10 Algoritmizálás és a C++ programozási nyelv használata

Előszó

Ez a jegyzet a Digitális kultúra tantárgy 10. évfolyamos tananyagából az Algoritmizálás és programozási nyelv használata témát dolgozza fel. Tematikájában, legtöbb feladatában és megjelenési formájában az állami tankönyv[1] és annak online kiegészítése az alapja, ahol lehet, ott a szöveg is azonos, de a tankönyvben bemutatott Python nyelv helyett a C++ nyelvet, illetve a Code::Blocks fejlesztőkörnyezetben történő programozást mutatja be.

A jegyzet – kihasználva, hogy nincs terjedelmi korlátozás – a tankönyvhöz képest jelentős kiegészítéseket tartalmaz:

- Alternatív megoldásokkal és a megoldások összehasonlításával segíti az érdeklődők igényeinek kielégítését, de tisztázza azt is, hogy mi a továbbhaladáshoz szükséges minimum.
- Az algoritmusok elemzése részletesebb – négyféle elágazásra és négyféle ciklusra mutat példát, alternatív megoldásokat.
- Az önálló tanulás támogatása érdekében tárgyalja a program lépésenkénti futtatását, a hibák értelmezését, az adatok beviteli módjait.
- Az elemi adatok mellett a tömb, a lista és a szöveg típusú adatsorozatok használatát is tanítja.
- Az algoritmus elemeit mondatszerű leírással és folyamatábrával is bemutatja.
- Az OOP alapjait a használattal, valamint a rekord továbbfejlesztéseként tárgyalja.
- A programozói gondolkodásmódot, a szokásokat is bemutatja.
- Tanulásmódszertani javaslatokat ad.
- Programozás- és kódolástechnikai ötleteket, módszertani ajánlásokat tesz.
- A fogalmak szemléltetése után a szaknyelv kifejezéseit használja.

A jegyzet teljes megtanulása elsőre soknak tűnhet, a többoldalú megközelítések miatt is ajánlott húzni belőle, az egyes részekre akkor visszatérni, amikor szükség van rá. Ugyanakkor, a lehetőségek áttekintése hozzájárulhat az egyéni preferenciák érvényesítéséhez.

Sikerekben gazdag tanulást kívánok:

Budapest, 2022.

Szalayné Tahy Zsuzsanna

[1] Digitális kultúra 10. tankönyv Oktatási Hivatal 2021.

TARTALOM

Eddig jutottunk 3
- Mindenhol programok 3
- Forráskód, programozási nyelvek és fejlesztői környezet 3
- Változók 4
- Elágazások 5
- Ciklusok és adatsorozatok 6
- Szövegek 8

A tanultak alkalmazása 10
- Elemi adattípusok és elágazások 10
- Ciklusok és adatsorozatok 13

Eljárások, függvények 17
- Eljárást írunk 18
- Függvényt írunk 19
- Eljárás vagy függvény? 20
- Vissza a kezdetekhez! 20
- A program részekre bontásának szabályai 21
- Függvények és eljárások a gyakorlatban 23

Variációk típusalgoritmusokra 35
- Mik azok a típusalgoritmusok? 35
- Kódolási, jegyzetelési praktikák 35
- Történetek a taxisról meg a rókáról 36
- A sorozatszámítás – összegzés és átlagolás 36
- Eldöntés 38
- Kiválasztás 44
- Keresés 45
- Megszámolás 48
- *Kiegészítés:* kiválogatás 49
- Maximum- vagy minimumkiválasztás 50
- A típusalgoritmusok genetikája 53

Feladatok típusalgoritmusokra 54

Kétdimenziós adatszerkezet 63
- Mik azok a kétdimenziós adatszerkezetek? 63
- Kétdimenziós adatstruktúrák feltöltése adattal 64
- Kétdimenziós adatstruktúrák bejárása 65
- Típusalgoritmusok a kétdimenziós adatszerkezetben 65

Objektumok 77

Objektumok sorozata, táblázata 82

Adatsorozat az objektumban 88
- *Kiegészítés:* objektum adatok beolvasása 91

Kétdimenziós adatsorozatok és objektumok a gyakorlatban 92
- *Kiegészítés:* Speciális adatsorozatok 93

Tárgymutató 95

Eddig jutottunk

Könyvünk előző kötetében belekóstoltunk a programozásba, és elég sok mindent megtanultunk – először ezeket ismételjük át.

Mindenhol programok

Tudjuk már, hogy a háztartási gépektől kezdve az autókon és a repülőgépeken keresztül a robotokig mindenben van számítógép, és ahol számítógépek, ott programok is vannak. A digitáliskultúra-órákon a bennünket jobban érdeklő, hagyományos értelemben vett számítógépek (laptopok, asztali gépek, szerverek) és mobileszközök a bekapcsolásukkor egy fő programot indítanak el, az operációs rendszert.

A többi program elindítása, futásuk közben az eszköz erőforrásaihoz (perifériák, háttértárak, memória, processzor) való hozzáférés szabályozása és a programok megállítása az operációs rendszer feladata. A programok egy része automatikusan indul, más részüket a felhasználó indítja el.

Kérdések

1. Milyen operációs rendszer fut a számítógépeden és milyen a mobileszközeiden?
2. Milyen háttértár van a számítógépedben, milyen a mobileszközeidben?

   A programok elindításukig csak a háttértáron találhatók meg. Elindításkor a memóriába tölti őket a számítógép, és a processzor megkezdi a végrehajtásukat. A programot tartalmazó fájl természetesen megmarad a háttértáron ilyenkor is. Egy program elindítása történhet az ikonjára való kattintással, az ikon ujjunkkal történő megérintésével, de minden program elindítható a számítógép parancssorából is.

3. Hogyan indítható el számítógépünk parancssorából egy szövegszerkesztő, egy képszerkesztőt és egy böngészőprogram? A gyakorlatban is valósítsuk meg!

Forráskód, programozási nyelvek és fejlesztői környezet

Programjainkat az esetek túlnyomó többségében forráskódként fogalmazzuk meg. A forráskód az angol nyelvből vett szavakon kívül rendszerint szép számban tartalmaz még mindenféle egyéb jelet és számot, hellyel-közzel mindenki el is tudja ezeket olvasni – a programozáshoz értők nyilván lényegesen eredményesebben.

A forráskódot sima szöveges fájlokban tároljuk és a fájl kiterjesztése általában a programozási nyelvre utal. Egy köszönést a képernyőre író egyszerű program kódfájljának a neve Ruby nyelv használata esetén lehet `szia.rb`, C++ nyelven dolgozva a fájlnév alighanem a `szia.cpp` formát ölti, C# nyelven `szia.cs` míg Python esetén `szia.py`.

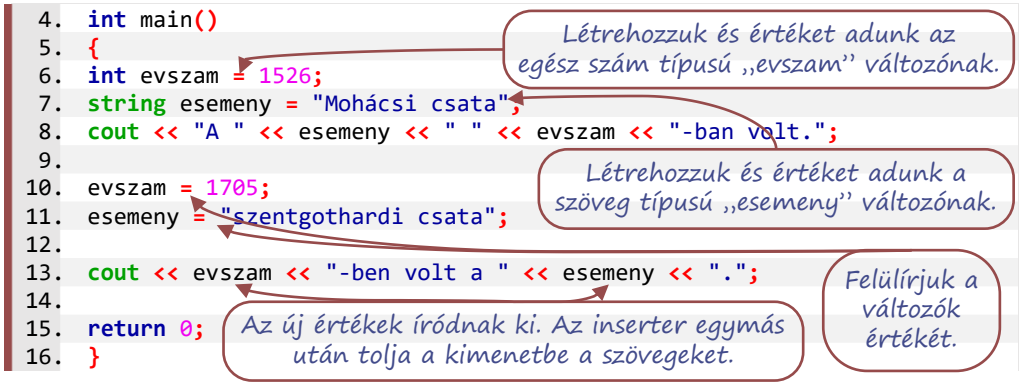
A programkódot egyszerű szerkesztőprogramban is írhatjuk, de általában fejlesztői környezetet, azaz IDE-t használunk. A legegyszerűbb IDE-ket „csak” az különbözteti meg az egyszerű szerkesztőktől, hogy színezéssel segítik a programozót a programban való jobb eligazodásban. A nagyobb tudású, több segítséget adó IDE-k egyben több eszközismeretet is igényelnek.

Nyissunk meg egy IDE-t, és írjuk meg benne azt a programot, amelyik elárulja, hogy épp egy programot futtatunk:


```
1.  #include <iostream>
2.
3.  using namespace std;
4.  int main()
5.  {
6.    setlocale(LC_ALL, "");
7.    cout << "Szia, én egy program vagyok, amit te futtatsz. ";
8.    return 0;
9.  }
```

Változók

A programjainknak gyakran kell adatokat tárolniuk. Az adatokat a gép a memóriájába teszi el, hogy a memórián belül pontosan hova, azt a legtöbbször nem tudjuk. Az eltett adatokat úgy tudjuk ismét elővenni, ha megadjuk azt a nevet, amit az eltett adathoz hozzárendelünk. Ezt a hozzárendelt nevet változónak hívjuk és az eltárolást „programozóul” úgy mondjuk: értéket adunk a változónak. Az értékadás egy művelet, ugyanúgy, mint az összeadás vagy az osztás. Van műveleti jele is, ami sok programozási nyelvben – a C++-ban is – az egyenlőségjel.




```
4.  int main()
5.  {
6.  int evszam = 1526;
7.  string esemeny = "Mohácsi csata";
8.  cout << "A " << esemeny << " " << evszam << "-ban volt.";
9.
10. evszam = 1705;
11. esemeny = "szentgothardi csata";
12.
13. cout << evszam << "-ben volt a " << esemeny << ".";
14.
15. return 0;
16. }
```

A fenti kódban a kimenetre (a képernyőre) az **inserter** „egymáshoz ragasztva” adja ki az adatokat. Eközben a számokból is szöveget (karaktersorozatot) készít. A kiíráskor a szóközökről és a sor végi enterről nekünk kell gondoskodni. Az Enter megfelelője az `endl`, amit külön egységként adunk meg, vagy a `'\n'`, amit karakterként írhatunk be.

```
cout << evszam << "-ben volt a " << esemeny << "." << endl;
cout << evszam << "-ben volt a " << esemeny << '\n';
```

A második sor végén, a `\n` egyetlen karakter. A `\` neve **escape karakter**, ami azt jelöli, hogy az utána lévő karaktert másként kell érteni. Hasznos ismerni a `\t` – tabulátor – karaktert, illetve a kódbéli funkciótól megkülönböztetés miatt speciálisan jelölendő karaktereket: `\'`, `\"` és `\\`. Érdemes az egyéb lehetőségek felől is tájékozódni (kulcsszavak: C++ escape sequence).

Változók típusai, típusátalakítás, adatbekérés

A felhasználó billentyűzeten beírt adatai vagy a programba másik programból beküldött adatok a „console input”-ban – **cin** – sorakoznak, ahonnan az **extractor** (**>>**) kiveszi és a célnak megfelelő típusúra átalakítja az adatot, majd betölti a megadott változóba. Az átalakítás módja az extractor hegyénél lévő változó típusától függ, ezért a változót mindig előre **deklarál**ni kell, azaz meg kell adni a típusát és a nevét. Az adat betöltése értékadás, ami a változó korábbi értékét felülírja.

Ha a program a felhasználótól vár adatot, illik ezt jeleznie a felhasználó felé. Emiatt az adat bekérését gyakran megelőzi egy kiírás arról, hogy mit vár a program. Nagy segítség, hogy az inserterhez hasonlóan, az extractor is tud sorozatban adatot bekérni. Ilyenkor a nem nyomtatható karakterek (szóköz, tabulátor, enter) jelzik az egymást követő adatok határát.

```
1.  #include <iostream>
2.  using namespace std;
3.  int main()
4.  {
5.    setlocale(LC_ALL, "");
6.    cout << "Szóközzel elválasztva add meg a szorzandót és a szorzót: ";
7.    int szorzando, szorzo;
8.    cin >> szorzando >> szorzo;
9.    cout << szorzando << " * "<< szorzo << " = ";
10.   cout << szorzando*szorzo << endl;
11.   return 0;
12. }
```

Eddig összesen ötféle adattípust tároltunk változóban:

- karaktersorozatot, más néven szöveget: **string**;
- egész számot: **int**, **unsigned int**
- ritkábban tizedestörtet, más néven lebegőpontos számot (persze tizedesponttal elválasztva a vessző helyett): **double**, **float**.
- karaktert: **char**
- logikai értéket: **bool** (az ilyen változók értéke **true** [igaz] vagy **false** [hamis] lehet);

Elágazások

Nagyon hamar felmerül az igény, hogy a programunk eltérő feltételek esetén másként viselkedjen. Például, ha elmúlt este nyolc, váltson sötét témára a telefon, ha helyesen adta meg a jelszót a felhasználó, akkor engedjük belépni. Az ilyen problémák megoldására való az **if** és az **else** utasítás. Mit csinál az alábbi program?

```
1.  #include <iostream>
2.  using namespace std;
3.  int main()
4.  {
5.    setlocale(LC_ALL, "");
6.    cout << "Ki volt a Piroska nevű szuperhős fő ellensége? ";
7.    string ellenseg;
8.    cin >> ellenseg;
9.    if(ellenseg == "farkas" || ellenseg == "Farkas") {
10.     cout << "Okos vagy."<< endl;
11.     cout << "Nem kicsit."<< endl;
12.   } else {
13.     cout << "Hááát..."<< endl;
14.     cout << "Nem."<< endl;
15.   }
16.   cout << "Legközelebb a hét törpét kérdezem."<< endl;
17.   return 0;
18. }
```

Két egyenlőségjel kell!

Több feltétel is megadható, közöttük ÉS vagy VAGY kapcsolattal.

Ez a két sor a HA ág – csak akkor futnak le, ha a feltétel teljesül.

Ez a két sor a különben ág.

Ez a sor mindenképp lefut, mert már az elágazás után van.

Ciklusok és adatsorozatok

Ismétlés amíg …

Ha egy feladatrész ismétlődik, akkor ciklust alkalmazhatunk. Van feltételes, számlálós és bejárós ciklusunk. A feltételes ciklus magja addig ismétlődik, amíg fennáll a ciklus elején megfogalmazott feltétel. Az „amíg” – angolul while – a feltételes ciklus elejét jelző utasítás. Ezt követi kerek zárójelben a feltétel, majd (nem a pontosvessző, hanem) az ismétlődően végrehajtandó utasítás és ennek végén a pontosvessző. Ha több utasítást kell végrehajtani a ciklusmagban, akkor kapcsoszárójelek közé kell tenni az utasításokat. A csukó kapcsoszárójel után nem kell pontosvesszőt írni.

```
3.  int main()
4.  {
5.     setlocale(LC_ALL,"");
6.     int valasz = 0;
7.     while(valasz != 4)
8.     {
9.        cout << "Mennyi kétszer kettő? " << endl;
10.       cin >> valasz;
11.    }
12.    cout << "Annyi";
13.    return 0;
14. }
```

Trükk:
Hibás válasz, hogy belépjen a ciklusba

Amig valasz nem 4, addig kérdez és választ vár.

Több utasítás ➔ kell { }

Trükk nélkül is megoldhatjuk a feladatot, hátultesztelős do-while-ciklussal. Ekkor a ciklusmag utasításait egyszer ellenőrzés nélkül végrehajtja a programunk, a végén a feltétellel azt vizsgáljuk, hogy szükséges-e ismétlés.

```
3.  int main()
4.  {
5.     setlocale(LC_ALL,"");
6.     int valasz;
7.     do {
8.        cout << "Mennyi kétszer kettő? " << endl;
9.        cin >> valasz;
10.    } while(valasz != 4);
11.    cout << "Annyi";
12.    return 0;
13. }
```

Itt tároljuk majd a választ.

a do-hoz lép vissza ha a válasz nem 4.

Pontosvessző, mert itt a ciklusutasítás vége.

Adatsorozatok

Az összetartozó adatokat tömbben vagy listában tároljuk. A lista olyan, mint egy vonat, aminek a végére bármikor fűzhetünk újabb elemeket. A tömb olyan, mint egy polcosszekrény, előre meg kell mondanunk, hogy hány adatnak van benne hely. C++ nyelven a listának és a tömbnek is csak azonos típusú adatai lehetnek.

C++ nyelven a tömb statikus adatsorozat, ezért méretét – konstans nem negatív szám – a programkódban tároljuk. A tömb összes adatának együtt egy változóneve van. Azzal különböztetjük meg az egyszerű adattól, hogy a neve után [ ] között megadjuk, hogy hány adat lehet benne. Ezután az egyes adatokra a tömbön belül [ ]-en belül az adat sorszámával – indexével – lehet hivatkozni. Mivel a tömb mérete nem módosítható, jellemzően jó nagyra méretezzük, hogy biztosan beleférjen minden adat. Ilyenkor külön változó(k)ban célszerű tárolni, hogy meddig (hol) van értelmes adat a tömbünkben.

A tömb egyes elemei a tömb indexével elérhető változók. A tömb elemeit tetszőleges sorrendben megadhatjuk, de könnyen programhibát okozhat, ha emiatt véletlenül olyan adatot szeretnénk használni, aminek még nem adtunk értéket. Ennek elkerülésére a tömb elemeit a feladattól függő speciális értékkel **inicializál**juk.

Ismétlés mindegyikre

Amennyiben a tömböt folytonosan töltjük fel adatokkal vagy inicializáltuk, akkor az elemeket számlálós ciklussal is sorra lehet venni. A ciklus változója, „számlálója”, jellemzően a tömb elemeinek az indexeit veszi sorra, ezen keresztül tudjuk a tömb elemek értékét elérni.

```
1.  #include <iostream>
2.  using namespace std;
3.  int main()
4.  {
5.    setlocale(LC_ALL, "");
6.    string varosok[4] = {"Miskolc", "Párizs", "Dublin", "Lajosmizse"};
7.
8.
9.    for (int i = 0; i < 4 ; i++)
10.     cout << varosok[i] << " egy város Európában." << endl;
11.   return 0;
12. }
```

4 elemű tömb.

Megmondjuk, mi a 4 elem.

i számlál 0-tól.

Belép, ha ez igaz.

Utána növeli i értékét.

Egy utasítás ➔ nem kell { }.

A lista dinamikus adatsorozat, a C++ lista típusa a `vector<>`. A lista számára (ha akarunk) előre lefoglalhatunk memóriaméretet, de az adatok egymásutáni beillesztésekor ez a méret szükség esetén módosul. A lista által lefoglalt memóriaméret a lista kapacitása (`capacity()`), amely mindig nagyobb vagy egyenlő, mint a lista mérete (`size()`). A listába új elemet mindig

a végére tesszük, nem tudunk „lyukat” hagyni a listaelemek között. A lista elemeit is elérhetjük az elemek indexével, ezért sorra vehetjük számlálós ciklussal. Mivel a lista folytonosan van feltöltve és pontosan annyi elem van benne, amekkora a mérete, ez elemein „bejárás”-sal is végigmehetünk. A bejárós ciklus ciklusváltozója a lista elemeit (és nem az indexeit) veszi sorra, minden ciklus végén a következő listaelemre lép.

```
1.  #include <iostream>
2.  #include <vector>
3.  using namespace std;
4.  int main()
5.  {
6.    setlocale(LC_ALL, "");
7.    vector<string> varosok {"Miskolc", "Párizs", "Dublin"};
8.    cout << varosok.capacity() << " helyen ";  //eredmény: 4
9.    cout << varosok.size() << " adat."<< endl; //eredmény: 3
10.   for (unsigned i = 0; i < varosok.size(); i++) {
11.     cout << varosok[i] << " egy város Európában." << endl;
12.   }
13.   varosok.push_back("Lajosmizse");
14.   for (string varos : varosok) {
15.     cout << varos << " egy város Európában." << endl;
16.   }
17. return 0;
18. }
```

lista típusú dinamikus tároló eszköz.

string-eket tároló lista.

3 adatot beleteszünk.

előkészített méret.

feltöltött méret.

lista végéhez hozzáad egy új elemet.

lista elemeit sorra képviselő adat.

Szövegek

Mit tudunk eddig a szövegekről? A szöveg – **string** – karakterek sorozata, amelyben minden karakter – a tömbhöz és listához hasonlóan – indexeléssel elérhető és módosítható. Ezért a `"Hello"[1] == 'e'` – a Hello string 1-es helyén lévő karakter az ’e’. A memóriában a program futása közben, dinamikusan jön létre, ebben a **vector<char>**-hoz hasonlít, de nincs külön kapacitás értéke, mert a mérete mindig egyenlő a lefoglalt memóriaterülettel. A **string** adatsorozat különleges tulajdonsága, hogy össze lehet fűzni – a + jellel – másik szöveggel vagy karakterrel. Ilyenkor az eredmény egy új **string** lesz, pontosan a szükséges méretű memóriát lefoglalva. A **string** mérete a lista méretéhez hasonlóan a **size()** függvénnyel adható meg.

Azt is tudjuk, hogy két szöveg egy összeadásjellel összefűzhető, ami egymás mellé írást jelent.

A konzolról az extractor (**>>**) operátorral egy szót tudunk egy változóban eltárolni. Hosszabb szöveget lehet tömbben vagy listában tárolni, amit az elemek – és szükség esetén közte szóközök – összefűzésével egyetlen stringben egyesíthetünk.

A szöveg beolvasásának másik módja a sorolvasás. A `getline(cin, str);` a `'\n'` karakterig, azaz az ENTER leütéséig olvassa be a szöveget a `str` nevű **string** típusú változóba. A függvény általánosabb felhasználási módjában három paramétert adhatunk meg:

```
getline(honnan, hova, meddig)
```

Ebben a „meddig” a beolvasandó karaktersorozat végét jelző karakter. Ezzel tudjuk beolvasni egyenként a – például – vesszővel vagy pontosvesszővel elválasztott adatokat.

```
1.  #include <iostream>
2.  #include <vector>
3.  using namespace std;
4.  int main()
5.  {
```

```
6.   setlocale(LC_ALL, "");
7.   string szoveg = "ez egy szöveg";    //be: getline(cin, szoveg);
8.   string szavak[10] = {"ez", "3", "szó"};    // be: cin >> szavak[i];
9.   int szavakDB = 3;  //beolvasáskor számlálni kell
10.  vector<char> charlist = {'l', 'i', 's', 't', 'a'};
11.     //be: char c; cin >> c;   charlist.push_back(c);
12.
13.  /*kiírás és összefűzés*/
14.  cout << szoveg << endl;
15.  for (int i = 0; i < szoveg.length(); i++) //szoveg.size() is jó
16.    cout << szoveg[i];  //kiírás karakterenként
17.  cout << endl;
18.
19.  string duma = "";
20.  for (int i = 0; i < szavakDB ; i++) {
21.    cout << szavak[i] /*<< " "*/;
22.    duma += szavak[i] /*+ " "*/;
23.  }
24.  cout << endl;
25.
26.  string lista = "";
27.  for (unsigned i = 0; i < charlist.size(); i++) {
28.    cout << charlist[i] /*<< " "*/;
29.    lista += charlist[i] /*+ " "*/;
30.  }
31.  cout << endl;
32.
33.  return 0;
34. }
```

Milyen esetben dorgálja meg az alábbi programrészlet a felhasználót?

```
6.  setlocale(LC_ALL, "");
7.  cout << "Írj be egy mondatot! ";
8.  string mondat;
9.  getline(cin, mondat);
10. int vege = mondat.length() - 1;
11. if (mondat[vege] != '!' && mondat[vege] != '?' && mondat[vege] != '.')
12.   cout << "Ejnye-bejnye!";
13. else
14.   cout << "Igazán gyönyörű mondat.";
```

Írjuk át úgy a programunkat, hogy addig kérjen új meg új mondatokat, amíg a felhasználó meg nem elégeli a mókát és üres bemenet ad (azaz nem ír be semmit, csak lenyomja az ENTERt)!

```
6.  setlocale(LC_ALL, "");
7.  string mondat;
8.  do{
9.    cout << "Írj be egy mondatot! ";
10.   getline(cin, mondat);
11.   int vege = mondat.length() - 1;
13.   if (mondat != "") {
14.     if(mondat[vege] !='!' && mondat[vege] !='?' && mondat[vege] !='.')
12.       cout << "Ejnye-bejnye!";
13.     else
14.       cout << "Igazán gyönyörű mondat.";
15.     }
16.   while (mondat != "");
```

A TANULTAK ALKALMAZÁSA

Elemi adattípusok és elágazások

Feladatok

1. Írjuk képernyőre programmal egy általunk választott vers két versszakát! A vers előtt adjuk meg a szerzőt és a címet, majd sorkihagyást követően az első, újabb sorkihagyást követően a második versszakot írjuk ki! A programunk legfeljebb három cout utasítást használhat.
2. Mondatszerű leírással (más szóval: pszeudokódban) megadunk egy programot. A program bemenete egy állatfaj és az állat legnagyobb sebessége km/h-ban kifejezve.

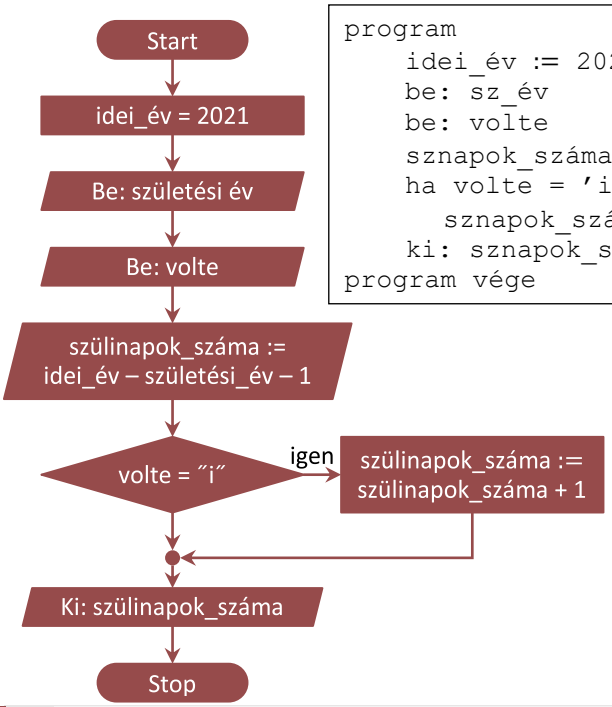
```
program
    be: állatfaj
    be: sebesség
    elágazás
       ha sebesség legfeljebb 50:
          hol := „városban”
       különbenha sebesség legfeljebb 90:
          hol := „országúton”
       különben:
          hol := „autópályán”
    elágazás vége
    ki: „Az”, állatfaj, „a legnagyobb
        sebességével”, hol, „haladhat.”
program vége
```

a) Mit csinál a program?
b) Írjuk át a mondatszerű leírást folyamatábrává!
c) Kódoljuk a programot!

```
 6.  setlocale(LC_ALL,"");
 7.  cout << "Add meg egy állatfaj nevét!" << endl;
 8.  string allat; cin >> allat;
 9.  cout << "Add meg az állatfaj sebességét!" << endl;
10.  int sebesseg; cin >> sebesseg;
11.  string hol;
12.  if (sebesseg <= 50) {
13.     hol = "a városban";
14.  } else if (sebesseg <= 90) {
15.     hol = "az országúton";
16.  } else {
17.     hol = "az autópályán";
18.  }
19.  cout<<"A(z) "<<allat<<" legnagyobb sebességével "<<hol<<" haladhat.";
20.  return 0;
```

d) Teszteljük a program működését az interneten fellelhető, a témába vágó adatok használatával!
e) Nagyon hamar találunk olyan állatot, amely esetében hibás ítéletet hoz a programunk. Mit kell tudnia az ilyen állatnak? Hogyan javítható a programunk?

3. Kérdezzük meg a felhasználótól, hogy melyik évben született, és hogy volt-e már idén születésnapja! A két adat ismeretében írjuk ki, hogy hányadik születésnapját ünnepelte! Először készítsük el a megoldás folyamatábráját vagy mondatszerű leírását. Ha elkészültünk, írjuk meg a programkódot is.



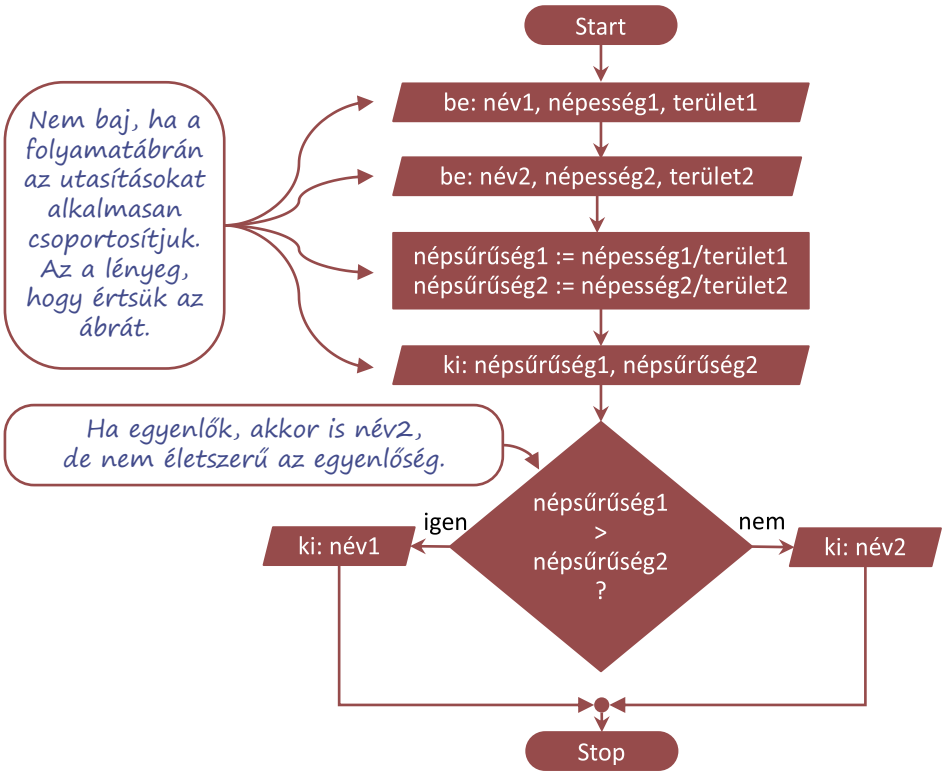

```
program
    idei_év := 2021
    be: sz_év
    be: volte
    sznapok_száma := idei_év - sz_év - 1
    ha volte = 'i':
       sznapok_száma := sznapok_száma + 1
    ki: sznapok_száma
program vége
```

```
 6.  setlocale(LC_ALL,"");
 7.  const int ideiEv = 2021;
 8.  int sznapDB;
 9.  cout << "Melyik evben szulettel? ";
10.  int szulEv; cin >> szulEv;
11.  cout << "Volt mar iden szuletesnapod? (igen/nem) ";
12.  string volte; cin >> volte;
13.  sznapDB = ideiEv - szulEv - 1;
14.  if (volte == "igen")
15.     sznapDB += 1;
16.  cout << "Te " << sznapDB << " eves vagy most.";
17.  return 0;
```

4. Kérdezzük meg a felhasználótól a nevét és azt, hogy a nap hányadik órájában járunk! Köszönjünk neki a nevét megemlítve a napszaknak megfelelően, reggel nyolcig „Jó reggelt”, este hatig „Jó napot”, aztán „Jó estét” kívánva!
   a) Készítsük el a program mondatszerű leírását vagy folyamatábráját!
   b) A leírás vagy az ábra alapján kódoljuk a programot!
   c) *Kihívást jelentő feladat:* Ne a felhasználótól kérdezzük meg az órát, hanem olvassuk ki a számítógép órájából! Az internet segíteni fog az óra kiolvasásának kódolásában.
5. Írjunk olyan programot, amelyik bekéri két autómárka nevét és az autók maximális sebességét! Írjuk ki, hogy melyik autó a gyorsabb.

```
program
    be: egyik_neve
    be: egyik_sebessége
    be: másik_neve
    be: másik_sebessége
    elágazás
     ha egyik_sebessége > másik_sebessége:
        ki: egyik_neve
     különben ha másik_sebessége > egyik_sebessége:
        ki: másik_neve
     különben:
        ki: 'Egyformán gyorsak.'
    elágazás vége
program vége
```

6. Írjunk olyan programot, amely bekéri két ország nevét, népességét és területét! Írjuk ki a két ország népsűrűségét és azt, hogy az az adat melyik országban a nagyobb! Milyen típusú adat a népsűrűség?



