10 Algoritmizálás és a C# programozási nyelv használata

Előszó

Ez a jegyzet a Digitális kultúra tantárgy 10. évfolyamos tananyagának 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 Visual Studio 2019 Community 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, 2021.

Szalayné Tahy Zsuzsanna

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

TARTALOM

Eddig jutottunk eddig ... 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 ... 14

Eljárások, függvények ... 18
- Eljárást írunk ... 18
- Függvényt írunk ... 20
- Eljárás vagy függvény? ... 20
- Vissza a kezdetekhez! ... 21
- A program részekre bontásának szabályai ... 21
- Függvények és eljárások a gyakorlatban ... 24

Variációk típusalgoritmusokra ... 37
- Mik azok a típusalgoritmusok? ... 37
- Kódolási, jegyzetelési praktikák ... 37
- Történetek a taxisról meg a rókáról ... 37
- A sorozatszámítás – összegzés és átlagolás ... 38
- Eldöntés ... 40
- Kiválasztás ... 45
- Keresés ... 47
- Megszámolás ... 50
- Kiegészítés: kiválogatás ... 51
- Maximum- vagy minimumkiválasztás ... 52
- A típusalgoritmusok genetikája ... 56

Feladatok típusalgoritmusokra ... 56

Kétdimenziós adatszerkezet ... 65
- Mik azok a kétdimenziós adatszerkezetek? ... 65
- A 2D mátrix ... 67
- Kitekintés: Kétdimenzió két struktúrával ... 72

Objektumok ... 76
- Rekord a C# nyelvben: struct ... 78
- Az osztály (class) C#-ban és az igazi objektumok ... 84

Objektumok sorozata, táblázata ... 87

Adatsorozat az objektumban ... 92
- Kiegészítés: objektum adatok beolvasása ... 96

Kétdimenziós adatsorozatok és objektumok a gyakorlatban ... 98
- Kiegészítés: Speciális adatsorozatok ... 98

Tárgymutató ... 101

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!

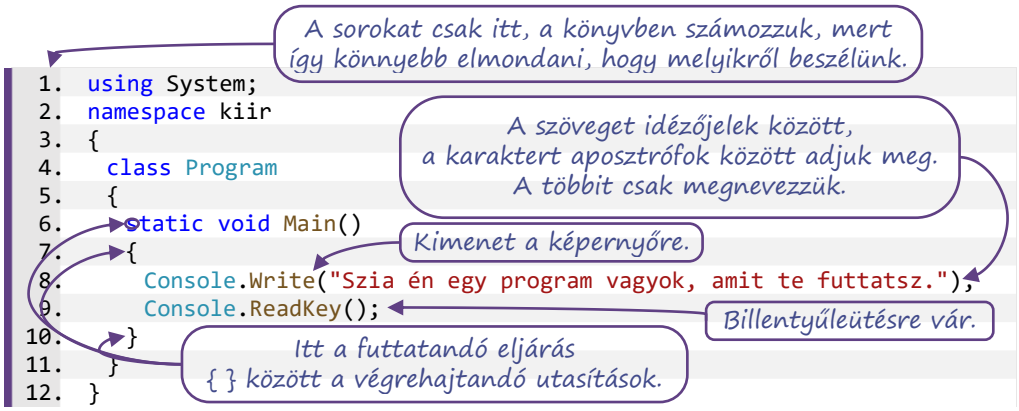
Forráskód, programozási nyelvek és fejlesztői környezet

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.

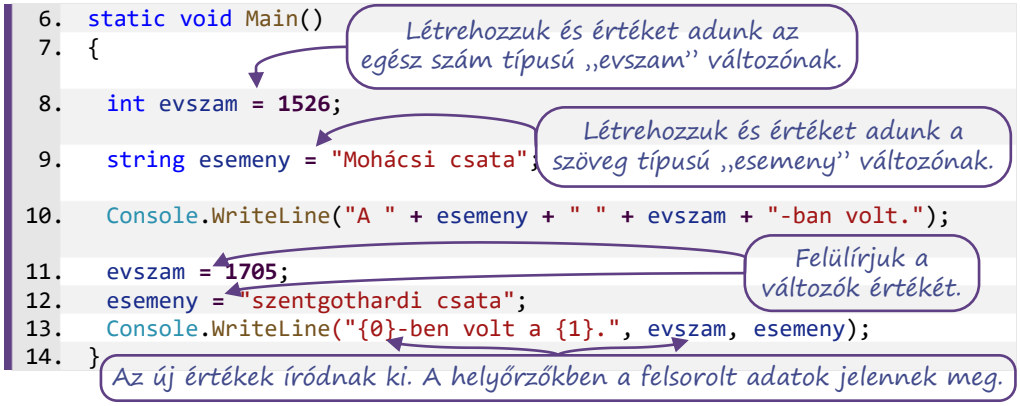
Nyissunk meg egy IDE-t – például Visual Studiot –, és írjuk meg benne azt a programot, amelyik elárulja, hogy épp egy programot futtatunk:




```
1.  using System;
2.  namespace kiir
3.  {
4.    class Program
5.    {
6.      static void Main()
7.      {
8.        Console.Write("Szia én egy program vagyok, amit te futtatsz.");
9.        Console.ReadKey();
10.     }
11.   }
12. }
```

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.




```
6.  static void Main()
7.  {
8.    int evszam = 1526;
9.    string esemeny = "Mohácsi csata";
10.   Console.WriteLine("A " + esemeny + " " + evszam + "-ban volt.");
11.   evszam = 1705;
12.   esemeny = "szentgothardi csata";
13.   Console.WriteLine("{0}-ben volt a {1}.", evszam, esemeny);
14. }
```

A fenti kódban a kimenetre (a képernyőre) az + jel „egymáshoz ragasztva” írja ki a szöveget és az adatokat. Eközben a számokból is szöveget (karaktersorozatot) készít. A kiíráskor a szóközökről nekünk kell gondoskodni. A `WriteLine()` eljárás „Line” része jelzi, hogy a sor végére Enter jel is kerül. Az ENTER leütésének megfelelője a `'\n'`, amit karakterként is beírhatunk, ezért az alábbi három utasításnak azonos a hatása:

```
Console.WriteLine("A " + esemeny + " " + evszam + "-ban volt.");
Console.Write("A " + esemeny + " " + evszam + "-ban volt.\n");
Console.Write("A " + esemeny + " " + evszam + "-ban volt." + '\n');
```

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).

A C# nyelv támogatja a szöveges, formázott kiírást, amelyben az adatokat helyőrzők segítségével illeszthetjük be a szövegbe a szöveg után felsorolt – 0-tól számozott – változókat. A kiírás ennél is „természetesebb”, de kevésbé átlátható módja a behelyettesítő mód, amelyben a helyőrzőbe az index helyett a változót adjuk meg.

```
Console.WriteLine("{0}-ben volt a {1}.", evszam, esemeny);
Console.WriteLine($"{evszam}-ben volt a {esemeny}.");
```

Az változó megjelenésének formáját a helyőrzőn belül adjuk meg: vesszőt követően az igazítás és a kiírás helyének hossza, kettőspont után a számformátum és tizedesjegyek száma írható be. Például a `{0,7:P2}` jelentése: a szöveg után elsőként (0) megadott változót 7 karakter hosszan, ezen belül jobbra igazítva (mert pozitív a szám), a P miatt százalékformátumban, két tizedesjegy pontossággal írja ki.

Még ebben a jegyzetben lesz arról szó, hogy a C# objektum orientált nyelv, emiatt minden változó, minden adat egyben objektum is. Minden objektumnak van szöveggé alakító függvénye, ez a `ToString`(). Kiíráskor az adatunkat – láthatatlanul is – ez a függvény alakítja szöveggé, a formátum ennek a függvénynek a paramétere.

Változók típusai, típusátalakítás, adatbekérés

A felhasználó billentyűzeten beírt adatait vagy a programba másik programból beküldött adatokat a `Console.ReadLine`() függvény fogadja és `string` adatként adja tovább. Az adat beolvasása mindig Enter-ig, végjelig történik, ha egyszerre (egy sorba) több adatot kap a program, akkor a beolvasott szöveget fel kell darabolni a `Split`(`char határoló`) függvény segítségével. A határozó általában a szóköz (`' '`), az eredmény egy szövegeket tartalmazó adattömb. Ha a változónk nem `string` típusú, akkor az adatot a megfelelő típusra át kell alakítani (azaz konvertálni kell). Erre minden adattípusnak van `Parse`() függvénye vagy használhatjuk a `Convert` eszközt.

```
 6.  static void Main()
 7.  {
 8.    Console.Write("Szóközzel elválasztva add meg a
                      szorzandót és a szorzót: ");
 9.    string sor = Console.ReadLine();
10.    string[] adatok = sor.Split(' ');
11.    int szorzando = int.Parse(adatok[0]);
12.    int szorzo = Convert.ToInt32(adatok[1]);
13.    Console.WriteLine("{0} * {1} = {2}",
                      szorzando, szorzo, szorzando * szorzo);
14.  }
```

Eddig összesen ötféle adattípust tároltunk változóban:

- karaktersorozatot, más néven szöveget: `string`;
- egész számot: `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.  using System;
2.  namespace elagazas
3.  {
4.    class Program
5.    {
6.      static void Main()
7.      {
8.        Console.WriteLine("Ki a Piroska nevű szuperhős fő ellensége? ");
9.        string ellenseg = Console.ReadLine();
10.       if (ellenseg == "farkas" || ellenseg == "Farkas")
11.       {
12.         Console.WriteLine("Okos vagy.");
13.         Console.WriteLine("Nem kicsit.");
14.       }
15.       else
16.       {
17.         Console.WriteLine("Hááát...");
18.         Console.WriteLine("Nem.");
19.       }
20.     Console.WriteLine("Legközelebb a hét törpét kérdezem.");
21.     }
22.   }
23. }
```

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.

```
6.  static void Main()
7.  {
8.    int valasz = 0;
9.    while (valasz != 4)
10.   {
11.     Console.Write("Mennyi kétszer kettő? ");
12.     valasz = int.Parse(Console.ReadLine());
13.   }
14.   Console.WriteLine("Annyi");
15. }
```

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, hátul tesztelő do-while-ciklussal. Ekkor a ciklusmag utasításait egyszer ellenőrzés nélkül végrehajtja a programunk, a végén a feltétellel azt vizsgáljuk, hogy szükséges-e ismétlés.

```
16.  static void Main()
17.  {
18.    int valasz;
19.    do
20.    {
21.      Console.Write("Mennyi kétszer kettő? ");
22.      valasz = int.Parse(Console.ReadLine());
23.    } while (valasz != 4);
24.    Console.WriteLine("Annyi");
25.  }
```

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 dinamikus adatsorozat, ezért méretét – nem negatív szám – a használat előtt meg kell adni. A tömb összes adatának együtt egy változóneve van. Azzal különböztetjük meg az egyszerű adattól, hogy a tömböt alkotó adattípus megnevezését kiegészítjük a **[ ]** zárójel párral. Ezután a tömböt létre kell hozni (new …) ekkor adjuk meg, 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 kellően nagyra méretezzük, hogy biztosan beleférjen minden adat. Ha a tömböt csak részlegesen töltjük fel, akkor 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.  using System;
2.  namespace for_cilkus
3.  {
4.    class Program
5.    {
6.      static void Main()
7.      {
8.        string[] varosok = new string[4]
                             { "Miskolc", "Párizs", "Dublin", "Lajosmizse" };
9.        for (int i = 0; i < 4; i++)
10.         Console.WriteLine(varosok[i] + " egy város Európában.");
11.     }
12.   }
13. }
```

több string egy néven.

létrehozzuk a 4 elemű tömböt.

Megmondjuk, mi a 4 elem.

i számlál 0-tól.

Belép, ha ez igaz.

Utána növeli i értékét.

Egy utasítás ➔ nem kell { }.

A lista dinamikusan bővíthető adatsorozat, a C# lista típusa a `List<>`. 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 (**Count**). 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 mindig 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.  using System;
2.  using System.Collections.Generic;
3.  namespace foreach_cilkus
4.  {
5.    class Program
6.    {
7.      static void Main()
8.      {
9.        List<string> varosok = new List<string>(4)
                                  { "Miskolc", "Párizs", "Dublin" };
10.       Console.Write(varosok.Capacity + " helyen ");  //kapacitás: 4
11.       Console.WriteLine(varosok.Count + " adat "); //adatok száma: 3
12.
13.       for (int i = 0; i < varosok.Count; i++)
14.         Console.WriteLine(varosok[i] + " egy város Európában.");
15.
16.       varosok.Add("Lajosmizse");
17.       foreach (string varos in varosok)
18.         Console.WriteLine(varos + " egy város Európában.");
19.     }
20.   }
21. }
```

dinamikus tároló eszközök csomagja.

3 adatot beleteszünk.

string-eket tároló lista.

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. 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 `List<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` méretét a **Length** tulajdonsága jellemzi, ami a(z 1 bájtos) karaktereinek száma.

A `string` adatoknak sok függvénye közül a `Split`() függvényt gyakran kell használnunk a beolvasott adatok darabolására. Azt is tudjuk, hogy két szöveg egy összeadásjellel összefűzhető, ami egymás mellé írást jelent. Bár a szöveg karaktereit a szövegbéli sorszámukkal egyenként elérjük, ezek módosítása felülírással nem lehetséges. A módosítás a szöveg függvényeivel oldható meg, vagy a `ToCharArray`() függvénnyel átalakíthatjuk a szövegünket karakterek tömbjévé.

Nézzük meg az alábbi kódban a `string`-ek és karaktersorozatok átalakításának módjait! Mi történik a szöveggel?

```
using System;
using System.Collections.Generic;
namespace karaktersorozatok
{
  class Program
  {
    static void Main()
    {
      string szoveg = "ez egy szöveg";
      int szovegDB = szoveg.Length; //karakterek száma: 13

      string[] szavak = szoveg.Split(' ');
      int szavakDB = szavak.Length; //a tömbnek is hossza van: 3
      List<string> szolista = new List<string>(szavak);

      char[] betuk = szoveg.ToCharArray(); //betűnként tömbben
      List<char> betulista = new List<char>(betuk);

      Console.WriteLine(szoveg.ToUpper());

      for (int i = 0; i < szavakDB; i++)
        Console.Write(szavak[i].ToUpper()[0]);
      Console.WriteLine();

      foreach (string szo in szolista)
      {
        string ujszo = szo.Replace('e', 'E');
        Console.Write(ujszo);
      }
      Console.WriteLine();

      for (int i = 0; i < betuk.Length; i++)
      {
        if (i == 0 || betuk[i - 1] == ' ')
          betuk[i] = char.ToUpper(betuk[i]);
        Console.Write(betuk[i]);
      }
      Console.WriteLine();

      string duma = "";
      foreach (char karakter in betulista)
        if(karakter != ' ')
          duma += karakter;
      Console.WriteLine(duma);
    }
  }
}
```

Milyen esetben dorgálja meg az alábbi programrészlet a felhasználót?

```
Console.Write("Írj be egy mondatot! ");
string mondat = Console.ReadLine();
int vege = mondat.Length - 1;
if (mondat[vege] != '!' && mondat[vege] != '?' && mondat[vege] != '.')
  Console.WriteLine("Ejnye-bejnye!");
else
  Console.WriteLine("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)!

```
 8.  string mondat;
 9.  do
10.  {
11.    Console.WriteLine("Írj be egy mondatot! ");
12.    mondat = Console.ReadLine();
13.    int vege = mondat.Length - 1;
14.    if (mondat != "")
15.    {
16.      if (mondat[vege] !='!' && mondat[vege] !='?' && mondat[vege] !='.')
17.        Console.WriteLine("Ejnye-bejnye!");
18.      else
19.        Console.WriteLine("Igazán gyönyörű mondat.");
20.    }
21.  } while (mondat != "");
```

A TANULTAK ALKALMAZÁSA

A kódolást segítik a snippetek: cw, do, for, foreach, if, el, switch vagy while, majd kétszer leütni a TAB billentyűt.

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 `Console.WriteLine()` 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
```

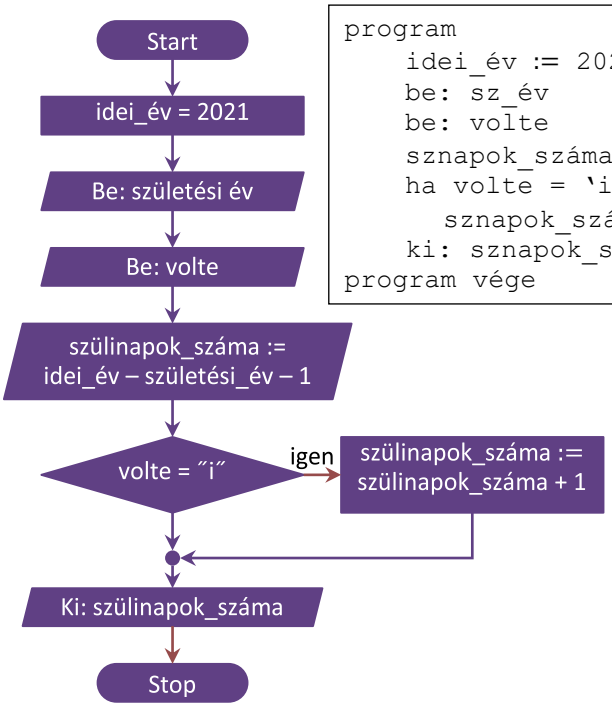
a) Mit csinál a program?
b) Írjuk át a mondatszerű leírást folyamatábrává!
c) Kódoljuk a programot!

