Eötvös Loránd Tudományegyetem
Informatikai Kar

Alkalmazott modul: Programozás

2. előadás

Procedurális programozás: adatfolyamok, adatsorok kezelése

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

Adatfolyamok kezelése
Adatfolyamok C++-ban

- *Adatfolyam*nak nevezzünk az adatok összefüggő, nem strukturált, de az elemek sorrendjét megtartó halmazt, amely írható, illetve olvasható
  - pl. konzol képernyő, fájl, hálózati csatorna
- A C++ az adatfolyamokat
  - hasonló módon kezeli, jórészt ugyanazon műveletek használhatóak minden adatfolyamra
  - hasonló viselkedéssel látja el, pl. ugyanolyan állapotlekérdezést, helyreállítást biztosít
  - olvasó és író műveletekkel látja el: `<<` (olvasás), `>>` (írás), `getline(<adatfolyam>,<szöveg>)` (sor olvasás)

Adatfolyamok kezelése
Adatfolyamok olvasása

- Az adatfolyamokból egyenként olvashatunk adatokat a `>>` operátorral (szöveg esetén szóközig olvas), vagy soronként
- Soronként olvasáshoz használható
  - a teljes sor beolvasása egy szövegbe:
    `getline(<adatfolyam>, <szöveg változó>)`
  - a sor egy részének beolvasása adott (határoló) karakterig:
    `getline(<adatfolyam>, <szöveges változó>, <határoló karakter>)`
  - a határoló karaktert az adatfolyam eldobja, és a következő olvasó művelet utána kezd
  - ha nem találja a határoló karaktert, a teljes sort beolvassa

Adatfolyamok kezelése
Adatfolyam manipulátorok

- Az adatfolyamok működését számos módon befolyásolhatjuk, pl.:
  - sortörés (`endl`), vízszintes tagolás (`left`, `right`, `internal`)
  - kiírás számrendszere (`dec`, `hex`, `oct`), valós számok formátuma (`fixed`, `scientific`)
  - whitespace karakterek átugrása (`skipws`, `noskipws`)
- További manipulátorok találhatóak az `iomanip` fájlban, pl.:
  - kitöltő karakter beállítása (`setfill(<karakter>)`), mező szélesség beállítása (`setw(<szám>)`)
  - tizedes jegyek száma (`setprecision(<szám>)`)

Adatfolyamok kezelése
Példa

Feladat: Olvassunk be egy valós számot, majd írjuk ki 5 tizedes jegy pontosan egy 15 széles mezőben, amelyet pontokkal töltünk ki.

Megoldás:

```
int main(){
   double nr;
   cout << "Kérek egy számot: "; cin >> nr;
   cout << right << fixed << setfill('.')
        << setw(15) << setprecision(5)
        << nr << endl;
      // formázás beállítása, majd kiírás
   return 0;
}
```


Adatfolyamok kezelése
Végjelig történő feldolgozás

- Amennyiben tetszőleges számú adatot akarunk elemenként feldolgozni, az olvasást végezhetjük egy lezáró adatig, úgynevezett *végjel*ig
- A végjelig történő feldolgozásban a végjel általában már nem kerül feldolgozásra, ezért mindig ellenőriznünk kell, hogy nem a végjelet olvastuk-e be, pl.:

```
while (…) // olvasó ciklus
{
   cin >> data; // olvassuk a következő adatot
   if (data == …) // ha az adat a végjel
      break; // kilépünk a ciklusból
   … // különben feldolgozzuk az adatot
}
```

Adatfolyamok kezelése
Végjelig történő feldolgozás

- Az ellenőrzés egyszerűsíthető az *előreolvasási technika* segítségével, ekkor
  - már a ciklus előtt beolvassuk az első adatot
  - a következő adatot a ciklusmag végén olvassuk, így a ciklusfeltétel ellenőrzése rögtön követi azt
  - pl.:

```
cin >> data; // előreolvasás
while (data != …) // amíg nem értünk a végjelhez
{
   … // feldolgozzuk az adatot
   cin >> data; // olvassuk a következő adatot
}
```


Adatfolyamok kezelése
Példa

Feladat: Adjuk meg a bemenetről olvasott egész számok között hány páros szám található. A számsorozatot 0-val zárjuk.