7. Kérjünk be két egész számot a felhasználótól és írjuk ki, hogy a kisebb osztója-e a nagyobbnak (azaz egész szám-e a hányadosuk)!
   a) Készítsük el a feladatot megoldó algoritmus mondatszerű leírását!
   b) Kódoljuk az algoritmust: készítsük el a programot!
   c) A példamegoldás csak pozitív egészekkel boldogul. Módosítsuk úgy akár a példát, akár saját működő programunkat, hogy negatív számokat is megadhassunk!

```
 7.  int egyik, masik, maradek, oszto, osztando;
 8.  cout << "Mi legyen az egyik szam?"; cin >> egyik;
 9.  cout << "Mi legyen a masik szam?"; cin >> masik;
10.  if (egyik >= masik) { //mindig a nagyobb szám legyen az osztandó
11.     osztando = egyik;
12.     oszto = masik;
13.  } else {
14.     osztando = masik;
15.     oszto = egyik;
16.  }
17.  maradek = osztando % oszto;/*A maradékos osztás jele a %.*/
18.  if (maradek == 0)
19.     cout << oszto << " osztoja " << osztando << "-nek.";
20.  else
21.     cout << oszto << " nem osztoja " << osztando << "-nek.";
22.  return 0;
```

Egysoros megjegyzés.

Többsoros megjegyzés.

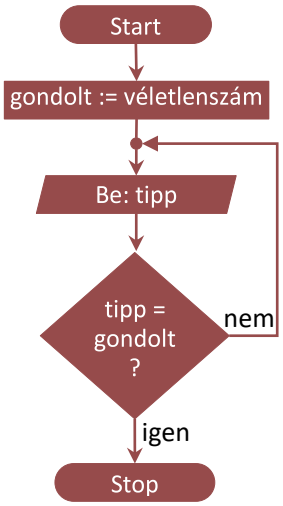
d) Módosítsuk úgy a programunkat, hogy csak a valódi osztókat minősítse osztónak!
e) Írjuk meg a megoldást úgy, hogy az alapműveletek közül csak az összeadást és kivonást használhatjuk.

Mint a legtöbb programozási nyelvben, a C++ nyelvben is van olyan osztás, amelyik a maradékot adja meg. Ez az úgynevezett moduloosztás, vagy egyszerűbben csak mod, a jele a %. Így a 9 % 4 == 1 mert igaz, hogy kilencet néggyel osztva egy a maradék. Ugyanakkor az egész számok (**int**) körében a / jel a bennfoglaló osztás jele, az eredménye egész szám. A 9 / 4 == 2, azaz kilencben a négy kétszer van meg (a maradékot a másik, a % jellel adjuk meg). A tizedestörtek (**double**) értékek között a / jel részekre osztást végez. Emiatt 9,0 / 4,0 == 2,25.

Ciklusok és adatsorozatok

Feladatok

1. Gondoljon a programunk egy számra egy és tíz között! A felhasználó feladata a szám kitalálása. Addig próbálkozhat, amíg el nem találja. A program folyamatábrával és mondatszerű leírással:



Mondatszerű leírás elöltesztelős (while) ciklussal:

```
program
    gondolt := véletlen(10)
    tipp := -1
    ciklus amíg tipp <> gondolt:
       be: tipp
    ciklus vége
program vége
```

... és hátultesztelős (do–while) ciklussal

```
program
    gondolt := véletlen(10)
    ciklus:
       be: tipp
    amíg tipp <> gondolt
    ciklus vége
program vége
```

a) Hogyan állítunk elő véletlen számot? (Ha nem emlékszünk, keressük interneten a random és cpp kifejezésekre!)

b) Kódoljuk az algoritmust!

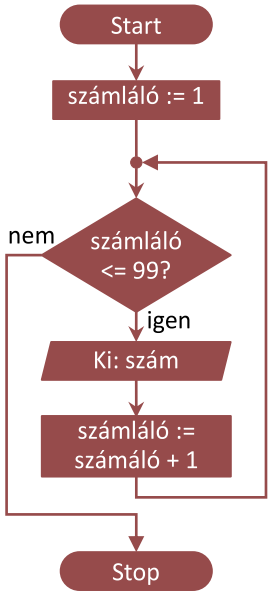
```
#include <iostream>
#include <cstdlib>  /*srand és rand függvényekhez*/
#include <ctime>    /*pontosidő kell a kezdőértékhez*/
using namespace std;
int main()
{
  setlocale(LC_ALL, "");
  srand(time(NULL));
  int gondolt = rand() % 10 + 1;
  int tipp;             //lehet int tipp = -1;
  do {                  //és itt while (tipp != gondolt)
     cout << "Tippelj: ";
     cin >> tipp;
  } while (tipp != gondolt); //ha elején van, akkor itt nincs while();
  return 0;
}
```

c) Javítsunk annyit a programon, hogy találat esetén dicsérje meg a felhasználót!

d) Módosítsuk úgy a programot, hogy írja ki a felhasználónak, hogy a hibás tipp kisebb vagy nagyobb a gondolt számnál!

2. Írjuk ki kilencvenkilencszer, hogy "Hurrá!"

a) A kiírást először feltételes, azaz while-ciklussal oldjuk meg. Ehhez szükségünk lesz egy számláló nevű változóra, aminek a ciklusba való belépés előtt az 1 értéket adjuk, majd minden kiírást követően növeljük az értékét a cikluson belül. A ciklusba való belépés feltétele az, hogy a számláló értéke ne legyen nagyobb 99-nél.




```
számláló :=1
ciklus amíg számláló <= 99:
    Ki: "Hurrá!"
    számláló = számláló + 1
ciklus vége
```

C++ kód részlete:

```
setlocale(LC_ALL,"");
int szamlalo = 1;
while (szamlalo <= 99){
   cout << "Hurrá!" << endl;
   szamlalo = szamlalo + 1;
}
```

b) Oldjuk meg a feladatot számlálós ciklussal is!

```
ciklus i = 1 től 99-ig:
    Ki: "Hurrá!"
ciklus vége
```

C++ kód részlete:

```
for (int i = 0; i < 99; i++)
    cout << "Hurrá!" << endl;
```

c) Írjuk át mindkét programunkat úgy, hogy a „Hurrá!”-k kiírása között szóköz legyen, csak a végén legyen új sor!

3. Vegyünk fel a programunkba egy listát (`vector<>`-t): "barack", "körte", "dinnye", "narancs"! Írjuk ki bejárós ciklussal mindegyikről, hogy egy gyümölcs!

   C++ kód részlete:

```
#include <iostream>
#include <vector>
using namespace std;
int main()
{
  setlocale(LC_ALL, "");
  vector<string> gyumik {"barack", "körte", "dinnye", "narancs"};
    for (string gyumi : gyumik)
    cout << "A " << gyumi << " egy gyümölcs." << endl;
  return 0;
}
```

4. Írjuk ki az egy és száz közötti, hárommal osztható számokat!
   a) Írjuk meg a programot hármasával számoló számlálós ciklussal!

```
for (int i = 3; i <= 100; i += 3)
   cout << "A(z) " << i << " osztható hárommal." << endl;
```

   b) Írjuk meg a programot egyesével számláló számlálós ciklust használva, mondatszerű leírással! Ha a szám osztható hárommal, akkor írjuk ki a számot, a többi esetben pedig egy pontot!
   c) Kódoljuk a programot!
   d) Számoljuk meg és a végén írjuk ki, hogy hány hárommal osztható számot találunk!
   e) Módosítsuk úgy a programot, hogy az egy és száz helyett a felhasználó által megadott számokat használjuk!
   f) Írjuk át úgy a programot, hogy a felhasználó mondhasson számot a három helyett!
   g) Írjuk meg a programot a másik ciklustípus felhasználásával!

A megoldás számlálós ciklussal:

```
setlocale(LC_ALL,"");
cout << "Honnan induljunk? ";
int eleje; cin >> eleje;
cout << "Meddig számoljunk el? ";
int vege; cin >> vege;
cout << "Hánnyal osztható számokat keresünk? ";
int oszto; cin >> oszto;
int darab = 0;
for (int i = eleje; i <= vege; i++) {
   if (i % oszto == 0) {
      cout << " " << i << " ";
      darab += 1;
   } else {
      cout << ".";
   }
}
cout << endl << darab << " ilyen számot találtam." << endl;
```

A feltételes ciklust használó megoldásban a 21. és a 14. sor elé kell egy-egy sort beszúrni. A 14. sor elé kerül a számláló deklarálása és kezdőértéke: `int i = eleje;` a ciklusmag vége elé kerül a számláló növelése `i++;` valamint a `for` helyett a `while`-nak adjuk meg a ciklusba lépés feltételét.

5. Állítsunk elő egy ezer és tízezer közötti egész számokat tartalmazó, húszelemű adatsorozatot! A sorozat elemei (véletlen számok) most azoknak a járműveknek a tömegét adják meg, amiket ma egy komphajó átvitt a folyón. Nehéznek számítanak a 9300 kilogrammnál nehezebb járművek. Írjunk programot, ami válaszol a következő kérdésekre:
   a) Volt-e olyan jármű ma a hajón, ami nehéznek számít? Írjuk ki, ha volt ilyen!
   b) Hány ilyen jármű volt?
   c) Hány kiló járművet vitt át a komp ma összesen?
   d) Mennyi a ma átvitt, nehéznek számító járművek össztömege?
   e) Ha a „nehéz” holnaptól nem 9300, hanem 9000 kilogramm, hány helyen kell átírni a programot? Mit kell tennünk, ha azt szeretnénk, hogy az ilyen változások egyszerűen, egyetlen helyen való átírást jelentsenek?

A kód írását a szükséges eszközök bevonásával kezdjük. A program elején inicializáljuk a véletlenszámgenerátort.

```
1.  #include <iostream>
2.  #include <cstdlib>  /*srand és rand függvényekhez*/
3.  #include <ctime>    /*pontosidő kell a kezdőértékhez*/
4.  #include <vector>   //akkor kell, ha listában tároljuk az adatot
5.  using namespace std;
6.  int main()
7.  {
8.    setlocale(LC_ALL, "");
9.    srand(time(NULL));
```

A feladathoz szükséges adatsorozatot tárolhatjuk …

`int[]` tömbben:

```
10.   int tomegek[20];
11.   for (int i = 0; i < 20; i++)
12.     tomegek[i] =
          rand() % 9000 + 1000 ;
```

`vector<int>` listában:

```
10.   vector<int> tomegek(20);
11.   for (int i = 0; i < 20; i++)
12.     tomegek.push_back(
          rand() % 9000 + 1000);
```

Az a)–d) részfeladatokat egyenként is megoldhatjuk, azonban gyorsabban végez a programunk, ha csak egyszer vesszük végig az összes adatot és közben mind a négy kérdéshez gyűjtjük az információt. A feladat számlálós, lista esetén bejárós ciklussal is megoldható, itt feltételes ciklust használunk:

```
15.   int ez = 0;                    //indexeléshez
16.   bool vanNehez = false;         //a) feladathoz kezdőérték
17.   int nehezDB = 0;               //b) feladathoz feltételes számláló
18.   int ossztomeg = 0;             //c) feladathoz részösszeg tárolása
19.   int nehezOssztomeg = 0;        //d) feladathoz feltételes összegzés
20.   while (ez < 20) {
21.     cout << tomegek[ez] << " ";
22.     ossztomeg = ossztomeg + tomegek[ez]; //jó így is a hozzáadás
23.     if (tomegek[ez] > 9300) {
24.       vanNehez = true;   //vagy a végén: vanNehez = (nehezDB>0);
25.       nehezDB++;
26.       nehezOssztomeg += tomegek[ez]; //így rövidebb a hozzáadás
27.     } //if vége, else nincs
28.     ez++; //a feladatok elvégzése után lép a következőre
29.   }//while-ciklus vége
```

```
30.   cout << endl << "Válaszok:" << endl;
31.   if (vanNehez) cout << "Volt 9300 kilónál nehezebb jármű" << endl;
32.   cout << nehezDB << " db 9300 kilónál nehezebb jármű volt." << endl;
33.   cout << "Ma " << ossztomeg << " kg-ot vitt át a komp." << endl;
34.   cout << "Ebből a nehéz járművek össztömege" << nehezOssztomeg
          << " kg." << endl;
35.   return 0;
36. }
```

Eljárások, függvények

Nagyon ritka az olyan program, amelyik csak egyetlen számítást tartalmaz. Egy-egy adatsorral kapcsolatban általában több kérdés feltehető, de viszonylag ritka, hogy a kérdések mindegyikére választ várjunk minden programfuttatás alkalmával. Jellemzőbb, hogy valamilyen menüből lehet kiválasztani, hogy éppen melyik kérdésre szeretnénk választ kapni, a programunknak melyik részét használnánk. A program fejlődése együtt jár a programsorok számának növekedésével, amit egyre nehezebb áttekinteni. Ráadásul egy nagyobb programban lehetnek olyan részletek, amelyek bizonyos esetekben ismétlődnek. Például hasznos lehet külön egységben megírni egy adatsorozat képernyőre írását, mert néha hasznos ellenőrizni az adatokat. Ezért célszerű a programot részekre, eljárásokra bontani, egyes számításokat elkülönítetten, függvényként megírni.

Eljárást írunk

Nézünk egy példaprogramot! A program mindössze annyit tesz, hogy kiír háromsornyi szöveget, aláhúzva. Persze a parancssorban nem tudunk aláhúzott betűket írni, így az „aláhúzás” valójában egy új sor, benne kötőjelekkel. A kötőjeleket kiíró eljárás a 4–9. sorban van.

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  void alahuzas()
5.  {
6.    for (int i = 0; i < 10; i++)
7.      cout << "-";
8.    cout << endl;
9.  }
10.
11. int main()
12. {
13.   setlocale(LC_ALL, "");
14.   cout << "Ez egy fontos figyelmeztetés!" << endl;
15.   alahuzas();
16.   cout << "Minden sora fontos!" << endl;
17.   alahuzas();
18.   cout << "Komolyan!" << endl;
19.   alahuzas();
20.
21.   return 0;
22. }
```

Jelentése üres, nincs visszatérési értéke.

Az alahuzas() az eljárás neve. A zárójel akkor is kell, ha semmit sem írunk bele.

*{ } között az eljárás **definíciója**, ezt fogja csinálni.*

Ez is egy eljárás, a C++ nyelv része.

Meghívjuk az eljárást: a nevét beírva végrehajtódik az, amit a definícióban megadtunk.

Az eljárásunk szerkezete nagyon hasonlít a főprogramunkhoz, de nem **int**, hanem **void** az első szó és a végén nincs „**return 0;**” A C++ nyelvben – a C nyelvből örökölten – a főprogram

egy függvény, ami egy egész számot ad át az őt futtató operációs rendszernek. A `main()` függvény utolsó sora azt jelenti, hogy a hibakód 0, azaz a program hiba nélkül lefutott. Az `alahuzas()` eljárásnak nincs ilyen eredménye, nem ad vissza semmit. A végére beírhatjuk, hogy „`return;`” de minek, ha egyszer enélkül sincs más választása. A „void” jelentése üres. Mivel az eljárásnak nincs „visszaadott értéke”, ezért az érték típusának helye „üres”.

Kiegészítés: Miért kell kiírni, hogy valami nincs?

Miért kell kiírni a **`void`** szót, miért nem lehet elhagyni? Azért, mert van valami, ami majdnem olyan, mint egy eljárás, mert nincs visszatérési értéke, ugyanakkor a végrehajtása során a memóriában létrehoz valamit, mintha függvény lenne. Ez a létrehozó a konstruktor. A C++ nyelv az eljárást speciális függvénynek tekinti, ezért szintaktikailag (nyelvtanilag) a függvénnyel azonos írásmódot alkalmazzuk. Más nyelvekben a függvény és az eljárás lehet két különböző eszköz és a konstruktor – ha egyáltalán létezik az adott nyelvben – lehet speciális eljárás.

Amikor a fordítóprogram a kódból elkészíti a futtatható exe fájlt, a kódot elölről olvassa, ezért előbb kell megismernie az `alahuzas()` eljárást és csak utána tudja azt használni. A `main()` függvényen belül a `setlocale()` eljárást a fordítóprogram ismeri, az `alahuzas()` eljárást előre megtanítottuk.

Mire kellenek a zárójelek?

Tegyük fel, hogy többféle aláhúzást is szeretnénk, mondjuk csillagokból állót, meg hullámvonalakból készültet. Akkor most írjunk három eljárást sima_alahuzas(), csillagos_ alahuzas() és hullamos_alahuzas() néven? Csak van valami jobb módszer! És tényleg van: a paraméteres eljárás.

Írjuk át az eljárásunkat így:

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  void alahuzas(char jel)
5.  {
6.    for (int i = 0; i < 10; i++)
7.      cout << jel;
8.    cout << endl;
9.  }
10.
11. int main()
12. {
13.   setlocale(LC_ALL, "");
14.   cout << "Ez egy fontos figyelmeztetés!" << endl;
15.   alahuzas('?');
16.   cout << "Minden sora fontos!" << endl;
17.   alahuzas('~');
18.   cout << "Komolyan!" << endl;
19.   alahuzas('*');
20.
21.   return 0;
22. }
```

karakter típusú **paraméter.**

A **paraméterre** hivatkozhatunk az eliárásban.

Két **paramétere** van.

Az eljárást különböző **argumentum**okkal hívjuk meg.

Programunk ilyenkor a következőket csinálja:

- Amikor az eljárás neve után talál valamit a zárójelben (pl.: 15. sor), azt átadja az eljárásnak, ellenőrzi, hogy az adott helyen lévő paraméter típusának megfelel-e a kapott adat. (4. sor)

- Megjegyzi, hogy az adatnak milyen paraméter felel meg (nálunk ez a jel). A paraméter kicsit olyan, mint egy változónév, aminek az adat lett az értéke (az argumentuma).
- Az eljárás belsejében a paraméter nevével hivatkozunk az átadott értékre. Példánkban a jel paramétert az eljáráson belül úgy használjuk, mint egy változót (7. sor).

Függvényt írunk

A függvény ugyanúgy a program egy elkülönült, magában is értelmezhető része, mint az eljárás. Ugyanabban a két esetben használjuk, mint az eljárást. Ugyanúgy paramétert lehet neki átadni, mint az eljárásnak. Két különbség van: nem `void`-ot ad vissza, hanem valami létező adattípust, aminek az értéke az lesz, ami a `return` utasítás után van. A hasonlóság nem véletlen, mivel a C++ nyelvben az eljárást speciális függvénynek tekintik. A matematikai függvényekhez képest, a programozásban használt függvények nem csak az eredmény kiszámítására képesek, a függvény definícióban az eredmény kiszámításán túl bármilyen programrészlet lehet. Ezt látjuk a főprogramunk `main()` függvényénél is.

Alaposan hasonlítsuk össze a két kódot, figyeljük meg a definiálás és felhasználás során miben tér el két, azonos működésű program, ha eljárásként, illetve, ha függvényként írjuk meg!

Eljárás:

```
void pluszhat(int szam)
{
   cout << (szam + 6) << endl;
}
```

Függvény:

```
int pluszhat(int szam)
{
   return (szam + 6);
}
```

Felhasználás:

```
   cout << "4 + 6 = ";
   pluszhat(4);
   cout << "5 + 6 = ";
   pluszhat(5);
```

Felhasználás:

```
   cout << "4 + 6 = " <<
           pluszhat(4) << endl;
   cout << "5 + 6 = " <<
           pluszhat(5) << endl;
```

A bal oldali kódban eljárással valósítjuk meg a feladatot. Az eljárás paraméterében mondjuk meg, hogy melyik számhoz kell hatot hozzáadni. Amikor az eljárást hívjuk (azaz leírjuk a nevét a felhasználás során, akkor lefut: elvégzi az összeadást, és az eredményt kiírja. A futás végeztével a kód végrehajtása visszakerül az eljárás hívásának helyére.

A jobb oldalon függvénnyel valósítjuk meg a feladatot. A függvény a hívását követően lefut: elvégzi az összeadást, visszatérve a hívás helyére, az eredményt átadja a főprogramnak. Olyan, mintha a függvény futása utáni pillanatban a függvény által **visszaadott érték** kerülne a függvény nevének helyére. A kiírást a főprogramunk végzi.

Eljárás vagy függvény?

Eljárás és függvény között tehát az a különbség, hogy a függvénynek van visszatérési értéke, az eljárásnak nincs. A visszatérési értéket a függvényben a `return` szót követően adjuk meg.

Az eljárást és a függvényt a legtöbb programozási nyelv nem különbözteti meg élesen, csak a függvény szót használja. Az ilyen nyelvek – mint a C++ is – és az ilyen nyelveken fejlesztők az eljárásokra csak visszatérési érték nélküli függvényekként tekintenek.

Figyelembe véve, hogy a főprogramunk is egy függvény, megállapíthatjuk, hogy egy függvény definíciójában a számításokon – más függvények felhasználásán – kívül eljárások és vezérlési

utasítások is lehetnek, a függvénynek a visszatérési érték kiszámításán túl **mellékhatás**a is lehet, például a függvény is írhat a képernyőre. Az eljárás – mivel nincs visszatérési értéke – egy olyan függvény, aminek csak mellékhatása van.

Vissza a kezdetekhez!

Ha visszanézzük eddig megírt programjainkat, akkor láthatjuk, hogy egy egész szám bekérése a felhasználótól kiszervezhető egy függvénybe, egy szöveges sor kiírása – a végén enterrel – pedig eljárás lehet. Említettük, hogy függvényeket és eljárásokat két esetben írunk:

- ha valamit többször kell végrehajtani, vagy
- olvashatóbbá válik a főprogram.

A felhasználói programtól függően, ezekben az esetekben mindkét szempont érvényes lehet.

Egy sort kiíró eljárás:

```
5.  void ki(string duma)
6.  {
7.  cout << duma << endl;
8.  }
9.
```

Egész számot bekérő függvény:

```
10.  int be(string duma)
11.  {
12.  cout << duma;
13.  int szam;
14.  cin >> szam;
15.  return szam;
16.  }
```

Felhasználásuk például a 2 × 2 értékét bekérő programunkban:

```
18.  int main()
19.  {
20.    setlocale(LC_ALL,"");
21.    int valasz;
22.    do {
23.      valasz = be("Mennyi kétszer kettő? ");    //be() függvény használata
24.    } while(valasz != 4);
25.    ki("Annyi");                                //ki() eljárás használata
26.    return 0;
27.  }
```

A program részekre bontásának szabályai

Deklaráció és halasztott definíció

Már volt arról szó, hogy a függvénynek (ezzel együtt az eljárásnak is) a programunkban a felhasználás előtt kell szerepelnie. Sok függvény esetén ez azt eredményezi, hogy a `main()` függvényt, ami az egész programunk kezdőpontja, valahol a kód végén kell keresni, ami a programozók számára nem kellemes. Ezért a változók létrehozásához hasonlóan, a függvényeknél is van mód arra, hogy a felhasználás előtt csak „bemutatjuk” a fordítóprogramnak, azaz **deklarál**juk, hogy létezik, míg a működésének a leírását, azaz a **függvénydefiníció**t a `main()` után oda írjuk, ahol az számunkra megfelelő.

A függvény deklarációja nem más, mint a definíció első sora, pontosvesszővel lezárva. Például:

```
    void ki(string duma);
```

Mivel a deklarációban még nincs szerepe a paraméterek nevének, ezért ezt el lehet hagyni. Például:

```
    int be(string);
```

A program írása során könnyebb az első megoldás, mert másolható, de ha később egy paraméter nevét módosítjuk, akkor több helyen kell módosítani, mert a definícióban mindenhol és még a program elején a deklarációban is.

Paraméterlista

Egy függvénynek/eljárásnak több paramétere is lehet. Ilyenkor a paramétereket a függvény neve után álló zárójelben, vesszővel elválasztva soroljuk fel. Felhasználáskor a típusoknak megfelelő adatokat (argumentumokat) a definíció sorrendjében kell megadni.

```
3.  double osszeg (double egyik, double masik)
4.  {
5.    return egyik + masik;
6.  }
7.
8.  double osztas (double osztando, double oszto)
9.  {
10.   return osztando / oszto;
11. }
    /*felhasználás*/
    cout << osszeg(3, 6 ) << endl; //egyik-be behelyettesítjük a 3-at …
    cout << osszeg(6, 3 ) << endl; //most a masik értéke lesz 3.
    cout << osztas(3, 6 ) << endl; //kiírva: 0,5
```

Adatok élettartama, láthatósága

Már a számlálós és bejárós ciklusoknál is láthattuk, hogy egy ciklusfejben vagy ciklusmagban létrehozott változó csak a cikluson belül használható, „élete” a ciklus befejezésével véget ér. Azt is láthattuk, hogy a ciklus előtt – továbbá az elágazás előtt – létrehozott változók a ciklusmagban – illetve az elágazás igaz és hamis ágában – használhatók, azaz értéküket ezeken belül fel tudjuk használni és tudjuk módosítani is.

Függvények esetén a változók használhatósága ehhez hasonló.

- Ami a paraméter listában van az a kapcsoszárójelek között használható.
- Amelyik változót a függvénydefiníció kapcsoszárójelei között hozunk létre, az a létrehozás után, a függvénydefiníció végéig használható.
- Sőt, az a változó, amelyet a függvényen – minden függvényen – kívül deklarálunk (függetlenül attól, hogy kezdőértéket is adunk neki vagy sem), a megadás helye után a függvényeken belül is használhatók. Ezek a program globális változói.

Mind az eljárások, mind a függvények tervezésekor érdemes az alábbi szabályokat betartani:

- A teljes program során használt adatokat globális változóként deklaráljuk, mert ezeket minden függvény látja és módosítani is tudja. Tipikusan ilyen globális adat a tömb, amelyet a várható legnagyobb méretűre – minden függvényen kívül – előre létrehozunk.
- A függvények és eljárások paramétereit – a tömb kivételével – az eljárás vagy a függvény belsejében ne módosítsuk. A függvény használatakor a paraméterekbe az adatok másolata kerül, a módosításokból csak az marad meg, amit a `return`-nel visszaadunk.

Szabályok kihívást kedvelőknek:

- Ha a függvényünk egyik paramétere tömb, akkor figyelni kell arra, hogy nem a másolattal dolgozunk, hanem közvetlenül a tömb elemeivel, azaz minden módosítást az eredeti tömbelemeken végzünk. A tömb paraméterről a függvény nem tudja, hogy a felhasználáskor hány elemű lesz és azt sem, hogy hány elemet használunk belőle. (Akkor sem tudja, ha a szögletes zárójelbe beírtunk valamit.)
- Ha egy tömb globális változó, akkor a felhasznált mérete is globális változó legyen. Ha tömböt paraméterként adjuk át a függvénynek, akkor a felhasznált méretet is paraméterként adjuk át. Ha módosul a méret, akkor a függvénynek a felhasznált méret legyen a visszatérési értéke.
- Ha azt akarjuk, hogy egy paraméterbe behelyettesített adatot a függvényünk módosítson, akkor nem a másolatát kell kérni, hanem az adatra hivatkozást, referenciát. Ezt a típus után írt & jellel adhatjuk meg. A példaprogramban a „`string duma`” egy adat másolatát jelenti, míg a „`string& duma`” a behelyettesített adatot, azaz úgy fog viselkedni, mint egy tömbelem.
- További lehetőségek – például alapértelmezett érték megadása, egy függvénynév többféle paraméterrel való használata, konstans értékek használata – a C++ nyelv dokumentációjából és online fórumokon ismerhetők meg. Kulcsszavak: C++ function parameter.

Az „iparban” a függvényt nagyon gyakran nem ugyanaz a fejlesztő írja, aki majd a programjában használja. Azt szeretjük, ha a függvény fekete doboz: a függvényt használó fejlesztőnek nem kell ismernie a függvény belső működését. Nem tudja, mi történik belül a „dobozban”, csak átadja a függvénynek a feldolgozandó értékeket, a függvény meg visszaadja az eredményt. Az eljárásokat – mint speciális függvényeket – szintén fekete dobozként kezeljük, a megvalósítás módját a felhasználónak nem kell ismernie, csak a mellékhatását tapasztalja.

Bár lehetne másképp is, mégis a fekete doboz elvét szem előtt tartva írjuk meg függvényeinket: aminek van visszatérési értéke, annak ne legyen mellékhatása. Természetesen a függvény a fejlesztése alatt, tesztelési-hibakeresési célból írhat a képernyőre, de egy kész függvény esetén ez zavart okozhat.

Eddig is rengeteg fekete dobozként működő függvényt használtunk – valójában függvény minden olyan „beépített” utasításunk, aminek a neve után zárójel van. Függvény a `rand()`, a `time()`, a `size()`, a `stoi()`; eljárás a `setlocale()` és a `srand()`.

Függvények és eljárások a gyakorlatban

Feladatok

1. Írjunk programrészletet, amely a számára átadott, literben kifejezett térfogatot átváltja akóba, és az eredményt képernyőre írja! (Keressük meg az interneten, hogy egy akó hány liter! Sokféle akó van, használjuk a nekünk tetszőt!)
   a) Írjunk eljárást a feladat megoldására és hívjuk meg az eljárást úgy a főprogramból, hogy kiderüljön, hogy 999 liter hány akó!

Átváltás eljárás mondatszerű leírása:

```
eljárás akoba(liter):
    Ki: liter/58,6
eljárás vége
```

Az átváltás eljárás C++ kód részlete:

```
3.  void akoba (double liter )
4.  {
5.    cout << liter / 58.6 << endl;
6.  }

    /*felhasználás*/
      akoba(999);
```

b) Írjuk át a fenti programot úgy, hogy eljárás helyett függvény végezze az átváltást! Ne felejtsük el, hogy a főprogramot is módosítani kell!