```
 8.  Console.Write("Add meg egy állatfaj nevét! ");
 9.  string allat = Console.ReadLine();
10.  Console.Write("Add meg az állatfaj sebességét! ");
11.  int sebesseg = int.Parse(Console.ReadLine());
12.  string hol;
13.  if (sebesseg <= 50)
14.    hol = "a városban";
15.  else if (sebesseg <= 90)
16.    hol = "az országúton";
17.  else
18.    hol = "az autópályán";
19.  Console.WriteLine("A(z) {0} legnagyobb sebességével {1} haladhat.",
                        allat, hol);
```

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.



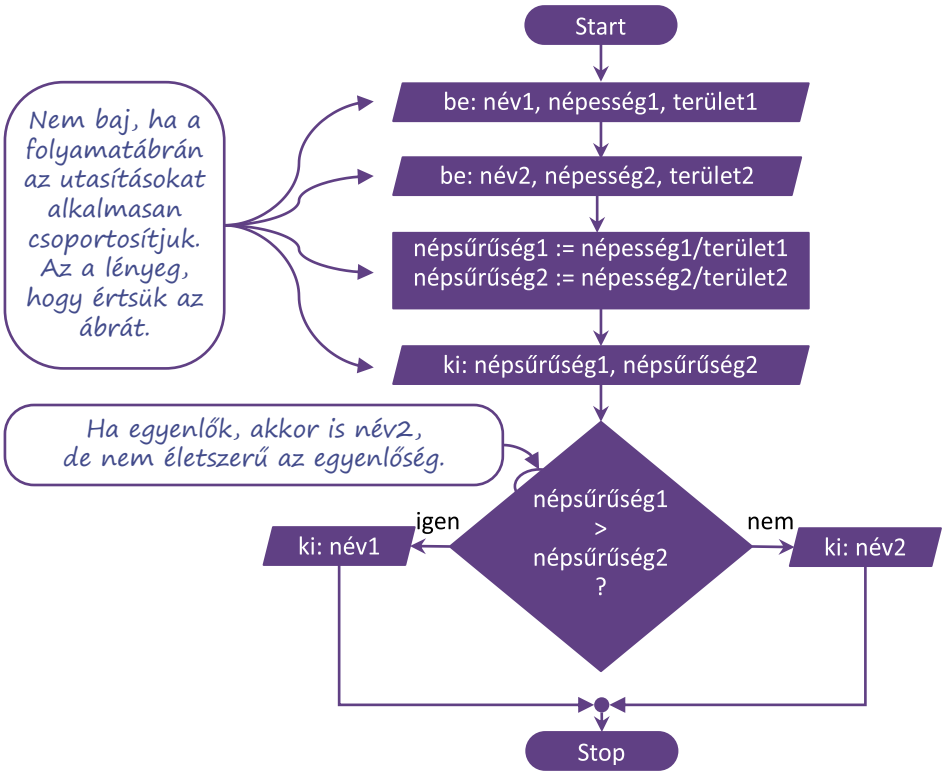

```
program
    idei_év := 2021
    be: sz_év
    be: volte
    sznapok_száma := idei_év - sz_év - 1
    ha volte = 'i':
       sznapok_száma := sznapok_száma + 1
    ki: sznapok_száma
program vége
```

```
 8.  const int ideiEv = 2021;
 9.  int sznapDB;
10.  Console.Write("Melyik evben szulettel? ");
11.  int szulEv = int.Parse(Console.ReadLine());
12.  Console.Write("Volt mar iden szuletesnapod? (igen/nem) ");
13.  string volte = Console.ReadLine();
14.  sznapDB = ideiEv - szulEv - 1;
15.  if (volte == "igen")
16.    sznapDB += 1;
17.  Console.WriteLine($"Te {sznapDB} eves vagy most.");
```

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!

```
 8.  int egyik, masik, maradek, oszto, osztando;
 9.  Console.Write("Mi legyen az egyik szam? ");
10.  egyik = int.Parse(Console.ReadLine());
11.  Console.Write("Mi legyen a masik szam?");
12.  masik = int.Parse(Console.ReadLine());
13.  if (egyik >= masik) //mindig a nagyobb szám legyen az osztandó
14.  {
15.    osztando = egyik;
16.    oszto = masik;
17.  }
18.  else
19.  {
20.    osztando = masik;
21.    oszto = egyik;
22.  }
23.  maradek = osztando % oszto; /*A maradékos osztás jele a %.*/
24.  if (maradek == 0)
25.    Console.Write("Az " + oszto + " osztója " + osztando + "-nek.");
26.  else
27.    Console.Write("Az " + oszto + " nem osztója " + osztando + "-nek.");
```

Egysoros megjegyzés.

Többsoros megjegyzés.

   d) Módosítsuk úgy a programunkat, hogy csak a valódi osztókat minősítse osztónak!

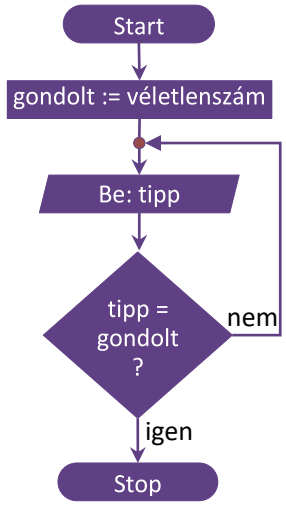
e) Írjuk meg a megoldást úgy, hogy az alapműveletek közül csak az összeadást és kivonást használhatjuk.

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 elöltesztelő (while) ciklussal:

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

… és hátul-tesztelő (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 c# kifejezésekre!)

b) Kódoljuk az algoritmust!

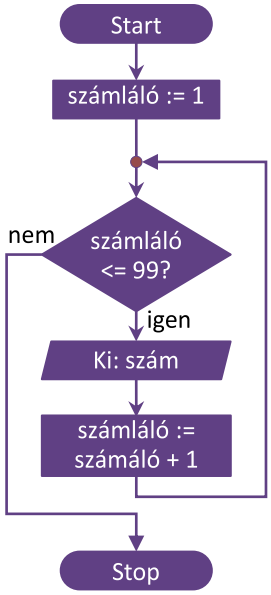
```
 8.  Random dobo = new Random();   //véletlenszám generáló eszköz
 9.  int gondolt = dobo.Next(10) + 1;
10.  int tipp;           //lehet: int tipp = -1;
11.  do                  //és itt: while (tipp != gondolt)
12.  {
13.    Console.Write("Tippelj: ");
14.    tipp = int.Parse(Console.ReadLine());
15.  } while (tipp != gondolt); //ha elején van, akkor itt nincs while();
```

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:

```
 8.  int szamlalo = 1;
 9.  while (szamlalo <= 99)
10.  {
11.    Console.WriteLine("Hurrá!");
12.    szamlalo = szamlalo + 1;
13.  }
```

b) Oldjuk meg a feladatot számlálós ciklussal is!

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

C# kód részlete:

```
14.  for (int i = 0; i < 99; i++)
15.    Console.WriteLine("Hurrá!");
```

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 (`List<>`-et): "barack", "körte", "dinnye", "narancs"! Írjuk ki bejárós ciklussal mindegyikről, hogy egy gyümölcs!

C# kód:

```
 1.  using System;
 2.  using System.Collections.Generic;
 3.  namespace gyumilista
 4.  {
 5.    class Program
 6.    {
 7.      static void Main()
 8.      {
 9.        List<string> gyumik = new List<string> //nem kell új sor
           { "barack", "körte", "dinnye", "narancs" };
10.        foreach (string gyumi in gyumik)
11.          Console.WriteLine($"A {gyumi} egy gyümölcs.");
12.      }
13.    }
14.  }
```

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

```
 8.  for (int i = 3; i <= 100; i += 3)
 9.    Console.WriteLine("A(z) " + i + " osztható hárommal");
```

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:

```
 8.  Console.Write("Honnan induljunk? ");
 9.  int eleje = int.Parse(Console.ReadLine());
10.  Console.Write("Meddig számoljunk el? ");
11.  int vege = int.Parse(Console.ReadLine());
12.  Console.Write("Hánnyal osztható számokat keresünk? ");
13.  int oszto = int.Parse(Console.ReadLine());
14.  int darab = 0;
15.  for (int i = eleje; i <= vege; i++)
16.  {
17.    if (i % oszto == 0)
18.    {
19.      Console.Write(" {0} ", i);
20.      darab += 1;
21.    }
22.    else
23.      Console.Write(".");
24.  }
25.  Console.WriteLine("\n{0} ilyen számot találtam.", darab);
```

A feltételes ciklust használó megoldásban a 24. és a 15. sor elé kell egy-egy sort beszúrni. A 15. 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 létrehozzuk a véletlenszámgenerátort.

```
1.  using System;
2.  using System.Collections.Generic;
3.  namespace jarmutomegek
4.  {
5.    class Program
6.    {
7.      static void Main()
8.      {
9.        Random rnd = new Random();
```

A feladathoz szükséges adatsorozatot tárolhatjuk …

`int[]` tömbben:

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

`List<int>` listában:

```
10.  List<int> tomegek =
                    new List<int>(20);
11.  for (int i = 0; i < 20; i++)
12.    tomegek.Add(rnd.Next(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:

```
13.       int ez = 0;                //indexeléshez
14.       bool vanNehez = false;     //a) feladathoz kezdőérték
15.       int nehezDB = 0;           //b) feladathoz feltételes számláló
16.       int ossztomeg = 0;         //c) feladathoz részösszeg tárolása
17.       int nehezOssztomeg = 0;    //d) feladathoz feltételes összegzés
18.       while (ez < 20)
19.       {
20.         Console.Write(tomegek[ez] + " ");
21.         ossztomeg = ossztomeg + tomegek[ez]; //jó így is a hozzáadás
22.         if (tomegek[ez] > 9300)
23.         {
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.       Console.WriteLine("\nVálaszok:");
31.       if (vanNehez)
32.         Console.WriteLine("Volt 9300 kilónál nehezebb jármű");
33.       Console.WriteLine("{0} db 9300 kilónál nehezebb jármű volt.",
                            nehezDB);
34.       Console.WriteLine("Ma " + ossztomeg + " kg-ot vitt át a komp.");
35.       Console.WriteLine($"Ebből a nehéz járművek össztömege
                            {nehezOssztomeg} kg.");
36.     }
37.   }
38.  }
```

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.  using System;
2.  namespace alahuzasok
3.  {
4.  class Program
5.    {
6.      static void Alahuzas()
7.      {
8.        for (int i = 0; i < 10; i++)
9.          Console.Write("-");
10.       Console.WriteLine();
11.     }
12.
13.     static void Main()
14.     {
15.       Console.WriteLine("Ez egy fontos figyelmeztetés!");
16.       Alahuzas();
17.       Console.WriteLine("Minden sora fontos!");
18.       Alahuzas();
19.       Console.WriteLine("Komolyan!");
20.       Alahuzas();
21.     }
22.   }
23. }
```

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.*

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

Az eljárásunk szerkezete nagyon hasonlít a főprogramunkhoz, mert a főprogram is egy eljárás. Az `Alahuzas`() eljárást a `Main`()-ben meghívjuk, arra utasítjuk a gépünket, hogy azon a ponton hajtsa végre az utasításokat. Korábban is meghívtunk eljárásokat, legelőször a `Console.WriteLine`() eljárást. Az a System csomagban megtalálható, előre elkészített eljárás, az `Alahuzas`() pedig egy általunk készített eljárás, de a használatuk egyformán történik. Az eljárásnak nincs a futás után, a továbbiakban használható eredménye, nem ad semmit az őt meghívónak. Ezt a semmit jelzi a „`void`”, aminek a jelentése: üres. Ha lenne valamilyen felhasználható eredmény, akkor nem eljárás lenne, hanem függvény, mint például a `ReadLine`(), aminek a lefutás utáni eredménye a beolvasott szöveg. A függvénynek van eredménye, szaknyelven: visszatérési értéke, ott ennek az értéknek a típusát adjuk meg a függvény neve előtt.

Az eljárásokat, függvényeket mindig egy osztályon belül kell írni. Még csak a `Program` osztály létezik a programunkban, ezért abba, a {} jelei közé írjuk. Figyeljünk arra, hogy a `class`-on belül, de a többi eljárás vagy függvényen kívül – alatta vagy fölötte – definiáljuk (kódoljuk) az új eljárást/függvényt. Amikor a fordítóprogram a kódból elkészíti a futtatható exe fájlt, a kódot elemzi: a felhasználás előtt kell megismernie az `Alahuzas()` eljárást. Ezért, ha egy addig ismeretlen eljárással találkozik, akkor az adott osztályon belül keresi a definíciót.

A `Program` osztály specialitása, hogy az operációsrendszer hívja meg és nem egy másik osztály valamelyik függvénye/eljárása, ezért ezen az osztályon belül minden kódolási egység elé ki kell írni a `static` jelzőt.

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, azaz nincs visszatérési értéke, ugyanakkor a végrehajtása során a memóriában létrehoz valamit, mintha függvény lenne. Ez a létrehozó, rendes nevén a konstruktor. A C# nyelv az eljárást speciális függvénynek tekinti, ezért szintaktikailag (nyelvtanilag) a függvénnyel azonos írásmódot alkalmazzuk. Egy nem objektumorientált nyelvben nincs az objektum, ezért annak nincs konstruktora sem, amit így nem kell megkülönböztetni az eljárástól.

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.

```
24.  using System;
25.  namespace alahuzasok
26.  {
27.  class Program
28.   {
29.     static void Alahuzas(char jel)
30.     {
31.       for (int i = 0; i < 10; i++)
32.         Console.Write(jel);
33.       Console.WriteLine();
34.     }
35.
36.     static void Main()
37.     {
38.       Console.WriteLine("Ez egy fontos figyelmeztetés!");
39.       Alahuzas('?');
40.       Console.WriteLine("Minden sora fontos!");
41.       Alahuzas('~');
42.       Console.WriteLine("Komolyan!");
43.       Alahuzas('*');
44.     }
45.   }
46.  }
```

karakter típusú **paraméter.**

A **paraméterre** hivatkozhatunk az eljárásban.

A Console.WriteLine() is eljárás. (19-féle **paraméterezése** van.)

Az eljárást különböző **argumentum**okkal hívjuk meg. Mind char (1-féle paraméter)

Programunk ilyenkor a következőket csinálja:

- Amikor az eljárás neve után talál valamit a zárójelben (pl.: 16. sor), azt átadja az eljárásnak, ellenőrzi, hogy az adott helyen lévő paraméter típusának megfelel-e a kapott adat. (6. 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 (9. sor).

Függvényt írunk

A függvény ugyanúgy a program – pontosabban a `class` – 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 `return` a általában függvény definíciójának az utolsó utasítása, feladata az, hogy átadja az eredményt a függvény meghívójának és felszabadítsa a működése közben lefoglalt memóriát.

A hasonlóság nem véletlen, mivel a C# - és sok más programozási nyelvben – az eljárást speciális függvénynek tekintik. A C# nyelv specialitása, hogy a kódolás megkönnyítése érdekében, az eljárások végén nem kell kiírni a `return`-t. A matematikai függvényekhez képest, a programozásban használt függvények nem csak az eredmény kiszámítására képesek, a függvény definícióban az eredmény kiszámításán túl bármilyen programrészlet lehet.

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

Eljárás:

```
static void Pluszhat(int szam)
{
   Console.WriteLine(szam + 6);
}
```

Felhasználás:

```
Console.Write("4 + 6 =");
Pluszhat(4);
Console.Write("5 + 6 =");
Pluszhat(5);
```

Függvény:

```
static int Pluszhat(int szam)
{
   return (szam + 6);
}
```

Felhasználás:

```
Console.WriteLine(
   "4 + 6 = {0}", Pluszhat(4));
Console.WriteLine(
   "5 + 6 = {0}", Pluszhat(5));
```

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 – rövid nevű – eljárás:

```
 6.  static void Ki(string duma)
 7.  {
 8.    Console.WriteLine(duma);
 9.  }
10.
```

Egész számot bekérő függvény:

```
11.  static int Be(string duma)
12.  {
13.    Console.Write(duma + " ");
14.    return int.Parse(
                 Console.ReadLine());
15.  }
```

Felhasználásuk például a 2 × 2 értékét bekérő programunkban:

```
17.  static void Main()
18.  {
19.    int valasz;
20.    do
21.    {
22.      valasz = Be("Mennyi kétszer kettő?"); //Be() függvény használata
23.    } while (valasz != 4);
24.    Ki("Annyi");                            //Ki() eljárás használata
25.  }
```

Mennyit nyerünk a `Ki`() eljárással? „`Console.WriteLine`” helyett „`Ki`” – ha nem használunk snippetet, akkor 15 karakterrel kevesebb, ráadásul ebben a jegyzetben eddig 60-szor szerepel a kódokban. A `Be`() függvény a „Console.Write;int.Parse(Console.ReadLine())” legalább 43 karakterét helyettesíti 2 karakterrel és a kódrészlet eddig 12-szer olvasható.

A program részekre bontásának szabályai

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.

```
 6.  static double osszeg (double egyik, double masik)
 7.  {
 8.  return egyik + masik;
 9.  }
10.
11.  static double osztas (double osztando, double oszto)
12.  {
13.  return osztando / oszto;
14.  }
     /*felhasználás*/
Console.WriteLine(Osszeg(3, 6)); //egyik-be behelyettesítjük a 3-at …
Console.WriteLine(Osszeg(6, 3)); //most a masik értéke lesz 3.
Console.WriteLine(Osztas(3, 6)); //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 fel tudjuk használni és tudjuk módosítani is.

Függvények és eljárások 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 függvényeken és eljárásokon kívül – pontosabban közöttük – deklarálunk (függetlenül attól, hogy kezdőértéket is adunk neki vagy sem), abban a `class`-ban amin belül van, használható. Ez az osztályra nézve globális változó. A Program osztályon belül ez elé is ki kell írni a `static` kulcsszót.

Az adatokat sok esetben nem csak felhasználni szeretnénk, hanem módosítani is. Amikor mód van rá, az adat módosításához függvényt írjunk, aminek a visszaadott értéke lesz az adatunk új értéke. Ilyenkor az sem jelent problémát, ha a függvény argumentuma és a módosított adat ugyanaz a változó. Például a korábban szereplő `Pluszhat`() függvény-t használva:

```
int n = 4;
n = Pluszhat(n); //ezután n értéke 10.
```

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 lehet 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.

Előfordulhat, hogy egy függvénnyel több adatot szeretnénk módosítani, azonban csak egy változónak tud a függvény értéket adni. Ilyen esetekre érdemes az alábbi szabályokat betartani:

- Azokat az adatok, amelyeket több függvény vagy eljárás használ és módosít is legyenek az osztályra nézve globális változók, mert ezeket az osztály minden függvénye és eljárása nem csak látja, de módosítani is tudja anélkül, hogy paraméterként megadnánk őket. Tipikusan ilyen globális adat az az adatsorozat, amellyel egy-egy feladaton belül foglalkozunk.
- A függvények és eljárások paramétereit az eljárás vagy a függvény belsejében ne módosítsuk. A módosítás egyes esetekben csak a paraméterre – a függvényen belül – lesz érvényes, más esetekben az eredeti adat fog módosulni.