- számlálás programozási tételt használunk, a szám páros, ha kettővel osztva 0-t ad maradékul
- a 0 a végjel karakter, amit már nem veszünk figyelembe
- előreolvasási technikát alkalmazunk

Megoldás:

```
int main(){
   int nr;
   int c = 0; // számláló
```


Adatfolyamok kezelése
Példa

Megoldás:

```
   cout << "Kérem a számokat: ";
   cin >> nr; // előreolvasás
   while (nr != 0) // a 0-val kilépünk
   {
      if (nr % 2 == 0) // ha a szám páros
        c++; // növeljük a számlálót
      cin >> nr; // olvassuk a következő számot
   }
   cout << "A számok közül " << c
        << " volt páros." << endl;

   return 0;
}
```


Adatfolyamok kezelése
Adatfolyamok hibakezelése

- Az adatfolyamok állapotellenőrzésére több művelet áll rendelkezésünkre
  - a `good()` művelet megadja, hogy az adatfolyam konzisztens állapotban van-e és elérhető
  - a `fail()` művelet megadja, hogy hiba történt-e az adatfolyamban
  - az `eof()` művelet megadja, hogy az adatfolyamnak vége van-e
  - a `clear()` művelet törli az adatfolyam hibás állapotát
  - továbbá minden beolvasás (`>>`, `getline`) visszatérési értéke felhasználható állapotfelmérésre

Adatfolyamok kezelése
Adatfolyamok hibakezelése

- Pl.:

```
int num;
cin >> num; // megpróbálunk számot beolvasni
if (cin.fail()) // ha sikertelen volt a művelet
   cout << "A megadott érték nem szám!";
else
   cout << "A beolvasás sikeres!";

// ugyanez:
if (cin >> num) // sikeres volt a beolvasás
   cout << "A beolvasás sikeres!";
else
   cout << "A megadott érték nem szám!";
```


Adatfolyamok kezelése
Példa

Feladat: Írjuk ki egy számnak a rákövetkezőjét. Ha nem számot adott meg a felhasználó, lépjünk ki.

Megoldás:

```
int main(){
   int nr;
   cout << "Kérek egy egész számot: ";
   if (cin >> nr) // ha be tudtuk olvasni
      cout << ++nr << endl;
   else // ha nem tudtuk beolvasni
      cout << "A megadott érték nem egész szám!"
           << endl;
   return 0;
}
```

Adatfolyamok kezelése
Adatfolyamok hibakezelése

- A hibás állapotot helyre kell állítanunk a `clear()` utasítással
- Amennyiben nem megfelelő adat kerül a bemenetre, a sikertelen beolvasás nem törli az adatot, csak jelzi a hibát
  - a hibás adat átugrását az `ignore(<karakterek száma>, <terminál karakter>)` utasítással végezhetjük
- Pl.:

```
if (cin.fail()){
   cin.clear(); // töröljük a hibajelzést
   cin.ignore(256, '\n');
      // karakterek átugrása a sortörésig
}
cin >> num; // megpróbálunk újra olvasni
```


Adatfolyamok kezelése
Példa

Feladat: Írjuk ki egy számnak a rákövetkezőjét. Ha nem számot ad meg a felhasználó, akkor kérjük be újra.

- egy ciklusban ellenőrizzük a beolvasást, töröljük az állapotot, és újra bekérjük a számot
- mivel azt ellenőrzést a beolvasás után végezzük, használhatunk előreolvasást

Megoldás:

```
int main()
{
   int nr;
   cout << "Kérek egy egész számot: ";
   cin >> nr; // előreolvasás
```


Adatfolyamok kezelése
Példa

Megoldás:

```
     while (cin.fail()) // ha sikertelen a beolvasás
     {
        cout << "A megadott érték nem egész szám!"
             << endl;
        cin.clear(); // töröljük a hibajelzést
        cin.ignore(256, '\n');
           // karakterek átugrása a sortörésig
        cout << "Kérek egy egész számot: ";
        cin >> nr;
      }
      cout << ++nr << endl;
      return 0;
}
```