Az átváltás függvényének mondatszerű leírása:

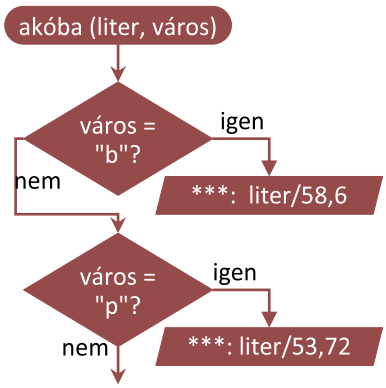
```
függvény akoba(liter):
    vissza: liter/58,6
függvény vége
```

Az átváltás függvény C++ kód részlete:

```
3.  double akoba (double liter )
4.  {
5.    return liter / 58.6;
6.  }

    /*felhasználás*/
    cout << akoba(999) << endl;
```

c) Írjuk át az akóátváltást úgy, hogy budai és pesti akóba is tudjon váltani! Az első paraméter a mennyiség literben kifejezve legyen, második paramétere lehet „b” vagy „p”, ez alapján döntse el, hogy mennyivel oszt.
A megoldást először folyamatábrán és mondatszerű leírással adjuk meg! A feladat alapján írhatunk számot eredményező függvényt vagy az eredményt kiíró eljárást. A kérdéses helyeken jelöljük ***-gal, hogy erről még nem döntöttünk!




```
*** akóba(liter, város)
    elágazás:
    ha város = "b":
      ***: liter/58,6
    különben ha város = "p"
      ***: liter/53,72
    elágazás vége
*** vége
```

A *** vagy az eljárás/függvény szavakat helyettesíti, vagy a ki/vissza utasításokat. A folyamatábrán látszik, hogy valami hiányzik, nincs befejezve: a második elágazásnak a hamiságon nincs folytatása; nincs se stop se vissza. A mondatszerű leírásban a hiány kevésbé látszik, de alaposan megnézve, a „ha–különben ha–különben” elágazásból hiányzik az utolsó rész, a „különben”.

A programunk valóban hiányos: Mi van, ha a felhasználó nem „b”-t vagy „p”-t ír be, hanem – mondjuk – „d”-t, esetleg „Pest”-et?

Ha eljárást írunk, akkor a program lefut úgy, hogy nem ír ki semmit. Ilyenkor a folyamatábra aljára kell még egy „stop” utasítás. A függvény esetén azonban súlyos probléma, ha nem ad vissza semmilyen értéket. Ilyenkor az eredmény – jobb esetben – programhiba,

rosszabb esetben egy nem megalapozott érték. Például 0 vagy −1 vagy 2.122e−314. Azért ez a rosszabb eset, mert ha a függvényünket sokszor használjuk, akkor nehéz lesz kiszűrni a hibás, nem valódi értékeket. Sőt, észre sem vesszük, a további számításaink viszont hibásak lesznek. Ne feledjük: a függvény visszaad egy eredményt, amit a hívás helyén a programunk felhasznál. Ezért függvényt úgy írjunk, hogy a legvégén biztosan legyen visszatérési értéke.

Az akóba váltó függvény megírásához kell még egy szabály. Mondhatjuk azt, hogy ha a város nem „b” és nem „p”, akkor az eredmény legyen 0. Vagy, mondhatjuk, hogy ha a város nem „b”, akkor „p”-nek tekintjük, bármi is legyen …

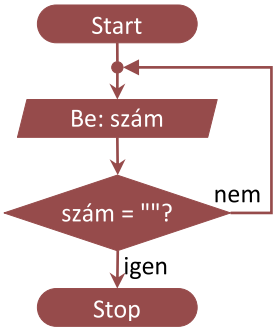
d) Módosítsuk az akóba váltó függvényünket úgy, hogy a város nem megfelelő értékére is legyen eredmény! Próbáljunk ki többféle!

```
3.  double akoba (double liter, string varos )
4.  {
5.    if (varos == "b")
6.      return liter / 58.6;
7.    else if (varos == "p")
8.      return liter / 53.72;
9.    /*return ?;*/
10. }

      /*felhasználás*/
      cout << akoba(999, "d") << endl;
```

2. Írjunk olyan programot, amely számokat kér a felhasználótól, amíg üres bemenetet nem kap, majd eljárással kiírja, hogy az épp beírt szám pozitív, negatív vagy nulla!

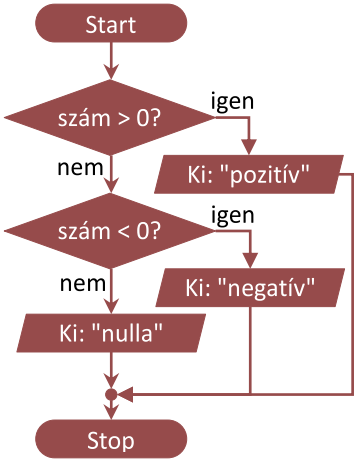
a) Először írjuk meg a főprogramot, azaz azt a részt, amely bekéri a számokat (de még ne csináljunk semmit a számmal)!




```
program
    ciklus:
      be: szám
    amíg szám <> ""
    ciklus vége
program vége
```

```
6.  setlocale(LC_ALL, "");
7.  string szamstr;
8.  do {
9.    cout << "Írj be egy egész számot: ";
10.   getline(cin, szamstr);
11.  } while (szamstr != "");
```

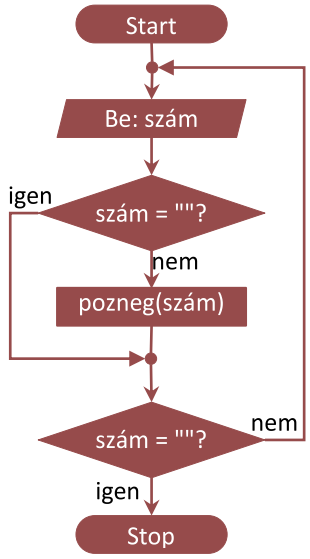
b) A folyamatábra vagy a mondatszerű leírás alapján kódoljuk az eljárást! Az eljárások, függvények nevének megválasztásakor is érdemes figyelnünk arra, hogy a kódot olvasó embert a név segítse a kód megértésében.




```
eljárás pozneg(szám)
    elágazás:
     ha szám >0:
      ki: szám, "pozitív"
     különben ha szám < 0:
      ki: szám, "negatív"
     különben:
      ki: "A szám nulla"
    elágazás vége
eljárás vége
```

```
3.  void pozneg (int szam)
4.  {
5.    if (szam > 0)
6.      cout << szam << " pozitív." << endl;
7.    else if (szam < 0)
8.      cout << szam << " negatív." << endl;
9.    else
10.       cout << "A szám nulla." << endl;
11. }
```

c) Helyezzük el az eljárást hívó részt a főprogramban! Az eljárás nincs felkészülve rá, hogy üres bemenetet kapjon – ha ilyet kap, a program hibaüzenettel kilép. A főprogramnak figyelnie kell arra, hogy csak akkor hívja az eljárást, ha a bemenet nem üres.




```
program
    ciklus:
     be: szám
     ha szám <> "":
      pozneg(szám)
    amíg szám <> ""
    ciklus vége
program vége
```

Mivel az üres bemenet szöveg típusú, ezért a kódban bemenetet ellenőrzése után át kell alakítani szám típusúvá. Ehhez include-oljuk a `<sting>` csomagot és használjuk a `stoi()` függvényt: a `szamstr` szöveget átadjuk a `stoi()` függvénynek, amit ez visszaad azt rögtön betesszük a `pozneg()` eljárásunkba, ez lesz az „`int szam`” paraméter argumentuma.

```
#include <iostream>
#include <string>
namespace std;

void pozneg(int); //itt csak deklarálva, a definíció a main után

int main()
{
  setlocale(LC_ALL, "");
  string szamstr;
  do {
    cout << "Írj be egy egész számot: ";
    getline(cin, szamstr);
    if (szamstr != "")
      pozneg(stoi(szamstr)); //pozneg eljárás és stoi függvény hívása.
  } while (szamstr != "");
  return 0;
}
```

A megoldásunk C++ kódja eléggé bonyolult lett a mondatszerű leíráshoz képest. Ráadásul, a feladatban csak az üres bemenetet vizsgáljuk, pedig futási hibát okozhat az is, ha nem számot (esetleg nem egész számot vagy túl nagy számot) ír be a felhasználó. A programok írása során a felhasználótól bekért adatok ellenőrzése és a programunk megvédése a hibásan megadott adatoktól nagyobb feladat lehet, mint magának a programnak a megírása. Ezért jellemző, hogy az adatbekérést külön eljárásba vagy függvénybe szervezik, amelyet a felhasználási körülményektől függően fejlesztenek, tesznek „bolondbiztossá”. Egy program készítése így általában legalább két részre – eljárásra – bontható: 1. adatok bekérése; 2: feladat megoldása.

3. Írjunk olyan függvényt, amely a paraméterként kapott egyjegyű pozitív számot betűkkel leírva adja vissza! Ötlet: a számok neveit írjuk adatsorozatba, például egy tömbbe úgy, hogy a tömbben a helye (indexe) éppen a nevének feleljen meg.

```
string betukkel(int szam)
{
  string szamnevek[10] = {"nulla", "egy", "kettő", "három", "négy",
                          "öt", "hat", "hét", "nyolc", "kilenc"};
  return szamnevek[szam];
}
  /*felhasználás*/
  for(int i = 0; i < 10; i++)
    cout << betukkel(i) << ", " << endl;
```

4. Megírandó programunk egy – igencsak sztereotip döntéseket hozó – bébiszittert szimulál. A program megkérdezi a gyerekek nevét, majd a lányoknak babát, a fiúknak autót ad játszani. A főprogram dolga, hogy neveket kérdezgessen, amíg üres bemenetet nem kap. Egy eljárás dönti el a gyerekek nemét az alapján, hogy a nevük benne van-e a lánynevek vagy a fiúnevek listában. Ugyanez az eljárás írja ki, hogy az adott gyerek mit kapott játszani. A gyerek nevét az eljárás paramétereként adja át a főprogram.

   Azt, hogy a gyerek neve benne van-e a lányok. illetve a fiúnevek listájában, függvénnyel döntsük el! Ez a függvény a bemenetén megkapja a gyerek nevét és a függvényen belül tárolt nevek alapján **true** vagy **false** értéket ad vissza. Például így:

```
1.  bool lany(string nev)
2.  {
3.    bool benne = false;
4.    if (nev == "Anna" || nev == "Dóra" || nev == "Zita" /* … */)
5.      benne = true;
6.    return benne;
7.  }
```

A fenti megoldás jó, bár nem nevezhető elegánsnak. Hasonló – jó, de nem elegáns – megoldás az is, ha a neveket egyenként adjuk meg többszörös elágazásban. Jobb megoldást jelent, ha a neveket valamilyen adatsorozatban tároljuk és ebben valamilyen ciklust használva vizsgáljuk meg, hogy a paraméterül kapott név a felsorolt nevek között megtalálható-e.

Mivel a felsorolt keresztnevek száma előre felmérhetetlen, célszerű a tárolásukra inkább listát használni, mint tömböt. (Ebben az esetben ne felejtsük el a program elején a vectort include-olni.) A ciklus-típusok közül olyat érdemes választani, amelyik megállítható a találat helyén – nem néz meg feleslegesen minden nevet –, ezért a while-ciklust vagy a (nem egészen számlálós) for-ciklust érdemes írni. A függvényünk például így nézhet ki:

```
1.  bool fiu(string nev)
2.  {
3.    vector<string> nevek{"Bence", "Endre", "Ferenc" /* … */}
4.    unsigned ez = 0;
5.    while (ez < nevek.size() && nev != nevek[ez]) //név, de nem ez a név
6.      ez++;                                       //akkor továbblép
7.    return ez < nevek.size(); //ez a nevek egyikének az indexe ➔ true
8.  }
```

A kódoláskor a C++ nyelv érzékenységére is figyelnünk kell: Mivel a szó hossza – az adatsorozat mérete – nem lehet negatív szám, ezért az összehasonlításhoz olyan adattípust kell választanunk, ami nem lehet negatív. Ezért használjuk az előjel nélküli egész számot, az **`unsigned int`** típust. Ennek a típusnak a rövidebb neve **`unsigned`**. Ha **`int`** típusú lenne az „ez” változónk, a fordítóprogram figyelmeztetne, hogy ebből baj lehet. Valóban, a negatív számok bináris tárolásának szabályai miatt a −1 bináris alakja egyenlő a legnagyobb(!) megadható **`size()`** értékével.

A feladatot megoldó eljárásban – függetlenül attól, hogy a függvényeinket hogyan írtuk –, valahogy így használhatjuk fel:

```
9.  void eztkapja(string nev)
10. {
      /*eljárás eleje*/
      if (lany(nev))
        cout << "Babát kap.";
      /*eljárás folytatása*/
      }
```

a) Írjuk meg a programot!
b) Egészítsük ki a programot úgy, hogy ha egyik listában sem szerepel a név, akkor kérdezze meg, hogy a gyerek milyen nemű!
c) Írjunk játéksorsoló `random.choice()` függvényt, amely a gyerek nemétől függően, mind a fiúk, mind a lányok számára több játékból véletlenszerűen választ ki egy játékot! A programunk a baba vagy autó helyett az így kiválasztott játék nevét írja ki!

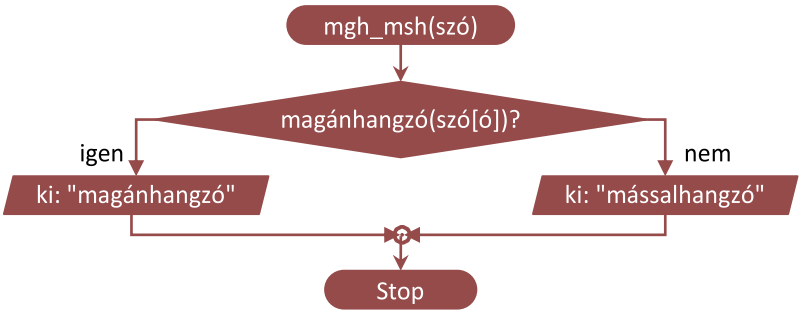
5. Írjunk olyan programot, amelyik eltárolja a felhasználó által megadott városokat (például azokat a városokat, amelyekben az előző 5 évben járt), majd egy eljárás megszámolja és kiírja, hogy hány európai főváros van az átadott listában.
   a) A program megírását az eljárás megírásával kezdjük, mégpedig azért, mert így sokkal egyszerűbb tesztelni az eljárás működését. Az eljárás mondatszerű leírása:

```
eljárás fővárosDB(városok):
    db = 0
    ciklus városok minden város-ára:
     ha város eleme fővárosok-nak:
       db = db + 1
     elágazás vége
    ciklus vége
    ki: db
eljárás vége
```

   A városokat és a fővárosokat is globális változóban tároljuk! A „város eleme fővárosnak” feltételhez az előző feladathoz hasonlóan írjunk függvényt:

```
függvény főváros(város):
    //fővárosok tömbje globálisan megadva
    ez = 0
    ciklus amíg ez < fővárosok_száma ÉS
                város <> fővárosok[ez]:
     ez = ez + 1
    ciklus vége
    vissza: (ez < fővárosok_száma)
függvény vége
```

   Kódoljuk a függvényt és az eljárást!
   b) Gondolkodhatunk fordítva is: a ciklusban a fővárosok lista elemeit járjuk be, és megnézzük, hogy melyik elem található meg az átadott listában. Ha a felhasználó városainak listája {"Bécs", "Bécs"}, akkor a megoldás a megírt program, illetve a fordítva vizsgálás esetén 1 vagy 2 lesz?
   c) Írjuk meg a városokat bekérő eljárást, ami üres bemenetig fogadja a városok nevét! A főprogram a városok bevitele után – ellenőrzésképpen – írja ki a lista elemeit egymás mellé, vesszővel és szóközzel elválasztva. Ezt követően írja ki a fővárosok számát.
6. Írjunk olyan eljárást, amely egy paraméterként kapott szóról eldönti, hogy magánhangzóval vagy mássalhangzóval kezdődik, és a döntését képernyőre írja!
   a) Hogyan tudjuk egy változóban tárolt szó első betűjét megkapni? És a többi betűjét?
   b) Meg kell vizsgálnunk, hogy a megkapott első betű benne van-e egy listában. A mássalhangzókat vagy a magánhangzókat érdemes listába gyűjtenünk?
   c) Hogyan célszerű betűk (karakterek) sorozatát megadni? Stringek listája vagy tömbjeként, karakterek listája vagy tömbjeként vagy esetleg egyetlen stringben? Hogyan lehet megadni, hogy egy betű ebben a karaktersorozatban benne van-e?




```
eljárás mgh_msh(szó):
    maganhangzók = "aáeéiíoóöőuúüű"
    ha magánhangzó(szó[0]):
      ki: "magánhangzó"
    különben:
      ki: "mássalhangző"
eljárás vége
```

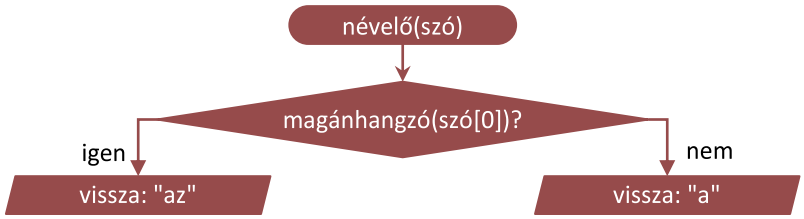
Az eljárás előtt kell szerepelnie a `maganhangzo()` függvénynek – de legalább a deklarációjának – vagy kell helyette egy megfelelő C++ függvény (pl. `find()`). A függvényt most úgy írjuk meg, hogy a magánhangzókat is paraméterként adjuk meg, így ez a függvény később a mássalhangzó detektálására is használható, sőt, bármilyen karakterről meg tudja mondani, hogy egy adott szövegben megtalálható-e. Emiatt a neve nem `maganhangzo`, hanem `bennevan` lesz.

```
15.  void mgh_msh(string szo)
16.  {
17.    string mghk = "aáeéiíoóöőuúüű";
18.    if (bennevan(szo[0], mghk)
19.      cout << "magánhangzóval kezdődik";
20.    else
21.      cout << "mássalhangzóval kezdődik";
22.  }
```

Az eljárásban felhasznált `bennevan` függvény:

```
 5.  bool bennevan(char betu, string minta)
 6.  {
 7.    unsigned i = 0;
 8.    while(i < minta.size() && betu != minta[i])
 9.      i++;
10.    return i < minta.size();
11.  }
```

7. Írjunk olyan függvényt, ami a paraméterként kapott szóhoz a megfelelő határozott névelőt adja vissza! A függvény nagyon hasonlít az előző eljárásunkhoz, amelyikkel eldöntöttük, hogy az átadott szó elején magán- vagy mássalhangzó áll.




```
függvény névelő(szó):
    mghk = "aáeéiíoóöőuúüű"
    itt = 0
    ciklus amíg itt < mghk.hossz ÉS mghk[itt] <> szó[0]:
       itt := itt + 1
    ciklus vége
    ha itt < mghk.hossz:
       vissza: "az"
    különben:
       vissza: "a"
függvény vége
```

A folyamatábrán és a mondatszerű leírásból is látszik, hogy a függvény végén egy olyan elágazás van, aminek mindkét ága a függvény végét, a visszatérési értéket jelenti. Ha a több befejezés csak a megfelelő érték kiválasztásában tér el, akkor ebből nem lehet probléma, azonban arra mindig figyelni kell, hogy minden esetben legyen megoldás.

```
 3.  string maganhangzo(string szo)
 4.  {
 5.    string mghk = "aáeéiíoóöőuúüű";
 6.    unsigned itt = 0;
 7.    while (itt < mghk.size() && szo[0] != minta[itt])
 8.      itt++;
 9.    if (itt < mghk.size())
10.      return "az";
11.    else
12.      return "a";
13.  }
```

Amennyiben egy elágazásnak csak annyi a szerepe, hogy egy adott érték ennyi vagy annyi legyen, akkor lehetőségünk van egy rövidebb forma, a feltételes értékadás, azaz a háromoperandusú operátor használatára. Az elágazás lényeges részeit kiemelve: „ha a <feltétel> teljesül, akkor az érték <ennyi>, különben <annyi> lesz” így írható:

```
    érték = <feltétel>?<ennyi>:<annyi>;
```

Ezt felhasználva programunk kódja kifejezőbb lesz:

```
1.  string maganhangzo(string szo)
2.  {
3.    string mghk = "aáeéiíoóöőuúüű";
4.    unsigned itt = 0;
5.    while (itt < mghk.size() && szo[0] != minta[itt])
6.      itt++;
7.    return (itt < mghk.size()) ? "az" : "a";
8.  }
```

Kihívást jelentő feladat: Hogyan tudjuk ezt a függvényt a számok neveit visszaadóval együtt arra használni, hogy a számok elé is a megfelelő névelőt tegye a programunk?

8. Írjunk olyan függvényt, amelynek visszatérési értéke a paraméterként kapott szó hangrendjét adja meg! Szükségünk lesz egy listára a magas, és egy másikra a mély magánhangzókkal. Érdemes lehet logikai változó értékének beállításával jelezni, hogy találtunk-e magas, illetve mély magánhangzót, majd a két változó értéke alapján meghozni a döntést.

   A feladathoz használjuk a már korábban is megírt `bennevan()` függvényt:

```
 5.  bool bennevan(char betu, string minta)
 6.  {
 7.    unsigned i = 0;
 8.    while(i < minta.size() && betu != minta[i])
 9.      i++;
10.    return i < minta.size();
11.  }
```

   A hangrendet meghatározó függvény mondatszerű leírása:

```
függvény hangrend(szó):
    mély = "aáoóuú"
    magas = "eéiíöőüű"
    voltmély = HAMIS
    voltmagas = HAMIS
    ciklus minden betű-re a szó-ban:
     elágazás
      ha bennevan(betű, mély): voltmély := IGAZ
      különben ha bennevan(betű, magas): voltmagas := IGAZ
     elágazás vége
    ciklus vége
    elágazás
     ha voltmély ÉS NEM voltmagas: vissza: "mély"
     különben ha NEM voltmély ÉS voltmagas: vissza: "magas"
     különben ha voltmély ÉS voltmagas:  vissza: "vegyes"
     különben:  vissza: "nem volt magánhangzó a szóban
    elágazás vége
függvény vége
```

A C++ kódban – a változatosság kedvéért – bejárós ciklust használunk, de ugyanolyan jó a feltételes vagy számlálós ciklus alkalmazása is. Ebben a programban az elágazásokban csak egy-egy utasítás van (a mondatszerű leírásban minden ág egy sor), ezért a kapcsos zárójeleket nem kell kitenni és – a rövid feltétel miatt – olvasható marad a kód akkor is, ha az elágazás feltételét és ágát egy sorba írjuk.

```
string hangrend(string szo)
{
  string mely = "aáoóuú";
  string magas = "eéiíöőüű";
  bool voltmely = false;
  bool voltmagas = false;
  for (char betu : szo) {
    if (bennevan(betu, mely)) voltmely = true;
    else if (bennevan(betu, magas)) voltmagas = true;
  }
  if (voltmely && !voltmagas)      return "mély";
  else if (!voltmely && voltmagas) return "magas";
  else if (voltmely && voltmagas)  return "vegyes";
  /*else*/ return "nincs magánhangzó a szóban";
}
```

```
/*felhasználása*/
cout << "írj ide egy szót! ";
string szo; cin >> szo;
cout << hangrend(szo);
```

Programunk ebben a formában csak kisbetűs szavakkal működik helyesen. Hogyan tudunk segíteni a problémán?
Kihívást jelentő feladat: Hogyan oldható meg a nagybetűs szavak helyes feldolgozása a magánhangzók listájának bővítése nélkül?

9. Írjunk olyan függvényt, amely egy nevet kapva paraméterként visszaadja a névből képzett monogramot! A megoldásban feltételezhetjük, hogy a név részei között egy szóköz van és a név végén nincs szóköz.
   a) Az első megoldásban tételezzük fel, hogy a név csak egy vezetéknévből és egy keresztnévből áll.
   b) Gyakori, hogy több vezetékneve vagy több keresztneve van egy embernek. Oldjuk meg a feladatot úgy, hogy minden nevet figyelembe veszünk!

```
string monogram(string nev)
{
  string mg = nev[0] + ".";
  for(unsigned i = 0; i < nev.size() - 1; i++)
    if (nev[i] == ' ')
      mg += nev[i + 1] + ".";
  return mg;
}
```

   c) A vezetéknév nem csak az apa családneve lehet. Mindkét szülő nevét megkaphatja a gyerek, ilyenkor kötőjel van a két név között. A függvényünk figyeljen erre a lehetőségre is, mindkét név kezdőbetűjét vegye bele a monogramba! Például: Kis-Nagy Ede Pál monogramja K.N.E.P legyen!
   d) *Kihívást jelentő feladat:* A program jelen formájában például Zsákos Bilbó monogramját hibásan adja meg. Oldjuk meg a problémát! Ötlet: a két és három karakterből összetett mássalhangzókat tároljuk egy tömbben, ehhez hasonlítsuk a szavak elejét.
10. Írjunk angol megszólítást előállító függvényt! A függvény paramétere egy név és a megszólítandó kora, azaz az éveinek a száma. Ha a legfeljebb tizenhét éves Kate-et kell megszólítani, akkor a megszólítás tegeződős: „Hi Kate”. Ha az évek száma nagyobb tizenhétnél, akkor a függvény névlista alapján dönti el, hogy a név férfi vagy női, és a „Dear Mr. Smith” vagy a „Dear Ms. Smith” formát ölti a megszólítás.

A feladatot bontsuk részekre, írjunk az egyes részekhez függvényeket, eljárásokat!

a) Feltételezzük, hogy az angol név két szóból áll, amelyikből az első a keresztnév, a második a vezetéknév. Írjuk meg a `string keresztnev(string angol_nev)` függvényt és a `string vezeteknev(string angol_nev)` függvényt!
b) A korábbi feladatokhoz hasonlóan írjuk meg a `noi(nev)` és `ffi(nev)` – logikai értéket visszaadó – függvényeket! Az angol keresztneveket a függvényeken belül adjuk meg!
c) A a) és b) részfeladatokban megírt függvények felhasználásával írjunk felnőttnek szánt megszólítást, ami a keresztnévtől függően `"Dear Mr. " + vezeteknev(angol_nev)`, illetve `"Dear Ms. " + vezeteknev(angol_nev)` legyen! Egészítsük ki a feladatot úgy, hogy ha a keresztnév alapján nem derül ki a megszólítandó neme, akkor a `"Dear " + angol_nev` legyen a megszólítás!
d) Készítsük el a megszólítás programot, amely a 17 év alattiakat tegeződő stílusban köszönt, az idősebbek esetén a c) feladatban elkészített hivatalos formát alkalmazza!

11. A vogon költészetről az irodalom egyik, mára klasszikussá vált sci-fijéből, a Galaxis útikalauz stopposoknak című műből szerzett tudomást az emberiség. Ha el akarunk benne mélyedni, ám tegyük, de most elég annyit tudni róla, hogy borzasztó. Mi is hasonlóan borzasztó verseket állítunk elő a következő programunkkal.
A versek sorai mindig egy jelzőből, egy tárgyból, egy állítmányból, valamint mondat végi írásjelből állnak („Nyekergő ablakokat dobálunk!”, „Gömbölyű kútkávát eszünk?”)

a) Írjunk meg egy olyan, `verssor()` nevű, paraméter nélküli függvényt, amely egy ilyen verssort ad vissza, a mondat négy részét négy listából véletlenszerűen válogatva!
b) Az a) feladatban írt függvényt hívja egy másik, `versszak()` nevű függvény. A `versszak()` paramétere egy szám, ami megadja, hogy hány sorból álljon a versszak. Ez a függvény egy teljes versszakot ad vissza egy szöveg típusú változóban.
c) Az utolsó függvényünk neve: `vers()`. Az első paramétere a versszakok számát megadó szám, a második a verssorok száma versszakonként, visszatérési értéke a teljes vers.
d) A verseink túlcsorduló érzelmekről tesznek tanúbizonyságot, legalábbis az írásjelek alapján. Módosítsuk a programunkat úgy, hogy a mondatok végére gyakrabban kerüljön pont!
e) *Kihívást jelentő feladat:* Írjuk át úgy a `vers()` függvényt, hogy paraméterként egy listát várjon! Ha a lista `{2, 2, 4}`, akkor olyan verset adjon vissza, amelyik három versszakból áll, és a versszakok rendre 2, 2 és 4 sor hosszúak!

Az alábbi megoldás több nyelvi specialitást tartalmaz, ezért a megoldás az itt olvashatónál jelentősen hosszabb lehet. Tanulmányozása a saját megoldás elkészítése után javasolt.

```
1.  #include <iostream>
2.  #include <cstdlib>  //random
3.  #include <ctime>    //random inicializálás
4.  #include <vector>   //az a) és e) feladathoz
5.  using namespace std;
6.
```

```
 7.  string verssor()
 8.  {
 9.    vector<string> mondat[3] ={  //vector-ok tömbje
10.    {"Nyekergő", "Gömbölyű", "Piros", "Vidám", "Iszonytató"},
11.    {"amőbát", "ablakokat", "sárgolyót", "kútkávát"},
12.    {"dobálunk", "eszünk", "álmodunk", "éneklünk"}    };
13.    vector<string> vege{".", "?", "!", "?!"};
14.    string verssor = "";
15.    for(int i = 0; i < 3; i++)   //véletlen választások az indexek közül
16.      verssor += mondat[i][rand() % mondat[i].size()] + " ";  //2D index
17.    return verssor + '\b' + vege[rand() % vege.size()]; //backspace jel
18.  }
19.
20.  string versszak(int sorDB)
21.  {
22.    string vszak = "";
23.    for(int i = 0; i < sorDB; i++)
24.      vszak += verssor() + '\n';
25.    return vszak;
26.  }
27.
28.  string vers(int versszakDB, int sorDB)
29.  {
30.    string vers = "";
31.    for(int i = 0; i < versszakDB; i++)
32.      vers += versszak(sorDB) + '\n';
33.    return vers;
34.  }
35.
36.  string szabadvers(vector<int> sorDB_lista)
37.  {
38.    string vers = "";
39.    for(int db : sorDB_lista)    //listát bejáró, azaz foreach-ciklus
40.      vers += versszak(db) + '\n';
41.    return vers;
42.  }
43.
44.  int main()
45.  {
46.    setlocale(LC_ALL,"");
47.    srand(time(NULL));
48.    cout << "Vers:\n" << vers(3, 4) << endl;
49.    cout << "Szabadvers:\n" << szabadvers({2, 2, 4}) << endl; //lista {…}
50.    return 0;
51.  }
```

VARIÁCIÓK TÍPUSALGORITMUSOKRA

Mik azok a típusalgoritmusok?

Egészen egyszerűen olyan algoritmusok, amelyekkel a programírás során gyakran felmerülő feladattípusok megoldhatók. Más néven programozási tételeknek is nevezik őket. Könyvünkben hat-hét egyszerűbb algoritmussal ismerkedünk meg behatóan. Ezek közül négy már a korábbi feladatokból ismerős lehet, mivel az eddigi feladatokban is „mindennapi” kérdésekre kerestük a választ.

Kódolási, jegyzetelési praktikák

Az eljárások és függvények gyakorlása során láthattuk, hogy a feladatok két részre oszthatók: 1. adatok bekérése 2. feladat megoldása – az adatokkal kapcsolatos kérdésre a válasz előállítása. Most csak az utóbbira fókuszálunk, ezért a feladatokban az adatok beolvasása helyett teszt adatokat tárolunk egy globális változóban.

Az egyes feladatok megoldását írhatjuk egyetlen programon belül függvényként. A `main()` függvényben ilyenkor elegendő csak az éppen megírt feladat megoldását kiírni. Természetesen függvényírás nélkül is megoldhatók a feladatok, de ebben az esetben a kódunk egyre kevésbé lesz áttekinthető, ezért vagy kiírással, vagy kommenttel választjuk el egymástól a feladatokat.