Kihívást kedvelőknek a paraméter módosíthatóságának néhány szabálya:

- Azok az – eddig tanult – adattípusok, amelyeknél a változó létrehozásához kell a `new`, mind referencia típusok (reference type). A többi – jellemzően egyszerű – adattípus, aminél elég a változó név megadása és az értékadás, az érték típusok (value type).
  - Az érték típusú paraméterbe az argumentum másolata kerül. Ezért a függvényen/eljáráson belüli módosulás csak a függvényen/eljáráson belül, a másolaton fog végbe menni, az eredeti adat változatlan marad.
  - A referencia típusú paraméterbe az argumentumra történő hivatkozás kerül, így a módosítás a hivatkozáson keresztül az eredeti adatra fog vonatkozni.
- Az érték típusú adatot is át lehet adni referenciaként, de ezt a függvény definíciójában és a meghíváskor is jelezni kell. Ráadásul, a módosítás módjától függően kétféle megoldás van.
  - A `ref` módosítót használhatjuk akkor, ha a hivatkozott adatnak már van értéke, de módosítanánk azt. Tipikusan akkor használatos, ha feltétlenül több adat módosulna. Ilyen például két érték felcserélése. Ez a módosító arra is jó lehet, hogy nagyméretű `string` típusú adatnak ne a másolatát használja a függvényünk, ugyanis a `string` nem „egyszerű”, de érték típusú. Egy karaktersorozat sok memóriát lefoglalhat, az érték másolata ezt duplázza.
  - Az `out` módosítót akkor használhatjuk, ha a hivatkozott adatnak esetleg nincs értéke, jellemző, hogy az eljárásnak az a feladata, hogy értéket adjon a hivatkozott változóknak. Erre egyik példa az érték típusok `TryParse`() függvénye, aminek az eredménye igaz, ha sikerült az adatot konvertálni, de eközben a konverzió eredményét átadja a kimeneti paraméterének. Ezzel a módosítóval lehet – például – egy bennfoglaló osztás hányadosát és maradékát egyetlen függvénnyel meghatározni.

Példa a `ref` és `out` használatára:

```
 6.  static void Csere(ref int a, ref int b)
 7.  {
 8.    int c = a;
 9.    a = b;
10.    b = c;
11.  }
12.
13.  static int Hanyados(int a, int b, out int marad)
14.  {
15.    marad = a % b;
16.    return a / b;
17.  }
```

```
   /*felhasználás*/
int x = 2;
int y = 9;
Csere(ref x, ref y);
Console.WriteLine($"x = {x}, y = {y}"); // x = 9, y = 2
int m;
int h = Hanyados(x, y, out m);
Console.WriteLine($"{x} / {y} = {h}, marad: {m}"); // 9 / 2 = 4, marad 1
```

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:

```
6.  static void Akoba(double liter)
7.  {
8.    Console.WriteLine(liter / 58.6);
9.  }
      /*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:

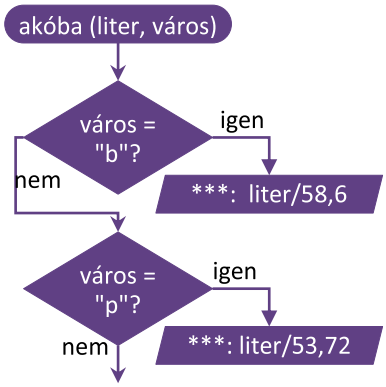
```
függvény akoba(liter):
    vissza: liter/58,6
függvény vége
```

Az átváltás függvény C# kód részlete:

```
6.  static double Akoba(double liter)
7.  {
8.    return liter / 58.6;
9.  }
      /*felhasználás*/
      Console.WriteLine(Akoba(999));
```

   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 …

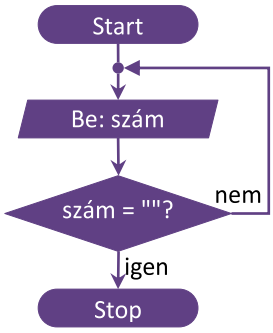
d) Módosítsuk az akóba váltó függvényünket úgy, hogy a város nem megfelelő értékére is legyen eredmény! Próbáljunk ki többféle megoldást!

```
 6.  static double Akoba (double liter, string varos )
 7.  {
 8.    if (varos == "b")
 9.      return liter / 58.6;
10.    else if (varos == "p")
11.      return liter / 53.72;
12.   /*return ?;*/
13.   }
```

```
    /*felhasználás*/
Console.WriteLine(Akoba(999, "d"));
```

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!

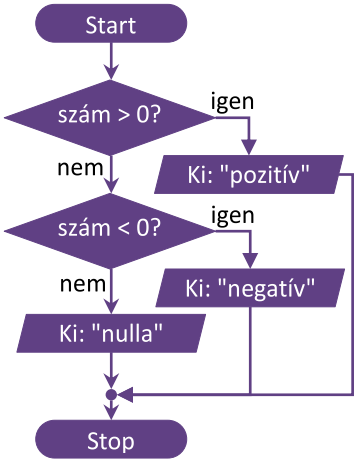
a) Először írjuk meg a főprogramot, azaz azt a részt, amely bekéri a számokat (de még ne csináljunk semmit a számmal)!




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

```
 6.  string szamstr;
 7.  do
 8.  {
 9.    Console.Write("Írj be egy egész számot: ");
10.    szamstr = Console.ReadLine();
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.

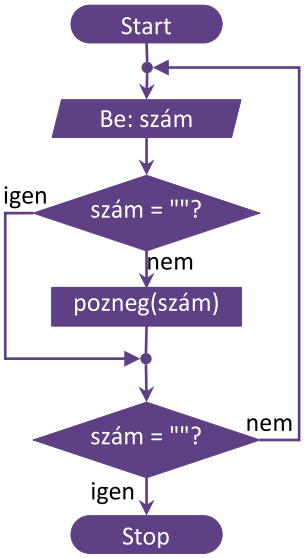



```
eljárás pozneg(szám)
    elágazás:
     ha szám >0:
       ki: szám, "pozitív"
     különben ha szám < 0:
       ki: szám, "negatív"
     különben:
       ki: "A szám nulla"
    elágazás vége
eljárás vége
```

Mivel a főprogramunk írását már elkezdtük, ezt kiegészítést írjuk a `Main`() függvény után. (Ez nem befolyásolja azt, hogy a program a `Main`() vége lesz a futtatott program vége.)

```
20.  static void PozNeg(int szam)
21.  {
22.    if (szam > 0)
23.      Console.WriteLine("{0} pozitív.", szam);
24.    else if (szam < 0)
25.      Console.WriteLine("{0} negatív.", szam);
26.    else
27.      Console.WriteLine("A szám nulla.");
28.  }
```

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.

```
1.  using System;
2.  namespace pozneg
3.  {
4.    class Program
5.    {
6.      static void Main()
7.      {
8.        string szamstr;
9.        do
10.       {
11.         Console.Write("Írj be egy egész számot: ");
12.         szamstr = Console.ReadLine();
13.         if (szamstr != "")
14.           PozNeg(int.Parse(szamstr)); //először Parse, utána PozNeg
15.       } while (szamstr != "");
16.     }
17.
18.     static void PozNeg(int szam) …
```

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.

Haladóknak: Próbáljuk ki az `int.TryParse()` függvény használatát! A 14. sor helyett:

```
int szamertek;
if (int.TryParse(szamstr, out szamertek))
   PozNeg(szamertek);
```

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.

```
 6.  static string Betukkel(int szam)
 7.  {
 8.    string[] szamnevek = new string[10] { "nulla", "egy", "kettő",
                "három", "négy", "öt", "hat", "hét", "nyolc", "kilenc" };
 9.    return szamnevek[szam];
10.  }

     /*felhasználás*/
   for (int i = 0; i < 10; i++)
     Console.WriteLine(Betukkel(i));
```

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:

```
 7.  static bool Lany(string nev)
 8.  {
 9.    bool benne = false;
10.    if (nev == "Anna" || nev == "Dóra" || nev == "Zita" /* … */)
11.      benne = true;
12.    return benne;
13.  }
14.
```

   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 felvenni a System.Collections.Generic névteret) 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:

```
15.  static bool Fiu(string nev)
16.  {
17.    List<string> nevek = new List<string>() { "Bence", "Endre", "Ferenc"
                                                   /*… stb …*/};
18.    int ez = 0;
19.    while (ez < nevek.Count && nev != nevek[ez]) //név, de nem ez a név
20.      ez++;                                      //akkor továbblép
21.    return ez < nevek.Count;     //ez a nevek egyikének az indexe    true
22.  }
23.
```

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

```
24.  static void Eztkapja(string nev)
25.  {
26.    /*eljárás eleje*/
27.    if (Lany(nev))
28.    {
29.      Console.WriteLine("Babát kap");
30.    }
31.    /*eljárás folytatása*/
32.  }
```

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 a `Program` globális változójaként 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?

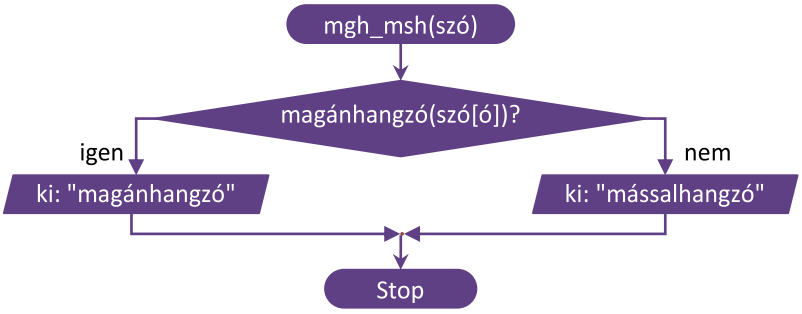
c) Írjuk meg a városokat bekérő eljárást, ami üres bemenetig fogadja a városok nevét! A főprogram a városok bevitele után – ellenőrzésképpen – írja ki a lista elemeit egymás mellé, vesszővel és szóközzel elválasztva. Ezt követően írja ki a fővárosok számát.

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?

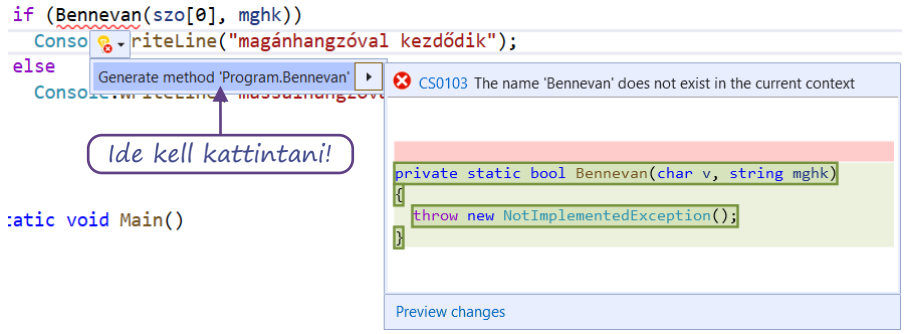



```
eljárás mgh_msh(szó):
    maganhangzók = "aáeéiíoóöőuúüű"
    ha magánhangzó(szó[0]):
     ki: "magánhangzó"
    különben:
     ki: "mássalhangzó"
eljárás vége
```

Az eljárásban szükségünk van a `maganhangzo()` függvényre – vagy kell helyette egy megfelelő C# függvény (`.Contains()`). 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.

```
 6.  static void MghMsh(string szo)
 7.  {
 8.    string mghk = "aáeéiíoóöőuúüű";
 9.    if (Bennevan(szo[0], mghk))
10.      Console.WriteLine("magánhangzóval kezdődik");
11.    else
12.      Console.WriteLine("mássalhangzóval kezdődik");
13.  }
14.
```

Az eljárás írása során a Visual Studio hibát jelez, mert nem ismeri a Bennevan() függvényt. Ha az egérrel a rámutatunk a hibára, akkor felajánlja a hiba javítását, amit érdemes elfogadni:



A kapott kódrészletben a private jelölő nem árt, de nem is szükséges. A függvény paramétereit érdemes átnevezni, „beszédessé” tenni az általános használathoz. A függvény definícióját nyilván nem fogja kitalálni helyettünk a Visual Studio; a **throw**… azt jelzi, hogy ez a függvény még nincs megírva. Próbáljuk ki, mi történik, ha így futtatjuk a programot!

Ha nem használjuk a MghMsh()-t

```
25.  static void Main()
26.  {
27.
28.  }
```

A program hibamentesen lefut

Ha használjuk az MghMsh()-t:

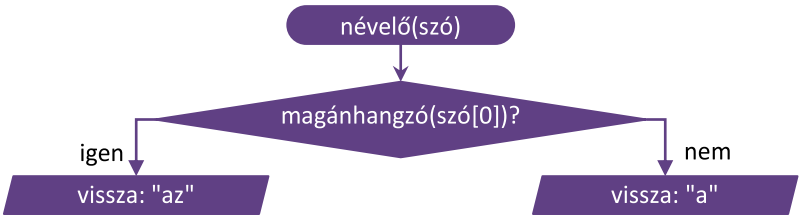
```
25.  static void Main()
26.  {
27.    MghMsh("alma");
28.  }
```

A program futása a Bennevan() függvénynél megakad, jelzi, hogy nincs még megírva.

Írjuk meg a Bennevan() függvény definícióját:

```
15.  static bool Bennevan(char betu, string minta)
16.  {
17.    int i = 0;
18.    while (i < minta.Length && betu != minta[i])
19.      i++;
20.    return i < minta.Length;
21.  }
```

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.



A magánhangzó(szó[0]) vizsgálatát a mondatszerű leírásban kifejtve látható, hogy lényegében az előző feladat Bennevan() függvénye jelenik meg a kódban.

```
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
```

Oldjuk meg most a feladatot másképp, használjuk a C# `string` típusának `Contains`() függvényét! Ebben az esetben arra kell figyelni, hogy a függvény paramétere `string`, de a szó első betűje `char` típusú, azaz karakter. Vagy a karaktert át kell alakítani `string`-re vagy kell keresni egy függvényt, ami a szónak ki tudja választani egy részét (mint táblázatkezelésben a BAL(), JOBB és KÖZÉP() függvények.)

```
 6.  static string NevElo(string szo)
 7.  {
 8.    string mghk = "aáeéiíoóöőuúüű";
 9.    if (mghk.Contains(szo[0].ToString()))
10.      return "az";
11.  else
12.      return "a";
13.  }
```

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.

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 három operandusú 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.  static string NevElo(string szo)
2.  {
3.    string mghk = "aáeéiíoóöőuúüű";
4.    return mghk.Contains(szo[0].ToString())? "az" : "a";
5.  }
```

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:

```
22.  static bool Bennevan(char betu, string minta)
23.  {
24.    int i = 0;
25.    while (i < minta.Length && betu != minta[i])
26.      i++;
27.    return i < minta.Length;
28.  }
29.
```

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 a kódban is egy sorba írjuk.

```
 6.  static string Hangrend(string szo)
 7.  {
 8.    string mely = "aáoóuú";
 9.    string magas = "eéiíöőüű";
10.    bool voltmely = false;
11.    bool voltmagas = false;
12.    foreach (char betu in szo)
13.    {
14.      if (Bennevan(betu, mely)) voltmely = true;
15.      else if (Bennevan(betu, magas)) voltmagas = true;
16.    }
17.    if (voltmely && !voltmagas) return "mély";
18.    else if (!voltmely && voltmagas) return "magas";
19.    else if (voltmely && voltmagas) return "vegyes";
20.    /*else – minden más esetben*/
21.    return "nincs magánhangzó a szóban";
22.  }
     /*felhasználása*/
Console.Write("Írj be egy szót: ");
string szo = Console.ReadLine();
Console.WriteLine(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!

```
 6.  static string Monogram(string nev)
 7.  {
 8.    string mg = nev[0] + ".";
 9.    for(int i = 0; i < nev.Length - 1; i++)
10.      if (nev[i] == ' ')
11.        mg += nev[i + 1] + ".";
12.    return mg;
13.  }
```

   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 vagy más lehet. Tanulmányozása a saját megoldás elkészítése után javasolt.

```
1.  using System;
2.  using System.Collections.Generic;
3.  namespace vogonVers
4.  {
5.    class Program
6.    {
7.      static Random Rand = new Random();
8.
```

```
    static string Verssor()
    {
      List<string>[] mondat = new List<string>[3] { /*3 List<> tömbje*/
       /*Listák létrehozása és feltöltése*/
      new List<string>{"Nyekergő", "Gömbölyű", "Piros", "Vidám", },
      new List<string>{"amőbát", "ablakokat", "sárgolyót", "kútkávát"},
      new List<string>{"dobálunk", "eszünk", "álmodunk", "éneklünk"}
      };  /*itt a tömb vége*/
      List<string> vege = new List<string> { ".", "?", "!", "?!" };
      string verssor = "";
      for (int i = 0; i < 3; i++)
       /*véletlen választások a listaelemek közül*/
       verssor += mondat[i][Rand.Next(mondat[i].Count)] + " ";
        /*2D indexelés: i-edik tömbelem véletlenedik listaeleme*/
      return verssor + '\b' + vege[Rand.Next(vege.Count)];
       /*backspace jel az utolsó szóköz törlésére*/
    }

    static string Versszak(int sorDB)
    {
      string vszak = "";
      for (int i = 0; i < sorDB; i++)
        vszak += Verssor() + '\n';
      return vszak;
    }

    static string Vers(int versszakDB, int sorDB)
    {
      string vers = "";
      for (int i = 0; i < versszakDB; i++)
        vers += Versszak(sorDB) + '\n';
      return vers;
    }

    static string Szabadvers(List<int> sorDB_lista)
    {
      string vers = "";
      foreach (int db in sorDB_lista) /*listát bejáró foreach-ciklus*/
        vers += Versszak(db) + '\n';
      return vers;
    }

    static void Main()
    {
      Console.WriteLine("Vers:\n" + Vers(3, 4));
      Console.WriteLine("Szabadvers:");
      Console.WriteLine(Szabadvers(new List<int> { 2, 2, 4 }));
       /*A paraméterben egy lista létrehozása és feltöltése*/
    }
  }
}
```

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 áttekintését segíti a `#region<leírás>` … `#endregion` utasításpár, amelyekkel elrejthetők lesznek az éppen nem vizsgált kódsorok. További rendszerezési lehetőség, hogy feladatonként vagy képernyőre írással, vagy a kódban kommenttel választjuk el egymástól a feladatokat.

A munkánkat könnyíti, ha az éppen nem használt kódrészleteket megjegyzéssé alakítjuk. Ehhez a `//` jelölést érdemes használni, amit a menüben elérhető `Comment` gombbal egyszerre minden kijelölt sor elé be lehet tenni, illetve az `Uncomment` gombbal lehet törölni. Emellett a megoldáshoz írt jegyzetekhez a `/**/` megjegyzési formát alkalmazzuk.

