Tömbök II

Kormányos Andor

Komplex Rendszerek Fizikája Tanszék

2023 szeptember 26.

Tömbök kezelése `std::array` és `std::vector` segítségével

C++-ban másfajta lehetőség is van a tömbök kezelésére, mint a beépített tömb adattípus

- ezek ún osztály sablon segítségével vannak deklarálva
- C++-ban definiált ún `standard container` -k közé tartoznak (ennek részletei most nem túl fontosak)
- sok előre megírt rutin elérhető rájuk

C++ tömbszerű tárolók:

- `std::array` fordításkor kiszámítható méretű, futás közben a méret nem változhat
- `std::vector` futási időben változó méretű (amíg belefér a memóriába)
  - C++ -ban van lehetőség arra, hogy hatékonyan lehessen hozzáadni új elemeket egy tömbhöz, vagyis a tömb mérete a futás során változik

std::array használata

Példa vektorok definiálására és használatára:

```
#include <iostream>
#include <array>

int main()
{
std::array<int, 3> M = {1, 2, 3};
std::cout << M[0] << std::endl;
}
```

- a használathoz kell a `#include <array>`
- meg kell adni, hogy milyen típusú emeket akarunk benne tárolni (`int`) és hányat (3)
- ezután lehet inicializálni az elemek értékét
- `M[0]`: egyes elemek elérése index segítségével
- a tömb bejárása `for` ciklussal

std::vector használata

Példák vektorok definiálására:

```
#include <iostream>
#include <vector>

int main()
{
std::vector<double> v1;
std::vector<int> v2(10,-3);
std::vector<int> v3{10,-3};
std::vector<int> v4=v2;
}
```

- a használathoz kell a `#include <vector>`
- meg kell adni, hogy milyen típusú emeket akarunk benne tárolni (`double`)
- a `(..)`-ben megadhatjuk, hogy mekkora legyen kezdetben a vektor (10) és mivel inicializáljuk az elemeket (-3)
- `{..}` lista segítségével is inicializálhatjuk
- `v3{10,-3}` jelentése: egy kételemű vektor, amelynek elemei 10 és -3
- átmásolhatjuk egy vektor elemeit egy ugyanolyan típusú másikba (beépített tömb típus esetén nem lehetséges)

Elemek hozzáadása a vektorhoz

Töltsünk fel egy vektort a standard input-ról tetszőleges számú elemmel!

```
double szam=0.0;
int meret=0;
std::vector<double> szamvekt;

while (cin >> szam)
{
szamvekt.push_back(szam);
++meret;
}
```

- a `push_back` parancsot használjuk
- ez minden lépésben növeli a `szamvekt` méretét és a végéhez hozzáadja a beolvasott elemet
- `meret`-ben tároljuk a `szamvekt` aktuális méretét

Tömb elemeinek feldolgozása

Pl: ki akarjuk íratni a fent vektor összes elemét miután befejeztük az elemek bevitelét

- használjuk a `meret` változót és egy `for` ciklust, hogy bejárjuk a vektort (mint pl C-ben)
- de használhatjuk a `.size` operációt is

```
for ( size_t i=0; i<szamvekt.size(); i++)
 {
   cout << i << "  " << szamvekt[i] << "\n";
   \\ az elem indexe is elerheto
 }
  cout << endl;
```

  - a `size_t` egy speciális, előjel nélküli integer adattípus
  - ha egyszerűen `int`-ként definiáljuk az i-t, akkor a fordító figyelmeztetést ad

Tömb elemeinek feldolgozása: range alapú `for` ciklus

- akkor használjuk, ha egy tömb minden elemén végig szeretnénk menni, de nem érdekes, hogy az egyes elemeknek mi az indexe a tömben
- nem fordulhat elő, hogy véletlenül a tömb méreténél nagyobb indexű elemet próbálunk elérni

Az előbbi példában használt `szamvekt` tömböt használva:

```
for (auto elem: szamvekt)
{
cout << elem << "\n";
}
```

- az `auto` típusdefiníció olyankor használható, ha a fordító ki tudja találni a változó típusát
- a range alapú `for` ciklusban az `elem` típusa olyan lesz, mint amilyen a `szamvekt` tömb típusa (double)

Kétdimenziós tömbök

Itt csak mátrixokat tekintünk:

- egy kétdimenziós tömb olyan vektor, amelynek minden eleme egy vektor

Ha a mátrix mérete a futási idő előtt ismert (és nem túl nagy):

```
1   std::array<std::array<int, 4>, 4> smallmatrix;
```

- ez egy 4×4 mátrix lesz
- feltöltéshez vagy feldolgozáshoz a `matrix` elemeit elérhetjük az indexek segítségével

```
for (size_t  i = 0; i < 4; i++)
{
    for (size_t j = 0; j < 4; j++)
    {
        cout << smallmatrix[i][j] << "  ";
    }
    cout << endl;
}
```

Kétdimenziós tömbök

Ha a mátrix mérete a futási idő előtt nem ismert:

```
1    std::vector<vector<double>> matrix;
```

- a vektor elemvektorai tetszőleges hosszúak lehetnek
- tehát a fenti deklaráció igazából általánosabb adatszerkezet, mint egy mátrix

Kétdimenziós tömbök

Ha a mátrix mérete a futási idő előtt nem ismert:

```
1    std::vector<vector<double>> matrix;
```

- a vektor elemvektorai tetszőleges hosszúak lehetnek
- tehát a fenti deklaráció igazából általánosabb adatszerkezet, mint egy mátrix
- feltöltéshez vagy feldolgozáshoz a `matrix` elemeit elérhetjük az indexek segítségével
- pl kiíratás

```
for (size_t  i = 0; i < matrix.size(); i++)
    {
        for (size_t j = 0; j < matrix[i].size(); j++)
        {
            cout << matrix[i][j] << "  ";
        }
        cout << endl;
    }
```

Memóriahasználat és elérés szempontjából nem a legjobb eszköz

Kétdimenziós tömbök

Egy kétdimenziós tömböt tárolhatunk egy egydimenziós vektorban is

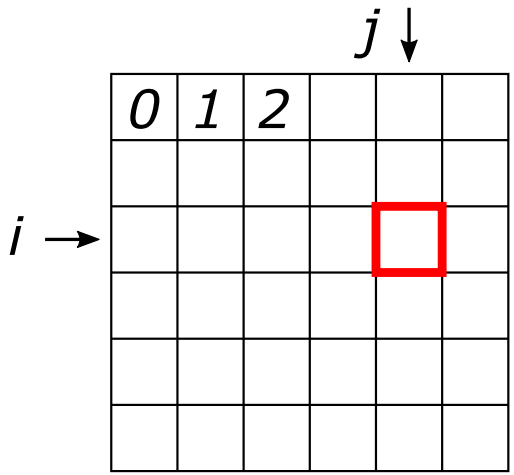
- nem merülnek fel `std::vector<vector<double»` esetén említett memóriaelérés kérdések

Kétdimenziós tömbök

Egy kétdimenziós tömböt tárolhatunk egy egydimenziós vektorban is

- nem merülnek fel `std::vector<vector<double»` esetén említett memóriaelérés kérdések

Hogyan helyezzük el a mátrix elemeit a vektorban?




```
std::vector<double> matrix;
```

- két változót használunk a méret tárolására: `rows`, `cols`
- a mátrixelemeket két index-szel érjük el: $0 \leq i < rows$, $0 \leq j < cols$
- A mátrix soronkénti (row-major) tárolása esetén az i, j elem:

```
matrix[i * cols + j]
```