlóan működnek, mint a `list` esetén.
- Egy `vector` esetén az `n` indexű elemre mutató iterátor elérhető a `vect.begin()+n` szintaxissal (ahol `vect` egy `vector` objektum).
- A törlés/beszúrás nem olyan hatékony, mivel a vektor közepéről törölve vagy oda beszúrva a későbbi elemeket mozgatni kell.

A `vector` osztály iterátora

- A `vector` objektumban is használhatunk iterátort:

  `vector<típus>::iterator it;`
- A `begin()`, `end()`, `erase`, `insert`, `pop_back` függvények és a `++it`, `--it` operátorok hasonlóan működnek, mint a `list` esetén.
- Egy `vector` esetén az `n` indexű elemre mutató iterátor elérhető a `vect.begin()+n` szintaxissal (ahol `vect` egy `vector` objektum).
- A törlés/beszúrás nem olyan hatékony, mivel a vektor közepéről törölve vagy oda beszúrva a későbbi elemeket mozgatni kell.
- Iterátorral ill. tartomány alapú `for` ciklussal ugyanúgy bejárhatjuk a `vector`-t, ahogy a `list`-et.

A set tároló

- A set tárolóban azonos típusú elemeket tárolhatunk.
- Lényegében úgy működik, mint a matematikai halmaz: minden elem 1-szer vagy 0-szor szerepelhet benne.
- A set osztály a set fejállományban található, deklaráció:

```
set<típus> változónév;
```

- A set tárolóban olyan típusokat tárolhatunk, amelyekre definiálva vannak az == és < operátorok.
- A set tároló tipikusan bináris keresőfaként van megvalósítva.
- Hatékonyan megvalósíthatók benne a beszúrás/törlés műveletek.
- Hatékonyan lehet benne elemet keresni (lekérdezni, hogy egy adott elem benne van-e a halmazban).
- Az n-edik elem direkt elérése cserébe nem lehetséges.

A set tároló

- Legyenek `S`, `T` azonos típusú elemeket tároló `set` objektumok, `e` pedig egy ilyen típusú elem, ekkor
  - elem beszúrása: `S.insert(e);` beszúrja `S`-be az `e` elemet, ha még nincs benne (különben nem történik semmi),
  - elem törlése: `S.erase(e);` törli `S`-ből az `e` elemet, ha benne van (különben nem változik semmi),
  - összes elem törlése: `S.clear();`
  - lekérdezés: `S.count(e);` 1-et ad vissza, ha `e` benne van `S`-ben, különben 0-t,
  - elemek száma: `S.size();` (visszatérési típus `size_type`, `int`-té konvertálható),
  - `S.empty();` igazzal tér vissza, ha `S` üres, hamissal különben,
  - `S==T;` igazzal tér vissza, ha `S` és `T` ugyanazon elemeket tartalmazzák, hamissal különben,
  - `S=T;` `T` tartalma az `S`-be másolódik felülírva `S` korábbi tartalmát.

A set tároló

- Az elemek elérése iterátorokon keresztül történhet:

```
set<típus>::iterator it;
```

- A set osztály begin() és end() tagfüggvényei a szokásos iterátorokat adják vissza.
- Az it++, ++it, it--, --it operátorok működése a szokásos.
- Az iterátor által mutatott elem *it, ez az iterátoron keresztül lekérdezhető, de nem módosítható!
- Az S.find(e); megkeresi az e elemet az S-ben és egy rá mutató iterátort ad vissza, ha e nincs S-ben, akkor a visszatérési érték S.end().
- Példa az elemek bejárására:

```
for(it = S.begin(); it != S.end(); ++it){
    e = *it;
}
```