Végső soron az is megoldás lehet, hogy minden feladatnak külön programot írunk, de ezt később – 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 az `Edit/Advanced/Format Document` menüpont 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:

```
static int[] Bevetelek = new int[5] { 1, 5, 2, 3, 4 };
static int BeCount = 5;

static int[] Libak = new int[5] { 1, 5, 2, 3, 4 };
static int LCount = 5;
```

vagy `List<int>` listában (ne feledjük: `using` System.Collections.Generic;)

```
static List<int> Bevetelek = new List<int>(5) { 1, 5, 2, 3, 4 };

static List<int> Libak = new List<int>(5) { 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 < BeCount; i++) /*Listára Bevetelek.Count*/
3.    osszes += Bevetelek[i];
4.  Console.WriteLine("Napi bevétel {0} picula", osszes);
```

Ugyanez listán bejárós ciklussal:

```
1.  int osszes = 0;
2.  foreach (int ez in Bevetelek)
3.    osszes += ez;
4.  Console.WriteLine("Napi bevétel {0} picula", osszes);
```

Ugyanez feltételes (while-) ciklussal:

```
1.  int osszes = 0;
2.  int j = 0;
3.  while (j < BeCount) /*Listára Bevetelek.Count*/
4.  {
5.    osszes += Bevetelek[j];
6.    j++;
7.  }
8.  Console.WriteLine("Napi bevétel {0} picula", osszes);
9.
```

Kiegészítés: megoldás függvénnyel

Az algoritmust olyan gyakran használják, hogy nagyon sok programozási nyelven előregyártott függvény is megtalálható. A C# ezeket az adatsorozatokra alkalmazható függvényeket a System.Linq névtérben tartalmazza. (LINQ a Language Integrated Query rövidítése, a C# programozási nyelven belül egy adatsorozat elemzésére specializált nyelvi csomag.)

```
using System.Linq;
/*…*/
Console.WriteLine("Napi bevétel {0} picula", Bevetelek.Sum());
```

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 az elsőt átírva:

```
1.  int osszes = 0;
2.  for (int i = 0; i < LCount; i++)
3.    if(Libak[i] <= 3)
4.      osszes += Libak[i];
5.  Console.WriteLine("A rókának {0} kilogramm liba maradt", osszes);
```

Kiegészítés: megoldás függvénnyel

Bár ezt a feladatot nem lehet egyszerű összegzéssel megoldani – „bűvészkedni” kellene a LINQ-s megoldással –, 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 < LCount; i++)
3.    if(Libak[i] < 3 || Libak[i] == 5)
4.      osszes += Libak[i];
5.  Console.WriteLine("A rókának {0} kilogramm liba maradt", osszes);
```

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.

Átlagszámítás

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.

```
int osszes = 0;
for (int i = 0; i < Bevetelek.Count; i++) /*Tömbre az elemszámig*/
  osszes += Bevetelek[i];
int atlag = osszes / Bevetelek.Count; /*Tömbre az elemszámmal osztva*/
Console.WriteLine("Az átlagos bevétel {0} picula", osszes);
```

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;
foreach (int ez in Libak)
  if (ez <= 3)
  {
    osszes += ez;
    db++;
  }
double atlag = (double)osszes / db;
Console.WriteLine("A róka libái átlagosan {0} kilósak", atlag);
```

Kiegészítés: megoldás függvénnyel

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. Azonban egyszerű esetekre a LINQ `Average`() függvényét mi is tudjuk használni.

```
using System.Linq;
/*…*/
Console.WriteLine("Az átlagos bevétel {0} picula", Bevetelek.Average());
```

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.  static bool Van5()
2.  {
3.    bool vanilyen = false;
4.    for (int i = 0; i < Bevetelek.Count; i++) /*lehet másféle ciklus*/
5.      if (Bevetelek[i] == 5)
6.        vanilyen = true;
7.    return vanilyen;
8.  }
```

```
    /*felhasználás*/
if (Van5())
    Console.WriteLine("Volt 5 piculás bevétel.");
else
    Console.WriteLine("Nem volt 5 piculás bevétel.");
```

Az algoritmus és a kód 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.  static bool Van5()
2.  {
3.    bool vanilyen = false;
4.    for (int i = 0; i < Bevetelek.Count && !vanilyen; i++)
5.      if (Bevetelek[i] == 5)
6.        vanilyen = true;
7.    return vanilyen;
8.  }
```

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

```
1.  static bool Van5()
2.  {
3.    bool vanilyen = false;
4.    int i = 0;
5.    while (i < Bevetelek.Count && !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.  static bool Van5()
2.  {
3.    int i = 0;
4.    while (i < Bevetelek.Count && !(Bevetelek[i] == 5))
5.      i++;
6.    return i < Bevetelek.Count
7.    }
```

Eddig a feladatra láthattunk 4, 5 és 8 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 3 soros:

```
1.  static bool Van5()
2.  {
3.    int i;
4.    for (i = 0; i < Bevetelek.Count && Bevetelek[i] != 5); i++) ;
5.    return i < Bevetelek.Count;
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.  static bool Van5()
2.  {
3.    bool vanilyen = false;
4.    for (int i = 0; i < Bevetelek.Count; i++) /*lehet másféle ciklus*/
5.      if (Bevetelek[i] == 5)
6.      {
7.        vanilyen = true;
8.        break;
9.    return vanilyen;
10.  }
11.
```

```
   /*felhasználás*/
if (Van5())
   Console.WriteLine("Volt 5 piculás bevétel.");
else
   Console.WriteLine("Nem volt 5 piculás bevétel.");
```

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.  static bool Van5()
2.  {
3.    for (int i = 0; i < Bevetelek.Count; i++) /*lehet másféle ciklus*/
4.       if (Bevetelek[i] == 5)
5.         return true;
6.    return false;
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.  static bool Voltnagy()
2.  {
3.    bool vanilyen = false;
4.    int i = 0;
5.    while (i < Libak.Count && !vanilyen)
6.    {
7.      if (Libak[i] >= 5)
8.        vanilyen = true;
9.      i++;
10.   }
11.   return vanilyen;
12. }
```

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.  static bool Elozonelkisebb()
2.  {
3.    int i = 1;
4.    while (i < Libak.Count && !(Libak[i] < Libak[i - 1]))
5.      i++;
6.    return i < Libak.Count
7.    }
```

Kiegészítés: megoldás függvénnyel

A legegyszerűbb esetben, amikor csak arra vagyunk kíváncsiak, hogy egy adat megtalálható-e a listában, használhatjuk a LINQ megfelelő függvényét:

```
using System.Linq;
/*…*/
if (Bevetelek.Contains(5))
   Console.WriteLine("Volt 5 piculás bevétel.");
else
   Console.WriteLine("Nem volt 5 piculás bevétel.");
```

A függvény – nem véletlenül – ismerős, a `string` típusú adatoknál ugyanilyen nevű függvénnyel lehet megtudni, hogy egy adott részlet megtalálható-e a szövegben. Amikor használtuk, gondot okozott, hogy nekünk karaktert kellett keresni, de a függvény csak szöveg keresésére van felkészítve. Az egyéb adatsorozatokra használható LINQ-beli `Contains()` – ahogy a többi függvény is, ennél sokkal okosabb, de a helyes paraméterezéséhez – ahol megadhatjuk a feltételeinket – sokkal mélyebb programozási ismeretre van szükség.

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*/
    Console.WriteLine("Az ötpiculás fuvar sorszáma: {0}", HolVan5() + 1);
```

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.  static int Elsonagy
2.  {
3.    int i;
4.    for (i = 0; Liba[i] < 3; i++);
5.    return i;
6.  }
```

Ide kell a **pontosvessző**!

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.  static void Hol5()
2.  {
3.    int i = 0;
4.    while (i < Bevetelek.Count && ! (Bevetelek[i] == 5))
5.      i++;
6.    if (i < Bevetelek. Count)
7.      Console.WriteLine("Az 5 piculás fuvar sorszáma: {0}", i + 1);
8.    else
9.      Console.WriteLine("Nem volt 5 piculás fuvar.");
10. }
```

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.  static void Hol5()
2.  {
3.    int ez;
4.    for (ez = 0; ez < Bevetelek.Count; ez++) /*lehet másféle ciklus*/
5.      if (Bevetelek[ez] == 5)
6.        break; /*kiugrik, amikor „ez” az elem 5 értékű*/
7.    if (ez < Bevetelek.Count)
8.      Console.WriteLine("Az 5 piculás fuvar sorszáma: {0}", i + 1);
9.    else
10.     Console.WriteLine("Nem volt 5 piculás fuvar.");
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# `Lista<>` típusok – esetén ezt a `Count`, `string` típus esetén a `Length` tulajdonság adja meg. Tömböt használva a `Length` 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 (int i = 0; i < Libak.Count; i++)
3.    if (Libak[i] <= 3)
4.      db++;
5.  Console.WriteLine("A rókának {0} libája marad.", db);
```

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.

Kiegészítés: megoldás függvénnyel

A megszámlálásnak is van „előregyártott" függvénye a fejlett programozási nyelvekben és adatkezelő alkalmazásokban, a LINQ-ban ez a `Count`() függvény, ahol a zárójelbe kellene beírni a számbavétel feltételét. Ez nem azonos a **`Count`** tulajdonsággal, amelyiknek nincs zárójele.

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 `List<>` 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.  List<int> rokae = new List<int>();
3.  for (int i = 0; i < LCount; i++)
4.    if (Libak[i] <= 3)
5.    {
6.      rokae.Add(Libak[i]);
7.      db++;
8.    }
9.  Console.WriteLine("A rókának ez a {0} liba jutott.", db);
10. foreach (int liba in rokae)
11.   Console.Write("{0} kilós, ", liba);
12. Console.WriteLine("\b\b.");
```

Megjegyzések a kódhoz:

- Ha a kigyűjtést `List<>`-be végezzük, akkor a `db` változóra nincs szükség, mert a `List<>` `Count` tulajdonsága 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.

Kiegészítés: megoldás függvénnyel

Táblázatkezelőben a „mennyi” kérdésre válaszol a MAX() és MIN() függvény. De a „melyik” 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.

A táblázatkezelő függvényhez hasonló módon használhatjuk a LINQ Max() és Min() függvényét. Paraméter nélkül megkapjuk egy egyszerű adatokat tartalmazó adatsorozat legnagyobb értékét, haladóbb ismeretekkel a paraméterben megadható, hogy milyen szempont szerint lesz egy objektum legnagyobb.

```
using System.Linq;
/*…*/
Console.WriteLine("Az legnagyobb érték {0}", Bevetelek.Max());
```

Melyik és mennyi

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:

```
int legtobb = Bevetelek[0];
foreach (int ez in Bevetelek)
  if (ez > legtobb)
    legtobb = ez;
Console.WriteLine("A legdrágább fuvar {0} piculát ért.", legtobb);
```

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.

```
int legtobb = Bevetelek[0];
for (int i = 1; i < Bevetelek.Count; i++)
  if (Bevetelek[i] > legtobb)
    legtobb = Bevetelek[i];
Console.WriteLine("A legdrágább fuvar {0} piculát ért.", legtobb);
```

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]
```

```
int maxhely = 0;
for (int i = 1; i < Bevetelek.Count; i++)
  if (Bevetelek[i] > Bevetelek[maxhely])
    maxhely = i;
Console.WriteLine("A legdrágább fuvar az {0}.", maxhely);
Console.WriteLine("Ezért {0} piculát fizettek.", Bevetelek[maxhely]);
```

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.

```
static int Maxh()
{
  int maxhely = 0;
  for (int i = 1; i < Bevetelek.Count; i++)
    if (Bevetelek[i] > Bevetelek[maxhely])
      maxhely = i;
return maxhely
}
```

```
/*felhasználás*/
int maxindex = Maxh(); /*tároljuk, hogy ne futtassa 2-szer a fgv-t.*/
Console.WriteLine("A legdrágább fuvar az {0}.", maxindex);
Console.WriteLine("Ezért {0} piculát fizettek.", Bevetelek[maxindex]);
```

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;
foreach (int ez in Libak)
  if (ez > 3 && ez < farkasmin)
    farkasmin = ez;
if (farkasmin == 1000)
  Console.WriteLine("A farkas nem vitt el egy libát sem.");
else
  Console.WriteLine("A farkas legkisebb libája {0} kg.", farkasmin);
```

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.

```
static void FarkasLegkisebbLibaja()
{
  int i = 0;
  while (i < LiCount && !(Libak[i] > 3))
    i++;
  if (i == LCount) /*végére ért, nem volt nagy liba */
    Console.WriteLine("A farkas nem vitt el egy libát sem.");
  else /*az i-ediket a farkas elvitte, ezután …*/
  {
    int minid = i; /*… tovább nézzük, minimumot keresve*/
    while (i < LCount)
    {
      if (Libak[i] > 3 && Libak[i] < Libak[minid])
        minid = i;
      i++;
    }
    Console.Write("A farkas legkisebb libája {0} kg.", Libak[minid]);
  }
}
```

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, hosszát a `Length` tulajdonsága adja meg. A `string` típusú adatnak számos függvénye van, például a kisbetűsre alakító szoveg.`ToLower`(), a nagybetűsre alakító `ToUpper` vagy a szövegrész tartalmazását vizsgáló `Contains`, amelyek mindegyike a teljes szövegre vonatkoznak. Az egyes karakterekre a `char` típusnak vannak függvényei. A `char`.`ToUpper`() nagybetűsre alakít, a `char`.`IsUpper`() megmondja, hogy egy karakter (betű) nagybetű-e. Hasonlóan a `char`.`IsLower`() azt adja meg, hogy a karakter kisbetű-e, a `char`.`IsDigit`() azt, hogy számjegy-e… Ezeknek a karaktervizsgáló függvényeknek kétféle paramétere lehet: megadhatunk egy karaktert vagy egy szöveget és egy indexet – a kérdéses karakternek a szövegbéli sorszámát.

   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?

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

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 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 típus.`Parse(string)` függvényeket, ami bármilyen számtípusra létezik. Az `int` és `double` értékek közötti átalakítás *explicit casttal* lehetséges, például egész szám az `(int)3.4`.

   A példákban használt adatsorozat legyen a `Program` osztályon belül 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 – valamelyik 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 írásjel-e (`IsPunctuation`()). 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 `char.IsUpper(char)` vagy a `char.IsUpper(string, 0)`függvényt.

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 `List<double>` típusú adatsorozatot használtunk. Ennek megfelelően egyes megoldások bejárós ciklussal is megoldhatók. Minden megoldás egy függvény, aminek a neve a feladat betűjelével azonos. Az eredmény kiírása a főprogramban van.

```
1.  using System;
2.  using System.Collections.Generic;
3.  using System.Linq; /*pl. B()*/
4.
5.  namespace pincer
6.  {
7.    class Program
8.    {
9.      static List<double> Tarca = new List<double>()
                                    { 3, 8, 10, 19.35, -6, 5.1, 9, 20 };
10.
```

```
11.     static bool A()
12.     {
13.       foreach (double font in Tarca)
14.         if (font < 0)
15.           return true;
16.       return false;
17.     }
18.
19.     static double B()
20.     {/*hogy B() legyen a neve*/
21.       return Tarca.Sum();
22.     }
23.
24.     static int C()
25.     {
26.       int db = 0;
27.       foreach (double font in Tarca)
28.         if (font > 0 && font != (int)font)
29.           db++;
30.       return db;
31.     }
32.
33.     static int D()
34.     {
35.       double penny = 0;
36.       foreach (double font in Tarca)
37.         if (font > 0 && font != (int)font)
38.           penny += font - (int)font;
39.       return (int)(penny * 100);
40.     }
41.
42.     static int E()
43.     {
44.       int db = 0;
45.       foreach (double font in Tarca)
46.         if (font >= 5)
47.           db++;
48.       return db;
49.     }
50.
51.     static double F()
52.     {
53.       double maxe = Tarca[0];
54.       foreach (double font in Tarca)
55.         if (maxe < font)
56.           maxe = font;
57.       return maxe;
58.     }
59.
```

```
60.     static double G(double kezd, List<double> bevetel)
61.     {
62.       double veg = kezd;
63.       foreach (double font in bevetel)
64.         veg += font;
65.       return veg;
66.     }
67.
68.     static int H()
69.     {
70.       int ez = 0;
71.       while (Tarca[ez] != 9)
72.         ez++;
73.       return ez; /*0-tól számozva*/
74.     }
75.
76.     static string I()
77.     {
78.       int ez;
79.       for (ez = 0; ez < Tarca.Count && Tarca[ez] <= 10; ez++) ;
80.       string valasz;
81.       if (ez < Tarca.Count)
82.         valasz = "Az első tíz fontnál többet fizető vendég a(z) " +
                    (ez + 1).ToString() + ". vendég.";
83.       else
84.         valasz = "Minden vendég 10 fontnál kevesebbet fizetett.";
85.       return valasz;
86.     }
87.
88.     static string J()
89.     {
90.       int ez;
91.       for (ez = Tarca.Count - 1; ez >= 0 && Tarca[ez] <= 10; ez--) ;
92.       string valasz;
93.       if (ez >= 0)
94.         valasz = "Az utolsó tíz fontnál többet fizető vendég a(z) " +
                    (ez + 1).ToString() + ". vendég.";
95.       else
96.         valasz = "Minden vendég 10 fontnál kevesebbet fizetett.";
97.       return valasz;
98.     }
99.
100.    static bool K()
101.    {
102.      int i = 0;
103.      while (i < Tarca.Count && !(Tarca[i] == (int)Tarca[i] &&
                                      Tarca[i] > 0 && (int)Tarca[i] % 5 == 0))
104.        i++;
105.      if (i < Tarca.Count)
106.        return true;
107.      /*else*/
108.      return false;
109.    }
110.
```