A munkánkat könnyíti, ha az éppen nem használt kódrészleteket megjegyzéssé alakítjuk. Ehhez a `//` jelölést érdemes használni, amit az `Edit/Comment` menüponttal vagy CTRL+SHIFT+C billentyűkombinációval egyszerre minden kijelölt sor elé be lehet tenni, illetve az `Uncomment` menüponttal, CTRL+SHIFT+X billentyűkombinációval lehet törölni. Emellett a megoldáshoz írt jegyzetekhez a `/**/` megjegyzési formát alkalmazzuk, hogy elkerüljük a keveredést.

Végső soron az is megoldás lehet, hogy minden feladatnak külön programot írunk, de ezt a végén – a sok kódfájl miatt – nehezebb áttekinteni, nehezebb a különböző megoldásokat összehasonlítani.

A kód rendezettsége, olvashatósága sok szerkesztés következtében romolhat. A fejlesztőkörnyezet automatikus kódformázással segíti a kód átláthatóságának megőrzését, de ha írás közben ez nem sikerül, akkor külön kérhetjük a helyi menüből (jobb-klikk) `Format use AStyle` kiválasztásával.

A példák megoldásainak részletei többféle módon lehetnek elrendezve, innentől a sorszámozás nem követi a kódfájlban várható értéket, a jegyzetben minden kódrészlet számozása 1-től indul.

Történetek a taxisról meg a rókáról

A feladatokhoz két tesztadatsort használunk.

- A taxisról szóló történethez tároljuk, hogy a taxis egy nap során hány piculát keresett az egyes fuvarjaival.
- A rókáról szóló mesénkben a róka libát lop a faluból. A libák súlyát – pontosabban tömegét – adjuk meg.

Az adatokat tárolhatjuk …

tömbben:

```
int bevetelek[5] {1,5,2,3,4};
int bDB = 5;

int libak[5] {1, 5, 2, 3, 4};
int lDB = 5;
```

vagy `vector<int>` listában (#include …):

```
vector<int> bevetelek{1,5,2,3,4};

vector<int> libak{1, 5, 2, 3, 4};
```

A sorozatszámítás – összegzés és átlagolás

1. A taxisnak mennyi lett a napi bevétele összesen?
   A megoldáshoz szükségünk lesz egy változóra, amiben tároljuk a pillanatnyi bevételt. Ez a nap elején 0 picula, majd minden fuvar után növekszik. Minden fuvart figyelembe kell venni, ezért szükségünk lesz egy ciklusra, amellyel az adatsor összes elemét elérjük.

```
összes := 0
ciklus i = 0-tól a bevetelek_számáig:
    összes = összes + bevetelek[i]
ciklus vége
ki: összes
```

```
1.  int osszes = 0;
2.  for (int i = 0; i < bevetelek.size(); i++)
3.    osszes += bevetelek[i];
4.  cout << "Napi bevétel " << osszes << " picula." << endl;
```

   Ugyanez listán bejárós ciklussal:

```
1.  int osszes = 0;
2.  for (int ez : bevetelek)
3.    osszes = osszes + ez; /*így is lehet*/
4.  cout << "Napi bevétel " << osszes << " picula." << endl;
```

   Ugyanez feltételes (while-) ciklussal, tömbre:

```
1.  int osszes = 0;
2.  int i = 0;
3.  while (i < bDB) {
4.    osszes += bevetelek[i];
5.    i++;
6.  }
7.  cout << "Napi bevétel " << osszes << " picula." << endl;
```

Az algoritmust olyan gyakran használják, hogy listára az **algorithm** csomagban van kész függvény is. Angolul SUM(), magyarul SZUM() néven, a táblázatkezelőben is – jellemzően a számításokra képes alkalmazásokban – megtaláljuk az összegző függvényt, amely ezt az algoritmust használja. Csakhogy ez az egyszerű megoldás nem minden esetben használható. Ha az összegzésbe nem vennénk be minden elemet, akkor az algoritmusunk is módosul.

2. A róka libát lop a faluból, de a farkas a dűlőútnál várja a rókát és a három kilogrammnál nehezebb libákat elveszi, míg a kisebbeket – nagylelkűen – otthagyja a rókának. Hány kilogramm libát ehet meg a róka?
   Az előző feladat algoritmusát felhasználva, azt kiegészítve írjuk meg azt az algoritmust, amelyik ezt a problémát oldja meg!

```
összes := 0
ciklus i = 0-tól a libak_számáig:
    ha liba[i] <= 3 :
      összes = összes + liba[i]
    elágazás vége
ciklus vége
ki: összes
```

Az előző feladat alternatív megoldásai közül például a középsőt átírva:

```
1.  int osszes = 0;
2.  for (int ez : libak)
3.    if ( ez <= 3)
4.      osszes += ez;
5.  cout << "A rókának " << osszes << " kilogramm liba marad." << endl;
```

Bár ezt a feladatot nem lehet egyszerű összegzéssel megoldani, azért egy táblázatkezelőben találunk rá függvényt. A magyarul SZUMHA() és SZUMHATÖBB(), angolul SUMIF() és SUMIFS() függvények az ilyen típusú, egy vagy több feltételtől függő összegzések megoldására adnak módot. Csakhogy ezeknek a függvényeknek is megvan a korlátjuk. Az alábbi feladat megoldására – például – nem alkalmasak.

3. *Kitérő:* A róka rájött, hogy több jut neki – és kevesebb munkájába kerül – ha háromkilós libákat lop. De a farkas sem hagyta annyiban. Az a megegyezés született, hogy a rókának csak a három kilónál kisebb libák maradnak meg, valamint – ösztönzésképpen – az ötkilós libákat is megtarthatja a róka. Hány kiló libát ehet meg ezután a róka?

```
1.  int osszes = 0;
2.  for (int i = 0; i < lDB; i++) /*vector<int> esetén libak.size()*/
3.    if ( libak[i] < 3 || libak[i] == 5)
4.      osszes += libak[i];
5.  cout << "A rókának " << osszes << " kilónyi liba marad." << endl;
```

A megoldás az alternatív feltételek (|| VAGY) miatt lesz túl összetett, ha táblázatkezelő függvénnyel szeretnénk megoldani. Még nehezebb lenne táblázatkezelőben a megoldás akkor, ha a farkas minden harmadik libát és a túlsúlyosakat (5 kilósnál nehezebbeket) venné el. A programunkban – a sorozatszámítás algoritmusában a feltétel bármilyen lehet, az algoritmus lényegét nem módosítja.

4. Átlagosan hány piculát keres a taxisunk egy fuvarral?
   Az előző taxis feladat algoritmusán csak annyit kell módosítanunk, hogy az összeget elosztjuk a lista elemszámával. Ha az előző taxisról szóló feladat megoldásából indulunk ki, akkor a végén a kapott összeget el kell osztani a fuvarok számával.

```
1.  int osszes = 0;
2.  for (int i = 0; i < bevetelek.size(); i++) /*vagy … < bDB*/
3.    osszes += bevetelek[i];
4.  int atlag = osszes / bevetelek.size()
5.  cout << "Az átlagos bevétel " << atlag << " picula." << endl;
```

Az átlag számításakor megfontolandó, hogy milyen típusú szám legyen az eredmény. A picula valószínűleg értelmes egész értékként, de a libahús kilogrammjának – főleg egy róka esetén – a tört része is számít.

Ha az összegzés feltételtől függ, akkor nem tudjuk, hogy hány adatból számoljuk majd az átlagot, ezért az összeggel a bevett adatok számát számlálni kell.

5. Átlagosan hány kilósak a rókának maradt libák, ha a háromkilósnál nagyobbat a farkas elvette?

```
int osszes = 0;
int db = 0;
for (int ez : libak)
  if ( ez <= 3) {
    osszes += ez;
    db++;
}
double atlag = (double) osszes / db;
cout << "A róka libái átlagosan " << atlag << " kilósak." << endl;
```

Az átlagszámítást is tudja a legtöbb számítást végző alkalmazás és a táblázatkezelőkben a feltételes átlagszámításra is van függvény: ÁTLAG(), ÁTLAGHATÖBB(), AVG(), AVGIFS(). A feltételek módosítása az előre elkészített függvények használata esetén jelentősen megnehezítheti a megoldást, míg a programkódban csak néhány részlet módosul.

Eldöntés

Megnézzük, hogy van-e adott tulajdonságú elem a listánkban.

Ilyen típusú feladat volt az is, amikor megnéztük, hogy egy betű megtalálható-e a magánhangzók között, egy név szerepel-e a lányok nevei között… A feladattípus tehát nem új, de a hatékony (idő- és memóriatakarékos) megoldása több megfontolást igényel.

Mivel a válasz alapvetően kétféle lehet – ami a logikai típusnak felel meg –, ezt a feladattípust függvényként érdemes megírni. Felhasználása jellemzően egy elágazás feltételében történik, a visszaadott értéktől függ, hogy az igaz vagy a hamis ág fog-e teljesülni.

1. Volt-e a taxisnak ma ötpiculás bevételű fuvara?
   Ezúttal az a feladatunk, hogy addig vizsgálgatjuk ciklussal az értékeket, amíg meg nem találjuk az első olyat, amelyik eleget tesz a feltételnek – ami a mi esetünkben öt.

```
van5 függvény
    vanilyen := hamis
    ciklus i = 0-tól a bevetelek_számáig:
     ha bevetel[i] = 5:
       vanilyen := igaz
     elágazás vége
    ciklus vége
vissza: vanilyen
függvény vége
```

```
eljárás részlet

    ha van5:
     ki: "van ilyen"
    különben:
     ki: "nincs ilyen"
…
eljárás vége
```

```
1.  bool van5()
2.  {
3.    bool vanilyen = false;
4.    for (int i = 0; i < bevetelek.size(); i++) /*lehet másféle ciklus*/
5.      if (bevetelek[i] == 5)
6.        vanilyen = true;
7.    return vanilyen;
8.  }

      /*felhasználás*/
      if (van5())
        cout << "Volt 5 piculás bevétel." << endl;
      else
        cout << "Nem volt 5 piculás bevétel." << endl;
```

Az algoritmusunk megoldja a problémát, de picit pazarló, hogy nem takarékoskodik a gép erőforrásaival. Végignézi ugyanis az összes értéket a listában, még azt követően is, hogy talált már a feltételnek megfelelőt. Persze ezúttal ez nem gond, de mi van, ha a taxis összes bevétele ott van a listában, mondjuk az előző harminc évről? Szerencsére erre is van megoldás. Talán már azt is megszoktuk, hogy több megoldás is van. Ez az egyik:

```
van5 függvény
    vanilyen := hamis
    i := 0
    ciklus amíg i < bevetelek_száma ÉS NEM vanilyen:
     ha bevetel[i] = 5:
       vanilyen := igaz
     elágazás vége
     i := i + 1
    ciklus vége
vissza: vanilyen
függvény vége
```

Látható, hogy ebben a megoldásban olyan ciklust kell használnunk, ahol a ciklusba belépés feltételhez köthető. Ilyen a while-ciklus, illetve a C++ for-ciklusa is.

Először az előző kódot módosítsuk úgy, hogy a ciklusba csak akkor lépjen be a program, ha a `vanilyen` változó értéke **`false`**. Ekkor a tagadása, a `!vanilyen` igaz, azaz az értéke **`true`**.

```
1.  bool van5()
2.  {
3.    bool vanilyen = false;
4.    for (int i = 0; i < bevetelek.size() && !vanilyen; i++)
5.      if (bevetelek[i] == 5)
6.        vanilyen = true;
7.    return vanilyen;
8.  }
```

NEM vanilyen, azaz nincs ilyen

Ugyanez while-ciklussal, ahogy a mondatszerű leírásban is látható:

```
1.  bool van5()
2.  {
3.    bool vanilyen = false;
4.    int i = 0;
5.    while (i < bevetelek.size() && !vanilyen)
6.    {
7.      if (bevetelek[i] == 5)
8.        vanilyen = true;
9.      i++;
10.   }
11.   return vanilyen;
12. }
```

Ez a megoldás nemcsak jó eredményt ad, de futási időt tekintve is jó. Azonban a változók számán lehetne spórolni. A `vanilyen` változó nélkül is megoldható a feladat az alábbi gondolkodásmóddal.

Eldöntés algoritmusa

Az adatsor első elemétől indulva addig lépkedünk a következőre, amíg az éppen vizsgált elem bár létezik, de nem megfelelő tulajdonságú. Így a továbblépést akkor fejezzük be, ha már nincs több elem, amit megvizsgálhatnánk (és eddig egyik sem volt megfelelő), vagy, ha egy létező, éppen vizsgált elem megfelelő. Azaz, ha `i < bevetelek_száma`, akkor a válaszunk az, hogy „van”, ha `i >= bevetelek_száma`, akkor a válaszunk „nincs”.

```
van5 függvény
    i := 0
    ciklus amíg i < bevetelek_száma ÉS NEM (bevetel[i] = 5):
       i := i + 1
    ciklus vége
    ha i < bevetelek_száma:
       vissza: igaz
    különben
       vissza: hamis
függvény vége
```

Programunk szempontjából létfontosságú, hogy **minden olyan feltételnél** – mint itt is –, **amiben egy adatsorozat elemét az indexén keresztül érjük el** – a `bevetel[i]` elemet az `i` indexen keresztül érjük el –, **először azt kell ellenőrizni kell, hogy az adott indexű elem létezik-e** – azaz, az `i < bevételek_száma` az első feltétel –. Ha nem tartjuk be a helyes sorrendet, akkor programunk futtatása akár „kékhalál”-t is okozhat.

Az algoritmus végén a választ érdemes táblázatban megvizsgálni: a visszatérési érték megegyezik a feltétellel. Ilyenkor felesleges az elágazás, mert a feltétel maga lesz a válasz.

i < bevetelek_száma	vissza:
igaz	igaz
hamis	hamis

```
1.  bool van5()
2.  {
3.    int i = 0;
4.    while (i < bevetelek.size() && !(bevetelek[i] == 5))
5.      i++;
6.    return i < bevetelek.size();
7.  }
```

Eddig a feladatra láthattunk 7, 8 és 11 sorból álló megoldást, mindegyik jó. Minél hosszabb egy megoldás, annál részletesebben vannak kifejtve az egyes lépések. De a programozók (és a matematikusok) ahol csak lehet, inkább rövidítenek a kódon, például a while-ciklus helyett inkább a for-ciklust használják. Ez a megoldás csak 6 soros (a lényegi része 3 sor):

```
1.  bool van5()
2.  {
3.    int i;
4.    for (i = 0; i < bevetelek.size() && bevetelek[i] != 5; i++) ;
5.    return i < bevetelek.size();
6.  }
```

*Itt most kell a **pontosvessző**!*

A megoldás rövid, de velős … Figyeljük meg, miben változott:

- Az „i” kezdőértékét a for-ciklus fejében állítjuk, de az `i`-t előtte (3. sor) kell deklarálni, hogy a visszatérésnél (5. sor) is létezzen.
- A léptetés is bekerült a for-ciklus fejébe, de így nem maradt semmi a ciklusmagban. Ezt a ciklusfej után írt pontosvesszővel jelezzük (4. sor vége). A ciklust általában körnek képzeljük el – vagy esetleg ellipszisnek –, de itt a kör egy szakasszá lapult, a program a feltétel és a léptetés között „pattog”.
- Lerövidíti a kódot, de az érthetőségén is ront, ha a keresett tulajdonság helyett az ellentettjét írjuk, zárójel és tagadás nélkül. Egyszerű állításnál – mint itt – ez szinte természetes, de egy összetettebb feltétel esetén matematikus legyen a talpán, aki minden szempontból helyesen adja meg a feltételt.

Eldöntés „kiugrással”

Az első megoldásunkban végignéztük az összes adatot, emiatt nem volt hatékony az a megoldás. Az eldöntés algoritmusa a feladat újragondolását igényelte, de a másként gondolkodás nehéz. Új megoldási mód keresése helyett könnyebb a már meglévő gondolatot javítgatni. Ezért sok programozási nyelv lehetőséget ad arra, hogy egy ciklusmagon belül megváltoztassuk eredeti szándékunkat és a végrehajtandó utasítássorozatot megszakítsuk. A megszakításnak kétféle módja van.

Az egyik megszakítási mód olyan, mintha az iskolában úgy lépne valaki a következő évfolyamba, hogy az év utolsó hónapjait kihagyja. Ezt a **`continue;`** utasítással lehet elérni. Hatására arról a pontról, ahol a ciklusmagon belül kiadjuk, a ciklusfejre ugrik a programunk. For-ciklus esetén lépteti a ciklusváltozót és ellenőrzi a ciklusfeltételt, while-ciklus esetén rögtön a ciklusfeltételt ellenőrzi.

A másik megszakítási mód olyan, mintha az iskolát valaki az osztályok kijárása nélkül befejezné. Nem csak az adott ciklusmag végrehajtását szakítja meg, hanem a ciklikus folyamatot is. Ezt a **`break;`** utasítással lehet elérni.

Bármelyik kiugrási utasítást használjuk, programozóként figyelnünk kell arra, hogy a kiugrás minden lehetséges futási helyzetben biztonságos legyen, ne legyen nem kívánt mellékhatása.

Ezt leginkább úgy érhetjük el, hogyha ezt a két lehetőséget csak meghatározott, átlátható helyzetekben használjuk. Nézzük a **break;** használatát az eldöntés feladatban.

Egy eldöntési feladatot megoldhatunk úgy is, hogy ha találunk egy feltételnek megfelelő elemet az adatsorozatban, akkor az eredmény rögzítése után kiugrunk a ciklusból. Erre a megoldásra mondatszerű leírásban nincs nyelvi megoldás, de C++ nyelvben használhatjuk a **break;** utasítást.

```
1.  bool van5()
2.  {
3.    bool vanilyen = false;
4.    for (int i = 0; i < bevetelek.size(); i++) /*lehet bejárós is*/
5.      if (bevetelek[i] == 5) {
6.        vanilyen = true;
7.        break;
8.      }
9.    return vanilyen;
10. }
```

```
    /*felhasználás*/
    if (van5())
      cout << "Volt 5 piculás bevétel. " << endl;
    else
      cout << "Nem volt 5 piculás bevétel. " << endl;
```

Ha a for-cikluson belül csak egy elágazás van és annak nincs „különben” ága, valamint a **break;** az igaz-ág végén van, akkor a kiugrás biztonságos.

De, ha már ugrunk, ugorjunk nagyot! A `vanilyen` változó csak azért kell, hogy egyetlen helyen, a függvény végén fejeződjön be a függvényünk futása. Mivel a for-ciklus után nincs más, csak a függvény vége, a **break;** után a **return** jön. A kódunk 6. és 7. sora a 9. sorral egyetlen utasításban megad egy részmegoldást, azt, amikor a `vanilyen` értéke igaz, azaz helyettesíthető a **return true;** utasítással. A végére csak a **return false;** marad, a `vanilyen` változóra pedig nincs szükség.

```
1.  bool van5()
2.  {
3.    for (int i = 0; i < bevetelek.size(); i++) /*lehet bejárós is*/
4.      if (bevetelek[i] == 5)
5.        return true; /*megvan, kiugrik a ciklsuból és a függvényből*/
6.    return false; /*végignézte, de nem talált*/
7.  }
```

Ezt a rövid, de kifejező kódot szeretik azok használni, akik úgy gondolkodnak, hogy „végig nézzük, és ha találunk, akkor ott visszaadunk egy **true**-t, vagy a végén visszaadunk egy **false**-t”.

2. Előfordult-e olyan, hogy a róka legalább háromkilós libát lopott?
   Nézzük meg, hogy a taxis bevételére használt algoritmus a rókával kapcsolatos kérdések megoldására is használható-e! A függvényeket a főprogramban használjuk fel: ott írjuk ki a kérdést és a választ is!

A függvényre egy részletes megoldás:

```
1.  bool voltnagy()
2.  {
3.    bool vanilyen = false;
4.    int i = 0;
5.    while (i < libak.size() && !vanilyen) { /*tömbben … < lDB*/
6.      if (libak[i] >= 3)
7.        vanilyen = true;
8.      i++;
9.    }
10.   return vanilyen;
11. }
```

Módosult a függvény neve, az adatsor neve és a feltétel: `libak[i] >= 3`. A taxis bevételére adott bármelyik megoldást nézzük, ezeket a kódrészleteket kell módosítanunk. Válasszuk ki kedvenc kódunkat és ezzel oldjuk meg a feladatot!

3. Előfordult-e olyan, hogy a róka kisebb libát hozott, mint az előző napon?
   Itt már kicsit jobban kell figyelni, mert az „előző nap” az első napra nem értelmes. Ezért a 0 indexű adat helyett, az 1 indexűvel kell kezdeni a vizsgálatot:

```
1.  bool elozonelkisebb()
2.  {
3.    int i = 1;
4.    while (i < libak.size() && ! (libak[i] < libak[i - 1]))
5.      i++;
6.    return i < libak.size();
7.  }
```

Kiválasztás

Az eldöntéshez képest, ha kiválasztás a feladat, akkor annyi a különbség, hogy tudjuk, hogy van adott tulajdonságú elem (például létezik ötpiculás bevétel, háromkilósnál nagyobb liba) a listában. Kérdés, hogy hol van ez az elem? Hányas sorszámú helyen áll?

1. A taxis bevételei közül hányadik volt az ötpiculás?

A kiválasztás az algoritmusunkban az eldöntés algoritmusához hasonlóan, az adatsor elejéről indulva addig lépkedünk, amíg az éppen vizsgált elem nem megfelelő tulajdonságú. Lépkedésünk biztosan megáll valamelyik elemnél – mert tudjuk, hogy van megfelelő elem –; ahol megállunk, az lesz jó, annak az elemnek az indexére lesz szükségünk.

A válaszban csak arra kell figyelnünk, hogy az adatsorunk indexelése 0-tól indul, miközben a feladatban – magyar hagyománynak megfelelően – az adatsor kezdete az első adat.

Nézzük, hogyan módosul az eldöntés algoritmusa, ha kiválasztásról van szó:

```
holvan5 függvény
    i := 0
    ciklus amíg ~~i < bevetelek_száma~~ ~~ÉS~~ NEM (bevetel[i] = 5):
       i := i + 1
    ciklus vége
    ~~ha i < bevetelek_száma:~~
    ~~   vissza: igaz~~
    ~~különben~~
    ~~   vissza: hamis~~
    vissza: i
függvény vége
```

Röviden:

```
holvan5 függvény
    i := 0
    ciklus amíg NEM (bevetel[i] = 5):
       i := i + 1
    ciklus vége
    vissza: i
függvény vége
```

Kódolva és felhasználva a főprogramban:

```
1.  int holvan5()
2.  {
3.    int i = 0;
4.    while (!(bevetelek[i] == 5)) //vagy: while (bevetelek[i] != 5)
5.      i++;
6.    return i;
7.  }

      /*felhasználás*/
      cout << "Az ötpiculás fuvar sorszáma: " << holvan5() + 1 << endl;
```

Megszoktuk már, hogy a taxis után jön a róka a libáival. Hogy ne legyen annyira unalmas, nézzük meg, hogy az eldöntés szuperrövid megoldása hogyan alakítható át kiválasztásra!

2. Hányadik napon sikerült a rókának először legalább háromkilós libát lopnia?

```
1.  int elsonagy()
2.  {
3.    int i;
4.    for (i = 0; liba[i] < 3; i++);
5.    return i;
6.  }
```

Ennyi … De! Attól, hogy egy kód rövid, még nem lesz könnyebben megtanulható vagy megérthető, mert benne van minden gondolat, minden megfontolás, ami a hosszabb kódban megtalálható. Akkor érdemes rövidített írásmódra áttérni, ha a gondolatunk előrébb jár, azaz gyorsabb, mint az ujjaink.

Keresés

A keresés algoritmusa nem más, mint az eldöntés és a kiválasztás egybeépítése. Az eldöntésnél az a kérdés, hogy van-e olyan elem a listában, amit keresünk, a kiválasztásnál pedig az, hogy hányadik ez az elem (mert azt már tudjuk, hogy van olyan). Itt mindkét kérdésre válaszolunk.

A „mindkét kérdésre válaszolás” miatt a keresés feladatokat könnyebb eljárásként írni, mert egy függvény csak egyféle adattípust tud visszaadni. Az eddigi példákban a kiválasztott érték egy index volt, ami egy egész számot jelent, de lehetne az eredmény az adatsor egy eleme is, ami lehet bármilyen szám vagy szöveg, vagy – egy igazi alkalmazásban – bármilyen virtuális objektum. Ezzel szemben, ha a keresett elem nem található, akkor nagy kérdés, hogy hogyan lehet olyan értéket visszaadnia a függvényünknek, ami a keresett adattal azonos típusú, de egyértelmű, hogy nem létezik.

Kitérő: a programozók dilemmája

Minden olyan programban, ahol részfeladatként megjelenik a keresés, a programozónak el kell döntenie: hogyan fogja a keresés függvénye jelezni, hogy nincs találat.

A problémát mi most úgy oldjuk meg, hogy eljárásként írjuk meg a megoldást. Ezzel lényegében úgy döntünk, hogy az eredmény – keressünk bármilyen adatot – végül is egy, a képernyőre kiírt szöveg lesz. De ha a keresés eredménye nem végeredmény, ha máshol fel szeretnénk használni, akkor ez nem megoldás. Mit lehet tenni?

- Táblázatkezelésben a keresés függvénye például a HOL.VAN() – angolul MATCH() –, de van keresési algoritmus a FKERES() és a VKERES() függvényekben is. Az újabb alkalmazásokban található az XHOL.VAN() és az XKERES() függvény, amelyek szintén keresést végeznek, bizonyos esetekben a most vizsgált *lineáris keresés algoritmus*t is használják. Ezek a függvények a találat sorszámát 1-től számítva adják meg, ha nincs találat, akkor a #HIÁNYZIK hibajelzést adják. Ha ezt az eredményt máshol fel szeretnénk használni, akkor a hibajelzés alapján speciális értéket adhatunk a cellának, például legyen üres, vagy 0.
- Néhány programozási nyelvben az adatsorokat 1-től indexelik. Ilyen például a Pascal. Ezeken a nyelveken írt programokban a nemlétező elem indexe lehet a 0. Ennek mintájára, azokban a programozási nyelvekben, ahol 0-tól indexelik az adatsorozatot, dönthet úgy a programozó, hogy 1-től tárolja az adatokat és a 0. indexű elem a nemlétező elem.
- A C alapú nyelvekben az adatsorokat 0-tól indexelik. Gyakori, hogy a programozó úgy dönt, hogy ha a keresésnek nincs eredménye, akkor egy negatív szám – jellemzően –1 – lesz a függvény által visszaadott érték. A negatív érték egyértelműen hibás válasz a „hányadik elem” kérdésre, könnyen kezelhető a további felhasználás során.
- A negatív visszaadott érték egyértelműen hibás, de a függvényen belül ezt külön be kell állítani. A keresés eredménytelenségét úgy is lehet jelezni, hogy a lehetséges indexértékeknél nagyobb értéket adunk meg. Az első ilyen érték a 0-tól indexelt tömb esetén az adatsor elemszáma. Persze ilyenkor a felhasználás során nagyon kell figyelni arra, hogy a kapott eredmény létező index-e. Ezért azt a programozót, aki ilyen függvényt ír, nem kedvelik a munkatársai.
- Gyakori megoldás, hogy az adatsorozat végére „strázsát” vagy végjelet tesz a programozó, azaz az utolsó adat után, az elemszámadik helyen egy olyan adat van, amelyet a későbbiekben „üres” adatként lehet használni. A string adattípusú „üres szöveg” lehet egyetlen

karakter, a „lezáró nulla”. Fizikailag az a 0 ASCII kódú karakter, a jele: `'\0'`. Beolvasáskor viszont az ENTER leütése hatására kerül a szövegbe a végjel. Ehhez hasonlóan lehet egy lista végére valamilyen, a keresés függvény felhasználása során értelmesnek tűnő, de nem valódi elemet vagy rá mutató hivatkozást tenni. Egyes esetekben ezt jelöli a NULL érték.

1. Keressük azt a fuvart, amikor a taxis először keresett öt piculát.
Az eldöntés és a kiválasztás algoritmusának ismeretében alkossuk meg az algoritmust, majd kódoljuk.

Az első eldöntésre írt algoritmust alakítsuk át és adjunk a kiválasztáshoz egy változót:

```
hol5 függvény
    vanilyen := hamis
    holvan := -1
    ciklus i = 0-tól a bevetelek_számáig:
     ha bevetel[i] = 5:
      vanilyen := igaz
      holvan := i
     elágazás vége
    ciklus vége
    ha vanilyen:
     ki: holvan
    különben:
     ki: "nincs ilyen"
    elágazás vége
eljárás vége
```

Látható, hogy a `vanilyen` és a `holvan` változók egy dologról szólnak. Matekosan: A `vanilyen` akkor és csak akkor hamis, ha a `holvan` értéke −1. Mivel a válaszban a `holvan` értékét kell megjeleníteni, a `vanilyen` változó felesleges:

```
hol5 függvény
    holvan := -1
    ciklus i = 0-tól a bevetelek_számáig:
     ha bevetel[i] = 5:
      holvan := i
     elágazás vége
    ciklus vége
    ha holvan <> -1:
     ki: holvan
    különben:
     ki: "nincs ilyen"
    elágazás vége
eljárás vége
```

Ez az algoritmus egész jónak tűnik, de mégsem az. Az eldöntés algoritmusában az algoritmus hatékonyságát rontotta, hogy végignéztük az összes adatot akkor is, ha már az elején kiderült, hogy van megfelelő elem. A keresésnél azonban ez a lazaság már hiba, mert hibás eredményt adhat a függvényünk. Ha az adatsorban több megfelelő elem van, akkor az algoritmusunk minden megfelelő elemnél átállítja a `holvan` értékét és így az adatsor végére érve nem az első találatot adja, hanem az utolsót. (A feladat az első ötpiculás fuvarra kérdez rá.)

Feltételes ciklussal – és csak addig vizsgálva az adatokat, amíg nincs találat – az index értékéből megmondhatjuk az eredményt, a `holvan` változó is felesleges:

```
hol5 eljárás
    i := 0
    ciklus amíg i < bevetelek_száma ÉS NEM (bevetel[i] = 5):
       i := i + 1
    ciklus vége
    ha i < bevetelek_száma:
       ki: i
    különben
       ki: "nincs ilyen"
eljárás vége
```

Kódolva:

```
1.  void hol5()
2.  {
3.    int i = 0;
4.    while (i < bevetelek.size() && ! (bevetelek[i] == 5)) {
5.      i++;
6.    } // felesleges {}, mert csak 1 utasítás maradt a ciklusmagban
7.    if (i < bevetelek.size())
8.      cout << "Az 5 piculás fuvar sorszáma: " << i + 1<< endl;
9.    else
10.     cout << "Nincs 5 piculás fuvar." << endl;
11. }
```

Mondatszerű leírással továbbra sem lehet, de C++ nyelven megoldható, hogy a ciklusfejben az összes adat végignézését tervezzük, de találat esetén kiugorjunk. Ha függvényt írunk, akkor a visszaadott érték (**return**) a logikai érték helyett a sorszám szokott lenni, illetve, ha nincs a keresett tulajdonságú elem, akkor –1. Eljárásként így kódolhatjuk:

```
1.  void hol5()
2.  {
3.    int ez;
4.    for (ez = 0; ez < bevetelek.size(); ez++) /*tömbre … < bDB*/
5.      if (bevetelek[ez] == 5)
6.        break; //kiugrik, amikor „ez” az elem 5 értékű
7.    if (ez < bevetelek.size())
8.      cout << "Az 5 piculás fuvar sorszáma: " << i + 1<< endl;
9.    else
10.     cout << "Nincs 5 piculás fuvar." << endl;
11. }
```

2. Melyik (hányadik) a róka első legalább háromkilós libája?
   A megoldásban az taxis ötpiculás első keresetének megadásához képest csak néhány helyen kell aktualizálni: másik adatsorozat, más a feltétel, más a kiírt szöveg.