Adatfolyamok kezelése
Fájlkezelés

- *Fájlkezelés* során az adatfolyamot egy fájl biztosítja, amelybe ugyanúgy tudunk írni és olvasni adatok, mint a képernyő esetén, de olvasás esetén előre adott a tartalom
  - az adatok különféle módon lehetnek eltárolva a fájlban, ilyen tekintetben a legegyszerűbb a *szöveges fájl*, de lehet *bináris fájl*, vagy összetettebb (pl. *DOC*, *XML*, *PDF*)
  - ha az adatokat sorban olvassuk be, illetve írjuk ki, akkor *szekvenciális fájlról* beszélünk
  - tudjuk, mikor van vége az állománynak, ugyanis minden fájl végén egy *fájl vége jel* (end of file: EOF), és le tudjuk kérdezni, hogy ezt sikerült-e beolvasni

Adatfolyamok kezelése
Fájltípusok

- A fájlok szolgálhatnak bemenetként, illetve kimenetként is
  - általában egy fájlt egyszerre csak egy célra használunk, és ez megjelenik a forráskódban is, de lehetőség van felváltva írni, illetve olvasni egy fájlból
- A fájlkezelésben megkülönböztetjük a
  - *fizikai fájl*t: a háttértáron, elérési útvonallal kapcsolt tartalom
  - *logikai fájl*t: a fájl megjelenése a programkódban, lényegében egy adott típusú változó
  - a logikai és fizikai fájlt a programban társítanunk kell

Adatfolyamok kezelése
Szekvenciális fájlkezelés

- C++-ban a szöveges fájlok kezelését az `fstream` fájl biztosítja, amely az `std` névtérben található
- Három logikai fájltípust használhatunk:
  - bemeneti fájl: `ifstream`
  - kimeneti fájl: `ofstream`
  - be- és kimeneti fájl: `fstream`
- A fizikai fájl társítását és megnyitását az `open(<fizikai fájlnév>)` paranccsal végezhetjük, pl.:

```
ifstream f; // f nevű logikai fájl
f.open("adatok.txt"); // társítás és megnyitás
```

Adatfolyamok kezelése
Szekvenciális fájlkezelés

- A létrehozás és a megnyitás egyesíthető, ha közvetlenül megadunk egy fájlnevet, pl.: `ifstream f("adatok.txt");`
- A megnyitásnál alapértelmezett tevékenységeket végez (pl. írás esetén ha nem létezett a fájl, létrehozza, ha létezett, törli a tartalmát), amelyeket felülírhatunk az üzemmód megadásával, úgymint hozzáfűzésre (`ios::app`), vagy csak megnyitás (`ios::nocreate`), stb.
  - az üzemmódokat kombinálhatjuk is a `|` operátorral
  - pl.:

```
ofstream f; // f nevű logikai fájl
f.open("adatok.txt", ios::nocreate | ios::app);
// nem hozza létre, hozzáfűzi a tartalmat
```


Adatfolyamok kezelése
Szekvenciális fájlkezelés

- Fizikai fájlnévként karaktertömb (`char[]`) típusú adatokat adhatunk meg, amit lehet változó is
- Ha `string` típusú változóba szeretnénk bekérni a fájlnevet, akkor azt át kell alakítanunk
  - a szöveg típusban található egy olyan függvény, amely karaktertömbbé alakítja a szöveget, ezt használjuk: *`<változónév>`*`.c_str()`
  - pl.:

```
ifstream file;  // logikai fájl
   string fname;   // egy string
   cin >> fname;   // beolvassuk a stringet
   file.open(fname.c_str());
   // a beolvasott fájlnevet próbáljuk megnyitni
```


Adatfolyamok kezelése
Szekvenciális fájlkezelés

- Ha nem sikerült a fájl megnyitása, az adatfolyam állapota hibás lesz, amely a `fail()` művelettel ellenőrizhető, és a `clear()` művelettel helyreállítható
- Pl.:

```
f.open("data/bemenet.dat"); // megnyitás
if (f.fail()) { // ha nem sikerült megnyitni
   cout << "Nem sikerült megnyitni a fájlt!";
   f.clear(); // állapot helyreállítása

   cout << "Kérek a fájlnevet: ";
   string fname; cin >> fname;
   f.open(fname.c_str());
}
```