```
111.     static double L()
112.     {
113.       int db = 0;
114.         foreach (double font in Tarca)
115.         if (font > 0)
116.           db++;
117.       return 0.5 * db;
118.     }
119.
120.     static void Main()
121.     {
122.       Console.WriteLine("Válaszok");
123.       Console.WriteLine("a)\t{0}", A() ?
                              "Vásárolt is." : "Csak fizettek neki.");
124.       Console.WriteLine("b)\tA pénztárcában {0} font van.", B());
125.       Console.WriteLine("c)\t{0} alkalommal kapott pennyt.", C());
126.       Console.WriteLine("d)\tA pincér {0} pennyt kapott.", D());
127.       Console.WriteLine("e)\t{0} esetben kapott min. 5 fontot.", E());
128.       Console.WriteLine("f)\tA legnagyobb számla {0} font.", F());
129.       Console.Write("Mennyi pénz volt a tárcában? ");
130.       double ind = double.Parse(Console.ReadLine());
131.       Console.WriteLine("g)\tMost {0} van benne.", G(ind, Tarca));
132.       Console.WriteLine("h)\tA {0}. vendég fizetett 9 fontot.", H()+1);
133.       Console.WriteLine("i)\t{0}", I());
134.       Console.WriteLine("j)\t{0}", J());
135.       Console.WriteLine("k)\t{0}", K() ?
          "Volt, aki ötösökkel tudott fizetni." :
          "Nem volt olyan vendég, aki csak ötfontosokkal tudott fizetni.");
136.       Console.WriteLine("l)\tA pincér {0} font fizetést kap.", L());
137.     }
138.   }
139. }
```

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.  using System;
2.  namespace Mondat
3.  {
4.    class Program
5.    {
6.      const int maxdb = 100;
7.      static string[] Mondat = new string[100];
8.
```

```
 9.     static int Szodb()
10.     {
11.       int db = 0;
12.       string[] pelda =
                 { "Én", "elmentem", "a", "vásárba", "fél", "arannyal." };
13.       /*beolvasásra cserélhető*/
14.       while (db < maxdb && db < pelda.Length)
15.       {
16.         Mondat[db] = pelda[db];
17.         db++;
18.       }
19.       Console.WriteLine("a)\tA mondat {0} szóból áll.", db);
20.       return db;
21.     }
22.
23.     static void B(int N)
24.     {
25.       int minhossz = Mondat[0].Length;
26.       /*Biztosan létezik. Ha nincs Mondat, akkor az értéke 0.*/
27.       for (int i = 1; i < N; i++)
28.         if (Mondat[i].Length < minhossz)
29.           minhossz = Mondat[i].Length;
30.       Console.WriteLine("b)\tA legrövidebb szó {0} karakter.",
                            minhossz);
31.     }
32.
33.     static bool Irasjel(char c)
34.     {
35.       /*return char.IsPunctuation(c); //helyette */
36.       char[] jelek = { '.', '?', '!', ',', ';' };
37.       int i;
38.       for (i = 0; i < jelek.Length && c != jelek[i]; i++)
39.       { } /*nincs ciklusmag*/
40.       return i < 5;
41.     }
42.
43.     static void C(int N)
44.     {
45.       int i = 0;
46.       while (i < N && !Irasjel(Mondat[i][Mondat[i].Length - 1]))
47.         i++;
48.       if (i < N)
49.         Console.WriteLine("c)\tVan olyan szó, ami után írásjel áll.");
50.       else
51.         Console.WriteLine("c)\tNincs olyan szó, ami után írásjel áll.");
52.     }
53.
54.     static void D(int N)
55.     {
56.       int db = 0;
57.       for (int i = 0; i < N; i++)
58.         if (Mondat[i] == "a" || Mondat[i] == "az" ||
              Mondat[i] == "A" || Mondat[i] == "Az")
59.           db++;
60.       Console.Write("d)\tA mondatban {0} határozott névelő van.\n", db);
61.     }
62.
```

```
63.     static void E(int N, string szo)
64.     {
65.       int i;
66.       for (i = 0; i < N && Mondat[i] != szo; i++)
67.         ; /*nincs ciklusmag*/
68.       if (i < N)
69.         Console.WriteLine("e)\tA mondatban a(z) \"{0}\" szó a(z) {1}.
                              helyen áll.", szo, i + 1);
70.       else Console.WriteLine("e)");
71.     }
72.
73.     static void F(int N)
74.     {
75.       int i = 0;
76.       while (i < N && !char.IsUpper(Mondat[i][0]))
77.         i++;
78.       if (i < N)
79.         Console.WriteLine("f)\tA(z) {0}. szó kezdődik nagybetűvel.",
                              i + 1);
80.       else
81.         Console.WriteLine("f)\tNincs a mondatban nagy kezdőbetűs szó.");
82.     }
83.
84.     static void Main()
85.     {
86.       Console.WriteLine("Megoldások:");
87.       int N = Szodb();
88.       B(N);
89.       C(N);
90.       D(N);
91.       E(N, "fél");
92.       F(N);
93.     }
94.   }
95. }
```

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, a képernyő (grafikus és konzolos megjelenítők) és még sorolhatnánk … a táblákat, a táblázatokat. A kétdimenziós adatszerkezetek használata nagyon gyakori – pont úgy, ahogy a táblázatoké a valóságban.

Nézzük meg, hogyan lehet kétdimenziós adatszerkezetet létrehozni a már megismert egydimenziós adatsorozatokból! Egydimenziós adatsorozatból ismerjük az (egydimenziós) tömböt, a `List<>`-et és karaktersorozatként a `string`-et. A `string` elemei csak karakterek lehetnek, de a tömb és `List<>` eleme bármi lehet. Így elméletileg lehet egy tömb minden eleme tömb vagy `List<>` vagy `string`; lehet, egy `List<>` minden eleme `List<>` vagy tömb vagy `string`. Ez összesen hatféle kétdimenziós struktúra. De hiszen ebből kettőt már használtunk is! Tároltunk

`string`eket tömbben és `List<>`-ben, például a mondatelemzős feladatban ki ezt, ki azt, ki mindkettőt használhatta. Ott az `i`-edik szó utolsó karaktere:

```
Mondat[i][Mondat[i].Length - 1]
```

Az egyik dimenzió egy tömb vagy lista, aminek `Mondat[i]` az eleme. A `Mondat[i]` típusa `string`, azaz karaktersorozat. Ezért a teljes `Mondat` értelmezhető karaktersorozatként, amely szavanként van sorokra tördelve. Konzolképernyőre kiírva és a két indexe megadja, hogy egy karakter hol látható.

Ne számítsunk csodára, a másik négy variáció is hasonló, csak a létrehozásukban és feltöltésükben van eltérés.

- Tömbben tömbök
- Tömbben listák
- Listában tömbök
- Listában listák

A fenti négy kétdimenziós adatszerkezet minden olyan programozási nyelvben megtalálható, amelyben van tömb és van lista is. Ilyen például a C#, a C++, a Java.

Egyes nyelvek nem tartalmaznak tömböt – például a Python – Ezekben a nyelvekben a kétdimenziós adatszerkezetre az egyetlen megoldás a listában listák. Más nyelvekben ismeretlen a lista – például C –, itt csak a tömbben tömbökkel lehet megoldani a kétdimenziós adattárolást.

Azt már láthattuk, hogy a tömb és a lista között az a különbség, hogy a tömb mérete fix – ezért gyorsan létrehozható, de később nem bővíthető, ezzel szemben a lista folyamatosan bővíthető, de a bővítés időigényes. Azzal, hogy egy nyelvben a tömb is és a lista is használható, a feladat jellegéhez illeszkedően tudunk választani. Például az ötöslottó heti sorsolásait minden hétre 1-1 ötelemű tömbben, de hétről hétre új sorral bővülő listában érdemes tárolni.

Azonban amikor szorzótáblárol, sakktábláról vagy táblázatról beszélünk, nem pont ilyennek képzeljük az adatstruktúrát, hanem olyannak, aminek egységes a szélessége és a magassága is. Néhány programozási nyelvben – C# Visual Basic, Pascal és a mondatszerű leírásban – erre is van nyelvi megoldás.

Az igazi kétdimenziós adatstruktúra a tömb továbbfejlesztésének tekinthető, amelyben 1-1 elemére két adattal, a sor- és oszlopindexszel hivatkozhatunk. Külön neve is van a szaknyelvben: **mátrix**. Ráadásul, amelyik nyelvben van (kétdimenziós) mátrix, abban van több-dimenziós is, így a tér leírására háromdimenziós mátrix, a téridő leírására 4D mátrix… C#-ban egészen a 32-dimenziós mátrixig, ami összetettségében talán meg is felelne az azonos nevű filmben vizionált rendszernek.

Mintafeladat

A 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. A mátrixunk – táblázatunk – egy-egy sora egy-egy zenész jelenlétéről szól, ezért négy sor lesz a mátrixban; az oszlopok a hét munkanapjainak felelnek meg, így öt oszlopunk lesz.

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.

A feladatot először mátrixszal oldjuk meg majd kiegészítésként a másik négy adatszerkezettel is lesz részlet a megoldásból.

A 2D mátrix

A 2D mátrix a tömb értelemszerű kiterjesztése. Úgy is mondhatjuk, hogy a tömb az 1D mátrix. Programozás szempontjából az szokott nehézséget okozni, ha valaki összekeveri a sort és az oszlopot. A mátrix felépítése olyan, mintha egy hosszú tömböt sorokra tördelnénk. Egy adatelem helyének az első adata a sor száma, a második az oszlopé. Így értelmezve, a sorban egymás mellett lévő adatok a memóriában is egymás mellett vannak, míg az egymás alattiak egy sornyi távolságra. Úgy mondjuk, hogy a mátrixban az adatokat sorfolytonosan tároljuk.

Sajnos a sorok és oszlopok felcserélésére sok gyakorlati alkalmazás még rá is erősít. Ezek közül a két leginkább zavaró, a táblázatkezelésben használt „A1” hivatkozás, ami pont a fordítottja a programozásban használt sorrendnek és a koordinátarendszerben használt (x, y), ami nem csak fordítva van, de ráadásul az y értéket felfelé növeljük, míg a sorok számozása lefelé növekszik. Ezért a mátrixszal – igazából minden kétdimenziós adatstruktúrával – történő feladatmegoldáshoz célszerű puskát készíteni, lerajzolni a táblázatot és beleírni, hogy melyik adatát hogyan jelöljük.

A feladat megoldása előtt nézzük át a mátrix létrehozásának, adatokkal feltöltésének és képernyőre írásának néhány lehetőségét.

Egy függvényen belül a mátrix létrehozása, feltöltése adatokkal és a mátrix kiírása képernyőre így írható:

```
1.  int sorDb = 4; /*bekérhető a felhasználótól*/
2.  int oszlopDb = 5; /*bekérhető a felhasználótól*/
3.  /*megnevezés és létrehozás*/
4.  int[,] matrix = new int[sorDb, oszlopDb];
```

```
/*feltöltés*/
for (int i = 0; i < sorDb; i++)
{
  /*soronként, szóközökkel tagolt adatok bekérése*/
  string[] sor = Console.ReadLine().Split(' ');
  for (int j = 0; j < oszlopDb; j++)
  {
    /*adatonként értékadás (módosítás)*/
    matrix[i, j] = int.Parse(sor[j]);
  }
}
/*kiírás*/
for (int i = 0; i < sorDb; i++)
{
  for (int j = 0; j < oszlopDb; j++)
  {
    Console.Write("{0} ",matrix[i,j]);
       }
       Console.WriteLine(); /*sorvége, újsor*/
     }

```

A mátrix létrehozásakor az adatok helye elkészül, érték típusú adatoknak lesz értéke is (`int` esetén 0), de a referencia típusú adatok esetén a hivatkozás a semmire (`null` érték) mutat.

```
int[,] DefinialtMatrix = new int[4, 5];
Feltolt(DefinialtMatrix);
Kiir(DefinialtMatrix)
```

A `Feltolt`() eljárás a paraméterében kapott mátrixot nem módosítja, csak a mátrixban tárolt adatokat:

```
static void Feltolt(int[,] matrix)
{
  int sorDb = matrix.GetLength(0);
  int oszlopDb = matrix.GetLength(1);
  matrix = new int[sorDb, oszlopDb];
  for (int i = 0; i < sorDb; i++)
  {
    string[] sor = Console.ReadLine().Split(' ');
    for (int j = 0; j < oszlopDb; j++)
    {
      matrix[i, j] = int.Parse(sor[j]);
    }
  }
}
```

Van, hogy kezdetben nem tudjuk, mekkora legyen a mátrix mérete. Ilyenkor megoldás, hogy a mátrixot először csak deklaráljuk, megnevezzük

```
int[,] DeklaraltMatrix; /*csak megnevezés*/
Letrehoz(out DeklaraltMatrix);
Kiir(DeklaraltMatrix);
```

Ha egy eljáráson belül hozzuk létre és töltjük fel adatokkal a mátrixot, akkor az eljárás paraméteréhez az out jelzőt is meg kell adni, hiszen létrehozunk a paraméteren keresztül:

```
1.  static void Letrehoz(out int[,] matrix)
2.  {
3.    int sorDb = int.Parse(Console.ReadLine());
4.    int oszlopDb = int.Parse(Console.ReadLine());
5.    matrix = new int[sorDb, oszlopDb]; /*létrehozás 0 értékekkel*/
6.    for (int i = 0; i < sorDb; i++)
7.    {
8.      string[] sor = Console.ReadLine().Split(' ');
9.      for (int j = 0; j < oszlopDb; j++)
10.     {
11.       matrix[i, j] = int.Parse(sor[j]);
12.     }
13.   }
14. }
```

Megjegyzés: Ha nem eljárást, hanem függvényt írunk akkor a `DeklaraltMatrix` létrehozása a 8. sorban látható a `string`[] `sor` létrehozásához lesz hasonló, a `Letrehoz` függvénynek nem kell paraméter, de az 5. sor elején kell az `int`[,] és a 14. sorban lesz a `return matrix`;.

A feltöltés eredményének megtekintéséhez meg kell írni a `Kiir`(`int`[,] `matrix`) eljárást. A kiíró eljárásunk paramétere a mátrix, amelynek a mérete, a `Length` tulajdonsága az adatok számát adja meg. A mátrix egyes dimenzióinak a méretét a `GetLength`() függvénnyel tudjuk lekérdezni, a megadás sorrendjében, 0-tól indexelve.

```
1.  static void Kiir(int[,] matrix)
2.  {
3.    int N = matrix.GetLength(0); /*magasság, index: i*/
4.    int M = matrix.GetLength(1); /*szélesség, index: j*/
5.    for (int i = 0; i < N; i++)
6.    {
7.      for (int j = 0; j < M; j++)
8.      {
9.        Console.Write("{0} ", matrix[i, j]);
10.     }
11.     Console.WriteLine();
12.   }
13. }
```

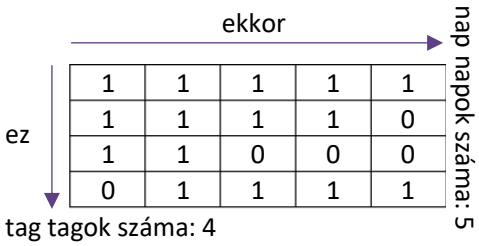
A `GetLength`() függvény, ha ciklusfeltételbe írnánk, minden ciklislépésben újra és újra lefutna, kiszámolná az értéket. Ez idő- és energia-pazarló lehet, ezért csak egyszer futtatjuk le a függvényt és eltároljuk az eredményt.

Megoldás

A feladat megoldásához – hogy ne kelljen bekérni az adatokat és a függvényekhez se kelljen paraméter – a jelenléti ívet a `Program` osztályban globális adatként adjuk meg. Mivel a `Program` osztályban minden adat és függvény `static` (azaz csak egyetlen példányban létezhet), a mátrixunk méretét sem lehet később felülbírálni, mert az egy másik mátrix lenne. Emiatt a méretének vagy konkrét számokat írunk be, vagy a változóknak konstansoknak kell lennie. A `const` jelző azt jelenti, hogy `static` és nem módosulhat.

```
15.  const int tag = 4;
16.  const int nap = 5;
17.  static int[,] Iv = new int[4, 5] {
18.                       { 1, 1, 1, 1, 1 },
19.                       { 1, 1, 1, 1, 0 },
20.                       { 1, 1, 0, 0, 0 },
21.                       { 0, 1, 1, 1, 1 } };
22.
```

A táblázatot ilyennek képzeljük:



Szebb lenne „ez” helyett „ő”, de esetleg gubancot okoz fordításkor az ékezetes karakter. Az „o” viszont a fejünkben okozhat gubancot, ha 0-nak nézzük.

A megoldásunkat – kiegészítve az adatok kiírásával – a főprogramban körülbelül képzeljük el:

```
1.  using System;
2.  namespace zeneszek
3.  {
4.    class Program
5.    {
6.      static void Main()
7.      {
8.        Kiir(Iv);
9.        Console.WriteLine("Összesen {0} adagot kell fizetni pénteken.",
                            EbedDb());
10.       Console.WriteLine("{0} volt a legtöbbet jelen.", LegtobbJelen());
11.       Console.WriteLine(Mindig() ?
                            "Volt, aki egyszer sem hiányzott.":
                            "Senki sem volt mindig jelen.");
12.       Console.WriteLine(Mindenki() ?
                            "Volt, amikor mindenki jelen volt." :
                            "Mindig hiányzott valaki");
13.     }
14.
```

Ezt a kódot akár előre be is írhatjuk a `Main`()-be és a Visual Studiora bízhatjuk a függvények deklarálását.

1. … Hány adagot fizetnek ezen a pénteken?

```
23.  static int EbedDb()
24.  {
25.    int db = 0;
26.    for (int ez = 0; ez < tag; ez++)
27.      for (int ekkor = 0; ekkor < nap; ekkor++)
28.        if (Iv[ez, ekkor] == 1)
29.          db++;
30.    return db;
31.  }
32.
```

2. Melyik zenész volt a legtöbbet jelen a héten?

```
33.  static string[] Nev = new string[tag]
                  { "Heg Edu", "Vio Lina", "Brá Csaba", "Csel Lotti" };
34.
35.  static string LegtobbJelen()
36.  {
37.    int[] jelen = new int[tag];
38.    for (int i = 0; i < tag; i++)
39.    {
40.      jelen[i] = 0; /*soronként megszámlálás*/
41.      for (int ekkor = 0; ekkor < nap; ekkor++)
42.        if (Iv[i, ekkor] == 1)
43.          jelen[i]++;
44.    }
45.    int maxi = 0; /*maximumkiválasztás tétele*/
46.    for (int i = 1; i < tag; i++)
47.      if (jelen[i] > jelen[maxi])
48.        maxi = i;
49.    return Nev[maxi];
50.  }
51.
```

3. Volt-e olyan zenész, aki mindig jelen volt?

```
52.  static bool Mindig()
53.  {
54.    bool mindig = false; /*eldöntés a személyre*/
55.    int ez = 0;
56.    while (ez < tag && !mindig)
57.    {
58.      int jelen = 0; /*soronként, azaz a jelenlétre eldöntés*/
59.      while (jelen < nap && Iv[ez, jelen] == 1)
60.        jelen++;
61.      if (jelen == nap)
62.        mindig = true;
63.      ez++;
64.    }
65.    return mindig;
66.  }
67.
```

4. Írjuk ki, ha volt olyan nap, amikor mindenki jelen volt a próbán!

```
68.  static bool Mindenki()
69.  {
70.    bool mindenki = false;
71.    for (int ekkor = 0; ekkor < nap && !mindenki; ekkor++)
72.    {
73.      int jelen;
74.      for (jelen = 0; jelen < tag && Iv[jelen, ekkor] == 1; jelen++)
75.        ;
76.      if (jelen == tag)
77.        mindenki = true;
78.    }
79.    return mindenki;
80.  }
81.  /*Kiir fgv…*/
```

Kitekintés: Kétdimenzió két struktúrával

Nagyon sok programozási nyelvben nincs mátrix, de a többdimenziós adattárolást ott sem lehet elkerülni. Ezért nézzük meg, „milyen lenne az élet mátrix nélkül”! Hogyan lehet létrehozni, feltölteni, kiírni egy táblázat adatait mátrix adattípus nélkül? Gondoljuk át:

- Mit csinálnak azok, akik nem C# nyelven tanulnak programozni?
- A mátrix előre megadott mérettel tud dolgozni. Milyen lehetőségeink vannak tetszőlegesen nyújtható, bővíthető táblázat létrehozására?