Megszámolás

A megszámolás algoritmusával azt derítjük ki, hogy adott tulajdonságú elemből mennyit találunk a listánkban, adattömbünkben. A lista bejárását végző ciklus előtt létrehozunk egy számláló szerepű változót és a nulla értéket adjuk neki. Minden egyes találatnál növeljük a számláló értékét.

A megszámolás algoritmus egyik speciális esete az, amikor az elem tulajdonsága az, hogy elme az adatsorozatnak, azaz, azt szeretnénk megtudni, hogy hány eleme van az adatsorozatunknak. Lista – a C++ `vector<>` típusok – és `string` típus esetén, az elemek számát a `size()` tulajdonság adja meg. Tömböt használva a `size()` nem az adatok számát adja meg, hanem a rendelkezésre álló helyek számát. A tömbben tárolt adatsorozathoz mindig külön változóban jegyezzük meg az adatsorozat aktuális hosszát.

Amikor a tömböt adatokkal feltöltjük, mellette folyamatosan számláljuk, hogy hány adatot írtunk be. Ez is a megszámlálás speciális esete, de ebben már szerepet kaphat az elem tulajdonsága is. Például számokat kérünk, ha a beírt adat nem szám, akkor „eldobjuk” a kapott adatot és vagy újra bekérjük, vagy ez az adat jelenti az adatbevitel végét. Korábban több ilyen programot írtunk. Hogy is volt?

1. Taxisunknak egész évben jegyeznie kell, hogy a fuvarjaiért hány piculát kapott. Az adatokat naponta írja be, a napi „utolsó fuvar” a garázsmenet, ennek bevétele 0 picula. Írjunk programrészletet, amely egy nap adatait tárolja és a végén kiírja, hogy aznap hány fuvarja volt a taxisnak.
2. Talán még emlékszünk rá, hogy a farkas a három kilónál nagyobb libákat veszi el a rókától. Hány libát tarthat meg a róka?
   Adjunk algoritmust a feladat megoldására, majd készítsük el belőle a kódolt programot!

```
számláló := 0
ciklus i = 0-tól a libak_számáig:
    ha liba[i] <= 3 :
      számláló = számláló +1
    elágazás vége
ciklus vége
ki: számláló
```

```
1.  int db = 0;
2.  for (unsigned i = 0; i < libak.size(); i++)
3.    if (libak[i] <= 3)
4.      db++;
5.  cout << "A rókának " << db << "libája marad." << endl;
```

Amikor a sorozatszámítás algoritmusát átlagszámításra alkalmazzuk, az elemek összege mellett az elemek számát is meg kellett mondanunk. Ezért ott lényegében az összegzés algoritmusát egybeépítettük a megszámlálás algoritmusával. Másképp: a megszámlálás olyan, mintha egyenként végeznénk összegzést.

A megszámlálásnak is van „előregyártott” függvénye a fejlett programozási nyelvekben és adatkezelő alkalmazásokban. Táblázatkezelőben a DARAB() – angolul COUNT() – számokat számlál, másik függvény az üres cellákat, illetve a nem üres cellákat. Egyéb feltételt a DARABTELI() – COUNTIF() – és a DARABHATÖBB() – COUNTIFS() – függvények segítségével végezhetünk, de az összegzéshez hasonlóan, ezek sem alkalmasak alternatív feltételek (ilyen vagy olyan tulajdonságú) esetén.

Kiegészítés: kiválogatás

Adott tulajdonságú elemek megszámolása sokszor kiegészül azzal, hogy a számba vett elemek listáját is szeretnénk megadni. Táblázatkezelő alkalmazásban ezt úgy mondanánk, hogy a megfelelő adatokat szeretnénk kiszűrni.

Az eredmények feljegyzéséhez a lista (tömb) bejárását végző ciklus előtt létrehozunk egy adattárolót, amibe a találatokat eltároljuk. Igénytől függően ebbe a tárolóba bemásolhatjuk a feltételnek megfelelő adatokat vagy – ez a gyakoribb – a helyük indexét. A megoldáshoz sokkal egyszerűbb `vector<>` listát használni, mert nem ismerjük az eredmény elemeinek a számát, emiatt egy tömböt úgy kell méretezni, hogy minden adat beleférjen – még akkor is, ha végül egy vagy talán egy eleme sem lesz.

Az adatokat végignéző cikluson belül nem(csak) a számláló értékét kell növelni, hanem az eredményhez hozzá kell adni a megfelelő adatot. Az eredmény kiírása is összetettebb lesz, mert ciklus kell az összes adat kiírásához.

1. Talán még emlékszünk rá, hogy a farkas a három kilónál nagyobb libákat veszi el a rókától. Hány kilós libák jutnak a rókának?
   Adjunk algoritmust a feladat megoldására, majd készítsük el belőle a kódolt programot!

```
számláló := 0
rókáé : indexek listája
ciklus i = 0-tól a libak_számáig:
    ha liba[i] <= 3 :
      rókáé-hoz ad liba[i]
      számláló = számláló +1
    elágazás vége
ciklus vége
ki: számláló
ciklus i = 0-tól számlálóig
    ki: rókáé[i]
ciklus vége
```

```
 1.  int db = 0;
 2.  vector<int> rokae;
 3.  for (unsigned i = 0; i < libak.size(); i++)
 4.    if (libak[i] <= 3) {
 5.      rokae.push_back(libak[i]);
 6.      db++;
 7.    }
 8.  cout << "A rókának ez a " << db << " liba jutott: ";
 9.  for(int liba : rokae)
10.    cout << liba << " kilós, ";
11.  cout << "\b\b." << endl;
```

Megjegyzések a kódhoz:

- Ha a kigyűjtést `vector<>`-ba végezzük, akkor a `db` változóra nincs szükség, mert a `vector<> size()` függvénye is ugyanezt az értéket adja meg.
- Ha a kigyűjtést tömbbe végezzük, akkor a `db` változó használata nem spórolható meg. A kigyűjtés mindig a tömb `db`-edik helyére történik. (A kódban `rokae[db] = libak[i];` lenne).
- Kiíráskor a listaelemek felsorolásának vége sokszor okoz fejtörést: az elemek között legyen vessző és szóköz, de a lista végén ne. A 11. sorban a `"\b\b."` jelentése: törölj vissza két karaktert (a szóközt és a vesszőt) majd írj egy pontot. A `'\b'` a BACKSPACE billentyű megfelelője.

Maximum- vagy minimumkiválasztás

A kiválasztás tétele arról szól, hogy tudjuk, hogy van adott tulajdonságú elem a listában, de meg kell mondanunk, hogy melyik az. Most is tudjuk, hogy van legnagyobb és legkisebb elem, de nem tudjuk, hogy melyek ezek, ráadásul, azt sem tudjuk, hogy mennyi az értékük. Ez az „apróság" jelentősen módosítja az algoritmusunkat. Míg egy adott tulajdonságú elem kiválasztásához elegendő az első találatig vizsgálni az adatokat, a „legnagyobb", illetve a „legkisebb" tulajdonságok nem adottak, hanem a többi adathoz viszonyítottak. Ezért az adatsorozat összes elemét meg kell vizsgálnunk, hogy megmondhassuk, melyik értékről van szó.

A maximum és a minimum kiválasztása végezhető együtt is, de jellemző, hogy egyszerre csak az egyikre vagyunk kíváncsiak, azért most is egy feladatban csak az egyiket adjuk meg. Az algoritmus elkészítése során az sem mindegy, hogy hogyan szól pontosan a kérdés: Azt akarjuk-e megtudni, hogy *mennyi* volt a legnagyobb/legkisebb érték, vagy azt, hogy *melyik* volt a legnagyobb/legkisebb érték. Az első kérdésre a legtöbb adatkezelő alkalmazás és programozási nyelv biztosít előregyártott függvényt. Táblázatkezelőben ez a MAX() és MIN() függvény. De a második kérdés megválaszolásához a függvények eredményének helyét az adatsorozatban minden esetben meg kell keresni, pontosabban ki kell választani. Táblázatkezelésben a „ki a legjobb" kérdés megválaszolásához legalább két függvényt kell használni, ebből az egyik – MAX() vagy MIN() – sorra veszi az összes adatot, a másik – HOL.VAN(), XKERES() – átlagosan az adatok felét újra megvizsgálja.

Amikor a legkisebbet/legnagyobbat megadó algoritmust és kódot írjuk, akkor hatékony megoldásra törekszünk.

- Ha az a kérdés, hogy *mennyi* a leg …, akkor az adatsorozat egyik elemének az értékét adjuk meg.
- Ha a kérdés úgy szól, hogy *melyik* a leg …, akkor az adatsorozatból annak az elemnek az indexét határozzuk meg, amelyiknek az értéke leg…. Ha több ilyen is van, akkor az elsőt választjuk ki.
- Ha a kérdés az, hogy *melyek* a leg…ek, akkor érdemes kétfelé bontani a megoldást: először meghatározzuk a leg … értéket, majd újra végig nézzük az adatsort és kigyűjtjük azokat az indexeket, amelyekben ez az érték előfordul.

1. Hány piculát kapott a legdrágább fuvarjáért a taxis?
   Írjuk meg az algoritmust, majd kódoljuk! Ötlet: vezessünk be egy változót, aminek a kezdőértéke a nap első fuvarjának díja. Nézzük végig egyesével listánk elemeit, és ha találunk a változóban tároltnál nagyobb értéket, akkor cseréljük erre a változó tartalmát.

A megoldás kivitelezésének a feltétele, hogy legyen a taxisnak első fuvarja. Persze, ha nincs első fuvarja, akkor egyáltalán nincs aznap egyetlen fuvarja sem, így legdrágább sincs. Egy probléma megoldása során erre az „apróságra" is oda kell figyelni, de most feltételezzük, hogy az adatsorozatunknak van eleme.

```
legtobb := bevetelek[0]
ciklus i = 0-tól a bevetelek_számáig:
    ha bevetelek[i] > legtobb :
      legtobb = bevetelek[i]
    elágazás vége
ciklus vége
ki: legtöbb
```

A kódolás C++ nyelven feltételes, számlálós és bejárós ciklussal is lehetséges. Például:

```
1.  int legtobb = bevetelek[0];
2.  for (int ez : bevetelek)
3.    if (ez > legtobb)
4.      legtobb = ez;
5.  cout << "A legdrágább fuvar " << legtobb << " piculát ért." << endl;
```

Jellemzőbb a számlálós ciklus alkalmazása, mivel egy picit javítani is lehet az algoritmuson azzal, ha a ciklust a 2. elemtől kezdjük. Merthogy, minek hasonlítsuk össze az első elemet önmagával, hogy nagyobb-e.

```
1.  int legtobb = bevetelek[0];
2.  for (int i = 1; i < bevetelek.size(); i++)
3.    if (bevetelek[i] > legtobb)
4.      legtobb = bevetelek[i];
5.  cout << "A legdrágább fuvar " << legtobb << " piculát ért." << endl;
```

2. Hányadik volt ma a taxis legdrágább fuvarja? Mennyit kapott ezért a fuvarért?
Most mindkét értéket meg kell adnunk. Ha ismét az értéket határoznánk meg, akkor utána keresni kellene, hogy melyik fuvar volt ez. De, ha az indexet adjuk meg, akkor az érték rögtön tudható: annyit kapott, amennyi az adatsor megadott helyén az érték.

```
maxhely := 0
ciklus i = 1-től a bevetelek_számáig:
    ha bevetelek[i] > bevetelek[maxhely] :
      maxhely = i
    elágazás vége
ciklus vége
ki: maxhely, bevetelek[maxhely]
```

```
1.  int maxhely = 0;
2.  for (int i = 1; i < bevetelek.size(); i++) /*tömbre … < bDB*/
3.    if (bevetelek[i] > bevetelek[maxhely])
4.      maxhely = i;
5.  cout << "A legdrágább fuvar a(z) " << maxhely + 1 << ". ";
6.  cout << "Ezért " << bevetelek[maxhely] << " piculát fizettek."<< endl;
```

Ha a megoldásunkat függvényként írjuk meg, akkor – néha csak az érték meghatározása során is – praktikusabb az index meghatározása, mert a maximum értéke ebből könnyen megadható.

```
1.  int maxh ()
2.  {
3.    int maxhely = 0;
4.    for (int i = 1; i < bevetelek.size(); i++) /*vagy … < bDB*/
5.      if (bevetelek[i] > bevetelek[maxhely])
6.        maxhely = i;
7.    return maxhely;
8.  }
      /*felhasználás*/
      int max = maxh();
      cout << "A legdrágább fuvar az " << max + 1 << ". ";
      cout << "Ezért " << bevetelek[max] << " piculát fizettek."<< endl;
```

3. Mekkora a legkisebb liba, amit a farkas elvesz a rókától?
Itt egy adatelem értékét kell megadni. Az előző feladattól eltérően, nem a legnagyobbat, hanem a legkisebbet, ami csak apró módosítást jelent az algoritmusunkban: akkor jegyezzük meg az aktuális értéket, ha az nem nagyobb, hanem kisebb az eltárolt eddigi legkisebb értéknél. De nekünk most nem egyszerűen a legkisebb liba tömege kell, hanem csak azok közül kell a legkisebb, amelyiket a farkas elvitt: a 3 kilósnál nagyobbak közül a legkisebb.

Ha az első libát a farkas elviszi, akkor kicsit bonyolultabb feltétellel, de használható az előbbi algoritmus. Csakhogy, erre elég kicsi az esély. Úgy is mondhatnánk, hogy csak félmegoldás lenne, ráadásul egy ilyen, hol jó, hol nem jó megoldás rosszabb, mint a hibás megoldás, mert néha jónak látszik.

Ha az első liba a rókánál maradhat, akkor ezt nem választhatjuk feltételezett minimumnak, mert a rókának kisebb tömegű libák maradnak. Esetleg mondhatjuk azt, hogy a farkas elvitt egy 1000 kg-os libát – jó nagy értéket válasszunk, hogy amit elvisz, az biztosan ennél kisebb legyen. Ez a megoldási mód nem biztonságos, mert meg kell tippelni, hogy mi a kellően nagy érték. Ráadásul, mi van akkor, ha egyik liba sem több 3 kilósnál? Akkor a farkas elvitt egy 1000 kg-os libát? Az eredményt mindenképpen ellenőrizni kell.

```
int farkasmin = 1000;
for (int ez : libak)
  if (ez > 3 && ez < farkasmin)
    farkasmin = ez;
if (farkasmin == 1000)
  cout << "A farkas nem vitt el egy libát sem." << endl;
else
  cout <<"A farkas legkisebb libája "<< farkasmin <<" kg."<<endl;
```

Biztonságosabb, de nehezebb megoldás, hogy először megkeressük az első olyan libát, amit a farkas elvitt. Ha van ilyen liba, akkor azt választhatjuk a minimum kezdőértékének, onnantól keresünk farkas által elvitt, de ennél kisebb libát. Azaz: először keresünk, utána kiválasztjuk a legkisebbet. Mivel a keresés eredménye egy index lesz, a minimumhoz is célszerű az indexet (a helyét) megkeresni. Ha az első liba indexe az adatsorozaton túl van, akkor nincs megoldás, egyébként a megadott indexű elem értéke lesz a megoldásunk.

```
void farkas_legkisebb_libaja()
{
  int i = 0;
  while (i < libak.size() && ! (libak[i] > 3)) /*vagy mindenhol lDB*/
    i++;
  if (i == libak.size()) /*végére ért, nem volt nagy liba */
    cout << " A farkas nem vitt el egy libát sem." << endl;
  else //az i-ediket a farkas elvitte, ezután …
  {
    int minid = i; //… tovább nézzük, minimumot keresve
    while (i < libak.size())
    {
      if(libak[i] > 3 && libak[i] < libak[minid])
        minid = i;
      i++;
    }
    cout <<"A farkas legkisebb libája "<< libak[minid] <<" kg."<<endl;
  }
```

A típusalgoritmusok genetikája

Láttunk hatféle típusalgoritmust, de ezek közül a sorozatszámításnak, a megszámlálásnak és a maximum- vagy minimumkiválasztásnak volt egyszerű és feltételes változata; az eldöntés és a keresés esetén kétféle gondolkodási mód miatt kétféle algoritmust láthattunk. Mindegyik algoritmusnak van valami specialitása, mégis, sok hasonlóság is felfedezhető bennük. Ezeket a hasonlóságokat gyűjtjük most össze.

- A sorozatszámítás, a megszámlálás és az egyszerű maximum- vagy minimumkiválasztás ciklusa számlálós vagy bejárós ciklus.
- Az eldöntés, a kiválasztás, a keresés általános megoldásában a ciklus feltételes, nem kell mindig minden adatot megnézni.
- Az eldöntés és a keresés esetén szoktak számlálós ciklust használni kiugrással, de ezt a kiválasztásnál is megtehetjük, akár úgy is, hogy a ciklusba belépés feltétele az index vizsgálata helyett `true`.
- Az egyszerű sorozatszámítás és a megszámlálás algoritmusában a cikluson belül nincs elágazás. De írhatunk bele: `if(true){}` 😊
- A maximum vagy a minimum kiválasztása valójában nem kiválasztás, hanem keresés. A feltételes maximum vagy minimum kiválasztása során nem megkerülhető a „mi van, ha nincs egy elem sem” kérdése. Ezért a feltételes maximum vagy minimum keresése a keresés algoritmusának és az egyszerű maximum- vagy minimumkiválasztás algoritmusának kombinációja.
- A sorozatszámítás, a megszámlálás, az egyszerű maximum- vagy minimumkiválasztás, az eldöntés, a keresés és a kiválasztás – esetleg a fentebb említett kiegészítésekkel – az alábbi gondolkodási sémával (algoritmusvázzal) megoldható:

```
kezdőérték adás a majdani eredmény változójának
ciklus i = 0-tól az adatsorozat összes elemére:
    ha feltétel teljesül :
      kezdőérték módosítása, eredmény aktualizálása
    elágazás vége
ciklus vége
kimenet az eredmény változója vagy ettől függő válasz
```

Az elemi vezérlési struktúrákból – egymás utáni utasítások, elágazások és ciklusok – alkotott egyszerű típusalgoritmusok jellemzője, hogy három utasítást tartalmaznak, amiből a középső egy cikluson belüli elágazás. Az összetett algoritmusok javarészt ezekből az egyszerű algoritmusokból épülnek fel. Például úgy, ahogy a feltételes maximumkiválasztásnál láthattuk: a keresés végén a feltételes válasz egyik ága a maximumkiválasztás. De az is jellemző – már több feladatban alkalmaztuk –, hogy a feltétel nem egy egyszerű reláció, hanem egy eldöntés függvény eredménye.

Feladatok típusalgoritmusokra

Fejtörők

Melyik típusalgoritmus való a következő feladatok megoldására? Elég az egyszerűbb forma, vagy kell-e a részletes? Elég érték szerint bejárnunk a listát, vagy indexszel kell hivatkozni az adatokra?

1. Listában tároljuk, hogy egy londoni pincér mekkora összegeket helyezett el a pénztárcájában az elmúlt órában. Amikor kivett a tárcából pénzt, azt negatív számmal jeleztük.
   Az alábbi feladatokhoz példaként ezeket az adatokat használhatjuk:

   ```
   {3, 8, 10, 19.35, -6, 5.1, 9, 20}
   ```

   a) Volt-e olyan, hogy a pincér vásárolt valamit, vagy mindig csak neki fizettek?
   b) Ha az óra elején üres a pénztárcája, mennyi van benne az óra végén?
   c) Hány alkalommal kapott biztosan pennyt is, nem csak fontot?
   d) Hány pennyt kapott összesen (feltételezve, hogy csak azt fizették pennyben, amit nem lehet fontban)?
   e) Hány esetben kapott legalább öt fontot?
   f) Milyen összegről szólt a legnagyobb értékű számla, amit fizettek nála?
   g) Ha az óra elején már volt 8 font 23 penny a tárcájában, mennyi pénz volt benne az óra végén?
   h) Hányadik vendég fizetett 9 fontot?
   i) Ha volt olyan vendég, aki tíz fontnál többet fizetett, akkor mondjuk meg, hogy hányadik vendég volt az első ilyen!
   j) Ha volt olyan vendég, aki tíz fontnál többet fizetett, akkor mondjuk meg, hogy hányadik vendég volt az utolsó ilyen!
   k) Volt-e olyan vendég, akinek módjában állt csupa ötfontossal kiegyenlíteni a számlát?
   l) Ha a főnöke minden vendég után fél fontot ad pincérünknek fizetésül, mekkora bevétellel zárta az órát?
2. A következő feladatok egy mondat szavaiból képzett listára vonatkoznak.

   ```
   {"Én", "elmentem", "a", "vásárba", "fél", "arannyal."}
   ```
   [2]

   Tudjuk, hogy a szöveg típus egy karaktersorozat, hossza például megállapítható a `size()` függvénnyel. A `bool isupper(char)` függvény megmondja, hogy egy karakter (betű) nagybetű-e, hasonlóan az `bool islower(char)` azt adja meg, hogy a karakter kisbetű-e. A nem betű karakterekre egyik sem igaz.

   a) Hány szóból áll a mondat?
   b) Hány betűs a legrövidebb szó?
   c) Van-e a mondatban olyan szó, ami mondatközi vagy mondatvégi írásjellel végződik?
   d) Hány névelő van a mondatban, illetve a szavait tartalmazó listában?
   e) Hányadik szó a „fél”?
   f) Van-e a mondatban nagy kezdőbetűs szó, és ha igen, akkor hányadik?

Feladatok

Miután megadtuk, hogy milyen típusú algoritmussal válaszolhatók meg a fenti két feladat kérdései, készítsük el a kódot is.

A típusalgoritmusok ismerete és használata a kód megírását jelentősen megkönnyíti, de általában számos részletet még tisztázni kell. Jellemző, hogy a feltett kérdés vagy megoldandó

[2] Asszociáció: „Én elmentem a vásárba félpénzzel”. Eredet: Kitrákotty mese (népdal); Vikár Béla gyűjtése; Felsőboldogfalva 1903. Feldolgozás: Kodály Zoltán: Székelyfonó 6. dal (1932)

probléma megfogalmazása nem elég pontos ahhoz, hogy kódoljuk. A feladatot részletesebben kell megadni, **specifikálni** kell. A specifikáció során pontosítjuk a megoldással kapcsolatos elvárásainkat. Általában döntéseket is kell hoznunk. Gyakori kérdés: „mit értünk az alatt, hogy …” vagy „mit csináljon a program, ha …”. Egy program minőségét és használhatóságát az ilyen kérdésekre adott válaszok jelentősen befolyásolják. A kódolás előtt gondoljuk át a lehetséges válaszokat!

1. Az első feladatcsoport megoldásához listában tároljuk, hogy egy londoni pincér mekkora összegeket helyezett el a pénztárcájában az elmúlt órában. Amikor kivett a tárcából pénzt, azt negatív számmal jeleztük.
   A megoldás kódolásához szükségünk lehet a szám szöveggé alakítására. Ehhez használjuk a `string to_string(int)` függvényt, ami bármilyen számtípusra használható. Az `int` és `double` értékek közötti átalakítás *explicit casttal* lehetséges, például egész szám az `(int)3.4` vagy az `int(3.4)`.

   A példákban használt adatsorozat legyen globálisan elérhető:

   ```
   {3, 8, 10, 19.35, -6, 5.1, 9, 20}
   ```

a) Volt-e olyan, hogy a pincér vásárolt valamit, vagy mindig csak neki fizettek?

   A megoldáshoz az eldöntés típusalgoritmusát használjuk. A feladat megoldható bejárós ciklussal és kiugrással vagy feltételes ciklussal.

b) Ha az óra elején üres a pénztárcája, mennyi van benne az óra végén?

   A megoldáshoz az sorozatszámítás – összegzés – egyszerű formáját használjuk.

c) Hány alkalommal kapott biztosan pennyt is, nemcsak fontot?

   A megoldáshoz a feltételes megszámolás algoritmust használjuk. Akkor kap pennyt is a pincér, ha pozitív törtszám a lista vizsgált eleme. Egy szám törtszám, ha nem egyenlő az egészre kerekített értékével.

d) Hány pennyt kapott összesen (feltételezve, hogy csak azt fizették pennyben, amit nem lehet fontban)?

   A megoldáshoz a feltételes összegzés típusalgoritmusát használjuk. Egy angol font száz pennyt ér, az eredményt egész számként adjuk meg.

e) Hány esetben kapott legalább öt fontot?

   A megoldáshoz a feltételes megszámolás algoritmusát használjuk.

f) Milyen összegről szólt a legnagyobb értékű számla, amit fizettek nála?

   A megoldáshoz a maximumkiválasztás algoritmusát használjuk. Minden adatot figyelembe vehetünk, mert a kifizetés negatív, ami csak akkor lehetséges, ha volt elegendő összeg a tárcában. (Másik specifikáció: ha hitele is lehetne, akkor figyelni kellene arra, hogy van-e egyáltalán befizetés.)

g) Ha az óra elején már volt 8 font 23 penny a tárcájában, mennyi pénz volt benne az óra végén?

   A megoldáshoz egyszerű összegzés algoritmust használunk. A b) feladathoz képest annyi az eltérés, hogy 8,23-at hozzá kell adnunk annak eredményéhez.

Módosítsuk a b) feladat specifikációját: a megoldás paramétere legyen a kezdőösszeg és a bevételek listája! A főprogramban kérdezzük meg a felhasználótól, hogy mennyi volt a kezdő összeg, majd ezt felhasználva írjuk ki a végösszeget!

h) Hányadik vendég fizetett 9 fontot?

A megoldáshoz a kiválasztás algoritmusát használjuk (tudjuk, hogy az egyik vendég ennyit fizetett). Figyeljünk arra, hogy 0-tól indexelt a listánk.

i) Ha volt olyan vendég, aki tíz fontnál többet fizetett, akkor mondjuk meg, hogy hányadik vendég volt az első ilyen!

A keresés algoritmusát használjuk, mert nem biztos, hogy volt ilyen vendég.

j) Ha volt olyan vendég, aki tíz fontnál többet fizetett, akkor mondjuk meg, hogy hányadik vendég volt az utolsó ilyen!

A megoldás nagyon hasonló az előző feladatéhoz. Az utolsó vendég fellelésére két lehetséges módszer: vagy hátulról előrefelé keressük az első találatot, vagy végignézzük a sorozatot és minden találatnál módosítjuk, a tervezett válaszunkat az éppen utolsó indexére.

k) Volt-e olyan vendég, akinek módjában állt csupa ötfontossal kiegyenlíteni a számlát?

A megoldáshoz az eldöntés algoritmusát használjuk, csak a pozitív egész értékeket szabad figyelembe venni, és vizsgálni, hogy öttel oszthatók-e. Önkényes specifikációnk: csak akkor írunk ki valamit, ha találunk megfelelő értéket, így a „nem talált” esetben nincs kiírás.

l) Ha a főnöke minden vendég után fél fontot ad pincérünknek fizetésül, mekkora bevétellel zárta az órát?

A megoldáshoz a megszámolás algoritmusát használjuk, csak a pozitív értékeket vesszük figyelembe. A végén szorozzuk fel a jutalékkal.

2. A feladatban egy mondat szavaival kapcsolatosak a kérdések. A mondat számára hozzunk létre globális változóként egy kellően nagy méretű, például 100 elemű tömböt és adjuk meg az első értékeit:

```
{"Én", "elmentem", "a", "vásárba", "fél", "arannyal."}
```

A tömb fel nem használt elemei az üres szöveget `""` fogják tartalmazni. Az egyes részfeladatokat – az a) feladat kivételével – eljárás formájában írjuk meg és hívjuk meg a főprogramban.

a) Hány szóból áll a mondat?

A megoldáshoz egyszerű számlálás algoritmust használunk, a mondat végét a tömb méretének elérése vagy az első üres szöveg jelzi. A méretet tároljuk el a főprogramban egy változóban és paraméterként adjuk át a többi eljárásnak!

b) Hány betűs a legrövidebb szó?

Egyszerű minimumkeresés, az egyes szavak hosszát a szó `size()` függvénye adja meg. Specifikáció: eltekintünk a szót követő speciális írásjelek hatásától, a karakterek számával számolunk.

c) Van-e a mondatban olyan szó, ami mondatközi vagy mondatvégi írásjellel végződik?

Az eldöntés algoritmusát kell használni, amin belül a vizsgált tulajdonság az, hogy a szó utolsó betűje kisbetű-e vagy nagybetű-e. Másik lehetőség, hogy ezt is eldöntés algoritmussal vizsgáljuk: az írásjelek tömbjében benne van-e a kérdéses karakter. Specifikáció kérdése, hogy milyen írásjeleket veszünk fel a tömbbe.

d) Hány névelő van a mondatban, illetve a szavait tartalmazó listában?

A megoldás feltételes megszámlálás. A feltétel hasonló az előző feladatéhoz, de az egész szót kell hasonlítani. Mivel összesen három névelő van (a, az, egy), itt talán érdemes azt vizsgálni, hogy a szó megegyezik-e valamelyikkel. Specifikációs probléma, hogy figyelünk-e a kis- és nagybetűkre is (Az és az), de komoly kihívást az „egy” beszámíthatósága jelenti. Mi van, ha nem határozatlan névelő, hanem kiírt számérték? Legyen a pontosított kérdés: Hány határozott névelő van a mondatban?

e) Hányadik szó a „fél”?

A megoldáshoz a keresés algoritmusát használjuk, mert lehet, hogy nincs benne. Írjuk meg a megoldást úgy, hogy a keresett szót paraméterként adjuk meg, így könnyen általánosítható a feladat! A kérdést ezzel átfogalmazzuk: A mondat hányadik szava a megadott szó?

f) Van-e a mondatban nagy kezdőbetűs szó, és ha igen, akkor hol?

Ismét a keresés algoritmusát használjuk. A feltétel vizsgálatához használhatjuk a C++ `bool isupper(char)` függvényét.

Megoldás

Írjunk az egyes feladatok megoldására függvényt vagy eljárást! A kérdésekre adott válaszokat (a program kimenetét) a feladat betűjelének feltüntetésével, a főprogramban rendezzük egymás után. Akkor is csak egy megoldást írjunk, ha többféle specifikációra lenne mód.