Adatfolyamok kezelése
Szekvenciális fájlkezelés

- Fizikai fájlt használat után be lehet zárni a `close()` utasítással
  - a bezárás automatikusan megtörténik a blokk végén, de azért célszerű külön bezárást végezni, amint befejeztük a programban a fájl használatát (különösen írás esetén)
  - bezárást követően a logikai fájlnevet újra használhatjuk másik fizikai fájl kezelésére
  - pl.:

```
ifstream input("adat.txt");
   // adat.txt megnyitása
input.close(); // adat.txt bezárása
input.open("adat2.txt");
   // adat2.txt megnyitása
```


Adatfolyamok kezelése
Szekvenciális fájlkezelés

- Bemeneti fájlból olvashatunk (`>>`, `getline`), kimeneti fájlba írhatunk (`<<`)
  - olvasásnál célszerű mindig ellenőrizni a beolvasott adatot, illetve a fájl állapotát
  - egyszerű adatok esetén a végjelig történő feldolgozást könnyű ciklusba foglalni, pl.:

```
ifstream f("adatok.txt");
… // megnyitás ellenőrzése
while (f >> data) {
   // amíg sikerül adatokat beolvasni
   … // addig feldolgozzuk
}
f.close(); // fájl bezárása
```


Adatfolyamok kezelése
Szekvenciális fájlkezelés

- Részletesebb ellenőrzést is végezhetünk, ha nem akarjuk megszakítani a fájlolvasást az első hiba esetén, pl.:

```
f >> data; // beolvasás előreolvasással
while (!f.eof()) { // amíg nincs vége a fájlnak
   if (f.fail()){ // sikertelen beolvasás
      f.clear(); // állapot törlése
      f.ignore(1024, '\n'); // adatok átugrása
   }
   else {
      … // sikeres beolvasás, feldolgozzuk
   }
   f >> data; // következő adat beolvasása
}
f.close();
```

Adatfolyamok kezelése
Példa

Feladat: Írjuk ki a képernyőre a `szamok.txt` fájlban tárolt egész számok összegét.

- bementi fájlt feldolgozzuk összegzés tételével
- bármennyi szám lehet a fájlban, ezért addig dolgozunk, amíg sikeres a beolvasás (egész számot olvastunk be)

Megoldás:

```
int main(){
   int nr, sum = 0;
   ifstream f("szamok.txt"); // megnyitás

   if (f.fail()) // ha nem sikerült
      cout << "Rossz fájlnév!" << endl;
```


Adatfolyamok kezelése
Példa

Megoldás:

```
   else { // ha sikerült
      while (f >> nr){
         // amíg sikerül adatot olvasnunk
         sum += nr;
      }
      f.close(); // fájl bezárása
      cout << "A számok összege: " << sum << endl;
   }
   return 0;
}
```


Adatfolyamok kezelése
Példa

Feladat: Határozzuk meg egy fájlban tárolt soroknak a hosszát, és írjuk a képernyőre.

- a fájlnevet kérjük be a felhasználótól, sikertelen esetben lépjünk ki a programból

Megoldás:

```
int main(){
   string fileName, line;
   ifstream f;

   cout << "Bemenő adatok fájlja: ";
   cin >> fileName;
   f.open(fileName.c_str());
```


Adatfolyamok kezelése
Példa

Megoldás:

```
   if (f.fail()){ // ha sikertelen
      cout << "Nem található a fájl!" << endl;
      return 1;  // kilépés
   }
   // ha sikeres:
   while(getline(f, line)) {
      // egész sorok olvasása
      cout << line.length() << endl; // sor hossza
   }
   f.close();
   return 0;
}
```


Adatfolyamok kezelése
Példa

Feladat: Egy `telkonyv.txt` fájlban a következő formátumban vannak a sorok: `vezetéknév keresztnév,cím,telefonszám`. Írjuk ki az adatokat az `uj_telkonyv.txt` fájlba a következő formátumba: `keresztnév vezetéknév,telefonszám (15 hosszan, jobbra igazítva), cím`.