A vizsgálat eredményére tanulmányaink során várhatóan nem lesz szükségünk. De elvégzése azért nem haszontalan, mert módot ad a tömb és lista használatának alaposabb ismeretére. A következőkben a 2D mátrix fejezetben látható feladatokra adunk más megoldásokat, ezért a kódok összehasonlítására is van mód.

Tömbben tárolt tömbök

Ez az adatszerkezet hasonlít leginkább a mátrixhoz, de macerásabb a létrehozása, azonban van egy speciális tulajdonsága: minden sornak egyedileg megadható a hossza. Az ilyen kétdimenziós struktúrák „szaggatott szélűek”, a tömbös kivitelezés angol neve Jagged Array.

```
1.  int sorDb = 4;
2.  int oszlopDb = 5; /*most állandó de soronként változhat*/
3.  /*megnevezés és létrehozás*/
4.  int[][] TT = new int[sorDb][];
5.  for (int i = 0; i < sorDb; i++)
6.  {
7.    TT[i] = new int[oszlopDb]; /*soronként kell létrehozni*/
8.  }
9.  /*feltöltés*/
10. for (int i = 0; i < sorDb; i++) /*vagy … < TT.Length*/
11. {
12.   /*soronként, szóközökkel tagolt adatok bekérése*/
13.   string[] sor = Console.ReadLine().Split(' ');
14.   for (int j = 0; j < oszlopDb; j++) /*vagy: … < TT[i].Length*/
15.   {
16.     /*adatonként értékadás (módosítás)*/
17.     TT[i][j] = int.Parse(sor[j]);
18.   }
19. }
```

```
/*kiírás*/
for (int i = 0; i < TT.Length; i++)
{
  for (int j = 0; j < TT[i].Length; j++)
  {
    Console.Write("{0} ",TT[i][j]);
       }
       Console.WriteLine(); /*sorvége, újsor*/
     }

```

Arra kell figyelni – a mátrixhoz képest –, hogy

- a soroknak először csak a hivatkozásuk van meg, mindegyiket külön létre kell hozni;
- a sorok számát és a soron belül az elemek számát a megfelelő adat `Length` tulajdonságából tudjuk;
- az elemre hivatkozás két szögletes zárójelpárt igényel (nem lehet vesszővel felsorolni).

Listában tárolt listák

Ez a méretezésben legkötetlenebb kétdimenziós táblázat. Létrehozáskor minden irányban 0 méretű, nincs „kiterjedése”. A mérete az adatok feltöltésével alakul ki. A tömbös megoldáshoz képest

- bonyolultabb az adattípus létrehozása, mert nem hozzátoldás, hanem belső bővítéssel alakul ki a második dimenzió;
- a listának a hosszát a `Count` tulajdonság adja meg;
- a bejáró foreach-ciklus használata is ajánlott.

```
/*megnevezés és 0 db sor 0 db adat létrehozása*/
List<List<int>> LL = new List<List<int>>();
/*létrehozás feltöltéssel*/
for (int i = 0; i < 3; i++) /*3 sor létrehozása*/
{
  LL.Add(new List<int>()); /*üres i-edik sor*/
  string[] adatok = Console.ReadLine().Split(' ');
  for (int j = 0; j < adatok.Length; j++)
                /*csak az adatok hossza ismert*/
  {
    LL[i].Add(int.Parse(adatok[j]));
  }
}
/*kiírás számlálós ciklussal*/
for (int i = 0; i < LL.Count; i++)
{
  for (int j = 0; j < LL[i].Count; j++)
  {
    Console.Write("{0} ", LL[i][j]);
  }
  Console.WriteLine(); /*sorvége, újsor*/
}
```

```
/*kiírás bejáró ciklussal*/
foreach (List<int> sor in LL)
{
  foreach (int adat in sor)
  {
    Console.Write("{0} ", adat);
  }
  Console.WriteLine();
}
```

Tömbben tárolt listák

Ezt a konstrukciót csak azokon a nyelveken lehet megvalósítani, amelyekben van tömb is és lista is. Ahol csak az egyik adatsorozat használatos, ott annak a létrehozása nagyon egyszerű. Ezért például Pythonban a „listában listák” adatstruktúra egyáltalán nem bonyolult, mert minden adatsorozat „magától” lista lesz. Ahol több konstrukció van adatsorozatra, ott törvényszerű, hogy az adattípus megadása nehézkesebb legyen, mert meg kell különböztetni a típusokat. A vegyes használat során pedig a hasonlóságok ellenére is pontosan kell „éreznie” a programozónak, hogy éppen melyik típust használja.

Egy-egy feladatot sokféle adatstruktúrával meg lehet oldani, de van, amikor éppen egy tömbben tárolt lista a legkifejezőbb. Például, amikor ismerjük, hogy egy osztály tanulói közül ki hányast kapott és ki szeretnénk gyűjteni jegyenként a neveket. Hasonló feladat a fakultációs névsorok készítése is. A közös jellemző az, hogy pontos korlátja van a listák számának, ezért a listákat előre rögzített méretű tömbben tároljuk. Másrészt, az egyes listákban akárhány elem lehetséges, rögzített méret esetén úgy kellene tervezni, mintha minden listába minden lehetséges adat bekerülhetne, de így a lefoglalt méret a szükségesnek sokszorosa lenne.

```
/*megnevezés és 3 sor előkészítése*/
List<int>[] TL = new List<int>[3];
```

Azaz: `List<int>` típusú adatokat tartalmazó három elemű tömb. A feltöltésnél figyeljünk arra, hogy a tömbelemek csak deklarációi a listáknak, a listákat egyenként létre kell hozni.

```
/*létrehozás feltöltéssel*/
for (int i = 0; i < TL.Length; i++) /*3 sor létrehozása*/
{
  TL[i] = new List<int>(); /*üres i-edik sor létrehozása*/
  string[] adatok = Console.ReadLine().Split(' ');
  for (int j = 0; j < adatok.Length; j++)
                /*csak az adatok hossza ismert*/
  {
    TL[i].Add(int.Parse(adatok[j]));
  }
}
```

Ahogy feltöltjük adatokkal, később ugyanúgy lehet a sorokon belül bővíteni, bármelyik `TL[i]` listát. Ráadásul lehet törölni is belőlük, így mindig a szükséges mennyiségű adatra lesz memória lefoglalva.

Kiíráskor figyelni kell arra, hogy `Count` vagy `Length` – és aminek `Count`-ja van, azt lehet bejárni foreach-ciklussal is.

```
13.  /*kiírás számlálós ciklussal*/
14.  for (int i = 0; i < TL.Length; i++)
15.  {
16.    for (int j = 0; j < TL[i].Count; j++)
17.    {
18.      Console.Write("{0} ", TL[i][j]);
19.    }
20.    Console.WriteLine(); /*sorvége, újsor*/
21.  }
22.  /*kiírás bejáró ciklussal*/
23.  for (int i = 0; i < TL.Length; i++)
24.  {
25.    foreach (int adat in TL[i])
26.    {
27.      Console.Write("{0} ", adat);
28.    }
29.    Console.WriteLine();
30.  }
```

Listában tömbök

Volt már arról szó, hogy ennek a struktúrának is van létjogosultsága. Például a lottó sorsolások hosszútávú feljegyzésekor, vagy a tanév folyamán az ebédlő ügyeletes párok feljegyzésekor. Az nem jelent problémát, ha a lottószámok listájában néha 5-ös máskor a 6-os lottó számait tároljuk és az ügyeletesek is lehetnek néha többen, máskor kevesebben. Az adatstruktúra szempontjából csak az számít, hogy egy-egy listaelemről – ami tömb – a létrehozáskor pontosan tudjuk, hogy hány elemet tartalmazhat, mert később ez nem bővíthető.

```
1.   /*megnevezés és lista előkészítése 0 sorral*/
2.   List<int[]> LT = new List<int[]>();
3.   /*létrehozás feltöltéssel*/
4.   for (int i = 0; i < 3; i++) /*3 sor létrehozása*/
5.   {
6.     LT.Add(new int[5]); /*5 elemű i-edik sor: 0 0 0 0 0*/
7.     string[] adatok = Console.ReadLine().Split(' ');
8.     for (int j = 0; j < adatok.Length && j < 5; j++)
                  /*valahány adat, de max. 5*/
9.     {
10.      LT[i][j] = int.Parse(adatok[j]);
11.    }
12.  }
13.  /*kiírás számlálós ciklussal*/
14.  for (int i = 0; i < LT.Count; i++)
15.  {
16.    for (int j = 0; j < LT[i].Length; j++)
17.    {
18.      Console.Write("{0} ", LT[i][j]);
19.    }
20.    Console.WriteLine(); /*sorvége, újsor*/
21.  }
```

```
22.  /*kiírás bejáró ciklussal*/
23.  foreach (int[] sor in LT)
24.  {
25.    for (int j = 0; j < sor.Length; j++)
26.    {
27.      Console.Write("{0} ", sor[j]);
28.    }
29.    Console.WriteLine();
30.  }
```

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 (szövegként a 9 jóval „nagyobb", mint a „16"). Másrészt számítást igénylő feladatoknál 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 szöveges 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 adattagoknak, 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 `Length` tulajdonsága, a `List<> Add()` eljárása – és sorolhatnánk percekig a már tanult függvényeket, eljárásokat, tulajdonságokat – is ponttal kapcsolódik az adathoz, csakhogy az egyik függvény, a másik eljárás, míg a hagyományos rekordnak ilyen részei nincsenek. Épp emiatt a mai 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.

A C# nyelvben a rekord megfelelője a `struct`, de a hagyományoshoz képest sokkal többet tud, majdnem olyan, mint a `class`. Alapvető különbség közöttük, hogy ami `struct`-ból jön létre, az érték típusú, ami `class`-ból, az referencia típusú. Ez a különbség eddig a függvények paramétereinél jelentett eltérést: a paraméterek belső adatait referencia típusú adatok esetén lehet módosítani, az érték típusú adatoknak – így a `struct`-ként létrejövőknek is a másolatát használja a függvény. Valószínűleg ennél hamarabb okoz problémát, hogy egy érték típusú adatnak hogyan lesz része egy referencia típusú, bármi olyan adattípus, aminek a definiálásához kell a `new`. Mivel a struct használata egyszerűbb, először ezt próbáljuk ki, de az objektum orientált C# nyelvben az igazi megoldás a class használata, ezért ezt is megnézzük.

Rekord a C# nyelvben: struct

A kódolást segíti a struct snippet: struct és kétszer leütve a TAB billentyűt, már csak a nevét és a tartalmát kell megadni.

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

   A `struct`-tal adattípust hozunk létre, de a `Main`()-en belül – általában függvényeken belül – a már létező típusokat használunk, ezért az első megoldást a `Program` osztályon belül, a `Main`() mellé (elé) írjuk.

```
 1.  using System;
 2.  namespace tanulo
 3.  {
 4.    class Program
 5.    {
 6.      struct Tanulo
 7.      {
 8.        public string nev;
 9.        public char nem;
10.        public int kor;
11.      }
12.
13.      static void Main()
14.      {
15.        Tanulo ali;
16.        ali.nev = "Kis Aladár";
17.        ali.kor = 16;
18.        ali.nem = 'f';
19.        Console.WriteLine("{0} {1} éves.", ali.nev, ali.kor);
20.      }
21.    }
22.  }
```

   A `struct` adattagjai elé ki kell írni a `public` jelzőt, másként a `Main`()-en belül nem fogjuk tudni használni, a struct adattagjai alapértelmezetten rejtettek (`private`-ok) az osztályon belüli, „szomszédos” függvények számára. Azért ezen érdemes elgondolkodni… Ha tömb vagy `List`<> elemeinél tejesen természetes, hogy elérhetők, akkor egy `struct` adattagjainál miért kell ezt külön jelezni? A válaszra később visszatérünk, egyelőre jegyezzük meg, hogy mindig ki kell írnunk, hogy `public`. Közben figyeljük meg azt is, hogy nem kell kiírni a `static` jelzőt.

   A `Tanulo` – színéből láthatóan – a `Program` osztályhoz hasonló, majdnem `class`. Ez azt mutatja, hogy a `struct`-ot a `class` mellett, de a `namespace`-en belül is el lehet készíteni, sőt… inkább ott a helye. Az viszont elég zavaró, hogy egy értékadás három tulajdonsággal négy sor. Ha több tulajdonsága van a struktúrának és több adatot szeretnénk eltárolni, akkor elég sok kódsort kell írni. Ezen egy alkalmas függvénnyel lehetne segíteni.

2. Írjuk a `Tanulo` típus definícióját a `Main`()-en kívül és írjunk egy `Letrehoz`() függvényt, amivel később egy sorban megoldható az értékadás!

   A függvényünk visszaadott értéke `Tanulo` típusú lesz, paraméterei pedig a `Tanulo` változói.

```
1.  using System;
2.  namespace tanulo
3.  {
4.    struct Tanulo
5.    {
6.      public string nev;
7.      public char nem;
8.      public int kor;
9.    }
10.   class Program
11.   {
12.     static Tanulo Letrehoz(string neve, int kora, char neme )
13.     {
14.       Tanulo t;
15.       t.nev = neve;
16.       t.kor = kora;
17.       t.nem = neme;
18.       return t;
19.     }
20.
21.     static void Main()
22.     {
23.       Tanulo ali = Letrehoz("Kis Aladár", 16, 'f');
24.       Tanulo bea = Letrehoz("Nagy Bea", ali.kor, 'l'); /*egyidősek*/
25.     }
26.   }
27. }
```

Azzal, hogy a `Tanulo` és a `Program` egy szintre került, érthetőbbé válik, hogy miért nem kell a `static` és miért kell a `public` jelző:

- A `static` a `Program` osztályba azért kell, mert a `Program` osztályt az operációsrendszer használja és a benne lévő adatokat, függvényeket csak egy közös példányban használhatja. A `Tanulo`-t a `Program` használja, a `Program` osztályban hozunk létre `Tanulo` típusú változókat. Többet is, különböző adatokkal. Érdemes kísérletezni, a kor legyen `static`

`static` jelző a struktúrában:

```
1.  struct Tanulo
2.    {
3.      public string nev;
4.      public char nem;
5.      public static int kor;
6.    }
```

A `static` hatására `Tanulo`-k közös adata a `kor`, mindenki azonos korú.

`static` jelző hatása felhasználáskor

```
16.  static Tanulo Letrehoz(string
     neve, int kora, char neme )
17.     {
18.       Tanulo t;
19.       t.nev = neve;
20.       Tanulo.kor = kora;
21.       t.nem = neme;
22.       return t;
23.     }
24.
```

A tapasztaltak rámutatnak arra, hogy a `kor.ToString()`, illetve az `int.Parse()` hogyan keletkezett. Hogyan lehet az egész típusú adatnak szöveggé alakító függvénye, az egész típusnak értelemző függvénye.

- A `public` jelző szerepe is érthetőbbé válik, ha belegondolunk abba, hogy a `Program` osztály osztályon belül globális változóinak, függvényeinek és eljárásainak láthatónak kellene-e

lennie a `Tanulo` számára. Látható, hogy nem magától értetődő, hogy a névtéren belül levő egységek mindenüket megosszák egymással.

A `Letrehoz`() függvénnyel kezelhetőbb a `Tanulo` struktúra. Ennyi éppen elég ahhoz, hogy azokat a feladatokat megoldjuk, amelyekben a kapott adatokat nem kell módosítani, csak elemezni, további adatokat meghatározni belőlük.

A kötelezőn innen és túl

A tantervi követelmények elvileg körülbelül eddig tartanak. A megismert algoritmizálási és C# nyelvi eszközök majdnem elegendőek a következő fejezet feladatainak megoldásához, megvan hozzá az alapszókincs. Azonban napjaink menő programozási technikája az objektumorientált programozás, amihez el kell jutni az objektumok készítéséig. Már nincs sok hátra és aki eléri a célt, annak a feladatok megoldásához a jelenleginél jobb, használhatóbb eszközei lesznek. Aki pedig csak továbbmegy egy picit, annak sem vész kárba az ideje, mert a feladatok többsége az eddig tanultak gyakorlása.

Két dologra van még szükség – a gyakorlaton túl – ahhoz, hogy a feladatok megoldása ne okozzon gondot.

- Több olyan feladatunk volt már eddig is, amelyben az adatokat nem beolvasással, hanem a programba kódolva adtuk meg. Tömb vagy lista esetén ilyenkor kapcsoszárójelek között tudjuk felsorolni az elemeket, de ha egy elem `struct` vagy `class`, azt nem tudjuk kapcsoszárójelek között megadni, ezeket létre kell hozni. Ehhez pedig nem elég egy `Letrehoz`() függvény, konstruktor kell (azaz `new`…). A megírására a 6. feladatban van minta.
- A `struct` érték típusú, ezért listában ilyen elemeket tárolva nem módosítható az adattagok értéke. A megoldást nem kell tudni, mivel a listát sem kell tudni használni. Minden feladat megoldható tömb használatával, ez is. Vagy: csúnya de működő megoldás, ha az egyébként helyes struktúra `struct` szavát átírjuk `class`-ra. Ezzel az adattípusunk referencia típusú lesz.