1. Az alábbi megoldásokhoz a `vector<double>` típusú adatsorozatot használtunk. Ennek megfelelően egyes megoldások bejárós ciklussal is megoldhatók, ha szükséges, akkor az indexeléshez előjelnélküli – `unsinged` – egész típust használunk. Minden megoldás egy függvény, aminek a neve a feladat betűjelével azonos. Az eredmény kiírása a főprogramban van.

```
#include <iostream>
#include <vector>
using namespace std;

vector<double> tarca{3, 8, 10, 19.35, -6, 5.1, 9, 20};

bool A()
{
  for(double font : tarca)
    if(font < 0)
      return true;
  return false;
}

```

```
15.  double B()
16.  {
17.    double ossz = 0;
18.    for(double font : tarca)
19.      ossz += font;
20.    return ossz;
21.  }
22.
23.  int C()
24.  {
25.    int db = 0;
26.    for(double font : tarca)
27.      if(font > 0 && font != (int)font)
28.        db++;
29.    return db;
30.  }
31.
32.  int D()
33.  {
34.    double penny = 0;
35.    for(double font : tarca)
36.      if(font > 0 && font != (int)font)
37.        penny += font - (int)font;
38.    return (int)(penny * 100);
39.  }
40.
41.  int E()
42.  {
43.    int db = 0;
44.    for(double font : tarca)
45.      if(font >= 5)
46.        db++;
47.    return db;
48.  }
49.
50.  double F()
51.  {
52.    double maxe = tarca[0];
53.    for (double font : tarca)
54.      if(maxe < font)
55.      maxe = font;
56.    return maxe;
57.  }
58.
59.  double G(double kezd, vector<double> bevetel)
60.  {
61.    double veg = kezd;
62.    for (double font : bevetel)
63.      veg += font;
64.    return veg;
65.  }
66.
```

```
67.  int H()
68.  {
69.    int ez = 0;
70.    while (tarca[ez] != 9)
71.      ez++;
72.    return ez; //0-tól számozva
73.  }
74.
75.  string I()
76.  {
77.    unsigned ez;
78.    for(ez = 0; ez < tarca.size() && tarca[ez] <= 10; ez++);
79.    string valasz;
80.    if(ez < tarca.size())
81.      valasz = "Az első tíz fontnál többet fizető vendég a(z) " +
                    to_string(ez + 1) + ". vendég.";
82.    else
83.      valasz = "Minden vendég 10 fontnál kevesebbet fizetett.";
84.    return valasz;
85.  }
86.
87.  string J()
88.  {
89.    unsigned ez;
90.    for(ez = tarca.size() - 1; ez >= 0 && tarca[ez] <= 10; ez--);
91.    string valasz;
92.    if(ez >= 0)
93.      valasz = "Az utolsó tíz fontnál többet fizető vendég a(z) " +
                    to_string(ez + 1) + ". vendég.";
94.    else
95.      valasz = "Minden vendég 10 fontnál kevesebbet fizetett.";
96.    return valasz;
97.  }
98.
99.  bool K()
100. {
101.   unsigned i = 0;
102.   while(i < tarca.size() && ! (tarca[i] == (int)tarca[i]
                          && tarca[i] > 0 && (int)tarca[i] % 5 == 0))
103.     i++;
104.   if (i < tarca.size())
105.     return true;
106.   else
107.     return false;
108. }
109.
110. double L()
111. {
112.   int db = 0;
113.   for(double font : tarca)
114.     if(font > 0)
115.     db++;
116.   return 0.5 * db;
117. }
118.
```

```
119.  int main()
120.  {
121.    setlocale(LC_ALL, "");
122.    cout<<"Válaszok" << endl;
123.    cout<<"a)\t"<< (A()?"Vásárolt is.":"Csak fizettek neki.") <<endl;
124.    cout<<"b)\t"<< "A pénztárcában " << B() << " font van." <<endl;
125.    cout<<"c)\t"<< C() << " alkalommal kapott pennyt." <<endl;
126.    cout<<"d)\t"<< "A pincér " << D() << " pennyt kapott." <<endl;
127.    cout<<"e)\t"<< E() << " alkalommal kapott min. 5 fontot." <<endl;
128.    cout<<"f)\t"<< "A legnagyobb számla " << F() <<" font." <<endl;
129.    cout<<"Mennyi pénz volt a tárcában? ";
130.    double ind; cin >> ind;
131.    cout<<"g)\t"<< "Most " << G(ind, tarca) << " van benne." <<endl;
132.    cout<<"h)\t"<< "A "<< H()+1 <<". vendég fizetett 9 fontot."<<endl;
133.    cout<<"i)\t"<< I() <<endl;
134.    cout<<"j)\t"<< J() <<endl;
135.    cout<<"k)\t"<< (K()?"Volt, aki ötösökkel tudott fizetni.":"Nem volt
      olyan vendég, aki csak ötfontosokkal tudott fizetni.")<<endl;
136.    cout << "l)\t" <<"A pincér "<< L() <<" font fizetést kap."<<endl;
137.    return 0;
```

2. Az alábbi megoldásokhoz egy 100 elemű tömböt használunk, amelyet elölről folyamatosan töltünk fel a mondat szavaival, a maradék helyeken "" áll. Az egyes megoldások – ahol értelmes, ott – a feladat betűjelének megfelelő eljárások, az a) feladat megoldása és néhány segéd algoritmus megvalósítása függvénnyel történik.

```
1.  #include <iostream>
2.  using namespace std;
3.  const int maxdb = 100;
4.  string mondat[maxdb]
                {"Én", "elmentem", "a", "vásárba", "fél", "arannyal."};
5.
```

```
6.  int szodb()
7.  {
8.    int db = 0;
9.    while (db < maxdb && mondat[db] != "")
10.     db++;
11.   cout << "a)\tA mondat " << db << " szóból áll." << endl;
12.   return db;
13. }
14.
```

```
15. void b(int N)
16. {
17.   unsigned minhossz = mondat[0].size();
    //Biztosan létezik. Ha nincs mondat, akkor az értéke 0.
18.   for (int i = 1; i < N; i++)
19.     if (mondat[i].size() < minhossz)
20.       minhossz = mondat[i].size();
21.   cout << "b)\tA legrövidebb szó " << minhossz << " karakter." << endl;
22. }
23.
```

```
24.  bool irasjel(char c)
25.  {
26.    char jelek[5] {'.', '?', '!', ',', ';'};
27.    int i;
28.    for (i = 0; i < 5 && c != jelek[i]; i++);
29.    return i < 5;
30.  }
31.
32.  void c(int N)
33.  {
34.    int i = 0;
35.    while (i < N && ! irasjel(mondat[i][mondat[i].size() - 1]))
36.      i++;
37.    if (i < N)
38.      cout << "c)\tVan olyan szó, ami után írásjel áll." << endl;
39.    else
40.      cout << "c)\tNincs olyan szó, ami után írásjel áll." << endl;
41.  }
42.
43.  void d(int N)
44.  {
45.    int db = 0;
46.    for (int i = 0; i < N; i++)
47.      if (mondat[i] == "a" || mondat[i] == "az" || mondat[i] == "A" ||
           mondat[i] == "Az")
48.        db++;
49.    cout << "d)\tA mondatban "<< db << " határozott névelő van." << endl;
50.  }
51.
52.  void e(int N, string szo)
53.  {
54.    int i;
55.    for (i = 0; i < N && mondat[i] != szo; i++)
56.    ; //nincs ciklusmag
57.    if (i < N)
58.      cout << "e)\tA mondatban a(z) \"" << szo << "\" szó a(z) "
             << i + 1 << ". helyen áll." << endl;
59.    else cout << "e)" << endl;
60.  }
61.
62.  void f(int N)
63.  {
64.    int i = 0;
65.    while (i < N && ! isupper(mondat[i][0]))
66.      i++;
67.    if (i < N)
68.      cout<<"f)\tA(z) "<< i + 1 << ". szó kezdődik nagybetűvel." <<endl;
69.    else
70.      cout << "f)\tNincs a mondatban nagybetűvel kezdődő szó." << endl;
71.  }
72.
```

```
73.  int main()
74.  {
75.    setlocale(LC_ALL,"");
76.    cout << "Megoldások:" << endl;
77.    int n = szodb();
78.    b(n);
79.    c(n);
80.    d(n);
81.    e(n, "fél");
82.    f(n);
83.    return 0;
84.  }
```

KÉTDIMENZIÓS ADATSZERKEZET

Mik azok a kétdimenziós adatszerkezetek?

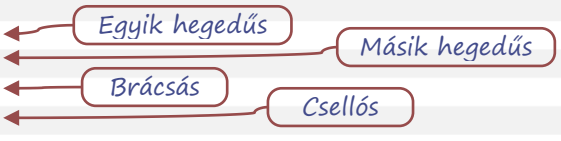
Az egyszerű adatsorozatok egydimenziós adatszerkezetek – azaz csak hosszuk van, mint egy szakasznak a geometriában. Azonban az adatsorozatokban elhelyezhetünk olyan elemeket is, amelyek saját maguk is adatsorozatok. Így lesz az adatszerkezet kétdimenziós: van „szélessége” és „magassága”. Láttunk, használtunk már ehhez hasonlót? Igen. Valószínűleg nem is egyszer. Ilyenek a szorzótábla, a sakktábla, lényegében a pixelgrafikus képek és még sorolhatnánk … a táblákat, a táblázatokat. A kétdimenziós adatszerkezetek használata nagyon gyakori – pont úgy, ahogy a táblázatoké a valóságban.

Nézzük meg, hogyan lehet C++ nyelven kétdimenziós adatszerkezetet létrehozni! Egydimenziós adatsorozatból ismerjük az (egydimenziós) tömböt, a `vector<>`-t és karaktersorozatként a `string`-et. A `string` elemei csak karakterek lehetnek, de a tömb és `vector<>` eleme bármi lehet. Így elméletileg lehet egy tömb minden eleme tömb vagy `vector<>` vagy `string`; lehet egy `vector<>` minden eleme `vector<>` vagy tömb vagy `string`. Ez összesen hatféle kétdimenziós struktúra. De hiszen ebből kettőt már használtunk is! Tároltunk `string`eket tömbben és `vector<>`-ban, például a mondatelemzős feladatban ki ezt, ki azt, ki mindkettőt használhatta. Ott az `i`-edik szó első karaktere: `mondat[i][0] volt`. Ne számítsunk csodára, a másik négy variáció is hasonló.

A következő kétdimenziós adatszerkezet egy vonósnégyes tagjainak jelenléti íve. Az ív egy hét munkanapjain mutatja a jelenlétet: ahol 1 szerepel benne, ott jelen volt a zenész, ahol 0, ott nem. Egy-egy „kis lista” egy zenész jelenlétét mutatja be, a „nagy” lista pedig az egész zenekarét.

A `ttiv`: tömbök tömbje, mindkét irányban rögzített a mérete

```
1.  int ttiv[4][5] {
2.    {1, 1, 1, 1, 1},
3.    {1, 1, 1, 1, 0},
4.    {1, 1, 0, 0, 0},
5.    {0, 1, 1, 1, 1}
6.  };
```



A vviv: vector<>-ok vector<>-a
Mindkét irányban dinamikusan bővíthető

```
vector<vector<int> > vviv {
  {1, 1, 1, 1, 1},
  {1, 1, 1, 1, 0},
  {1, 1, 0, 0, 0},
  {0, 1, 1, 1, 1}
};
```

A tviv: tömbbe teszünk vector<>-okat
A sorok hossza különböző lehet.

```
vector<int> tviv[4] {
  {1, 1, 1, 1, 1},
  {1, 1, 1, 1, 0},
  {1, 1, 0, 0, 0},
  {0, 1, 1, 1, 1}
};
```

A vector<>-ba egyelőre nem tudunk tömböt tenni, mert az eddig megismert nyelvi eszközkészletünk ehhez kevés. Például nem lehet, hogy egy adattípusban változónév legyen.

```
vector<int/*tombneve?*/[5]>;
```

A probléma lényege, hogy amikor egy tömböt definiálunk, akkor a tömb neve és a szögletes zárójel nem a teljes adatot (minden elemével) jelöli, hanem csak az adat kezdőcímét. Bár a megadott méretet a programunk lefoglalja, nincs „fizikai” védelem a túlindexelés ellen. Egy tömb nincs egységbe fogva, nem objektum – ezért nincs zölden megjeleníthető típusneve sem. vector<>-ba tömböt kétféle módon tehetünk. Vagy becsomagoljuk egy objektumba – ilyen a string karaktersorozata –; vagy a tömb kezdetének címét tesszük be a listánkba. A profi programozók egyik ismérve, hogy címeken keresztül is tudják használni az adatokat.

Melyiket kell tudni? Bármelyiket, egyiket. Aki eddig a tömböt használta, annak a tömbök tömbje a legegyszerűbb. A lista-fanok választhatják a vector<>-ok vector<>-át, a programozás virtuózainak a vector<>-ok tömbje sem jelent gondot. Aki programozónak készül és egyetemi szintű ismeretektől sem riad vissza, az kísérletezhet a tömbök listájával, számukra határ a „csillagos ég”. A változatok között nagy az eltérés az adatok feltöltésekor és a függvények paramétereinek megadásakor.

Kétdimenziós adatstruktúrák feltöltése adattal

Láthatjuk: ha létrehozáskor adjuk meg a táblázat adatait, akkor mindegy, melyik struktúrát használjuk, az adatsorokat 1-1 elemnek tekintve, egymástól vesszővel elválasztva adjuk meg a táblázatot.

Ha az adatokat a létrehozás pillanatában még nem ismerjük, hanem csak később – esetleg a felhasználótól bekérve – szeretnénk megadni, akkor figyelembe kell venni a tömb és vector<> adatbevitelének szabályait.

- Kétdimenziós tömb típusú táblázat esetén is a szükséges legnagyobb méretet hozzuk létre, ilyenkor minden adatnak le lesz foglalva a memóriában megfelelő méretű hely, az adatot csak „be kell tenni” a megfelelő helyre.

```
ttiv[3][4] = 1;
```

- Kétdimenziós vector<> típusú táblázat esetén a push_back() eljárást kell használni. Új sor létrehozásához először külön változóban hozzuk létre a sort (legalább egy elemet célszerű beletenni), majd ezt a sort adjuk hozzá a táblázatunkhoz. Meglévő sorra az indexével hivatkozhatunk, aminek a végére szintén push_back() eljárással kerül új elem.

```
1.  vector<int> sor;
2.  sor.push_back(0);
3.  vviv.push_back(sor);
4.  vviv[3].push_back(0);
```

- Ha tömbbe teszünk `vector<>`-t, akkor a megadott számú sorhoz le lesz foglalva a memória és 0 hosszúságú `vector<>`-ok is létrejönnek. új sort nem lehet létrehozni, de bármelyik sorba `push_back()` eljárással tehetünk új elemeket, úgy, ahogy a fenti 4. sorban látható.

Kétdimenziós adatstruktúrák bejárása

Amikor egy kétdimenziós adatszerkezet minden elemét végignézzük – például azért, hogy kiírjuk –, mindkét dimenzióhoz külön ciklust használunk. A vonósnégyesünk esetén kell egy ciklus a zenészekre és ezen belül(!), hogy minden zenészre külön-külön érvényesüljön, kell egy ciklus a napokhoz.

Ha a kétdimenziós struktúra sorai `vector<>` típusúak, akkor ennek kell a belső ciklusnak lennie. Ilyen adatstruktúrákban az is lehetséges, hogy a sorok különböző hosszúak, azaz „szaggatott szélű” – jagged – a táblázatunk. A kétdimenziós tömbök sorai azonos hosszúságúak, a méret rögzített. Az ilyen táblázatot bejárhatjuk akár úgy is, hogy oszloponként, azon belül fentről lefelé haladunk, de az sem okoz elvi nehézséget, ha lóugrásban lépkedünk.

A kétdimenziós adatstruktúra végignézéséhez a while-ciklus mindig használható, de sokszor nem kényelmes, mert a ciklusmagokon belül több utasítás is lesz, ami miatt több kapcsos zárójelre lesz szükség. A számlálós for-ciklus egyszerűbb kódot eredményez. `vector<>` esetén használhatjuk a bejárós for-ciklust, a ciklusfejben egyértelmű, hogy a vector adattípusa egy táblázat esetén egy adatsor, illetve egy adatsoron belül egy adatelem.

Típusalgoritmusok a kétdimenziós adatszerkezetben

Az alábbi példákban a vonósnégyes jelenléti íveivel dolgozunk. Az adatokat globális változóként tároljuk, az egyes feladatok megoldása egy-egy függvény lesz. Az eredmény felhasználása a főprogramban van, ennek megírása egyéni feladat lesz.

A négyféle táblázatban az adatokkal kapcsolatos kérdések megválaszolásához jellemzően 2-2 ciklus szükséges, mindegyik ciklusra van 2 vagy 3 választási lehetőségünk, így egy részfeladat kódolására több, mint 20 megoldás van. A feladat megoldásához válasszunk ki egy táblázat típust és a megoldások során próbáljuk a már begyakorolt kódolást alkalmazni!

Mintafeladat

1. A zenészek a közeli kifőzdében szoktak ebédelni. Ismerik őket, úgyhogy hét közben csak felírják a fogyasztott adagok számát, és péntekenként fizetik az egész heti számlát. Hány adagot fizetnek ezen a pénteken?
   A megoldásban megszámoljuk, hogy hány egyes van a „táblázatban”.

2. Melyik zenész volt a legtöbbet jelen a héten?
   A megoldáshoz soronként eltároljuk a részeredményeket. Egy segédtáblázatba írjuk ki soronként, hogy az egyes zenészek a héten hányszor voltak jelen. Ezután már csak ebben a táblázatban kell megkeresni a legnagyobb, illetve legkisebb érték indexét. Jobb híján, ez a szám lesz az eredmény. A megoldás továbbfejlesztéseként egy újabb tömbben tároljuk a zenészek nevét, ekkor az index segítségével néven tudjuk nevezni a legjobbat.

A kód apró módosításával megadható az is, hogy ki hiányzott legtöbbet, illetve az is, hogy ki volt jelen legkevesebb alkalommal. A két kérdésre milyen esetben lesz eltérő a válaszunk?

3. Volt-e olyan zenész, aki mindig jelen volt?
   A feladat az előzőnél egyszerűbbnek látszik: nem kell maximumot keresni. De egy táblázatban az adatsornál is jobban számít, hogy végignézzük-e akkor is, ha már biztosan tudjuk a választ.

4. Írjuk ki, ha volt olyan nap, amikor mindenki jelen volt a próbán!
   A feladat nehézsége, hogy ezúttal nem egy vonóst, hanem egy napot vizsgálunk. Úgy is mondhatjuk, hogy nem egy sor értékeit összegezzük, hanem egy oszlopét.

Megoldás kétdimenziós tömbbel

A megoldás tartalmazza az adatok beolvasását és kiírását is. A gyakorlati alkalmazásokhoz igazodva, a megoldás általánosabb, nem csak vonósnégyes, hanem más kamarazenekarra is alkalmassá tesszük és a hétvégi próbák tárolására is felkészítjük programunkat. Így a jelenléti íven 20 fő 7 napjának a tárolására van lehetőség. A zenekar tagjainak számát és a tárolt napok számát a függvényeknek és eljárásoknak paraméterként adjuk át.

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  int ttiv[20][7]; //inkább legyen több hely
5.
6.  void tablazatBe(int tag, int nap)
7.  {
8.    for (int ez = 0; ez < tag; ez++)
9.      for (int ekkor = 0; ekkor < nap; ekkor++)
10.       cin >> ttiv[ez][ekkor];
11. }
12.
13. void tablazatKi(int tag, int nap)
14. {
15.   for (int ez = 0; ez < tag; ez++) {
16.     for (int ekkor = 0; ekkor < nap; ekkor++)
17.       cout << ttiv[ez][ekkor];
18.     cout << endl; //sorvég
19.   }
20. }
21.
```

1. A zenészek a közeli kifőzdében szoktak ebédelni. … Hány adagot fizetnek ezen a pénteken?

```
22. int ebedDb(int tag, int nap)
23. {
24.   int db = 0;
25.   for (int ez = 0; ez < tag; ez++)
26.     for (int ekkor = 0; ekkor < nap; ekkor++)
27.       if (ttiv[ez][ekkor] == 1)
28.         db++;
29.   return db;
30. }
31.
```

2. Melyik zenész volt a legtöbbet jelen a héten?
Nevezzük nevén! Ehhez hozzuk létre a zenészek névsorát:

```
32.  string nev[20] {"Heg Edu", "Vio Lina", "Brá Csaba", "Csel Lotti"};
33.
```

Először hozzunk létre egy segédtömböt, amiben tároljuk a jelenlétek számát. Utána ebből határozzuk meg, hogy melyik a legtöbb. (Megoldható segédtömb nélkül is.)

```
34.  string legtobbJelen(int tag, int nap)
35.  {
36.    int jelen[20];
37.    for (int i = 0; i < tag; i++) {
38.      jelen[i] = 0;  //soronként megszámlálás
39.      for (int ekkor = 0; ekkor < nap; ekkor++)
40.        if (ttiv[i][ekkor] == 1)
41.          jelen[i]++;
42.    }
43.    int maxi = 0;   //maximumkiválasztás tétele
44.    for (int i = 1; i < tag; i++)
45.      if (jelen[i] > jelen[maxi])
46.        maxi = i;
47.    return nev[maxi];
48.  }
```

3. Volt-e olyan zenész, aki mindig jelen volt?
A kérdésre az eldöntés algoritmus segítségével adunk választ. Ilyenkor nem szükséges végignézni minden adatot. Ha egy zenész egyszer hiányzott, akkor nem kell tovább néznünk az ő adatait és ha egy zenész végig jelen volt, akkor nem kell néznünk az utána következő zenészeket. Ezért most while-ciklusokat használunk.

```
50.  bool mindig(int tag, int nap)
51.  {
52.    bool mindig = false;  //eldöntés a személyre
53.    int ez = 0;
54.    while (ez < tag && !mindig) {
55.      int jelen = 0;       //soronként, azaz a jelenlétre eldöntés
56.      while(jelen < nap && ttiv[ez][jelen] == 1)
57.        jelen++;
58.      if (jelen == nap)
59.        mindig = true;
60.      ez++;
61.    }
62.    return mindig;
63.  }
64.
```

4. Írjuk ki, ha volt olyan nap, amikor mindenki jelen volt a próbán!
Erre a kérdésre is eldöntés algoritmusa segítségével adunk választ. Most a for-ciklus feltételes változatát alkalmazzuk.

```
65.  bool mindenki(int tag, int nap)
66.  {
67.    bool mindenki = false;
68.    for (int ekkor = 0; ekkor < nap && !mindenki; ekkor++) {
69.      int jelen;
70.      for (jelen = 0; jelen < tag && ttiv[jelen][ekkor] == 1; jelen++)
71.        ;
72.      if (jelen == tag)
73.        mindenki = true;
74.    }
75.    return mindenki;
76.  }
```

Mindegyik feladatban mindegyik ciklus lehet for-ciklus és while-ciklus is, de nem lehet bejárós ciklus. A kétdimenziós tömb mérete rögzített, ennél kisebb táblázattal dolgozhatunk. Minden feladathoz megadtuk a sorok és oszlopok tényleges számát. Ezt akkor is meg kell tennünk, ha a tömböt is paraméterként adjuk át. Bár itt nem emeltük ki, de mindegyik megoldás két részre bontható, amelyben az egyik dimenzióra (általában sorra) egy típusalgoritmussal adunk megoldást majd ennek eredményéből egy másik – nem feltétlen eltérő – típusalgoritmussal oldjuk meg a feladatot.

Megoldás kétdimenziós listával

A megoldás tartalmazza az adatok beolvasását és kiírását is. A gyakorlati alkalmazásokhoz igazodva, a megoldás általánosabb, nem ismerjük sem a zenekar tagjainak a számát, sem a napok számát. Ezért mindenhol az adott lista (`vector<>`) méretét használjuk.

Az adatok beírásához számlálós ciklust használunk, mert feltételezzük, hogy előre ismerjük az adatok számát és minden zenészhez ugyanannyi adat tartozik. Ez egy általánosabb megoldásban módosulhat, például minden zenész adatait egy sorba írva az Enter leütéséig olvassuk az adatokat és üres sor jelzi, hogy befejeztük a beírást.

Az adatok kiírásakor már ismert az adatok száma, használhatunk bejáró-ciklust. Természetesen bármely másik ciklus-típus is használható. Figyeljük meg, hogy a külső ciklusban a teljes sorokon iterálunk, a belső ciklus listája a sor, az elemei egészek.

```
1.  #include <iostream>
2.  #include <vector>
3.  using namespace std;
4.
5.  vector<vector<int>> vviv;
6.
```

```
 7.  void listaBe(int tag, int nap)
 8.  {
 9.    for (int ez = 0; ez < tag; ez++) {
10.      vector<int> napok;
11.      for (int ekkor = 0; ekkor < nap; ekkor++) {
12.        int adat;
13.        cin >> adat;
14.        napok.push_back(adat);
15.      }
16.      vviv.push_back(napok);
17.    }
18.  }
20.  void listaKi()
21.  {
22.    for (vector<int> tag : vviv) {
23.      for (int jel : tag)
24.        cout << jel;
25.      cout << endl;
26.    }
27.  }
28.
```

1. A zenészek a közeli kifőzdében szoktak ebédelni. … Hány adagot fizetnek ezen a pénteken?
   A megoldásban – a minta kedvéért – a külső ciklus számláló, a belső bejáró.

```
29.  int ebedDb()
30.  {
31.    int db = 0;
32.    for (unsigned ez = 0; ez < vviv.size(); ez++)
33.      for (int jel : vviv[ez])
34.        if (jel == 1)
35.          db++;
36.    return db;
37.  }
38.
```

2. Melyik zenész volt a legtöbbet jelen a héten?
   Nevezzük nevén! Hozzuk létre a zenészek névsorát:

```
39.  vector<string> nev{"Heg Edu", "Vio Lina", "Brá Csaba", "Csel Lotti"};
40.   //Nem jó, ha kevesebb a név a tárolt tagok számánál
```

A ciklusokat az előző feladathoz hasonlóan szervezzük, de most a belső ciklus megszámláláshoz kell, a külső a maximum meghatározásához. Mivel a zenész sorszáma lesz az eredmény, azaz a sor indexét kell meghatároznunk, ezért a külső ciklus nem lehet bejáró ciklus.

A részeredmények tárolásához létrehozunk egy segédlistát, ami pont olyan hosszú, mint a jelenléti ív. Ebben tároljuk a megszámlálás eredményét. Másik megoldási lehetőség: a lista

helyett elegendő egyetlen változó, amelyben az aktuális maximális részvétel mennyiségét tároljuk.

```
41.   string legtobbJelen()
42.  {
43.    vector<int> jelen(vviv.size()); //annyi elem, ahány sor
44.    int maxi = 0;
45.    for (unsigned ez = 0; ez < vviv.size(); ez++) {
46.      jelen[ez] = 0;    //soronként megszámlálás
47.      for (int jel : vviv[ez])
48.        if (jel == 1)
49.          jelen[ez]++;
50.      if (jelen[ez] > jelen[maxi])
51.        maxi = ez;
52.    }
53.    return nev[maxi];
54.  }
55.
```

3. Volt-e olyan zenész, aki mindig jelen volt?
A kérdésre az eldöntés algoritmusa segítségével adhatunk választ. Ilyenkor nem szükséges végignézni minden adatot. Ha egy zenész egyszer hiányzott, akkor nem kell tovább néznünk az ő adatait és ha egy zenész végig jelen volt, akkor nem kell néznünk az utána következő zenészeket. A listák esetén használható bejáró-ciklusból „kiugrunk”, ha az adott kérdés eldőlt. A megoldásban a két ciklusra két logikai változót használunk, hogy a szerepük és a kiugrás helye egyértelmű legyen. Egyetlen logikai változóval is megoldható a feladat, de ennek helyes használata versenyszintű logikai feladat.

```
56.  bool mindig()
57.  {
58.    bool van = false;
59.    for (vector<int> tag : vviv) {
60.      bool mindig = true; //soronként eldöntés
61.      for (int jel : tag)
62.        if(jel == 0){
63.          mindig = false;
64.          break;
65.        }
66.      if (mindig){
67.        van = true;
68.        break;
69.      }
70.    }
71.    return van;
72.  }
73.
```

4. Írjuk ki, ha volt olyan nap, amikor mindenki jelen volt a próbán!
Erre a kérdésre az eldöntés és a megszámlálás algoritmusa alapján adható meg a válasz, de most nem a listába fűzött listák természetes bejárási sorrendjét használjuk.

   A bejárás során illik figyelni arra, hogy a sorok hossza különböző lehet, ezért mindig megvizsgáljuk, hogy van-e adat a sorban az adott napra. Mivel a kérdés arról szól, hogy mindenki jelen volt-e, ezért az első adatsor hossza alkalmas a vizsgálandó napok számának behatárolására (78. sor `vviv[0]` használata). Ha egyes adatsorok hosszabbak lennének, az

eredmény akkor sem módosulna, mert azokon a napokon az első zenész nincs jelen. A feladatot meg lehetne úgy is oldani, hogy először a megadjuk a legrövidebb adatsornak a hosszát (más feladattípus esetén a leghosszabbat lehetne meghatározni).

A megoldásban az eldöntéshez while-ciklust, a megszámláláshoz bejáró-ciklust használunk, így jól látható a 80–81. sorban, hogy az eldöntés minden ciklusléptében a teljes sort újra meg újra használjuk, megnézzük, hogy van-e az adott pozíción adat és ezt követ vizsgáljuk az adott napra beírt értéket. A külső ciklus nem lehet bejáró ciklus, mert a vizsgált napot egyenként nézzük az egyes sorokban.

```
74.  bool mindenki()
75.  {
76.    bool mindenki = false;
77.    unsigned ekkor = 0;
78.    while (ekkor < vviv[0].size() && !mindenki) {
79.      unsigned jelen = 0;
80.      for(vector<int> tag : vviv)
81.        if (ekkor < tag.size() && tag[ekkor] == 1) //sorhosszra figyelni!
82.          jelen++;
83.      if (jelen == vviv.size())
84.        mindenki = true;
85.      ekkor++;
86.    }
87.    return mindenki;
88.  }
```

Listák listájában tárolt adatok elemzéséhez sokszor hasznos eszköz a bejáró-lista, de ha indexre (sorszámra) vagyunk kíváncsiak, akkor ez a ciklus nem használható. A while-ciklus mindig használható, de általában áttekinthetőbb kódot eredményez a feltételekkel kiegészített for-ciklus. A kétdimenziós lista mérete nem rögzített, a két dimenzió nem egyenrangú. A „sorok” listák, az elemek a sorok elemei. Az egyes sorok hossza eltérő lehet, az a program futása során is módosulhat. Emellett a sorok száma is tetszés szerint módosítható.

Megoldás tömbben tárolt listákkal

Ez a megoldási mód azoknak lehet érdekes, akik a tömbök és listák (`vector<>`) használatában is gyakorlottak. Ezen belül fontos előismeret az adat helye (indexe) és az adat értéke közötti különbség, valamint a kapcsolatuk.

A megoldás során a zenészek száma legfeljebb 20 lehet, ekkora méretű tömböt foglalunk le. A mintaként használt vonósnégyes tagjainak száma a feladatok megoldása során paramétere lesz. A jelenléti adatokat minden zenészre külön listában tároljuk, ennek mennyiségét az adatok beolvasásakor megadjuk, de később a lista hosszát vesszük figyelembe. Az egyes részfeladatok megoldásánál nem a sokféleség, hanem a tiszta (könnyen értelmezhető), rövid kód írása a szempont.