- feltételezzük, hogy a fájl létezik, és a formátuma helyes
- négy lépésben olvassuk be a sort négy szöveg változóba
- előreolvasási technikát használunk
- kiíráskor megfelelő manipulátorokkal (`right`, `setw`) módosítjuk a kimenetet

Adatfolyamok kezelése
Példa

Megoldás:

```
int main()
{
   string sName, gName, addr, nr;
   ifstream phb("telkonyv.txt");
   ofstream nPhb("uj_telkonyv.txt");

   // sor beolvasása 4 lépésben:
   getline(phb, sName, ' ');
   // olvasás az első szóközig, a szóköz karakter
   // elveszik
   getline(phb, gName, ',');
   getline(phb, addr, ',');
   getline(phb, nr); // olvasás a sor végéig
```

Adatfolyamok kezelése
Példa

Megoldás:

```
    while(phb.good()){
       // formázott kiírás:
       nPhb << gName << " " << sName << ",";
       nPhb << right << setw(15) << nr << ",";
       nPhb << addr << endl;
       // következő sor beolvasása:
       getline(phb, sName, ' ');
       getline(phb, gName, ',');
       getline(phb, addr, ','); getline(phb, nr);
    }
    phb.close(); nPhb.close();
    return 0;
}
```


Adatfolyamok kezelése
Szövegfolyamok

- A *szövegfolyamok* olyan adatfolyamok, ahol az írás és olvasás szövegen keresztül történik a memóriában
  - típusa a `stringstream`, használatához szükséges az `sstream` fájl és az `std` névtér
  - tetszőleges típusú változót írhatunk be (`<<`) és olvashatunk ki (`>>`, `getline`), a sorrend megmarad
  - alkalmas olyan típuskonverziókat elvégezni, amik automatikusan nem történnek meg (pl. szöveg-szám)
  - az `str()` művelet egyben, szövegként tudja kiadni a szövegfolyam tartalmát, míg az `str(`*`<szöveg>`*`)` kitörli a korábbi tartalmat, és a paraméterben megadottat helyezi be

Adatfolyamok kezelése
Szövegfolyamok

- Szám-szöveg konverzió esetén a szövegfolyam hasonló feladatot lát el, mint a korábbi `itoa` és `atoi` műveletek, amelyek csak karaktersorozatokkal tudnak dolgozni
- Pl. szám átalakítása szöveggé:

```
int num;
… // num értéket kap
stringstream converter;
   // az átalakítást szövegfolyammal végezzük
converter << num; // behelyezzük a számot
string output = converter.str();
   // szövegként kapjuk meg a tartalmat
   // ugyanez:
   // converter >> output;
```


Adatfolyamok kezelése
Szövegfolyamok

- Pl. szöveg átalakítása számmá:

```
int num;
string input;
cin >> input; // beolvassuk a számot szövegesen

stringstream converter;
converter << input;
   // behelyezzük a szövegfolyamba
converter >> num;
   // megpróbáljuk kiolvasni számként
if (converter.fail())
   // ha sikertelen az átalakítás
   cout << converter.str() << " nem szám!";
   // kiírjuk a szövegfolyam tartalmát
```


Adatfolyamok kezelése
Példa

Feladat: Írjuk ki egy számnak a rákövetkezőjét. Ha nem számot adott meg a felhasználó, lépjünk ki.

- a bemenetet nem rögtön számként, hanem szövegként olvassuk be
- használjunk szövegfolyamot a konverzió elvégzésére, csak akkor végezzük el a növelést, ha sikeres a konverzió

Megoldás:

```
int main()
{
   string textInp;
   int numInp;
   stringstream sstr; // szövegfolyam
```


Adatfolyamok kezelése
Példa

Megoldás:

```
    cout << "Kérek egy számot: ";
    cin >> textInp;

    sstr << textInp; // beírjuk a szöveget
    sstr >> numInp; // kiolvasunk egy számot
    if (sstr.fail())
       // ha lehetett konvertálni
       cout << sstr.str() << " nem szám!";
    else
       // ha nem lehetett konvertálni
       cout << ++numInp << endl;
    return 0;
}
```