3. Nézzük, hogyan lehet egy adattagot módosítani: Minden tanuló öregszik. Írjunk eljárást és függvényt a `Program` osztályon belül, ami egy `Tanulo` korát 1 évvel megnöveli!

```
21.  static void Oregit(ref Tanulo t)
22.  {
23.    t.kor += 1;
24.  }
25.  static int Plusz1(Tanulo t)
26.  {
27.    return t.kor + 1;
28.  }
29.  static void Main()
30.  {
31.    Tanulo ali = Letrehoz("Kis Aladár", 16, 'f');
32.    Tanulo bea = Letrehoz("Nagy Bea", ali.kor, 'l'); /*egyidősek*/
33.    Oregit(ref ali);
34.    bea.kor = Plusz1(bea);
35.    Console.Write($"{bea.nev} {bea.kor}, {ali.nev} {ali.kor} éves.");
36.  }
```

Az `Oregit`() eljárásban a paraméterben megadott adat egy részét – adattagját – szeretnénk módosítani, de a `struct`-ok érték típusú adatok, ezért a `ref` módosító nélkül `ali` adatainak

másolatát módosítaná, aminek nincs maradandó hatása. (Futtassuk a programot a `ref` módosítók nélkül is!) A `ref` módosító megoldja a problémát, de eléggé kényelmetlen, hogy fejben kell tartani, az összes felhasználási helyen figyelni kell arra, hogy ne maradjon le a `ref`. Ráadásul a hiánya eléggé sunyi módon okoz problémát, mert a program működni fog, csak az eredmény lesz hibás.

A `Plusz1`() függvénnyel növelt kor jobbnak tűnik ebből a szempontból, de a kódot épp úgy nem nevezhetjük biztonságosnak. Bár a kódban `bea` korát növeljük, de ez a függvény bárkinek a korát be tudja állítani egy másik `Tanulo` koránál eggyel nagyobbra.

Mindkét megoldás jó, de egyik sem az igazi. A kényelmetlenség forrása az, hogy az öregedés a tanuló belső folyamata, nem a `Program` osztályból kellene módosítani a kort, hanem a `Tanulo` struktúrán belülről. Belső folyamatként az öregedés mellékhatása, hogy módosul a kor, a struktúrából kifelé nem kell eredményt adnia, ezért eljárás formájában kellene megvalósítani. Más lenne a helyzet, ha azt kellene egy tanulónak magáról elmondania.

4. Lépjünk túl a hagyományos struktúra – a mondatszerű leírásbeli értelmezése – keretein! Egy hagyományos struktúrában csak adattagok vannak, de az `Oregit`() eljárást a C# `Tanulo struct`-on belül írjuk meg!

```
1.  using System;
2.  namespace tanulo
3.  {
4.    struct Tanulo
5.    {
6.      public string nev;
7.      public char nem;
8.      public int kor;
9.      public void Oregit()
10.     {
11.       kor += 1;
12.     }
13.   }
14.  /*eddig Tanulo, innentől Program*/
15.   class Program
16.   {
17.     static public Tanulo Letrehoz(string neve, int kora, char neme)
18.     {
19.       Tanulo t;
20.       t.nev = neve;
21.       t.kor = kora;
22.       t.nem = neme;
23.       return t;
24.     }
25.
26.     static void Main()
27.     {
28.       Tanulo ali = Letrehoz("Kis Aladár", 16, 'f');
29.       Tanulo bea = Letrehoz("Nagy Bea", ali.kor, 'l');
30.       ali.Oregit();
31.       Console.Write($"{bea.nev} {bea.kor}, {ali.nev} {ali.kor} éves.");
32.     }
33.   }
34. }
```

Most, hogy kiderült, a C# `struct`-ba lehet eljárást írni, írjunk egy függvényt is.

5. Írjunk függvényt, amely megadja egy tanuló első (magyarul általában vezeték) nevét!

   A `Tanulo`-n belüli kiegészítés és a `Main`()-en belüli felhasználás:

```
/*… Tanulo …*/
public string Elsonev()
{
   return nev.Split(' ')[0];
}
/*… felhasználás: Program Main() …*/
string feleseg = ali.Elsonev() + "né";
Console.Write("{0} feleségének neve {1} lesz.", ali.nev, feleseg);
```

6. A `Tanulo` egyre inkább önállósul, vannak saját adattagjai, eljárása, függvénye, de a `Letrehoz`() függvény egy másik egységben, a `Program` osztályban van. Ezt is a `Tanulo`-n belül kellene megírni, hiszen oda tartozik.
   Először egyszerű átmásolással próbáljuk megoldani a feladatot. Az eredmény elgondolkodtató: A `Letrehoz`() használatához kell a `static`.

```
Tanulo ali = Tanulo.Letrehoz("Kis Aladár", 16, 'f');
```

Ha töröljük a `static` jelzőt, akkor egy létező tanuló kellene az új tanuló létrehozásához, de az első tanulót egyedileg kell létrehozni. Lássuk be, ez így – bár működik, – nem normális.

```
Tanulo ali;
ali.nev = "Kis Aladár";
ali.kor = 16;
ali.nem = 'f';
Tanulo bea = ali.Letrehoz("Nagy Bea", ali.kor, 'l');
```

A `Tanulo static public Letrehoz`() függvénye alkalmas `Tanulo` típusú adatok létrehozására. Mondhatnánk, tökéletes megoldás, azonban érdemes belegondolni abba, hogy az `Oregit`() és az `Elsonev`() konkrétan a `Tanulo` struktúrára értelmezhető eljárás illetve függvény, de a létrehozásra mindig van igény. Ezért erre van speciális nyelvi forma, a konstruktor.

```
static public Tanulo Letrehoz(string neve, int kora, char neme )
{
   Tanulo t;
   t.nev = neve;
   t.kor = kora;
   t.nem = neme;
   return t;
}
/*felhasználás*/
Tanulo ali = Tanulo.Letrehoz("Kis Aladár", 16, 'f');
```

Konstruktor ugyanerre:

```
public Tanulo (string neve, int kora, char neme )
{
   nev = neve;
   kor = kora;
   nem = neme;
}
/*felhasználás*/
Tanulo ali = new Tanulo ("Kis Aladár", 16, 'f');
```

Megjelent a new! Ez azt is jelenti, hogy a struct érték típusú adatból ezzel referencia típusú adatszerkezetté vált?

7. Készítsünk a Program osztályon belül eljárást, amely a lányok 'l' nemét 'n'-re változtatja, hiszen a felnőttek nem fiúk vagy lányok, hanem férfiak vagy nők.

```
1.  class Program
2.  {
3.    static void No(Tanulo t)
4.    {
5.      t.nem = (t.nem == 'l' ? 'n' : 'f');
6.    }
7.    static void Main()
8.    {
9.      Tanulo bea = new Tanulo("Nagy Bea", 18, 'l');
10.     No(bea);
11.     Console.WriteLine("{0} neme {1}", bea.nev, bea.nem);
12.   }
```

Ha a programot az fenti számozás szerinti 6. sor előtt megállítjuk, még azt látjuk, hogy t.nem értéke 'n'. Azonban a 11. sorban bea.nem értéke 'l'. Ebből következik, hogy a Tanulo továbbra is érték típusú, a No(Tanulo t) t paraméterébe a bea másolata kerül.

8. Az ember kora érzékeny adat. Évente módosul, de nincs értelme a módosításnak. Oldjuk meg, hogy a létrehozást követően a kor értékét csak az Oregit() eljárással lehessen módosítani, de azért az aktuális értéket meg lehessen jeleníteni!

   További feladat egyéni megoldással, hogy hasonló módon a név és nem módosíthatóságát és megtekinthetőségét is szabályozzuk.

   A Tanulo adattagjai azért láthatók és módosíthatók, mert a struct definiálásakor public jelölőt írtunk elé. Ha a public-ot töröljük, akkor nem tudunk hozzáférni a Program osztályban egyetlen Tanulo korához sem. Azonban ez nem okoz problémát ali és bea létrehozásakor. A konstruktor és az Oregit() eljárás public jelölésű, elérhetők és futtathatók is. Ezeket a Tanulo struct-on belül definiáltuk azaz a Tanulo-béli kor az Oregit() számára globális változó, amit elér és módosítani is tud. Hasonló módon a kor kiírása is megoldható: kell egy publikus függvény, ami a Tanulo struct saját függvénye és a visszatérési értéke a kor.

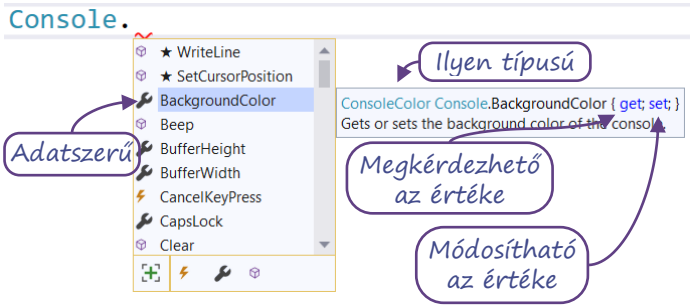
```
1.  using System;
2.  namespace tanulo
3.  {
4.    struct Tanulo
5.    {
6.      public string nev;
7.      public char nem;
8.      int kor;
9.      public Tanulo (string neve, int kora, char neme)
10.     {
11.       nev = neve;
12.       kor = kora;
13.       nem = neme;
14.     }
15.     public void Oregit()
16.     {
17.       kor += 1;
18.     }
```

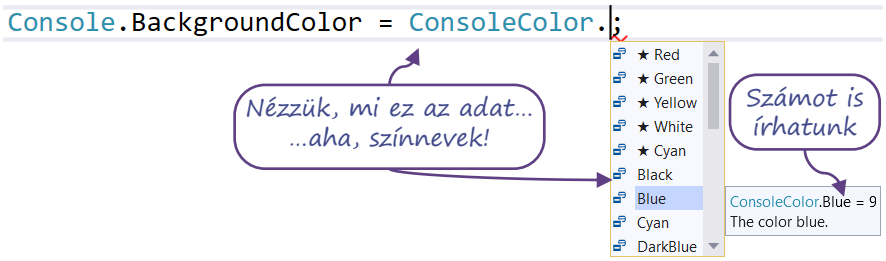
```
19.     public int GetKor()
20.     {
21.       return kor;
22.     }
23.   }
24.
25.   class Program
26.   {
27.     static void Main()
28.     {
29.       Tanulo bea = new Tanulo("Nagy Bea", 18, 'l');
30.       Console.WriteLine("{0} kora {1}, neme {2}",
                            bea.nev, bea.GetKor(), bea.nem);
31.       bea.kor -= 2; /*hibás, a kor nem létezik*/
32.       Console.WriteLine(bea.kor); /*hibás, a kor nem létezik*/
33.     }
34.   }
35. }
```

Az osztály (class) C#-ban és az igazi objektumok

Az előző feladatban elrejtettük a `Tanulo` kor tulajdonságát. Épp így elrejthetnénk a nemét is, hogy csak olvasható legyen és valószínűleg a név tulajdonság sem egy pillanatonként változó adat. Amikor szoftvereket használunk, rendszeresen módosítjuk különböző objektumok adatait – egy nyíl méretét, egy nyomógomb helyét, egy szövegdoboz szegélyének színét – de ezt minden esetben csak meghatározott módon tehetjük meg, nem férünk hozzá közvetlenül a változókhoz. A programozók ezt úgy érik el, hogy az objektumok adatait priváttá teszik és eljárásokkal, függvényekkel biztosítják a hozzáférést. A **setter** értékadó eljárás teszi lehetővé a módosítást, a **getter** értéket visszaadó függvény pedig a megjelenítést. Ha egy adathoz nincs megírva a setter, akkor nem módosítható, ha nincs getter, akkor nem látható. Gyakorlatban sokszor előfordul, hogy egy setter több változó értékét módosítja, egy getter több változóból ad eredményt. Például, ha egy képpont színét állítjuk be vagy nézzük meg egyben az RGB három értékével.

Ha C#-ban egy osztályon belül létrehozunk egy változót, az belső, privát, az osztályra nézve globális változó lesz. Ilyeneket hoztunk létre a `Program` osztályon belül. Már néztük programozási tanulmányaink kezdetén is, de bármikor tanulmányozhatjuk, ha Visual Studioban a `Console` vagy más osztály, illetve változó után írt pontnál megállunk és megnézzük, mit ajánl fel folytatásként. Egy osztálynak kívülről láthatóak a **`tulajdonság`**ai, `eljárás`ai, `függvény`ei és még néhány dolog, de nem láthatók a `változó`i.





Azt is láthatjuk, hogy a tulajdonságnak van get-je és set-je. Az a kérdés, hogy hogyan lesz egy getterből és egy setterből egyetlen tulajdonság. Már csak ezt kell megtudjuk ahhoz, hogy minden meglegyen egy `class` megírásához.

A C#-ban a tulajdonság egy rövidített kódolási megoldás, amiben egyetlen névvel adjuk meg a getter függvényt és a setter eljárást.

`struct`-ban getter és setter:

```
1.  private int kor;
2.
3.
4.  public int GetKor()
5.  {
6.    return kor;
7.  }
8.  public void SetKor(int value)
9.  {
10.   kor = value;
11. }
```

`class`-ban tulajdonság

```
1.  private int kor;
2.  public int Kor
3.  {
4.    get
5.    {
6.      return kor;
7.    }
8.    set
9.    {
10.     kor = value;
11.   }
12. }
```

Miért is jó a tulajdonság, ha hosszabb a beírása? Azért, mert így leírni csak akkor kell, ha a 6. vagy 10. sorban nem ezt, hanem valami speciális dolgot szeretnénk írni. Ha az itt kifejtett 12-soros kódra gondolunk, akkor ezt C#-ban ezzel a rövidítéssel adjuk meg:

```
1.  public int Kor {get; set;}
```

Ha csak olvasható tulajdonságot szeretnénk, akkor `struct` esetén nem írjuk meg a `SetKor()` eljárást, illetve `class` esetén ki kell írni a `private` változót és a `get` részt.

A tulajdonság (property) kódolását nem csak C# nyelvi elem segíti, hanem a Visual Studio is.

A get-set tulajdonság snippettel: prop és kétszer leütni a TAB billentyűt.
Majd ki kell tölteni a típus és a Név helyeket.

Látható, hogy a `class`-ban a privát változónév kisbetűvel kezdődik, a tulajdonság nagy kezdőbetűs, de egyébként megegyeznek. Amikor megadunk egy tulajdonságot, akkor – még a programozó elöl is elrejtve – létrejön az alapértelmezett azonos, de kiskezdőbetűs privát változó is, ezért a Visual Studio figyelmeztet, ha nem nagybetűs a tulajdonság neve.

9. Írjunk `Diak` néven osztályt (`class`-t) a `Tanulo` struktúra mintájára!

   A megoldáshoz három részt kell módosítani. Egyrészt, a `struct` helyett `class` kell.

Az osztály létrehozása snippettel: class és kétszer leütni a TAB billentyűt, utána már csak a nevét és a tartalmát kell megadni.

Másrészt, cseréljük le az adattagokat tulajdonságokra. Harmadrészt írjuk újra vagy módosítsuk a konstruktort, függvényeket és eljárásokat, merthogy a változóink eltűntek, helyettük tulajdonságok lesznek.

A konstruktor snippetje: ctor és kétszer leütni a TAB billentyűt. A paramétereit felhasználva adunk kezdőértéket a tulajdonságoknak.

```
1.  using System;
2.  namespace diak
3.  {
4.    class Diak
5.    {
6.      public string Nev { get; set; }
7.      private int kor;
8.      public int Kor { get { return kor; } }
9.      public char Nem { get; set; }
10.
11.     public Diak(string neve, int kora, char neme)
12.     {
13.       Nev = neve;
14.       kor = kora;
15.       Nem = neme;
16.     }
17.     public void Oregit()
18.     {
19.       kor += 1;
20.     }
21.     public string ElsoNev()
22.     {
23.       return Nev.Split(' ')[0];
24.     }
25.   }
26.
27.   class Program
28.   {
29.     static void No(Diak d)
30.     {
31.       d.Nem = (d.Nem == 'l' ? 'n' : 'f');
32.     }
33.
```

```
34.     static void Main()
35.     {
36.       Diak bea = new Diak("Nagy Bea", 18, 'l');
37.       Console.WriteLine("{0} kora {1}, neme {2}",
                            bea.Nev, bea.Kor, bea.Nem);
38.       bea.Oregit();
39.       No(bea);
40.       Console.WriteLine("{0} új kora {1}, neme {2}",
                            bea.ElsoNev(),bea.Kor,bea.Nem);
41.       //bea.Kor -= 2; /*uncomment-elve hibás, mert nincs set*/
42.     }
43.   }
44. }
```

A kódolás és futtatás során figyeljük meg:

- Hogyan módosul a tulajdonság létrehozása és a konstruktorban az értékadás, ha csak olvasható a tulajdonság.
- Módosul `bea` neme, azaz a `Diak class` referencia típusú, paraméterként az eredeti adatot kapják meg az eljárások, függvények (nem a másolatot).
- Amikor a főprogramban felhasználjuk a `Diak` osztályt, a Visual Studio megjeleníti a tulajdonságokat, függvényeket… bár leírás nincs hozzá, de láthatóan olyasmit készítettünk, amilyet eddig csak használtunk.

10. Készítsünk egy második konstruktort is, amelyben egyetlen `string`ből – amelyben az adatok a megfelelő sorrendben szóközökkel vannak elválasztva – hozzuk létre a `Diak` objektumot!

```
1.  public Diak(string adatsor)
2.  {
3.    string[] adatok = adatsor.Split(' ');
4.    Nev = adatok[0];
5.    kor = int.Parse(adatok[1]);
6.    Nem = adatok[2][0];
7.  }
```

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 a struktúrák és az osztályok adattípusok, az objektumok adatok, az objektumok neve változónév. Az azonos típusú objektumokat a már korábban tanult módon tudjuk adatsorozatban (tömbben, listában) tárolni.

Feladatok

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` `struct`-ban megadjuk a tulajdonságokat, az egyszerűség kedvéért mindegyik tulajdonság publikus lesz.