```
1.  #include <iostream>
2.  #include <vector>
3.  using namespace std;
4.
5.  vector<int> tviv[20];
6.
```

```
 7.  void adatokBe(int tag, int nap)
 8.  {
 9.    for (int ez = 0; ez < tag; ez++)
10.      for (int ekkor = 0; ekkor < nap; ekkor++) {
11.        int adat;
12.        cin >> adat;
13.        tviv[ez].push_back(adat);
14.      }
15.  }
16.
17.  void adatokKi(int tag)
18.  {
19.    for (int ez = 0; ez < tag; ez++) {
20.      for (int jel : tviv[ez])
21.        cout << jel;
22.      cout << endl;
23.    }
24.  }
25.
```

1. A zenészek a közeli kifőzdében szoktak ebédelni. … Hány adagot fizetnek ezen a pénteken?
A megoldásban az adattípushoz igazodva, a külső ciklus számláló, a belső bejáró.

```
26.  int ebedDb(int tag)
27.  {
28.    int db = 0;
29.    for (int ez = 0; ez < tag; ez++)
30.      for (int jel : tviv[ez])
31.        if (jel == 1)
32.          db++;
33.    return db;
34.  }
35.
```

2. Melyik zenész volt a legtöbbet jelen a héten?
Nevezzük nevén! Hozzuk létre a zenészek névsorát:

```
36.  string nev[20] {"Heg Edu", "Vio Lina", "Brá Csaba", "Csel Lotti"};
     //a többi adat az "" lesz
37.
```

A ciklusokat az előző feladathoz hasonlóan szervezzük. A belső ciklus megszámlálás, ami a külső, maximum-kiválasztás ciklus számára értékeket állít elő. A maximum-kiválasztáshoz tároljuk az aktuális maximum indexét és értékét is.

```
38.  string legtobbJelen(int tag)
39.  {
40.    int maxi = 0;
41.    int maxe = -1;
42.    for (int ez = 0; ez < tag; ez++) {
43.      int jelen = 0;
44.      for (int jel : tviv[ez])
45.        if (jel == 1)
46.          jelen++;
47.      if (jelen > maxe){
48.        maxi = ez;
49.        maxe = jelen;
50.      }
51.    }
52.    return nev[maxi];
53.  }
54.
```

3. Volt-e olyan zenész, aki mindig jelen volt?
   A megoldáshoz az eldöntés algoritmusára kétszer van szükség. Mindkét esetben a feltételes for-ciklust használunk. Addig vizsgáljuk a zenészek jelenlétét, amíg nincs olyan, aki mindig ott volt. Egy-egy zenész esetén a jelenlétet addig figyeljük, amíg folyamatosan jelen-volt.

   A belső ciklusban az eldöntés algoritmusának egy speciális formáját kódoljuk. Nem azt keressük, hogy van-e (létezik-e) adott tulajdonságú elem, hanem azt, hogy mindegyik adott tulajdonságú-e. Ezért a ciklusfejben a jó (az elvárt) feltétel szerepel és a válaszunk akkor **true**, ha „végére ért az adatoknak”. Ugyanilyen jó de „optimista” megoldás az lenne, ha az alapértelmezett érték a **true** lenne, majd azt figyelnénk, hogy hiányzott-e a zenész. Ebben az esetben, ha hiányzás miatt szakad meg a ciklus, azaz nem ért a végére, a visszatérés értékét **false**-ra kell állítani.

   A megoldásban a belső for-ciklus ciklusfeje a feltételek miatt hosszú, de a ciklusmagban már nincs utasítás, amit a ciklusmag helyén egy pontosvessző jelöl.

```
55.  bool mindig(int tag)
56.  {
57.    bool mindig = false;
58.    for (int ez = 0; ez < tag && !mindig; ez++) {
59.      unsigned jelen; //soronként eldöntés
60.      for(jelen = 0;   jelen < tviv[ez].size() &&
                          tviv[ez][jelen] == 1; jelen++)
61.      ;
62.      if (jelen == tviv[ez].size())
63.        mindig = true;
64.    }
65.    return mindig;
66.  }
67.
```

4. Írjuk ki, ha volt olyan nap, amikor mindenki jelen volt a próbán!
   Mivel a kérdés arról szól, hogy mindenki jelen volt-e, ezért az első adatsor hossza alkalmas a vizsgálandó napok számának behatárolására. A megoldásban az eldöntéshez while-ciklust, a megszámláláshoz for-ciklust használunk, emiatt a 73. sorban az adatelemre sor-oszlop indexszel hivatkozunk.

```
68.  bool mindenki(int tag)
69.  {
70.    bool mindenki = false;
71.    for(unsigned ekkor=0; ekkor < tviv[0].size() && !mindenki; ekkor++) {
72.      int ez = 0;
73.      while (ez < tag && tviv[ez][ekkor] == 1)
74.        ez++;
75.      if (ez == tag)
76.        mindenki = true;
77.    }
78.    return mindenki;
79.  }
```

Kitekintés: Megoldás listában tárolt tömbökkel

Ez a fejezet átugorható, csak a teljesség kedvéért szerepel a jegyzetben. Tartalma a programozó szakmai ismeretek felé adnak kitekintést, de érdemi használatra nem tér ki. A fejezet megértéséhez szükséges a lista és a tömb adattípusok alapos ismerete, az előző három megoldás rutinos használata.

A mintaként használt vonósnégyes tagjainak száma a listába tett adatsorok számossága; a napok száma a feladatok megoldása során paraméter lesz. Az egyes részfeladatok megoldásánál a lista tulajdonságainak felhasználása és többféle megoldási mód bemutatása a szempont.

A megoldás erősen lebutított annak érdekében, hogy a tárolandó tömbök címei és adatai biztonságosan elérhetők legyenek. Ezért a lista elemei egy létező tömb soraiból lesznek „kiválasztva”. A megoldás során – hogy lehessen „választani” – a zenészek száma legfeljebb 10 lehet, a lefoglalt tömb kétszer ennyi sorból áll és soronként 7 darab egész szám tárolására ad módot.

C++ nyelven az egész számok tömbjének `int T[]` jelölése a matematikában szokásos jelölést követi. A fordítóprogram számára ez a jelölés nagyjából annyit jelent, hogy `T` néven el kell tárolni egy memóriacímet, ami egy int típusú adat helyének a kezdő címe lesz. Ugyanennek másik, technikai jelölése `int* T`. Mindkét írásmódban a `T` változó tartalma egy memóriacím. Mivel megadjuk, hogy `int` típusra mutat, ezért a memóriacím 1-gyel való növelése e következő szabad címre (4 Byte-tal odébb) fog mutatni. Amikor a szögletes zárójelek között a tömb méretét megadjuk, megnöveljük az adatok számára fenntartott méretet. A tömb elemeinek indexelésekor a `T` címéből, az adattípus méretéből és az index értékéből számítható az indexelt adat címe. A számítást nem akadályozza meg az, hogy az index a megengedettnél nagyobb, emiatt a tömbön kívüli adatok elérése is lehetséges – ez a túlindexelés –, ami programhibához vezethet. Amikor egy függvénynek vagy egy `vector<>`-nak adunk meg tömb adattípus, akkor mindig a kezdő elemének a címét adjuk át. A tömb elemeinek száma a program számára ismeretlen, a programozónak kell figyelnie arra, hogy milyen címekre hivatkozik.

```
1.  #include <iostream>
2.  #include <vector>
3.  using namespace std;
4.
5.  vector<int*> vtiv;  //int* azaz „int-re mutató” (pointer)
6.  int adatok[20][7];
7.
```

A fenti kódban a `vector<int*>` jelentheti azt is, hogy egész értékekre mutató címeket listázunk, de a címekkel megadhatjuk akárhány egész számokból álló adatsorozat kezdeteit is. Ennek szemléltetésére az alábbi adatbeolvasásban az a `vector<int*>` egy-egy elemét előre megadjuk, változóban tároljuk. Az `int* sor` az adatok tömb egyik sorát jelöli ki. A háttérben az `adatok` tömb címéhez hozzáadódik 2 * 7 *`ez`*4 Byte, ezt a címet tárolja a `sor`.

```
 8.  void listaBe(int tag, int nap) //tag <=10, nap <= 7
 9.  {
10.    for (int ez = 0; ez < tag; ez++) {
11.      int* sor = adatok[2 * ez]; //minden második sort használjuk
12.      for (int ekkor = 0; ekkor < nap; ekkor++)
13.        cin >> sor[ekkor];
14.      vtiv.push_back(sor);
15.    }
16.  }
17.
```

Kísérletezzünk: az adatok tömb lehet egydimenziós, így is megadható, hogy hányadik indexnél kezdődik a sor. A kétdimenziós tömbben az adatok „sorfolytonosan” vannak tárolva, ezért az sem okoz a programunkban gondot, ha egy közbülső sort túlindexelünk, mert a következő sor adatát fogja ott megtalálni.

A `vtiv` lista a tagok számának megfelelő számú címet tartalmaz, amelyek mindegyike egy-egy egész számra mutat. A bejáró ciklusunk változója ezért egész számra mutató pointer, azaz `int*`. A listaelemek kiírásakor a tárolt adatok megfelelő részét írjuk ki, programozóként tudjuk, hogy kiíráskor ugyanazt az értéket kell megadnunk paraméternek, mint amennyit a beolvasáskor megadtunk.

```
18.  void listaKi(int nap)
19.  {
20.    for (int* tag: vtiv) {
21.      for (int ekkor = 0; ekkor < nap; ekkor ++)
22.        cout << tag[ekkor];
23.      cout << endl;
24.    }
25.  }
26.
```

Kísérletezzünk: A `listaBe(4, 5)` eljárás futtatásával írjuk be a mintaadatokat, majd a `listaKi(70)` eljárással írjuk ki a listánkban tárolt tömbök adatait! Látható, hogy minden zenész első 5 adata helyes, az ezt követő 65 adatban megjelennek a listában utána következő zenészek adatai.

1. A zenészek a közeli kifőzdében szoktak ebédelni. … Hány adagot fizetnek ezen a pénteken? A megoldásban az adattípushoz igazodva, a külső ciklus számlálós is lehet, a `vector<int*>` típus elemei indexelve is elérhetők.

```
27.  int ebedDb(int nap)
28.  {
29.    int db = 0;
30.    for (unsigned ez = 0; ez < vtiv.size(); ez++)
31.      for (int ekkor = 0; ekkor < nap; ekkor ++)
32.        if (vtiv[ez][ekkor] == 1)
33.          db++;
34.    return db;
35.  }
36.
```

2. Melyik zenész volt a legtöbbet jelen a héten?
Használjuk a már megszokott névsort:

```
37.  string nev[20] {"Heg Edu", "Vio Lina", "Brá Csaba", "Csel Lotti"};
38.
```

Az alábbi megoldásban megfigyelhetjük, hogy a `vtiv` listában tárolt adat tényleg csak rámutat az eredeti helyre, az adatainkat az eredeti helyükön, az `adatok` tömbben is megtaláljuk.

```
39.  string legtobbJelen(int nap)
40.  {
41.    int jelen[20];
42.    int maxi = 0;
43.    for (unsigned ez = 0; ez < vtiv.size(); ez++) {
44.      jelen[ez] = 0;
45.      for (int ekkor = 0; ekkor < nap; ekkor ++)
46.        if (adatok[2 * ez][ekkor] == 1)  //másik nevén is elérjük a sort!
47.          jelen[ez]++;
48.      if (jelen[ez] > jelen[maxi])
49.        maxi = ez;
50.    }
51.    return nev[maxi];
52.  }
53.
```

3. Volt-e olyan zenész, aki mindig jelen volt?
A megoldáshoz az eldöntés algoritmusára kétszer van szükség. A belső ciklusban a hagyományos, while-ciklust használó eldöntés algoritmus ellentettét alkalmazzuk (kérdés: minden napra igaz-e) egy-egy zenész esetén, miközben a külső eldöntési algoritmusban bejáró ciklust és (ebből következően) kiugrást használunk.

A kódot megírva látható, hogy miután a logikai változó igazra vált, már csak a ciklus megszakítása és a visszatérési érték megadása szükséges. Ebben az esetben a visszatérési érték **`true`** lesz, minden más esetben az eredmény **`false`**. Logikai változóra nincs szükség, mert a ciklusból kiugrás a függvény végén folytatódna, így ott helyben befejezhető a függvény működése **`true`** érték visszaadásával; a mennyiben a program futása eléri a program végét ott az eredmény egyértelműen **`false`**.

```
54.  bool mindig(int nap)
55.  {
56.    //bool mindig = false;
57.    for (int* sor: vtiv) {
58.      int jelen = 0;    //soronként eldöntés
59.      while(jelen < nap && sor[jelen] == 1)
60.        jelen++;
61.      if (jelen == nap)
62.        return true;    //mindig = true; break; return mindig;
63.    }
64.    return false;       //return mindig;
65.  }
66.
```

4. Írjuk ki, ha volt olyan nap, amikor mindenki jelen volt a próbán!
   Az előző kérdéshez képest megfordul a külső és belső ciklus szerepe. Most a belső ciklus lehet bejáró, és ha bejáró ciklust használunk, akkor az eldöntés algoritmusához kiugró feltételt kell írni. A külső ciklus – mivel tömb adatait veszi sorra – nem lehet bejáró, de ekkor is megfogalmazhatunk a számláló ciklusból kiugró feltételt. Itt is igaz, hogy a külső ciklusból kiugrás lehet egyúttal a függvény befejezése is.

```
67.  bool mindenki(int nap)
68.  {
69.    for (int ekkor = 0; ekkor < nap; ekkor++) {
70.      bool mindenki = true;
71.      for (int* sor: vtiv)
72.        if(sor[ekkor] != 1) {
73.          mindenki = false;
74.          break;
75.        }
76.      if (mindenki)  //ha nem volt break;, akkor return true;
77.        return true;
78.    }
79.    return false;
80.  }
```

Másik megoldási lehetőség a listában tárolt tömbökre az lehet, hogy a tömböt valami olyan adatszerkezetbe tesszük bele, mint amilyen a `string` a karaktertömb számára. Erre ad majd lehetőséget az objektumok definiálása (meghatározása), az adattípust leíró osztály készítése. Egy kis kitérő után, a következő fejezet végén erre is fogunk példát látni.

Objektumok

Tömbök és listák esetén elvárás, hogy az elemek azonos típusúak legyenek. A megoldandó feladatokban szereplő dolgokról nagyon ritkán mondhatjuk el, hogy a róluk ismert adatok típusa azonos. Egy tanulóról például megadjuk a nevét, a nemét, a korát és az e-mail címét, akkor elgondolkodtató, hogy a kora lehet-e szöveges adat. Ha a tanulók legalább felső tagozatosak, de legfeljebb egy évet veszítve még nem érettségiztek, a szövegesen tárolt adatok is helyesen adják meg, hogy elmúltak-e 14 vagy 16 évesek. Számítást igénylő feladatoknál pedig a számítás előtt át lehet konvertálni az adatot a megfelelő típusba, az eredményt pedig vissza lehet alakítani szöveggé. Van olyan programozási nyelv, abban megírt alkalmazások, amelyek lényegében így kezelik a különböző típusú adatokat. Amikor egy program minden adatot szö-

veges fájlból olvas és az eredményt fájlba írja ki, akkor is, minden adat, minden számítási eredmény szövegként van tárolva, a konzolról is alapvetően szöveget olvasunk be és szöveget írunk ki.

Nagyon sok program a háttértáron szövegként tárolja az adatait, a bemenetén szöveget vár, de ha ezek a szövegek értéket jelentenek, akkor a program futásakor, az operatív memóriában célszerű az értéküket tárolni, mert a program hatékonysága jelentősen romlik, ha az értékkel végzett művelethez oda-vissza kell alakítani az adatokat. Ezért az egynemű adatsorok mellett a különböző típusú adatok összetartozásának jelzésére is vannak megoldások.

Ha csak annyi az elvárásunk, hogy egy dologról különböző típusú adatokat egyetlen változónéven tároljunk, akkor az adatokból rekordot, vagy más néven struktúrát képezhetünk. Ehhez nagyjából azt kell megadnunk, hogy mi lesz az új adattípusunk neve, valamint az egyes adatoknak, mintha belső változók lennének, megadjuk a típusát és a nevét. Az ilyen típusú változónknak egyenként meg tudjuk nézni minden adatát.

Mondatszerű leírásban a Tanuló rekord (struktúra) leírása és két Tanuló típusú változónak értékadására nézzük meg az alábbi mintát:

```
Tanuló:
    név : Szöveg
    nem : Karakter
    kor : Egész
    mail : Szöveg
ali : Tanuló
ali.név := "Kis Aladár"
ali.kor := 16
ali.nem := 'f'
bea : Tanuló("Nagy Beáta", 'l', 17, "")
```

Sok programozási nyelvre jellemző, hogy az egyes adatok a változón belül ponttal érhetők el. Ugyancsak jellemző, hogy az adatoknak tetszőleges sorrendben, akár csak részben adhatunk kezdőértéket, de figyelni kell arra, hogy aminek nem adtunk értéket, ott lehet, hogy memóriaszemét van. Jellemző az is a programozási nyelvekre, hogy a meghatározás sorrendjében egyszerre minden adatot megadhatunk, ilyenkor nem kell megmondani, hogy melyik belső változónak szánjuk az egyes értékeket.

Nem teljesen új a ponttal elválasztás. Az adatsorozatok `size()` tulajdonsága, a `vector<>` `push_back()` eljárása is ponttal kapcsolódik az adathoz, csakhogy az egyik függvény, a másik eljárás, de a hagyományos rekordnak ilyen részei nincsenek. Épp emiatt a modern programozási nyelvekben – ha meg is maradt a használatának lehetősége – az „élettelen” rekord helyett inkább az „élő” osztály definiálása a megszokott. Az osztály a hagyományos rekord továbbfejlesztett változatának tekinthető. Mennyiben több? Lássunk néhány alapvető tulajdonságot:

- Egy osztály nemcsak adattagokat tartalmazhat, hanem lehetnek függvényei eljárásai is.
- Az osztály adattagjait el lehet rejteni a felhasználó környezet elől, míg egy rekord minden adata publikus. Jellemző, hogy egy osztály adataihoz egyenként meghatározzák, hogy külső használói láthatják-e (olvasható-e az értéke) illetve tudják-e módosítani.
- Az osztálydefiníció (leírás) egy olyan adattípust határoz meg, amely kifelé egységet mutat. Ezért az így megadott összetett változó egy objektum. Ezzel szemben a hagyományos rekord csak adatok listája.

- Egy osztály definíciója tartalmazhat az adattípus létrehozására vonatkozó leírásokat, azaz a programozó által megírt konstruktorokat. A rekord típusú adat létrehozása egyféleképpen történhet, ami az alapértelmezett konstruktornak felel meg: a létrejött adat részeinek utólag kell megadni a kezdőértéket. Az objektumok konstruktorában a létrehozáskor (példányosításkor) az egyes részadatoknak értéket is adunk.
- Egy objektum klónozható, lehet olyan konstruktort írni, ami egy másik objektumpéldány másolataként állítja elő az objektumot.

Belegondolva, megállapíthatjuk, hogy az általunk használt programok mindegyike objektumokkal dolgozik: pixelekkel, bekezdésekkel, alakzatokkal, cellákkal, gombokkal, mezőkkel; a játékainkban autó, csillag, fegyver vagy manó objektumokkal manipulálunk. Az eddig megírt programjaink témája nagyon le volt szűkítve azzal, hogy minden dologról csak egy tulajdonságot vettünk figyelembe. Például egy taxis pénztárcájának a tartalmát figyeltük, de azt már nem, hogy mire költött, vagy hogy milyen hosszú utat tett meg.

Mivel a programokban döntő jelentősége van az osztályok meghatározásának és az objektumok használatának, a beszűkített lehetőségű rekord a C++ nyelvben annyira jelentéktelenné vált, hogy lényegében meg is szűnt. Bár az osztály definiálására a **`class`** szót használják, a fordító program számára a **`struct`** (a rekord hagyományos neve a C-alapú nyelvekben) szó lényegében ennek szinonimája. Egyetlen különbség van a két kulcsszó között: Ha nem adjuk meg hogy egy adattag vagy függvény a felhasználó számára látható legyen-e vagy sem, akkor a **`class`**-ban rejtve (**`private`**) lesz, míg a **`struct`**-ban látható (**`public`**) lesz.

1. Definiáljuk C# nyelven a **`Tanulo`** struktúrát és hozzunk létre két **`Tanulo`** típusú változót, amelyeknek (változó)neve `ali`, illetve `bea`!

```
 1.  #include <iostream>
 2.  using namespace std;
 3.
 4.  struct Tanulo     //struct vagy class, ezt követi a típus neve
 5.  {                 //kapcsos zárójelek között a lényeg: a definíció
 6.    string nev;     //belső adatok mintha változónevek lennének
 7.    char nem;
 8.    int kor;
 9.  };                //a pontosvessző is kell a végére
10.
11.  int main()
12.  {
13.    setlocale(LC_ALL, "");
14.    int a = 2;   //teszteléshez
15.    Tanulo ali;
16.    ali.nev = "Kis Aladár";
17.    ali.kor = 16;
18.    ali.nem = 'f';
19.
20.    Tanulo bea = {"Nagy Beáta", 'l', ali.kor}; //változót is megadhatunk
21.    return 0;
22.  }
```

2. Futtassuk a programot lépésenként! Figyeljük meg Watch ablakban is
   - az adat létrehozásának és a belső adattagok értékadásának a lépéseit;
   - a 21. sort követően a program visszalép a 20-as és 15-ös sorokra – ekkor fut le a destruktor – majd a 22. sorra lép. A 14. sorban egy egyszerű változó van, ennek nincs destruktora.

3. Módosítsuk a programot!
   - Mi történik, ha a **struct** szó helyett a **class** szót használjuk?
   - Mi történik, ha a bea objektum létrehozásakor felcseréljük az adatokat?
4. Írjunk konstruktort, amiben csak a nevet és a nemet adjuk meg, a kor legyen 14 év! Ilyenkor az alapértelmezett konstruktor eltűnik, ezért ezt is pótoljuk. Írjunk paraméter nélküli konstruktort!
   A konstruktor olyan, mint egy eljárás, aminek semmilyen visszatérési értéke sincs és a neve az osztály nevével megegyezik.

```
23.  #include <iostream>
24.  using namespace std;
25.
26.  struct Tanulo {
27.    string nev;
28.    char nem;
29.    int kor;
30.    Tanulo(string n, char c)     //konstruktor, void sincs
31.    {
32.      nev = n;    //a paraméterek alapján beállítjuk az értéket
33.      nem = c;
34.      kor = 14;   //nem kérdezzük, mert ennyinek kell lennie
35.    }
36.    Tanulo() //paraméter nélküli konstruktor
37.    {
38.      nev = "";   //praktikus mindig megadni valamit
39.      nem = 'x';
40.      kor = -1;
41.    }
42.  };
43.
44.  int main()
45.  {
46.    setlocale(LC_ALL, "");
47.    Tanulo ali("Kis Aladár", 'f');      //ali 14 éves lett
48.    ali.kor = 16;
49.    Tanulo bea;     //bea név nélküli, neme x, kora -1
50.    return 0;
51.  }
```

Az osztály neve és a konstruktorok neve azonos

5. Futtassuk lépésenként is a programunkat, Watch ablakban figyeljük meg az objektumok keletkezését, belső adatainak állapotát!
6. Írjunk a **Tanulo** osztályba
   - egy függvényt, ami megadja a tanuló első nevét és
   - egy öregítő eljárást, ami a korát 1 évvel növeli

   A függvényt és az eljárást így szeretnénk felhasználni:

```
35.  int main()
36.  {
37.  setlocale(LC_ALL, "");
38.  Tanulo ali("Kis Aladár", 'f');
39.  cout << ali.elsonev() << " kora " << ali.kor << " évről ";
40.  ali.oregit();
41.  cout << ali.kor << " évre módosult. " << endl;
42.  return 0;
43.  }
```

Függvények és eljárások számára az osztály egy elkülönített programterület. Mint egy program a programban. Az osztály adatai az osztályon belül globális változók, a függvényeken belül létrehozhatunk lokális változókat is. Egy lehetséges megoldás részlete így néz ki:

```
 4.  struct Tanulo {
   /*a program korábbi része*/
20.   string elsonev()
21.   {
22.    string s = "";
23.    for (unsigned i = 0; i < nev.size() && nev[i] != ' '; i++)
24.     s += nev[i];
25.    return s;
26.   }
27.
28.   void oregit()
29.   {
30.    kor++;
31.   }
32.  };
```

7. Mi a különbség az egyszerű változók definiálása és az objektum létrehozása között?
Elvileg sok, de a C++ nyelvben az egyszerű változót is lehet objektumnak tekinteni. Ezzel együtt az objektumról is beszélhetünk úgy, mintha egyszerű változó lenne. Lényegében tanulmányaink kezdete óta ezt tesszük, mivel a `string` valójában egy osztály, de „egyszerű” változóként kezeljük. Próbáljuk ki az alábbi létrehozási módokat, írjunk ki néhány adatot konzolra az eredmény ellenőrzéséhez:

```
44.   int a = 1;
45.   int b(a);
46.   int c{3 * b};
47.   int d = {3};
48.   Tanulo ali = {"Kis", 'f'};
49.   Tanulo bea(ali);
50.   bea.nem = 'l';
51.   Tanulo cili{"Ma Cili", 'l'};
52.   Tanulo dani("Lab Dani", 'f');
```

8. *Kitekintés:* Módosítsuk úgy a programot, hogy a `kor` értékét ne lehessen másképp módosítani a főprogramban, csak egyenként, az `oregit()` eljárást futtatva! Írjunk a `kor` lekérdezéséhez is egy függvényt! Ez a függvény csak abban az esetben írja ki a tanuló korát, ha paraméterként a „bocs” szöveget kapja, minden más esetben az eredmény −1 legyen! A megoldás kulcsa, hogy a `kor`-nak privát adatnak kell lennie, míg a többi adattag, a konstruktorok, eljárások és függvények publikusak maradnak.
A **`Tanulo`** osztály módosuló részei:

```
 4.  class Tanulo {   //maradhat struct is
 5.   private:
 6.    int kor;
 7.   public:
 8.    string nev;
   /*az osztály további sorai, kivéve a kor*/
36.    int kora(string jelszo)
37.    {
38.     return jelszo == "bocs" ? kor : -1;
39.    }
40.  };
```

Objektumok sorozata, táblázata

Az osztályleírás alapján programunkban a konstruktorokkal objektumokat hozunk létre. Az objektumokat – sok szempontból – változóknak tekintjük, de lehetnek eljárásaik, függvényeik is. Napjainkban egyre inkább elmosódik a határ az egyszerű és összetett adat között. Néhány programozási nyelvben az int típusnak is vannak tulajdonságai, függvényei; a `string` egyszerűnek tűnik, pedig karaktersorozatot tárol és számos függvénye van. A felhasználás szempontjából az osztályok adattípusok, az objektumok adatok, az objektumok neve változónév. Az azonos típusú objektumokat a már korábban tanult módon tudjuk adatsorozatban (tömbben, listában) tárolni.

Feladat

A következő feladatban egy-egy tanulónak megadjuk a nevét, a nemét és a korát. Ezt követően tanulócsoporttal kapcsolatos feladatokban használjuk a tanulók adatait.

1. Adjuk meg a csoport adatait a programban!
2. Kérjük be a felhasználótól egy (másik) csoport adatait!
3. Készítsünk eljárást, amellyel ki lehet írni a csoporttagok nevét és korát!
4. Írjuk ki a csoport átlagéletkorát!
5. Adjuk meg, hogy a lányok, vagy a fiúk vannak-e többen!
6. Kérjünk be egy nevet és írjuk ki az illető korát!
7. Léptessük a csoporttagok életkorát 1 évvel!

Megoldások

A **`Tanulo`** osztályban megadjuk a tulajdonságokat, az egyszerűség kedvéért mindegyik tulajdonság publikus lesz. Konstruktort nem írunk, az alapértelmezettet használjuk.

1. Adjuk meg a csoport adatait a programban.

   A csoport adatait tömbbe és listába is felvesszük, hogy később mindkettőt tudjuk használni. Mivel az adatsorozatok elemeit és az objektumok adattagjait is megadhatjuk kapcsoszárójelek között, az objektumok tömbje vagy listája a kétdimenziós adatsorozatokhoz hasonlóan adható meg, vesszővel elválasztott és kapcsoszárójelezett egységekben.

```
1.  #include <iostream>
2.  #include <vector>
3.
4.  using namespace std;
5.
6.  struct Tanulo {
7.    string nev;
8.    char nem;
9.    int kor;
10. };
11.
```

```
12. Tanulo csoport[3] = {
13.   {"Tamara", 'l', 15},
14.   {Tibi", 'f', 14},
15.   {"Tomi", 'f', 14}
16. };
17.
18. vector<Tanulo> csoportlista{
19.   {"Laci", 'f', 16},
20.   {"Lili", 'l', 17},
21.   {"Lujzi", 'l', 16}
22. };
23.
```

Ezt követően a feladatok megoldását a `main()` függvényben ezekre az adatokra vonatkozóan megoldhatjuk. Az egyszerű adatoktól annyiban tér el a megoldásunk, hogy nem elég az adat nevét beírni, meg kell adni – ponttal elválasztva – a megfelelő részét, ami már egyszerű adat.

A 3. feladatra – például – ez lehet a megoldás, ha nem eljárást írunk, hanem csak a főprogramban kódoljuk:

```
for(int i = 0; i < 3; i++)
  cout << csoport[i].nev << " = " << csoport[i].kor << endl;
for(unsigned i =0; i < csoportlista.size(); i++)
  cout<< csoportlista[i].nev << " = " << csoportlista[i].kor <<endl;
```

A 7. feladat megoldása az előzőktől annyiban tér el, hogy itt adatmódosítás is szükséges. Amennyiben az objektumokat indexükkel érjük el, akkor ez sem okoz problémát:

```
for (int i = 0; i < 3; i++){
  csoport[i].kor++;
  csoportlista[i].kor++;
}
```

Bonyolultabb a helyzet a bejárós ciklus használatakor, mert a bejárós ciklus egy `vector<>` objektumon belül levő adat másolatát használja. Ezt hiába módosítjuk, mert amint a ciklus a következő adatra lép, a módosítás elvész. Ezért a bejárós ciklusban – ha az adatot módosítani szeretnénk, akkor az adat – itt most a **`Tanulo`** objektum – címére van szükségünk.

Próbáljuk ki az alábbi programrészletet az `&` referencia operátor nélkül is:

```
for(Tanulo& t : csoportlista){
  t.kor++;
  cout << t.nev << " (" << t.kor << ")" << endl;
}
for(Tanulo t : csoportlista)
  cout << t.nev << " ++ " << t.kor << endl;
```

Megjegyzés: A bejárós ciklusban az `auto` kulcsszóval rábízhatjuk a fordítóprogramra, hogy kitalálja a bejáró adatnak a típusát, de azt már nem fogja kitalálni, hogy az adatra vagy a rá történő hivatkozásra van-e szükségünk. Ezért – ha módosítani szeretnénk az objektum bármely adatát, akkor – az `auto&` típus helyettesítővel kell jeleznünk, hogy az adatokra hivatkozással járja be az adatsort.

Mivel a következő feladatban futtatás során kérjük be az adatokat, készítsük elő egy szöveges fájlban azt, amit majd be szeretnénk írni. Ezt a listát CTRL+C-vel másolva, a megfelelő helyen CTRL+V-vel vagy jobb egérgombbal a konzolablakba kattintva „be tudjuk írni”.

A kétféle adatrögzítés miatt tömbből és listából is kettő lesz, ezért a további feladatok megoldásához paraméteres eljárásokat és függvényeket készítünk, minden feladatra kettőt: az egyik tömböt, a másik listát fog használni. A megoldás áttekinthetőbb lesz, ha a függvényeket a főprogram előtt csak deklaráljuk, a definíciókat – azt, hogy mit kell csinálniuk – a `main()` után írjuk meg.

A már megkezdett programunkat folytatva, adjuk meg az elkészítendő függvények deklarációit! A megoldás során fontoljuk meg az alábbiakat:

- Ha egy paraméter valamilyen adatok tömbje, akkor külön paraméterként meg kell adnunk az elemek számát is, mert a tömb jelöléssel csak az adatsorozat kezdőcímét adtuk meg. Ugyanakkor, mivel a címet adjuk át, a tömb elemeit (akár objektumokat is) az eljáráson belülről elérjük, tudjuk módosítani.
- Ha egy paraméter valamilyen adatok listája, akkor az összes adat „be van zárva” egy `vector<>` típusú objektumba, ezért ez egy objektum másolatának az átadását jelenti.

Ha módosítani akarjuk a listát vagy egyes elemeit – jelen esetben a lista **Tanulo** objektumának bármely adatát – akkor a lista címére kell hivatkozni az **&** operátorral.

- Ha nem akarunk módosítani egy adatot (csak megnézzük, felhasználjuk az értékeket), akkor az & operátor nem szükséges, de használata nem is árt. Ebből következően minden olyan paraméter esetén, ami nem tömb – nincs a neve után **[]** –, írhatunk **&** referenciaképző operátort. Ez a programunk futását is gyorsítja, mivel lista esetén a lista másolása is jelentős erőforrást igényelhet.

Az alábbi megoldásokban az első két pontot alkalmazzuk, de a harmadikat nem, mert így látszik, hogy hol okoz hibát az **&** hiánya. Érdemes kipróbálni a megoldásokat enélkül is.

A feladatoknak megfelelő eljárások és függvények deklarációi:

```
24. /*2. beolvasás*/
25. void inputT(Tanulo T[], int db);
26. void inputL(vector<Tanulo>& L, int db);
27. /*3. kiírás*/
28. void kiirT(Tanulo T[], int db);
29. void kiirL(vector<Tanulo> L);
30. /*4. átlagéletkor*/
31. double atlagT(Tanulo T[], int db);
32. double atlagL(vector<Tanulo> L);
33. /*5. fiú vs. lány többség*/
34. void tobbsegT(Tanulo T[], int db);
35. void tobbsegL(vector<Tanulo> L);
36. /*6. név keresés -> hány éves*/
37. int keresT(string nev, Tanulo T[], int db);
38. int keresL(string nev, vector<Tanulo> L);
39. /*7. évfolyamváltás (kor léptetése)*/
40. void lepT(Tanulo T[], int db);
41. void lepL(vector<Tanulo>& L);
42.
```

A főprogramban minden megoldást tesztelhetünk. Az alábbi minta programban mindegyik programrészt egyszer próbálunk ki, de ezt tetszőlegesen bővíthetjük, vagy csak egy-egy részletre fókuszálva, szűkíthetjük.