Adatsorozatok kezelése
Egyszerű tömbök

- A C++ biztosít számunkra egy egyszerű tömb típust, amelyen az egyedüli értelmezett művelet az indexelő operátor használata
  - pl.:
```
int t[10]; // 10 méretű tömb létrehozása
cin >> t[0]; // tömb első eleme értéket kap
```
  - ezzel a tömbbel több probléma is felmerül:
    - nem lehet lekérdezni a méretét (korlátozottan használható a `sizeof` operátor erre a célra, de ez nem praktikus)
    - csak konstans adható meg méretként
    - nem lehet a méretét futás közben megváltoztatni

Adatsorozatok kezelése
Intelligens tömbök

- A C++ ezért biztosít egy speciális tömb típust `vector` néven, amely a korábbi hiányosságokat pótolja
  - használatához szükséges a `vector` fájl és az `std` névtér
  - létrehozásához speciálisan kell megadnunk az elemtípust, illetve a méretet (amely lehet változó, illetve 0 is, csak negatív szám nem), pl.:
```
vector<int> v1(10); // 10 elemű egész tömb
int size = 5;
vector<string> v2(size); // 5 elemű szövegtömb
```
  - a méretet nem kötelező megadni, ekkor egy 0 méretű tömböt hoz létre, pl.:
```
vector<int> v3; // 0 elemű egész tömb
```


Adatsorozatok kezelése
Intelligens tömbök

- mindig ismeri a méretét, és ez lekérdezhető
- futás közben átméretezhető, akár teljesen ki is üríthető
- fontosabb műveletei:
  - elem lekérdezés, módosítás: `<változónév>[<index>]`
  - méret lekérdezése: `<változónév>.size()`
  - kiürítés: `<változónév>.clear()`
  - átméretezés: `<változónév>.resize(<új méret>)`
  - új elem behelyezése a tömb végére (és egyúttal a méretnövelése): `<változónév>.push_back(<érték>)`
  - utolsó elem kivétele (és egyúttal a méret csökkentése): `<változónév>.pop_back()`

Adatfolyamok kezelése
Szövegfolyamok

- Pl. számok sorrendjének megfordítása:
```
vector v; // itt v.size() == 0
int nr, d; cin >> nr;
for (int i = 0; i < nr; i++)
{
   cin >> d;
   v.push_back(d); // vektor bővítése
} // itt v.size() == nr

for (int i = v.size(); i > 0; i--)
{
   d = v.pop_back(); // vektor csökkentése
   cout << d << endl;
} // itt v.size() == 0
```


Adatsorozatok kezelése
Példa

Feladat: Olvassunk be valós számokat a bemenetről amíg 0-t nem írunk, majd adjuk meg az átlagnál nagyobb elemek számát.

- előbb egy összegzést (az eredményt osztjuk a számok számával), majd egy számlálást kell végeznünk
- ehhez azonban el kell tárolnunk az elemeket
- mivel nem tudjuk az elemek számát, olyan adatszerkezet kell, amelynek mérete növelhető futás közben, tehát `vector`-t kell használnunk, és annak a `push_back` függvényét
- beolvasáskor érdemes ellenőrizni, hogy számot kaptunk-e

Adatsorozatok kezelése
Példa

Megoldás:
```
#include <vector> // használjuk a vector-t
…
int main(){
   vector<float> v;
   // float típusú vector, alapból 0 hosszú lesz
   float act;

   while (cin >> act && act != 0)
   {
      // amíg számot kaptunk, amely nem a végjel

      v.push_back(akt); // berakjuk a végére
   }
```

Adatsorozatok kezelése
Példa

Megoldás:

```
float avg = 0; // összegzés
for (int i = 0; i < v.size(); i++)
   avg += v[i]; // kiolvasunk minden elemet
avg /= v.size(); // mérettel osztunk

int c = 0; // számlálás
for (int i = 0; i < v.size(); i++)
    if (v[i] > avg) c++;

cout << "Az átlagnál " << c << " elem
         nagyobb." << endl;
return 0;
}
```