1. Adjuk meg a csoport adatait a programban.

   A csoport adatait tömbbe és listába is felvesszük, hogy később mindkettőt tudjuk használni (a feladat megoldásához elegendő az egyik). Az adatsorozatok elemeit csak konstruktorral tudjuk létrehozni, ezt is meg kell írnunk a `Tanulo` adattípuson belül.

```
1.  using System;
2.  using System.Collections.
    Generic;
3.  namespace osztaly
4.  {
5.    struct Tanulo //vagy class
6.    {
7.      public string Nev;
      //; vagy { get; set; }
8.      public char Nem;
9.      public int Kor;
10.     public Tanulo(
    string nev, char nem, int kor)
11.     {
12.       Nev = nev;
13.       Nem = nem;
14.       Kor = kor;
15.     }
16.   };
```

```
17.
18.  class Program
19.  {
20.  static Tanulo[] csoport =
           new Tanulo[3] {
21.    new Tanulo("Tamara",'l', 15),
22.    new Tanulo("Tibi", 'f', 14),
23.    new Tanulo("Tomi", 'f', 14)
24.  };
25.
26.  static List<Tanulo> csoplista=
           new List<Tanulo>() {
27.    new Tanulo("Laci", 'f', 16),
28.    new Tanulo("Lili", 'l', 17),
29.    new Tanulo("Lujzi", 'l', 16)
30.  };
31.
```

   Amennyiben a `struct` helyett `class`-t írunk az 5. sorban, akkor a 7–10. sorban a változók utáni pontosvessző helyett a { `get`; `set`; } ugyanolyan néven tulajdonság lesz

Az adattípus definíció és adatsorozatok megadását követően a feladatok megoldását a `Main`() függvényben részben ezekre az adatokra vonatkozóan oldjuk meg.

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”.

Mintaadatok tömbhöz:

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

Mintaadatok listához:

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

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, ha a függvényeket a főprogram után írjuk meg, így előbb láthatjuk a felhasználást, láthatjuk, hogy mit keressünk később.

A feladatok típusalgoritmusok alkalmazását igényli, a megoldás a korábbi, egyszerű adatokkal foglalkozó megoldásoktól annyiban tér el, hogy nem elég egy változónevet beírni, mindig meg kell adni – ponttal elválasztva – a megfelelő adattagot is.

A főprogramban minden megoldást tesztelhetünk. Az alábbi minta programban mindegyik programrészt legalább egyszer kipróbálunk, de ez tetszőlegesen bővíthető, vagy – ha csak tömbbel, vagy csak listával készítjük el a megoldást, akkor – szűkebb is lehet.

A `Main`() függvény – a kód egyben megoldási terv is.

```
32.  static void Main()
33.  {
34.    Tanulo[] osztaly = new Tanulo[4];
35.    List<Tanulo> osztalylista = new List<Tanulo>();
36.
37.    InputT(osztaly, 4);
38.    InputL(osztalylista, 4);
39.    Console.WriteLine("Átlag tömbből-> csoport: " + AtlagT(csoport, 3));
40.    Console.WriteLine(" beírt osztály: " + AtlagT(osztaly, 4));
41.    Console.WriteLine("Átlag listából -> csoport: "+ AtlagL(csoplista));
42.    Console.WriteLine(" beírt osztály: " + AtlagL(osztalylista));
43.    TobbsegT(csoport, 3);
44.    TobbsegT(osztaly, 4);
45.    TobbsegL(csoplista);
46.    TobbsegL(osztalylista);
47.    Console.Write("Név: ");
48.    string nev = Console.ReadLine();
49.    Console.WriteLine($"{nev} {KeresT(nev, osztaly, 4)} éves.");
50.    Console.WriteLine("{0} {1} éves.", nev, KeresL(nev, csoplista));
51.    LepT(osztaly, 4);
52.    LepL(osztalylista);
53.    KiirT(osztaly, 4);
54.    KiirL(osztalylista);
55.  }
```

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:

```
56.     /*2-3. beolvasás és kiírás*/
57.     static void InputT(Tanulo[] T, int db)
58.     {
59.       for (int i = 0; i < db; i++)
60.       {
61.         string[] be = Console.ReadLine().Split(' ');
62.         T[i] = new Tanulo(be[0], be[1][0], int.Parse(be[2]));
63.       }
64.     }
65.
```

```
66.     static void KiirT(Tanulo[] T, int db)
67.     {
68.       for (int i = 0; i < db; i++)
69.         Console.WriteLine("{0} {1} {2}", T[i].Nev, T[i].Nem, T[i].Kor);
70.     }
71.
```

A 2. és 3. feladat megoldása `List<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.

```
72.     static void InputL(List<Tanulo> L, int db)
73.     {
74.       for (int i = 0; i < db; i++)
75.       {
76.         string[] be = Console.ReadLine().Split(' ');
77.         L.Add(new Tanulo(be[0], be[1][0], int.Parse(be[2])));
78.       }
79.     }
80.
```

Az adatok beolvasásának kódja láthatóan hosszabb egy lista esetén, mert a listába beillesztés előtt fel kell bontani az adatsort, létre kell hozni az objektumokat. A kiírás viszont már nem jelent új kihívást. Használhatjuk a bejáró ciklust.

```
81.     static void KiirL(List<Tanulo> L)
82.     {
83.       foreach (Tanulo t in L)
84.         Console.WriteLine("{0} {1} {2}", t.Nev, t.Nem, t.Kor);
85.     }
```

4. Írjuk ki a csoport átlagéletkorát!

```
86.     /*4. átlagéletkor*/
87.     static double AtlagT(Tanulo[] T, int db)
88.     {
89.       double a = 0;
90.       for (int i = 0; i < db; i++)
91.         a += T[i].Kor;
92.       return a / db;
93.     }
94.
```

```
95.     static double AtlagL(List<Tanulo> L)
96.     {
97.       double a = 0;
98.       foreach (Tanulo t in L)
99.         a += t.Kor;
100.      return a / L.Count;
101.    }
```

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.

```
102.     /*5. fiú vs. lány többség*/
103.     static void TobbsegT(Tanulo[] T, int db)
104.     {
105.       int merleg = 0;
106.       for (int i = 0; i < db; i++)
107.         if (T[i].Nem == 'f') merleg += 1;
108.
109.         else merleg -= 1;
110.       if (merleg > 0) Console.WriteLine("Fiúk vannak többen.");
111.       else if (merleg < 0) Console.WriteLine("Lányok vannak többen.");
112.       else Console.WriteLine("Ugyanannyi fiú van mint lány.");
113.     }
114.
```

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

```
115.     static void TobbsegL(List<Tanulo> L)
116.     {
117.       int merleg = 0;
118.       foreach (Tanulo t in L)
119.         merleg += (t.Nem == 'f') ? 1 : -1;
120.       //if, else másképp
121.       if (merleg > 0) Console.WriteLine("Fiúk vannak többen.");
122.       else if (merleg < 0) Console.WriteLine("Lányok vannak többen.");
123.       else Console.WriteLine("Ugyanannyi fiú van mint lány.");
124.     }
```

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:

```
125.     /*6. név keresés -> hány éves*/
126.     static int KeresT(string nev, Tanulo[] T, int db)
127.     {
128.       int ez = 0;
129.       while (ez < db && T[ez].Nev != nev)
130.         ez++;
131.       return ez < db ? T[ez].Kor : -1;
132.     }
133.
```

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

```
134.     static int KeresL(string nev, List<Tanulo> L)
135.     {
136.       foreach (Tanulo t in L)
137.         if (t.Nev == nev)
138.           return t.Kor;
139.       return -1;
140.     }
```

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. Ezt egy érték típusú `struct` nem tudja, a `Tanulo` `class` kell.

```
141.     /*7. évfolyamváltás (kor léptetése)*/
142.     static void LepT(Tanulo[] T, int db)
143.     {
144.       for (int i = 0; i < db; i++)
145.         T[i].Kor++;
146.     }
147.
148.     static void LepL(List<Tanulo> L)
149.     {
150.       foreach (Tanulo t in L)
151.         t.Kor++;
152.     }
153.   }
154. }
```

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. A táblázatunk egy-egy sora egy-egy zenész jelenlétéről szól, négy zenész öt munkanapjáról lesz adatunk.

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!

Megoldások

Az eddig öt megoldást néztünk meg, amelyekben mátrixban – táblázatban – tároltuk a jelenléti adatokat és külön tömbben tároltuk a zenészek nevét. 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 öt 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 megoldásnál: a vonós jelenlétét ötelemű tömbben tároljuk.
- A `struct` (érték típusú) adatot választjuk (valójában nem lesz osztály). A módosíthatóságot úgy biztosítjuk, hogy a Program osztályon belül statikus, globális adatként tároljuk az adatokat. Ezért nem kell semmit paraméterben megjeleníteni, de korlátlan lesz a hozzáférés az adatokhoz.
- Adatok beolvasása és konstruktorok készítése nem kerülhető meg. A konstruktorunkban megadjuk az összes adattag értékét. 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.

```
1.  using System;
2.  using System.Collections.Generic;
3.  namespace zeneszek
4.  {
5.    struct Vonos
6.    {
7.      public string Nev;/*class esetén { get; set; }*/
8.      public int[] Iv;/*class esetén { get; set; }*/
9.      public Vonos(string nev, int[] iv) /*kostruktor*/
10.     {
11.       Nev = nev;
12.       Iv = new int[iv.Length];
13.       for (int i = 0; i < iv.Length; i++)
14.         Iv[i] = iv[i];
15.     }
16.     public int Jelen()
17.     {
18.       int db = 0;
19.       foreach (int j in Iv)
20.         db += j;
21.       return db;
22.     }
23.   }
```

A zenekar adatait beírjuk a programkódba. `List<Vonos>` elemeit kapcsos-zárójelek között soroljuk fel. Az elemeket vesszővel választjuk el egymástól, minden elemnek { } 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 { } 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!

```
24.   class Program
25.   {
26.     const int napok = 5;
27.     static List<Vonos> Zenekar = new List<Vonos>(){
28.       new Vonos("Heg Edu",       new int[napok]{ 1, 1, 1, 1, 1 }),
29.       new Vonos("Vio Lina",      new int[napok]{ 1, 1, 1, 1, 0 }),
30.       new Vonos("Brá Csaba",     new int[napok]{ 1, 1, 0, 0, 0 }),
31.       new Vonos("Csel Lotti",    new int[napok]{ 0, 1, 1, 1, 1 })
32.       };
33.
```

Ellenőrzésképpen írjuk ki az adatokat képernyőre:

```
34.     static void Kiir(List<Vonos> zenekar)
35.     {
36.       foreach (Vonos v in zenekar)
37.       {
38.         Console.Write(v.Nev + ":");
39.         for (int i = 0; i < v.Iv.Length; i++)
40.           Console.Write(" {0}", v.Iv[i]);
41.         Console.WriteLine();
42.       }
43.     }
44.
```

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.

```
45.     static int EbedDb()
46.     {
47.       int db = 0;
48.       foreach (Vonos vonos in Zenekar)
49.         db += vonos.Jelen();/*itt nem feltételes az összegzés*/
50.       return db;
51.     }
52.
```

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.

```
53.     static Vonos LegtobbJelen()
54.     {
55.       Vonos maxv = Zenekar[0];  /*az objektum egy változó*/
56.       foreach (Vonos v in Zenekar)
57.         if (v.Jelen() > maxv.Jelen()) /*a sajátfüggvény értéke alapján*/
58.       maxv = v; /*a v minden adata maxv-be*/
59.       return maxv;
60.     }
61.
```

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.

```
62.     static bool Mindig()
63.     {
64.       int ez = 0;
65.       while (ez < Zenekar.Count &&
                  Zenekar[ez].Jelen() < Zenekar[ez].Iv.Length)
66.         ez++;
67.       return ez < Zenekar.Count;
68.     }
69.
```

Azért a hatékonyságon itt is érdemes elgondolkodni … A while-ciklus magjában csak egy értéknövelés van, de a feltételében függvény szerepel. Elvileg egy-egy ciklusléptésben az `i`-edik tagra a jelenléti ív minden adatát megvizsgálja. Ha nem a `Jelen`() függvényt használnánk a függvényt, akkor tagonként csak az első 0 értékig kellene vizsgálni a jelenlétet, ami a `Vonos` egy másik sajátfüggvénye lehetne.

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.

```
70.     static bool Mindenki()
71.     {
72.       bool mindenki = false;
73.       for (int ekkor = 0; ekkor < napok && !mindenki; ekkor++)
74.       {
75.         int jelen;
76.         for (jelen = 0; jelen < Zenekar.Count &&
                            Zenekar[jelen].Iv[ekkor] == 1; jelen++)
77.           ; /*üres*/   /*^^ minek a micsodája? ^^ */
78.         if (jelen == Zenekar.Count)
79.           mindenki = true;
80.       }
81.       return mindenki;
82.     }
83.
```

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:

```
84.     static void Main()
85.     {
86.       Kiir(Zenekar); /*olvasásra paraméterhasználat tesztelése*/
87.       Console.WriteLine("Összesen {0} adagot kell fizetni pénteken.",
                            EbedDb());
88.       Console.WriteLine("{0} volt a legtöbbet jelen.",
                            LegtobbJelen().Nev);
89.       Console.WriteLine(Mindig() ?
                            "Volt, aki egyszer sem hiányzott." :
                            "Senki sem volt mindig jelen.");
90.       Console.WriteLine(Mindenki() ?
                            "Volt, amikor mindenki jelen volt." :
                            "Mindig hiányzott valaki");
91.     }}}
```

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. Sajnos azt sem tudjuk, hogy hány neve van a vendégnek és egy vendég esetén nem biztos, hogy egész hétre érdemes adatokat írogatni, főleg, ha csak a hét elején jön. Ez azt jelenti, hogy a jelenlét megadásakor, ha ötnél kevesebb számadatot kap a program, akkor a maradék napokra a nulla értékekkel kell kiegészíteni 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# nyelvben az adatok konverziójának van hibafigyelő – TryParse() – formája

is. A TryParse() függvény akkor ad `true` értéket, ha sikerült konvertálni az adatot. Ha az eredmény `false`, akkor a kimeneti paraméterben nem történt változás.

Az vendégeket beolvasó eljárás

- egy sorban, szóközökkel elválasztott adatokat vár;
- az első adatok alkotják a vendég nevét;
- a neveket követően legfeljebb 5 db 1-es vagy 0-s érték jelzi a napi jelenlétet;
- ötnél kevesebb szám esetén a hiányzó értékek 0-ként jelennek meg;
- amelyik jelenléti érték nem 0, azt 1 értékként tárolja;
- több, mint 5 napra szóló részvétel esetén a felesleget eldobja;
- addig kér be új adatsort, amíg a kapott adatsorból meghatározható egy név.

Figyeljük meg, hogyan érvényesülnek a feltételek az alábbi megoldásban!

```
1.  statix void Vendeg()
2.  {
3.    bool hiba = false;
4.    while (!hiba)
5.    {
6.      Console.WriteLine("Add meg a vendég nevét és heti jelenlét!");
7.      string[] adatok = Console.ReadLine().Split(' ');
                          /*Az üres szöveg 1 elemű*/
8.      int idx = 0;
9.      string nev = "";
10.     int[] iv = new int[5]; /*0 értékekkel feltöltődik*/
11.     int szam;
12.     while (idx < adatok.Length && !int.TryParse(adatok[idx], out szam))
13.     {
14.       nev += adatok[idx] + " ";
15.       idx++;
16.     }
17.     if (nev.Length <= 1)
18.       hiba = true; /*nincs megadva név*/
19.     else
20.     {
21.       nev = nev.Substring(0, nev.Length - 1);
                    /*utolsó szóköz eltávolítása*/
22.       int napdb = 0;
23.       while (idx < adatok.Length && napdb < 5 &&
                                    int.TryParse(adatok[idx], out szam))
24.       {
25.         iv[napdb] = (szam == 0 ? 0 : 1);
                    /*tudjuk, hogy egész szám, ha nem 0, akkor 1*/
26.         idx++;
27.         napdb++;
28.       }
29.     }
30.     if (!hiba)
31.       Zenekar.Add(new Vonos(nev, iv));
32.   }
33. }
```

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 `List<>` típuson tudjuk használni a bejárós ciklust, ami lényegesen egyszerűbb a számlálós ciklusnál. Szerencsére a `List<>` adatszerkezet létrehozása nem feladatunk, azt nagyon profi programozók sok éves tapasztalattal és tesztelés során írták meg.

A tömb és a `List<>` adatstruktúrák között nem csak adatszerkezeti különbség van, hanem a felhasználásuk célja is más. A tömb előre elkészített adathelyeit bármikor megadhatjuk, módosíthatjuk, könnyen bővíthetjük többdimenziós mátrixra; a `List<>`-ba új elemet (jellemzően a végére vagy az elejére) hozzáfűzéssel adhatunk, nem lehet lyuk az adatok között.

Az adatsorozatok tárolására nem csak a mátrix és a `List<>` 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 gyűjtemények (collection).

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

- `ArrayList<>`: 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. Megtudhatjuk, hogy egy adott elem benne van-e. A közbülső adatok felsorolhatók, de nem módosíthatók.
- `LinkedList`<> két-irányú-láncolt lista, amibe lehet csomópontot (elemet) beszúrni és kivenni is, az elemek helye ettől nem változik, csak a lánc, az összekötés módosul.
- `Stack`<>: általános nevén verem (LIFO), aminek a végéhez tudunk adatot hozzáadni és ugyaninnen lehet törölni az adatot. Megtudhatjuk, hogy egy adott elem benne van-e. A közbülső adatok felsorolhatók, de nem módosíthatók
- `Dictionary`<>: 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.
- `HashSet`<>: általános nevén halmaz, amibe ugyanazt az adatot kétszer betéve is csak egy példányt fog tartalmazni.

A gyűjtemények működését osztálydefiníciók tartalmazzák, ami a System.Collections.Generic névtérben van. Pontosan úgy, ahogy a `List`<> is.

Feladatok

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

5. 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?

6. 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?

7. Készítsünk magyar–angol szótárt! Használjunk a szavak tárolására tömböt, `List`<>-et, esetleg próbáljuk ki a `Dictionary`<>-t!

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 76, 80, 93
argumentum 20, 21
break 44
cast 58
Comment 37
continue 43
definíció 21
érték típus 23, 68, 77, 80, 83
escape karakter 4
Format Document 37
függvénydefiníció 20, 22
getter 84
globális változó 22, 23, 29
három operandusú operátor 32
index 42, 45, 47, 55
inicializál 7
Jagged Array 72
konstruktor 19, 82
lineáris keresés 47
LINQ 38, 39, 40, 45, 51, 53
mellékhatás 21, 22, 44
null 68
objektum 47, 77, 94, 96
osztály 77, 93
out 23
paraméter 19, 20, 21, 22, 23
példányosítás 77
ref 23, 80, 81
referencia típus 23, 68, 77, 80, 83, 87
region 37
rekord 1, 76, 77
setter 84
specifikáció 58, 59, 60
struktúra 65, 76
szintaktika 19
Uncomment 37
visszaadott érték 20, 47, 48, 50
visszatérési érték 19, 20, 21, 22, 25