```
43. int main()
44. {
45.   setlocale(LC_ALL, "");
46.   Tanulo osztaly[4];
47.   vector<Tanulo> osztalylista;
48.
49.   inputT(osztaly, 4);
50.   inputL(osztalylista, 4);
51.   cout << "Átlag tömbből-> csoport: " << atlagT(csoport, 3);
52.   cout << " beírt osztály: " << atlagT(osztaly, 4) << endl;
53.   cout << "Átlag listából -> csoport: " << atlagL(csoportlista);
54.   cout << " beírt osztály: " << atlagL(osztalylista) << endl;
```

```
  tobbsegT(csoport, 3);
  tobbsegT(osztaly, 4);
  tobbsegL(csoportlista);
  tobbsegL(osztalylista);
  string nev;
  cout << "Név: "; cin >> nev;
  cout << nev << " " << keresT(nev, osztaly, 4) << " éves." << endl;
  cout << nev << " " << keresL(nev, csoportlista) << " éves." << endl;
  lepT(osztaly, 4);
  lepL(osztalylista);
  kiirT(osztaly, 4);
  kiirL(osztalylista);
  return 0;
}

```

2. Kérjük be a felhasználótól egy (másik) csoport adatait!
3. Készítsünk eljárást, amellyel ki lehet írni a csoporttagok nevét és korát!

A két feladat megoldása **Tanulo** objektumokat tartalmazó tömbökre, for-ciklussal. A tömbelemeket – a **Tanulo** adattípust – indexszel érjük el, majd ennek egy részét pont operátorral választjuk ki:

```
Anna l 15
Bella l 16
Csaba f 16
Dani f 17
```

```
void inputT(Tanulo T[], int db)
{
  for (int i = 0; i < db; i++) {
    cin >> T[i].nev >> T[i].nem >> T[i].kor;
  }
}

void kiirT(Tanulo T[], int db)
{
  for (int i = 0; i < db; i++)
    cout << T[i].nev << " " << T[i].kor << " éves." << endl;
}

```

A 2. és 3. feladat megoldása **vector<Tanulo>** listára több odafigyelést igényel. A beolvasásnál még nincs mit bejárni, ezért itt megadjuk az elemszámot és for-ciklussal számláljuk a beolvasott adatsorokat. Másik megoldás lehet a végjeles adatbevitel (például adat nélküli sor jelezheti a bevitel végét). Ekkor az objektumok számára nincs szükség.

Mivel adatbevitelről van szó, módosul a lista, ezért a lista paramétert hivatkozással kell megadnunk.

```
Ede f 15
Feri f 16
Gizi l 16
Hajni l 17
```

```
83.  void inputL(vector<Tanulo>& L, int db)
84.  {
85.    string s;
86.    char c;
87.    int k;
88.    for(int i = 0; i < db; i++) {
89.      cin >> s >> c >> k;
90.      L.push_back({s, c, k});
91.    }
92.  }
93.
```

Az adatok beolvasásának kódja láthatóan hosszabb egy lista esetén, mert a listába beillesztés előtt létre kell hozni az objektumot. Másik – de nem szép – megoldás lehet, hogy egy „üres” objektumot teszünk be a listába, majd az indexével hivatkozva az adatait felülírjuk. A kiírás viszont már nem jelent új kihívást. Használhatjuk a bejáró ciklust.

```
94.  void kiirL(vector<Tanulo> L)
95.  {
96.    for(Tanulo t : L)
97.      cout << t.nev << " (" << t.kor << ")"<< endl;
98.  }
99.
```

4. Írjuk ki a csoport átlagéletkorát!

```
100.  double atlagT(Tanulo T[], int db)
101.  {
102.    double a = 0;
103.    for (int i = 0; i < db; i++) {
104.      a += T[i].kor;
105.    }
106.    return a / db;
107.  }
108.
109.  double atlagL(vector<Tanulo> L)
110.  {
111.    double a = 0;
112.    for(Tanulo t : L)
113.      a += t.kor;
114.    return a / L.size();
115.  }
116.
```

5. Adjuk meg, hogy a lányok, vagy a fiúk vannak-e többen!

A megoldásnak többféle módja van, amelyek valamilyen formában a megszámláláshoz kapcsolódnak. Az objektumok használatában nincs újdonság, de változók számában spórolós következő megoldás.

```
117.  void tobbsegT(Tanulo T[], int db)
118.  {
119.    int merleg = 0;
120.    for (int i = 0; i < db; i++)
121.      if(T[i].nem =='f') merleg += 1;
122.      else merleg -=1;
123.    if(merleg > 0) cout << "Fiúk vannak többen." << endl;
124.    else if (merleg < 0) cout << "Lányok vannak többen." << endl;
125.    else cout << "Ugyanannyi fiú van mint lány." << endl;
126.  }
127.
```

Mivel az egyetlen „számláló” változónk értéke csak egy feltételtől függ, akár a háromoperandusú értékadást is használhatjuk.

```
128.  void tobbsegL(vector<Tanulo> L)
129.  {
130.    int merleg = 0;
131.    for(Tanulo t : L)
132.      merleg += (t.nem == 'f')? 1 : -1;
133.    //if, else másképp
134.    if(merleg > 0) cout << "Fiúk vannak többen." << endl;
135.    else if (merleg < 0) cout << "Lányok vannak többen." << endl;
136.    else cout << "Ugyanannyi fiú van mint lány." << endl;
137.  }
138.
```

6. Kérjünk be egy nevet és írjuk ki az illető korát!

A nevet a főprogramban kérjük be, itt paraméterként adjuk át. A megoldás a keresés algoritmusát használja:

```
139.  int keresT(string nev, Tanulo T[], int db)
140.  {
141.    int ez = 0;
142.    while (ez < db && T[ez].nev != nev)
143.      ez++;
144.    return ez<db? T[ez].kor : -1;
145.  }
146.
```

A listában kereséshez használhatjuk a bejárásból kiugrásos algoritmust. Mivel függvényt írunk, a kiugrás lehet egyben a függvényérték visszaadása is.

```
147.  int keresL(string nev, vector<Tanulo> L)
148.  {
149.    for(Tanulo t : L)
150.      if(t.nev == nev)
151.        return t.kor;
152.    return -1;
153.  }
154.
```

7. Léptessük a csoporttagok életkorát 1 évvel!

   A feladat egyszerű, de a listával történő megoldáshoz dupla odafigyelés szükséges. Mivel az adat változik, ezért a bejáró ciklusban hivatkozni kell az adatokra. Ráadásul ezzel a lista egésze is változik, emiatt az eljárás paraméterében is hivatkozásra van szükség.

```
155.  void lepT(Tanulo T[], int db)
156.  {
157.    for(int i = 0; i < db; i++)
158.      T[i].kor++;
159.  }
160.
```

```
161.  void lepL(vector<Tanulo>& L)
162.  {
163.    for(Tanulo& t : L)
164.      t.kor++;
165.  }
166.
```

ADATSOROZAT AZ OBJEKTUMBAN

Láthattuk, hogy az objektumok az egyszerű adatokhoz hasonlóan lehetnek elemei a különböző adatsorozatoknak. De a tartalmazási viszonyt meg is fordíthatjuk. Az objektumnak lehet adatsorozat típusú adata. Ennek kipróbálására vegyük elő újra, a vonósok jelenlétét firtató feladatunkat!

Feladatok

Adott egy vonósnégyes tagjainak jelenléti íve. Az ív egy hét munkanapjain mutatja a jelenlétet, ahol 1 szerepel benne, ott jelen volt a zenész, ahol 0, ott nem. Egy „kis lista” egy zenész jelenlétét mutatja be, a teljes lista pedig az egész zenekarét.

1. Hány adag ebédért fizetnek a zenészek ezen a héten?
2. Melyik zenész volt a legtöbbet jelen a héten?
3. Volt-e olyan zenész, aki mindig jelen volt?
4. Írjuk ki, ha volt olyan nap, amikor mindenki jelen volt a próbán!

Megoldás

Az eddigi négy megoldás közül a legnehezebb az volt, amelyikben a vonósok listáján belül próbáltuk a napok tömbjét kezelni. Most készítsünk a vonós heti jelenlétének adminisztrációjához egy osztályt. Így egy-egy vonós minden adata egy-egy objektumon belül lesz. Az előző fejezet alapján pedig a vonósok objektumait kezelhetjük akár tömbben, akár listában.

A **Vonos** osztály elkészítéséhez érdemes előre áttekinteni az összes megoldandó feladatot, megtervezni a megoldást:

- A korábbi négy megoldásban mindig külön tömbben vettük fel a vonósok nevét. Az osztály lehetőséget ad arra, hogy a nevet a jelenléti adatokkal együtt tároljuk. Ez szinte természetes … legfeljebb azon érdemes gondolkodni, hogy a név egyetlen szöveges adat legyen, vagy legyen külön vezetéknév és keresztnév adattag. Most a kevesebb munka érdekében a teljes név egy adat lesz.
- A jelenléti adatok tárolására is rengeteg módszer létezik. Lehetne naponként adattag, vagy napnevek – esetleg hosszabb időtartamra dátumok – listája, de most maradunk az eredeti „nehéz” megoldásnál: a vonós jelenlétét ötelemű tömbben tároljuk.
- Adatok beolvasása és konstruktorok készítése területén még inkább spórolunk az energiánkkal. Az alapértelmezett konstruktort használjuk és az adatokat beleírjuk a kódba.

- A feladatokat átnézve láthatjuk, hogy négy kérdésből háromban számít a zenész heti jelenlétének a száma, ezért erre az osztályon belül készítünk függvényt.
- Végül, most maradjunk annyiban, hogy minden adat publikus lesz, mert így egyszerűbb. A jelenléti adatokat hétről hétre egyébként is átírhatnánk, de a vonós nevének nem szabadna változtathatónak lennie. Megígérjük, hogy a lehetőség ellenére, a vonósok nevét nem írjuk át.

```
#include <iostream>
#include <vector>
using namespace std;

struct Vonos {
  string nev;
  int iv[5];
  int jelen()
  {
    int db = 0;
    for (int i = 0; i < 5; i++)
      if(iv[i] == 1)
        db++;
    return db;
  }
};

```

A zenekar adatait beírjuk a programkódba. A `vector<Vonos>` elemeit kapcsos-zárójelek között soroljuk fel. Az elemeket vesszővel választjuk el egymástól, minden elemnek kapcsos-zárójelek között adjuk meg az adatait. Az adattagokat is vesszővel választjuk el egymástól. Az első adattag egy `string` – idézőjelek között –, a második adattag egy tömb, ezért ennek kapcsos-zárójelek között adjuk meg az elemeit – az egész számokat –, vesszővel elválasztva.

Az egészet lehetne egy sorba írni, de célszerű az olvashatóságot segítő tördelést alkalmazni. A beírás során a nyitó-csukó zárójelpárokat egymásután írjuk be, ezt követően írjuk közé a tartalmat!

```
vector<Vonos> zenekar = {
  {"Heg Edu", {1, 1, 1, 1, 1}},
  {"Vio Lina", {1, 1, 1, 1, 0}},
  {"Brá Csaba", {1, 1, 0, 0, 0}},
  {"Csel Lotti", {0, 1, 1, 1, 1}}
};

```

1. Hány adag ebédért fizetnek a zenészek ezen a héten?
   Mivel egy-egy zenész heti jelenlétének számát az objektum saját függvénye megadja, itt csak ezek összegzésére van szükség.

```
int ebedDb()
{
  int db = 0;
  for (Vonos v : zenekar)
    db += v.jelen();    //itt nem feltételes az összegzés
  return db;
}

```

2. Melyik zenész volt a legtöbbet jelen a héten?
A válasz lehet a lista eleméből kiolvasott név, azaz egy `string`; lehet a megoldás listán belüli indexe – egész szám –, amelyen keresztül a név is elérhető. Most – hogy kipróbáljuk – egy **Vonos** objektum lesz a megoldás, amiben – természetesen – a név is szerepel.

   A maximális érték kiválasztása egy objektum esetén nem magától értetődő. Lehetne név szerinti rendezés alapján is leg-et választani. Ezért az objektumok közötti nagyságviszony mindig tisztázandó: a reláció most a `jelen()` függvényekben számított egész értékekre vonatkoznak.

```
33.  Vonos legtobbJelen()          //objektum lesz az eredmény
34.  {
35.    Vonos maxv = zenekar[0];     //az objektum egy változó
36.    for (Vonos v : zenekar)
37.       if (v.jelen() > maxv.jelen()) //a saját függvények értéke alapján
38.         maxv = v;   //a v minden adatának átmásolása maxv-be
39.    return maxv;
40.  }
41.
```

   Bár a megoldásunk jó, de nem igazán hatékony. Ha egy objektum függvényét használjuk fel, akkor arra számíthatunk, hogy minden egyes felhasználáshoz a kiszámítást elvégzi a programunk. `Heg Edu` volt legtöbbször jelen, aki a zenészek listájában az első. A megoldásunkban a `maxv.jelen()` minden esetben `Heg Edu` jelenléteinek az összegzése, így négyszer számolja ki a programunk ugyanazt. Mi lenne egy 100-tagú zenekar egyéves jelenléti statisztikája esetén?

   A megoldás hatékonyságát két helyen javíthatjuk:

   - A zenész objektum helyett a maximum keresése során a vizsgált értéket és az objektum indexét jegyezzük fel. Így a függvény végén az index alapján tudjuk megadni a visszatérési értéket.
   - Az osztály definícióban a `jelen()` függvény helyett változóban tárolhatjuk a heti jelenlétet. Ez azonban további függvények megírását teszik szükségessé, amelyek biztosítják, hogy az iv tömb elemeinek változása magával vonja az összesített érték változását is.

3. Volt-e olyan zenész, aki mindig jelen volt?
Az osztálydefiníció lehetővé teszi, hogy a korábban összetett algoritmus helyett itt is egy tipikus kódot, most éppen az eldöntés algoritmusát használjuk.

```
42.  bool mindig()
43.  {
44.    unsigned i = 0;
45.    while ( i < zenekar.size() && zenekar[i].jelen() != 5)
46.      i++;
47.    return i < zenekar.size();
48.  }
49.
```

   Azért a hatékonyságon itt is érdemes elgondolkodni … A while-ciklus magjában csak egy értéknövelés van, de a feltételében két függvény szerepel. Elvileg egy-egy ciklusléptésben programunk egyszer végig nézi a zenekar listát, hogy megállapítsa, hány tagja van – van-e

i-edik tagja. Ezt követően az i-edik tagra a jelenléti ív minden adatát megvizsgálja. Jelentősen javítana a helyzeten, ha a `zenekar.size()` értéket a legelején egy változóban eltárolnánk. Mégsem szokták ezt a programozók sem megtenni, mert ezzel romlik a kód olvashatósága. A program futását azonban nem lassítja le, mert a C++ fordítóprogramjai fel vannak készítve arra, hogy ilyen esetekben a kódot optimalizálják, azaz a bináris kód az elemek számát csak egyszer határozza meg.

4. Írjuk ki, ha volt olyan nap, amikor mindenki jelen volt a próbán!
Mivel egy-egy vonósra van objektumunk, ahhoz, hogy egy nap minden adata alapján döntésünk, minden egyes napot minden vonósnál meg kell vizsgálni. Így a megoldás hasonló lesz egy kétdimenziós tárolás oszlopok szerinti bejárásához.

```
50.  bool mindenki()
51.  {
52.    bool mindenki = false;
53.    unsigned ekkor = 0;
54.    while (ekkor < 5 && !mindenki) {
55.      unsigned i = 0;
56.      while(i < zenekar.size() && zenekar[i].iv[ekkor] == 1)
57.        i++;
58.      if (i == zenekar.size())
59.        mindenki = true;
60.      ekkor++;
61.    }
62.    return mindenki;
63.  }
64.
```

Felmerül a kérdés, hogy mi indokolja, hogy az adatokat a vonósok listájában tároljuk. Gondolkodhatunk másképp is: készítünk egy jelenléti ívet, amiben tároljuk a dátumot és a jelenlevők listáját. A heti jelenlétet ilyen jelenléti ívek listájával adjuk meg. Ez is megvalósítható, de a kérdések nagyobb része lenne nehezebben megoldható a napok listájában.

A főprogram:

```
65.  int main()
66.  {
67.    setlocale(LC_ALL, "");
68.    cout << "Heti ebédek száma: " << ebedDb() << endl;
69.    cout << "A legtöbbet jelenlevő zenész: "<< legtobbJelen().nev <<endl;
70.    cout << (mindig()? "Volt" : "Nem volt") <<" mindig jelenlevő."<<endl;
71.    cout << (mindenki()? "Volt" : "Nem volt") <<" teljes próba." << endl;
72.    return 0;
73.  }
```

Kiegészítés: objektum adatok beolvasása

A vonósnégyeshez vendégek érkeznek a menedzser kíséretében, akiknek a jelenlétét szintén regisztrálni kell. Adjuk hozzá a vonósok listájához a próbákon vendégként résztvevőket!

A megoldást nehezítsük azzal, hogy nem tudjuk, hány vendég jön, addig kérjük be az adatokat, amíg van neve az újonnan beírt zenekari „tag”-nak. Egy vendég esetén nem biztos, hogy egész hétre érdemes adatokat írogatni, főleg, ha csak a hét elején jön. Ezért a jelenlét megadásakor, ha ötnél kevesebb adatot írunk be, akkor a maradék napokra a program egészítse ki nulla értékekkel a hátralevő napokra a tömböt!

Látható, hogy objektumok esetén a beolvasás több helyen is megszakadhat, ráadásul a különböző típusú adatok bevitele sokféle igényt támaszthat – például ellenőrizni kellene, hogy a beírt szám 1 vagy 0 egyike-e. A megoldás a már megszokott „végjeles” bevitellel, akár do-while-ciklussal, akár előre olvasás után while-ciklussal kódoljuk, elég összetett, nehezen kezelhető. Ezért a C++ nyelv vezérlési szerkezeteiben lehetővé tették a logikai kifejezések mellett az eljárások vizsgálatát is. Egy eljárás akkor ad **true** értéket, ha hibamentes a végrehajtása, míg hiba **false** értéknek látszik.

Figyeljük meg, hogy a beolvasáskor hogyan használható ez a bővítés:

```
1.  void vendeg()
2.  {
3.    cout << "Add meg a vendégek adatait... vége: Enter." << endl;
4.    Vonos v;
5.    while (cout << "név: " && getline(cin, v.nev) && !v.nev.empty()) {
6.      cout << "Jelenlét (1|0 szóközökkel elválasztva): ";
7.      int nap;
8.      for (nap = 0; nap < 5 && cin >> v.iv[nap]; nap++)
9.        cout << "+"; //figyeljük meg, mikor jelenik meg!
10.     if (cin.fail())
11.     {//az >> nem tudta az adatot számmá alakítani, beragadt az adat
12.       cout << "A cin lezárva. Kinyitom." << endl;
13.       cin.clear();
14.       cout << "A továbbiakban a jelenlét: 0." << endl;
15.       for (; nap < 5; nap++) //kezdőérték megadása kimaradhat.
16.         v.iv[nap] = 0;
17.     }
18.     string qka; //az >> a szám utáni \n-t bent hagyja
19.     getline(cin, qka); //a következő sorolvasás előtt takarítás
20.     zenekar.push_back(v);
21.   }
22. }
```

Objektum adatainak beolvasásakor gyakran előfordul, hogy a szöveges adat több szóból áll, aminek a végét enter jelzi. Ennek beolvasására a `getline()` a praktikus. A számok, egyszerű értékek beolvasásához az extractor (`>>`) a megfelelőbb. Vegyes használatuk esetén figyelni kell arra, hogy az extractor a nem nyomtatható karaktereket (szóközt, entert, tabulátort) az adat előtt elnyeli, de az adat után otthagyja. Ezért a **cin >>** utáni `getline()` nem a következő sort fogja beolvasni, hanem az adott sort a sorvégi enterig.

Kétdimenziós adatsorozatok és objektumok a gyakorlatban

Híres szakemberek az adatszerkezetekről

- *Fred Brooks*, a világ egyik leghíresebb számítógéptudósa mondta még 1975-ben: „Mutasd meg a folyamatábráidat és rejtsd el a táblázataidat és homályban maradok. Mutasd meg a táblázataidat és rendszerint nem kellenek majd a folyamatábrák – minden nyilvánvaló lesz.” – Mai szemmel talán érdemes a folyamatábra helyére a „kód” szót, a táblázat helyére az „adatszerkezet” szót tennünk, és látjuk, hogy tényleg így van, mind a mai napig.
- *Linus Torvalds*, az első Linux megalkotója és a Linux kernel karbantartását végző, mára hatalmas fejlesztői csapat feje szerint „A rossz programozók aggódnak a kódjuk miatt. A jó programozók az adatszerkezeteikkel és azok kapcsolataival törődnek.”

- *Eric S. Raymond* szerint „Az okos adatszerkezet és a buta kód sokkal jobban működik, mint fordítva.”
- *Guido van Rossum*, a Python megalkotója pedig arra figyelmeztet: „Könnyű olyan hibákat elkövetni, amik csak később jönnek elő, miután rengeteg kódot megírtunk. Az ember ráébred, hogy másik adatszerkezetet kellett volna használni. Kezdhetjük elölről.”

Sokan, sokféleképpen megfogalmazták az adatszerkezet és az algoritmus kapcsolatát, de mindegyiknek lényeges vonása, hogy kiemeli a kettő kapcsolatát. Egy összetett adatszerkezet megírása sok időt, átgondolást, munkát igényel; használata is lehet nehézkesebb a pontok és zárójelezések miatt. Ráadásul az adatszerkezet elkészítése során a feladatban megfogalmazott kérdésre még nem adunk választ, nem látszanak a részeredmények. De egy jó adatszerkezet elkészítése után a programkód nagymértékben leegyszerűsödik, a kérdésekre a válaszok szinte maguktól adódnak.

Kiegészítés: Speciális adatsorozatok

Az adatstruktúra és algoritmus kapcsolatára már eddig is láthattunk példát. Tanulmányainkhoz elegendő a tömb adatstruktúra ismerete, de a feladatok megoldása során a `vector<>` típuson tudjuk használni a bejárós ciklust, ami lényegesen egyszerűbb a számlálós ciklusnál. Szerencsére a `vector<>` adatszerkezet létrehozása nem feladatunk, azt nagyon profi programozók sok éves tapasztalattal és tesztelés során írták meg.

A tömb és a `vector<>` adatstruktúrák között nem csak adatszerkezeti különbség van, hanem a felhasználásuk célja is más. A tömb véletlen elérésű, bármikor bármelyik elemét megadhatjuk; a `vector<>`-ba új elemeit csak a végére vagy az elejére hozzáfűzéssel adhatunk. Az adatsorozatok tárolására nem csak a `vector<>` létezik. A magas szintű programozási nyelvekre jellemző, hogy különböző tulajdonságú előre elkészített adattárolókat használhatunk. Közös tulajdonságuk, hogy több, azonos típusú adat – adatsorozat – tárolására készültek, úgy nevezett konténerek (container).

Az ismertebb konténerek C++ nyelven – a `vector<>`-on kívül:

- `array<>`: ez hasonlít leginkább a tömbhöz, mert fix a mérete, de ezt a méretet dinamikusan, a program futása közben is megadhatjuk.
- `queue<>`: általános nevén sor (FIFO), aminek csak a végéhez lehet adatot hozzáadni és csak az elejéről lehet törölni. A közbülső adatok láthatatlanok (nincs `[]`).
- `deque<>`: két-irányú sor, mindkét végéhez adhatunk új adatot és a végekről törölhetünk is, de belső adathoz nincs hozzáférésünk (nincs `[]`).
- `stack<>`: általános nevén verem (LIFO), aminek a végéhez tudunk adatot hozzáadni és ugyaninnen lehet törölni az adatot. A veremben (mélyebben) levő adatok láthatatlanok (nincs `[]`).
- `map<>`: általános néven szótár. Az index helyett is adatot tárol – ez a kulcs –, amely alapján rendez. Például egy értékadás: `szotar["alma"] = "apple";` Egy új elem beszúrásakor az elem a kulcs szerinti rendezésnek megfelelő helyre kerül, ezért a kulcs szerinti keresés nagyon gyors (bináris keresés), miközben az érték szerinti a tanult, lineáris keresést alkalmazza.
- `set<>`: általános nevén halmaz, amibe ugyanazt az adatot kétszer betéve is csak egy példányt fog tartalmazni.

A konténerek működését osztálydefiníciók tartalmazzák, ha használni szeretnénk őket, akkor az osztálydefiníciót be kell venni a programunkba. Pontosan úgy, ahogy a `vector<>` esetén, szükséges a program elején az `include<konténer_név>`.

Feladatok

Az alábbi feladatokhoz tervezzünk adatstruktúrákat, az egyes kérdések megválaszolásához, adjuk meg a megoldásához szükséges típus algoritmust, algoritmusokat! Ezt követően készítsük el a programot!

1. Rendelkezésünkre áll egy bolthálózat négy boltjának ezer forintban kifejezett bevétele az elmúlt hét napból.

130	29	143	133
156	15	222	132
231	210	98	182
112	11	101	121
96	191	184	148
311	14	201	199
231	302	87	187

   a) Melyik boltnak a legnagyobb a bevétele az elmúlt hét napban?
   b) Melyik boltnak a legnagyobb a tegnapi bevétele?
   c) Melyik boltban legnagyobb a legjobb és a legrosszabb nap közötti különbség?
   d) Mennyi volt a bolthálózat bevétele a tegnapi napon?
   e) Volt-e olyan nap, amelyiken a bolthálózat bevétele elérte a hatszázezer forintot?

2. Rendelkezésünkre áll a néhány európai ország ezer főben mért népessége (a régebbi adatok az akkor azon a területen élőkre vonatkoznak).

Ország	Évszám			
	0	1000	1500	2000 (1998-as adat)
Franciaország	5000	6500	15000	58805
Németország	3000	3500	12000	82029
Olaszország	7000	5000	10500	57592
Nagy-Britannia	800	2000	3942	59237

(Forrás: https://en.wikipedia.org/wiki/Demographics_of_Europe, 2020. 09. 17.)

   a) Melyik „országnak” a legnagyobb a népessége időszámításunk kezdetekor?
   b) Melyik az az „ország”, ahol találunk olyan évet, amelyben az előző adathoz képest alacsonyabb a népesség?
   c) Melyik „országnak” hányszorosára nőtt a népessége a megfigyelt két évezred alatt?

3. Készítsünk magyar–angol szótárt! Használjunk a szavak tárolására tömböt, `vector<>`-t, esetleg próbáljuk ki a `map<>` konténert!

Kulcs	Érték
nagy	big
pici	tiny
én	I
foci	soccer
ott	there
szarvas	deer
soha	never

a) Adjunk új szót – magyar és angol megfelelőjét – a szótárhoz!
b) A szó hozzáadását felhasználva töltsük fel a szótárt! néhány szópárral
c) Kérjünk be egy magyar szót, írjuk ki az angol megfelelőjét!
d) Kérjünk be egy angol szót, írjuk ki a magyar megfelelőjét!
e) Kérjünk be egy szót, írjuk ki az összes szópárt, amiben ez a szó magyar vagy angol megfelelőként szerepel!
f) Írjuk képernyőre a teljes szótárt a tárolás sorrendjében!

Tárgymutató

adattag 78, 80, 87, 88
argumentum.... 19, 21, 25
auto 82
break 41, 42
cast 55
continue 41
definíció.... 20, 21
deklaráció....20, 21, 82, 83
destruktor.... 78
escape karakter 4
extractor.... 5, 8, 91
függvénydefiníció 19, 20, 21
globális változó....21, 22, 28, 35
háromoperandusú operátor 30
index....40, 43, 45, 52
inicializál 7, 16
inserter 4
konstruktor 18, 79
lineáris keresés 45
mellékhatás.... 20, 22, 41
objektum..... 45, 63, 77, 78, 82, 88, 89, 90
osztály.... 76, 77, 78, 87
paraméter 8, 18, 19, 20, 21, 22, 25, 82, 83
példányosítás.... 78
referencia.... 22, 82, 83
rekord 1, 77, 78
specifikáció 55, 56, 57
struktúra 62, 63, 64, 77
szintaktika.... 18
unsigned 5, 27
visszaadott érték.... 19, 45, 47
visszatérési érték 18, 19, 20, 22, 24, 79