Adatsorozatok kezelése
Mátrixok

- Mivel egy tömb tetszőleges típusú elemek sorozata, az elemtípus maga is lehet tömb, így létrehozhatjuk *tömbök tömbjét*, azaz a *mátrix*ot
  - a tömb dimenziószáma az egymásba ágyazás mértéke
  - az egy dimenziós tömbök a vektorok, a két, illetve magasabb dimenziósok a mátrixok
  - pl.:

```
int t[10][5];
// egy 10, 5 elemű egészekből álló tömbből álló
// tömb, vagyis 10x5 méretű egész típusú mátrix
cin >> t[0][3];
// az első sor negyedik elemének bekérése
```


Adatsorozatok kezelése
Példa

Feladat: Adott 10 tanuló egyenként 5 jeggyel, állapítsuk meg a legjobb átlaggal rendelkező tanulót.

- ehhez használjunk egy mátrixot, kérjük be a jegyeket egyenként
- határozzuk meg minden hallgatóra az átlagot, majd abból keressük meg a maximumot, tehát több programozási tételt is alkalmazzunk a megoldáshoz

Megoldás:

```
int main()
{
   int m[10][5]; // jegyek mátrixának létrehozása
   float avg[10]; // átlagok tömbjének létrehozása
```


Adatsorozatok kezelése
Példa

Megoldás:

```
for (int i = 0; i < 10; i++)
   for (int j = 0; j < 5; j++){
       cout << i+1 << ". tanuló " << j+1
            << ". jegye: ";
       cin >> m[i][j]; // jegyek beolvasása
   }
// összegzés hallgatónként:
for (int i = 0; i < 10; i++){
   avg[i] = 0;
   for (int j = 0; j < 5; j++)
      avg[i] += m[i][j];
   avg[i] /= 5; // átlagot számítunk
}
```


Adatsorozatok kezelése
Példa

Megoldás:

```
int max = 0;
// maximumkeresés az átlagok vektorán
for (int i = 1; i < 10; i++){
    if (avg[i] > avg[max])
        max = i;
}

cout  << "Legjobban a(z) " << max+1
      << ". tanuló teljesített." << endl;
return 0;
}
```


Adatsorozatok kezelése
Mátrixok

- A `vector` típus is felhasználható mátrixok létrehozására
  - a létrehozása kissé körülményes, mivel csak a külső tömb mérete adható meg közvetlenül, a belső tömböket külön kell átméretezni
  - pl. egy $10 \cdot 5$-ös egészmátrix létrehozása:

```
vector<vector<int> > m(10);
   // külső tömb létrehozása
for (int i = 0; i < m.size(); i++)
   m[i].resize(5);
   // belső tömbök létrehozása
```

  - ezután az indexelés a megszokott módon történik, pl.: `m[0][0]`

Adatsorozatok kezelése
Példa

Feladat: Adott 10 tanuló egyenként 5 jeggyel, állapítsuk meg a legjobb átlaggal rendelkező tanulót.

- használjunk intelligens vektor alapú mátrixot

Megoldás:

```
int main(){
   vector<vector<int> > m(10);
   // sorok létrehozása
   for (int i = 0; i < m.size(); i++){
      m[i].resize(5); // oszlopok létrehozása
   }
   // innentől használható a mátrix
   vector<float> avg(m.size()); // átlagok tömbje
```


Adatsorozatok kezelése
Példa

Megoldás:

```
   for (int i = 0; i < m.size(); i++)
      for (int j = 0; j < m[i].size(); j++){
          cout << i+1 << ". tanuló " << j+1
               << ". jegye: ";
          cin >> m[i][j]; // jegyek beolvasása
      }
   // összegzés soronként:
   for (int i = 0; i < m.size(); i++){
      avg[i] = 0;
      for (int j = 0; j < m[i].size(); j++)
         avg[i] += m[i][j];
      avg[i] /= m[i].size(); // átlagot számítunk
   }
```


Adatsorozatok kezelése
Példa

Megoldás:

```
   int max = 0;
   // maximumkeresés az átlagok vektorán
   for (int i = 1; i < avg.size(); i++){
       if (avg[i] > avg[max])
           max = i;
   }

   cout  << "Legjobban a(z) " << max + 1
         << ". tanuló teljesített." << endl;
   return 0;
}
```